Learning and Memorization via Predictive Coding

Tommaso Salvatori

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2022

Abstract

Neural networks trained with backpropagation achieved impressive results in the last decade. However, training such models requires sequential backward updates and non-local computations, making it challenging to parallelize at scale, implement in novel hardware, and is unlike how learning works in the brain. Neuroscience-inspired learning algorithms, such as predictive coding, have the potential to overcome these limitations and advance beyond current deep learning technologies. This potential, however, has only recently gained the attention of the community. As a consequence, the properties of these algorithms are still underexplored. In this thesis, I aim at filling this gap by exploring three interesting properties of predictive coding: First, there exists a variation of predictive coding that is equivalent to backpropagation in supervised learning, second, predictive coding is able to perform powerful associative memories, and third, it is able to train neural networks with graphs of any topology. The first result implies that predictive coding networks can be as accurate as standard ones when used to perform supervised learning tasks. The last two, that they are able to perform tasks with a robustness and flexibility that is lacking in standard deep learning models. I then conclude by discussing future directions of research, such as neural architecture search, novel hardware implementations, and implications in neuroscience. All in all, the results presented in this thesis are coherent with recent trends in the literature, which show that neuroscience-inspired learning methods may have interesting machine learning properties, and that they should be considered as as a valid alternative to backpropagation.

Contents

1 Introduction **2**

1.1 Background . . . 3

1.2 Contributions . . . 5

1.2.1 Works included in the thesis . . . 7

2 Preliminaries and Related Work **9**

2.1 Predictive coding networks . . . 9

2.1.1 Neural networks and backpropagation . . . 12

2.1.2 Key differences with standard neural networks . . . 13

2.2 Algorithms derived from predictive coding . . . 14

2.2.1 Zero divergence from backpropagation . . . 16

2.2.2 Parallel predictive coding . . . 17

2.2.3 Experimental evaluation . . . 18

2.3 Related work on predictive coding . . . 20

2.3.1 Predictive coding as variational inference . . . 20

2.3.2 Generation . . . 21

2.3.3 PredNet . . . 21

2.3.4 Predictive coding for control and robotics . . . 22

3 Exact Backpropagation via Predictive Coding **24**

3.1 Convolutional neural networks . . . 27

3.1.1 Convolutional neural networks with backpropagation . . . 27

3.1.2 Predictive coding convolutional networks . . . 28

3.1.3 Predictive coding convolutional networks trained with Z-IL . . . 28

3.2 Recurrent neural networks . . . 29

3.2.1 Recurrent neural networks with backpropagation . . . 29

3.2.2 Predictive coding recurrent neural networks . . . 30

3.2.3 Predictive coding recurrent networks trained with Z-IL . . . 31

3.3 Computational graphs . . . 32

3.3.1 Backpropagation on computational graphs . . . 33
3.3.2 Predictive coding on computational graphs . . . 34
3.4 The problem of skip connections . . . 36
3.5 Levelled computational graphs . . . 37
3.6 Z-IL for levelled computational graphs . . . 39
3.7 Empirical evaluation . . . 41
3.8 Proofs of theorems . . . 42
3.8.1 Proof of Theorem 2 . . . 43
3.8.2 Proof of Theorem 3 . . . 48
3.8.3 Proof of Theorem 5 . . . 49
3.9 Why is this equivalence important? . . . 52

4 Associative Memory Capabilities **53**
4.1 Background on associative memories . . . 55
4.2 Basic model: A fully connected and linear graph . . . 56
4.2.1 Experiments . . . 58
4.3 Hierarchical model . . . 59
4.3.1 Analysis of different levels of noise . . . 63
4.3.2 Plots of images with extreme levels of noise . . . 64
4.3.3 Analysis of the retrieval function . . . 64
4.3.4 Analysis of the retrieval time . . . 65
4.4 Retrieval from partial data points . . . 65
4.5 More training data and/or deeper networks . . . 67
4.5.1 Deep generative predictive coding networks . . . 68
4.6 Comparison with deep associative neural networks . . . 69
4.7 Hetero-associative memory experiments . . . 70
4.8 Full-page reconstructions of ImageNet . . . 71
4.9 Implications to neuroscience . . . 72
4.10 Summary and discussion . . . 72

5 Learning on Arbitrary Graph Topologies **82**
5.1 PC graphs . . . 84
5.2 Experiments on fully connected graphs . . . 87
5.2.1 Associative memory . . . 90
5.3 Extension to different graph topologies . . . 91
5.4 Conditioning on labels . . . 95
5.5 Assembly of neurons . . . 95

5.6 Methodology and further experiments 96
5.6.1 Architectures and hyperparameters 96
5.6.2 Feedforward vs. recursive networks 98
5.6.3 Importance of weight decay 99
5.7 Restricted Boltzmann machines 100
5.8 Summary and discussion 101

6 Conclusion **102**
6.1 Summary 102
6.2 Outlook 103
6.2.1 Scaling up predictive coding 104
6.2.2 New hardware? 105
6.2.3 Implications to neuroscience 106

Bibliography **108**

List of Figures

2.1 Differences between BP and PC 15
2.2 Differences between PC, Z-IL, and PPC 16
2.3 PredNet 22

3.1 Historical sketch of the results unifying PC and BP 27
3.2 An example of a *convolution*. 28
3.3 An example of a *many-to-one* RNN. 31
3.4 Identity node 32
3.5 Difference between computational graphs and PC graphs 33
3.6 A PCN as a computational graph 35
3.7 Levelled graphs 38

4.1 Generative PCN with 2 layers 55
4.2 Memorization of linear PCNs 57
4.3 Experiments using recurrent linear networks 60
4.4 Associative memories on corrupted data 61

4.5 Quantitative experiments on corrupted data 62
4.6 Qualitative experiments on corrupted data 63
4.7 Different levels of noise 74
4.8 Associative memories on highly corrupted data 75
4.9 Analysis of the retrieval function 76
4.16 Quantitative results on incomplete data 76
4.17 Quantitative results on incomplete data 77
4.18 Comparison with modern Hopfield networks 77
4.19 Comparison with deep associative neural networks 77
4.20 Experiments on Hetero-associative memories 78
4.21 Quantitative results on hetero-associative memories 79
4.22 Reconstructions on ImageNet-1 80
4.23 Reconstructions on ImageNet-2 81

5.1 Difference in topology between an artificial neural network and a PC graph 83
5.2 Structure of a PC graph 85
5.3 Results on a PC graph 86
5.4 Memory experiments using PC graphs 90
5.5 PC graphs built by masking a part of the weights of a fully connected one . 91
5.6 Query by initialization on three different PC graph architectures 92
5.7 Query by conditioning on three different PC graph architectures 93
5.8 Generation experiments 94
5.9 Generation experiments on hierarchical models 99
5.10 Reconstructions experiments on hierarchical models 99
5.11 Reconstructed images given the label and by conditioning 100
5.12 Reconstructed and denoised images using RBMs. 100

List of Algorithms

1 Learning one training pair $(\bar{s}^{\text{in}}, \bar{s}^{\text{out}})$ with PC 12
2 Learning one training pair $(\bar{s}_{\text{in}}, \bar{s}_{\text{out}})$ with Z-IL 17
3 Learning $(\bar{s}_{in}, \bar{s}_{in})$ with PPC 18
4 Generating a levelled DAG G' from G 39
5 Z-IL for computational graphs. 40
6 Learning to generate $\bar{s}$ with PC 56
7 Retrieving $\bar{s}$ given a non corrupted fraction $\bar{s}'$ 67
8 Learning the external stimulus $\bar{s}$ 85

List of Tables

2.1 Accuracies of PC, PPC, and BP 19

3.1 Divergence between one update of weights of BP and Z-IL 26
3.2 Divergence between BP and improved Z-IL 41
3.3 Running times of a weight update of BP, PC, and Z-IL 42

4.1 Time to perform a retrieval from incomplete images on PCN 66

5.1 Test accuracy of different models on MNIST, FashionMNIST, and SVHN. . 88
5.2 Accuracy experiments on PC graphs 94

List of Equations

2.1 . . . 10
2.2 . . . 10
2.3 . . . 10
2.4 . . . 11
2.5 . . . 12
2.6 . . . 12
2.7 . . . 13
2.9 . . . 13
3.1 . . . 27
3.2 . . . 28
3.3 . . . 28
3.4 . . . 29
3.5 . . . 30
3.6 . . . 30
3.7 . . . 30
3.8 . . . 30
3.9 . . . 30
3.10 . . . 31
3.11 . . . 31
3.12 . . . 32
3.13 . . . 33
3.14 . . . 34
3.15 . . . 34
3.16 . . . 34
3.17 . . . 34
3.18 . . . 35
3.19 . . . 35
3.20 . . . 36
3.21 . . . 36

3.22 . . . 36
3.23 . . . 36
3.24 . . . 37
3.25 . . . 38
3.26 . . . 38
3.27 . . . 40
3.28 . . . 40
3.29 . . . 43
3.30 . . . 43
3.31 . . . 44
3.35 . . . 45
3.41 . . . 46
3.42 . . . 47
3.43 . . . 47
3.44 . . . 47
3.45 . . . 48
3.47 . . . 50
3.48 . . . 50
3.50 . . . 51
4.1 . . . 57
5.1 . . . 84
5.2 . . . 85

List of Theorems

1 Theorem (Equivalence result on fully connected networks) . . . 17

2 Theorem (Equivalence for CNNs) . . . 29
3 Theorem (Equivalence for on RNNs) . . . 31
4 Theorem (Levelled computational graphs) . . . 39
5 Theorem (Equivalence for computational graphs) . . . 40
6 Theorem . . . 43
7 Theorem . . . 48

8 Theorem 49

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Tommaso Salvatori
May 2022

Chapter 1

Introduction

Learning and memorization are key properties of both human and machine intelligence. In fact, the ability to memorize a training set, learn its patterns, and generalize to an unseen test set, is the main goal of any machine learning algorithm. Hence, the generalization performance of trained models are often evaluated in terms of the *generalization gap* — the difference between accuracy on the training set and accuracy on the test set. At first sight, memorization and generalization seem to be opposite concepts, as having enough parameters to memorize a training set has often been considered an obstacle for generalizing to a test set. The idea that over-parameterized functions would generalize poorly to unseen data has initially been introduced in classical theories of learning, such as VC-dimension and Rademacher complexity [1, 2]. At the time, the common belief was that the more it is memorized by a model, the less it has been learned. Overparametrized deep learning models, on the other hand, empirically challenged this assumption, achieving excellent results on multiple tasks despite being able to perfectly fit the training set [3, 4, 5]. This is reasonable if seen from the perspective of human intelligence: memorization is important to learn and understand the world, and allows to use memories to generate predictions [6]. Therefore, recent works on neural networks look at *memory* and *learning* as being two faces of the same coin, instead of being opposites of each other [7, 8, 9, 10]. It has in fact been shown that deep networks can be reduced to kernel methods when highly overparameterized [11] or trained using the whole dataset as a single batch [7]. This could hint to the fact that, in classification tasks, trained networks do nothing more than scoring a test point accordingly to how *similar* it is to stored training points in a specific embedding space [12]. Furthermore, memorization is not only important when paired with learning: computational memory models able to store and retrieve datapoints, called *associative memories*, have been developed for different decades now, and still form an active topic of research [13, 14, 15, 16]. Surprisingly, these models have been shown to be strongly connected with modern deep learning architectures, such as transformers and MLP-Mixers [17, 15, 18].

Predictive coding is an influential neuroscientific theory that describes information processing among different levels of cortical hierarchies [19, 20, 21]. In short, this theory hypothesizes that the brain is constantly predicting sensory stimuli by generating an internal model of the external environment, and that learning happens by correcting errors in these predictions. While it was initially developed to simulate brain behavior [20], in the last years this framework became increasingly important in machine intelligence, as it allows to train deep learning models, also called *predictive coding networks*, with appealing properties [22, 23, 24, 25]. Particularly, it is an energy-based model that is able to approximate [22, 26] and perfectly replicate [24] the parameter update of backpropagation under some conditions, as well as obtaining good performance in multiple machine learning tasks and benchmarks, such as classification [25], generation [27, 28], associative memory [29], and prediction of the next video-frame [30, 31]. These results, together with other appealing properties of energy-based learning methods, such as the locality of operations and reduced external control [22] have raised the attention of the machine learning community. These properties can lead to efficient implementations of deep learning algorithms on new hardware, such as analog chips [32], hence addressing the need of having huge computational resources to train modern neural networks.

1.1 Background

Over the last decade, deep learning has significantly evolved, leading to a massive revolution in the fields of machine learning and artificial intelligence. Examples of the many breakthroughs include *image recognition* [33, 34, 35, 18], *speech recognition* [36], *game playing* [37, 38, 39], and natural language understanding [17, 40, 41]. Despite this initial success, there are still multiple challenges ahead of us: (i) state-of-the-art (SOTA) models now count billions of parameters, making them extremely expensive and energy consuming to be trained. This strongly increases inequalities, as these models are now beyond the reach of small business and research groups, and is putting the carbon footprint of deep learning under serious scrutiny: training a large language model has the same impact on the environment as dozens round trips from New York to Beijing [42]; (ii) we are still far from understanding how artificial neural networks really work [3], and how artificial general intelligence can be reached. While predicting the steps that will allow us to solve the aforementioned problems is ambitious, a promising direction focuses on understanding, and reverse engineering, how the human brain learns models of the world. There is a common belief that this is the correct direction to pursue because of two main reasons that I now discuss. The first one looks at past progress in the field, and it is more speculative, while

the second one is more concrete, and looks at properties of brain networks and human intelligence that would significantly improve modern deep learning.

Simply put, the vast majority of the past progress in deep learning are neuroscience-inspired. Already the very basic concept of artificial neural networks (ANNs) is based on its biological counterpart: networks of neurons, pairwise connected by synaptic weights, with a hierarchical structure [43]. Because of this biological structure, it was natural to improve them by taking inspirations from progress in neuroscience. Examples of such improvements are convolutional layers [44], whose connectivity pattern between neurons resembles the organization of the animal visual cortex, skip [34, 45] and recurrent [46] connections, present in many areas of the neocortex, and dropout [47].

Despite the aforementioned improvements, that have brought ANNs closer to biological networks, the latter ones still present appealing properties, which are yet to be emulated in a proper algorithm. The implementation of these appealing properties represents a concrete research direction worth exploring to address current problems in *artificial intelligence* (AI) and machine learning, such as energy and parameter efficiency, and its inadequacy in reaching human-level intelligence [48]. In fact, on the one hand, deep learning requires a huge amount of energy: training a language model can consume up to 700 KWh [49], and takes several days. On the other hand, the human brain consumes about the same amount of energy as a few light bulbs, thanks to its biological "hardware".

Informally, the predictive coding theory claims that there are two families of neurons that define an internal model of the world: the first, in charge of making predictions, the second, in charge of computing errors in these predictions. This allows to define hierarchical networks with a neural implementation, where synapses are updated via inhibitory and excitatory connections in an Hebbian fashion [50, 20]. The first concrete application of predictive coding dates back to 1982, where it was used to describe neural properties of the retina [51]. However, it was only ten years later that Mumford proposed predictive coding as a general theory of information processing in the neocortex [19]. A computational model of Mumford's theory was then built a couple of years later, in the breakthrough paper by Rao and Ballard [20], where every single operation, from changes in neural activities to synaptic updates, was governed by the same energy function, that encoded the prediction error of every neuron of the network. This was then extended in a series of papers that provide a better theoretical understanding of the model [52, 53, 54].

During the deep learning era, started in 2012, when convolutional networks were used to reach SOTA on ImageNet [33], research on neuroscience-inspired learning methods by the machine learning community was not very active. In fact, deep neural networks trained with backpropagation [55] were, and are, achieving impressive results. However,

in the last couple of years, the question whether backpropagation was enough to reach human-level intelligence, has brought attention back to neuroscience-inspired learning. In fact, backpropagation is unable to describe credit assignment in the brain, the rule that governs synaptic updates in the neocortex [56, 57, 58]. From the neuroscience perspective, predictive coding is a promising direction to pursue, as it is able to perform both top-down and bottom-up computations, useful to perform generation and classification tasks simultaneusly [20, 22, 59], is energy efficient [60], and allows to perform every computation in parallel. This last property allows it to be potentially faster than backpropagation in training deep neural networks by parallelizing operations of different layers [61].

In a work that proposes predictive coding for supervised learning, it has been shown that it is able to perform well on simple datasets, such as MNIST, and approximate the parameter update of backpropagation on multi layer perceptrons, even if only under some strict conditions [22]. This first result led to a direction of research that aims to study the similarities between predictive coding and backpropagation. More generally, exploring the similarities between neuroscience-inspired algorithms and backpropagation has in fact been an active area of recent research [56], as the desire to find a novel learning method that aims to replace backpropagation in some task cannot be blind of the impressive performance reached by this algorithm on almost any pattern recognition tasks. Recently, different interesting results have been extending the approximation results of Whittington and Bogacz [22]. In 2020, a generalization of their result has shown that predictive coding is able to approximate backpropagation on general computational graphs [26]. In parallel, an exactness result was also proposed, as a variation of predictive coding has been proved to perform exact backpropagation on multi-layer perceptrons [24], with no restrictive condition. I will briefly review this variation in Section 2.2.1.

1.2 Contributions

The contributions of this thesis consist of the study of the generalization and memorization capabilities of predictive coding networks, as well as other interesting properties from the machine learning point of view. More in detail, I will study three main properties of this family of networks: I will present (i) an exactness result that strongly connects predictive coding and modern machine learning; (ii) an analysis of their associative memory capabilities; and (iii) a study of how they can be used to perform machine learning tasks using networks with any topology.

Exact backpropagation I will show that a variation of predictive coding is equivalent to backpropagation. Understanding this connection is important, as no neuroscience-inspired training method can yet replicate the performance of BP [56]. To fill this gap, I show that a variation of predictive coding is able to perfectly replicate the update of the parameters of backpropagation on any computational graph, and hence, neural networks with arbitrary structure. This is the first biologically plausible algorithm that is completely equivalent to backpropagation in the way of updating parameters on complex models, and it is thus a bridge for the interdisciplinary research of neuroscience and deep learning. This result implies that predictive coding networks have the potential to generalize to unseen data as well as modern neural networks. To show this results, I merge the aforementioned approximation results [26, 22] with the exactness result proven for multi-layer perceptrons [24], showing that this variation of PC is able to exactly replicate the parameter update of backpropagation on any computational graph. This is covered in Chapter 3 of this thesis.

Associative memories I will then test the associative memory capabilities of predictive coding networks, showing that, when used as bottom-up generative networks, they are able to memorize and retrieve complex data points, such as high quality pictures, from corrupted or incomplete keys. I compare the results against multiple associative memory models present in the literature, and conclude that predictive coding models perform better than both neural networks trained with BP and one-shot memory models, such as Hopfield networks, when asked to memorize and retrieve data points. The resulting model is not only interesting for the machine learning community, but could also have implications in neuroscience, as it provides a computational model of a recent framework that aims to explain memory storage and retrieval in the brain following a predictive coding rule [6]. These results are covered in Chapter 4 of this thesis.

Training on graphs with any topology I conclude by addressing a limitation of artificial neural networks, where information only flows in one direction (feedforward pass), and the error in the opposite one (backward pass). This implies that learning with backpropagation reduces to learning functions $\mathbb{R}^d \to \mathbb{R}^k$, allowing a neural network to perform only one specific task, such as classification, image generation, or image completion. To perform a different task, a brand new and different training process has to be performed. In Chapter 5, I will use the energy-based formulation of predictive coding to go beyond the multilayer structure, and train networks with any topology. The resulting class of networks that can be trained this way is that of parameterized models that converge to a solution via an energy minimization process, instead of that of functions, of which it is a superset. This

allows the design of new models that are able to perform multiple and different generation tasks regardless of the training procedure, and to train models that resembles the entangled networks present in the brain. In fact, biological networks are not fully hierarchical, but are divided in multiple brain regions sparsely connected among each other [62].

1.2.1 Works included in the thesis

During my journey as a DPhil student, I have worked on several projects. Among them, the following are included in this thesis:

- *Associative Memories via Predictive Coding*, of which I am first author, and is a joint work with Song, Y., Lukasiewicz, T., Xu, Z., Bogacz, R., Hong,Y., and Simon, F. It corresponds to the results reported in the chapter with the same name. This work is published in NeurIPS 2021.

- *Reverse Differentiation via Predictive Coding* of which I am first author, and is a joint work with Song, Y., Lukasiewicz, T., Xu, Z. and Bogacz, R. It corresponds to the results reported in the chapter with the same name. This work is published in AAAI 2022.

- *Predictive Coding Can Do Exact Backpropagation on CNNs and RNNs* of which I am first author, and is a joint work with Song, Y., Lukasiewicz, T., Xu, Z. and Bogacz, R. This work is available on ArXiv.

- *Learning on Arbitrary Graph Topologies via Predictive Coding*, of which I am first author, and is a joint work with Pinchetti, L., Song, Y., , Millidge, B., Lukasiewicz, T., and Bogacz, R. It corresponds to all the results reported in Chapter 5. It is available on ArXiv, and is currently under conference submission.

- *Predictive Coding: Towards a Future of Deep Learning beyond Backpropagation?* of which I am joint first author with Millidge, B. (alphabetical order), and is a joint work with Song, Y., Lukasiewicz, T., and Bogacz, R. The sections I have written have served as an inspiration for part of the literature review and introduction. This work is published on IJCAI survey track.

I have also co-authored the following papers, that are excluded from the thesis:

- *BoxE: A Box Embedding Model for Knowledge Base Completion*, a joint work with Abboud, R., Ceylan, İ. İ., and Lukasiewicz, T., This work is published in NeurIPS 2020.

- *Prospective Configuration: A New Perspective on Learning Beyond Backpropagation*, at the moment submitted for journal publication, joint work with Song, Y., Millidge, B., Lukasiewicz, T., Xu, Z. and Bogacz, R.

- *Parallel Predictive Coding: A Biologically Inspired Learning Algorithm*, at the time of writing submitted for conference publication, joint work with Song, Y., Millidge, B., Lukasiewicz, T., and Bogacz, R.

- *Universal Hopfield Networks: A General Framework for Single-Shot Associative Memory Models*, at the moment submitted for conference publication and available on ArXiv, joint work with Song, Y., Millidge, B., Lukasiewicz, T., and Bogacz, R.

- *Semantic Hopfield Networks*, of which I am first author and is at the moment submitted for conference publication, joint work with Song, Y., Millidge, B., Lukasiewicz, T., and Bogacz, R.

- *Backprop at the weak-feedback limit of energy based models: unifying predictive coding, equilibrium propagation, and contrastive hebbian learning*, at the moment submitted for conference publication, joint work with Song, Y., Millidge, B., Lukasiewicz, T., and Bogacz, R.

Chapter 2

Preliminaries and Related Work

In this chapter, we introduce *predictive coding networks* (PCNs), and compare them against standard deep learning models trained with backpropagation (BP). Particularly, we first start by defining multi-layer PC networks, and compare their learning dynamics against the equivalent model in standard machine learning, the multi-layer perceptron (MLP). Then, we present an overview of the current state of predictive coding in the machine learning community. Particularly, we first introduce different variations of this algorithm, and show how they perform in classification tasks. Then, we review related approaches in the literature. To make the dimension of variables explicit, we denote vectors with a bar; e.g., $\overline{x} = (x_1, x_2, \ldots, x_n)$, and matrices in bold and capital, e.g., $\mathbf{W} = (\overline{\mathbf{w_1}}, \ldots, \overline{\mathbf{w_n}})$. In what follows, we will derive the equations that describe the dynamics of predictive coding networks from a general energy function.

2.1 Predictive coding networks

Classic deep learning has achieved remarkable results by using objective functions defined only on specific layers. Every parameter is then updated to minimize this function using reverse differentiation. In the brain, there is no such thing as an objective function, and every synaptic connection is updated locally, according to the information of its pre and post-synaptic neurons. Our neurons constantly make predictions about the world; most of the times, the prediction is in line with the stimuli we get from external sensorial inputs. However, when our prediction differs from the reality, our brain immediately corrects the error (difference between real and predicted) by updating the neural activities and the strength of some synaptic connections. Let us assume we see a can of beer on our kitchen table that we believe to be full. Before grabbing it, our brain has already predicted its weight, mostly given by the amount of beer we expect to be inside it. If the can turns out to be empty, we will notice without the need of trying to drink it: we will find the

can to be much lighter than expected, and our brain will immediately use this error in the prediction to update its belief on the amount of beer in the can. The learning algorithm we now describe follows this logic at the neuronal level, as both learning and inference happen by constantly minimizing the difference between predicted and actual values in every neuron of the network. Particularly, we now introduce PCNs, an hierarchical network of neurons with a layer-wise structure, similar to that of MLPs. The original learning rule has been introduced by Rao and Ballard in 1999 [20]. Following the standard literature in computational neuroscience, we invert the usual notation used to index the layers in machine learning: we index the output layer by 0 and the input layer by L. To make the notation lighter, we avoid considering bias values. Furthermore, we denote by θ^l the weight matrix relative to layer l. The framework we now describe is that of supervised learning, where a labelled point $(\bar{s}_{in}, \bar{s}_{out})$ is presented to the network during training [22]. Given a specific input, the value of the output neurons is computed via an energy minimization process that will be now explained. Before doing that, we define the three fundamental quantities of PCNs: *values*, *predictions*, and *errors*.

Every neuron is composed of three computational units that change over time steps t. The first one is the *value node* $x^l_{i,t}$, where the indices i and l refer to the i-th neuron of the l-th layer, and it is a trainable parameter of the model. The second computational unit is the *prediction node* $\mu^l_{i,t}$, that is a function of the value nodes of the following layer, and is computed as follows:

$$\mu^l_{i,t} = \textstyle\sum_{j=1}^{n^{l+1}} \theta^{l+1}_{i,j} f(x^{l+1}_{j,t}), \tag{2.1}$$

where f is a non-linear function. The third computational unit is the prediction error $\varepsilon^l_{i,t}$, given by the difference between its value node and its prediction node, i.e.,

$$\varepsilon^l_{i,t} = x^l_{i,t} - \mu^l_{i,t}. \tag{2.2}$$

This local definition of error, that exists not only in the output neurons, but in every neuron of the network, is what allows PC to learn using only local information. The goal of PCNs is to minimize an energy function defined as the sum of the squared errors of all the neurons of the network:

$$E_t = \frac{1}{2}\textstyle\sum_{i,l}(\varepsilon^l_{i,t})^2. \tag{2.3}$$

Given a labelled point, two phases are needed to perform a single weight update. The first one, called *inference*, is used both in the training phase, to compute the best configuration

of value nodes to perform a weight update, and in the prediction phase, to compute an output when provided a specific input. During inference, the weights are frozen, and only the internal value nodes are updated to minimize the energy of Eq. (2.3). The second phase happens after the inference has converged, and hence the 'best' neural activities are computed. Here, the opposite happens: all the value nodes are now frozen, and a single weight update is performed to further minimize the same energy function. We will now provide a more formal description of the two phases.

Inference: This is the basic and most important operation that can be performed using PC, and has the goal of updating the value nodes (i.e., neural activities) to minimize the energy of the network. To update the value nodes when a data point $\bar{s}_{in}$ is presented to the network, the value nodes $x^L_{i,t}$ of the input layer are fixed to the entries of $\bar{s}_{in}$ for every time step t. The rest of the value nodes, randomly initialized at $t = 0$, are updated to minimize the energy function of Eq. (2.3) via gradient descent (GD), a first-order method that updates the parameters of the network at every iteration. Particularly, the equation for the update of the value nodes is the following:

$$\Delta x^l_{i,t} = -\gamma \cdot \frac{\partial E_t}{\partial x^l_{i,t}} = \begin{cases} 0 & \text{if } l = L \\ \gamma \cdot (-\varepsilon^l_{i,t} + f'(x^l_{i,t}) \sum_{k=1}^{n^{l-1}} \varepsilon^{l-1}_{k,t} \theta^l_{k,i}) & \text{if } 0 < l < L \\ \gamma \cdot (-\varepsilon^l_{i,t}) & \text{if } l = 0, \end{cases} \tag{2.4}$$

where γ is the learning rate for the value nodes. This method is used to make predictions, i.e., to compute the output when providing a specific input. Let us assume we are computing inference on a data point $\bar{s}_{in}$, and that the total energy has converged at time T. The prediction of the network are then the value nodes $x^0_{i,T}$ of the output neurons at convergence.

Weight Update: To assure that the predictions are meaningful, and indicative of the presented training point, the model has first to be trained. To update the value nodes when a labelled point $(\bar{s}_{in}, \bar{s}_{out})$ is presented to the network, the value nodes of the input neurons are fixed to be equal to the entries of $\bar{s}_{in}$ for every t, and the same happens for the output neurons with the label $\bar{s}_{out}$. At this point, inference is run until convergence to minimize the error of the internal neurons. This allows the error on the output neurons to be well spread in the internal neurons of the network. After the inference has converged, every synaptic weight $\theta^l_{i,t}$ is updated to further minimize the error of its post-synaptic neuron, according to the same energy function. Particularly, the update rule of a single weight is the following:

$$\Delta\theta^{l+1}_{i,j} = -\alpha \cdot \partial E_t / \partial \theta^{l+1}_{i,j} = \alpha \cdot \varepsilon^l_{i,t} f(x^{l+1}_{j,t}), \tag{2.5}$$

Algorithm 1 Learning one training pair $(\overline{s}^{\text{in}}, \overline{s}^{\text{out}})$ with PC

Require: $\overline{x}_0^{l_{\max}}$ is fixed to $\overline{s}^{\text{in}}$; $\overline{x}_0^0$ is fixed to $\overline{s}^{\text{out}}$.
1: **for** $t = 0$ to T (included) **do**
2: **for** each neuron i in each level l **do**
3: Update $x_{i,t}^l$ to minimize E_t via Eq. (2.4)
4: **if** $t = T$ **then**
5: Update each $\theta_{i,j}^{l+1}$ to minimize E_t via Eq. (2.5)

Where α is the learning rate for the synaptic update. The alternation of these two phases, inference and weight update, defines the learning algorithm used to train PCNs [20, 22]. Interestingly, while every computation is local, both update rules minimize the same energy function, globally defined on the whole network. The pseudocode of PC can be found in Alg. 1.

2.1.1 Neural networks and backpropagation

We now introduce the basic architecture for deep learning, the MLP. We present it in a slightly unconventional way, as this will help the reader to better draw a connection between MLPs and PCNs with the same structure. Particularly, we index the layers following the PC formulation introduced above, with the input layer being layer L and the output layer being layer 0.

An MLP is a function $\mathbb{R}^d \longrightarrow \mathbb{R}^{d_{out}}$, parameterized by L weight matrices, denoted $\boldsymbol{W}^1, \ldots, \boldsymbol{W}^L$. This function is a composition of linear layers, that define a linear mapping, each followed by a non-linearity f. Again, L is the number of layers in the network, and d and d_{out} represent the input and output dimension of the model, respectively. We denote by y_i^l the input of the i-th node in the l-th layer, which is conceptually similar to the predictions $\mu_{i,t}^l$ of PCNs. Hence, for the non-linear case we have

$$y_i^l = \sum_{j=1}^{n^{l+1}} \boldsymbol{W}_{i,j}^{l+1} f(y_j^{l+1}), \tag{2.6}$$

Prediction: During this phase, a data point $\overline{s}_{\text{in}}$ is presented as an input to the network, and the value of every internal neuron y_i^l is computed iteratively according to Eq. (2.6). The prediction of the network is then the value of the output layer $\overline{y}^0$. This process is also known as *forward pass*.

Training: Again, using a neural network to make a prediction is only informative if its parameters have been previously trained. In supervised learning, the network is presented with a training point $\bar{s}_{in}$ and its label $\bar{s}_{out}$, and the weight parameters of the network are updated to minimize a loss function F defined on the output layer. This function depends on the prediction of the network when presented with $\bar{s}_{in}$, and on its supervised signal $\bar{s}_{out}$. As a loss function, we consider the *mean squared error* (MSE):

$$F = \frac{1}{2}\sum_{i=1}^{d_{out}}(s_{out,i} - y_i^0)^2, \tag{2.7}$$

where y_i^0 depends on an input $\bar{s}_{in}$. This loss function is then minimized via GD. Particularly, a single weight update performed is computed as follows:

$$\Delta \boldsymbol{W}_{i,j}^{l+1} = -\alpha \cdot \partial F / \partial \boldsymbol{W}_{i,j}^{l+1} = \alpha \cdot \delta_i^l f(y_j^{l+1}), \tag{2.8}$$

where α is the learning rate, and $\delta_i^l = \partial F / \partial y_i^l$ is the *error term*, given as follows:

$$\delta_i^l = \begin{cases} s_i^{\text{out}} - y_i^0 & \text{if } l = 0\,; \\ f'(y_i^l)\sum_{k=1}^{n^{l-1}} \delta_k^{l-1} \boldsymbol{W}_{k,i}^l & \text{if } l \in \{1, \ldots, l_L{-}1\}\,. \end{cases} \tag{2.9}$$

Note that that both the equation governing the weight update and the definition of the error δ_i^l, resemble the weight update and error $\varepsilon_{i,t}^l$ defined for PCNs.

When the network is too large, the computation of the gradients may become computationally expensive. To solve this problem, an algorithm that efficiently computes the gradients using reverse differentiation, called backpropagation, has been developed [55]. This method has allowed to train large and complex neural networks in a reasonable amount of time, as it allows to quickly compute the derivatives needed for the update.

2.1.2 Key differences with standard neural networks

As the above sections have shown, there are multiple similarities between PCNs and standard networks trained with backpropagation. However, there are also several differences, such as the ability of incorporating cycles inside the neural architecture and the possibility of parallelizing every operation, thanks to the use of only local information. The study of these differences is worth exploring, as progress in this direction would allow to use PCNs to perform tasks not accessible to standard neural networks. We now briefly discuss them.

Cyclic graphs: While so far we have only introduced MLPs and their PC counterpart, neural networks are not limited to sequences of linear layers and activation functions. The most used models are in fact composed by different kinds of layers, such as convolutional layers [33] or transformers [17]. More generally, it is possible to use backpropagation to train any network that has the form of a directed acyclic graph. If a cycle would be present inside the neural structure of a network, an infinite loop that makes learning not possible would be created during the first forward pass. More generally, the computational graph of any function $F : \mathbb{R}^d \rightarrow \mathbb{R}^k$ is acyclic. Hence, at the end of the training process, neural networks are able to predict the label given a specific input, but fail to invert the process: inferring the input from the label requires a new and independent training process. PC allows to further relax this assumption, as it can be used to train networks with cycles inside their neural structure. This allows the network to pass information to adjacent neurons, but also to learn from them. This is not possible in multilayer networks, which learn via a forward and backward pass and not via energy minimization.

Local information: Every operation performed in this model is strictly local and can hence run in parallel. Learning by local computations is a key property of the brain, which allows the design of cyclic graphs, but also to parallelize every operation. This is not possible in multilayer networks trained with BP, where layer-wise operations have to be performed in sequence. In fact, in standard deep learning it is common practice to define a loss function on a specific layer, and use reverse differentiation to update all the parameters of the networks, even if they are not strictly connected to that particular layer. The conceptual difference between the two approaches is represented in Fig. 2.1.

