Associative Memories and Predictive Coding



1047773

Anonymous

University of Oxford

Supervisor: Anonymous

Msc Computer Science

Trinity 2021

Abstract

Since 1982, backpropagation (BP) has been the core of machine learning algorithms. However, its biological plausibility is widely disputed. Because BP requires information not local (loss function) to the neurons in order to update their connections, it does not satisfy synaptic plasticity, e.g. weight updates should be local, like in the brain. In this work, we analyze a promising biologically plausible model widely used in neuroscience, predictive coding networks (PCNs).

Concretely, we study this model with respects to an important ability of the human brain, the ability to store and retrieve patterns, or associative memories (AM) in short. There are two types of AM, auto-associative memories which stores and recovers the same pattern from a corrupted version, and hetero-associative memories which relates one type of pattern with another.

PCNs are trained via predictive coding, an error based algorithm that updates the connections between neurons based on local information only, inspired by learning in the cortex. I propose a novel neural model based on PCNs that is able to implement both types of AM, and compare its performance with BP autoencoders.

In general, in auto-associative memory experiments, I found that the proposed model outperforms BP autoencoders in terms of number of memories stored and retrieved. In particular, in completing partial data points, this model achieves remarkable results on natural image datasets, such as ImageNet, with a surprisingly high accuracy, even when only a tiny fraction of pixels of the original images are presented. In hetero-associative memory experiments, I found that the model was able to recover close to 100 images (about 3000 pixels) when captions (about 30 in size) of said images are presented.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 2

1.3 Contributions 3

1.4 Structure of the Document 4

2 Background **5**

2.1 Backpropagation (BP) 5

2.2 Predictive Coding Networks (PCNs) 7

2.2.1 Inference Learning (IL) 9

2.2.2 Zero-divergence Inference Learning (Z-IL) 10

2.2.3 Concurrent Predictive Coding (CPC) 11

2.3 Associative Memory 13

2.3.1 Hopfield Networks 14

2.3.2 Modern Hopfield Networks (Dense Associative Memories) . . . 17

2.3.3 Overparameterized Neural Networks 19

3 Methodology **21**

3.1 Data 21

3.2 Backpropagation 23

3.3 PCNs 27

3.4 Dense Associative Memory 29

3.5 General Procedure 30

4 Implementation **32**

4.1 Software Packages Used 32

4.1.1 Pytorch 32

4.1.2 General Energy Nets 33

4.1.3 Hopfield Layers 36

4.1.4 Other Packages 36
4.2 Datasets 37
4.3 Software Development 40
4.3.1 Code structure 40

5 Experimental Results and Analysis 42
5.1 AA Memory Experiments 42
5.1.1 MNIST Experiments 42
5.1.2 CIFAR10 Experiments 44
5.1.3 TinyImageNet Experiments 47
5.1.4 Imagenet Experiments 50
5.1.5 Dense Associative Memory (DAM) experiments 51
5.2 HA Memory Experiments 53
5.2.1 Network Depth Comparison 54
5.2.2 Greyscale Images Versus Colored Images 58
5.2.3 Increasing Image Size 59
5.2.4 Normalization Scale 60
5.2.5 Comparison with BP Networks 61
5.2.6 Analysis of the Recovery Process 62

6 Conclusions 64

Bibliography 67

A Auto Associative Memory Results 71

B Hetero Associative Memory Results 76

List of Figures

2.1 An illustration of a multi-layer perceptron in BP. 6
2.2 The neural dynamics of PCNs, taken from [21]. 8
2.3 Illustration of a Hopfield Network. 15
2.4 Image retrieval using traditional Hopfield network 16

3.1 Illustration the backpropagation model used. 25
3.2 Illustration of a two layer PCN. 28

4.1 Illustration of how to use PC Layer as part of a Pytorch model 34
4.2 Illustration of how to use PC trainer to train a PC model 35
4.3 Illustration of how to implement a Hopfield network. 37

5.1 10 MNIST digits recovered from 50% of original images. 43
5.2 10 MNIST digits recovered from 25% of original image. 44
5.3 10 MNIST digits recovered from noisy images. 44
5.4 10 CIFAR10 images recovered from noisy images 45
5.5 Percentage of correctly retrieved CIFAR10 images by depth 1 networks. 45
5.6 Percentage of correctly retrieved Imagenet images by depth 1 networks. 47
5.7 Percentage of correctly retrieved partial Imagenet images by depth 1 networks. 48
5.8 Reconstructions of two Imagenet images 49
5.9 Percentage of correctly recovered partial Imagenet images for networks of different depth. 49
5.10 4 reconstructed Imagenet images. 50
5.11 50 reconstructed images by DAM. 51
5.12 Number of recovered captions and images for PCNs of different depth. 56
5.13 10 recovered $3 \times 32 \times 32$ images from captions. 56
5.14 25 32×32 greyscale images recovered from captions. 58
5.15 25 $3 \times 64 \times 64$ images recovered from captions. 59

5.16 Number of recovered captions and images for different normalization scales. 60
5.17 25 $3 \times 32 \times 32$ images recovered by BP autoencoders. 61
5.18 Recovery process of 10 $3 \times 32 \times 32$ images. 63

A.1 20 CIFAR10 images recovered by BP autoencoders. 72

List of Tables

4.1 Caption and vocabulary size. 39

5.1 Number of recovered images for 100 MNIST digits. 43
5.2 Number of recovered captions and images. 55
5.3 Number of recovered captions and greyscale 32×32 images. 58
5.4 Number of recovered captions and recovered images of size $3 \times 64 \times 64$ for networks of depth 11. 59
5.5 Number of recovered captions (left) and recovered images (right) for different caption normalization scales. 60
5.6 Number of recovered captions and images for PCNs and BP. 61

A.1 Number of retrieved $3 \times 32 \times 32$ noisy images by BP autoencoders. . 71
A.2 Number of recovered CIFAR10 noisy images by PCNs. 73
A.3 Number of recovered TInyImageNet noisy images by PCNs. 73
A.4 Number of recovered partial TinyImageNet images by PCNs of different depth. 73
A.5 Number of recovered partial TinyImageNet images by autoencoders of different depth. 74
A.6 Number of recovered partial TinyImageNet images by PCNs of depth1 width 512. 74
A.7 Number of recovered partial TinyImageNet images by PCNs of depth1 width 1024. 74
A.8 Number of recovered partial TinyImageNet images by PCNs of depth1 width 2048. 74
A.9 Number of recovered partial TinyImageNet images by PCNs of depth1 width 4096. 75

B.1 Number of recovered captions and colored 32×32 images for PCNs of depth 3. 76

B.2 Number of recovered captions and colored 32×32 images for PCNs of depth 5. 76
B.3 Number of recovered captions and colored 32×32 images for PCNs of depth 7. 77
B.4 Number of recovered captions and colored 32×32 images for PCNs of depth 9. 77
B.5 Number of recovered captions and colored 32×32 images for PCNs of depth 11. 77
B.6 Number of recovered captions and greyscale Flickr 32×32 images for PCNs of depth 11. 78
B.7 Number of recovered captions and colored Flickr 64×64 images for PCNs of depth 11. 78
B.8 Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 2]$. 79
B.9 Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 5]$. 79
B.10 Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 10]$. 79
B.11 Number of recovered captions and colored Flickr 32×32 images for BP autoencoders of depth 11, with captions normalized to $[0, 1]$. . . . 80

Chapter 1

Introduction

1.1 Motivation

Currently, nearly all state-of-the-art machine learning methods rely on an algorithm that was proposed in 1986: backpropagation (BP) [20]. Despite its huge success, its biological plausibility has been questioned in the recent years [22], such as the lack of local error representation or unrealistic models of neurons. The most criticized aspect is perhaps the way in which BP updates the synaptic weights: these updates depend on an objective function defined on specific neurons. This way, most of the updates depend on non-local information. In contrast, updates in biological synapses only depend on the post-synaptic and pre-synaptic neurons, which BP cannot implement.

In the past decade, there has been numerous research going on regarding biologically plausible models and learning methods that are capable of local computation, such as the free-energy principle [1]. The free-energy principle states that biological systems learn by minimizing an upper bound called free-energy. Based on this principle, predictive coding networks (PCNs) have been proposed, and the resulting biologically plausible algorithm is called *inference learning* (IL) [23]. Inspired by these models, researchers in Oxford, in collaboration with other institutions, have

proposed *zero-divergence inference learning* (Z-IL [21]) and Concurrent Predictive Coding (CPC), more efficient and stable versions of IL.

It is proven that CPC is more efficient than BP, and it can achieve simple classification tasks such as digit classification in MNIST. However, I seek to understand other important properties of the proposed model, and specifically its ability to implement associative memories. Associative memories in the brain receive and store patterns of activity registered by the sensory neurons, and are able to retrieve them when necessary. Due to their importance in human intelligence, computational models of associative memories have been developed for several decades now, for example, it is known that BP autoencoders can implement associative memories (AM) [18]. However, no study has shown if PCNs can implement AM. Moreover, recent work have shown that memorization and learning are related to each other [3] [17], which makes this property even more relevant.