2.2 Algorithms derived from predictive coding

In the previous section, we have introduced PC, and saw how its training regime is separated in two different phases: inference and weight update. In this section, we will introduce two variations of PC that do not follow this regime, but perform weight updates *during* the inference process. The first algorithm we introduce, is called *zero divergence inference learning* (Z-IL) [24], and has the peculiar property of being completely equivalent to BP in the way it updates the parameters of an MLP. The second, is called *parallel predictive coding* (PPC) [61], and is a fully automatic variation of PC: at every time step t, both the value nodes and weight parameters are updated in parallel, removing the need of a control signal that triggers the weight update when inference has converged (or has run for a fixed number of steps T). A drawing with the differences of the three algorithms can be found in

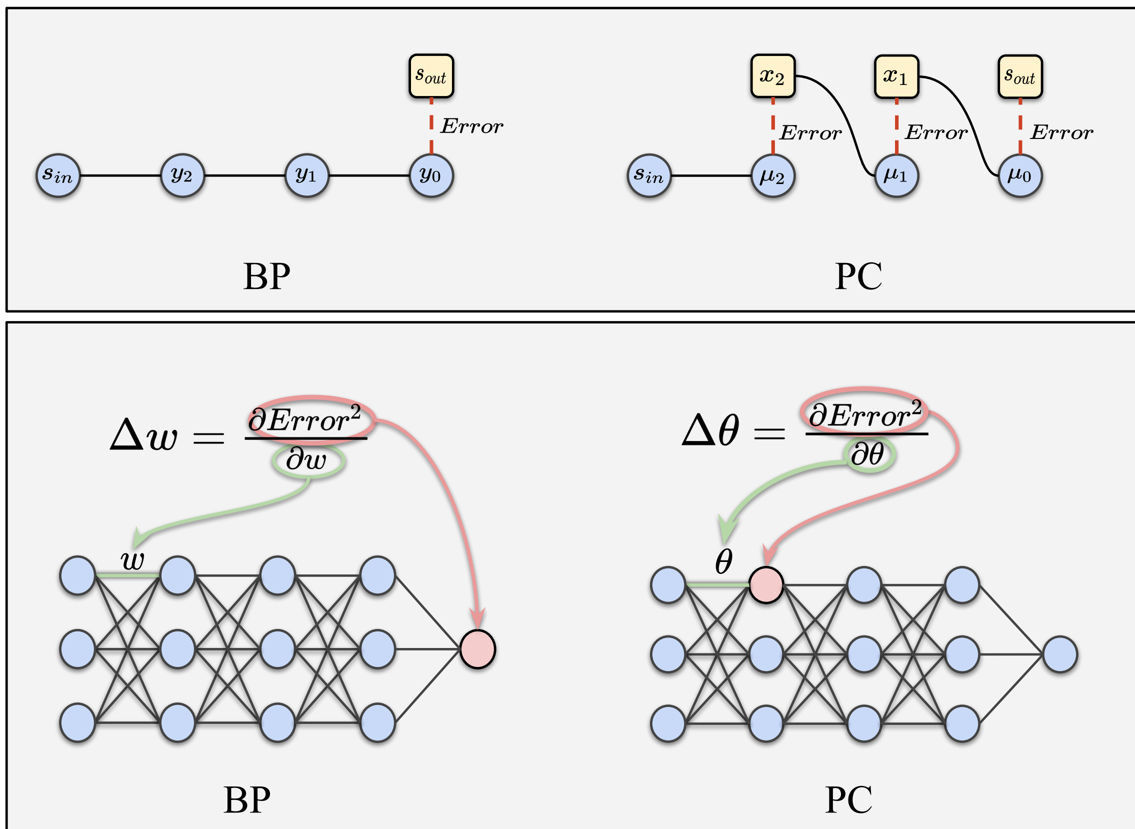


Figure 2.1: Top: representation of a three layer MLP. Standard neural networks trained with BP correct an error defined on the output neuron, which is the difference between prediction and supervised signal. PCNs define the error in every neuron of the network. Bottom: difference between BP and PC in terms of locality: BP updates a synapse w to minimize the output error, even if it is not directly connected to it. PCNs, on the other hand, update a synapse θ to correct the error of its postsynaptic neuron.

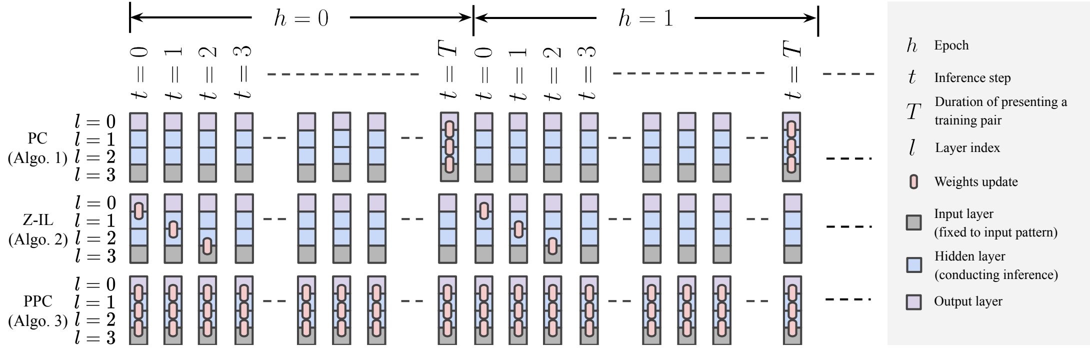


Figure 2.2: Comparison of the temporal training dynamics of PC, Z-IL, and PPC. We assume that we train the networks on a training pair $(\bar{s}_{in}, \bar{s}_{out})$ for a period of time T, before it is changed to another pair. The squares represent nodes in one layer, and pink rounded rectangles indicate when the connection weights are modified: PC (1st row) conducts inference until it converges ($t = T$) and updates the weights, while Z-IL (2nd row) only updates the weights at specific inference moments depending on which layer the weights belong to, and PPC updates the weights at every inference moment t.

Fig. 2.2. As it may lead to confusion, throughout this thesis we will mostly consider PCNs trained with PC, unless otherwise specified.

2.2.1 Zero divergence from backpropagation

As already mentioned in the introduction, it has been shown that PCNs trained to perform supervised learning tasks are able to approximate BP on any model under some specific conditions [22, 26]. We now review a variation of PC that is able to perform *exact* BP on MLPs [24].

Let us assume we train a PCN with a multilayer structure on a labelled point $(\bar{s}_{in}, \bar{s}_{out})$. As previously explained, the training process consists of first fixing the value nodes of the input and output layer to the entries of $\bar{s}_{in}$ and $\bar{s}_{out}$, respectively. Then, the internal value nodes are updated via inference, that runs for multiple time steps until convergence, and is followed by an update of all the weight parameters. In Z-IL, these two phases are mixed as follows: At $t = 0$, inference on all the value nodes, and a single weight update only on layer 0, i.e., θ_0 are computed in parallel, according to Eqs. (2.4) and (2.5) respectively. At $t = 1$, we run a single step of inference, and perform a single weight update *only* on layer 1. This process is then iterated L times, during which we only update the weights θ_l at time $t = l$. For a formal update rule, please refer to the pseudocode provided in Alg. 2.

In order to get equivalence with BP, two more conditions have to be added: (i) the learning rate γ of the value nodes has to be equal to 1.0, and (ii) the value nodes need a specific initialization. Specifically, the internal value nodes are initialized to be equal to

Algorithm 2 Learning one training pair $(\bar{s}_{\text{in}}, \bar{s}_{\text{out}})$ with Z-IL

Require: $\bar{x}_0^L$ is fixed to $\bar{s}_{\text{in}}$; $\bar{x}_0^0$ is fixed to $\bar{s}_{\text{out}}$.
Require: $x_{i,0}^l = \mu_{i,0}^l$ for $l \in \{1, \ldots, l_{\max} - 1\}$, and $\gamma = 1$.
1: **for** $t = 0$ to $L - 1$ (included) **do**
2: **for** each neuron i in each level l **do**
3: Update $x_{i,t}^l$ to minimize E_t via Eq. (2.4)
4: **if** $t = l$ **then**
5: Update each $\theta_{i,j}^{l+1}$ to minimize E_t via Eq. (2.5)

their predictions. This is done as follows: first, we set the value nodes of the input layer to be equal to the entries of the training point $\bar{s}_{in}$. Then, starting from the input layer, we set $\bar{x}_0^l = \bar{\mu}_0^l$. This does not apply to the output layer, whose value nodes are fixed to the label $\bar{s}_{out}$. The pseudocode of Z-IL on MLPs can be found in Alg. 2. This peculiar initialization generates a network that has zero error on every internal neuron, and the error on the output layer is equivalent to the MSE loss of a standard neural network. This guarantees that a full weight update performed by Z-IL, is equivalent to the weight update performed by BP, as formally stated in the following theorem:

Theorem 1 (Equivalence result on fully connected networks)**.** *Let M be a fully connected PCN trained with Z-IL, and let M' be its corresponding MLP, initialized as M, and trained with BP. Then, given the same data point $\bar{s}$ to both networks, we have*

$$\Delta\theta_l = \Delta \boldsymbol{W}_l$$

for every layer $l \geq 0$.

In the original work, the authors show that this model is also computationally as efficient as backpropagation. This is a large improvement over standard PC, which is order of magnitudes slower than BP [24].

2.2.2 Parallel predictive coding

PC has always been considered an influential theory in neuroscience. However, it has never been used for practical applications. The main drawback of standard PC, in fact, is the time needed for the inference process to converge. This adds a large computational overhead, that makes PCNs significantly slower than BP. We now introduce PPC, that not only solves this problem, but can also be faster than BP in specific contexts.

Simply put, the main difference between PC and PPC is that the weight updates and inference are performed at the same time step. So, at every time step t, both Equations (2.5) and (2.4) are performed. In a full batch training scenario, this model can be up to L

Algorithm 3 Learning $(\bar{s}_{in}, \bar{s}_{in})$ with PPC

Require: $x^0_{i,t}$ is fixed to s^{in}_i, $x^L_{i,t}$ is fixed to $\bar{s}^{in}_i$.
1: **for** $t = 0$ to $(T-1)$ **do**
2: **for** each neuron i in each level l **do**
3: Update $x^l_{i,t}$ to minimize E_t via Eq. (2.4)
4: Update each $\theta^l_{i,j,t}$ to minimize E_t via Eq. (2.5)

times faster than BP, where L is the number of hidden layers. This is a result of the total error being computed only once, and then continuously minimized via GD by updating both synapses and value nodes. An algorithmic formulation of PPC can be found in Alg.3.

2.2.3 Experimental evaluation

But where do PC models stand in terms of performance? To better show the capabilities of the introduced method, and variations, we test the performance of different experiments on standard benchmarks of small and medium size image classification, and compare them against BP.

Datasets: In our experimental evaluation, we use the following datasets, which will appear multiple times through this thesis:

- MNIST [63]: this dataset consists of 70000 black and white images, divided in a train-test split of $60000 - 10000$ images. Each image has size of 28×28 and represents a digit in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- FashionMNIST [64]: this dataset is slightly more complex than MNIST in terms of content, but comparable in any other aspect: it consists of 70000 black and white images, divided in a train-test split of $60000 - 10000$ images. Each image has size of 28×28 and represents a clothing item.
- *Street-view-house-number* (SVHN) dataset [65]. It consists of 60000 colored images of size 32×32, and is divided in a train-test split of $50000 - 10000$. Each image represents a picture of an house number taken from google maps, where only digits in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ are considered.
- CIFAR10 [33]. It consists of 60000 colored images of size 32×32, and is divided in a train-test split of $50000 - 10000$. It has 10 classes and each image represents a picture of an animal or big object, such as cars or boats.

Setup: The experiments are run using PC, PPC, and BP on (i) a MLP with 2 hidden layers and 64 hidden neurons per layer on the MNIST dataset; (ii) a mid-size convolutional neural network (CNN) with three convolutional layers with $64-128-64$ kernels followed by two fully connected layers on FashionMNIST, the Street View House Number (SVHN) dataset, and CIFAR10; and (iii) AlexNet [33], a large-scale CNN, on CIFAR10. In these experiments, a comprehensive grid-search on hyperparameters was performed: we tested over 8 learning rates (from 0.000001 to 0.01), 4 values of weight decay $(0.0001, 0.001, 0.01, 0.1)$, and 3 values of the integration step γ $(0.1, 0.5, 1.0)$. We report the average highest accuracies obtained with the same configuration after early stopping over 5 seeds. The results show that the performance of PC-based methods are comparable to the ones obtained by BP on equivalent models, as reported in Table 2.1.

Table 2.1: Final accuracy of BP, PC, and PPC on different architectures trained with different datasets.

	BP	PC	PPC
MLP on MNIST	$98.26\% \pm 0.12\%$	$98.55\% \pm 0.14\%$	$98.54\% \pm 0.86\%$
MLP on FashionMNIST	$88.54\% \pm 0.64\%$	$87.53\% \pm 0.75\%$	$89.13\% \pm 0.86\%$
CNN on SVHN	$95.35\% \pm 1.53\%$	$94.53\% \pm 1.54\%$	$96.45\% \pm 1.04\%$
CNN on CIFAR10	$69.34\% \pm 0.54\%$	$52.75\% \pm 0.64\%$	$72.54\% \pm 0.93\%$
AlexNet on CIFAR10	$75.64\% \pm 0.64\%$	$64.63\% \pm 1.55\%$	$72.42\% \pm 0.53\%$

Discussion: As already mentioned, the connections with BP for supervised learning suggests that PCNs should perform well on image classification tasks. This is indeed the case: PC is able to obtain performance comparable to BP on small MLPs trained on MNIST, as shown in [22]. A similar result was also obtained using a variation of PC not restricted to be trained using mean squared error as an energy function. It is in fact possible to define a generalization of PC, which uses layer-specific loss functions [27]. This variation, called *local representation alignment*, is able to reach a competitive performance with BP on MNIST and FashionMNIST. On more challenging tasks, a deep convolutional PCN is able to achieve a performance similar to BP on complex datasets, such as CIFAR10, and acceptable results on ImageNet [25]. This model updates the value nodes of one layer at a time, multiple times via lateral recurrent connections, before performing a weight update. It is intuitively similar to a deep convolutional network trained using the Z-IL algorithm, augmented with lateral connections.

2.3 Related work on predictive coding

The proposed methods do not cover all the applications of PC-inspired methods in the literature. In what follows, we will discuss the performance of these methods outside the supervised learning regime. Particularly, we will discuss about a probabilistic interpretation of PC, generative tasks on images and videos, and applications in control and robotics.

2.3.1 Predictive coding as variational inference

Mathematically, PCNs can also be expressed as variational inference on hierarchical Gaussian generative models. We define a hierarchy of layers indexed by l, where the distribution of activations at each layer is a Gaussian distribution with a mean given by a nonlinear function of the layer below $f(\bar{x}_{l+1})$ with parameters θ_{l+1} and an identity covariance I:

$$p(\bar{x}_{0,t} \dots \bar{x}_{L,t}) = p(\bar{x}_{L,t}) \prod_{l=0}^{L-1} p(\bar{x}_{l,t}|\bar{x}_{l+1,t})$$
$$p(\bar{x}_{l,t}|\bar{x}_{l+1,t}) = \mathcal{N}(\bar{x}_{l,t}; \theta_{l+1} f(\bar{x}_{l+1,t}), I)\,.$$

The PCN can then be "queried" by conditioning on a data or label item. For instance, suppose that the value nodes of the input layer $\bar{x}_L$ are fixed to some data item $\bar{s}_{in}$. We then wish to infer the state of the rest of the network given this conditioning $p(\bar{x}_{0,t} \dots \bar{x}_{L-1,t}|\bar{x}_L)$. This inference problem can be solved by variational inference [66, 67, 68]. In general, variational inference approximates an intractable inference through an optimization problem, where an approximate *variational posterior* distribution q is optimized, so as to minimize its distance from the optimal posterior p. This optimization procedure is performed by minimizing an upper bound on this divergence known as the *variational free energy* $\mathcal{F}_t$. All in all, this formulation defines a probabilistic model, where a posterior is parametrized through a PCN assuming a Gaussian distribution. We will now show how this formulation is connected to the one proposed above.

In PCNs, we assume that the variational posterior is factorized into independent posteriors for each layer $q(\bar{x}_{0,t} \dots \bar{x}_{L-1,t}) = \prod_{l=0}^{L-1} q(\bar{x}_{l,t})$ and, combined with the Laplace approximation [69], this allows us to considerably simplify the expression for the free energy into a sum of squared prediction errors (see [70]), equivalent to the one in Eq. 2.3 up to an additive constant:

$$\mathcal{F}_t \approx \sum_{l=0}^{L-1} \log p(\bar{x}_{l,t}|\bar{x}_{l+1,t}) \approx \sum_{l=0}^{L-1} \|\bar{\varepsilon}_{l,t}\|^2\,, \tag{2.10}$$

where $\bar{\varepsilon}_{l,t} = \bar{x}_{l,t} - \bar{\mu}_{l,t}$ is the prediction error of each layer. When applied to PCNs, we typically assume that the layerwise dependencies are parameterized by a parameter matrix

θ_{l+1}, which corresponds to the weights. Then, both the value nodes $\bar{x}_{l,t}$ and weights can be updated as a gradient descent on the free energy of Equation (2.10). Again, the updates of the parameters are divided in two phases, *inference* and *weight update*.

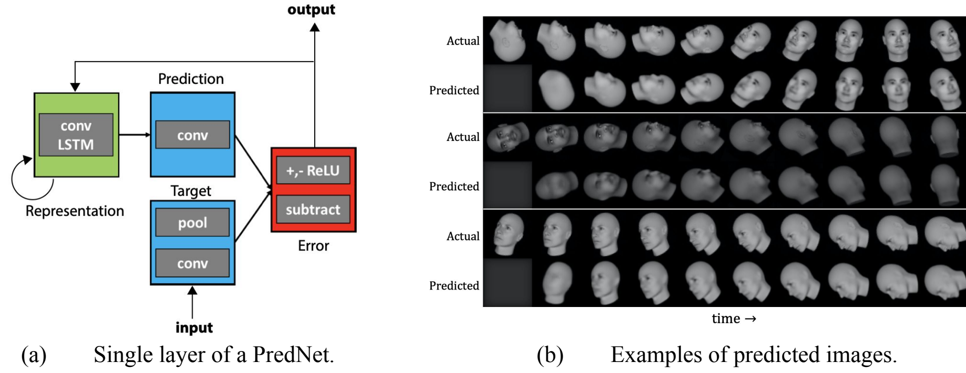
In the context of MLPs, PC recasts the feedforward pass in an ANN as an inference problem over the activations of the layers of an ANN given some conditioning on either input or output layers, or both, where the uncertainty about the "correct" activations is assumed to be Gaussian around a mean given by the top-down prediction from the layer above. Importantly, this inference problem is solved dynamically at each inference step, and the conditioning variables can be varied flexibly depending on the desired task. This enables the PCN to use its learned generative model (encoded in the weights θ_l) to be repurposed for different inference problems at run-time, and accounts for the superior flexibility of PCNs over MLPs, as it will be discussed in Chapter 5.

2.3.2 Generation

We have reviewed the connection between PC and BP for image classification, which suggests that it should be possible to use PC to train classifiers on large-scale datasets. PC is, however, a generative model, as suggested by its formulation as variational inference [20, 53]. This implies that the PC models surveyed in the previous section can also be used for data generation from labels. Additionally, PCNs can also be used directly as generative models due to their interpretation as probabilistic graphical models by swapping the "direction" of the network, so that the label is treated as the "input", and the data are treated as the "output". The basic architecture used to perform generative tasks resembles the decoder part of autoencoders. More complex models, which have a similar structure but are augmented with different kinds of connections, have been shown to generalize to unseen images [23, 71]. Particularly, three similar generative models have been presented in [23]: the first one is a novel model with recurrent connections, while the second and the third are implementations of Rao and Ballard's original PC [20], and of a model designed by Friston [72]. The extensive experiments provided in the paper show that generative PCNs are able to generate novel black and white images of different datasets. A qualitative evaluation against standard baselines in machine learning shows that this method obtains comparable results.

2.3.3 PredNet

Another successful line of work uses PC inspired architectures to predict the next frame of videos [31, 73]. This model is more complex than the standard PC models already reviewed in this survey, as every computational unit does not consist of a simple layer of neurons,



(a) Single layer of a PredNet. (b) Examples of predicted images.

Figure 2.3: Structure and epredictions of PredNets. Figures taken from the original paper [31].

but a large convolutional LSTM, which allows it to handle temporal data. However, it follows the error-driven framework of PC, where neural activities are propagated in one direction and the prediction error in the opposite one. Through the years, variations of the original PredNet model have been proposed and successfully used in a large number of applications, such as robotics [74, 75] and weather forecasting [76]. A deeper review of this model can be found in [30], where the authors test the model on challenging video-action classification datasets. A visual representation of a single layer of a PredNet is given in Fig. 2.3(a), while examples of predicted images on sequences of rendered rotating faces are shown in Fig. 2.3(b).

2.3.4 Predictive coding for control and robotics

In line with the free-energy principle [21, 77] in considering both perception and action as emerging from an imperative to minimize free energy, PC methods can also be directly applied to control problems, with close links to classical control theory, and have been applied productively to problems in robotics. Although the free energy does not explicitly include an action term, it includes one implicitly through the dependence on data $\bar{x}_0$ which could depend on actions a. This dependency can be computed explicitly by using the chain rule [78, 79],

$$\frac{da}{dt} \propto -\frac{\partial \mathcal{F}}{\partial a} = -\frac{\partial \mathcal{F}}{\partial \bar{x}_0} \cdot \frac{\partial \bar{x}_0}{\partial a}\,, \tag{2.11}$$

where $\partial \bar{x}_0 / \partial a$ is known as a *forward model*, which explicitly quantifies how data depend on observations. In the case of linear generative models and using generalized coordinates of motion [80], PC has been shown to be equivalent to PID control, a widely used and effective method in classical control theory [81]. PC also provides a generalization of PID

for both additional dynamical orders (i.e., using fourth- and higher-order derivatives) as well as instant generalizations to non-identity and ultimately nonlinear dynamics.

This method has also been widely used in robotics. The generative model in PC can be set to model the dynamics of a system and then torques can be inferred to realize a desired motion [82]. PC has thus been applied to a variety of robotics problems [82] as well as drone and quadcopter control [83, 84]. An additional advantage is that PC can also be used for state estimation [85, 86] (including with high dimensional image inputs [87], providing a joint solution of state estimation and control. For a recent full review of this literature, see [88].

More generally, if we consider PCNs as embodying an implicit probabilistic graphical model, and interpret some nodes of the PCN as "action nodes", then given that other nodes can be fixed as to a set of desired outcomes of control, then the inference process will infer the actions consistent with the conditioned outcomes [89, 90]. This mechanism is simply an example of the active inference [91, 92] and control as inference [93, 94, 95] framework, which interprets control problems as probabilistic inference problems, which simply require inferring the correct action or action sequences conditioned on achieving a high reward or desired state trajectory.

Chapter 3

Exact Backpropagation via Predictive Coding

Multiple algorithms have been historically proposed to replace backpropagation (BP) [96, 50]. However, none of them has succeeded, as they all fail to step up to large scale problems. BP, on the other hand, seem to have little limitation regarding the size of the model it is able to train successfully [41]. Theoretically, PC should not suffer of scalability problems, as we have seen that Z-IL is equivalent to BP in the way it updates its parameters on MLPs [24]. This is, however, not enough, as large architectures are not fully connected, and have complex and entangled structures.

In this chapter, I show how the equivalence result of Z-IL generalizes to some complex architectures, such as recurrent [97] and convolutional [63, 33] networks, and how it fails to generalize to others, such as transformer [17] and residual [34] networks. I first show these results empirically, and then formally prove the equivalence for convolutional and recurrent networks (CNNs and RNNs). This leaves the question on why it fails to generalize to more complex architectures unanswered. To address this, I first show the reason behind this failure, and solve it by adapting it to the very core framework governing the weight updates of deep learning models: computational graphs. Showing a result on computational graphs is quite important, as it means that it is true in the most general case, and specific cases can easily be derived from it.

Recently, neural networks have recently achieved amazing results in multiple fields, such as image recognition [34, 33], natural language processing [17, 40, 41], and game playing [37, 98]. All the models designed to solve these problems share a common ancestor, the multilayer perceptron (MLP), fully connected neural networks with a feedforward multilayer structure and a mapping function $\mathbb{R}^n \to \mathbb{R}^m$. Although MLPs are able to approximate any continuous function [99] and theoretically can be used for any task, none of the aforementioned successes has been reached merely using this architecture. In fact, complex and

task-oriented architectures perform significantly better than their fully connected counterpart, that, due to the lack of inductive bias towards the data, tend to easily overfit the dataset. Hence, the last decades have seen the use of different layer structures, such as recurrent neural networks (RNNs) [100], transformers [17], convolutional neural networks (CNNs) [63], and residual neural networks [34]. Albeit diverse architectures may look completely different, their parameters are all trained using gradient-based methods, creating a need for a general framework to efficiently compute gradients. Computational graphs, that are decompositions of complex functions into elementary ones, represent the ideal solution for this task, as they generalize the concept of a neural network. In fact, they allow the use of reverse differentiation to efficiently compute derivatives and hence update the parameters of the network, used in backpropagation to quickly propagate the output error through the network.

As already stated in the introduction, PC has been shown to be able to approximate asymptotically BP on MLPs, and on any other complex model [26]. Furthermore, Z-IL is a biologically plausible method with local connections and local plasticity, that can do exact backpropagation on MLPs. This shows the existence of a gap in our understanding of the biological plausibility of BP, which can be summarized as follows: there is an approximation result (PC), which has been shown to hold for any complex model [22, 26], and an exactness result (Z-IL), only proven for MLPs.

In this chapter, I close this gap by analyzing the Z-IL algorithm, and generalize the exactness result to every complex neural network. I start by performing one iteration of BP and one iteration of Z-IL on two identically initialized networks, and compare the two weight updates by computing their Euclidean distance. I have repeated this experiment for multiple architectures and parameterizations on different layer structures. The average results, reported in Table 3.1, show two interesting facts: first, they suggest that the exactness result holds for CNNs and many-to-one RNNs; second, they show that it does not hold for more complex architectures, such as residual and transformer neural networks. An analysis of the dynamics of the error propagation of Z-IL shows that the root of the problem is in the structure of the computational graph: in ResNets, for example, the skip connections design a pattern that does not allow Z-IL to exactly replicate the weight update of BP. The main topics covered in this chapter are briefly summarized as follows:

- I prove that Z-IL is also able to exactly implement BP on CNNs and RNNs, formally proving the results that I have shown empirically (3.1). To do that, I provide a direct derivation of the equations for Z-IL on CNNs and RNNs, and conclude by extending the proof of the original formulation of Z-IL on MLPs to these kind of architectures.

Table 3.1: Divergence between one update of weights of BP and Z-IL on different models, initialized in the same way.

	MLP	CNNs	RNNs	ResNet18	Transformer
Divergence:	0	0	0	4.53×10^7	7.29×10^4

- I then generalize Z-IL to the framework of computational graphs. I study why it is not possible to generalize the exactness result to any network structure, and show a practical, toy example, on residual networks. Understanding the behavior of Z-IL in this special and simple case allows to understand why the exactness result does not generalize to the general framework of computational graphs. Hence, I propose a variant of the Z-IL algorithm, more general and directly defined on computational graphs. This new variant is provably equivalent to BP in the way of updating parameters on any neural network, and hence closes the theoretical gap between BP and PC. A sketch of how this results fits in the literature can be found in Fig. 3.1.
- I then experimentally analyze the running time of Z-IL, PC, and BP on different architectures. The experiments show that Z-IL is comparable to BP in terms of efficiency, and several orders of magnitude faster than PC due to the few number of inference steps needed.

This novel formulation of BP in terms of PC may inspire other neuroscience-based alternatives to BP, which have both the advantages of biological neural systems (e.g., computations being local and parallel) and BP (e.g., having a high accuracy). At the same time, the first biologically plausible algorithm that exactly replicates the weight updates of BP on mapping functions of complex models may have an impact in neuroscience, as it shows that simulations of brain behavior using deep learning could be meaningful. This bridges the interdisciplinary research of neuroscience and deep learning, and allows PC to be considered as an alternative to BP, instead of just a theoretical tool to simulate information processing in the brain. A similar derivation of specific equations for CNNs and computational graph for standard PC can be found in [26]. As the proofs of the theorems of this chapter are similar, I provide a sketch of the general idea at the beginning of Section 3.8. In the same Section, a detailed proof of every specific theorem is also provided. To conclude, when introducing PCNs, I will always consider them trained with standard PC, unless otherwise specified, i.e., in the section *convolutional PCNs*, I consider convolutional PCNs trained with standard PC, and specify when it is not the case, i.e., in the section *convolutional PCNs trained with Z-IL*.

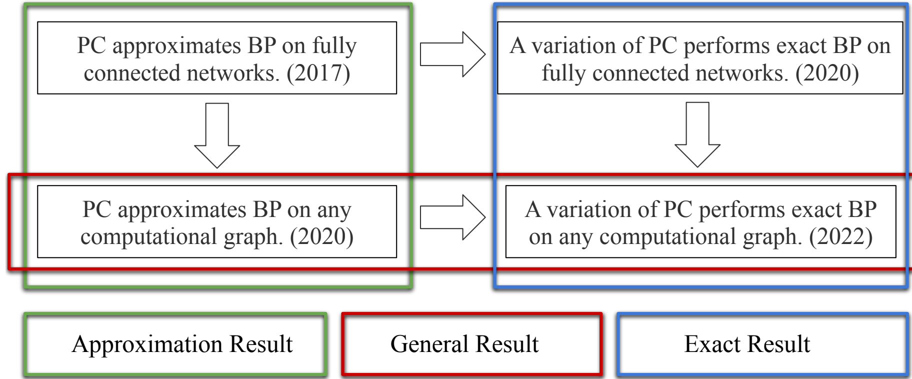


Figure 3.1: Historical and conceptual sketch of the results unifying predictive coding and backpropagation, that includes the works [22, 26, 24], and (red and blue) the main result of this chapter.

3.1 Convolutional neural networks

Convolutional neural networks (*CNNs*) are biologically inspired networks with a connectivity pattern (given by a set of *kernels*) that resembles the structure of the visual cortex. Networks with this particular architecture are widely used in image recognition tasks. A CNN is formed by a sequence of convolutional layers, followed by a sequence of fully connected ones. For simplicity of notation, we now consider convolutional layers with one kernel; we will then show how to extend the results to the general case.

3.1.1 Convolutional neural networks with backpropagation

The learnable parameters of a convolutional layer are contained in different kernels. Each kernel $\bar{\rho}^l$ can be seen as an m-dimensional vector that acts on the input vector $f(\bar{y}^l)$ using an operation "$*$", called *convolution*. This operation is equivalent to a linear transformation with a sparse matrix $\boldsymbol{W}^l$, whose non-zero entries equal to the entries of the kernel $\bar{\rho}^l$. This particular matrix is called doubly-block circulant matrix [101]. For every entry ρ_a^l of a kernel, we denote by $\mathcal{C}_a^l$ the set of indices (i,j) such that $\boldsymbol{W}_{i,j}^l = \rho_a^l$.

Let $f(\bar{y}^{l+1})$ be the input of a convolutional layer with kernel $\bar{\rho}^l$. The output $\bar{y}^l$ can then be computed as in the fully connected case: it suffices to use Eq. (2.9), where $\boldsymbol{W}^l$ is the doubly-block circulant matrix with parameters in $\bar{\rho}^l$. During the learning phase, BP updates the parameters of $\bar{\rho}^{l+1}$ according to the following equation:

$$\Delta\rho_a^{l+1} = -\alpha \cdot \partial F/\partial\rho_a^{l+1} = \textstyle\sum_{(i,j)\in\mathcal{C}_a^{l+1}} \Delta\boldsymbol{W}_{i,j}^{l+1}. \tag{3.1}$$

The value $\Delta\mathbf{W}_{\mathbf{i,j}}^{\mathbf{l}}$ can be computed using Eq. (2.8).

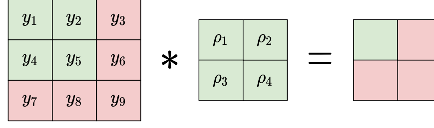


Figure 3.2: An example of a *convolution*.

3.1.2 Predictive coding convolutional networks

Given a convolutional network, we call $\overline{\lambda}^l$ the kernels of dimension k, and θ^l the related double-block circular matrix which describes the convolution operation. Note that $\overline{\lambda}^l$ mirrors $\overline{\rho}^l$ defined for standard CNNs, while θ^l mirrors $\boldsymbol{W}^l$. The dynamics of the feedforward pass is the same as the one described in the fully connected case. Hence, $\Delta x^l_{i,t}$ is computed as in Eq. (2.4). The update of the entries of the kernels, on the other hand, is the following:

$$\Delta\lambda_a^{l+1} = -\alpha \cdot \frac{\partial E_t}{\partial \lambda_a^{l+1}} = \sum_{(i,j)\in \mathcal{C}_a^{l+1}} \Delta\theta_{i,j}^{l+1}, \tag{3.2}$$

where $\Delta\theta^l_{i,j}$ is computed according to Eq. (2.5).

3.1.3 Predictive coding convolutional networks trained with Z-IL

Above, we have described the training and prediction phases on a single point $s = (\bar{s}^{\text{in}}, \bar{s}^{\text{out}})$ under different architectures. Similarly to how it has been explained in the preliminary chapter, the training phase of PC on a single point runs for T iterations, during which inference is conducted. So, the inference phase starts at $t = 0$ and ends at $t = T$, which is also when the network parameters are updated via Eqs. (2.5) and (3.2).

To show that PC is able to do exact BP on both the fully connected and convolutional layers of a CNN, we define constraints on the weights update of PC.

Z-IL: Let M be a PCN model with l_{max} layers. The inference phase runs for $T = l_{\text{max}}$ iterations. Instead of updating all the weights simultaneously at $t = T$, the parameters of every layer θ^{l+1} are updated at $t = l$. Hence, the prediction phase of Z-IL is equivalent to the one of PC, while the learning phase updates the parameters according to the equation

$$\Delta\theta_{i,j}^{l+1,t} = \begin{cases} 0 & \text{if } t \neq l \\ \alpha \cdot \varepsilon_i^l f(x_{j,t}^{l+1}) & \text{if } t = l. \end{cases} \tag{3.3}$$

I now show that, under a specific choice of hyperparameters, Z-IL is equivalent to BP on CNNs. Particularly, we add the following two conditions: $\varepsilon^l_{i,0} = 0$ for $l > 0$, and $\gamma = 1$.

The first condition can be obtained by setting $\bar{x}_0^l = \bar{\mu}_0^l$ for every $l > 0$ at the start of inference. Considering the vector $\bar{\mu}_0^l$ is computed from $\bar{x}_0^{l+1}$, this allows PC to start from a prediction stage equivalent to the one of BP. The second condition guarantees the propagation of the error during the inference phase to match the one of BP. Without it, Z-IL would be equivalent to a variation of BP, where the weight updates of single layers would have different learning rates.

The following theorem shows that Z-IL on convolutional PCNs is equivalent to BP on classical CNNs. Both a sketch of the proof and the complete version can be found in Section 3.8

Theorem 2 (Equivalence for CNNs)**.** *Let M be a CNN trained with BP, and let M' be its corresponding convolutional PCN, initialized as M, and trained with Z-IL with $\gamma = 1$ and $\varepsilon_{i,0}^l = 0$ for $l > 0$. Then, given the same training point $(\bar{s}_{in}, \bar{s}_{out})$ to both networks, we have*

$$\Delta\theta_{i,j}^{l+1} = \Delta \boldsymbol{W}_{i,j}^{l+1} \quad \textit{and} \quad \Delta\lambda_i^{l+1} = \Delta\rho_i^{l+1}, \tag{3.4}$$

for every $i, j, l \geq 0$.

3.2 Recurrent neural networks

While CNNs achieve impressive results in computer vision tasks, their performance drops when handling data with sequential structure, such as natural language sentences. An example is sentiment analysis: given a sentence S^{in} with words $(\bar{s}_1^{\text{in}}, \dots, \bar{s}_N^{\text{in}})$, predict whether this sentence is positive, negative, or neutral. To perform classification and regression tasks on this kind of data we normally use *many-to-one* RNNs. An example of such an architecture is shown in Fig. 3.3. In this section, we show that the results of Z-IL already existing for MLPs can be extended to RNNs as well. We first recall RNNs trained with BP, and then show how to define a recurrent PCN.

3.2.1 Recurrent neural networks with backpropagation

An RNN for classification and regression tasks has three different weight matrices $\boldsymbol{W}^x$, $\boldsymbol{W}^h$, and $\boldsymbol{W}^y$, N hidden layers of dimension n, and an output layer of dimension n^{out}. When it does not lead to confusion, we will alternate the notation between $k = \text{out}$ and $k = N + 1$. This guarantees a lighter notation in the formulas. A sequential input $S^{\text{in}} = \{\bar{s}_1^{\text{in}}, \dots, \bar{s}_N^{\text{in}}\}$ is a sequence of N vectors of dimension n^{in}. The first hidden layer is computed using the first vector of the sequential input, while the output layer is computed by multiplying the

last hidden layer by the matrix $\boldsymbol{W}^y$, i.e., $\bar{y}^{\text{out}} = \boldsymbol{W}^y \cdot f(\bar{y}^N)$. By assuming $\bar{y}^0 = \bar{0}$, the local computations of the network can be written as follows:

$$
\begin{aligned}
y_i^k &= \sum_{j=1}^{n} \boldsymbol{W}_{i,j}^h f(y_j^{k-1}) + \sum_{j=1}^{n^{\text{in}}} \boldsymbol{W}_{i,j}^x s_{k,j}^{\text{in}}, \\
y_i^{\text{out}} &= \sum_{j=1}^{n} \boldsymbol{W}_{i,j}^y f(y_j^N).
\end{aligned} \tag{3.5}
$$

Prediction: Given a sequential value S^{in} as input, every y_i^k in the RNN is computed via Eq. (3.5).

Learning: Given a sequential value S^{in} as input, the output $\bar{y}^{\text{out}}$ is then compared with the label $\bar{s}^{\text{out}}$ using MSE Loss. We now show how BP updates the weights of the three weight matrices. Note that $\boldsymbol{W}^y$ is a fully connected layer that connects the last hidden layer to the output layer. We have already computed this specific weight update in Eq. (2.8):

$$
\Delta \boldsymbol{W}_{i,j}^y = \alpha \cdot \delta_i^{\text{out}} f(y_j^N) \quad \text{with} \quad \delta_i^{\text{out}} = s_i^{\text{out}} - y_i^{\text{out}}. \tag{3.6}
$$

The gradients of F relative to the single entries of $\boldsymbol{W}^x$ and $\boldsymbol{W}^y$ are the sum of the gradients at each recurrent layer k. Thus,

$$
\begin{aligned}
\Delta \boldsymbol{W}_{i,j}^x &= \alpha \cdot \sum_{k=1}^{N} \delta_i^k s_{k,j}^{\text{in}} \\
\Delta \boldsymbol{W}_{i,j}^h &= \alpha \cdot \sum_{k=1}^{N} \delta_i^k f(y_j^{k-1}).
\end{aligned} \tag{3.7}
$$

The error term $\delta_i^k = \partial F / \partial y_i^k$ is defined as in Eq. (2.9):

$$
\delta_i^k = f'(y_i^k) \sum_{j=1}^{n} \delta_j^{k+1} \boldsymbol{W}_{j,i}^h. \tag{3.8}
$$

3.2.2 Predictive coding recurrent neural networks

We show how to define a recurrent PCN trained with PC. Recurrent PCNs have the same layer structure as the network introduced in the previous section. Hence, by assuming $\bar{x}^0 = \bar{0}$, the forward pass is given by as follows:

$$
\begin{aligned}
\mu_{i,t}^k &= \sum_{j=1}^{n} \theta_{i,j}^h f(x_{j,t}^{k-1}) + \sum_{j=1}^{n} \theta_{i,j}^x s_{k,j}^{\text{in}}, \\
\mu_{i,t}^{\text{out}} &= \sum_{j=1}^{n} \theta_{i,j}^y f(x_{j,t}^N).
\end{aligned} \tag{3.9}
$$

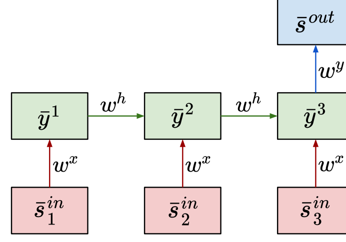


Figure 3.3: An example of a *many-to-one* RNN.

Here, θ^x, θ^h, and θ^y are the weight matrices paralleling $\boldsymbol{W}^x$, $\boldsymbol{W}^h$, and $\boldsymbol{W}^y$, respectively. The μ_i^k and x_i^k are defined as in the preliminaries. Again, error nodes computes the error between them $\varepsilon_{i,t}^k = x_{i,t}^k - \mu_{i,t}^k$. During the inference phase, the value nodes $x_{i,t}^k$ are updated to minimize the usual energy function E_t. During the learning phase, this update is done via:

$$\Delta x_{i,t}^k = \begin{cases} \gamma \cdot (-\varepsilon_{i,t}^k + f'(x_{i,t}^k) \sum_{j=1}^n \varepsilon_{j,t}^{k-1} \theta_{j,i}^h) & \text{if } k \geq 1 \\ 0 & \text{if } k = \text{out}. \end{cases} \tag{3.10}$$

Prediction: Given a sequential value S^{in} as input, every μ_i^k in the RNN is computed as the prediction via Eq. (3.9). Again, all error nodes converge to zero when $t \to \infty$, thus, $x_i^k = \mu_i^k$.

Learning: Given a sequential value S^{in} as input, the error in the output layer is set to $\varepsilon_{i,0}^{\text{out}} = s_i^{\text{out}} - \mu_{i,0}^{\text{out}}$. From here, the inference phase spreads the error among all the neurons of the network. Once this process has converged to an equilibrium, the parameters of the network get updated in order to minimize the total energy function. This causes the following weight updates:

$$\begin{aligned} \Delta\theta_{i,j}^x &= \alpha \cdot \textstyle\sum_{k=1}^N \varepsilon_{i,t}^k s_{k,j}^{\text{in}} \\ \Delta\theta_{i,j}^h &= \alpha \cdot \textstyle\sum_{k=1}^N \varepsilon_{i,t}^k f(x_j^{k-1}) \\ \Delta\theta_{i,j}^y &= \alpha \cdot \varepsilon_{i,t}^{\text{out}} f(x_j^N). \end{aligned} \tag{3.11}$$

3.2.3 Predictive coding recurrent networks trained with Z-IL

We now show that Z-IL can also be carried over and scaled to RNNs, and that the equivalence of Theorem 3 also holds for the considered RNNs. Both a sketch of the proof and the complete version can be found in Section 3.8

Theorem 3 (Equivalence for on RNNs)**.** *Let M be an RNN trained with BP, and let M' be the corrisponding predictive coding RNN initialized as M and trained with Z-IL with $\gamma = 1$*

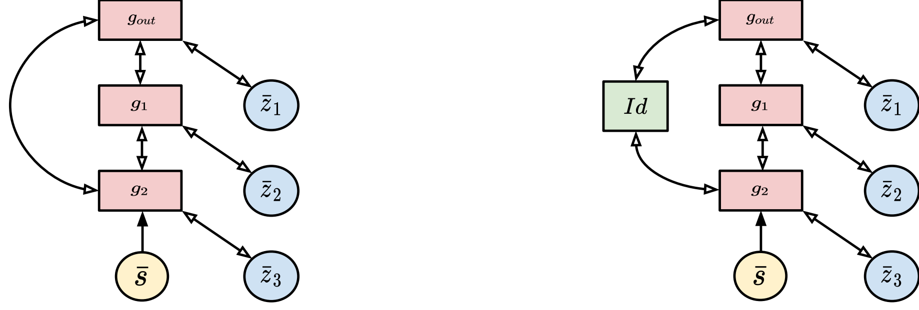


Figure 3.4: Left: computational graph of a 3-layer MLP with a residual connection, corresponding to the function $\mathcal{G}(s, \bar{z}) = sz_3 + sz_3z_2z_1$. Right: an equivalent computational graph, with the addition of an identity node.

and $\varepsilon^k_{i,0} = 0$ *for* $k > 0$. *Then, given the same sequential input* $S = \{\bar{s}_1, \ldots, \bar{s}_N\}$ *and label* s^{out} *to both,*

$$\begin{aligned} \Delta\theta^x_{i,j} &= \Delta \boldsymbol{W}^x_{i,j} \\ \Delta\theta^h_{i,j} &= \Delta \boldsymbol{W}^h_{i,j} \\ \Delta\theta^y_{i,j} &= \Delta \boldsymbol{W}^y_{i,j}, \end{aligned} \tag{3.12}$$

for every $i, j > 0$.

3.3 Computational graphs

A *computational graph* $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite nonempty set of vertices, and $\mathcal{E}$ is a finite set of edges, is a directed acyclic graph (DAG) that represents a complex function $\mathcal{G}$ as a composition of elementary functions. Every internal vertex v_i, i.e., not a leaf vertex of the graph, is associated with one elementary function g_i, and represents the computational step expressed by g_i. Every edge pointing to this vertex represents an input of g_i. For ease of presentation, the direction considered when using this notation is the reverse pass (downwards arrows in Fig. 3.5). Furthermore, we call $e_{i,j} \in E$ the directed edge that starts at v_i and ends at v_j. The first n vertices $v_1, \ldots, v_n$ are the leafs of the graph and represent the n inputs of the function $\mathcal{G}$, while the last vertex v^{out} represents the output of the function. we call d_i the minimum distance from the output node v^{out} to v_i (i.e., the length of the shortest path from v_{out} to v_i). An example of a computational graph for the function $\mathcal{G}(z_1, z_2) = (\sqrt{z_1} + z_2)^2$ is shown in Fig. 3.5, where the arrows pointing upwards denote the forward pass, and the ones pointing downwards the reverse pass. We call $C(i)$ and $P(i)$ the indices of the child and parent vertices of v_i, respectively. Hence, input vertices (at the

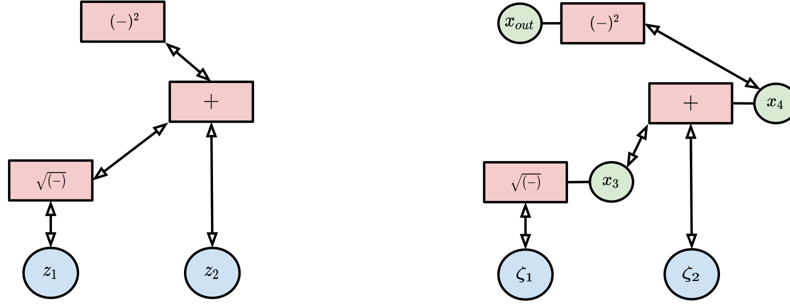


Figure 3.5: Left: computational graph of the function $\mathcal{G}(z_1,\, z_2) = (\sqrt{z_1} + z_2)^2$. Every internal vertex (red box) pictures its associated function g_i. Right: its predictive coding counterpart. Pointed upwards, the arrows related to the feedforward pass. The value nodes x_i of the input neurons are set to the input of the function (ζ_1 and ζ_2). Hence, we have omitted them from the plots to make the notation lighter. The same notation is adopted in later figures.

bottom) have no child vertices, and the output vertex v_{out} (at the top) has no parent vertices. we now briefly recall reverse differentiation, and so BP, on computational graphs, and show how it can be used to update the input variables in a supervised scenario.

3.3.1 Backpropagation on computational graphs

Let $\mathcal{G}$: $\mathbb{R}^n \to \mathbb{R}$ be a differentiable function, and $\{g_i\}$ be a factorization of $\mathcal{G}$ in elementary functions, which have to be computed according to a computational graph. Particularly, a computational graph $G = (\mathcal{V}, \mathcal{E})$ associated with $\mathcal{G}$ is formed by a set of vertices $\mathcal{V}$ with cardinality $|\mathcal{V}|$, and a set of directed edges $\mathcal{E}$. To every vertex $v_i \in \mathcal{V}$, we associate the corresponding elementary function g_i:$\mathbb{R}^{k_i} \to \mathbb{R}$, where k_i is the number of edges pointing to v_i (hence, the cardinality of the children set $C(i)$). The choice of these functions is not unique, as there exist infinitely many ways of factoring $\mathcal{G}$. It hence defines the structure of a particular computational graph. Given an input vector $\bar{z} \in \mathbb{R}^n$, we denote by μ_i the value of the vertex v_i during the forward pass. This value is computed iteratively as follows:

$$\mu_i = \begin{cases} z_i & \text{for } i \leq n\,; \\ g_i(\{\mu_j\}_{j \in C(i)}) & \text{for } i > n\,. \end{cases} \tag{3.13}$$

We then have $\mathcal{G}(\bar{z}) = \mu_{|V|} = \mu_{out}$. The computational flow just described is represented by the upward arrows in Fig. 3.5.

We now introduce the classical problem of reverse differentiation, and show how it is used to compute the derivative relative to the output. Let $\bar{z} = (z_1, \ldots, z_n)$ be an input (which in the case of MLPs will correspond to the weight parameters on the basis of which the output

of the network is computed, as it will be explained in the next section), and $\mathcal{G}(\bar{z}) = \mu_{out}$ be the output. Reverse differentiation is a key technique in machine learning and AI, as it allows to compute $\partial \mathcal{G}/\partial z_i$ for every $i < n$ efficiently, and hence update the weights. This is necessary to implement BP at a reasonable computational cost, especially considering the extremely overparameterized architectures used today. This is done iteratively as follows:

$$\frac{\partial \mathcal{G}}{\partial \mu_i} = \sum_{j \in P(i)} \frac{\partial \mathcal{G}}{\partial \mu_j} \cdot \frac{\partial \mu_j}{\partial \mu_i} = \sum_{j \in P(i)} \frac{\partial \mathcal{G}}{\partial \mu_j} \cdot \frac{\partial g_j}{\partial \mu_i}. \tag{3.14}$$

To obtain the desired formula for the input variables, it suffices to recall that $\mu_i = z_i$ for every $i \leq n$.

Update of the leaf nodes: Given an input $\bar{z}$, we consider a desired output y for the function $\mathcal{G}$. The goal of a learning algorithm is to update the input parameters $(z_1, \ldots, z_n)$ of a computational graph to minimize the MSE $F = \frac{1}{2}(\mu_{out} - y)^2$. Hence, the input parameters are updated by:

$$\Delta z_i = -\alpha \cdot \frac{\partial F}{\partial z_i} = \alpha \cdot \sum_{j \in P(i)} \delta_j \frac{\partial g_j}{\partial z_i}, \tag{3.15}$$

where α is the learning rate, and $\frac{\partial F}{\partial z_i}$ is computed using reverse differentiation. We use the parameter δ_j to represent the error signal, i.e., the propagation of the output error among the vertices of the graph. It can be computed according to the following recursive formula:

$$\delta_i = \begin{cases} \mu_{out} - y & \text{if } i = |V|; \\ \sum_{j \in P(i)} \delta_j \frac{\partial g_j}{\partial z_i} & \text{if } n < i < |V|. \end{cases} \tag{3.16}$$

3.3.2 Predictive coding on computational graphs

We now show how the just introduced forward and backward passes change when considering a PC computational graph $G = (\mathcal{V}, \mathcal{E})$ of the same function $\mathcal{G}$. We associate with every vertex v_i, with $i > n$, a new time-dependent random variable $x_{i,t}$, called *value node*, and a prediction error $\varepsilon_{i,t}$. We denote a parameter vector (which for MLPs corresponds to weights) by $(\zeta_1, \ldots, \zeta_n)$, so ζ_i in PC mirrors z_i in BP. The values μ_i are computed as follows: for the leaf vertices, we have $\mu_{i,t} = \zeta_i$ and $\varepsilon_{i,t} = 0$ for $i \leq n$, while for the other values, we have

$$\mu_{i,t} = g_i(\{x_{j,t}\}_{j \in C(i)}) \quad \text{and} \quad \varepsilon_{i,t} = \mu_{i,t} - x_{i,t}. \tag{3.17}$$

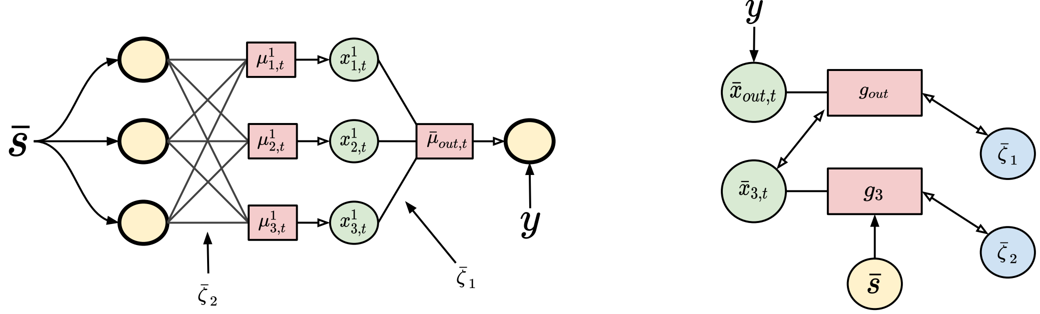


Figure 3.6: Left: example of a 2-layer PCN. In these networks, it is possible to realize every computation locally using error nodes and value nodes in a biologically plausible way. Right: the corresponding computational graph.

This allows to compute the value $\mu_{i,t}$ of a vertex by only using information coming from vertices connected to v_i. As in the case of PC networks, every computation is strictly local. The value nodes of the network are updated continuously to minimize the following loss function, defined on all the vertices of G:

$$E_t = \frac{1}{2} \sum_{i=1}^{|V|} (\varepsilon_{i,t})^2. \tag{3.18}$$

The output x_{out} of $\mathcal{G}(\bar{\zeta})$ is then computed by minimizing this energy function through an inference process. The update rule is $\Delta x_{i,t} = -\gamma\, \partial E_t / \partial x_{i,t}$, where γ is a small positive constant, called *integration step*. Expanding this gives:

$$\Delta x_{i,t} = -\gamma\, \partial E_t / \partial x_{i,t} = \gamma (\varepsilon_{i,t} + \sum_{j \in P(i)} \varepsilon_{j,t}\, \partial \mu_j / \partial x_{i,t}) \,. \tag{3.19}$$

Note that during the forward pass, all the value nodes $x_{i,t}$ converge to μ_i, as t grows to infinity. This makes the final output of the forward passes of inference learning on the PC formulation of computational graphs equivalent to that of standard computational graphs.

Update of the leaf nodes: Let $\bar{\zeta}$ be a parameter vector, and y be a fixed target. To update the parameter vector and minimize the error on the output, we fix $x_{out} = y$. Thus, we have $\varepsilon_{out,t} = \mu_{out} - y$. By fixing the value node $x_{out,t}$, the total error of the graph can no longer decay to zero. Hence, the output error $\varepsilon_{out,t}$ gets spread among the other error nodes on each vertex of the computational graph by running inference. When the inference process has either converged, or it has run for a fixed number of iterations T, the parameter vector gets updated by minimizing the same loss function E_t. Thus, we have:

$$\Delta \zeta_i = -\alpha \partial E_t / \partial \zeta_i = \alpha \sum_{j \in P(i)} \varepsilon_{j,t} \partial \mu_j / \partial \zeta_i. \tag{3.20}$$

All computations are local (with local plasticity) in PC, and the model can autonomously switch between prediction and learning via running inference. The main difference between BP and PC on computational graphs is that the update of the parameters of BP is invariant of the structure of the computational graph: the way of decomposing the original function $\mathcal{G}$ into elementary functions does not affect the update of the parameters. This is not the case for PC, as different decompositions lead to different updates of the value nodes, and so of the parameters. However, it has been shown that, while following different dynamics, these updates can be asymptotically equivalent under some specific conditions [26].

3.4 The problem of skip connections

In this section, we provide a toy example that shows how Z-IL and BP behave on the computational graph of a neural network with a skip connection. Particularly, we show that it is impossible for Z-IL to replicate the same update of BP on all the parameters, unless the structure of the computational graph is altered. Consider the following function, corresponding to a one dimensional linear MLP with a skip connection, represented in Fig. 3.4, left side:

$$\mathcal{G}(s, \bar{z}) = sz_3 + sz_3z_2z_1. \tag{3.21}$$

BP: Given an input value s and a desired target y, BP computes the gradient of every leaf node using reverse differentiation, and updates the parameters of z_3 as follows:

$$\Delta z_3 = -\alpha \cdot \frac{\partial F}{\partial z_3} = \alpha \cdot \delta(z_1z_2 + 1)s, \tag{3.22}$$

where $\delta = (\mu_{out} - y)$, and F is the quadratic loss defined on the output node.

Z-IL: Given an input value s and a desired target y, the inference phase propagates the output error through the graph via Eq. (3.20). Z-IL updates ζ_3 at $t = 3$, as it belongs to the third hidden layer. This leads to the following:

$$\Delta\zeta_3 = -\alpha \cdot \frac{\partial E_3}{\partial\zeta_3} = \alpha \cdot \delta\zeta_1\zeta_2 s, \tag{3.23}$$

where $\delta = \varepsilon_{out,0} = (\mu_{out,0} - y)$, and E_2 is computed according to Eq. (3.18). Note that this update is different from the one obtained by BP. We now analyze the reason for this mismatch and provide a solution.

Identity vertices The error signal propagated by the inference reaches ζ_3 in two different moments: $t = 2$ from the output vertex, and $t = 3$ from g_2. Dealing with vertices that receive error signals in different moments is problematic for the original formulation of the Z-IL algorithm, as every leaf node only gets updated once. Furthermore, changing the update rule of Z-IL does not solve the problem, as no other combination of updates produces the same weight update defined in Eq. (3.22). To solve this problem, we then have to assure that every node of the graph is reached by the error signal in a single time step. This is trivially obtained on computational graphs that are levelled DAGs, i.e., graphs where every directed path connecting two vertices has the same length. Here, the error reaches every vertex at a single, specific time step, no matter how complex the graph structure is. We now show how to make every computational graph levelled, without affecting the underlying function and the computations of the derivatives.

As previously mentioned, for every function $\mathcal{G}$ there exist infinitely many computational graphs that represent it. Every elementary function g_i can be written as a composition with the identity function, i.e., $g_i \circ Id$. Given two vertices v_i and v_j connected via the edge $e_{i,j}$, it is then possible to add a new vertex v_k by splitting the edge $e_{i,j}$ into $e_{i,k}$ and $e_{k,j}$, whose associated function g_k is the identity. This leaves the function expressed by the computational graph unvaried, as well as the computation of the derivatives, the forward pass, and the backward pass of BP. However, we will show that placing the identity vertices in the correct places, makes the computational graph levelled, allowing every vertex to receive the error signals at the same time step. Consider now the levelled graph of Fig. 3.4, right side, where an identity node has been added in the skip connection. The error signal of both g_1 and g_{out} reaches g_2 simultaneously at $t = 2$. Hence, at $t = 3$, Z-IL updates ζ_3 as follows:

$$\Delta\zeta_3 = -\alpha \cdot \frac{\partial E_3}{\partial \zeta_3} = \alpha \cdot \delta(\zeta_1\zeta_2 + 1)s. \tag{3.24}$$

If we start with $\zeta_i = z_i$, this weight update is equivalent to the one performed by BP and expressed in Eq. (3.22). Hence, Z-IL is able to produce the same weight update of BP in a simple neural network with one skip connection, thanks to a single identity vertex. In the next section, we generalize this result.

3.5 Levelled computational graphs

In this section, we show that, given any computational graph, it is always possible to generate an equivalent, levelled version of it. Particularly, we provide an algorithm that performs this task by adding identity nodes. This leads to the first result needed to prove our main

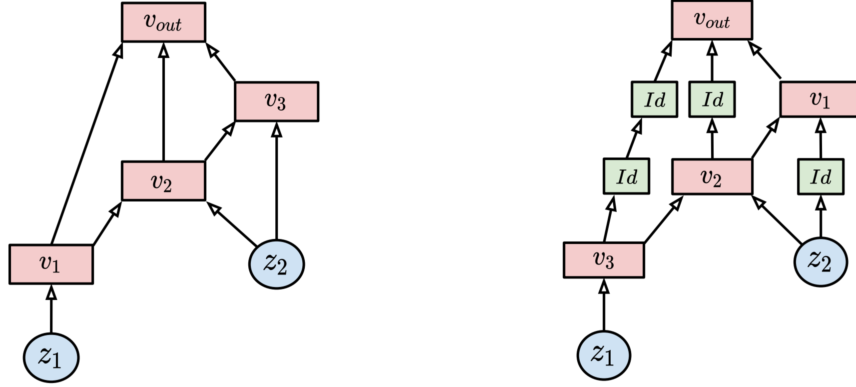


Figure 3.7: Computational graphs of the same function $\mathcal{G}$. Left: the original graph G. Right: the transformed graph, with the identity vertices in green.

theorem: given any function $\mathcal{G}$, it is always possible to consider an equivalent, levelled, computational graph. This allows to partition the nodes of G in a *level structure*, where a level structure of a directed graph is a partition of the vertices into subsets that have the same distance from the top vertex.

Let G be a computational graph, and $S_1, \dots, S_K$ be the family of subsets of $\mathcal{V}$ defined as follows: a vertex v_i is contained in S_k if there exists a directed path of length k connecting v_i to v_{out}, i.e.,

$$S_k = \{v_i \in \mathcal{V} | \ \exists \text{ a path } (e_{out,j_1}, \dots, e_{j_{k-1},i})\} \,. \tag{3.25}$$

Hence, we have that v_{out} is contained in S_0, its children vertices in S_1, and so on. Note that a graph is levelled if and only if every vertex is contained in one and only one of the subsets, and this partition defines its level structure. Let D_i be the maximum distance between v_{out} and the parent nodes of v_i, i.e., $D_i = \max_{v_j \in P(i)} d_j$. We now show for every DAG G how to make every vertex v_i to be contained in only one subset S_k, without altering the dynamics of the computational graph via the addition of identity nodes.

Let G be a DAG with root v_N, and let $(v_N, v_{N-1}, \dots, v_0)$ be a topological sort[1] of the vertices of G. Starting from the root, for every vertex v_j, we replace every existing edge $e_{i,j}$ with the following path:

$$v_i \to Id \to \cdots \to Id \to v_j, \tag{3.26}$$

which connects v_i to v_j via $d_j - D_i$ identity nodes. When this process has been repeated on all the vertices, we obtain a levelled DAG. This is equivalent to having every $v_i \in G$ that

[1]A topological sort of a DAG is a linear ordering of its vertices such that for every directed edge $e_{i,j}$ from vertex v_i to vertex v_j, v_i comes before v_j in the ordering. A topological sort always exists for DAGs

Algorithm 4 Generating a levelled DAG G' from G

Require: G is a DAG, and $(v_0, \ldots, v_n)$ a topological sort.
1: **for** every j in $(0, n)$ included **do**
2: **for** each vertex v_i in $P(j)$ **do**
3: Add $(d_j - D_i)$ identity vertices to $e_{i,j}$

belongs to one and only one subset S_k, as every pair of disconnected paths between two vertices has the same length, thanks to the addition of identity vertices.

More formally, for every vertex v_k, every path from v_0 to v_k in the new graph has length D_k: on any such path, take the last vertex $v_i \in \{v_0, v_1, \ldots, v_{k-1}\}$ preceding D_k. We must have taken D_i steps to get to v_i, and then followed the newly added path of length $D_k - D_i$, so the total length of the path is D_k. Note that this also apply to every newly added identity vertex v: let us assume v was added when we were building a path v_i to v_j, then every path from v_0 to v is the composition of a path v_0 to v_i, plus the initial segment of the path v_i to v_j ending at v. All paths v_0 to v_i have the same length, so we conclude all paths v_0 to v have the same length. Also, this process does not affect any path from v_0 to previously processed vertices, as we have only changed how we get from previous vertices to v_k. So, when we finally process the last vertex v_n, the graph will be levelled. Hence, we have the following:

Theorem 4 (Levelled computational graphs)**.** *Given a function $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$ and any factorization of it expressed by elementary functions $\{g_i\}$, there exist a levelled computational graph $G = (\mathcal{V}, \mathcal{E})$ that represents this factorization.*

The above theorem shows that every neural network can be expressed as a levelled computational graph, and hence that every result shown for levelled computational graphs can be naturally extended to every possible neural network.

3.6 Z-IL for levelled computational graphs

In this section, we show that a generalized version of Z-IL, called Z-IL for computational graphs, allows PCNs to do exact BP on any computational graph. Let $G = (\mathcal{V}, \mathcal{E})$ be the levelled computational graph of a function $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$, and consider the partition of $\mathcal{V}$ via its level structure $S_1, \ldots, S_K$. This algorithm is similar to PC, but the following two differences are introduced:

Forward pass: Differently from PC, where input and output are presented simultaneously, Z-IL first presents the input vector to the function, and performs a forward pass. Then, once the values μ_i of all the internal vertices have been computed, the value nodes are initialized

Algorithm 5 Z-IL for computational graphs.

Require: x_{out} is fixed to a label y,
Require: $\{S_k\}_{k=0,\dots,K}$ is a level structure of $G = (\mathcal{V}, \mathcal{E})$;
Require: $x_{i,0} = \mu_{i,0}$ for every internal node.
1: **for** $t = 0$ to K **do**
2: **for** each internal vertex v_i **do**
3: Update $x_{i,t}$ to minimize E_t via Eq. (3.19)
4: **if** $t = k$ **then**
5: Update each leaf node $\zeta_{i,t} \in S_k$ to minimize E_t via Eq. (3.20)

to have zero error, i.e., $x_{i,0} = \mu_i$, and the output node is set equal to the label y. This is done to emulate the behaviour of BP, which first computes the output vector, and then compares it to the label.

Update of the leaf nodes: Instead of continuously running inference on all the leaf nodes of G, we only run it on the internal vertices. Then, at every time step t, we update all the leaf nodes $v_i \in S_t$, if any. More formally, for every internal vertex v_i, training continues as usual via Eq. (3.19), while leaf nodes are updated according to the folllowing equation:

$$\Delta\zeta_{i,t} = \begin{cases} \gamma \cdot \sum_{j \in P(i)} \varepsilon_{j,t} \frac{\partial \mu_j}{\partial \zeta_i}. & \text{if } v_i \in S_t \\ 0 & \text{if } v_i \notin S_t. \end{cases} \tag{3.27}$$

This shows that one full update of the parameters requires $t = K$ steps. Overall, the functioning of Z-IL for computational graphs is summarized in Algorithm 5. We now show that this new formulation of Z-IL is able to replicate the same weight update of BP on any function $\mathcal{G}$. Both a sketch of the proof and the complete version can be found in Section 3.8

Theorem 5 (Equivalence for computational graphs)**.** *Let $(\bar{z}, y)$ and $(\bar{\zeta}, y)$ be two points with the same label y, and $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$ be a function. Assume that the update $\Delta\bar{z}$ is computed using BP, and the update $\Delta\bar{\zeta}$ using Z-IL with $\gamma = 1$. Then, if $\bar{z} = \bar{\zeta}$, and we consider a levelled computational graph of $\mathcal{G}$, we have*

$$\Delta z_i = \Delta\zeta_i \tag{3.28}$$

for every $i \leq n$.

This proves the main claims made about Z-IL: (i) exact BP and exact reverse differentiation can be made biologically plausible on the computational graph of *any* function, and (ii) Z-IL is a learning algorithm that allows PCNs to perfectly replicate the dynamics of BP on any function. Particularly, adding identity nodes to the computational graphs to

Table 3.2: Divergence between one update of weights of BP and Z-IL on different models, initialized in the same way.

	MLP	CNNs	RNNs	ResNet18	Transformer
Divergence:	0	0	0	0	0

produce equivalence to BP has non-trivial implications: it shows that the key difference between the PC model of learning in the brain and BP lies in the synchronization of error propagation. This offers a novel perspective to investigate the gap between BP and neural models. A formal proof of the above theorem is presented in Section 3.8. However, to empirically test the correctness of the theorem, we have replicated the experiments that have generated Tab. 3.1, and checked whether there was a divergence between BP and Z-IL for computational graphs. The results, presented in Tab. 3.2, confirm the correctness of our formal arguments.

3.7 Empirical evaluation

In the above sections, we have theoretically proved that the proposed generalized version of Z-IL is equivalent to BP on every possible neural model. Multiple experiments, reported further confirmed this: the divergences of weight updating between BP and Z-IL are always zero on all tested neural networks. So, there is no need for detailed experimental evaluation for the equivalence. In this section, we will complete the analysis via experimental studies that evaluate the computational efficiency of Z-IL, and quantitatively compare it with those of BP and PC. Particularly, we perform extensive experiments on different architectures, testing multiple models per architecture. The results of BP, PC, and Z-IL, averaged over all the experiments per model, are reported in Table 3.3. These are the same networks and set-ups used to produce Table 3.1 and Table 3.2.

MLP: I have trained three different PCNs and MLPs, with different depth (2, 3, and 4 layers, respectively) and hidden dimension 128 on a classification task, using the FashionMNIST dataset with batch size 20 and learning rate 0.01.

AlexNet and ResNet: For these architectures, we have trained them to classify pictures of the ImageNet dataset. We have defined the architectures as in the original works [33, 34]. The batch size and learning rate are the same as for the MLPs, and the results are reported in Tab. 3.3.

Table 3.3: Average running time of each weights update (in ms) of BP, PC, and Z-IL for computational graphs.

Method	MLP	AlexNet [33]	RNN	ResNet18 [34]	Transformer [17]
BP	3.72	8.61	5.64	12.43	20.43
PC	594.25	661.53	420.01	1452.34	1842.64
Z-IL	3.81	8.86	5.67	12.53	20.53

RNNs: To test our results on a RNN, we have trained a reinforcement learning agent on a single-layer many-to-one RNN on eight different Atari games. Similarly, the size of hidden and output layers is 128, batch size and learning rate are 20 and 0.01, respectively, and the reported results are averaged over eight games.

Transformer: We have trained a 1-layer transformer architecture using torchtext to generate a Wikitext-2 dataset. As parameters, we have used the same ones of the original work [17], while the batch size and learning rate were set to 4 and 0.01, respectively.

Results and evaluations As shown in Table 3.3, the computational time of Z-IL is very close to that of BP, and orders of magnitude lower than that of PC. This proves that Z-IL is an efficient alternative to BP in practice, instead of just being a theoretical tool. The high computational time of PC is due to the large number of iterations T. For example, for small MLPs, T is set to 20 in [22], and as larger models require higher numbers of iterations to converge, T is set between 100 and 200 for mid-size architectures, such as RNNs and CNNs in [26].

3.8 Proofs of theorems

The proofs of all the theorems follow a similar structure. We now provide a sketch of the general reasoning, and then provide each proof in detail. Each proof is divided into two steps, with each step proving a specific claim:

Step 1: For multilayer networks, at $t = l$, we have $\varepsilon^l_{i,l} = \delta^l_i$ for every l. For computational graphs, we have the distance from the vertex instead of the layer index l, i.e., $\varepsilon^l_{i,d_i} = \delta^{d_i}_i$, however, to keep the notation easier, in this sketch of the proofs we will use the same notation l for both layers of the network and levels of the graph. This does not lead to confusion, as the two proofs live in different frameworks: the concept of layer never appears in computational graphs, and the concept of distance never appears in neural networks. However, they are somehow related: in a multilayer network, the layer index l actually has distance l from the top in the respective computational graph.

This part of the proof is done by induction on the number of layers, or levels of the graph. We begin by noting that $W^l_{i,j} = \theta^l_{i,j}$, because of the same initialization of the network. This, together with the condition $\varepsilon^l_{i,0} = 0$ for $l > 0$, shows the equivalence of the initial states. The condition $\gamma = 1$ guarantees that the error $\varepsilon^l_{i,t}$ spreads as BP among different layers.

Step 2: We have $\Delta\theta^{l+1}_{i,j} = \Delta W^{l+1}_{i,j}$ for every $i, j, l \geq 0$.

The result proven in the first step allows to substitute $\varepsilon^l_{i,l} = \delta^l_i$ in the following equations:

$$\begin{aligned} \Delta W^{l+1}_{i,j} &= \alpha \cdot \delta^l_i f(y^{l+1}_j) \\ \Delta\theta^{l+1}_{i,j} &= \alpha \cdot \varepsilon^l_i f(x^{l+1}_{j,l}). \end{aligned} \tag{3.29}$$

The proof of this claim is sufficient to show the equivalence between BP and PCN in MLPs. Details on how to adapt this result to each specific case is provided in the following subsections.

3.8.1 Proof of Theorem 2

We will now provide the details needed to prove the following Theorem:

Theorem 6. *Let M be a CNN trained with BP, and let M' be its corresponding convolutional PCN, initialized as M, and trained with Z-IL with $\gamma = 1$ and $\varepsilon^l_{i,0} = 0$ for $l > 0$. Then, given the same training point $(\bar{s}_{in}, \bar{s}_{out})$ to both networks, we have*

$$\Delta\theta^{l+1}_{i,j} = \Delta W^{l+1}_{i,j} \quad \textit{and} \quad \Delta\lambda^{l+1}_i = \Delta\rho^{l+1}_i, \tag{3.30}$$

for every $i, j, l \geq 0$.

A convolutional network is formed by a sequence of convolutional layers followed by a sequence of fully connected ones. First, we prove the following:

Claim 1: At $t = l$, we have $\varepsilon^l_{i,t} = \delta^l_i$.

This first partial result is proven by induction on the depth l of the two networks, and does not change whether the layer considered is convolutional or fully connected. For PCNs, as $t = l$, it is also inducing on the inference moments. We begin by noting that, in Z-IL, $\varepsilon^l_{i,t} = \varepsilon^l_{i,l}$.

- *Base Case, $l = 0$*:

  The condition $\varepsilon^l_{i,0} = 0$ gives us $\mu^l_{i,0} = y^l_i$. Placing this result into Eq. (3.17) and Eq. (3.13), we get $\varepsilon^l_{i,l} = \delta^l_i$.

- *Induction Step:*. For $l \in \{1, \ldots, l_{\max} - 1\}$, we have:

$$\begin{aligned} \varepsilon^l_{i,l} &= f'(\mu^l_{i,0}) \textstyle\sum_{k=1}^{n^{l-1}} \varepsilon^{l-1}_{k,l-1} \theta^l_{k,i} && \text{by Lemma 3.8.1} \\ \delta^l_i &= f'(y^l_i) \textstyle\sum_{k=1}^{n^{l-1}} \delta^{l-1}_k \boldsymbol{W}^l_{k,i} && \text{by Eq. (2.9).} \end{aligned}$$

  Furthermore, note that $w^l_{i,j} = \theta^l_{i,j}$, because of the same initialization of the network, and $\mu^l_{i,0} = y^l_i$, because of $\varepsilon^l_{i,0} = 0$ for $l > 0$. Plugging these two equalities into the error equations above gives

$$\varepsilon^l_{i,l} = \delta^l_i, \;\; \text{if} \;\; \varepsilon^{l-1}_{k,l-1} = \delta^{l-1}_k. \tag{3.31}$$

  This concludes the induction step and proves the claim.