There are two types of associative memory: auto associative memory (AA), which associates the data to itself; and hetero associative memory (HA), which links one type of data to another, e.g. image with text. The task for AA is to recover original patterns from corrupted patterns, and the task in HA is to recover one type of data starting from another. I want to investigate experimentally if the proposed PCN model trained with CPC can also achieve both types of associative memory, and compare the performance of PCNs with BP autoencoders.

1.2 Related Work

The concept of associative memory comes from psychology, and perhaps the most important paper in this area is [7], published in 1982. In this paper, J. J. Hopfield proposed a biologically plausible network capable of memorizing binary data. This network is referred to as the Hopfield network and is often the benchmark for newer

models that implement associative memory. His work was updated and Modern Hopfield Networks was proposed for continuous data [19], also known as Dense Associative Memories (DAM).

After BP was proposed in 1986, other models based on this algorithm emerged. For example, Jia Liu et al. proposed a deep associative neural network [14] with a complex architecture that is able to implement both types of associative memory. Moreover, recent work has also shown that the simplest neural model, multi-layer perceptrons (autoencoders), is also capable of auto-associative memory [18].

1.3 Contributions

The contributions are:

- Showed that PCNs trained with CPC, a biologically plausible model, is able to perform tasks related to auto-associative memory as well as hetero-associative memory.
- Proposed and compared a PCN trained with CPC with BP autoencoders, focusing on their capability to implement auto-associative and hetero-associative memory. Moreover, I showed that PCNs outperforms BP autoencoders at both tasks, in terms of capacity and efficiency.
- Compared CPC with a state-of-the-art associative memory model (DAM), also showing that the proposed network can outperform it.
- Conducted the experiments on several datasets (`MNIST`, `CIFAR10`, `TinyImageNet`, `ImageNet`, `flickr_30k`), and for a wide range of possible parameters. Furthermore, detailed analysis are provided for each dataset.

1.4 Structure of the Document

In Chapter 2, the theoretical background of the used models and training algorithms are presented, as well as a brief introduction to associative memories. In Chapter 3, I show the methodology followed in this work as well as model details. In Chapter 4, implementation details such as external packages used and technologies used are given. In Chapter 5, AA and HA experiments as well as their results are shown, together with a thorough analysis. Finally, in Chapter 6 I conclude the project, together with remarks about possible future work.

More experimental results, such as additional tables and figures are shown in the Appendices 1 and 2.

Chapter 2

Background

2.1 Backpropagation (BP)

Deep neural networks are organized in layers, with different numbers of artificial neurons in each layer. Following the notation used in [21] and [23], we denote by $(\overline{s}^{in}, \overline{s}^{out})$ labeled training points, and by t the time axis during the inference phase. Also, the output layer is indexed by 0, and the input layer by L, contrary to how the layers are normally numbered. Vectors are denoted by a bar, e.g. $\overline{x} = (x_1, x_2, ..., x_d)$, and matrices are represented using boldface symbols. The value contained in the i-th neuron from layer l is $x_{i,t}^l$, where t denotes the training epoch. The number of neurons in each layer is n^l. In Figure 2.1, an illustration of a multi-layer perceptron with this notation is shown.

In the **learning phase**, the weights between layers are updated by first computing the loss function given inputs and outputs, a step that is usually called the forward pass. This is done by first fixing the input layers to $\overline{s}^{in}$, and then computing the values in the intermediate neurons from layer L to layer 0 according to Eq. (2.1). Afterwards, a loss function over the output and the values of the final layer $\ell(\overline{s}^{out}, \overline{x}^0)$ is computed. At each layer, a non-linearity function f is applied.

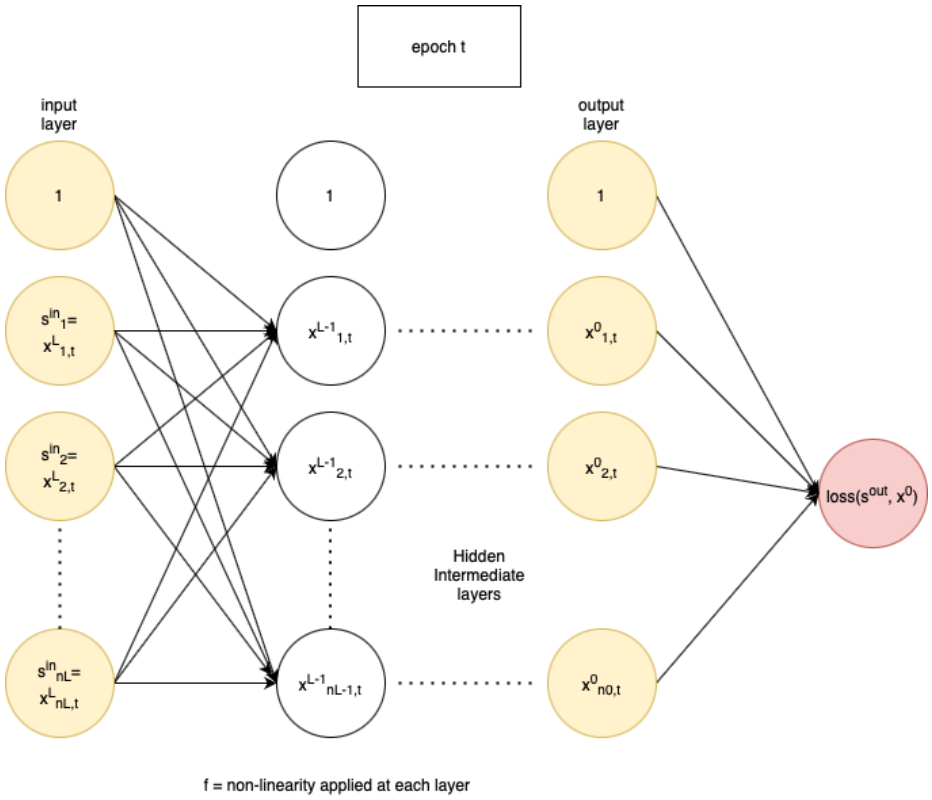


Figure 2.1: An illustration of a multi-layer perceptron in BP.

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

Secondly, the final loss function is derived with respect to the weights, obtaining the gradient and then applying an update according to the chosen optimization rule, e.g., stochastic gradient descent. Given a learning rate α, we have:

$$\Delta\theta^{l+1}_{i,j} = -\alpha \frac{\delta \ell}{\delta \theta^{l+1}_{i,j}} = \alpha \delta^l_i f(x^{l+1}_{j,t}), \tag{2.2}$$

where $\delta^l_i = \frac{\delta \ell}{\delta x^l_{i,t}}$ is the error term given by:

$$\delta^l_{i,t} = \begin{cases} \frac{\delta \ell}{\delta x^0_{i,t}}, & \text{if } l = 0 \\ f'(x^l_{i,t}) \sum_{k=1}^{n^{l-1}} \delta^{l-1}_k \theta^l_{k,i}, & \text{otherwise.} \end{cases} \tag{2.3}$$

This is usually referred to as the backward pass. These two steps are repeated for as many training epochs (weight updates) as necessary.

After the network is trained, we want to use it to predict the output given a certain input. To **predict** using the network we simply fix the input of the neural network to the new data $\overline{s}^{new}$ and do a forward pass to obtain $\overline{x}^0_{new}$, which is referred to as the prediction made by the network. This process can be applied to new, unseen data to predict the output, or to existing data to evaluate the performance of the model by comparing the predicted output and the actual output.

Because the weight updates depend on the loss function, the synaptic update between two neurons depends on information that is not local (ℓ). This rules out BP as a biologically plausible model, as the synaptic update in the human brain only depends on the pre-synaptic and post-synaptic neurons, a phenomenon known as Hebbian Plasticity.

2.2 Predictive Coding Networks (PCNs)

In PCNs, the layers are formed by value nodes, each containing a value $x^l_{i,t}$. Additionally, the value nodes have an associated prediction error node $\varepsilon^l_{i,t}$. The index l indicates layer, and i indicates position in the layer. The difference with BP is that, instead of propagating the neuron activity from lower layers to upper layers directly, PCNs propagate the neuron activity via the error nodes:

$$\mu^l_{i,t} = \sum_{j=1}^{n^{l+1}} \theta^{l+1}_{i,j} f(x^{l+1}_{j,t}) \text{ and } \varepsilon^l_{i,t} = x^l_{i,t} - \mu^l_{i,t}, \tag{2.4}$$

where $\theta^{l+1}_{i,j}$ represents the synapse with the pre-synaptic neuron x^{l+1}_i and post-synaptic neuron x^l_i, and f represents a non linear function. Then, the vector $\overline{\mu}^l_t$ is the input of layer l after layer $l+1$ with activation $\overline{x}^{l+1}_t$, and the error node $\varepsilon^l_{i,t}$ represents the difference between the actual and predicted value of a neuron, i.e., $\varepsilon^l_{i,t} = x^l_{i,t} - \mu^l_{i,t}$.