We now have to show the equivalence of the weights updates. We start our study from fully connected layers.

Claim 2: We have $\Delta\theta^{l+1}_{i,j} = \Delta\boldsymbol{W}^{l+1}_{i,j}$ for every $i, j, l \geq 0$.

Eqs. (3.15) and (3.20) state the following:

$$\begin{aligned} \Delta\theta^{l+1}_{i,j} &= \alpha \cdot \varepsilon^l_{i,l} f(x^{l+1}_{j,l}), \\ \Delta\boldsymbol{W}^{l+1}_{i,j} &= \alpha \cdot \delta^l_i f(y^{l+1}_j). \end{aligned}$$

Claim 1 gives $\varepsilon^l_{i,l} = \delta^l_i$. We now have to show that $f(x^{l+1}_{j,l}) = f(y^{l+1}_j)$. The equivalence of the initial state between PC and BP gives $x^{l+1}_{j,0} = \mu^{l+1}_{j,0} = y^{l+1}_j$. Then, Lemma 3.8.1 shows that $x^{l+1}_{j,l} = x^{l+1}_{j,0}$. So, $f(x^{l+1}_{j,l}) = f(y^{l+1}_j)$.

Claim 3: We have $\Delta\lambda^{l+1}_a = \Delta\rho^{l+1}_a$ for every $a, l \geq 0$.

The law that regulates the updates of the kernels is given by the following equations:

$$\Delta\lambda^{l+1}_a = -\alpha \cdot \frac{\partial E_t}{\partial \lambda^l_a} = \textstyle\sum_{(i,j)\in\mathcal{C}^{l+1}_a} \Delta\theta^l_{i,j} \tag{3.32}$$

$$\Delta\rho^{l+1}_a = -\alpha \cdot \partial F / \partial \rho^{l+1}_a = \textstyle\sum_{(i,j)\in\mathcal{C}^{l+1}_a} \Delta\boldsymbol{W}^{l+1}_{i,j}. \tag{3.33}$$

These equations are equal if $\Delta\theta^l_{i,j} = \Delta\boldsymbol{W}^l_{i,j}$ for every $i, j, l > 0$, which is the result shown in Claim 2. Thus, the weight update at every iteration of Z-IL is equivalent to the one of BP for both convolutional and fully connected layers.

Let M be a convolutional PCN trained with Z-IL with $\gamma=1$ and $\varepsilon^l_{i,0}=0$ for $l>0$. Then, a variable $\bar{x}^l_t$ can only diverge from its corresponding initial state at time $t = l$. Formally,

$$\overline{x}^l_{t<l} = \overline{x}^l_0, \overline{\varepsilon}^l_{t<l} = \overline{\varepsilon}^l_0 = 0, \overline{\mu}^{l-1}_{t<l} = \overline{\mu}^{l-1}_0, \text{ i.e.,}$$
$$\Delta\overline{x}^l_{t<l-1} = \overline{0}, \Delta\overline{\varepsilon}^l_{t<l-1} = \overline{0}, \Delta\overline{\mu}^{l-1}_{t<l-1} = \overline{0}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$.

Proof. Starting from the inference moment $t = 0$, $\overline{x}^0_0$ is dragged away from $\overline{\mu}^0_0$ and fixed to $\overline{s}^{\text{out}}$, i.e., $\overline{\varepsilon}^0_0$ turns into nonzero from zero. Since $\overline{x}$ in each layer is updated only on the basis of $\overline{\varepsilon}$ in the same and previous adjacent layer, as indicated by Eq. (3.3), also considering that $\varepsilon^l_{i,0}=0$, for all layers but the output layer, it will take l time steps to modify $\overline{x}^l_t$ at layer l from the initial state. Hence, $\overline{x}^l_t$ will remain in that initial state $\overline{x}^l_0$ for all $t < l$, i.e., $\overline{x}^l_{t<l} = \overline{x}^l_0$. Furthermore, any change in $\overline{x}^l_t$ causes a change in $\overline{\varepsilon}^l_t$ and $\overline{\mu}^{l-1}_t$ instantly via Eq. (3.17) (otherwise $\overline{\varepsilon}^l_t$ and $\overline{\mu}^{l-1}_t$ remain in their corresponding initial states). Thus, we know $\overline{\varepsilon}^l_{t<l} = \overline{\varepsilon}^l_0$ and $\overline{\mu}^{l-1}_{t<l} = \overline{\mu}^{l-1}_0$. Also, according to Eq. (3.3), $\overline{\varepsilon}^l_{t<l} = \overline{\varepsilon}^l_0 = 0$. Equivalently, we have $\Delta\overline{x}^l_{t<l-1} = \overline{0}$, $\Delta\overline{\varepsilon}^l_{t<l-1} = \overline{0}$, and $\Delta\overline{\mu}^{l-1}_{t<l-1} = \overline{0}$. □

Let M be a convolutional PCN trained with Z-IL with $\gamma=1$ and $\varepsilon^l_{i,0}=0$ for $l>0$. Then, the prediction error $\varepsilon^l_{i,t}$ at $t = l$ (i.e., $\varepsilon^l_{i,l}$) can be derived from itself at previous inference moments in the previous layer. Formally:

$$\varepsilon^l_{i,l} = f'(\mu^l_{i,0}) \textstyle\sum_{k=1}^{n^{l-1}} \varepsilon^{l-1}_{k,l-1} \theta^l_{k,i}, \tag{3.34}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$.

Proof. We first write a dynamic version of $\varepsilon^l_{i,t} = x^l_{i,t} - \mu^l_{i,t}$:

$$\varepsilon^l_{i,t} = \varepsilon^l_{i,t-1} + \left(\Delta x^l_{i,t-1} - \Delta\mu^l_{i,t-1}\right), \tag{3.35}$$

where $\Delta\mu^l_{i,t-1} = \mu^l_{i,t} - \mu^l_{i,t-1}$. Then, we expand $\varepsilon^l_{i,l}$ with the above equation and simplify it with Lemma 3.8.1, i.e., $\varepsilon^l_{i,t<l} = 0$ and $\Delta\mu^{l-1}_{i,t<l-1} = 0$:

$$\varepsilon^l_{i,l} = \varepsilon^l_{i,l-1} + \left(\Delta x^l_{i,l-1} - \Delta\mu^l_{i,l-1}\right) = \Delta x^l_{i,l-1}, . \tag{3.36}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$. We further investigate $\Delta x^l_{i,l-1}$ expanded with the inference dynamic Eq. (3.3) and simplify it with Lemma 3.8.1, i.e., $\varepsilon^l_{i,t<l} = 0$,

$$\Delta x^l_{i,l-1} = \gamma(-\varepsilon^l_{i,l-1} + f'(x^l_{i,l-1})) \textstyle\sum_{k=1}^{n^{l-1}} \varepsilon^{l-1}_{k,l-1} \theta^l_{k,i} \tag{3.37}$$

$$= \gamma f'(x^l_{i,l-1}) \textstyle\sum_{k=1}^{n^{l-1}} \varepsilon^{l-1}_{k,l-1} \theta^l_{k,i}, \tag{3.38}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$. Putting Eq. (3.38) into Eq. (3.36), we obtain:

$$\varepsilon_{i,l}^{l} = \gamma f'(x_{i,l-1}^{l}) \sum_{k=1}^{n^{l-1}} \varepsilon_{k,l-1}^{l-1} \theta_{k,i}^{l}, \tag{3.39}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$. With Lemma 3.8.1, $x_{i,l-1}^{l}$ can be replaced with $x_{i,0}^{l}$. With $\varepsilon_{i,0}^{l} = 0$ for $l > 0$, we can further replace $x_{i,0}^{l}$ with $\mu_{i,0}^{l}$. Thus, the above equation becomes:

$$\varepsilon_{i,l}^{l} = \gamma f'(\mu_{i,0}^{l}) \sum_{k=1}^{n^{l-1}} \varepsilon_{k,l-1}^{l-1} \theta_{k,i}^{l}, \tag{3.40}$$

for $l \in \{1, \ldots, l_{\max} - 1\}$. Then, put $\gamma = 1$, into the above equation. □

Extension to the case of multiple kernels per layer: In the theorem proved in the previous section, we have only considered CNNs with one kernel per layer. While networks of this kind are theoretically interesting, in practice a convolutional layer is made of multiple kernels. We now show that the result of Theorem 2 still holds if we consider networks of this kind. Let M_l be the number of kernels present in layer l. In Theorem 2, we have considered the case $M_l = 1$ for every convolutional layer. Consider now the following three cases:

- **Case 1:** $M_l > 1, M_{l-1} = 1$. We have a network with a convolutional layer at position l with M_l different kernels $\{\bar{\rho}^{l,1}, \ldots, \bar{\rho}^{l,M_l}\}$ of the same size k. The result of the convolution between the input $f(\bar{y}^l)$ and a single kernel $\bar{\rho}^{l,m}$ is called *channel*. The final output $\bar{y}^{l-1}$ of a convolutional layer is obtained by concatenating all the channels into a single vector. The operation generated by convolutions and concatenation just described, can be written as a linear map $\boldsymbol{W}^l \cdot f(\bar{y}^l)$, where the matrix $\boldsymbol{W}^l$ is formed by M_l doubly-block circulant matrices stocked vertically, each of which has entries equal to the ones of kernel $\bar{\rho}^{l,m}$. For each entry $\rho_a^{l,m}$ of each kernel in layer l, we denote by $\mathcal{C}_{m,a}^{l}$ the set of indices (i, j) such that $\boldsymbol{W}_{i,j}^{l} = \rho_a^{l,m}$. The equation describing the changes of parameters in the kernels is then the following:

$$\Delta\rho_a^{l,m} = -\alpha \cdot \partial F / \partial \rho_a^{l,m} = \sum_{(i,j) \in \mathcal{C}_{m,a}^{l}} \Delta \boldsymbol{W}_{i,j}^{l}. \tag{3.41}$$

- **Case 2:** $M_l = 1, M_{l-1} > 1$. We now analyze what happens in a layer with only one kernel, when the input $f(\bar{y}^{l-1})$ comes from a layer with multiple kernels. This case differs from Case 1, because the input represents a concatenation of M_{l-1} different channels. In fact, the kernel $\bar{\rho}^l$ gets convoluted with every channel independently. The resulting vectors of these convolutions are then summed together, obtaining $\bar{y}^l$. The operation generated by convolutions and summations just described, can be written as

a linear map $\boldsymbol{W}^l \cdot f(\bar{y}^l)$. In this case, the matrix $\boldsymbol{W}^l$ is formed by M_{l-1} doubly-block circulant matrices stocked horizontally, each of which has entries equal to the ones of the kernel $\bar{\rho}^l$. For every entry ρ_a^l, we denote by $\mathcal{C}_a^l$ the set of indices (i,j) such that $\boldsymbol{W}_{i,j}^l = \rho_a^l$. The equation that describes the changes of parameters in the kernels is then the following:

$$\Delta\rho_a^l = -\alpha \cdot \partial F/\partial \rho_a^l = \textstyle\sum_{(i,j)\in\mathcal{C}_a^l} \Delta \boldsymbol{W}_{i,j}^l. \tag{3.42}$$

- **Case 3 (General Case):** $M_l, M_{l-1} > 1$. We now move to the most general case: a convolutional layer at position l with M_l different kernels $\{\bar{\rho}^{l,1},\ldots,\bar{\rho}^{l,M_l}\}$, whose input $f(\bar{y}^l)$ is a vector formed by M_{l-1} channels. In this case, every kernel does a convolution with every channel. The output $\bar{y}^{l+1}$ is obtained as follows: the results obtained using the same kernel on different channels are summed together, and concatenated with the results obtained using the other kernels. Again, this operation can be written as a linear map $\boldsymbol{W}^l \cdot f(\bar{y}^l)$. By merging the results obtained from Case 1 and Case 2, we have that the matrix $\boldsymbol{W}^l$ is a grid of $M_l \times M_{l+1}$ doubly-block circulant submatrices. For every entry $\rho_a^{l,m}$ of every kernel in layer l, we denote by $\mathcal{C}_{m,a}^l$ the set of indices (i,j) such that $\boldsymbol{W}_{i,j}^l = \rho_a^{l,m}$. The equation describing the changes of parameters in the kernels is then the following:

$$\Delta\rho_a^{l,m} = -\alpha \cdot \partial F/\partial \rho_a^{l,m} = \textstyle\sum_{(i,j)\in\mathcal{C}_{m,a}^l} \Delta \boldsymbol{W}_{i,j}^l. \tag{3.43}$$

To integrate this general case in the proof of Theorem 2, it suffices to consider Eq. (3.43), and its equivalent formulation in the language of a convolutional PCN,

$$\Delta\rho_a^{l+1} = -\alpha \cdot \partial F/\partial \rho_a^{l+1} = \textstyle\sum_{(i,j)\in\mathcal{C}_a^{l+1}} \Delta \boldsymbol{W}_{i,j}^{l+1} \tag{3.44}$$

instead of Eqs. (3.32) and (3.33). Note that both equations are fully determined once we have computed $\Delta\boldsymbol{W}_{i,j}^l$ and $\Delta\theta_{i,j}^l$ for every $i,j > 0$. Hence, the result follows directly by doing the same computations.

3.8.2 Proof of Theorem 3

Theorem 7. *Let M be an RNN trained with BP, and let M' be the corrisponding predictive coding RNN initialized as M and trained with Z-IL with $\gamma = 1$ and $\varepsilon^k_{i,0} = 0$ for $k > 0$. Then, given the same sequential input $S = \{\bar{s}_1, \ldots, \bar{s}_N\}$ and label s^{out} to both,*

$$\begin{aligned} \Delta\theta^x_{i,j} &= \Delta \boldsymbol{W}^x_{i,j} \\ \Delta\theta^h_{i,j} &= \Delta \boldsymbol{W}^h_{i,j} \\ \Delta\theta^y_{i,j} &= \Delta \boldsymbol{W}^y_{i,j}, \end{aligned} \tag{3.45}$$

for every $i, j > 0$.

The network M has depth 2; hence, we set $T = 2$. We now prove the following three equivalences: (1) $\Delta\theta^y = \Delta\boldsymbol{W}^y$, (2) $\Delta\theta^h = \Delta\boldsymbol{W}^h$, and (3) $\Delta\theta^x = \Delta\boldsymbol{W}^x$.

The proof of (1) is straightforward, since both the output layers θ^y and $\boldsymbol{W}^y$ are fully connected. Particularly, we have already shown the equivalence for this kind of layers in Theorem 2. Before proving (2) and (3), we show an intermediate result needed in both cases.

Claim: Given a sequential input S^{in} of length N, at $t = 1$ we have $\varepsilon^k_{i,1} = \delta^k_i$ for every $k \leq N$.

This part of the proof is done by induction on N.

- Base Case: $N = 1$. Given a sequential input of length 1, we have a fully connected network of depth 2 with $\boldsymbol{W}^1 = \boldsymbol{W}^y$ (resp. $\theta^1 = \theta^y$) and $\boldsymbol{W}^2 = \boldsymbol{W}^x$ (resp. $\theta^2 = \theta^x$). We have already proved this result in Theorem 2.

- Induction Step. Let us assume that, given a sequential input S^{in} of length N, the claim $\varepsilon^k_{i,1} = \delta^k_i$ holds for every $k \in \{1, \ldots, N\}$. Let us now assume we have a sequential input of length $N+1$. Note that the errors $\varepsilon^k_{i,1}$ and δ^k_i are computed backwards starting from $k = N+1$. Hence, the quantities $\varepsilon^k_{i,1}$ and δ^k_i for $k \in \{2, \ldots, N+1\}$ are computed as they were the errors of a sequential input of length N. It follows by the induction argument that $\varepsilon^k_{i,1} = \delta^k_i$ for every $k \in \{2, \ldots, N+1\}$. To conclude the proof, we have to show that $\varepsilon^1_{i,1} = \delta^1_i$. For $k = 1$, we have:

$$\begin{aligned} \varepsilon^1_{i,l} &= f'(\mu^1_{i,0}) \textstyle\sum_{j=1}^n \varepsilon^2_{j,t} \theta^h_{j,i} && \text{by Lemma 3.8.2} \\ \delta^1_i &= f'(y^1_i) \textstyle\sum_{j=1}^n \delta^2_j \boldsymbol{W}^h_{j,i}. && \text{by Eq. (3.8).} \end{aligned}$$

Note that $W^h_{i,j} = \theta^h_{i,j}$, because of the same initialization of the network. Furthermore, $\mu^k_{i,0} = y^k_i$ for every k because of $\varepsilon^k_{i,0} = 0$. Plugging these two equalities into the error equations above gives $\varepsilon^1_{i,1} = \delta^1_i$. This concludes the induction step and proves the claim.

(2) $\Delta\theta^h = \Delta W^h$. Recall that Eqs. (3.11) and (3.7) state that

$$\Delta\theta^h_{i,j} = \alpha \cdot \sum_{k=1}^{N} \varepsilon^k_{i,t} f(x^{k-1}_{j,1})$$
$$\Delta W^h_{i,j} = \alpha \cdot \sum_{k=1}^{N} \delta^k_i f(y^{k-1}_j).$$

The claim shown above gives $\varepsilon^k_{i,1} = \delta^k_i$. We thus have to show that $x^k_{j,1} = y^k_j$. The condition $\varepsilon^k_{j,0} = 0$ gives $x^k_{j,0} = \mu^k_{j,0} = y^k_j$. Moreover, by Lemma 3.8.1, $x^k_{j,1} = x^k_{j,0}$. So, $x^k_{j,1} = y^k_j$.
(3) $\Delta\theta^x = \Delta W^x$. Recall that Eqs. (3.11) and (3.7) state that

$$\Delta\theta^x_{i,j} = \alpha \cdot \sum_{k=1}^{N} \varepsilon^k_{i,t} s^{\text{in}}_{k,j}$$
$$\Delta W^x_{i,j} = \alpha \cdot \sum_{k=1}^{N} \delta^k_i s^{\text{in}}_{k,j}.$$

The equality $\Delta\theta^x = \Delta W^x$ directly follows from $\varepsilon^k_{i,1} = \delta^k_i$.

Let M be a recurrent PCN trained with Z-IL on a sequential input S^{in} of length N. Furthermore, let us assume that $\gamma = 1$ and $\varepsilon^k_{i,0} = 0$ for every $k \in \{1, \dots, N\}$. Then, the prediction error $\varepsilon^k_{i,t}$ at $t = 1$ (i.e., $\varepsilon^k_{i,1}$) can be derived from the previous recurrent layer. Formally:

$$\varepsilon^k_{i,1} = f'(\mu^k_{i,0}) \sum_{j=1}^{n^{k+1}} \varepsilon^{k+1}_{j,1} \theta^h_{j,i}, \tag{3.46}$$

for $k \in \{1, \dots, N-1\}$.

Proof. Equivalent to the one of Lemma 3.8.1. The only difference is that in Lemma 3.8.1 we iterate over the previous layer l at time $t = l$, while here the iterations happen over the previous recurrent layer k at fixed time $t = 1$. □

3.8.3 Proof of Theorem 5

In this section, we prove the main theorem of this chapter, which generalizes the previous ones to work for any computational graph:

Theorem 8. *Let $(\bar{z}, y)$ and $(\bar{\zeta}, y)$ be two points with the same label y, and $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$ be a function. Assume that the update $\Delta\bar{z}$ is computed using BP, and the update $\Delta\bar{\zeta}$ using Z-IL with $\gamma = 1$. Then, if $\bar{z} = \bar{\zeta}$, and we consider a levelled computational graph of $\mathcal{G}$, we have*

$$\Delta z_i = \Delta\zeta_i \tag{3.47}$$

for every $i \leq n$.

As Z-IL acts on the levelled version of G, in this proof we consider levelled computational graphs, i.e., graphs where the distance from the top generates a partition of the vertices. We denote d_i the distance of a vertex v_i to the root vertex v_{out}, i.e., $d_i = k$ if $v_i \in S_k$. Furthermore, we denote by d_{max} the maximum distance between the root and any vertex v_i, i.e., $d_{max} = \max_i d_i$.

I divide the proof in two parts, Claim 1 and Claim 2. The first part of the proof (i.e., Claim 1) consists in showing that the errors δ_i and $\varepsilon_{i,t}$ are equal when $v_i \in S_k$ and $t = k$, which is the time at which the input parameters get updated. Particularly:

Claim 1**:** At any fixed time t, we have $\varepsilon_{i,t} = \delta_i$ for every $v_i \in S_t$.

We prove this claim by induction on d_{max}. Let us start with the *basic step* $d_{max} = 1$:

We have the output vertex v_{out} and leaf vertices. The value $\mu_{out,t}$ of the output node is given by the elementary function g_{out} defined on all the input variables. Hence, we have

$$\delta_i = \varepsilon_{i,0} = \mu_{out,t} - y. \tag{3.48}$$

This proves the basic case. Now we move to the *induction step*: let us assume that Claim 1 holds for every computation graph with $d_{max} = m$.

Let $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$ be a function whose computation graph $G = (\mathcal{V}, \mathcal{E})$ has $d_{max} = m + 1$. For every non-leaf node v_i such that $d_i < m$ and $v_i \in S_t$, we have that $\delta_i = \varepsilon_{i,t}$. Furthermore, note that $\varepsilon_{i,t} = \varepsilon_{i,d_i}$.

$$\varepsilon_{i,d_i} = \sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_{j,d_i}}{\partial x_{i,0}} \qquad \text{by Lemma 3.8.3,}$$

$$\delta_i = \sum_{j \in P(i)} \delta_j \frac{\partial \mu_j}{\partial \mu_i} \qquad \text{by Eq. (3.16).}$$

The two quantities above are equal. This follows from the induction step, which gives $\varepsilon_{j,d_i-1} = \delta_i$ and from the condition that states that $\mu_{i,t} = x_{i,0}$ for $d < d_i$. This concludes the proof of Claim 1.

Claim 2**:** We have $\Delta z_i = \Delta \zeta_i$ for every $i \leq n$.

Eqs. (3.15) and (3.20) state the following:

$$\Delta z_i = \alpha \cdot \sum_{j \in P(i)} \delta_j \frac{\partial \mu_j}{\partial z_i},$$

$$\Delta \zeta_i = \alpha \sum_{j \in P(i)} \varepsilon_{j,t} \frac{\partial \mu_{j,t}}{\partial \zeta_i}.$$

The update of every input parameter ζ_i in Z-IL happens at $t = d_i$. Claim 1 shows that, at that specific time, we have $\delta_i = \varepsilon_{i,t}$, while Lemma 3.8.3 states that $\mu_{i,t} = \mu_{i,0}$ for every $t \leq d_i$. The proof of the claim, and hence, the whole theorem, follows from $\zeta_i = z_i$ for every $i \leq n$.

Let $\bar{\zeta}$ be an input of a continuous and differentiable function $\mathcal{G} : \mathbb{R}^n \to \mathbb{R}$ with computational graph $G = (\mathcal{V}, \mathcal{E})$, and also assume that the update $\Delta\bar{\zeta}$ using Z-IL with the partition of $\mathcal{V}$ described by Eq. (3.25), we then have $\mu_{i,t} = \mu_{i,0}$ and $\varepsilon_{i,t} = 0$ for every $t \leq d_i$.

Proof. This directly follows from the fact that we are applying Z-IL on a levelled graph. In fact, the value μ_{i,d_i} of every vertex v_i differs from its initial state $\mu_{i,0}$ only if the node values $\{x_{j,t}\}_{j \in C(i)}$ of the children vertices have changed in the time interval $[0, d_i]$. This may only happen if we have $d_j < d_i$ for one of the vertices $\{v_j\}_{j \in C(i)}$. But this is impossible, as the distance from the top d_i of a parent node is always strictly smaller than the one of any of its children nodes in a levelled graph. □

The prediction error in Z-IL at $t = d_i$, i.e., $\varepsilon_{i,t}$, can be derived from itself at previous inference moments. Formally,

$$\varepsilon_{i,d_i} = \gamma \sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_{j,d_i}}{\partial x_{i,0}}. \tag{3.49}$$

Proof. Let us write $\varepsilon_{i,t}$ as a function of $\varepsilon_{i,t-1}$:

$$\varepsilon_{i,t} = \varepsilon_{i,t-1} + \left(\Delta x_{i,t-1} - \Delta \mu_{i,t-1}\right), \tag{3.50}$$

where $\Delta\mu_{i,t-1} = \mu_{i,t} - \mu_{i,t-1}$. Then, we expand ε_{i,d_i} with the above equation and simplify it with Lemma 3.8.3, i.e., $\varepsilon_{i,d_i-1} = 0$ and $\Delta\mu_{i,t<d_i-1} = 0$:

$$\varepsilon_{i,d_i} = \varepsilon_{i,d_i-1} + \left(\Delta x_{i,d_i-1} - \Delta\mu_{i,d_i-1}\right) = \Delta x_{i,d_i-1}. \tag{3.51}$$

We further investigate $\Delta x_{i,d_i-1}$ expanded with the inference dynamic Eq. (3.19) and simplify it with Lemma 3.8.3, i.e., $\varepsilon_{i,t<d_i} = 0$,

$$\Delta x_{i,d_i-1} = \gamma\big(\varepsilon_{i,d_i-1} + \textstyle\sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_j}{\partial x_{i,d_i-1}}\big) \tag{3.52}$$

$$= \gamma \textstyle\sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_j}{\partial x_{i,d_i-1}}. \tag{3.53}$$

Putting Eq. (3.53) into Eq. (3.51), we obtain:

$$\varepsilon_{i,d_i} = \gamma \sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_{j,d_i}}{\partial x_{i,d_i-1}} \quad (3.54)$$

$$= \gamma \sum_{j \in P(i)} \varepsilon_{j,d_i-1} \frac{\partial \mu_{j,d_i}}{\partial x_{i,0}}. \quad (3.55)$$

With Lemma 3.8.3, x_{i,d_i-1} can be replaced with $x_{i,0}$. □

3.9 Why is this equivalence important?

In this chapter, I have explored the pattern recognition capabilities of PC, by comparing them to BP, the best learning algorithm available at the moment, showing that these capabilities are actually identical. Particularly, I have generalized existing equivalence results to work on any graph architecture. This is a step into partially filling an existing gap between machine learning and neuroscience: on the one hand, recent neural architectures trained by BP reach impressive performance in machine learning; on the other hand, models in neuroscience can only match the performance of BP in small-scale problems. There is thus a crucial open question of whether the advanced architectures in machine learning are actually relevant for neuroscientists. In this chapter, I show that all these advanced architectures can be trained with one of their neural models: the proposed generalization of Z-IL is always equivalent to BP, with no extra restriction on the mapping function and the type of neural networks. Generally speaking, I have shown the similarity between BP and PC when it comes to learning. In the next chapters, I will explore some of the differences between these two algorithms.

Chapter 4

Associative Memory Capabilities

Through our life, we learn a huge number of associations between concepts: the taste of a particular food, the meaning of a gesture, or to stop when we see a red light. Every time we acquire new information of this kind, it gets stored in our long-term memory, situated in distributed networks of brain areas [102]. In particular, visual memories are stored in a hierarchical network of visual and associative areas [103]. These regions learn progressively more abstract representation of visual stimuli, so they participate in both perception and memory as each area memorizes relationships present in their inputs [104]. Accordingly, early visual areas learn common regularities present in the stimuli [20], while at the top of this hierarchy, associative areas (such as hippocampus, entorhinal cortex, and perirhinal cortex) store the relationships between extracted features, which encode an entire stimulus or episode [105]. The memory system of the brain is able to both recall complex memories [102, 6], and use them to generate predictions to guide behavior [106]. Learning in these associative memories shapes our understanding of the world around us and builds the foundations of human intelligence.

Due to their importance, computational models of associative memories have been developed for several decades now. They include autoassociative memories, which allow for storing data points and retrieving a stored data point s when provided with a noisy or partial variant of s, and heteroassociative memories, which are able to store and recall multi-modal data. An effective way to realize associative memory models is to store data points as attractors[1], so that they can be easily recovered via an energy minimization process when presenting their corrupted variants [13, 16, 15]. Classic associative memory models include Hopfield networks, and their modern formulation [107, 15]. Modern Hopfield networks (MHNs) are one-shot learners, able to store exponentially many memories in the number of parameters, and to perfectly retrieve them. However, the retrieval process often fails

[1] An attractor is a set of points toward which a physical system tends to evolve. Having training examples stored as attractors implies that iterations of a function acts as a retrieval mechanism.

when dealing with complex data, such as natural images. Recent works have shown that overparameterized autoencoders are excellent associative memories as well. Particularly, when training an autoencoders to generate a specific point s when s itself is presented as an input, it gets stored as an attractor [9].

In the previous chapter, I have studied the generalization potential of PC, showing that it can be as powerful as BP. This chapter, on the other hand, is dedicated to the study of the memorization capabilities of PC. More in detail, I show how PC can be used as a memory model that is able to store and retrieve memories with a high level of precision. The idea that PC may naturally be related to associative memories is inspired by recent works showing that the generative neural architecture that connects the hippocampus to the neocortex is based on an error-driven learning algorithm, which can be interpreted with a PC framework [6, 108]. In machine learning, associative memory models are simply data storage and retrieval systems, which store data points as attractors of a specific energy function, often following a thermodynamic formulation [109, 13, 110]. In more detail, the goal of an associative memory model is to return the stored data point that is *most similar* to a presented input, called *key*. However, while artificial and biological memory models are conceptually related, memory models in the literature do not resemble the hierarchical shape of the neocortical structure and the credit assignment described in Barron et al.[6]. The parameters of the generative model I will propose in this chapter, on the other hand, are updated until its predictions reach low/zero error in the sensory neurons. To conclude, popular models have a strict retrieval, which means they *always* retrieve a stored data point, no matter what the key is [15]. This is problematic in practical applications due to the impossibility to determine whether the retrieved memory is actually related to the key.

In this chapter, I show that the proposed memory model clearly outperforms popular models in the literature in retrieval robustness, is able to store and retrieve complex data points, such as ImageNet pictures, with a high level of precision, and does not perform strict retrievals. The contributions of this chapter are summarized as follows:

- I first define a basic model of AM that has a shape similar to Hopfield networks, is linear, and does not perform strict retrievals. The advantage is that it takes only a couple of seconds to train, however, it has low capacity, and requires a number of parameters that is quadratic to the input dimension. These two drawbacks make it not feasible for large-scale applications. However, I show that in medium-scale tasks, such as memorizing 100 pictures of the CIFAR10 dataset, this model performs quite well, outperforming Hopfield networks.

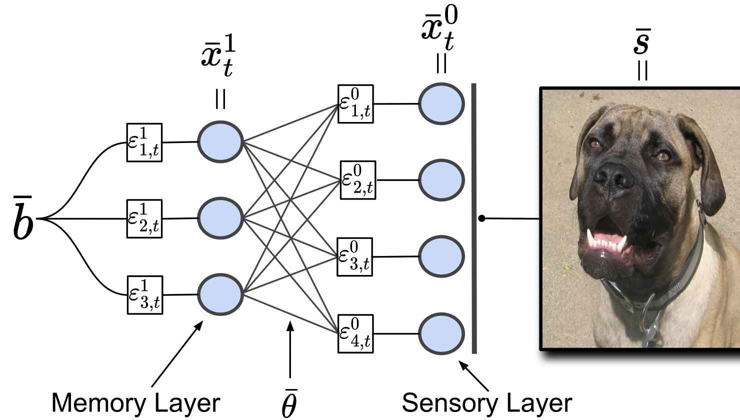


Figure 4.1: Generative PCN with 2 layers. Each line linking neurons denotes a pair of connections (excitatory in left direction, and inhibitory in right direction). At $t = 0$, the value nodes of the output layer are fixed to the vector representing the pixels of the figure. Then, during the learning phase, both weight parameters and value nodes are updated.

- To solve the above problems, I then define hierarchical generative PCNs, and empirically show that they store data points as attractors of their dynamics, that are restored from corrupted versions. In an extensive comparison of this memory model against standard autoencoders, I show the superior performances of generative PCNs in storage capacity, retrieval accuracy, and robustness. The proposed model naturally solves the task of reconstructing complex and colored images with a surprisingly high accuracy, outperforming autoencoders and MHNs by a large margin on Tiny ImageNet and CIFAR10. I then test the proposed model on ImageNet, perfectly reconstructing single pictures even after removing all but $1/8$ of the original image.

- I show that the proposed model is also able to handle multi-modal data, and hence perform hetero-associative memory experiments. Particularly, I train a model to memorize captioned images, where the captions are taken from a dictionary of 1000 words, and use the description to retrieve the original image and viceversa. Note that retrieving the images from the captions, implies retrieving 3072 pixels, using a vector of only 25 dimensions, which corresponds to less than 1% of the total information. Other memory models fail in performing this complex task.

4.1 Background on associative memories

In computer science, the concept of associative memory dates back to 1961, to the introduction of the *learn matrix* [111], a hardware implementation of hetero-associative memories

Algorithm 6 Learning to generate $\overline{s}$ with PC

Require: $\overline{x}_0$ is fixed to $\overline{s}$.
1: **for** $t = 0$ to T **do**
2: **for** each neuron i in each level l **do**
3: Update $x_{i,t}^l$ to minimize E_t.
4: Update each $\theta_{i,j}^{l+1}$ and b_i to minimize E_t

using ferromagnetic properties. However, the milestones of the field are Hopfield networks, presented in the early eighties in their discrete [13] and continuous [14] version. Recently, associative memory models have increasingly gained popularity. Particularly, a 2-layer version of Hopfield networks with polynomial activations has been shown to have a surprisingly high capacity [107]. This capacity can be increased even more when having exponential activations [112, 109, 15], and has been extended to the most used model in the literature, the continuous state Hopfield network [15], also called *modern* Hopfield network. To improve the retrieval process of stored memories, different energy-based models have been developed, such as [113]

The recent focus of the machine learning community on memory models, however, does not lie in applying them directly to solve practical hetero-associative memory tasks. On the contrary, it has been shown that modern architectures are implicitly associative memory models. Examples of these are standard deep neural networks [8, 9] and transformers [15]. Hence, there is a growing belief that understanding memory models will help the understanding of deep learning models [114, 8, 7, 10]. Furthermore, different classification models have been shown to perform well when augmented with AM capabilities. For example, *deep associative neural networks* [115], e.g., use deep belief networks to filter information, while [116] improves over standard convolutional models by adding an external memory that memorizes associations among images.

4.2 Basic model: A fully connected and linear graph

In this section, we consider a linear, fully connected graph, where every pair of neurons is connected via two directed connections. Let us assume we are training on a dataset where every data point has dimension d. Then, the basic model has exactly d neurons and $d(d-1)$ directed connections. Furthermore, here we do not have activation functions, making this model completely linear. This structure is similar to that of Hopfield networks, with the main difference being that here the weight connections are directed. A graphical representation of a linear and recurrent PC model with three neurons is represented in Fig. 4.2(a).

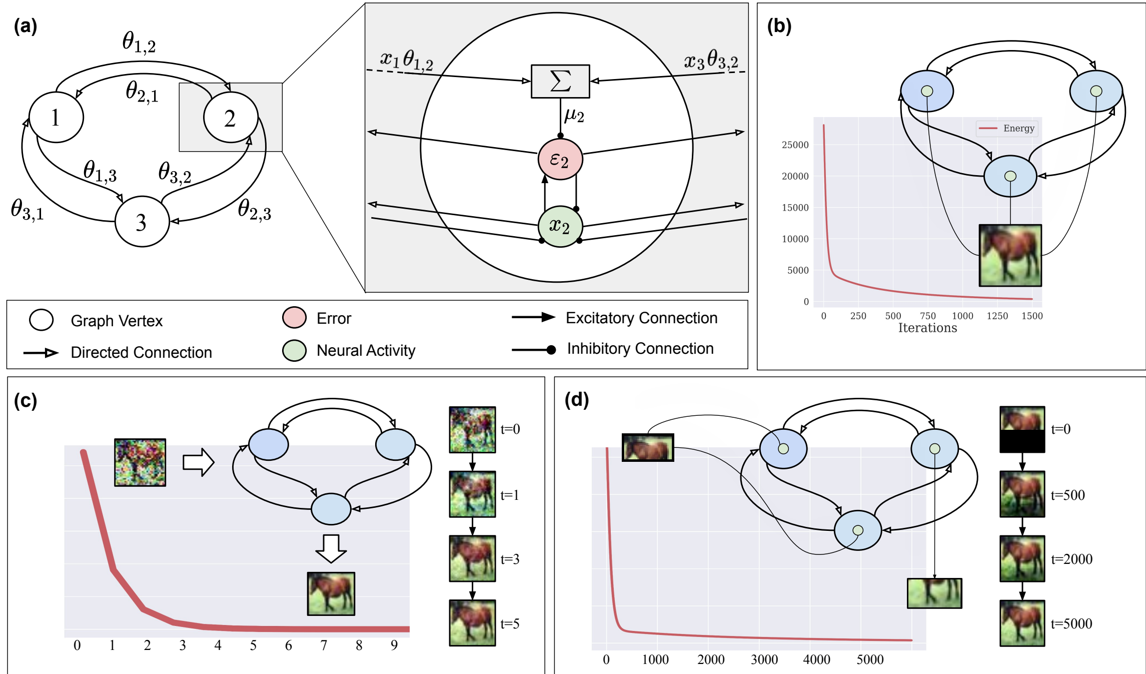


Figure 4.2: Memorization and Retrievals with the recurrent structure. (a) Neural implementation of a recurrent network with only three neurons, where each operation can be described via inhibitory and excitatory connections. (b) Memorization process of a recurrent network, where the value nodes are all fixed to the pixels of a specific image, and the synaptic strengths are updated via an energy minimization process. (c) Retrieval process of a recurrent network presented with a corrupted key. The neural activities of the network are initialized to the pixels of the corrupted image, and then updated via the same energy minimization process. (d) Retrieval process of a recurrent network presented with incomplete data.