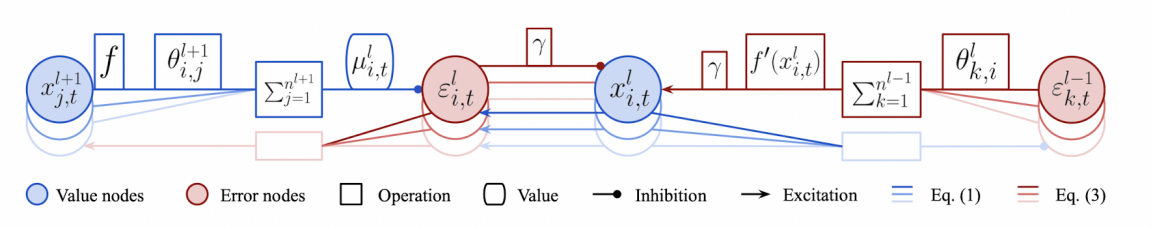


Figure 2.2: The neural dynamics of PCNs, taken from [21].

This architecture is common throughout IL, Z-IL and CPC, see Figure 2.2 for a graphical representation of this architecture.

Predictive coding networks have an associated energy which we intend to minimize:

$$F_t = \frac{1}{2}\sum_{i,l}(\varepsilon_{i,t}^l)^2, \tag{2.5}$$

which is the squared sum of error of the error nodes. Minimizing this value means that we are reducing the error associated with each value neuron. Therefore, each value node $x_{i,t}^l$ tends to move close to $\mu_{i,t}^l$.

Following [23], the derivative used for optimization methods is:

$$\Delta x_{i,t}^l = \begin{cases} 0, & \text{if } l = L \\ \gamma \cdot (-\varepsilon_{i,t}^l + f'(-x_{i,t}^l)\sum_{k=1}^{n^{l-1}} \varepsilon_{k,t}^{l-1}\theta_{k,i}^l), & \text{if } 0 < l < L \\ \gamma \cdot (-\varepsilon_{i,t}^l), & \text{if } l = 0, \end{cases} \tag{2.6}$$

where γ is the integration step. Then, the weights are updated to minimize the same objective function F_t as follows:

$$\Delta\theta_{i,j}^{l+1} = -\alpha \cdot \frac{\delta F_t}{\delta \theta_{i,j}^{l+1}} = \alpha \cdot \varepsilon_{i,t}^l f(x_{j,t}^{l+1}), \tag{2.7}$$

where α is the learning rate.

2.2.1 Inference Learning (IL)

In IL, during the training phase, the input (layer L) and output (layer 0) value nodes are fixed to $\overline{s}^{in}$ and $\overline{s}^{out}$ respectively. Then, the other values nodes $x_{i,t}^{l}$ are modified to minimize the overall energy F_t, the square sum of all error nodes. This process is called inference. Note that the energy does not converge to 0 during inference since the input and output are fixed. When the inference converges (in practice the number of iterations for the inference is capped at some number T), the weights are updated once to minimize the energy function F_t, following Eq. (2.7). This process of inference and weight update constitutes a single epoch, and is repeated for as many epochs as necessary.

During the prediction phase, the input is fixed to $\overline{s}_{in}$, and the error nodes are optimized and decay to 0 as $t \to \infty$. Note that the output of the network is not fixed, therefore, the output $\overline{x}_t^0$ of the network is free to be updated. See Algorithm 1 for IL training algorithm.

Algorithm 1: Learning one training pair $(\overline{s}^{in}, \overline{s}^{out})$ presented for the duration T with IL

Require: $\overline{x}_0^L$ is fixed to $\overline{s}^{in}$, $\overline{x}_0^0$ is fixed to $\overline{s}^{out}$.
for *t = 0 to T* **do**
    for *each neuron i in each level l* **do**
        Update $x_{i,t}^{l}$ to minimize F_t via Eq. (2.6)
        if $t = T$ **then**
            Update each $\theta_{i,j}^{l+1}$ to minimize F_t via Eq. (2.7)
        end
    end
end

2.2.2 Zero-divergence Inference Learning (Z-IL)

Z-IL is a variant of IL that was proposed in [21] in order to improve efficiency and stability. In Z-IL, if γ is set to 1, the synaptic update is equivalent to the one performed by BP. The difference with IL is that Z-IL sets to 0 all error nodes at the beginning of the learning phase, thus, $x^l_{i,0} = \mu^l_{i,0}$ for every neuron. This is equivalent to BP forward pass. Furthermore, Z-IL updates separately the weights of every layer $\theta^l_{i,j}$, at time $t = l$. That is, during time t in the inference process, the weights of the layer $l = t$ is updated. See Algorithm 2 for the summarized algorithm.

Algorithm 2: Learning one training pair $(\overline{s}^{in}, \overline{s}^{out})$ presented for the duration T with Z-IL

Require: $\overline{x}^L_0$ is fixed to $\overline{s}^{in}$, $\overline{x}^0_0$ is fixed to $\overline{s}^{out}$.
Require: $x^l_{i,0} = \mu^l_{i,0}$ for $l \in \{1, ..., L-1\}$ and $\gamma = 1$.
for *t = 0 to T* **do**
    for *each neuron i in each level l* **do**
        Update $x^l_{i,t}$ to minimize F_t via Eq. (2.6)
        if $t = l$ **then**
            Update each $\theta^{l+1}_{i,j}$ to minimize F_t via Eq. (2.7)
        end
    end
end

We can now compare BP, IL and Z-IL in terms of efficiency. Using *matrix multiplication steps* (MMS), the step of computation when multiplying a vector with a weight matrix, we can quantify the time complexity. This is because this operation is the most time consuming one of all algorithms. Furthermore, A simultaneous MMS (SMMS) is used when the matrix multiplications can be parallelized.

Both IL and Z-IL dedicates the majority of computing time running inference, T and $L-1$ inference steps respectively, before performing a single weight update. This

already shows that Z-IL is more efficient than IL, since T is usually in the hundreds and the number of layers are usually in the tens. Moreover, the learning phase of Z-IL is comparable to BP since both perform $2L-1$ MMSs to update the whole network during the learning phase: around L for a forward pass or inference, and $L-1$ for backpropagating the error and updating weights.

For BP, IL and Z-IL, no matrix multiplications can be parallelized. Note that when we parallelize the computation in multi-layer perceptrons, it is only done for batch parallelization. It is not possible to parallelize the internal matrix multiplications during forward or backward pass.

2.2.3 Concurrent Predictive Coding (CPC)

Algorithm 3: Learning one training pair $(\overline{s}^{in}, \overline{s}^{out})$ presented for the duration T with Z-IL

1 **Require:** $\overline{x}_0^L$ is fixed to $\overline{s}^{in}$, $\overline{x}_0^0$ is fixed to $\overline{s}^{out}$.
2 **Require:** $x_{i,0}^l = \mu_{i,0}^l$ for $l \in \{1, ..., L-1\}$ and $\gamma = 1$.
3 **for** *$t = 0$ to T* **do**
4 **for** *each neuron i in each level l* **do**
5 Update $x_{i,t}^l$ to minimize F_t via Eq. (2.6)
6 Update each $\theta_{i,j}^{l+1}$ to minimize F_t via Eq. (2.7)
7 **end**
8 **end**

The new proposed algorithm is CPC whose pseudo-algorithm is shown in Algorithm 3. This is a variant of IL, with two important properties:

1. Full automatization: CPC achieves this property by removing the stop control signal present in IL and Z-IL that triggers weight updates (Algorithm 1, line 5-6). This is another step further with regards to biological plausibility, as biological networks in the brain require on external control signal. This is the

first learning algorithm for deep neural networks in which neurons work fully autonomously. When the learning process begins, the algorithm immediately starts updating weights. This weight update is performed at each time step, in parallel with the inference steps. When the inference process has converged, it is stopped.

2. Efficiency: we have already seen that Z-IL is as efficient as BP. Furthermore, the efficiency of BP measured in MMS is $2L-1$. If we use SMMS, this number does not change since the matrix multiplications cannot be parallelized because the values or derivatives of one layer depend either on the previous layer or on the layer after. We note that the error signals of BP are cleared at the end of every iteration. Similarly, CPC also needs $2L-1$ MMSs or SMMSs to forward the network and spread the error in the first epoch. However, the error signal is never cleared, as the value node $x_{i,t}^l$ is not reset to $\mu_{i,t}^l$, and hence $\epsilon_{i,t}^l$ is not set to 0. The error remains and is constantly updated by Eq. (2.6), which allows the algorithm to constantly update the weights via Eq. (2.7). That is, after the first $2L-1$ MMSs, CPC needs

   (a) L MMSs for forwarding the data, that is, computing $\mu_{i,t}^l$ from $x_{i,t}^{l+1}$ according to Eq. (2.4). These matrix multiplications can be parallelized since they only depend on adjacent neurons, thus only requiring 1 SMMS to run.

   (b) $L-1$ MMSs for inference according to Eq. (2.6). Similarly, these $L-1$ MMSs can also be parallelized, requiring 1 SMMS to run.

   More formally, to complete K updates of weights on a network with L layers, BP needs $\mathcal{O}(K(2L-1)) = \mathcal{O}(KL)$ SMMSs while CPC only requires $\mathcal{O}(2(K-1)+(2L-1)) = \mathcal{O}(K+L)$ SMMSs.

We have hence proved the following theorem:

Theorem 1 *Let M and $\tilde{M}$ be two equivalent networks with L layers trained on the same dataset. Assume that M is trained using BP and $\tilde{M}$ is trained with CPC. Then, the time complexity measured by SMMSs needed to perform K updates of the weights is $\mathcal{O}(KL)$ and $\mathcal{O}(K+L)$ for BP and CPC, respectively.*

Unfortunately, the library [15] that supports running predictive coding networks Intelligent Systems is working on does not support fully parallelization. That is, currently we are unable to parallelize the operations in (i) and (ii). The full potential of the algorithm can only be reached we dive into lower level CUDA optimization, since the current libraries only support batch parallelization.

Although these two properties are important, there are many issues that remain open with this new model. The main question is, compared to BP, what is the model capable of. This includes tasks in supervised learning such as classification, and tasks in unsupervised learning such as clustering. However, it is also important to study if CPC is capable of memorization, which is a key part of the human brain. In the following section, one classical associative memory model and a more recent associative model are presented

2.3 Associative Memory

Associative memory (AM) [24] is defined as the ability to learn and remember the relationship between unrelated items. For example, associate an image of a cat with the label "cat". Associative memory can be divided into hetero-associative (HA) and auto-associative (AA) memories. In AA memory, a memory is linked to itself, which is important when we try to recover the original data from corrupted input. HA memory associates a pattern with another, as in the provided example.

AM might seem contradictory with the main machine learning paradigm, that is, to learn from a training set and generalize to new unseen data points. In AM, there

is no train-test split, and we do not need the model to generalize to new data points. Instead, the more "training" points the model can memorize, the better.

However, memorization is a crucial part of human learning and understanding of the world. Some recent work then look at *memory* and *learning* as two processes that are complementary of each other [3][17].

Feldman et al. [4] showed that learning data with a long-tailed distribution requires memorization. In such data distributions, memorization is key to achieve close to optimal generalization error. Later, empirical validation [5] supporting this statement were also submitted. A distribution is long-tailed if rare and atypical instances constitutes a significant portion of the data distribution. If this is the case, then useful examples (data points that are worth memorizing because they improve generalization error) are statistically indistinguishable from useless examples, such as outliers or mislabeled data. Then, memorization of useless examples necessary in order to obtain optimal results. The authors refer to this theory as the "Long tail theory"

Now, I present some important results and networks of associative memory.

2.3.1 Hopfield Networks

One of the earliest models that implements auto-associative memory is the Hopfield Network [7][26]. It is an unsupervised network that does not learn using backpropagation, instead, a more biologically plausible simple Hebbian learning rule is applied. The network is able recover from corrupted data the original data.

Essentially, it is a recurrent neural network (the output of the neurons goes back to the neurons) where each neuron is connected to all the others. Suppose that the data dimension is d, then the value stored by each neuron is $x_i, i = 1, ..., d$. Several assumptions are made: there are no self-connections $w_{ii} = 0, i = 1, ..., d$ and the connections are symmetrical $w_{ij} = w_{ji}$. The weights are recollected in the matrix $\mathbf{W}$. Now, suppose that we want to store the patterns $\overline{v}^1, ..., \overline{v}^N$ (binary data taking values

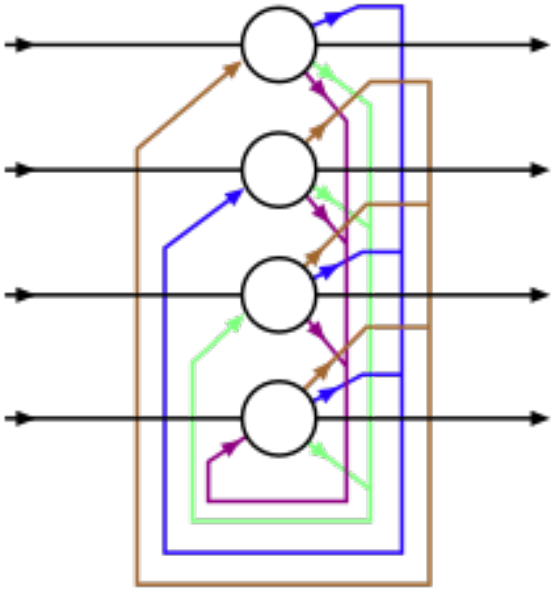


Figure 2.3: Illustration of a Hopfield Network, where each neuron is connected to the rest of neurons, taken from [25].

of 1 or -1), the network learns these patterns by using the following Hebbian learning rule: $w_{ij} = w_{ji} = \sum_{s=1}^{s=N} v_i^s v_j^s$. It is a Hebbian learning rule because if two neurons fires together then the weight between them increases. In this case, if two bits fires together then the weight between them is set to 1.

Each neuron of the network has an associated threshold θ_i, which determines if the neuron fires or not:

$$x_i \leftarrow 1 \quad \text{if} \sum_j w_{i,j} x_j \geq \theta_i$$

These values will be refered to using $\overline{\theta}$.

To recover an image, we start with a corrupted version $\overline{\xi}$ (refered to as state), and use the following rule to retrieve a stored pattern. Here, t is the iteration number:

$$\overline{\xi}^{t+1} = sign(\mathbf{W}\overline{\xi}^t - \overline{\theta}). \tag{2.8}$$

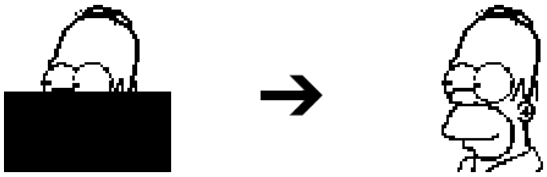


Figure 2.4: Image retrieval starting from half of an image, using a traditional Hopfield network for binary data, taken from [8].

With enough updates, it eventually converges to a minimum in the energy function, which is one of the stored memories V^s. The updates can be done synchronously, updating d elements of the state $\overline{\xi}$ at once, or asynchronously, where in each iteration only one weight ξ_i is updated. An example of a recovered image is shown in Figure 2.4.

The updates from Eq. (2.8) minimized the following energy function:

$$E = -\frac{1}{2}\sum_{i,j} w_{i,j}\xi_i\xi_j + \sum_i \theta_i\xi_i. \tag{2.9}$$

This simple network has a surprisingly high capacity:

$$C \cong \frac{d}{2\log(d)} \cong 0.14d. \tag{2.10}$$

However, it is a well known problem that the network suffers from imperfect retrievals when the number of stored pattern increases. This usually happens because the patterns are strongly correlated, therefore, a model that is able to distinguish between the stronly correlated patterns is needed.

2.3.2 Modern Hopfield Networks (Dense Associative Memories)

The discrete version of this model was first proposed by D. Krotov and J. Hopfield in [11] and generalized by Demircigil et al [2].

1. Krotov and Hopfield introduced the following energy function:

$$E = -\sum_{i=1}^{N} F(\overline{v}^i \cdot \overline{\xi}),$$

where F is a function $F(z) = z^a$, a being the parameter to choose. In the case of $a = 2$ the network is equivalent to the classical Hopfield network. This model has a storage capacity of error free retrieval of $C \cong \frac{1}{2(2a-3)!!}\frac{d^{a-1}}{\log(d)}$.

2. Demircigil et al generarized this model by using $G(z) = \exp(z)$, which can be viewed as the previous polynomial function F with $a \to infty$:

$$E = -\sum_{i=1}^{N} \exp(\overline{v}^i \cdot \overline{\xi}),$$

using $\mathbf{V} = (\overline{v}^1, ..., \overline{v}^N)$ as the matrix containing patterns, we can rewrite the above equation as:

$$E = -\exp(\text{lse}(1, \mathbf{V}^T\overline{\xi})),$$

where lse stands for log-sum-exp:

$$\text{lse}(\beta, \overline{z}) = \beta^{-1}\log\left(\sum_{l=1}^{N}\exp(\beta z_l)\right).$$

The storage capacity of this network is:

$$C \cong 2^{\frac{d}{2}}.$$

The parameter β is a constant which regulates the learning process, with high values of β giving more separable attractor basins (and perfect image recoveries); and lower values leading to metastates, which are juxtaposition of several images that are close together in the attractor space. The update rule for the state $\overline{\xi} \in \{-1, 1\}^d$ is common for both networks. We denote by ξ_i the i-th component of the vector $\overline{\xi}$, and $\overline{\xi}_{i-}$ to denote the vector that is the same as $\overline{\xi}$, with the i-th bit set to $+1$ and -1 respectively. The update rule for i-th component of the state vector is then the sign of the difference of the energy of the state vector with i-th bit flipped:

$$\xi_i^{t+1} = \text{sign}\left(-E(\overline{\xi}_{i+}^t) + E(\overline{\xi}_{i-}^t)\right),$$

where t is the time or epoch.