As already stated in Section 2.1, the dynamics of this model is divided in two phases: training and prediction. In this chapter, we will refer to the learning phase as *memorization*, and to the prediction phase as *generation*. As the dimension of a data point is the same as the number of neurons, every neuron is a sensory neuron. Hence, during the memorization phase, all the value nodes are constrained. Because of this reason, only the weight parameters are updated during the synaptic connections, according to Eq. (5.4). More in detail, during the memorization phase, a pattern (e.g., an image) $\bar{s}$ is presented to the model, and the value nodes $\bar{x}$ are fixed to the entries of this pattern. Then, the weigths are updated via gradient descent to minimize the energy E of the network until convergence, as shown in Fig. 4.2(b), according to

$$\Delta\theta_{i,j,t} = -\alpha \cdot \frac{\partial E_t}{\partial \theta_{i,j,t}} = \alpha \cdot \varepsilon_{i,t} x_{j,t}. \tag{4.1}$$

When retrieving from corrupted memories, one iteration simply consists in presenting the

pixels of the corrupted memories $\tilde{s}$ and compute the predictions $\bar{\mu}$, as shown in Fig. 4.2(c). The situation changes when retrieving from partial memories. In this case, we only fix the value nodes of the available pixels, and run Eq. (4.1) on the unconstrained neural activities until the energy has converged, as shown in Fig. 4.2(d).

4.2.1 Experiments

To test the associative memory capabilities of this model, we now perform multiple experiments, comparing them against popular models in the literature. Particularly, we test strict retrievals against autoencoders and modern Hopfield networks, and retrieval robustness against modern and classic Hopfield networks.

Set-up: We have trained a linear PC model on 100 images of a subset of 100 images of CIFAR10 and SVHN, and on 10 binarized images of MNIST and FashionMNIST. In all the experiments, we have used a learning rate of 0.001. In all cases, the energy had converged after 1500 iterations, a process that takes approximately 11 seconds. Furthermore, we have trained modern and classic Hopfield networks on the same tasks, as well as an autoencoder with 6 hidden layers and hidden dimension of 512 on CIFAR10. Then, we have have tested the trained models on different tasks:

1. **Strict retrievals**: we have presented randomly generated keys to the trained model, and checked the retrieved memory. This experiment is informative, as it allows to test whether our model returns an existing memory even when presented with an image which is pure noise.

2. **Retrieval quality**: We have then used the same two models to retrieve original memories by using both corrupted and incomplete keys. In the first case, we have corrupted the original memories using Gaussian noise of mean zero and variance $\eta = 0.2$. In the second case, we have presented the network a key consisting of only $1/2, 1/4$, and $1/8$ of the original pixels

3. **Classic Hopfield networks**: To compare against classic Hopfield networks, we have trained on 10 images of the MNIST and FashionMNIST datasets using the same hyperparameters, converted to binary digits as the original formulation only deals with binary data.

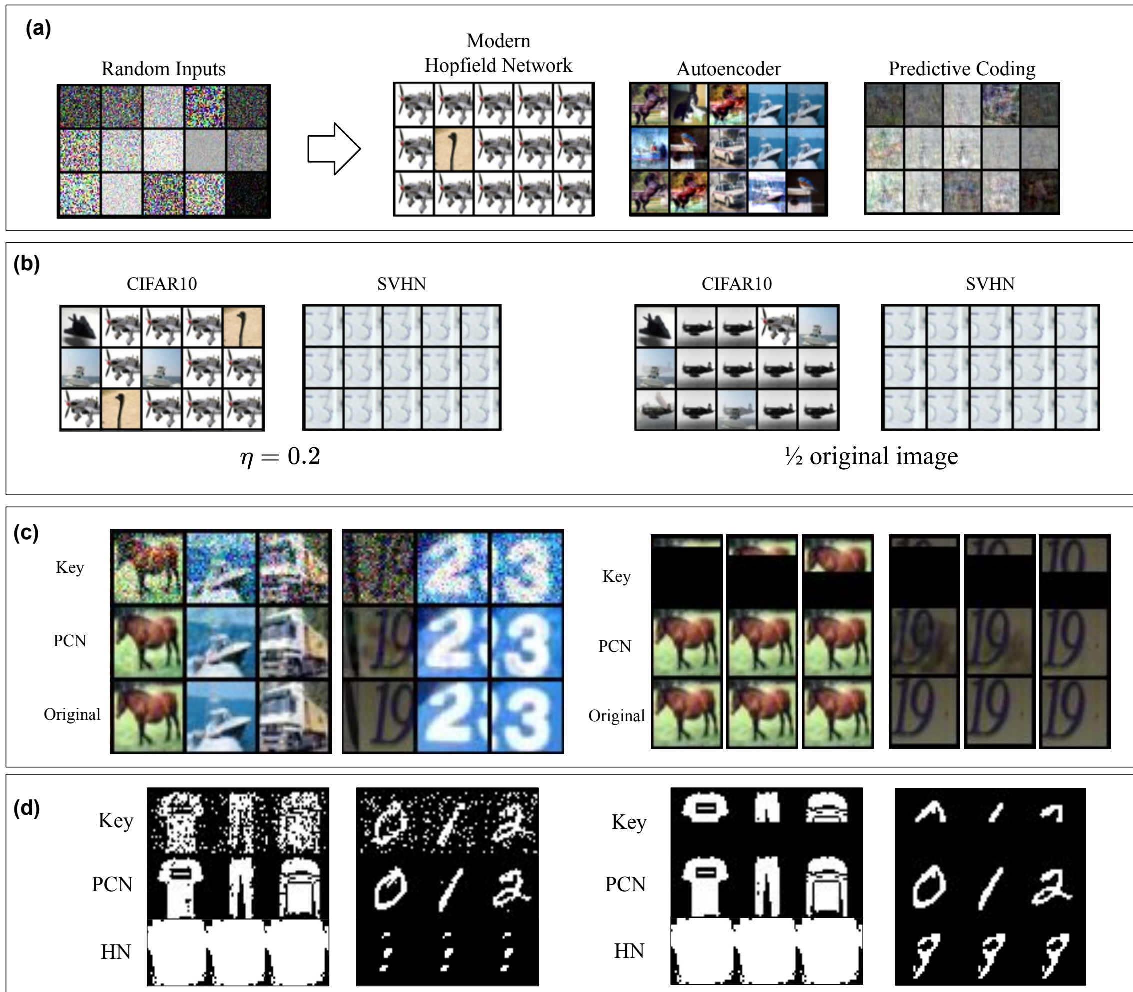
Results: To provide a numerical evaluation, an image is considered recovered when the error between the original image and the recovered image is less than 0.005. Fig. 4.3(a) shows that our model always returns random noise when presented with random noise, while both Hopfield networks and autoencoders fail to satisfy this property. Hence, we conclude that our model does not perform strict retrievals. In the second experiment, the model has correctly retrieved all the images when confronted with keys corrupted using Gaussian noise. In the case of incomplete keys, we have presented the model only $1/2, 1/4,$ and $1/8$ of the original pixels. In this case, the model has successfully recovered, respectively, 89, 79, 49 of the original memories in the experiments using CIFAR10, and 90, 84, and 81 using SVHN. A graphical representation of correctly retrieved memories can be found in Fig. 4.3(c). In contrast, modern Hopfield networks failed in recovering more than 5 images when presented with incomplete keys, and 11 images when presented with corrupted keys. To conclude, in the experiment using classic Hopfield networks, our model is always able to retrieve the original memories, while Hopfield networks fail in correctly storing them, and hence always returns the same noisy output.

All in all, these experiments show that this simple model is able to quickly store memories, to perform close to one-shot retrieval, to outperform popular models when it comes to retrieval robustness, and that it is reliable, as it does not perform strict retrievals, even when presented with randomly-generated keys. However, there are still some drawbacks: its capacity is limited, as it fails to correctly store and retrieve more than 200 data points, and the number of parameters is quadratic to the dimension of the inputs, which does not allow it to work on large data points, such as high quality images. We now show how a multi-layer structure automatically solves the aforementioned drawbacks, and even allows for a more accurate retrieval of original memories.

4.3 Hierarchical model

Let $\bar{s}$ be a training data point, and M be a hierarchical PCN considered above, already trained until convergence to generate the single memory $\bar{s}$. In this case, the energy function E_t has a global minimum where the value nodes of the sensory layer are equal to the entries of $\bar{s}$. This means that, $\bar{s}$ is an *attractor* of the dynamics of the energy: when providing the network a configuration of the value nodes that is not a local minimum, but close to it, inference will update the value nodes until the total energy reaches the minimum. More in detail, a point $\bar{x}$ is an attractor of the dynamics of a specific function if there exist a neighbor of $\bar{x}$ such that multiple iterations of the function on any point of the neighbor converge to $\bar{x}$.

Figure 4.3: Experiments using recurrent linear networks



(a) Retrieval of corrupted memories when presented a randomly generated key. (b) Outputs of an Hopfield network trained on 100 data points when presented with 15 different keys, each generated from a corrupted, different picture. We can observe the strict retrieval problem: when it fails to recognize a key, it simply outputs a memory stored as a strong attractor. (c) Retrieval from corrupted (left) and incomplete (right) keys, with different levels of noise or missing pixels. (d) Same task performed on binary pictures taken from the FashionMNIST (left) and MNIST (right) datasets, and compared against the standard formulation of Hopfield networks.

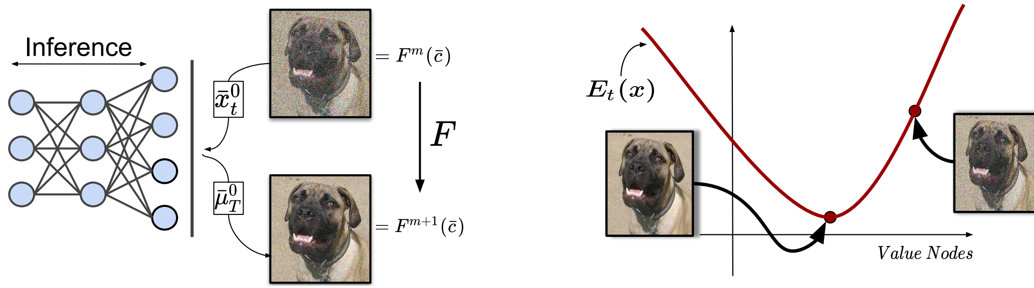


Figure 4.4: Left: representation of the function F, used to retrieve stored images. It can be decomposed into three steps: (1) The value nodes of the sensory layer are fixed to the pixels of the corrupted image $F^m(\bar{c})$. (2) Inference runs for T operations (until convergence). (3) We set $F^{m+1}(\bar{c})$ to the prediction of the sensory layer $\bar{\mu}_T^0$. Note that the weight parameters are never updated during the above steps. We have omitted the error nodes for simplicity. Right: representation of an AM, where an image $\bar{s}$ (photo of a dog) is stored as an attractor of the dynamics. A corrupted image that lies in a specific neighborhood of $\bar{s}$ converges to it when minimizing the total energy via running inference.

While giving multiple memories does not allow to reach exactly zero error, we empirically show that all the training points are stored in the above way as attractors.

To retrieve stored data points $\bar{s}$ given a corrupted version $\bar{c} \in \mathbb{R}^d$ of $\bar{s}$, we have to proceed as follows: first, we set the value nodes of the sensory layer to the corrupted points, i.e., $\bar{x}_t^0 = \bar{c}$ for the whole process. Then, we run inference until convergence and save the prediction $\bar{\mu}_T^0$ of the sensory layer. If the original data point was stored as an attractor, we expect the prediction $\bar{\mu}_T^0$ to be a less corrupted version of it. Let $F\colon \mathbb{R}^d \to \mathbb{R}^d$ be the function that sends $\bar{c}$ to $\bar{\mu}_T^0$ just described, and summarized in Fig. 4.4. We will show empirically that many iterations of this function allow to retrieve the stored data point. All in all, training points are stored in the memory vector $\bar{b}_T$ and the weight parameters, and what the algorithm does to retrieve them is simply the inference phase of PCNs. Since visual memories are stored in hierarchical networks of brain areas, the proposed model is a plausible algorithm to better understand how memory and prediction work in the brain.

To experimentally show that generative PCNs are AMs, we trained a 2-layer network with ReLU non-linearity on a subset of 100 images of the street view house number dataset (SVHN), Tiny ImageNet, and CIFAR10. After training, we presented the model with a corrupted variant (by adding Gaussian noise) of the training set. We then used the PCNs to reconstruct the original images from the corrupted ones. The experimental results confirm that the model is able to retrieve the original image, given a corrupted one. The obtained reconstructions for the Tiny Imagenet dataset (the most complex one, as each data point consists of $3 \times 64 \times 64$ pixels) are shown in Fig. 4.6.

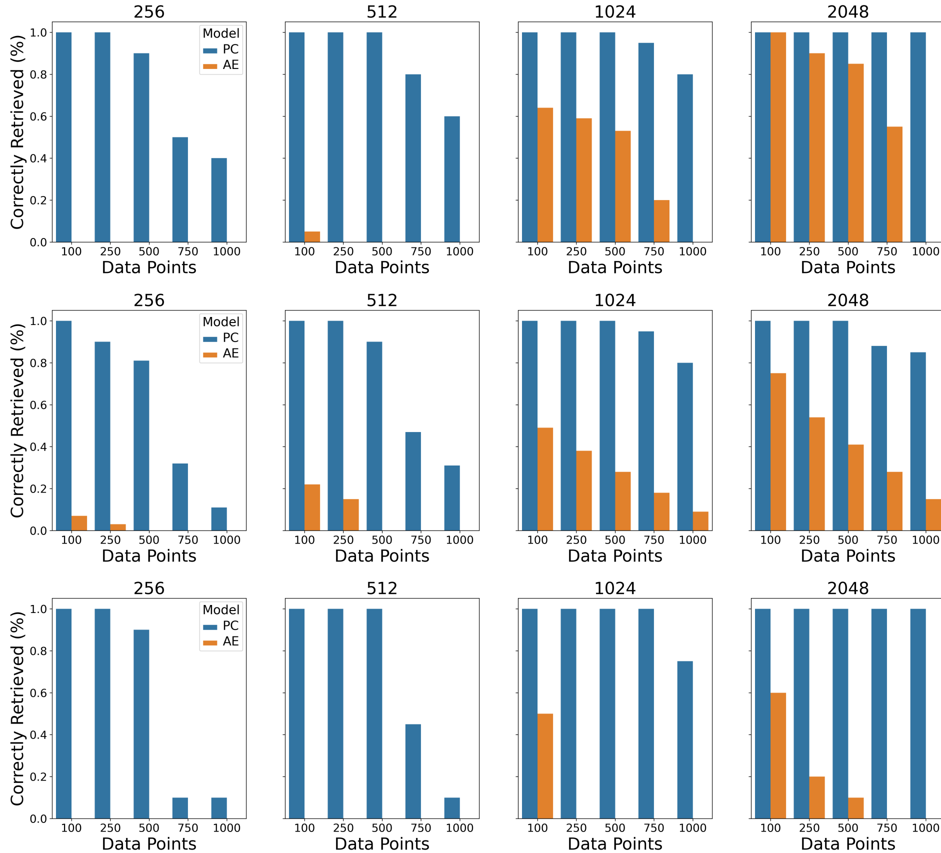


Figure 4.5: Percentage of correctly retrieved images by 2-layer generative PCNs (PC) and 3-layer autoencoders with hidden-layer dimensions of 256, 512, 1024, and 2048, when presented with a corrupted image with Gaussian noise of variance 0.2. Top row corresponds to experiments performed on CIFAR10, middle row the SVHN dataset, bottom row to Tiny Imagenet.

I now provide a more comprehensive analysis, which studies the capacity of generative PCNs when changing the number of data points and parameters.

Experiments: We trained 2-layer PCNs with ReLU non-linearity and hidden dimension $n \in \{256, 512, 1024, 2048\}$ on subsets of the aforementioned datasets of cardinality $N = \{100, 250, 500, 1000, 1500\}$. Every model is trained until convergence, and all the images are retrieved as explained in Section 4.3. Again, to provide a numerical evaluation, an image is considered recovered when the error between the original image and the recovered image is less than 0.005.

To compare our results against a standard baseline, we also trained 3-layer autoencoders with the same hidden dimension on the same task, and compared the results. Note that the number of parameters of a 2-layer PCN is smaller than the one of a 3-layer autoencoder with

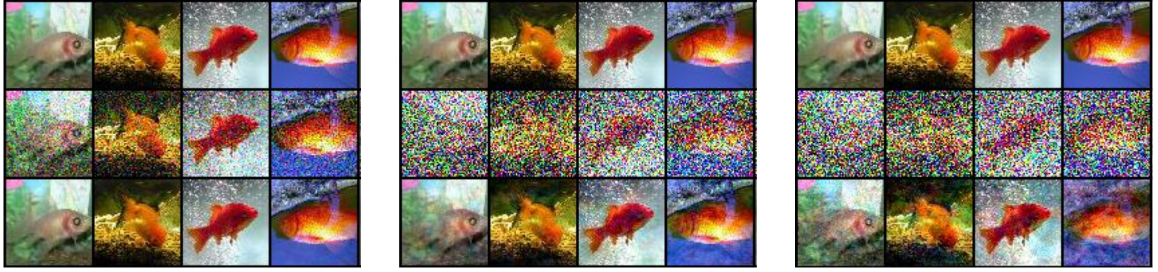


Figure 4.6: Reconstructions of 64×64 Tiny ImageNet images with noise levels 0.2, 0.7, and 0.9, respectively (original, noisy, and reconstructed image in first, second, and third row, respectively).

the same hidden dimension, as the additional layer (input layer, not needed in generative PCNs) almost doubles the number of parameters in some cases.

Results: The analysis shows that the proposed model is able to store and retrieve data points even when the network is not overparameterized (see Fig. 4.5). Autoencoders trained with BP did not perform well: autoencoders with less than 1024 hidden neurons always failed to restore even a single data point on the Tiny Imagenet dataset, and very few on other ones. The performance of autoencoders with 2048 hidden neurons were always worse than PCNs with 256 hidden neurons. This shows that overparameterization is essential for autoencoders to perform AM tasks, and that our proposed method offers a much more network-efficient alternative.

In terms of capacity, we show that 2-layer PCNs with 256 hidden neurons are able to correctly store datasets of 250 images of both CIFAR10 and Tiny ImageNet, and that networks with 2048 hidden neurons always managed to store and retrieve all the presented datasets. As typical in the AM domain, small models trained on large datasets fail to store data points, as the space of parameters is not large enough to store each data point as an independent attractor, and many attractors in the same small region lead to a chaotic dynamic. The proposed model is no different: PCNs with 256 and 512 hidden units are able to store almost 500 Tiny ImageNet images when trained on datasets of that size, but fail to store more than 200 when trained on larger ones (1000 and 1500).

4.3.1 Analysis of different levels of noise

So far, we analyzed images with Gaussian noise of variance 0.2. We now extend the analysis to different levels of corruptions. Particularly, we have used networks of width $n \in \{512, 1024, 2048\}$ trained on $N \in \{100, 250, 500\}$ natural images taken from the

Tiny ImageNet dataset, and tried to retrieve them using Gaussian noise of variance $\eta \in \{0.3, 0.5, 0.7, 1.0\}$.

Results: The results show that our method is able to retrieve stored data points even with a larger amount of variance, although the results get worse the more we increase the level of noise. Particularly, when presented with data points with Gaussian noise with variance 0.3, every network was able to restore at least one image, even when trained on 500 examples. The numbers of retrieved images decreased when increasing the noise. When presented with noise with variance $\eta = 1.0$, only networks with 2048 hidden units were able to retrieve a tiny fraction of the original data points. We report the results in Fig. 4.7

4.3.2 Plots of images with extreme levels of noise

We now plot the images generated using different levels of corruptions. Note that all the images shown here were clearly classified as *not* retrieved by the above analysis. However, we believe that an analysis of not well-retrieved images is still interesting, to understand the limitations of our method. Hence, we trained a generative PCN on 100 images of the Tiny ImageNet dataset, and tried to reconstruct them using extreme levels of noise. Then, we printed the reconstructions.

Results: These representations show that generative PCNs are able to identify the original data points, even if the retrieval process was not accurate enough for the error to fall below the decided threshold 0.005. Particularly, the proposed model was able to identify original memories even when presented with extreme levels of noise, although leaving visible amount of corruption. The reconstructions are given in Fig. 4.8.

4.3.3 Analysis of the retrieval function

Here, we show visually how the function F restores corrupted images. Particularly, we trained a generative PCN with 2048 hidden neurons on 100 images of the Tiny ImageNet dataset, and printed the reconstructions after 1, 6, and 11 iterations of the retrieval function F.

Results: The reconstructions show that the first iteration is able to clear most of the noise. However, further iterations are needed to clear the remaining details, especially if the noise level is high. In fact, it can be observed that one iteration of F is able to retrieve the original data point when the level of corruption is $\eta = 0.3$, but fails when $\eta = 1.0$. This process

usually converges around m = 15, depending on the number of iterations T and the level of noise. The results are shown in Fig. 4.9.

4.3.4 Analysis of the retrieval time

In the AM literature, it is important to quickly retrieve an image. In fact, modern Hopfield networks and similar models are able to retrieve memories 1-shot. The proposed model is unable to achieve this, as the retrieval process relies on an energy minimization framework, however, it is able to retrieve a memory in a couple of seconds. Note that this does not alter its biological plausibility, mainly because of two reasons:

- The brain performs computations much more efficiently, and hence the same process could take milliseconds. This is especially the case when these computations are local and performed using an energy minimization framework. The neural information processing is, in fact, certainly not based on digital GPU/CPU hardware and may realize the presented algorithm in a much more natural (and efficient) way, such as using analogue relaxation dynamics to rapidly converge to the attractor state.
- The retrieval process takes some time, because it retrieves exact memories at the pixel level, while memory recall is not as detailed. If the inference process is stopped after a shorter period of time, it simply recalls a slightly fuzzier memory. This flexibility of variable-length computation is also highly desirable and realistic from a biological viewpoint.

The time in seconds needed to retrieve original images is given in Table 4.1. Note that these numbers provide an upper bound on the real efficiency of the model, limited by the current implementation. In fact, while every computation during the retrieval process can theoretically run in parallel, this cannot be implemented in popular deep learning frameworks such as PyTorch and Tensorflow. A dedicated implementation, which would require significant engineering work, could make this process up to an order of magnitude faster. All experiments are conducted on two Nvidia GeForce GTX 1080Ti GPUs and eight Intel Core i7 CPUs, with 32 GB RAM.

4.4 Retrieval from partial data points

So far, we have shown how the proposed generative model can be used to retrieve stored data points when presented with corrupted variants. We now tackle the different task of retrieving data points when presented with partial ones. Let $\bar{s}$ be the stored data point, and

Table 4.1: Time (in seconds) to perform a retrieval from incomplete images on PCN with two hidden layers.

Hidden Dimension	256	512	1024	2048
One Iteration	0.02	0.04	0.09	0.15
Five Iterations	0.10	0.19	0.53	0.84
Ten Iterations	0.55	1.18	2.85	4.37

assume that a fraction of pixels of $\bar{s}$ are accessible, and the goal is to retrieve the remaining ones. Let $\bar{s}'$ be the vector of the same dimension of $\bar{s}$, where a fraction of pixels are equal to the ones of the stored data point, and assume that the position of the pixels that are equal to the ones of $\bar{s}$ is known. We now show how to retrieve the complete data point by using the same network M, trained as already shown in Section 4.3.

Let M be a PCN trained to generate $\bar{s}$. Then, given the partial version of the original data point $\bar{s}'$, it is possible to retrieve the full data point using M as follows: first, we only fix the value nodes of the sensory layer $\bar{x}_t^0$ to the entries of the partial data point $\bar{s}'$ that we know are equal to the ones of the stored data point, leaving the rest free to be updated. Then, we run inference on the whole network until convergence. At this point, we expect the value nodes $\bar{x}_T^0$ to have converged to the entries of the original data point, stored as an attractor. A graphical representation of the above mechanism is described in Fig. 4.4. To show the capabilities of this network, we have performed multiple experiments on the Tiny ImageNet and ImageNet datasets, and compared against existing models in the literature. We now start by providing visual evidence on the effectiveness of this method by masking parts of the images. Note that the geometry of the mask does not influence the final performance, as the proposed model simply memorizes single pixels.

Experiments: We trained two networks with hidden dimensions of 1024 and 2048, to generate 50 images of the first class of Tiny ImageNet (corresponding to goldfishes), and a network of 8192 hidden neurons to reconstruct 25 pictures taken form ImageNet. Then, we used inference as explained to retrieve the original images. We considered an image to be correctly reconstructed when the error between the original and the retrieved image was smaller than 0.001. Furthermore, we plotted the partial images together with their reconstructions, for a visual check. Note that we have used the thresholds that provided the fairest comparison: the denoising experiments fail to have a perfect retrieval, despite the fact that most of the images look visually good. Hence, we have determined the threshold to be equal to 0.005. Then, with the same threshold for the retrieval of partial images, our method

Algorithm 7 Retrieving $\bar{s}$ given a non corrupted fraction $\bar{s}'$

Require: if s'_i a correct entry of the original memory, then $x^0_{i,t}$ is fixed to s'_i.

1: **for** $t = 1$ to T **do**
2:     **for** each neuron i in each level l **do**
3:         Update $x^l_{i,t}$ to minimize E_t.
4: **return** $\bar{x}^0_T$

always successfully retrieved all the images, and so we have opted for a smaller threshold, which was more informative.

[Associative memories on incomplete data]Algorithm used to retrieve a stored data point from a fraction of it: (1) The value nodes of the sensory layer that correspond to the available entries of $\bar{s}$ are fixed to the respective values. (2) Inference runs for T operations (until convergence). (3) We set our prediction to be equal to the value nodes of the sensory layer $\bar{x}^0_T$. The weight parameters are never updated during the above steps. The error nodes are omitted for simplicity.

Results: On Tiny ImageNet, a generative PCN with 1024 hidden neurons managed to reconstruct all the images when presented with $1/8$ of the original image, and more than half for the smallest fraction considered, $1/16$. The network with 2048 neurons also failed to reconstruct all the images when presented with $1/16$ of the original image. However, even when presented with a portion as small as $1/16$ of the original image, the reconstruction was clear, although not perfect and hence now above our threshold. This result is shown in Fig. 4.5 (left).

Surprisingly, generative PCNs trained on ImageNet correctly stored all the presented training images, and correctly reconstructed them with no visible error. This shows that PCNs can be used to store high-dimensional and high-quality images in practical setups, which can be retrieved using only a low-dimensional key, formed by a fraction of the original image. Particularly, Fig. 4.5 (right) shows the perfect reconstruction obtained on ImageNet when the network is presented with only $1/8$ of the original pixels.

4.5 More training data and/or deeper networks

[Qualitative results on incomplete data]Partial images and their reconstructions (from top to bottom: partial image, reconstructed image, original image, and reconstruction error (difference between original and reconstruction)). Left: 64×64 Tiny ImageNet images presented with $1/4$, $1/8$, and $1/16$ of the original image, respectively, using a network of 1024 hidden neurons. Right: 224×224 ImageNet images presented with $1/8$ of the original image, using a network with 8196 hidden neurons.

In the above experiments, a shallow architecture is able to store 100 natural images from Tiny ImageNet, and to reconstruct them with no visible error when provided with $1/8$ of the original images. We now study how this changes when either the cardinality of the dataset is increased, or the number of provided pixels during reconstruction is updated. To study this, we have trained two models with $n = \{1024, 2048\}$ hidden units on subsets of Tiny ImageNet of cardinality $N = \{50, 100, 250, 500\}$, and reconstructed images with only $p = \{1/2, 1/4, 1/8, 1/16\}$ of the original pixels. According to our visualization experiments, we have noted that an image has no visible reconstruction noise when the error between the original image and the reconstruction is less than 0.001. Hence, we consider an image to be correctly retrieved if the reconstruction error is below this threshold.

Results: Every network with 50 stored memories was able to perfectly reconstruct the original image when provided with $1/4$ of the original image, and about half when provided with only $1/8$. However, this changes when increasing the training set, as no network trained on more than 50 images was able to correctly reconstruct an image when provided with $1/16$ of the original images, besides a few cases. In terms of capacity, the more images we train on, the harder the reconstruction becomes: no training image was correctly retrieved when training on $N = 500$ images when provided with a fraction smaller than $1/2$ of the original image, regardless of the width of the network. A summary of these results is given in Fig. 4.16.

These experiments show the limits of shallow generative PCNs on both reconstruction capabilities and capacity. As expected and common in standard AM experiments, increasing the number of training points and reducing the number of available information for reconstruction hurts the performance. We now show how increasing the number of layers solves this capacity problem.

4.5.1 Deep generative predictive coding networks

We now test how increasing the depth of generative PCNs increases their capacity. We then compare the results against overparameterized autoencoders. Particularly, we trained generative PCNs with $n = \{1024,\ 2048\}$ hidden neurons, depth $L = \{3, 5, 7, 10\}$ on $N = 500$ images of Tiny ImageNet. Furthermore, we reconstructed the stored images using a fraction of $p = \{1/2, 1/4\}$ of the original pixels. To further compare against state-of-the-art autoencoders, we trained equivalent autoencoders, which is the best-performing one according to [9]. Also, all the hyperparameters used for training are the ones reported by the authors. To make the comparison completely fair, we also assume that the correct pixels of the incomplete images are known when testing the autoencoder. Particularly, we fixed these

pixels at every iteration of the reconstruction process of the autoencoders. As above, we consider an image to be correctly retrieved if the reconstruction error is less than 0.001.

Results: The results confirm the hypothesis: the deeper the network, the higher the capacity. Particularly, a 10-layer network was able to reconstruct more than 98% (against the 72% of the autoencoder) of the images when providing half of the original pixels, and 74% (against the 48% of the autoencoder) of the images when providing $1/4$ of the pixels. This shows that the proposed model clearly outperforms state-of-the-art autoencoders in image reconstruction, even in the overparameterized regime. Fig. 4.17 summarizes these results.

4.6 Comparison with deep associative neural networks

As shown, the original formulation of MHNs does not allow to perform image reconstruction experiments. In fact, while Hopfield networks are the most famous AM architecture, it fails to perform accurate, pixel level reconstructions, as shown in Fig. 4.18. To solve this problem, a new model has been presented, which consists of a MHN augmented with a convolutional multilayer structure that performs unsupervised feature detection [115] using a deep belief network. The resulting architecture, called *deep associative neural network* (DANN), is not a pure associative memory model, as it is able to perform both AM and classification tasks. We now compare the image reconstruction experiments performed by the authors, with the ones performed by our generative PCN. We replicate the experiment proposed, which consists in presenting the network a partial image of the CIFAR10 dataset, where a squared patch covers the centre of each image.

Note that the comparison we propose is purely qualitative, as DANNs auguments MHNs with an unsupervised feature detection, using convolutions to further improve their results, and are both memorizing the images and learning the main feature patterns. This allows them to have a large capacity, which allows DANNs to be trained on large datasets. Hence, comparing it against a pure memory model such as ours may not be indicative. However, we believe that presenting comparisons against popular models that perform AM tasks such as DANNs is still interesting in understanding how the proposed model compares against existing ones in the literature.

Results: We first ran the same experiment on CIFAR10 as in [115]. Particularly, we used 500 images. The comparison is given in Fig. 4.19. Unlike the image in [115], our reconstruction shows no visible error. The results show that the reconstruction of our

memory model are much clearer than the ones in [115], and so consistently improving over this particular task. Furthermore, we replicated the experiment using 50 images of ImageNet, and showed that the proposed model is able to provide perfect reconstructions of the original images. However, DANNs show a larger capacity, as the provided images are obtained after training on the whole CIFAR10 dataset.

4.7 Hetero-associative memory experiments

Generally speaking, associative memories come in two high-level forms: auto-associative and hetero-associative memories [117, 13, 14, 15]. While both are able to retrieve memories given a set of inputs, auto-associative memories are primarily focused on recalling a specific pattern when provided a partial or noisy variant of X [13, 14, 15]. So far, we have only performed this kind of task. By contrast, hetero-associative memories are able to recall not only patterns of different sizes from their inputs, but are also able to memorize multi-modal information, such as associate natural language with sounds or images. In this section, we study the performance of this model with respect to multi-modal dataset, by training it on images and their textual descriptions, and then using the description to retrieve the related image.

Experiment: The dataset used for this task is `flickr_30k` [118], which consists of colored captioned images of varying size, that we resize and reduce to 32×32 colored images. Each image has 5 associated captions, from which only use the first one was picked in this experiment. The captions are tokenized by first building a vocabulary (using the package `spacy`) and then discretized using said vocabulary. Afterwards, they are transformed into floating point numbers and normalized to $[0, 1]$, to remain in the same range as the images. Finally, the captions are padded with 0s to the size of the longest caption, to maintain the same data size. For example, the captions associated with the first three images are:

1. Two young guys with shaggy hair look at their hands while hanging out in the yard;
2. Several men in hard hats are operating a giant pulley system;
3. A child in a pink dress is climbing up a set of stairs in an entry way.

Note that the task of retrieving images from captions is complex, as every pixel of an image image of size 3072 has to be retrieved using a key of dimension 25 (hence, using $< 1\%$ of the original information).

To show how depth influences the retrieval quality, we have trained different PCNs with depth $L \in \{1, 3, 5, 7, 9, 11\}$ and width $n \in \{512, 1024, 2048\}$ to memorize 25, 50, and 100 captioned images. Then, the networks are given the images without to recover the captions and vice-versa. The number of recovered captions and recovered images are then recorded. An image is considered to be recovered if its error is less than 0.001, while a caption is considered to be recovered if it is *exactly* recovered: unlike pixels, words are embedded as discrete information, and hence a word is either exactly retrieved, or mistaken by a wrong word of the dictionary. A caption is considered to be correctly recovered if every each word is correctly recovered.

Results: The number of images and captions recovered are given in Fig. 4.20. We found that increasing the number of neurons in each layer did not affect the results much, then only the best result from each depth is shown. The results indicate that increasing the number of layers in the network increases the capacity of the network, as expected. Most of the networks successfully recovered most of 25 images from the respective captions, which means that from $\simeq 1\%$ of the original data the networks remembered the remaining 99% and more than half of the images when trained on 50 data points. When trained on 100 data points, only deep networks (9 and 11 hidden layers) were able to perfectly retrieve more than half of the data points. The shallow networks of depth 1 performed poorly on all the tasks, not recovering any image or caption. Compared to autoencoders, PCNs perform slightly worse when retrieving captions, but significantly better when retrieving pictures, as autoencoders were never able to retrieve more than two, regardless of the parameterization or the number of data points used.

The results show that generative PCNs are able to both handle multimodal data, and retrieve stored memories using less than 1% of the original information. Furthermore, an analysis of the retrieved images of the autoencoders, show that they are always able to correctly retrieve images of the dataset, even when it does not correspond to the one we were looking for. This shows that autoencoders store images as attractors, but some attractors are stronger than others, and hence most of the retrieval processes converge to the unrelated memories. This result is visually shown in Fig. 4.21, which represents the recovery of 25 images by both PCNs and autoencoders.

4.8 Full-page reconstructions of ImageNet

To show the reconstruction capabilities of our method, in Figures 4.22 and 4.23 we plot full-page reconstructions of ImageNet pictures.

4.9 Implications to neuroscience

The proposed architecture closely mimics the behavior of the hippocampus as a memory index and generative model. As a generative model, the hippocampus accumulates prediction errors from sensory neocortical neurons lower in the hierarchy, and sends descending predictions to the neocortex, to correct the prediction errors in the neocortical neurons. During this phase, the generative model of the hippocampus is updated until the hippocampal predictions correct the prediction errors by suppressing neo-cortical activity [6]. The training phase of our generative model closely resembles this framework, as it also accumulates prediction errors from the sensory neurons to the hidden layers and memory vector, while a continuous update of the value nodes corrects the errors in the sensory neurons. Then, the parameters of our generative model are updated until its predictions reach low/zero error in the sensory neurons.