H. Ramsauer et al [19] generalized the model for **continuous patterns**, e.g. floating point numbers. They proposed a new energy function:

$$E = -\text{lse}(\beta, \mathbf{V}^T\overline{\xi}) + \frac{1}{2}\overline{\xi} \cdot \overline{\xi} + \frac{1}{\beta}\log N + \frac{1}{2}M^2, \tag{2.11}$$

where $\mathbf{V} = (\overline{v}^1, ..., \overline{v}^N)$ and M is the largest norm of all stored patterns. The authors also derived an update rule for the state pattern $\overline{\xi}$:

$$\overline{\xi}^{t+1} = \mathbf{V}\sigma(\beta\mathbf{V}^T\overline{\xi}^t), \tag{2.12}$$

where σ is the well known softmax function:

$$\sigma(\overline{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}},$$

for a vector $\overline{z}$ of dimension K. This dense associative memory model will also be used in experiments.

2.3.3 Overparameterized Neural Networks

A. Radhakrishnan et al [18] analyzed empirically how overparameterized autoencoders can implement auto-associative memory. They showed two main results:

1. Overparameterized autoencoders store training examples as attractors, and thus, viewing the autoencoder as a mapping function g, reiterating g leads to sample recovery.

2. The same method allows for memorizing sequences of data, which is an even more efficient mechanism for memory. However, in this work, we do not consider sequences of data.

Definition 1 *Let $g : \mathbb{R}^d \to \mathbb{R}^d$ be the mapping of a learned autoencoder trained on a dataset $\mathbf{V} = \{\overline{v}^i\}_{i=1}^N \subset \mathbb{R}^d$, a point $\overline{v}$ is a fixed point if $g(\overline{v}) = \overline{v}$.*

Note that we use $\overline{v}$ instead of the usual $\overline{x}$ to be consistent, since the training data are the patterns we want to memorize. Now, consider the sequence $\{g^k(\overline{v})\}_{k\in\mathbb{N}}$, where $g^k(\overline{v})$ denotes k compositions of g applied to $\overline{v}$. For fixed points, this sequence obviously converges to $\overline{v}$.

Definition 2 *A fixed point $\overline{v}^* \in \mathbb{R}^d$ is an attractor of $g : \mathbb{R}^d \to \mathbb{R}^d$ if there exists an open neighborhood (an open set), $\mathcal{O}$, of $\overline{v}^*$ such that for any $\overline{v} \in \mathcal{O}$, the sequence $\{g^k(\overline{v})\}_{k\in\mathbb{N}}$ converges to $\overline{v}^*$ as $k \to \infty$. The set S of all such points is the basin of attraction of $\overline{v}^*$.*

Now the following proposition is a well known condition that allows us to check if a point is an attractor:

Proposition 1 *A fixed point $\overline{v}^*$ is an attractor of a differentiable map g if the absolute value of all eigenvalues of the Jacobian of g at $\overline{v}^*$ are strictly less than 1. If any eigenvalues are greater than 1, $\overline{v}^*$ cannot be an attractor.*

Where The Jacobian matrix is a matrix that recollects all partial derivatives of g. This means that in order to check if a pattern we want to memorize is an attractor point, we just have to check the Jacobian matrix of the autoencoder map g. Since it is a multi-layer perceptron with almost everywhere differentiable non-linearities, the condition of being a differentiable map is met.

The authors also proved empirically that as the number of hidden layers of the network increases, the capacity of the network also increases, e.g. more images can be memorized.

Chapter 3

Methodology

In this chapter I present the models and the techniques used for both auto-associative memory and hetero-associative memory. One of the main objectives is to compare the Predictive Coding Networks to Backpropagation models. However, it is hard to establish an exact correspondence between PCN and BP models. To minimize the difference and try to maintain a fair comparison, I decided to use the simplest models for this task. The same models were used for both AA memory and HA memory experiments.

Recall that for AA memory, the task is to recover data points starting from corrupted data. For HA memory, the main task is to associate one type of memory with another, e.g. recover a caption from an image and vice-versa.

3.1 Data

There will be two kinds of data used, depending on auto-associative memory or hetero-associative memory experiments. A simple description is provided here, and more information about these datasets will be provided in the next chapter.

- For AA memory: the datasets used are images, including MNIST, CIFAR10,

ImageNet and TinyImageNet.

- For HA memory: one dataset will be used, flickr_30k, which consists of images with captions. To train the models on images with captions, I convert the captions to floating point numbers and run the training and test process as usual.

Parameter Search

There are several parameters in general that can be tuned with these datasets:

1. Data size: given any dataset, we can resize the images to for example $3 \times 32 \times 32$ or $3 \times 64 \times 64$, or convert them to greyscale images or even black and white images.

2. For flickr_30k, there are additional parameters regarding the captions. We can use a vocabulary to tokenize the captions or we can simply use the ASCII encoding of the characters.

3. For flickr_30k, the tokenized captions can be normalized to a certain interval, e.g. [0,1] or [0,5]. This is because otherwise the numbers will be orders of magnitude higher (up to 1000 to 1) than the images and can cause instability when learning images alongside

Corrupt Data

For HA memory, the task is different and we do not corrupt the original data. Instead, we try to recover images from captions and vice-versa. For AA memory, the task is to recover the original image starting from the corrupted images. The trained networks acts as a map from $g : \mathbb{R}^d \to \mathbb{R}^d$. By repeatedly applying this map, we recover the

original patterns. We denote one application of g a recovery iteration. There are numerous ways to corrupt data, in this work, the following are considered:

1. Adding Gaussian white noise to the image with mean 0 and standard deviation varying from 0.1 up to 0.5. Then we apply g several times to recover it.

2. Fixing part of the image, which can be $25\%, 50\%$. etc. That is, after each application of the map g to try to recover the original image, we overwrite the part of the image we want to fix. For example, if we fix half of the image, then the input to the second recovery iteration would be half of the recovered image from the first recovery iteration, and half of the original image.

When is Data Recovered?

The recovery conditions for images and captions are different:

- An image is considered to be recovered when it is close enough to the original image. That is, the mean squared loss (error) between both images are lower than a certain threshold. According to our visualization experiments, I noted that an image has no visible reconstruction noise when the error between the original image and the reconstruction is less than 0.005. Hence, I consider an image to be correctly retrieved if the reconstruction error is below this threshold.

- A caption is considered to be recovered if it is retrieved exactly, that is, the recovered words should match the original words exactly.

3.2 Backpropagation

Model

It has been shown that deep multi-layer perceptrons (MLP) can be used for auto-associative memory [18]. Essentially, the model used is a MLP autoencoder, with both

the input layer and the output layer having the same size as the data dimension. The number of hidden layers of the network is $D = L - 1$ (depth of the network) and the number of neurons in each hidden layer is W (width of the network).

A second reason for choosing autoencoders is because they are the most flexible model to perform associative memory tasks. In fact, their fully connected structure allows them to memorize single pixels without learning any pattern or using any kind of extra information (such as locality, naturally induced by convolutions). Hence, I have mostly cited works that perform associative memory experiments, and avoided the literature about generative models. Variational autoencoders (VAEs), for example, add probabilistic latent vectors which significantly improve the quality of the reconstruction when it comes to generating unseen images, but harm pure associative memory experiments.

Also, although autoencoders are not the state-of-the-art models in terms of capacity (Modern Hopfield Nets can have exponential capacity), no other work in the literature has shown reconstruction qualities and precision close to it. Particularly, no one has tried such reconstruction experiments with complex datasets, while the autoencoders are able to retrieve high quality pictures up to a great level of precision. Therefore, I believe that testing against this model is coherent with the scope of the thesis.

Note that following the notation established in [23] the output layer is indexed with 0. E.g., a network with 3 hidden layers has $3 + 1 + 1$ layers. An illustration of the model is provided in Figure 3.1.

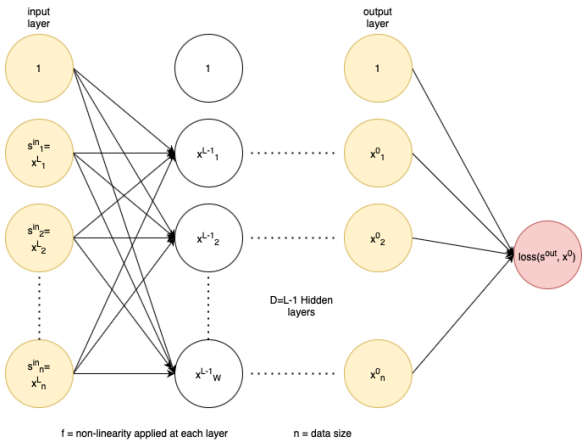


Figure 3.1: Illustration the backpropagation model used.

Train

To train the model, we feed the model the input data and let the model train according to BP update rules, using mean squared error as the loss function:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i^0 - s^o ut_i)^2.$$

Using this loss function, the output of the network should match as closely as possible with the input data after the training process is complete.

Test

For AA memory the input data are images. We test how many images are reconstructed by introducing noise in the input data and then recursively feeding the corrupted data to the trained network. That is, we feed forward the corrupted data to the network and obtain the output. We then feed this output to the network again. This process is repeated until the output does not change. Then, we compare the output with the original image and see if the recovered image is recognizable, or close enough to the original image. Let $\overline{s}^{cor}$ be the corrupted data, and g be the trained network, then we are comparing $g^k(\overline{s}^{cor})$ with $\overline{s}^{out}$. We refer to this process

of applying g as a recovery iteration.

For HA memory, there are two tests conducted:

1. We remove the caption and fix the image, then we try to retrieve the caption by reiterating using g, the trained network.

2. We remove the image and fix the caption, then we try to retrieve the image by reiterating.

Parameter Search

The possible parameters to tune are:

1. Learning rate (α): since we need the models to converge to a minimum in the loss function landscape, this parameter needs to be tuned. If it's too big, then the algorithm would not converge, and if it is too small, the convergence may be too slow.

2. Number of hidden layers: if the number of layers is too small, then the model would have few parameters to tune and cannot overfit (in this case, we actually want the model to overfit). However, if the number of layers is too large, we may suffer from error propagation, the phenomenon where error increases as the pass through layers. We might also suffer from the vanishing gradient problem, where the gradient gets lost with increasing number of layers. Some values are $1, 3, 5$, etc.

3. Number of neurons in each hidden layer: this also determines the number of parameters of the network, and has similar effect as the number of hidden layers with regards to overfitting. However, it does not affect much error propagation or the vanishing gradient problem.

4. Number of epochs: if this is too small then the model may not have enough time to converge.

3.3 PCNs

Model

The proposed model that intends to be equivalent to the backpropagation model is a network that has $D = L$ layers of dimension W and 1 layer of dimension d, which corresponds to the data dimension. The inner most layer is called the memory layer (indexed with L) and the output layer is called the sensory layer, which biologically corresponds to the sensory neurons. Furthermore, the memory layer has as input a vector $\overline{b}$ of dimension n, referred as the memory vector. Its purpose is to serve as the traditional *bias* for the first layer, and is updated during training. An example network of two layers is shown in Figure 3.2. Note that this network does not have an input layer.

Both networks are equivalent in the sense that the number of actual neurons that can be used to store information, the hidden layers of the BP model and the layers of the CPC model, excluding the output layer, have the same number of neurons. Furthermore, both networks only have fully connected linear layers, and the same non-linearity function f will be used. Further details will be provided in the next chapter.

Train

To train the model, we need to minimize the energy function, which is the squared sum of all error nodes:

$$F_t = \frac{1}{2} \sum_{i,l} (\varepsilon_{i,t}^l)^2.$$

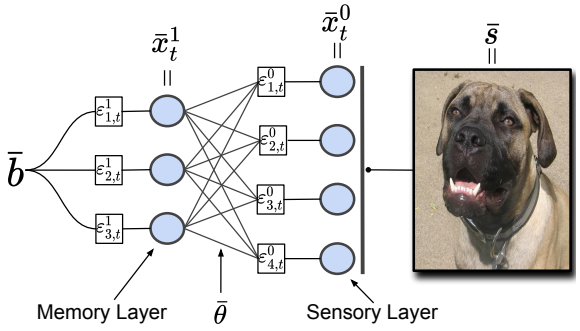


Figure 3.2: Illustration a two layer predictive coding network for associative memory model.

. Then, the training process depends on which PCN training algorithm is used. For most of the work, CPC was used. Therefore the weights are updated while inference is running, following algorithm 3.

Test

Independently of which PCN training algorithm is used, the testing process is the same for AA memory retrieval. Given a corrupted version of the data $\overline{s}^{cor}$,

1. First, fix the last layer of the network $\overline{x}^0$ to $\overline{s}^{cor}$.

2. Secondly, run inference until it converges. We expect that the prediction of the last layer $\overline{\mu}^0$ to be a less corrupted version of the data.

3. We can view steps 1 and 2 as a function $g : \mathbb{R}^d \to \mathbb{R}^d$ that sends $\overline{s}^{cor}$ to $\overline{\mu}^0$. Then, we repeat steps 1 and 2 until the data point is retrieved or until little difference is present between iterations, just as in BP.

For HA memory, the same tests as in BP is conducted:

1. We remove the caption and fix the image, then we try to retrieve the caption. using g.

2. We remove the image and fix the caption, then we try to retrieve the image.

Parameter Search

The possible parameters to tune are:

1. Training algorithm: mostly CPC will be used, but IL and Z-IL will also be considered
2. Inference learning rate (γ): learning rate for value nodes $\overline{x}^l$
3. Number of hidden layers: in this cases layers excluding the output layer.
4. Number of neurons in each hidden layer: same as above.
5. Number of iterations for training inference: This is because for IL, the number of iterations needed for convergence is usually too big and will make the algorithm too slow. Then, a limit T must be set. For the rest of algorithms, this is equivalent to the number of epochs
6. Number of training epochs.
7. Number of iterations for test inference: number of times to update the value nodes during image recovery.

3.4 Dense Associative Memory

The model used is the modern Hopfield network proposed by H. Ramsauer et al [19] for **continuous patterns**, presented in Chapter 2, Subsection 2.3.2.

Train

The network is trained by storing the patterns, and does not require weight updates. Let $\mathbf{V} = (\overline{v}^1, ..., \overline{v}^N)$ be the matrix containing the N patterns we want to store, and $\overline{\xi}$ be the state (input to the network). According to D. Krotov and J. Hopfield [10],

$\mathbf{V}^T$ can be viewed as the weights from $\overline{\xi}$ to the hidden neurons and $\mathbf{V}$ can be viewed as the weights from the hidden neurons to $\overline{\xi}$. Consequently, this model is the fastest to train.

Test

For AA memory or HA memory experiments, to recover the original pattern from corrupted or partial patterns, one just has to follow Eq. (2.12).

$$\overline{\xi}^{t+1} = \mathbf{V}\sigma(\beta\mathbf{V}^T\overline{\xi}^t).$$

We set $\overline{\xi}^t$ to the corrupted pattern and take $\overline{\xi}^{t+1}$ as the recovered pattern. Usually, it only takes one iteration to converge, and $\overline{\xi}^1$ is the output of the network.

3.5 General Procedure

New networks are constantly trained and saved, tuning the parameters mentioned before. However, even using the same model, since the initialization of the parameters is random, it may converge or not.

For auto-associative memory experiments, I evaluate the performance of a model by analyzing how many data points are recovered, starting from original, non-corrupted data. If the number is too low then this model will be retrained up to 3 times. If the result still does not improve, then I consider that the model with these configurations is not viable and try other parameters.

If a model is considered to have converged, e.g. able to recover most data points starting from the original data points, then I compute how many data points are recovered starting from corrupted data, according to the corruption methods mentioned in Section 3.1. The data will be aggregated and analyzed in Chapter 5.

For hetero-associative memory experiments, I evaluate a model's performance by the number of one type of data points recovered when the other type of data is given. For example, for captioned images, I evaluate how many images are recovered when captions are given and how many captions are recovered when images are given. The results are also aggregated and analyzed in Chapter 5.

Chapter 4

Implementation

In this chapter, implementation details are given to the reader. It is with special importance to mention that this project would not be possible without the library `general_energy_nets` developed and maintained by *Intelligent Systems* from the University of Oxford. The code I developed depends on said library, and can be found here [6].

4.1 Software Packages Used

4.1.1 Pytorch

The coding language used for this project is `Python`, version 3.8.8. The main reason for using `Python` for the project is because the main machine learning platforms are implemented in this language. In this case, the machine learning platform used is `Pytorch` 1.7.1.

`Pytorch` is a package that enables automatic differentiation and fast matrix operations with GPU support. Matrices and vectors are called tensors in `Pytorch`. It has methods and classes that helps the definition and training of artificial neural networks, with mainstream neural network layers, optimization algorithms and non-linearities

already implemented.

4.1.2 General Energy Nets

This library [15] was developed to facilitate the implementation of Predictive Coding Networks. It incorporates the energy function needed for predictive coding networks and makes it possible to minimize the energy function in Equation (2.5) instead of a loss function. It is based on the machine learning framework Pytorch, which traditionally supports BP algorithms. The two most important files are summarized below.