The similarities between the proposed model and the hierarchical structure connected to the hippocampus are not restricted to the training phase, but also to the reconstructions of past memories: as memory index, the hippocampus sends descending input to the neocortical neurons to reinstate activity patterns that recapitulate previous sensory experience. In the reconstruction phase of our generative model, the memory vector provides descending inputs to the sensory neurons, to generate patterns that recall previous sensory experience, e.g., stored data points.

From a machine learning perspective, this method is able to store and retrieve high-quality images, such as ImageNet figures, while provided with only a tiny fraction of the initial information. However, this model could also have an impact in the neuroscience community, as it provides a working computational model of the ”learn and recall” phases of the hierarchical structure that connects the hippocampus and the neocortex.

4.10 Summary and discussion

The results presented in this chapter could be interpreted in multiple ways, and hence influence different directions of research. As a brief summary, this chapter shows that predictive coding naturally implements associative memories that (i) have a high retrieval accuracy and robustness, (ii) can be expressed using small and simple fully connected neural architectures, and (iii) do not perform strict retrieval. This has been extensively shown in a large number of experiments, for different architectures and datasets. Particularly, I have performed denoising and image reconstruction tasks, comparing against popular AM frameworks. Furthermore, I have shown that this model is able to reconstruct with no visible

error natural high-resolution images of the ImageNet dataset, when provided with a small fraction of them. From the neuroscience perspective, I have shown that the proposed model closely resembles the ‘learn and recall’ phases of the hippocampus.

The first possible impact this work can have is about the future of associative memory models. Particularly, this model is slower than existing ones, and needs to be trained, but it is able to perform much better retrievals. Second, it makes use of an important feature of PC, which is not replaceable by standard networks trained with BP: the bidirectionality of the information flow. In fact, we have shown that, by simply fixing a fraction of the value nodes on a layer, the output layer in this case, it is possible to influence the other value nodes on the same layer. This is due to the error signal that is propagated up from the sensory layer, is used by the internal value nodes to update their neural activities, that then propagate their signal back to the sensory layer. Simply put, this chapter shows how precise this information flow can be when applied to quite complex memory tasks. To conclude, this chapter shows that it is possible to strengthen the connection between the machine learning and the neuroscience community, as it underlines the importance of predictive coding in both areas, both as a highly plausible algorithm to better understand how memory and prediction work in the brain, and as an approach to solve corresponding problems in machine learning and artificial intelligence.

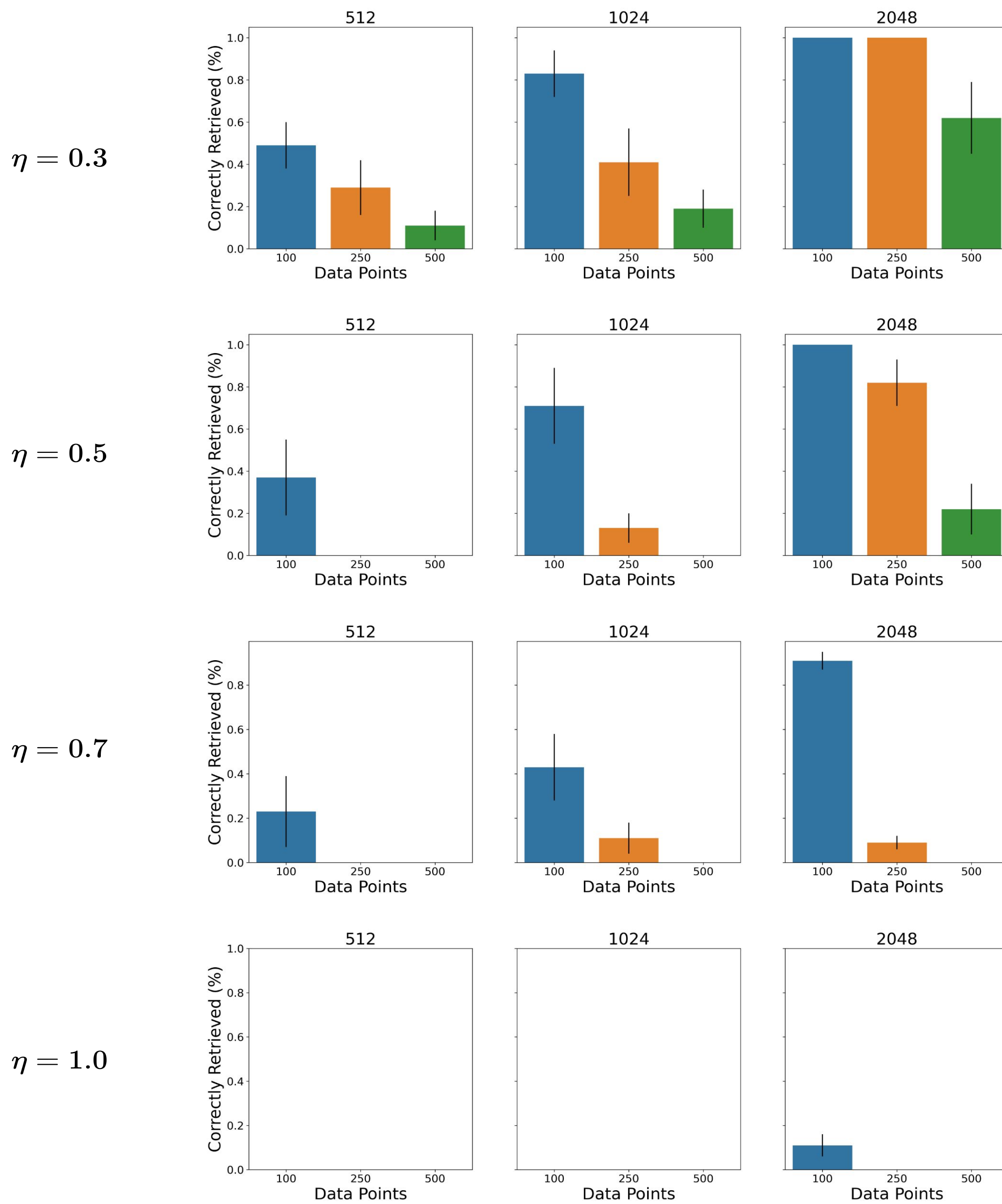


Figure 4.7: Percentage of correctly retrieved images by 2-layer generative PCNs (PC) with hidden-layer dimensions of 512, 1024, and 2048, when presented with a corrupted image with Gaussian noise of different variances, trained on the Tiny ImageNet dataset.

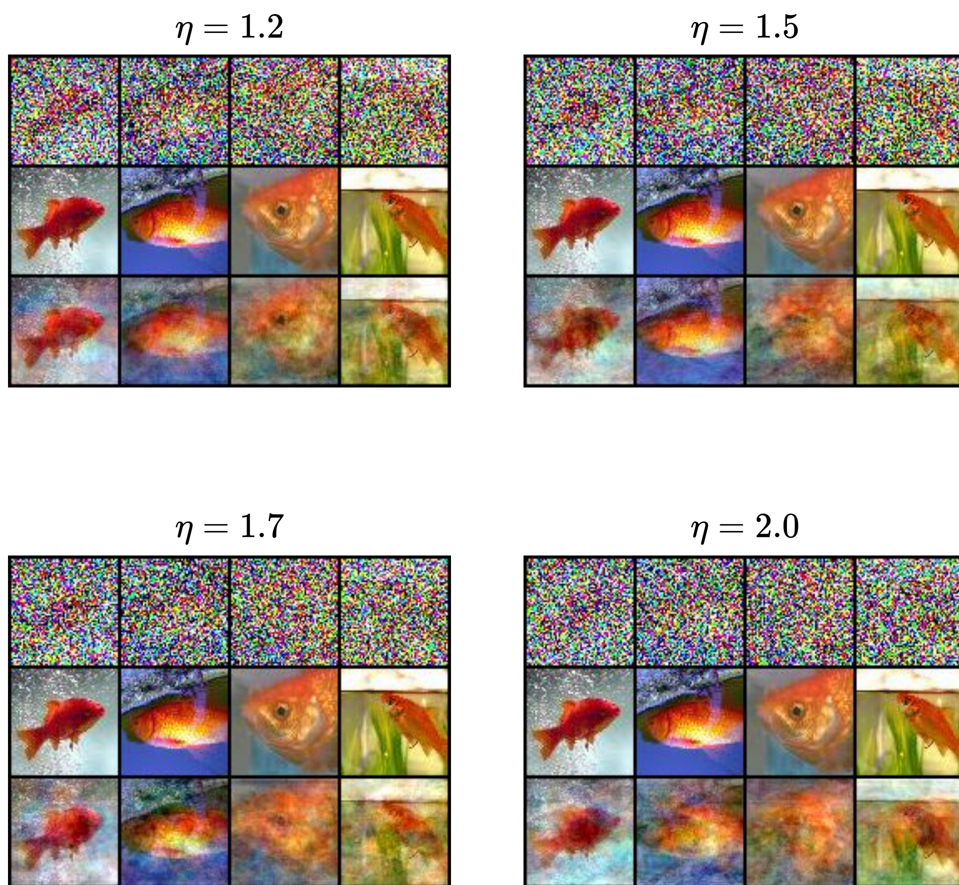


Figure 4.8: Analysis of the reconstruction capabilities of generative PCNs under different levels of noise $\eta \in \{1.2, 1.5, 1.7, 2.0\}$. Every plot was obtained using a 2-layer generative PCN with 2048 hidden units, trained on 100 images of the Tiny ImageNet dataset.

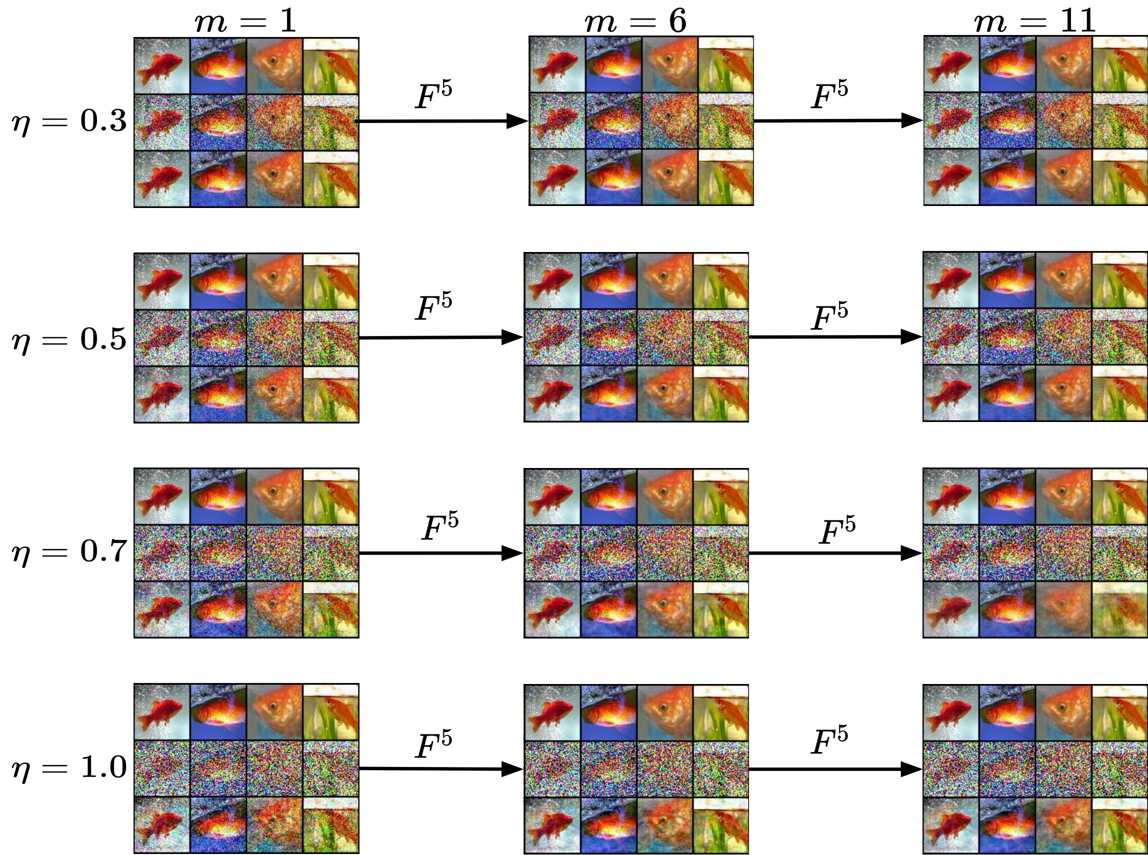


Figure 4.9: Analysis of the reconstruction capabilities of 10 iterations of the function F under different levels of noise $\eta \in \{0.3, 0.5, 0.7, 1.0\}$. Particularly, we plotted the results after 1, 6, and 11 iterations of F.

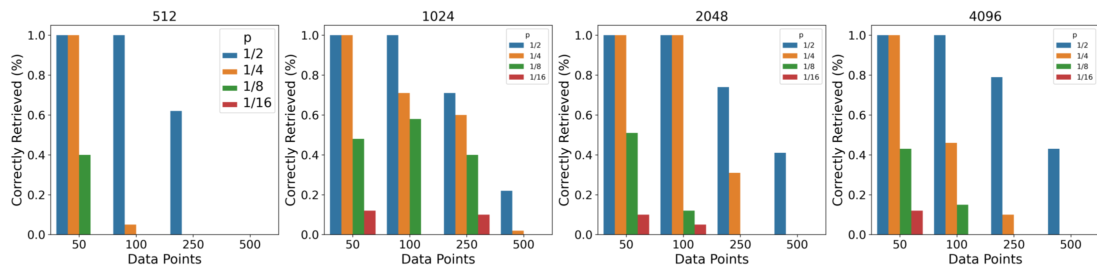


Figure 4.16: Percentage of perfectly retrieved images given different fractions of the original images for PCNs with 512, 1024, 2048, and 4096 hidden neurons trained on datasets of different sizes. Note that here we count as retrieved only images with non-visible errors. Most of the images that are not considered retrieved in this plot, were correctly retrieved, but only slightly noisy.

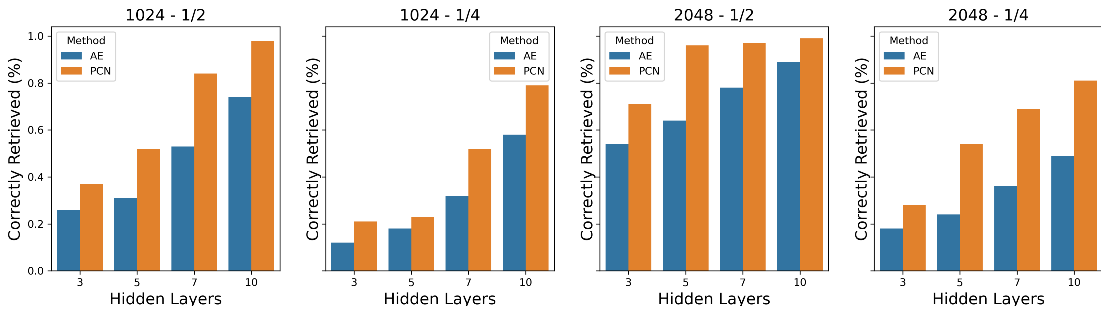


Figure 4.17: Percentage of correctly retrieved images as a function of the number of hidden layers, for both autoencoders and generative PCNs, on a dataset of 500 images. We used networks with $n = \{1024, 2048\}$ hidden neurons, and the images are taken from Tiny ImageNet.

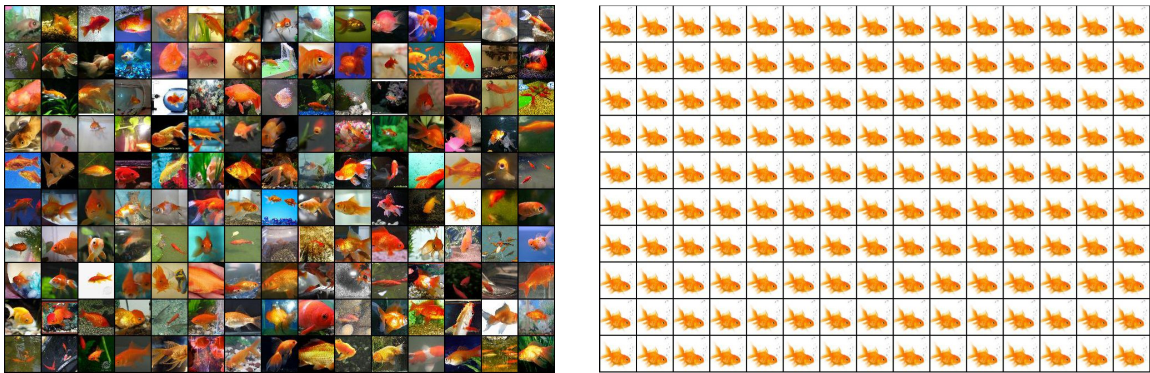


Figure 4.18: Left: reconstruction of 150 images of the Tiny ImageNet dataset when provided with $1/2$ of the original image. To generate the figure, we have used a generative PCN with 2 hidden layers and hidden dimension $n = 1024$. Right: same task performed using an MHN with $\beta = 2$. Overall, our method has retrieved all the presented images, while the MHN has retrieved only one, which seems to correspond to the only strong attractor of the dynamics.

Figure 4.19: Left: reconstruction of DANN on CIFAR10 $32{\times}32$ images (taken from [115]). Centre: our reconstruction using a generative PCN with 1024 hidden neurons on a fraction of the same dataset. Right: our reconstruction on 50 ImageNet $224{\times}224$ images.

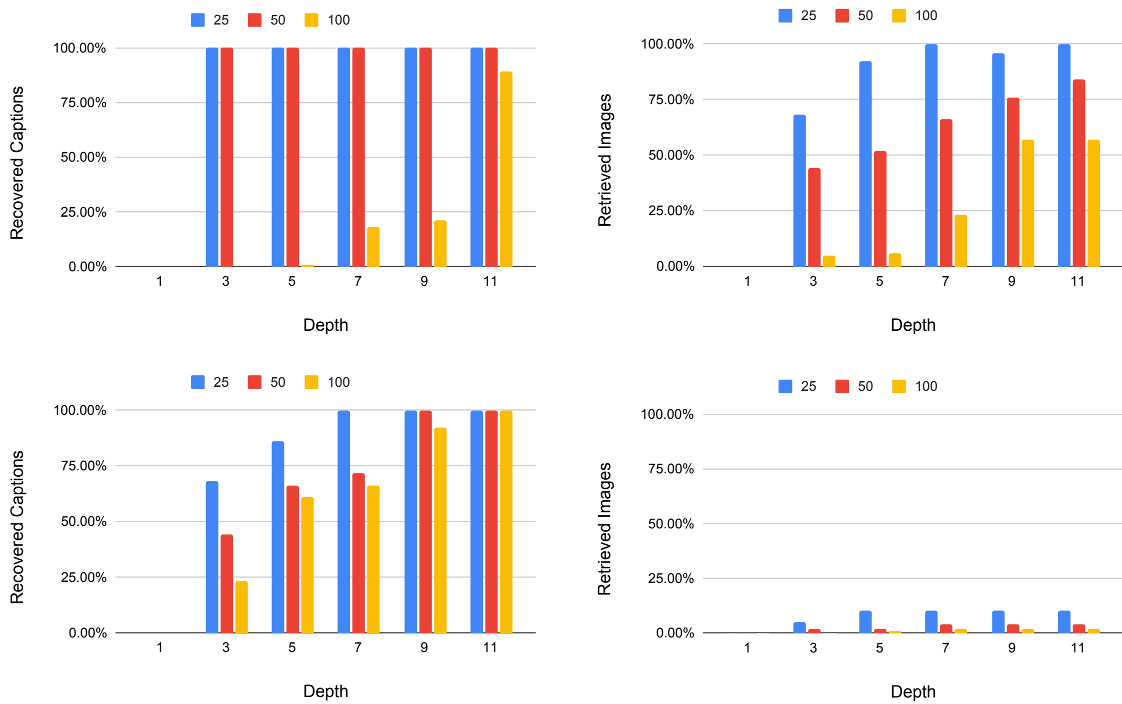


Figure 4.20: Depth comparison for PCNs with respect to hetero-associative memories. Top row shows the results for PCNs, bottom row the ones for autoencoders. The number of captions (resp., images) recovered is shown on the left (resp., right) side. Both models are trained for depth $L \in \{1, 3, 5, 7, 9, 11\}$ and width $n \in \{512, 1024, 2048\}$, with the best number across n being shown.

Figure 4.21: Recovery of 25 colored images from captions. Left: images recovered by our generative PCN. Right: images recovered by an autoencoder with the same number of parameters. For generative PCNs, all images are visually identical to the original ones. For autoencoders, only two images are recovered from captions, although we can see that several images are stored. Again, autoencoders perform strict retrieval, and hence it is not possible to trust their output.

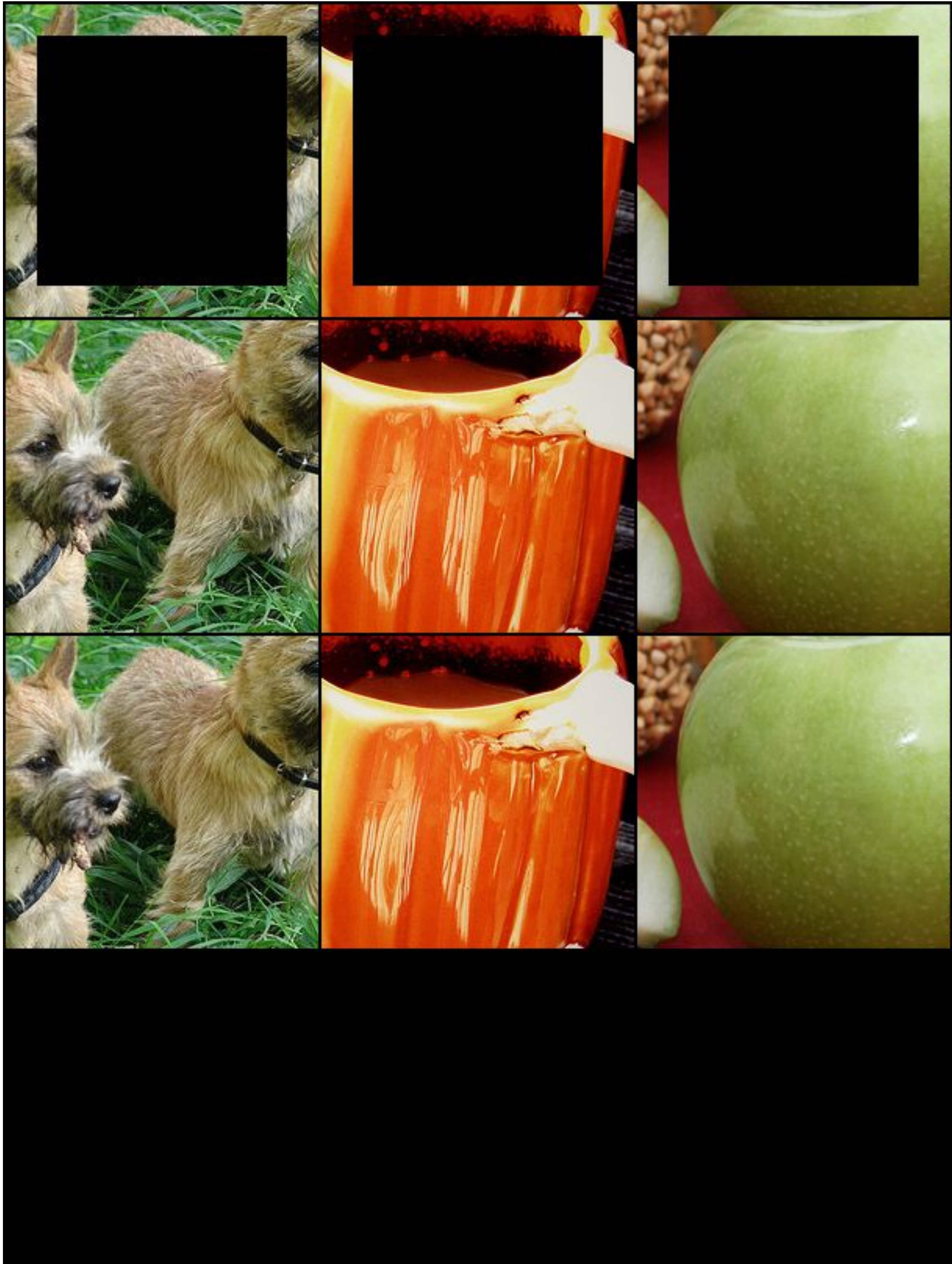


Figure 4.22: To show the level of accuracy of our proposed method, we represent reconstructions of ImageNet pictures. Here, a reconstruction when a patched image is presented to our generative PCN. From top to bottom: partial image, reconstructed image, original image, and reconstruction error (difference between original and reconstruction).

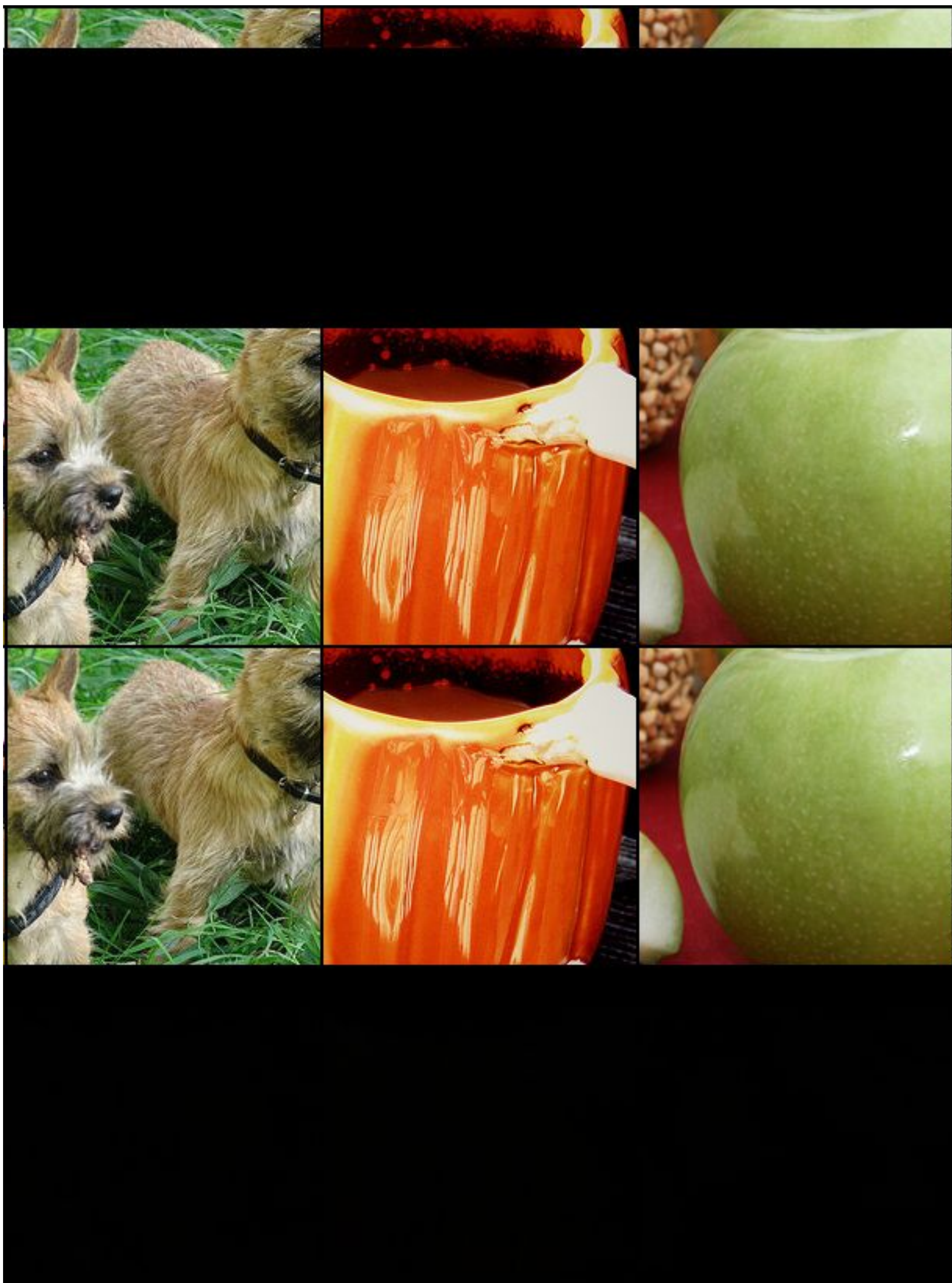


Figure 4.23: To show the level of accuracy of our proposed method, we represent reconstructions of ImageNet pictures. Here, a reconstruction when only $1/8$ of the original picture is presented to our generative PCN. From top to bottom: partial image, reconstructed image, original image, and reconstruction error (difference between original and reconstruction).

Chapter 5

Learning on Arbitrary Graph Topologies

In the previous chapters, I have studied the similarities between PC and BP on supervised learning, and the associative memory capabilities of PC when considering it as an unsupervised, generative, model. Here, I study the novel problem of training neural networks with any graph structure. This is a direction of research which is under explored in the field, as BP does not allow to train on such networks on static data. In fact, training with BP in standard deep learning consists of two main steps: a forward pass that maps a data point to its prediction, and a backward pass that propagates the error of this prediction back through the network. This process is highly effective when the goal is to minimize a specific objective function. However, it does not allow training on networks with cyclic or backward connections. This is a potential obstacle to reaching brain-like capabilities, as the highly complex heterarchical structure of the neural connections in the neocortex are potentially fundamental for its effectiveness. In this chapter, I show how predictive coding can be used to perform inference and learning on arbitrary graph topologies. To do that, I introduce the concept of *PC graphs*, a generalization of PCNs, and experimentally show how they can be used to flexibly perform different tasks with the same network by simply stimulating specific neurons, and investigate how the topology of the graph influences the final performance.

As already stated, training on networks of any structure is not possible in standard deep learning, where information only flows in one direction via the feedforward pass and then BP is performed in sequential steps backwards. If a cycle is present inside the computational graph of an artificial neural network, BP becomes stuck in an infinite loop. More generally, the computational graph of any function $F\colon\mathbb{R}^d \to \mathbb{R}^k$ is a poset, and hence acyclic. While the problem of training on some specific cyclic structures has been partially addressed using BP through time on sequential data [100], the restriction to pure sequential architectures may present a limitation to reaching brain-like intelligence, since the

Figure 5.1: Difference in topology between an artificial neural network (left), and a sketch of a network of structural connections that link distinct neural elements in a brain (right) [62].

human brain has an extremely complex and entangled neural structure that is heterarchically organized with small-world connections [62]—a topology that is likely highly optimized by evolution. This shape of structural brain networks, shown in Fig. 5.1, generates a unique communication dynamics that is fundamental for information processing in the brain, as different aspects of network topology imply different communication mechanisms, and hence perform different tasks [62]. The heterarchical topology of brain networks has motivated research that aims to develop learning methods on graphs of any topology. A popular example is the *assembly calculus* [48, 119], a Hebbian learning method that can perform different operations implicated in cognitive phenomena. However, Hebbian learning methods cannot perform well compared to error-driven ones such as BP, since they are fundamentally limited to simply detecting local pairwise correlations in the input data, and thus implement a local form of principal components analysis [120, 121, 122]. In this chapter, I address this problem by exploring *PC graphs*, a structure that allows to train on any directed graph using PC. I then demonstrate the flexibility of such networks by testing the same trained network on different tasks, which can be interpreted as Bayesian conditioning on different neurons of the network. Our PC graphs framework enables the model to be queried on stimuli with different structures, such as partial images, images with labels, or images without labels. This is significantly more flexible than the strict input-output structure of standard ANNs, which are limited to scenarios when they are always presented with data and labels in the same format.

Conceptually, PC graphs are similar to popular energy-based models, such as Hopfield networks [13] and Boltzmann machines [96], that perform inference by minimizing an energy function. However, when tested on classification tasks, the fully connected version of PC graphs performs better than them, and the multilayer versions perform better than standard restricted Boltzmann machines [96, 22]. In this chapter, I study the generation, classification, and associative memory capabilities of PC graphs, highlighting their flexibility

and theoretical advantages over standard baselines. The contributions of this chapter are briefly as follows:

- I introduce PC graphs, that generalize PC to arbitrary graph topologies, and show how a single model can be queried in multiple ways to solve different tasks by simply altering the values of specific nodes, without the need for retraining when switching between tasks. Particularly, I define two different techniques, called *query by conditioning* and *query by initialization*.

- I then experimentally show this in the most general case, i.e., for fully connected PC graphs. Here, I train different models on MNIST and FashionMNIST, and show how the two queries can be used to perform different generation tasks. Then, I test the model on classification tasks, and explore its capabilities as an associative memory model.

- I next investigate how different graph topologies influence the performance of PC graphs on generation tasks, reproducing common network architectures such as feedforward, recurrent, and residual networks as special cases of PC graphs, and investigate how the chosen structure influences the performance on generative tasks. Finally, I also show how PC graphs can be used to derive the popular *assembly calculus* model [48].

5.1 PC graphs

Here, we introduce PC graphs, and define two flexible techniques to query them. Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph, where $\mathcal{V}$ is a set of n neurons $\{1, 2, \ldots, n\}$, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed edges between them, where every edge $(i, j) \in \mathcal{E}$ has a weight parameter $\theta_{i,j}$. The set of neurons $\mathcal{V}$ is partitioned into two subsets, the *sensory* and *internal neurons*. External stimuli are always presented to the network via sensory neurons, that we consider to be the first d neurons of the graph, with $d < n$. The internal neurons, on the other hand, are used to represent the internal structure of the dataset. Again, the main quantity of every neuron is the *value node* $x_{i,t}$. We call the value nodes of the sensory neurons *sensory nodes*. Additionally, each neuron computes the *prediction* $\mu_{i,t}$ of its activity based on its input from value nodes of other neurons:

$$\mu_{i,t} = \textstyle\sum_j \theta_{j,i} f(x_{j,t}), \tag{5.1}$$

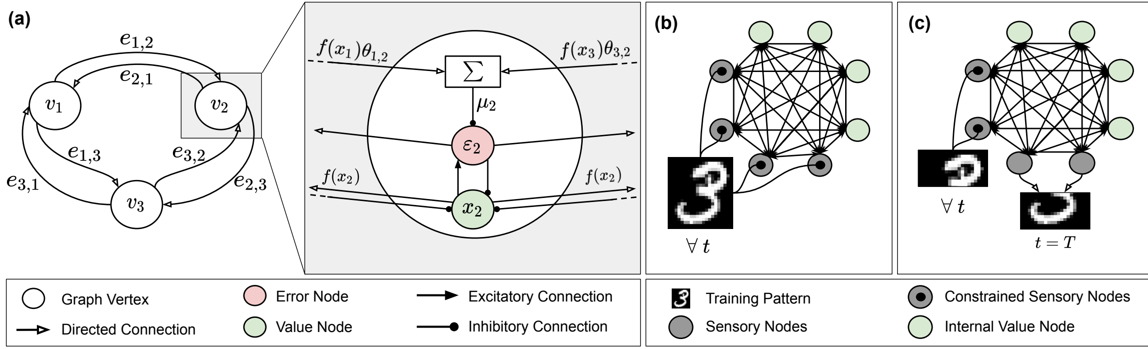


Figure 5.2: (a) An example of a fully connected PC graph with three neurons. Zoomed is the neural implementation of PC, where learning is made local via the demonstrated inhibitory and excitatory connections. This figue is similar to the one introduced in the previous chapter, with the addition of non-linearities. (b) A sketch of the training process, where the value nodes of the sensory neurons are fixed to the pixels of the image. (c) A sketch of query by conditioning, where a fraction of the value nodes is fixed to the top half of an image, and the bottom half is recovered via inference.

Algorithm 8 Learning the external stimulus $\bar{s}$

Require: $(x_{1,t}, \dots, x_{d,t})$ is fixed to $(s_1, \dots, s_d)$.
1: **for** $t = 0$ to T **do**
2: **for** each neuron i **do**
3: update $x_{i,t}$ to minimize E_t via Eq. (5.3)
4: **if** $t = T$ **then**
5: update every $\theta_{i,j}$ to minimize E_t via Eq. (5.4).
;

where the summation is over all the neurons j connected to i via outgoing edges, and f is a non-linearity. Equivalently, it is possible to consider the summation on every j, and have $\theta_{i,j} = 0$ if $(i, j) \notin \mathcal{E}$. To conclude, the value nodes $x_{i,t}$ and the weight parameters θ are updated to minimize the usual energy function

$$E_t = \tfrac{1}{2}\textstyle\sum_i (\varepsilon_{i,t})^2. \tag{5.2}$$

A fully connected PC graph with 3 neurons is sketched in Fig. 5.2a, along with the operations that describe the dynamics of the information flow, showing also how every operation can be represented via inhibitory and excitatory connections.