PC Layer

This is a class that inherits `torch.nn.Module`, and can be added to the neural network at any place. It implements the error neurons together with the value neurons illustrated in Figure 2.2, and it returns by default the squared energy. The library is general enough so that we can also use other energy functions, however, for this project only the squared energy function is used. Furthermore, although this layer can be added anywhere in the network, I followed the structure established in Figure 2.2: first the fully connected layer, then the PC layer, and finally the non-linearity. An example snippet of how to use the PCLayer is shown in Code 4.1.

PC trainer

This is a class that was implemented to train a network with PC layer. It is also able to train normal Pytorch networks without PC layers, in which case it acts as a normal trainer for Pytorch networks.

This class is capable of optimizing the energy function of the network, without requiring a loss function. It does so by instantiating two optimizers, one for updating the value nodes (running inference) and one for updating the weights of the network.

```
import torch.nn as nn
import predictive_coding as pc # general energy net module
class Example(nn.Module):
    def __init__(self) -> None:
        super(Example, self).__init__()

        self.fc = nn.Linear(256, 256)
        self.PClayer = pc.PCLayer()
        self.non_linearity = nn.ReLU()

    def forward(self, x):
        out = self.fc(x)
        out = self.PClayer(out)
        out = self.non_linearity(out)
        return out
```

Figure 4.1: Illustration of how to use PC Layer as part of a Pytorch model

Furthermore, the trainer is quite general, and also allows the optimization of a loss function or a loss function combined with an energy function. For this project, I only used energy function optimization.

Secondly, this class records and presents the evolution of the energy function during training time, which is essential for monitoring purposes. Thirdly, it supports IL, Z-IL and CPC, by controlling how many inference steps to take before updating the weights of the network. It is general enough that any optimizer can be used for inference, or for updating network weights. An example code snippet showing how to train the PC network shown in Code 4.1 for one epoch is provided in Code 4.2.

The most important arguments (all of them have default values) of this class are:

1. `optimizer_p_fn`: the optimizer for the weights of the network. The letter `p` is used to represent the weights throughout the library,

2. `optimizer_p_kwargs`: used to provide arguments to the optimizer for network weights, mostly the learning rate.

3. `optimizer_x_fn`: the optimizer for value nodes of the network. The letter `x` is

```
model = Example()
model_trainer = pc.PCTrainer(model,
    optimizer_x_kwargs={'lr': 0.5}, # value learning rate
    optimizer_p_kwargs={'lr': 1e-5}, # weight learnig rate
    # for inference only simply set p learning rate to 0
    optimizer_p_fn=torch.optim.Adam, # weight optimiser
    x_lr_discount=0.9,
    T = 100, # inference iterations
    is_sample_x_once = True,
    update_p_at='last') # when to update the weights,
    # last for IL, all for CPC
model.train()
data = torch.rand((1, 256))
model_trainer.train_on_batch(data.float())
```

Figure 4.2: Illustration of how to use PC trainer to train a PC model

used to represent the values stored by the neurons throughout the library.

4. `optimizer_x_kwargs`: used to provide arguments to the optimizer for value nodes, mostly the learning rate.

5. `T`: the number of inference steps that the trainer should run. Note that this is not equal to the number of inference steps the trainer should run before making a weight update, it's the total number of inference steps. It is worth noting that, the weight updates in PCNs are local and T must be large enough for the error to spread through the whole network. The authors of the library state at least $L + 1$ weight updates are needed, and recommends using $T > 4L$.

6. `update_p_at`: when to update the weights. If set to "all" then the trainer updates the weights after each inference step, thus effectively operating like CPC with `T` epochs. If set to "last" the trainer only updates the weights once after `T` inference steps, thus operating like IL with 1 epoch. Thus, if we want to run IL for N epochs we have to run the trainer N times, each time for `T` iterations.

It is worth noting that the method `train_on_batch` of the trainer has an argument `callback_after_t`. It allows us to perform actions after each inference iteration. This will be used to fix the output of the network to $\overline{s}^{out}$ during network training.

4.1.3 Hopfield Layers

The authors of the paper *Hopfield Networks is All You Need* [19] has implemented the modern Hopfield network and made it public. The library is available here: `https://github.com/ml-jku/hopfield-layers`. The library has several modules, implementing different Hopfield networks. For the purposed of this project, only the module `Hopfield` is used, which implements the described dense associative memory network explained in Chapter 2, Subsection 2.3.2.

An example code illustrating its usage is shown in Code 4.3, taken from the associated blog post [8].

4.1.4 Other Packages

It is worth mentioning that the following packages were also used:

- `matplotlib`, to generate plots.
- `spacy`. It is a package used for natural language processing, and I used this package to tokenize sentences as a preprocessing step for the dataset `flickr_30k`.
- `torchvision`, for loading and preprocessing several datasets. This package is intimately related to `Pytorch`.

```
hopfield = Hopfield(
    scaling=beta,

    # do not project layer input
    state_pattern_as_static=True,
    stored_pattern_as_static=True,
    pattern_projection_as_static=True,

    # do not pre-process layer input
    normalize_stored_pattern=False,
    normalize_stored_pattern_affine=False,
    normalize_state_pattern=False,
    normalize_state_pattern_affine=False,
    normalize_pattern_projection=False,
    normalize_pattern_projection_affine=False,

    # do not post-process layer output
    disable_out_projection=True)

# tuple of stored_pattern, state_pattern, pattern_projection
hopfield((Y, R, Y))
```

Figure 4.3: Illustration of how to use `Hopfield` to implement the dense associative memory network for continuous patterns from Chapter 2, Subsection 2.3.2
.

4.2 Datasets

MNIST

The MNIST dataset [13] consists of handwritten digits as greyscale images of size 28×28. It is a subset of the NIST dataset with 60000 training samples and 10000 test samples. The dataset is widely used in the machine learning community, since the images are well prepared and minimal effort for preprocessing and formatting is required.

CIFAR-10

The CIFAR-10 dataset [9] consists of 60000 RGB colored images of size 32×32, divided into 50000 training images and 10000 test. The images are divided into 10 classes such as airplane, cat or dog, with 6000 images in each class. Like MNIST, minimal preprocessing and formatting is required.

TinyImageNet

The TinyImageNet dataset [12] consists of 120000 images of 200 classes with 600 images in each class. For each class, the 600 images are divided into 500 training images, 50 test images and 50 validation images. The images are RGB colored of size 64×64.

ImageNet

This is a large dataset contains colored images of different sizes. For the purposes of this project, we only need up to 500 images from this dataset, independently of their label. The images are then resized to 224×224 images.

Flickr 30k

The Flickr 30k dataset [16] [27] consists of 30000 colored images of varying image sizes, each one of them is associated with 5 different captions. For the purposes of the project the images were downsized to 32×32 or 64×64, and sometimes converted to greyscale images. Furthermore, Only one caption is used as the "label" for the image. Then, the captions are unique, so one image is only associated with one caption and vice-versa.

The captions are strings of characters initially, and two ways of preprocessing them are considered:

1. After using `spacy` to tokenize the captions, the words are collected in a dictionary that assigns each word to a integer. Then, the tokenized captions are converted to lists of integers and to `Pytorch` tensors. However, the captions have different sizes, to balance it, the number 0 is used as "padding" so all the captions have the same size as the longest caption.

2. The second option is to use the numerical values of the characters directly, via the `ord` built in method in `Python`. Afterwards, the captions are converted to lists of integers and padded using 0.

Independently of the method used, the captions are converted to floating point numbers and then normalized to $[0, 1], [0, 2], [0, 5], [0, 10]$. This is because compared to the size of the image $3 \times 32 \times 32$, the captions are small with around 50 numbers. Since the images have values in the range $[0, 1]$, by increasing the contrast between image values and caption values, we hope that captions can **stand out** and be easier to memorize.

num images	1	5	10	25	50	100
caption size	18	21	22	28	41	56
vocabulary size	20	59	98	201	331	563

Table 4.1: Caption and vocabulary size with respect to number of images to memorize.

For tokenized captions, the size of the vocabulary used depends on the number of data points used. If more captioned images are used, the vocabulary size becomes larger and so does the caption size as it has to match the size of the longest caption. Detailed information about caption size and vocabulary size is shown in Table 4.1. Note that the more images there are, the more difficult the captions are to memorize. Firstly, they are longer, and secondly, as the vocabulary size increases the words are closer to each other. If we normalize the images to $[0, 1]$, then for 10 images consecutive words in the vocabulary differ by $\frac{1}{98}$, but for 100 images it decreases to $\frac{1}{563}$.

4.3 Software Development

The development of model train and test scripts has been done using the text editor Visual Studio Code. Furthermore, it was done via remote connection to the `PSSR2` cluster belonging to *Intelligent Systems* from the University of Oxford. This cluster has 4 powerful GPUs, NVIDIA TITAN RTX 24G GDDR6.

Furthermore, the source control system used is Git with Github. It is used only to store scripts and results such as images or text files. The trained models are quite large and are only stored locally, as well as the datasets.

Since the most part of the work is training and testing networks, I did not follow any specific software development cycle. Similar to most machine learning work, I spent most of the project tuning hyper-parameters and designing experiments rather than implementation. The results obtained were stored in shared documents with my supervisors, allowing them to access and comment at any time.