Learning: When presented with a training point $\bar{s}$ taken from a training set, the value nodes of the sensory neurons are fixed to be equal to the entries of $\bar{s}$ for the whole duration of the training process, i.e., for every t. A sketch of this is shown in Fig. 5.2b. Then, the

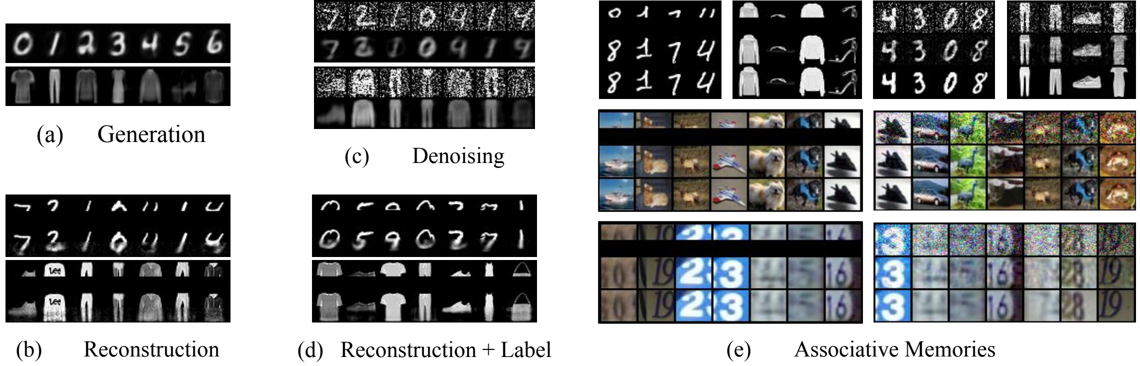


Figure 5.3: Generation experiments using the first 6 classes of the MNIST and Fashion-MNIST datasets from the labels {0, 1, 2, 3, 4, 5, 6} and {t-shirt, trouser, pullover, dress, coat, sandal, shirt}, respectively; (b) reconstruction of incomplete images using *query by conditioning*, when only the top half is available; (c) reconstruction of corrupted images using *query by initialization*; (d) reconstruction of incomplete images using *query by conditioning* when also providing the correct label of the test image; and (e) associative memory experiments when presented with half of a training image (left) or a corrupted version (right) that it has already seen and memorized; from top to bottom row: image provided to the network, retrieved image, and original image.

total energy of Eq. (5.2) is minimized via inference and weight update. During the inference phase, the weights are fixed, and the value nodes are continuously updated via gradient descent for T iterations, where T is a hyperparameter of the model. The update rule is the following (*inference*):

$$\Delta x_{i,t} = -\gamma \cdot \partial E_t / \partial x_{i,t} = \gamma \cdot (-\varepsilon_{i,t} + f'(x_{i,t}) \textstyle\sum_{k=1}^{n} \varepsilon_{k,t} \theta_{k,i}), \tag{5.3}$$

where γ is the learning rate of the value nodes. This process of iteratively updating the value nodes distributes the output error throughout the PC graph. When the inference phase is completed, the value nodes get fixed, and a single weight update is performed as follows:

$$\Delta \theta_{i,j} = -\alpha \cdot \partial E_t / \partial \theta_{i,j} = \alpha \cdot \varepsilon_{i,T} f(x_{j,T}), \tag{5.4}$$

where α is the learning rate of the weight update. We now describe two different ways to query the internal representation of a trained model, where the values of some sensory neurons are undefined, and have to be predicted. In both cases, the weight parameters $\theta_{i,j}$ are now fixed, and the total energy E is continuously minimized using gradient descent on the re-initialized value nodes via Eq. (5.3).

Query by conditioning: While each value node is randomly re-initialized, the value nodes of specific neurons are *fixed* to some desired value, and hence not allowed to change during

the energy minimization process. The unconstrained sensory neurons will then converge to the minimum of the energy given the fixed neurons, thus computing the conditional probability of the latent neurons given the observed stimulus. Formally, let $I = \{i_1, \ldots, i_q\} \subset \{1, 2, \ldots, n\}$ be a strict subset of neurons. Assume now that we know that a subset of the value nodes corresponding to the neurons I is equal to a stimulus $\bar{q} \in \mathbb{R}^q$. Then, running inference until convergence allows to estimate the conditional probability

$$P(\bar{x}_T \mid \forall t\colon (x_{i_1,t}, \ldots, x_{i_q,t}) = \bar{q}), \tag{5.5}$$

where $\bar{x}_T$ is the vector of value nodes at convergence. Examples of tasks performed this way are (i) classification, where internal nodes are fixed to the pixels of an image and the sensory nodes are fixed to a 1-hot vector with the labels, (ii) generation, where the single value node encoding the class information is fixed, and the value nodes of the sensory nodes converge to an image of that class, and (iii) reconstruction, such as image completion, where a fraction of the sensory nodes are fixed to the available pixels of an image, and the remaining ones converge to a reasonable completion of it. A sketch of this process is shown in Fig. 5.2c.

Query by initialization: Every value node is randomly initialized, but the value nodes of specific nodes are *initialized* (for $t = 0$ only), but not fixed (for all $t > 0$) to some desired value. This differs from the previous query, as here every value node is unconstrained, and hence free to change during inference. The sensory neurons will then converge to the minimum found by gradient descent, when provided with that specific initialization. Let $I = \{i_1, \ldots, i_q\} \subset \{1, 2, \ldots, n\}$ be a strict subset of neuron indices, and assume that we have an initial a stimulus $\bar{q} \in \mathbb{R}^q$. Then, we can estimate the probability

$$P(\bar{x}_T \mid (x_{i_1,0}, \ldots, x_{i_q,0}) = \bar{q})\,. \tag{5.6}$$

Examples of tasks performed this way are (i) denoising, such as image denoising, where the sensory neurons are initialized with a noisy version of an image, which is cleared during the energy minimization process, and (ii) reconstruction, such as image completion, where the fraction of missing pixels is now not known a priori.

5.2 Experiments on fully connected graphs

In this section, we perform experiments on a fully connected PC graph $G = (\mathcal{V}, \mathcal{E})$, i.e., where $\mathcal{E} = \mathcal{V} \times \mathcal{V}$. Such PC graphs are fully general and encode no implicit priors on the structure of the dataset. It is possible to obtain any possible graph topology by simply pruning specific weights of G. This model only differs from the fully connected one

Table 5.1: Test accuracy of different models on MNIST, FashionMNIST, and SVHN.

Model	Ours	Hopfield Network	Boltzmann Machine	Almeida Pineda
MNIST	91.76 ± 0.02 %	65.23 ± 2.21 %	79.23 ± 0.15 %	76.36 ± 0.14 %
FMNIST	83.72 ± 0.33 %	51.74 ± 3.94 %	61.31 ± 0.17 %	69.63 ± 1.64 %
SVHN	84.51 ± 0.11 %	48.92 ± 3.11 %	55.74 ± 1.23 %	59.14 ± 2.64 %

introduced in the associative memory chapter due to the presence of non linear activation functions and internal neurons.

Given a dataset $\mathcal{D} = \{\bar{s}_i\}_{i<m}$, with $\bar{s}_i \in \mathbb{R}^d$, we train the PC graph as described in Section 5.2: the first d neurons are fixed to the entries of a training point, and the energy function E_t is minimized via inference and weight updates, via Eqs. (5.3) and (5.4). When the training is complete, we show the different tasks that can be performed, without the need of retraining the model. We use MNIST and FashionMNIST, fixing the first d nodes to the data point, and show how to perform the tasks of generation, denoising, reconstruction (without and with labels), and classification by querying the PC graph as described in Section 5.1.

Setup: For every dataset, we have trained 3 models: one for generation and classification tasks, one for denoising and reconstructions, and one for associative memories. The first two models consist of a fully connected graph with 2000 neurons, trained with 794 sensory neurons for classification and generation tasks (784 pixels plus a 1-hot vector for the 10 labels), and 784 sensory neurons for reconstruction and denoising. Further details about other hyperparameters are given in Section 5.6.

Generation: To check the generation capabilities of a trained PC graph, we queried the model by conditioning on the labels: here, the value nodes dedicated to the 10 labels were fixed to each 1-hot value, and the energy of the model (Eq. (5.2)) was minimized using (Eq. (5.3)) until convergence. The generated images are then taken to be the value nodes of the unconstrained sensory neurons,that were originally fixed to the pixels of the images during training. An example of the images generated for each label is given in Fig. 5.3a.

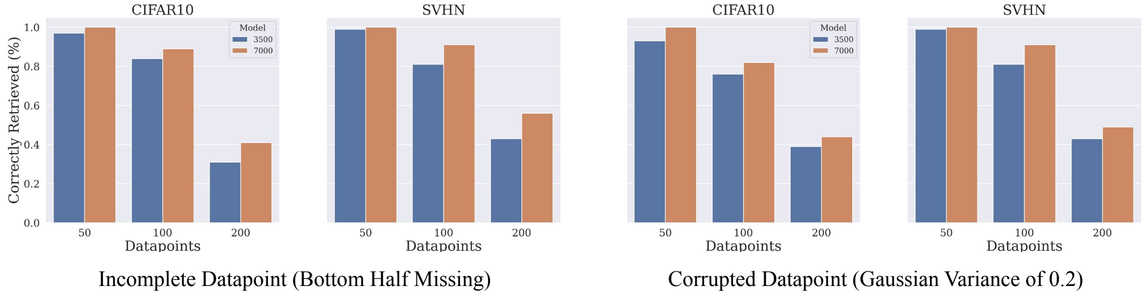
Reconstruction: We provide the PC graph with half of a test image, and ask it to reconstruct the second half. This can be done using both queries: when querying by conditioning, half of the pixels of a test image are fixed to the corresponding sensory nodes; when querying by initialization, the value nodes are simply initialized to the same values. At convergence, we consider the value nodes of the unconstrained nodes, which should reconstruct the

missing part of the image based on the information learned during training. The results are given in Fig. 5.3b. We have also replicated the same experiment using a network trained with the labels, and provided the label during the reconstruction. This computes the distribution of the missing pixels knowing the available ones *and* the label. The results in this case are visibly better and are given in Fig. 5.3d.

Denoising: We provide the PC graph with a corrupted image, obtained by adding zero-mean Gaussian noise with variance 0.5. This is done by querying by initialization: before running inference, the value nodes of the sensory nodes are initialized to be equal to the pixels of the corrupted image. At convergence, we consider the value nodes of the unconstrained nodes, which should reconstruct the original image. The results are given in Fig. 5.3c.

Results: As stated above, we picked a fully connected PC graph due to its generality, and not to obtain good performance. However, the results show that this framework is able to learn an internal representation of a dataset, and that it can be queried to solve multiple tasks with a reasonable accuracy. The PC graph was in fact able to always generate the correct digit, and almost always able to generate the correct clothing item, and always able to provide a noisy but reasonable reconstruction of incomplete test points. The same happened with denoising experiments, as a cleaner, plausible, image was always produced. In the next section, we will show how to improve all these performances by using different PC graph topologies.

Classification: We consider the same PC graph trained for the generation experiments. To check its generalization capabilities, we query by conditioning the pixels of every test image to the first 784 sensory nodes, and run inference to reconstruct the 1-hot label vector. We do not expect to obtain results directly comparable with standard MLPs for two reasons: firstly, the model does not contain any implicit hierarchy, which empirically appears crucial to obtaining good classification results. Secondly, the PC graph is also simultaneously learning to generate the pixels, which are much more numerous than labels. However, to check whether the obtained results were acceptable, we tested against different learning algorithms that train on similar or equivalent fully connected architectures, such as Hopfield networks, unconstrained Boltzmann machines, and a local variation of BP introduced in the late $'80$, called Almeida-Pineda [123, 124]. As for Hopfield networks, we used the implementation provided in [125]. The results, given in Table 5.1, show that our model outperforms every other learning algorithm that can be trained on fully connected architectures. Despite this,



Incomplete Datapoint (Bottom Half Missing) Corrupted Datapoint (Gaussian Variance of 0.2)

Figure 5.4: Number of correctly retrieved data points when presented with incomplete or corrupted variations. We used datasets of $\{50, 100, 200\}$ images of the CIFAR10 and SVHN datasets, and trained on fully connected PC graphs of size $\{3500, 7000\}$ neuron.

the results also show that the obtained test accuracy is not nearly comparable to the results obtained by MLPs, as they are only slightly better than a linear classifier, that obtains 88% accuracy on MNIST. However, this is not due to the learning rule of PC, which is well-known to be able to reach a competitive performance when provided with a hierarchical multilayer structure [22]. For the SVHN [65] experiment, we used models with 5000 neurons.

5.2.1 Associative memory

I now test whether PC graphs are able to perform complex experiments on training images, i.e., images that they have already seen. Particularly, we show that a fully connected PC graph is able to store complex data points, such as colored images, and retrieve them via running inference. To do that, we trained a novel fully connected PC graph on 100 data points of the MNIST, FashionMNIST, and CIFAR10 datasets. We have used a model with 1000 neurons for MNIST and FashionMNIST, and 3500 for SVHN and CIFAR10, and asked it to retrieve the original memories by presenting it either only half of the original pixels, or a corrupted version with Gaussian noise variance 0.2. This task is similar to image reconstruction and denoising, with the non-trivial difference that here we only use already seen data points, and hence no generalization is involved. The results of these experiments are given in Fig. 5.3e, and show that our method is able to successfully store and retrieve data points via energy minimization.

To show that non-linear PC graph are able to perform associative memory experiments, we trained fully connected PC graphs with $\{3500, 7000\}$ neuron on different subsets of cardinality $\{50, 100, 200\}$ of CIFAR10 and FashionMNIST. Then, we used query by initialization and conditioning to retrieve the original memories. In this setting, we considered a memory to be retrieved if the mean square error between the original training point and its reconstruction is less than 0.001. As corruption, we either removed the top half of the

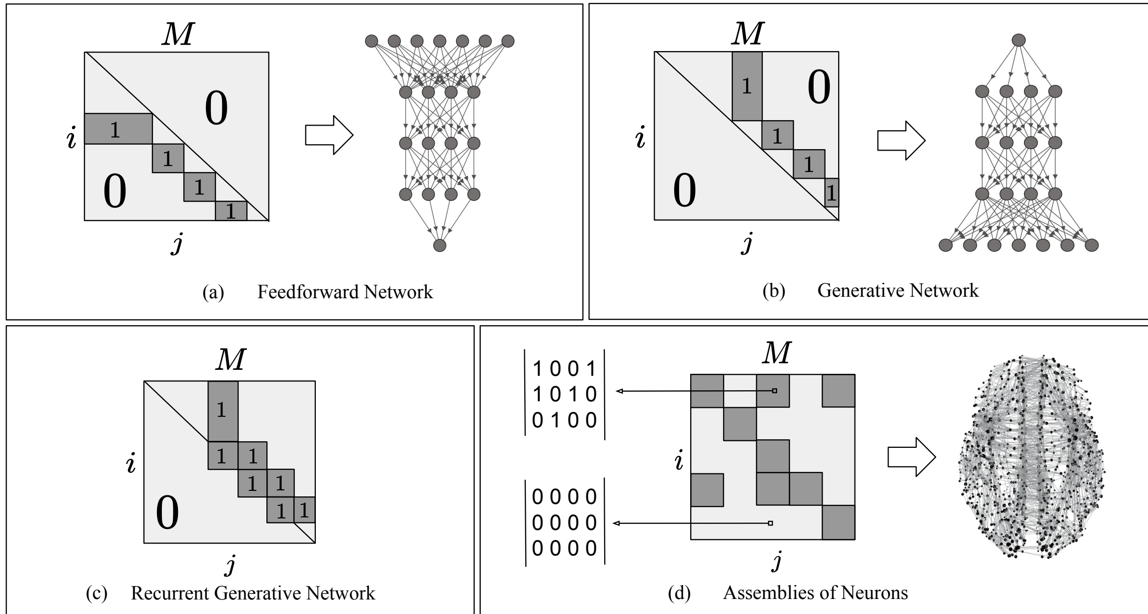


Figure 5.5: Examples of PC graphs that can be built by masking a part of the weights of a fully connected PC graph. (a) Masking required to build a standard multilayer architecture, such as the one in [22]. (b) Masking required to build a multilayer architecture, where the weights go in the opposite direction. Here, the sensory nodes are at the end of the hierarchical structure. This model is equivalent to the multilayer generative networks introduced in Chapter 4. (c) Examples of masking needed to implement popular architectures with lateral connections, similar to the model in [126]. (d) This is the model in [48], which consists of a set of Erdõs–Renyi graphs that simulate brain regions (dark squares on the diagonal) and connections between them.

image, or corrupted it with Gaussian noise of mean zero and variance 0.2. The results are shown in Fig. 5.4.

Results: The experiments show that our model is able to store and retrieve memories well, even when tested on colored images. The reconstruction quality, as expected, decreases when adding more memories, and improves when adding more parameters to the model. As hyperparameters, we used $\eta = 0.0001$, $\gamma = 0.5$, $T = 5$.

5.3 Extension to different graph topologies

As well-known in deep learning, the performance of the trained ANN strongly depends on its architecture: the number of neurons, layers, and their intrinsic structure. In Section 3, we studied the general architecture of fully connected PC graphs. Here, we show how to reduce a fully connected PC graph to lighter and even more powerful PC graphs. Particularly, we

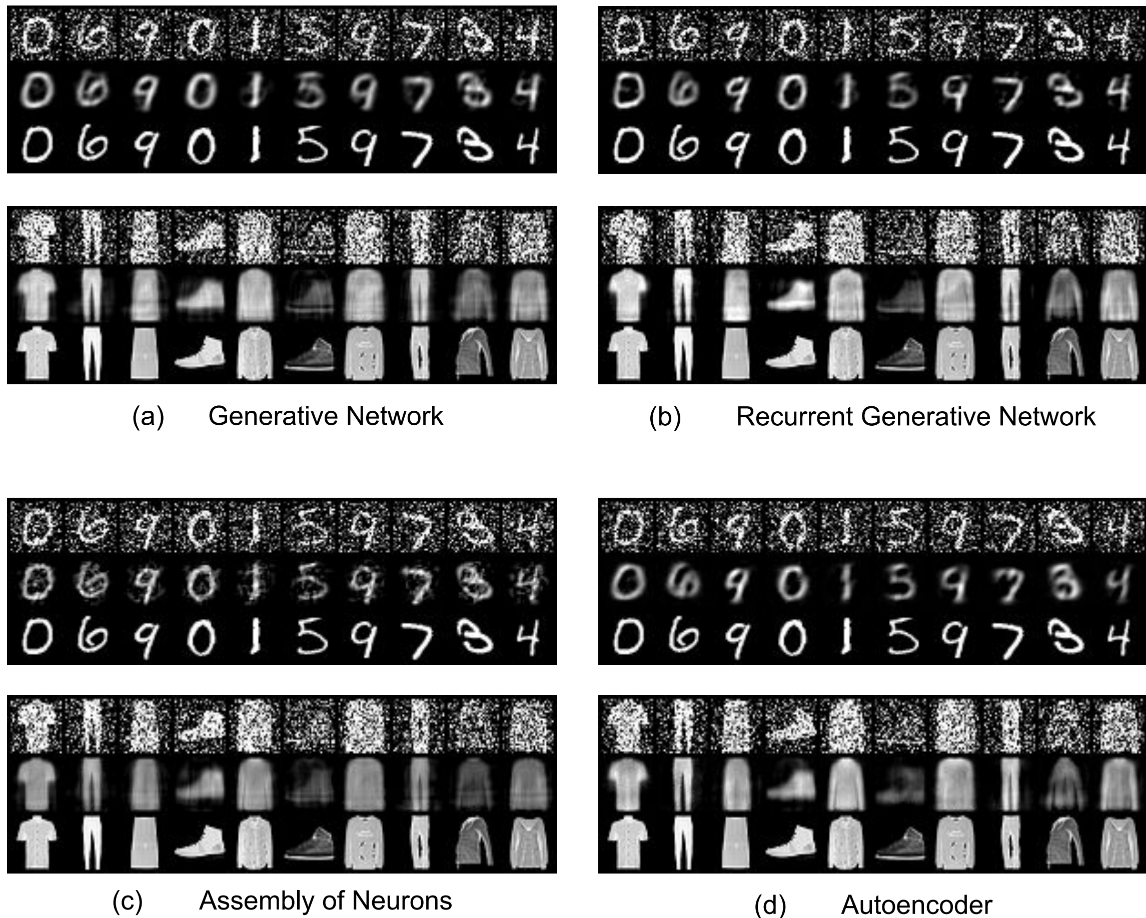


Figure 5.6: *Query by initialization* on three different PC graph architectures and different datasets. Particularly, we tested these PC graphs against ANN autoencoders trained with BP (d), which perform comparably to the PC graphs on denoising tasks.

show how to generate different neural architectures by simply pruning specific edges of a fully connected PC graph $G = (\mathcal{V}, \mathcal{E})$. Every architecture built this way is fully parallelizable, and only uses local information to update the weight parameters. In this case, the pruning is performed by applying a sparse mask $\boldsymbol{M}$. However, there are multiple equivalent ways of implementing it.

Consider now the weight matrix $\theta \in \mathbb{R}^{n \times n}$, where every entry $\theta_{i,j}$ represents the weight parameter connecting neuron i to neuron j. To generate a neural architecture that consists of a subset of the original connections, it suffices to *mask* the matrix θ via entry-wise multiplication with a binary matrix $\boldsymbol{M}$, where $\boldsymbol{M}_{i,j} = 1$ if the edge (i, j) exists in $\mathcal{E}$, and $\boldsymbol{M}_{i,j} = 0$ otherwise. This allows the creation of hierarchical discriminative architectures such as a PC equivalent of the multilayer perceptron (MLP) in Fig. 5.5a, or hierarchical generative networks in Fig. 5.5b, c. More generally, it creates a framework to generate

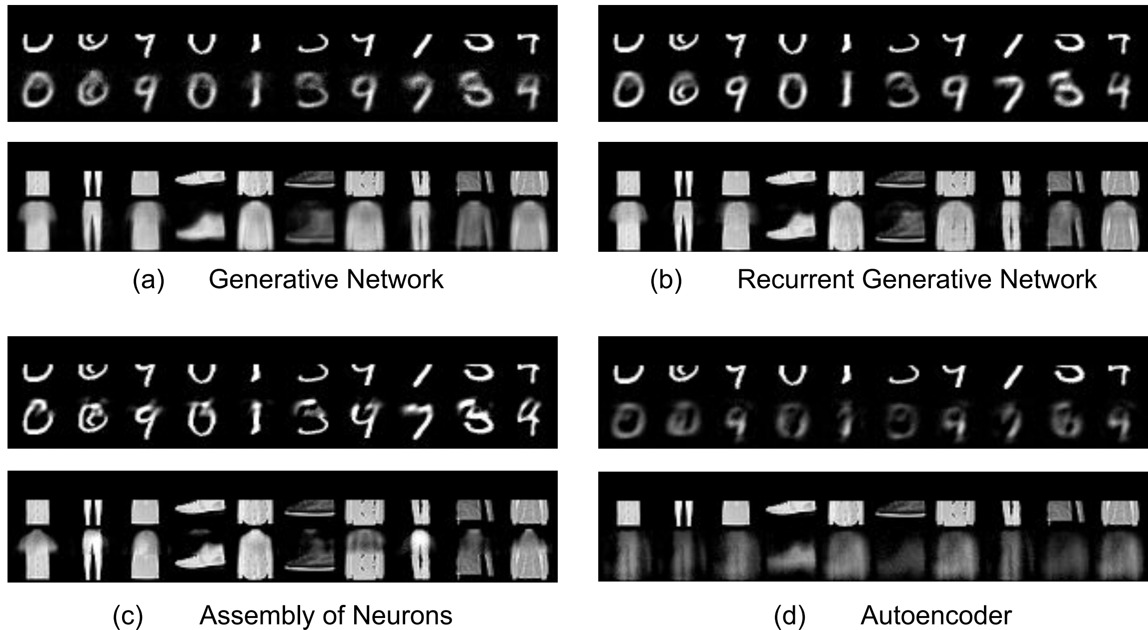


Figure 5.7: *Query by conditioning* on three different PC graph architectures and different datasets. Particularly, we tested these PC graphs against ANN autoencoders trained with BP (d), which perform less well on image reconstruction.

and study architectures with any topology, such as small-world networks inspired by brain regions [127], as shown in Fig. 5.5d.

Experiments: We test whether the above architectures are able to outperform a fully connected PC graph, and hence study how the network topology influences the final performance by providing a comparison between test accuracies of different models. Besides this, we perform the same experiments shown on the fully connected PC graph. We expect the generated images to be visibly better due to the enforced hierarchical structure of the PC graph.

Setup: We trained generative PC graphs, recurrent generative PC graphs, assemblies of neurons PC graphs, and standard BP autoencoders with different numbers of hidden layers and hidden dimension, and report the best results. For the generation results, we used the same setup, but added an input layer with 10 neurons, whose value nodes during training were initialized with the 1-hot label vector. More details about the hyperparameters used are given in Section 5.6, as well as a discussion on how different parameters influence the final performance of the architecture. A brief discussion comparing the generation capabilities of our model to restricted Boltzmann machines is also included at the end of the chapter.

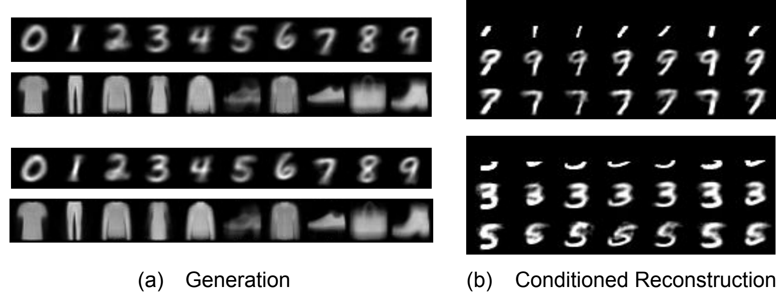

(a) Generation (b) Conditioned Reconstruction

Figure 5.8: Generated images given the labels using feedforward (top) and recurrent (bottom) PC graphs.

Model	PC	RBM	DAM	BP
MNIST	98.47 ± 0.12	94.12 ± 0.59	98.58	98.41 ± 0.18
FashionMNIST	89.92 ± 0.23	86.98 ± 0.49	90.22 ± 0.27	90.29 ± 0.33
SVHN	88.99 ± 0.26	85.09 ± 0.87	86.77 ± 0.22	89.31 ± 0.09
CIFAR10	56.23 ± 3.36	41.12 ± 3.88	46.06 ± 2.77	59.11 ± 2.47

Table 5.2: Test accuracy of multilayer PCNs (i.e., feedforward PC graphs) on MNIST, FasionMNIST, SVHN, and CIFAR10. The results are compared against popular models in the literature: restricted Boltzmann machines [96], dense associative memories [107], and MLPs trained with BP [55]. Classification on MNIST using DAM does not report variance, as it is taken from the original work, and the authors only report the average.

Results: The results for denoising tasks are given in Fig. 5.6a and b, while the ones for reconstruction tasks in Fig. 5.7a and b. As expected, the hierarchical structure of the considered PC graphs improves over the fully connected PC graph, despite being comparable in the number of parameters. Compared against autoencoders (Fig. 5.7d and 5.6d), the standard ANN baseline trained with BP, the PC graph results are similar in image denoising, and better in image reconstruction.

Classification results: We now compare against popular models in the literature, such as restricted Boltzmann machines (RBMs) [96] and *dense associative memories* (DAMs) [107]. Overall, PCNs are the only models able to perform similarly to BP on the test set. We performed experiments on 4 datasets: MNIST, FashionMNIST, SVHN, and CIFAR10, and the results are in Table 5.2.

Setup: The networks trained using PC and BP have $L = \{2, 3\}$, and 256 hidden neurons each. They are trained using Adam optimization, a weight decay $\lambda \in \{0.001, 0.0001, 0\}$ and a learning rate for the weights $\alpha \in \{0.001, 0.0001\}$. We report the best average results in

Table 5.2. For the RBM, we used a model with 512 hidden nodes, and for the DAM, we copied the official implementation provided by the authors, with the same hyperparameters.

5.4 Conditioning on labels

Let us assume we need to reconstruct a test image from an incomplete version of it. As seen above, this can be done by querying the model by conditioning. However, assume now that this time we are also provided with the label of the corrupted image. It would be useful to be able use this extra information to obtain a better reconstruction. In PC graphs, this is straightforward: it suffices to simultaneously fix the value nodes representing the labels to the 1-hot vector of the provided label, and the sensory nodes to the pixels of the corrupted image. We have already shown how this improves the reconstruction quality in Fig. 5.2d. This method, however, has a second, more significant, application. Sometimes, it is difficult to infer to which class an incomplete image belongs, and providing the label during the reconstruction allows the preferred label to influence the reconstruction. Hence, we perform the following task: we provide images of digits that look similar when incomplete, and ask the model to reconstruct the missing half when giving the label information, i.e., use the additional label information to correctly resolve the inherent ambiguity in the reconstruction task.

Experiments: We used the same PC graphs from above for generation tasks. We provided the PC graph the bottom 33% of random images representing 7s or 9s. Note that it is hard to distinguish between these two numbers when only this small portion of the image is available. Then, we generated the missing 67% of the pixels by first giving 7 as a label, and then giving 9. We have repeated the same task using 3s and 5s. The results, available in Fig 5.8b, show that our model is able to perform conditional inference, as the reconstructed digits always agree with the provided labels.

5.5 Assembly of neurons

In this section, we show an example of how this method can be used to train brain-inspired architectures. Recently, a model made by assemblies of neurons that are sparsely connected with each other has been proposed to emulate brain regions [48]. This model consists of m ordered clusters of neurons $(C_1, \dots, C_m)$, and any two ordered neurons of the same cluster are connected by a synapse with probability p, creating an Erdõs–Renyi graph $G_{m,p}$. Depending on the desired task, two clusters can be connected via sparse connections

following the same rule: if cluster C_a is connected to cluster C_b, then, given a neuron $v_i \in C_a$ and a neuron $v_j \in C_b$, there exists a synaptic connection connecting v_i to v_j with probability p. Note that this structure is highly general, and allows to build networks such as the one represented in Fig. 5.1b. To conclude, at each time step, only the k neurons of every cluster with the highest neural activity fire. In the original work, the authors propose a Hebbian-like learning algorithm, however, we show that the exact same structure can also be trained using PC graphs. A graphical representation on how to encode as a PC graph a network made by assemblies of neurons is given in Fig. 5.5d. In this case, each dark block on the diagonal represents connections between neurons of the same region. Unlike the other networks in the same figure, these are sparse matrices where every entry is either zero, or one with probability p. As in the brain, not every region is connected with the other, and whether two regions are directly connected has to be decided a priori when designing the architecture. Two neurons between connected regions are directly connected with probability p. In Fig. 5.5d, dark blocks off the diagonal represent the presence of directed connections between two regions C_a and C_b. If situated below the diagonal, the connections go from C_a to C_b, with $a < b$; if situated above the diagonal, they go from C_b to C_a.

Experiments: We replicated this structure, using 4 clusters with 3000 neurons each, connected in a feedforward way: the first cluster is connected with the second, which is connected with the third, which is connected to the fourth. As sparsity and top-k constants, we used $p = 0.1$ and $k = 0.2$, and performed the same generative experiments. The results are given in Fig. 5.7c. While the results look cleaner than the other methods, note that they are specific to MNIST and FashionMNIST, as the top-k activation on the last cluster well cleans the noise surrounding the reconstructions.

5.6 Methodology and further experiments

Compared to backpropagation (BP), predictive coding (PC) allows for more flexibility in the definition, training, and evaluation of the model. The reported experiments show the best results achieved on each specific task and, as a consequence, only the effects of a specific set of hyperparameters. Therefore, the complete range of possibilities that exist in PC has not been displayed, however those alternative configurations may be helpful in other scenarios.

5.6.1 Architectures and hyperparameters

In this section, we provide a detailed description of the models and parameters used to obtain the results in the various generation tasks presented in this chapter, to guarantee their

reproducibility. Note that our goal was to compare the performance of different models, hence we compare networks that have a similar number of parameters. We now briefly summarize the PC graphs used in this chapter:

- **Fully connected networks**: The experiments on fully connected networks are obtained by using a fully connected graph with 2000 neurons, trained with 794 sensory neurons for classification and generation tasks (784 pixels plus a 1-hot vector for the 10 labels), and 784 sensory neurons for reconstruction and denoising. For colored images, we used a network with 5000 neurons. We trained every model for 20 epochs, and reported the best results using early stopping. As learning rates, we used $\alpha \in \{1, 0.5\}$ for the value nodes, and $\eta \in \{0.0001, 0.00005\}$ for the weights, and a weight decay $\lambda = \{0.01, 0.001, 0.0001, 0\}$. To conclude, we computed each query using $T = 2000$, making sure that the energy had converged before reaching that value.

- **Feedforward network**: A network composed by a sequence of L fully connected layers of dimension H. The best results were achieved with $L \in \{3, 4\}$ and $H = 512$ for MNIST and $H = 1024$ on FashionMNIST. We did not experience any benefits in adding extra layers, as it only resulted in higher convergence times. The width, instead, directly determines the quality of the images produced: as expected, very narrow networks fail to store enough information to accurately reconstruct (or denoise) the input images. However, wide networks manifest sub-optimal performance as well. This follows, as having more parameters allows the network to easily overfit. As a consequence, the generation process is less stable, and the images can appear more noisy and composed by strokes belonging to different classes. Using a strong weight decay alleviates these problems, as we will later discuss.

- **Recurrent network**: A recurrent layer consists of a layer whose output is transformed by a non-linear transformation and fed in input to the layer. The recurrent networks used consist of two recurrent layers (for a total of four non-linear transformations) with hidden dimension $H = 512$ when trained on MNIST, and $H = 1024$ when trained on FashionMNIST. The behaviour, given the choice of width and depth, seems similar to feedforward networks. The performance, however, seems to be less impacted by the usage of wide layers. This is due to the recurrent connections that establish more constraints, and thus stability.

- **Assembly of neurons**:Here, we used models with 4 clusters with 3000 neurons each, connected in a feedforward way. As sparsity and top-k constants, we used $p = 0.1$ and $k = 0.2$, and performed the same generative experiments. We trained each model for

20 epochs, and reported the best results using early stopping. As learning rates, we used $\alpha \in \{1, 0.5\}$ for the value nodes, and $\eta \in \{0.0001, 0.00005\}$ for the weights. To conclude, we computed each query using $T = 2000$, making sure that the energy had converged before reaching that value.

- **Autoencoders:** The autoencoder was defined using the same shape as the feedforward networks: it is as a fully connected network with $L \in \{3, 4\}$ hidden layers of width $H \in \{256, 512, 1024\}$. In this way, the structure and the number of parameters directly correspond to the feedforward network trained using predictive coding. It was trained through BP using the *Adam* optimizer, with learning rate $\alpha = 1e^{-4}$ and weight decay of parameter $\lambda \in \{1e^{-2}, 1e^{-4}, 1e^{-6}, 0\}$ (the best results were achieved with the lowest value).

As predictive coding requires two sets of updatable parameters, the value nodes $x_{i,t}$ and the weights $\theta_{i,j}$, we defined two separate optimizers. The learning rate for the weights was set to $\alpha = 1e^{-4}$, and the optimizer algorithm chosen was *Adam* (as for the autoencoder). We experimented with different values of weight decays, noticing how the final performance is highly affected by this value. For the given tasks, the best results were achieved with *weight decay* $= 1e^{-2}$. Instead, the learning rate for the value nodes was set to $\gamma = 1.0$, and optimized using *SGD*.

5.6.2 Feedforward vs. recursive networks

In this chapter, we highlighted how in different situations, one may prefer to query by conditioning or by initialization. As a rule of thumb, conditioning means that we expect the partial data given to the network to be correct and be recognized as a *memory*, by being reconstructed by the network without modifications. Therefore, it makes sense to use it in the reconstruction generative task. Instead, when performing image denoising, we do not want the network to recall the noisy image from its memory, instead, we are asking it to retrieve the memory (or to generate a realistic sample), representing a plausible image, that is the closest to the noisy input. It makes therefore sense to only initialize the output layer, giving the network a direction to follow and let it evolve unconstrained. However, it may not always be clear which querying technique is most preferable. A desirable behavior may be using the network to identify which querying data is realistic (i.e., similar to the training samples) and which not. Ideally, we would like the network to perfectly fit previously seen data points, while struggling to reconstruct unfamiliar shapes. We tested both the feedforward and recursive networks by training them on the MNIST dataset and querying

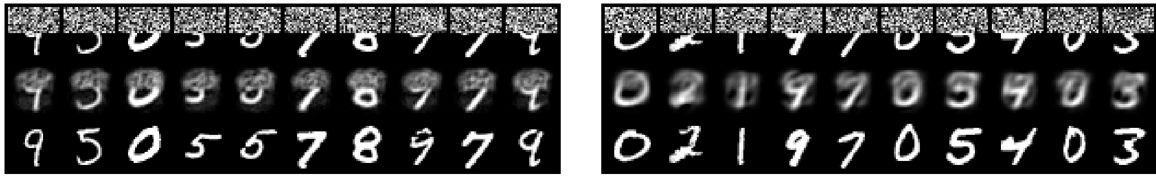


Figure 5.9: Reconstruction using query by conditioning on the whole output layer. The performance of feedforward networks (left) is noticeably improved by using recurrent connections (right), as the reconstructed images do not overfit the noise, but resemble plausible, albeit noisy, digits.

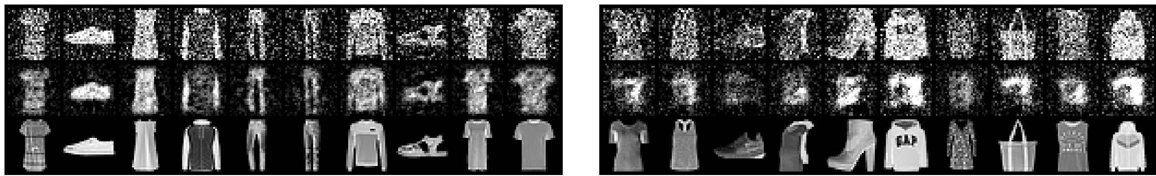


Figure 5.10: Reconstruction using query by conditioning using FashionMNIST samples after training on MNIST. Feedforward networks (left) simply overfit (i.e., reproduce without performing any modification) the input samples, despite being unrelated to the training data. Recurrent networks, instead, reproduce an unrecognizable and shady image, showing that they do not recognize the input samples, as they are not stable data points.

them by conditioning the output layer with a full size image composed by half uniform noise and half digit. The results are reported in Fig. 5.9. We can see how feedforward networks easily fit the noise, reconstructing the two halves independently. On the other hand, employing recurrent connections (and thus imposing stricter constraints) forces the network to reconstruct the image as a whole. We can see a similar behavior in Fig. 5.10, where networks trained on MNIST are use to denoise FashionMNIST images. Feedforward networks easily overfit the input samples. Recurrent networks, instead, correctly do not recognize the given images and reconstruct an unrelated and confused blob. In this last case, it would therefore be possible to distinguish between familiar and unfamiliar images by computing the distance between the input and output images.

5.6.3 Importance of weight decay

As previously mentioned, weight decay plays a fundamental role in determining the properties of the reconstructed images. Compared to other tasks (e.g., classification) or models (e.g., autoencoder trained by BP), a higher value of weight decay seems to be necessary when training with PC. From our experiments, weight decay prevents the networks from overlearning the task they are trained on (i.e., reproduce any image that they are given in input), and instead allows them to "understand" the several concept classes of each dataset.

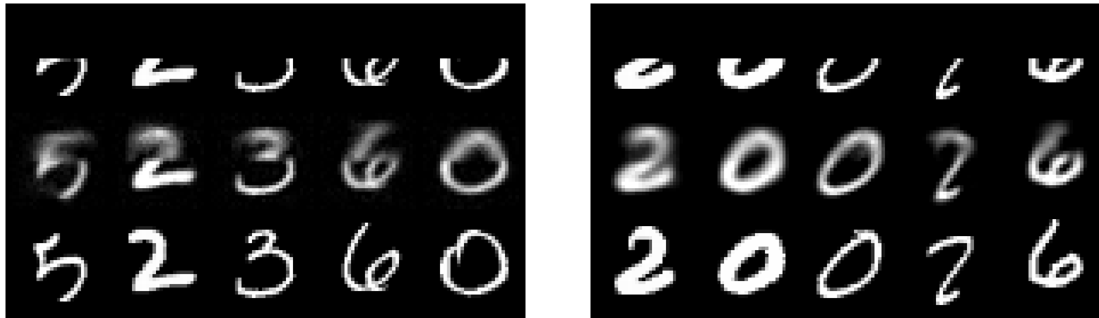


Figure 5.11: Reconstructed images given the label and by conditioning the bottom half. Using low weight decay values (left) causes the two halves of the images to be uncorrelated. As a result each digit is composed by almost unrelated lines. Contrarily, with higher values (right), each image is correctly generated.

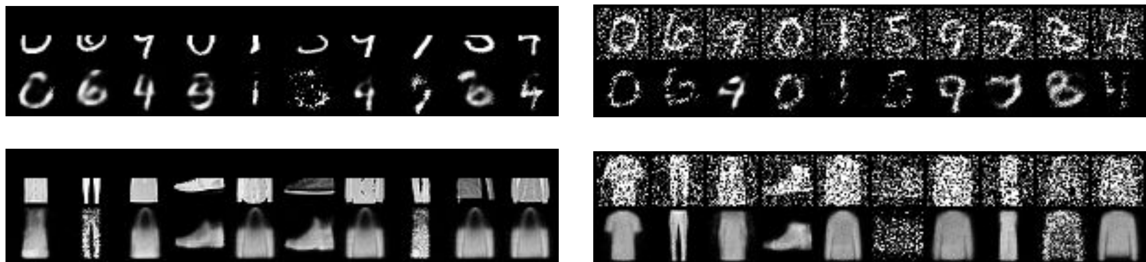


Figure 5.12: Reconstructed and denoised images using RBMs.

This behavior makes it possible to generalize their knowledge to new and unseen tasks, such as the denoising and reconstructing tasks seen in this chapter. It is worth noticing how, when optimizing for a single specific problem (e.g., image recognition), lower values of weight decay seem to be more effective.

To show this, we trained a recurrent network to reconstruct images by conditioning the bottom half of the output layer and giving the target class label in input. The result is that, with low weight decay, the network treats each half of the image independently, reconstructing the bottom part by fitting the conditioning data and the top half using the given label. It can be observed that there is no relation between the two halves. With higher weight decay, instead, we can see that the image is reconstructed as a whole, incorporating both the information provided via the label and the conditioning data (Fig. 5.11).

5.7 Restricted Boltzmann machines

To provide a full comparison between the generation capabilities of our model and existing ones in the literature, we trained a different RBM, and performed both reconstructions and denoising tasks. The results are in Fig. 5.12. Particularly, they show that RBMs sometimes

fail to retrieve the correct image, returning a blurry cloud of points in denoising, and tend to often return the same image even when presented with different inputs in reconstruction ones. This problem was consistent in different batches and parameterizations of RBMs, and had never happened in any of the experiments using PC graphs.

Setup: We trained several RBMs with $h \in \{256, 512, 1024\}$ hidden nodes, and performed $\{1, 2, 5, 10\}$ *Gibbs samplings*. We always picked the best result.

5.8 Summary and discussion

In this chapter, we have shown that PC is able to perform machine learning tasks on graphs of any topology, called PC graphs. Particularly, we have highlighted two main differences between our framework and standard deep learning: flexibility in structure and query. On the one hand, flexible structure allows for learning on any graph topology, hence including both classical deep learning models, and small-world networks that resemble sparse brain regions. On the other hand, flexible query allows the model to be trained and tested on data points that carry different kind of information: supervised signals, unsupervised, and incomplete. To this end, we have first proposed a proof-of-concept experiment, given by the fully connected PC graph. While not accurate enough to be used for practical tasks, the experiments proposed fully highlight the flexibility of the model. Then, we show that it suffices to act on the structure of the graph to improve the performance: it suffices to use an hierarchical model to match the performance of BP on both generation and classification tasks. To this end, we have proposed and explored multiple architectures, such as standard hierarchical models, models with lateral connections, and brain-like architectures, such as the one proposed by Papadimitriou [48]. This could inspire a new direction of research in the field of neural architecture search, where the space of models to explore is not anymore restricted to that of DAGs.

Chapter 6

Conclusion

In this thesis, and during my DPhil, my main focus has been to advance our knowledge of predictive coding networks, this unique computational model inspired by the neuroscientific theory of predictivce coding, to improve its applicability in machine learning. As every research, mine has the far-reaching goal of addressing an important, open questions in the field. Particularly, I have focused on improving the state of the art of predictive coding models, as well as providing a better understanding of some interesting properties of this class of algorithms. This fits well into the general literature of predictive coding applied to machine learning, that has recently seen an impressive growth: it was only in 2017 that Whittington and Bogacz showed how to train a small predictive coding classifier on MNIST [22], as well as an approximation result for MLPs that only holds under very specific conditions. A couple of years later, these models, or variations, are able to perform classification tasks on large-scale datasets [25].

6.1 Summary

This thesis has focused on three aspects of predictive coding networks: generalization, memorization, and flexibility. The generalization results are theoretical, and show that predictive coding networks are able to perform as well as BP in supervised learning tasks on any neural network, due to their ability to exactly replicate its learning rule. The results on memorization, on the other hand, are empirical, and show the excellent memory storage and retrieval robustness of predictive coding. To conclude, I have shown that predictive coding networks are much more flexible than standard networks when it comes to both the choice of network structure, and training procedure, as it allows to query the model in multiple ways by simply running inference.

What I personally found interesting and, initially, surprising about the predictive coding literature, is that all of these results are achieved by following a framework that is quite

simple, as every step, no matter whether it is inference or learning, is governed by the same energy function. This energy is nothing more than the sum of functions of the total error of every neuron of the network. In other words, predictive coding is able to derive the different concepts of perception and learning from the same, general, principle, with excellent results when tested on machine learning benchmarks.

6.2 Outlook

In the previous chapter of this thesis, I have shown that predictive coding is able to perform machine learning tasks on graphs of any topology. Particularly, I have highlighted two main differences between our framework and standard deep learning: flexibility in structure and query. A flexible structure allows learning on arbitrary graph topologies, hence including both classical deep learning models, and small-world networks that resemble sparse brain regions. A flexible query allows the model to be trained and tested on data points that carry different kind of information: supervised signals, unsupervised, and incomplete. This introduces a novel learning framework that is equivalent to prior works on standard predictive coding when tested on feedforward architectures, but it is also able to generalize to different research directions, that I now summarize.

- Predictive coding has a superior flexibility compared to standard deep learning models. This means that predictive coding networks can flexibly perform different inference tasks given the same network (while standard models would have to be trained separately for each task), and also allows predictive coding networks to naturally handle missing data, while standard models need heuristic imputation schemes. How can we use this superior flexibility to our advantage?

- A second promising direction to explore is then that of neural architecture search algorithms. This is an already developed field in standard machine learning, but would now not be limited to the space of functions expressed as hierarchical directed acyclic computational graphs, but to any graph topology. Connecting this with the results of Chapters 3 and 4, a natural question that arises is whether it is possible to merge the generalization capabilities of feedforward, top-down networks, similar to that of backpropagation, with the robustness of generative, bottom-up networks. This would allow to design networks that are good in both discriminative tasks, and in detecting out-of-distribution data points.

- Connected to the previous point, another interesting question is transfer learning, one of the most used industrial applications of deep learning. Transfer learning works as follows: given a pre-trained network $f : \mathbb{R}^d \to \mathbb{R}^{d_i}$, it is possible to glue a second neural network $g : \mathbb{R}^{d_i} \to \mathbb{R}^k$ to the output layer, that is trained on the outputs of f. Inference is then performed on the composition $g \circ f$. However, this approach is limited to the space of functions $\mathbb{R}^d \to \mathbb{R}^k$. Now that we have seen that it is possible to train neural networks with any topology, we are not limited in "gluing" networks one on top of each other. How does the landscape of transfer learning change?

On a much broader level, the work of this thesis strengthens the connection between the machine learning and the neuroscience communities, as it underlines the importance of predictive coding in both areas, both as a highly plausible algorithm to train brain-inspired architectures, and as an approach to solve corresponding problems in machine intelligence.

6.2.1 Scaling up predictive coding

While current research has shown that predictive coding is closely connected to backpropagation, and that predictive coding networks can often match the performance of classic models on a variety of machine learning benchmarks, there are no cases where predictive coding networks perform demonstrably *better* than comparable models. Thus, at present, predictive coding is not yet used in any industrial application. However, the promising properties highlighted in this thesis suggest that this may change in the next years. Hence, future research should be directed at finding applications where the unique properties of predictive coding can be utilized to outperform existing methods, as well as theoretical research exploring these properties further. All in all, the goal of my personal research, and the one pursued by other scientists working on the same topic, aims to answer the following question:

How can we improve predictive coding models up to the point that they will be used in large-scale and industrial applications?

It is worth noting that essentially all current benchmarks, layers, initializations, and other "tricks", have been heavily optimized specifically for standard deep learning, while no such optimization work has been performed for predictive coding networks. This indicates that an important line of future research will be to devise similar "tricks" to improve the performance of predictive coding networks. In fact, also backpropagation showed many flaws at the beginning that did not allow it to scale to deep architectures: in the late 90's, kernel learning methods were more used and effective than neural networks, which suffered

the vanishing gradient problem and the need of an heavy computational power. Most of these problems have now been addressed in over thirty years of research, by a collective effort of thousands of people, from diverse disciplines, and coming from both the public and private sector. Some problems, for example, have been solved by the development of tricks such as dropout and batch normalization, that make learning on huge networks possible. What are the "dropout" and "batch norm" equivalents for predictive coding? Addressing these questions is of vital importance to scale the applicability of these models, and should hence be topic of interest of hopefully more and more research.

6.2.2 New hardware?

None of the large-scale applications of deep learning we have today would have been possible without significant progress on the hardware side. Hardware limitations strongly influence research directions, as they are vital to exploit the full potential of the theoretical results. In fact, multiple ideas have been abandoned when the available hardware was too slow [128]. As for today, GPUs are still the only available hardware to train neural networks in practice, but new exotic technologies, such as memristor, spintronics, and optics, may soon change the landscape [129, 130, 131]. Developing algorithms that well adapt to these changes is a key direction for future progress.

Predictive coding is also promising in this regards, as an additional unique property of these models, shared by most of neuroscience-inspired models, is the locality of their weight updates and hence great parallelizability. If exploited properly, this may enable a substantially more efficient training of extremely large-scale PCN models by reducing the communication and wait time requirements induced by the sequential backward step of backpropagation, as well as lend itself to an efficient implementation on analog and/or neuromorphic hardware, which excels in energy-based learning methods that use local information for the updates [32].

Analog Computing: The main problem with GPUs is that they are time and energy consuming due to the *Von Neumann bottleneck*, i.e., the separation of memory and processing that forces to move data back and forth between memory and compute units. To solve this problem, we need a non-Von Neumann computing paradigm, where memory and processing are unified in the same computational paradigm. In recent years, research on such computational frameworks has attracted a large amount of funding due to its potential. As an example, an analog variant of equilibrium propagation has been simulated on analog hardware [32]. This energy-based learning algorithm introduced a couple of years ago [132], is ideal to train deep networks on analog hardware due to the energy minimization

framework: it is in fact possible to train neural network in an end-to-end fashion, where the whole network simultaneously fits into the same analog chip.

6.2.3 Implications to neuroscience

While not being the primary focus of my research, here I briefly discuss how the work explored in this thesis may have future implications in the field of computational neuroscience. Particularly, there is a long-lasting direction of research that aims to understand how the brain processes information via neuron activities and synapses. Such rule is called *credit assignment*.

Memories in the brain I have introduced a novel associative memory model, inspired by a recent hypothesis that describes the functioning of the memory system in our brain. This model achieves memory retrieval with unprecedented robustness: tested on natural images, it is able to retrieve ImageNet pictures when presented with corrupted keys of stored memories, even when the level of corruption does not allow us humans to infer the original data point. However, this model could also benefit research in neuroscience, as it provides a working computational model of the memory system introduced, only theoretically, in [6]. Future work with implication to neuroscience in this direction would be to collect data from visual cortex for a learning and memory task, as well as hippocampal data for the same task, and drawing some predictions from an hierarchical associative memory model that could be tested with on this data.

Backpropagation in the brain While it is clear that credit assignment is not performed using backpropagation, how different are the two? Multiple works [22, 24, 26, 28] have shown that backpropagation in artificial neural networks can be implemented in a more biologically plausible way using predictive coding, providing previously missing evidence to the debate on whether backpropagation could describe credit assignment in the brain [133]. In fact, for multiple reasons, backpropagation is not compatible with the anatomy of the brain and physiology, particularly in the neocortex. However, different biologically plausible implementations have been shown to be able to approximate it, opening the question of how accurate can simulations of brain behaviors be when these models are trained using backpropagation. Particularly, does it make sense to develop backpropagation-based models to study neural activities? Predictive coding is not the only framework that pushes to a positive answer. Other popular theories that are able to approximate backpropagation with biologically plausible constraints are equilibrium propagation [132, 134, 135] and feedback alignment [136, 137, 138, 139].

Connections to artificial learning Overall, this thesis takes biological learning as an inspiration to improve artificial learning, with a specific focus on one model of biological learning, called predictive coding. Despite the fact that modern neural networks have achieved unprecedented results and success in dozens of tasks, it is still to be proved that backpropagation is the best method to perform credit assignment in artificial neural networks. It could be, for example, that a better credit assignment is obtained using a simpler framework, where every parameter is locally updated via a global energy function, able to enclose both perception and learning.

Bibliography

[1] V. Vapnik, E. Levin, and Y. L. Cun, “Measuring the VC-dimension of a learning machine,” *Neural Computation*, vol. 6, no. 5, pp. 851–876, 1994.

[2] P. L. Bartlett and S. Mendelson, “Rademacher and Gaussian complexities: Risk bounds and structural results,” *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 463–482, 2002.

[3] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” in *5th International Conference on Learning Representations, ICLR*, 2017.

[4] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio, and S. Lacoste-Julien, “A closer look at memorization in deep networks,” *Proceedings of the 34 th International Conference on Machine Learning*, 2017.

[5] D. Krueger, N. Ballas, S. Jastrzebski, D. Arpit, S. Kanwal, T. Maharaj, E. Bengio, A. Fischer, and A. Courville, “Deep nets don’t learn via memorization,” *5th International Conference on Learning Representations, ICLR - Workshop track*, 2017.

[6] H. C. Barron, R. Auksztulewicz, and K. Friston, “Prediction and memory: A predictive coding account,” *Progress in Neurobiology*, vol. 192, p. 101821, 2020.

[7] P. Domingos, “Every model learned by gradient descent is approximately a kernel machine,” *arXiv:2012.00152*, 2020.

[8] T. Poggio, “From associative memories to powerful machines,” *CBMM Memo No. 114*, 2021.

[9] A. Radhakrishnan, M. Belkin, and C. Uhler, “Overparameterized neural networks implement associative memory,” *Proceedings of the National Academy of Sciences*, 2019.

[10] V. Feldman and C. Zhang, "What neural networks memorize and why: Discovering the long tail via influence estimation," *ArXiv:2008.03703*, 2020.

[11] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: convergence and generalization in neural networks," *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, Jun 2021.

[12] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, 2008.

[13] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, 1982.

[14] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences*, vol. 81, 1984.

[15] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, L. Gruber, M. Holzleitner, T. Adler, D. Kreil, M. K. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter, "Hopfield networks is all you need," in *International Conference on Learning Representations*, 2021.

[16] D. Krotov and J. J. Hopfield, "Unsupervised learning by competing hidden units," *Proceedings of the National Academy of Sciences*, vol. 116, no. 16, pp. 7723–7731, 2019.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[18] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "MLP-Mixer: An all-MLP architecture for vision," *Annual Conference on Neural Information Processing Systems*, 2021.

[19] D. Mumford, "On the computational architecture of the neocortex," *Biological cybernetics*, vol. 66, no. 3, pp. 241–251, 1992.

[20] R. P. Rao and D. H. Ballard, "Predictive coding in the visual cortex: A functional interpretation of some extra-classical receptive-field effects," *Nature Neuroscience*, vol. 2, no. 1, pp. 79–87, 1999.

[21] K. Friston, "The free-energy principle: a unified brain theory?," *Nature reviews neuroscience*, vol. 11, no. 2, pp. 127–138, 2010.

[22] J. C. Whittington and R. Bogacz, "An approximation of the error backpropagation algorithm in a predictive coding network with local Hebbian synaptic plasticity," *Neural Computation*, vol. 29, no. 5, 2017.

[23] A. Ororbia and D. Kifer, "The neural coding framework for learning generative models," *arXiv:2012.03405*, 2020.

[24] Y. Song, T. Lukasiewicz, Z. Xu, and R. Bogacz, "Can the brain do backpropagation? — Exact implementation of backpropagation in predictive coding networks," in *Advances in Neural Information Processing Systems*, 2020.

[25] K. Han, H. Wen, Y. Zhang, D. Fu, E. Culurciello, and Z. Liu, "Deep predictive coding network with local recurrent processing for object recognition," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[26] B. Millidge, A. Tschantz, and C. L. Buckley, "Predictive coding approximates backprop along arbitrary computation graphs," *arXiv:2006.04182*, 2020.

[27] A. G. Ororbia and A. Mali, "Biologically motivated algorithms for propagating local target representations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4651–4658, 2019.

[28] T. Salvatori, Y. Song, T. Lukasiewicz, R. Bogacz, and Z. Xu, "Reverse differentiation via predictive coding," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[29] T. Salvatori, Y. Song, Y. Hong, L. Sha, S. Frieder, Z. Xu, R. Bogacz, and T. Lukasiewicz, "Associative memories via predictive coding," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[30] R. P. Rane, E. Szügyi, V. Saxena, A. Ofner, and S. Stober, "Prednet and predictive coding: A critical review," in *Proceedings of the 2020 International Conference on Multimedia Retrieval*, 2020.

[31] W. Lotter, G. Kreiman, and D. Cox, "Deep predictive coding networks for video prediction and unsupervised learning," *International Conference on Learning Representations, ICLR*, 2017.

[32] J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier, "Training end-to-end analog neural networks with equilibrium propagation," *arXiv:2006.01981*, 2020.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *26th Annual Conference on Neural Information Processing Systems*, 2012.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[35] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *International Conference on Learning Representations, ICLR 2021*, 2021.

[36] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *International conference on acoustics, speech and signal processing*, 2013.

[37] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, 2017.

[38] M. Moravčík, M. Schmid, N. Burch, V. Lisỳ, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, 2017.

[39] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, 2019.

[40] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019*

Conference of the North American Chapter of the Association for Computational Linguistics., Association for Computational Linguistics, 2019.

[41] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, 2020.

[42] P. Dhar, "The carbon impact of artificial intelligence," *Nature Machine Intelligence*, vol. 2, no. 8, pp. 423–425, 2020.

[43] D. C. Van Essen, C. H. Anderson, and D. J. Felleman, "Information processing in the primate visual system: an integrated systems perspective," *Science*, vol. 255, 1992.

[44] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[45] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *ICML Deep Learning Workshop*, 2015.

[46] C. J. Spoerer, T. C. Kietzmann, J. Mehrer, I. Charest, and N. Kriegeskorte, "Recurrent neural networks can explain flexible trading of speed and accuracy in biological vision," *PLoS computational biology*, 2020.

[47] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, 2014.

[48] C. H. Papadimitriou, S. S. Vempala, D. Mitropolsky, M. Collins, and W. Maass, "Brain computation by assemblies of neurons," *Proceedings of the National Academy of Sciences*, vol. 117, no. 25, 2020.

[49] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[50] D. Hebb, "The organization of behavior," 1949.

[51] M. V. Srinivasan, S. B. Laughlin, and A. Dubs, "Predictive coding: a fresh view of inhibition in the retina," *Proceedings of the Royal Society of London. Series B. Biological Sciences*, vol. 216, 1982.

[52] K. Friston, "Learning and inference in the brain," *Neural Networks*, vol. 16, 2003.

[53] K. Friston, "A theory of cortical responses," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 360, no. 1456, 2005.

[54] J. M. Kilner, K. J. Friston, and C. D. Frith, "Predictive coding: an account of the mirror neuron system," *Cognitive processing*, vol. 8, no. 3, pp. 159–166, 2007.

[55] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[56] T. Lillicrap, A. Santoro, L. Marris, C. Akerman, and G. Hinton, "Backpropagation and the brain," *Nature Reviews Neuroscience*, vol. 21, 2020.

[57] F. Crick *et al.*, "The recent excitement about neural networks," *Nature*, vol. 337, no. 6203, pp. 129–132, 1989.

[58] S. Grossberg, "How does a brain build a cognitive code?," in *Studies of mind and brain*, pp. 1–52, Springer, 1982.

[59] J. Orchard, W. Sun, and N. Liu, "Why aren't all predictive coding networks generative?," in *Neural Information Processing Systems*, 2019.

[60] A. Ali, N. Ahmad, E. de Groot, M. A. van Gerven, and T. C. Kietzmann, "Predictive coding is a consequence of energy efficiency in recurrent neural networks," *bioRxiv:16.430904*, 2021.

[61] Y. Song, "Predictive coding inspires effective alternatives to backpropagation," *PhD Thesis , Univ. of Oxford*, 2022.

[62] A. Avena-Koenigsberger, B. Misic, and O. Sporns, "Communication dynamics in complex brain networks," *Nature Reviews Neuroscience*, vol. 19, no. 1, pp. 17–33, 2018.

[63] Y. LeCun and C. Cortes, "MNIST handwritten digit database," *The MNIST Database*, 2010.

[64] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," *arXiv:1708.07747*, 2017.

[65] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[66] M. Beal, *Variational algorithms for approximate Bayesian inference*. University College London, 2003.

[67] M. Wainwright and M. I. Jordan, *Graphical models, exponential families, and variational inference*. Now Publishers Inc, 2008.

[68] M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul, "An introduction to variational methods for graphical models," in *Learning in graphical models*, Springer, 1998.

[69] K. Friston, J. Mattout, N. Trujillo-Barreto, J. Ashburner, and W. Penny, "Variational free energy and the Laplace approximation," *Neuroimage*, 2007.

[70] C. Buckley, C. S. Kim, S. McGregor, and A. Seth, "The free energy principle for action and perception: A mathematical review," *Journal of Mathematical Psychology*, 2017.

[71] T. Salvatori, L. Pinchetti, B. Millidge, Y. Song, T. Bao, R. Bogacz, and T. Lukasiewicz, "Learning on arbitrary graph topologies via predictive coding," *arXiv:2201.13180*, 2022.

[72] K. Friston, "Hierarchical models in the brain," *PLoS Computational Biology*, 2008.

[73] N. Elsayed, A. S. Maida, and M. Bayoumi, "Reduced-gate convolutional lstm architecture for next-frame video prediction using predictive coding," in *International Joint Conference on Neural Networks (IJCNN)*, 2019.

[74] J. Zhong, A. Cangelosi, X. Zhang, and T. Ogata, "Afa-prednet: The action modulation within predictive coding," in *2018 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2018.

[75] J. Zhong, T. Ogata, and A. Cangelosi, "Encoding longer-term contextual multi-modal information in a predictive coding model," *arXiv:1804.06774*, 2018.

[76] R. Sato, H. Kashima, and T. Yamamoto, "Short-term precipitation prediction with skip-connected prednet," in *International Conference on Artificial Neural Networks*, Springer, 2018.

[77] K. Friston, J. Kilner, and L. Harrison, "A free energy principle for the brain," *Journal of Physiology*, 2006.

[78] K. Friston, J. Daunizeau, and S. Kiebel, "Reinforcement learning or active inference?," *PloS One*, 2009.

[79] K. Friston, "What is optimal about motor control?," *Neuron*, 2011.

[80] M. Baltieri and C. Buckley, "PID control as a process of active inference with linear generative models," *Entropy*, 2019.

[81] M. Johnson and M. Moradi, *PID Control.* Springer, 2005.

[82] P. Lanillos and G. Cheng, "Adaptive robot body learning and estimation through predictive coding," in *Proceedings of IROS*, 2018.

[83] A. Meera and M. Wisse, "Free energy principle based state and input observer design for linear systems with colored noise," in *2020 American Control Conference (ACC)*, IEEE, 2020.

[84] A. Meera and M. Wisse, "A brain inspired learning algorithm for the perception of a quadrotor in wind," *arXiv:2109.11971*, 2021.

[85] C. Pezzato, R. Ferrari, and C. Corbato, "A novel adaptive controller for robot manipulators based on active inference," *IEEE Robotics and Automation Letters*, 2020.

[86] G. Oliver, P. Lanillos, and G. Cheng, "An empirical study of active inference on a humanoid robot," *IEEE Transactions on Cognitive and Developmental Systems*, 2021.

[87] C. Sancaktar, M. van Gerven, and P. Lanillos, "End-to-end pixel-based deep active inference for body perception and action," in *2020 Joint IEEE 10th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, IEEE, 2020.

[88] P. Lanillos, C. Meo, C. Pezzato, A. Meera, M. Baioumy, W. Ohata, A. Tschantz, B. Millidge, M. Wisse, and C. Buckley, "Active inference in robotics and artificial agents: Survey and challenges," *arXiv:2112.01871*, 2021.

[89] R. Bogacz, "Dopamine role in learning and action inference," *Elife*, 2020.

[90] P. Kinghorn, B. Millidge, and C. Buckley, "Habitual and reflective control in hierarchical predictive coding," *arXiv:2109.00866*, 2021.

[91] K. J. Friston, T. Parr, and B. de Vries, "The graphical brain: Belief propagation and active inference," *Network Neuroscience*, vol. 1, no. 4, pp. 381–414, 2017.

[92] B. Millidge, A. Tschantz, A. Seth, and C. Buckley, "On the relationship between active inference and control as inference," in *International Workshop on Active Inference*, Springer, 2020.

[93] H. Attias, "Planning by probabilistic inference," in *International Workshop on Artificial Intelligence and Statistics*, PMLR, 2003.

[94] M. Toussaint and A. Storkey, "Probabilistic inference for solving discrete and continuous state Markov decision processes," in *Proceedings of ICML*, 2006.

[95] S. Levine, "Reinforcement learning and control as probabilistic inference: Tutorial and review," *arXiv:1805.00909*, 2018.

[96] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering," in *Proceedings of the 24th International Conference on Machine Learning*, 2007.

[97] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, 1997.

[98] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, 2016.

[99] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, 1989.

[100] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, 1997.

[101] A. Sedghi, V. Gupta, and P. Long, "The singular values of convolutional layers," in *Proceedings of the Conference ICLR*, 2020.

[102] L. Squire and S. Zola-Morgan, "The medial temporal lobe memory system," *Science*, vol. 253, no. 5026, 1991.

[103] D. J. Felleman and D. C. Van Essen, "Distributed hierarchical processing in the primate cerebral cortex.," *Cerebral cortex*, pp. 1–47, 1991.

[104] E. A. Murray, T. J. Bussey, and L. M. Saksida, "Visual perception and memory: a new view of medial temporal lobe function in primates and rodents," *Annual Review of Neuroscience*, vol. 30, pp. 99–122, 2007.

[105] J. Mcclelland, B. Mcnaughton, and R. O'Reilly, "Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory," *Psychological review*, vol. 102, pp. 419–57, 08 1995.

[106] K. L. Stachenfeld, M. M. Botvinick, and S. J. Gershman, "The hippocampus as a predictive map," *Nature Neuroscience*, vol. 20, no. 11, p. 1643, 2017.

[107] D. Krotov and J. J. Hopfield, "Dense associative memory for pattern recognition," in *Advances in Neural Information Processing Systems*, 2016.

[108] K. Liu, J. Sibille, and G. Dragoi, "Generative predictive codes by multiplexed hippocampal neuronal tuplets," *Neuron*, vol. 99, no. 6, pp. 1329–1341.e6, 2018.

[109] D. Krotov and J. J. Hopfield, "Large associative memory problem in neurobiology and machine learning," in *International Conference on Learning Representations*, 2021.

[110] L. A. Pineda, G. Fuentes, and R. Morales, "An entropic associative memory," *Scientific Reports*, vol. 11, no. 1, pp. 1–15, 2021.

[111] K. Steinbuch, "Die Lernmatrix," *Kybern.*, vol. 1, no. 1, pp. 36–45, 1961.

[112] M. Demircigil, J. Heusel, M. Löwe, S. Upgang, and F. Vermet, "On a model of associative memory with huge storage capacity," *Journal of Statistical Physics*, vol. 168, 2017.

[113] S. Bartunov, J. W. Rae, S. Osindero, and T. P. Lillicrap, "Meta-learning Deep Energy-based Memory Models," *arXiv:1910.02720*, 2019.

[114] M. Widrich, B. Schäfl, H. Ramsauer, M. Pavlović, L. Gruber, M. Holzleitner, J. Brandstetter, G. K. Sandve, V. Greiff, and S. Hochreiter, "Modern Hopfield Networks and Attention for Immune Repertoire Classification," *arXiv:2007.13505*, 2020.

[115] J. Liu, M. Gong, and H. He, "Deep associative neural network for associative memory based on unsupervised representation learning," *Neural Networks*, vol. 113, pp. 41–53, 2019.

[116] G. Yang and F. Ding, "Associative memory optimized method on deep neural networks for image classification," *Information Sciences*, vol. 533, 2020.

[117] S. Chartier and M. Boukadoum, "A bidirectional heteroassociative memory for binary and grey-level patterns," *IEEE Transactions on Neural Networks*, vol. 17, no. 2, pp. 385–396, 2006.

[118] B. A. Plummer, L. Wang, C. M. Cervantes, J. C. Caicedo, J. Hockenmaier, and S. Lazebnik, "Flickr30k entities: Collecting region-to-phrase correspondences for richer image-to-sentence models," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[119] M. Dabagia, C. H. Papadimitriou, and S. S. Vempala, "Assemblies of neurons can learn to classify well-separated distributions," *arXiv:2110.03171*, 2021.

[120] E. Oja, "Principal components, minor components, and linear neural networks," *Neural Networks*, vol. 5, no. 6, pp. 927–935, 1992.

[121] E. Oja, "PCA, ICA, and nonlinear Hebbian learning," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN 1995)*, pp. 89–94, 1995.

[122] S. Roweis and Z. Ghahramani, "A unifying review of linear Gaussian models," *Neural Computation*, vol. 11, no. 2, pp. 305–345, 1999.

[123] L. B. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment," *Artificial Neural Networks: Concept Learning*, pp. 102–111, 1990.

[124] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical Review Letters*, vol. 59, 1987.

[125] M. A. Belyaev and A. A. Velichko, "Classification of handwritten digits using the Hopfield network," in *IOP Conference Series: Materials Science and Engineering*, no. 5 in 052048, IOP Publishing, 2020.

[126] A. Ororbia and D. Kifer, "The neural coding framework for learning generative models," *arXiv:2012.03405*, 2020.

[127] Q. K. Telesford, K. E. Joyce, S. Hayasaka, J. H. Burdette, and P. J. Laurienti, "The ubiquity of small-world networks," *Brain Connectivity*, vol. 1, no. 5, pp. 367–375, 2011.

[128] Y. LeCun, “Deep learning hardware: Past, present, and future,” in *IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 12–19, 2019.

[129] L. Chua, “Memristor-the missing circuit element,” *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[130] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[131] J. Grollier, D. Querlioz, K. Camsari, K. Everschor-Sitte, S. Fukami, and M. D. Stiles, “Neuromorphic spintronics,” *Nature electronics*, vol. 3, no. 7, pp. 360–370, 2020.

[132] B. Scellier and Y. Bengio, “Equilibrium propagation: Bridging the gap between energy-based models and backpropagation,” *Frontiers in Computational Neuroscience*, vol. 11, p. 24, 2017.

[133] J. C. Whittington and R. Bogacz, “Theories of error back-propagation in the brain,” *Trends in Cognitive Sciences*, 2019.

[134] M. Ernoult, F. Normandin, A. Moudgil, S. Spinney, E. Belilovsky, I. Rish, B. Richards, and Y. Bengio, “Towards scaling difference target propagation by learning backprop targets,” *arXiv:2201.13415*, 2022.

[135] B. Scellier, A. Goyal, J. Binas, T. Mesnard, and Y. Bengio, “Generalization of equilibrium propagation to vector field dynamics,” *arXiv:1808.04873*, 2018.

[136] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, “Random synaptic feedback weights support error backpropagation for deep learning,” *Nature Communications*, vol. 7, no. 1, pp. 1–10, 2016.

[137] Q. Liao, J. Z. Leibo, and T. Poggio, “How important is weight symmetry in backpropagation?,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

[138] A. Nøkland, “Direct feedback alignment provides learning in deep neural networks,” in *Advances in Neural Information Processing Systems*, 2016.

[139] A. Nøkland and L. H. Eidnes, “Training neural networks with local error signals,” *arXiv:1901.06656*, 2019.