The main synchronization method was weekly meetings, which were held on Mondays to ensure that the project was progressing according to the established timetable. Furthermore, the instant messaging platform *Slack* was used to communicate progress when necessary.

4.3.1 Code structure

The code base is divided into 3 different sub directories:

- `assocmemo_cpc`: contains files related to auto-associative experiments for PCNs.
- `assocmemo`: contains files related to auto-associative experiments for BP autoencoders. The training scripts, test scripts, and generated result files and images are located within the sub directory `experiments`.
- `flickr_assocmemo`: contains files related to hetero-associative experiments. Inside the sub directory `models` I store the models and training results. Also,

generated images are stored in this folder.

- `dam_experiments`: contains files related to Dense Associative Memories experiments.

And note that trained models are not uploaded due to their large size.

Chapter 5

Experimental Results and Analysis

5.1 AA Memory Experiments

The main objective is to show experimentally that PCNs can implement associative memory, and compare its performance with BP autoencoders. To achieve this objective, I compare the models across the datasets MNIST, CIFAR10 and TinyImageNet.

5.1.1 MNIST Experiments

I used this dataset as a base benchmark to show that both PCNs and BP autoencoders can memorize MNIST images, since this dataset is widely used in the machine learning community. However, most experiments are done with more complex datasets.

I trained for 100 images autoencoders and PCNs of width $W = \{1024, 2048\}$, and depth $D = 1$. The I evaluated the models based on how many images were recovered after adding Gaussian noise and how many were recovered when part of the image is removed and the rest is fixed. The threshold used to test if the images are correctly retrieved is 0.005.

The results are shown in Table 5.1. We can see that the results for adding Gaussian noise with standard deviation 0.2 are similar, with PCNs recovering a bit more. Both

		Add Gaussian noise		Remove part of the image		
100 images	Width	std=0.2	std=0.4	25%	50%	75%
Autoencoders	1024	80	73	100	100	39
	2048	86	56	100	100	44
PCNs	1024	91	7	100	100	7
	2048	96	10	100	100	10

Table 5.1: Number of recovered images for 100 MNIST greyscale images, using networks of depth 1.

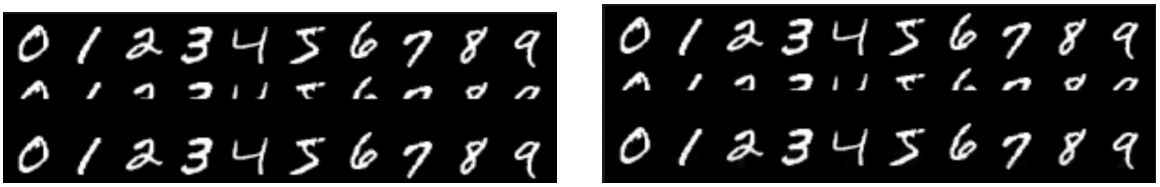


Figure 5.1: Recovery of 10 MNIST digits after removing 50% of the original image, by PCNs (left) and BP autoencoders (right).

models recover the images perfectly when 25% and 50% of the images are removed, however, autoencoders perform better when the images are further apart from the original images. This issue is not true for networks of higher depth, as I will show in the following experiments. An image showing perfect recovery by both networks is shown in Figure 5.1.

Moreover, examining the recovered images for the PCNs that performed poorly, and comparing them with autoencoders, I found an interesting difference. Observe in Figures 5.2 and 5.3, when PCNs fails to converge to the original image, they converge to an approximation of the original image. On the other hand, BP autoencoders store images as attractors, thus, they converge to the nearest stored attractor even though it is a completely different digit. This further supports the biological plausibility of PCNs, since when humans try to remember a handwritten digit an approximation of the digit is enough to recover the digit value. It is unlikely that we will remember an image of an entirely different digit starting from, for example, an image of a handwritten 1.

Also, this also explains why PCNs perform poorly with MNIST images when 75%

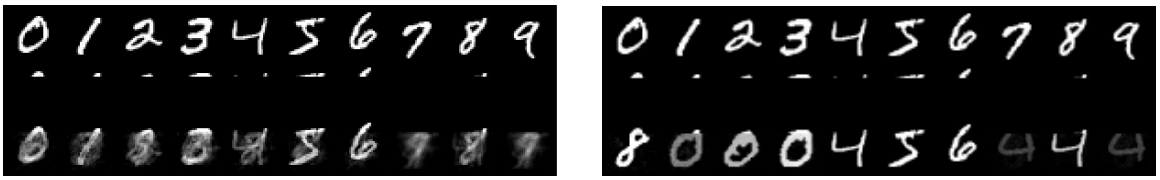


Figure 5.2: Recovery of 10 MNIST digits after removing 75% of the original image, by PCNs (left) and BP autoencoders (right).

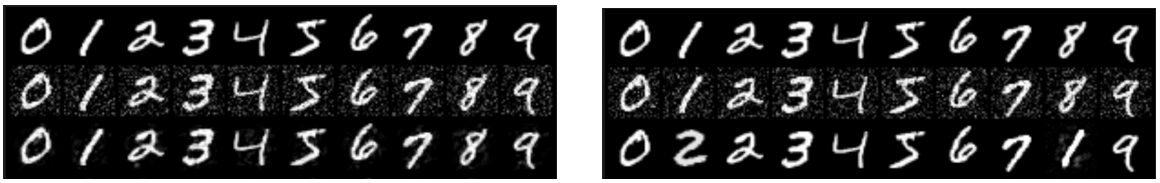


Figure 5.3: Recovery of 10 MNIST digits after adding Gaussian noise with std=0.2 to the original image, by PCNs (left) and BP autoencoders (right)

of the digits is removed. In MNIST the top 25% of the image are mostly black pixels containing a lot of useless information. This leads the network to converge to a state that resembles the juxtaposition of all the digits, as shown on the left side in Figure 5.2, for the digits 7 and 9. Observe that the autoencoder converges to a version of digit 4 instead. In the following sections, I found that PCNs are much more effective when the information contained on the part of the image that we fix is unique.

These results show the possibility of using PCNs as associative memory networks, and suggest they are more biologically plausible compared to the autoencoders. To further establish PCNs can implement associative memories experimentally, I examine the models using CIFAR10 and TinyImageNet datasets.

5.1.2 CIFAR10 Experiments

The dataset used consists of $3 \times 32 \times 32$ colored images, and I trained networks with the following parameters:

1. Depth $D = 1$, non-linearity ReLU.



Figure 5.4: 10 retrieved images of the CIFAR10 image dataset using depth 1 PCN with width 2048, adding Gaussian noise of standard deviation 0.2. From top to bottom: original images, images with added noise, and retrieved images.

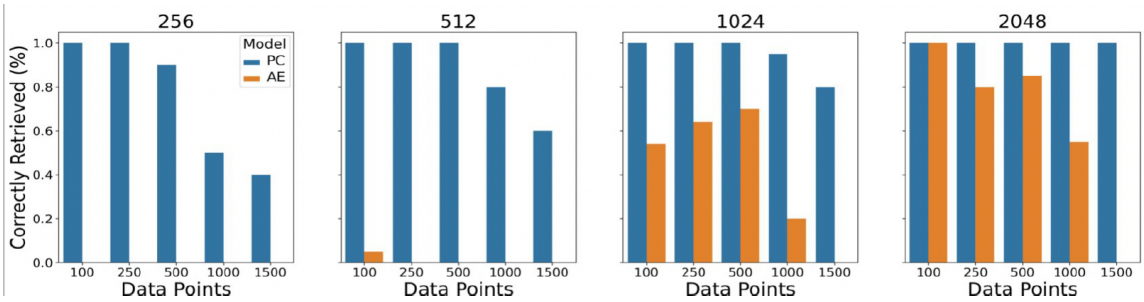


Figure 5.5: Percentage of correctly retrieved images by depth 1 generative PCNs and autoencoders with hidden-layer dimensions of 256, 512, 1024, and 2048, when presented with a corrupted image with Gaussian noise of standard deviation 0.2, trained on **CIFAR10**.

2. Width $W \in \{256, 512, 1024, 2048\}$

3. Number of images $N \in \{100, 250, 500, 1000, 1500\}$

And evaluated them by adding Gaussian noise with standard deviation $= 0.2$ to the original images and computing the number of recovered images. The recovery process takes about $3 - 5$ seconds for autoencoders and $1 - 2$ minutes for PCNs, since they require the inference to converge to finish a recovery iteration. The results are illustrated in the bar charts showed in Figure 5.5. The main objective of this experiment is to show the capacity of the network as the number of hidden neurons increases.

The results show that PCNs are able to store and retrieve data points even when

the network is not overparameterized. Autoencoders trained with BP did not perform well: autoencoders with less than 1024 hidden neurons always failed to restore even a single data point, and the performance of autoencoders with higher number of hidden neurons performed always worse than PCNs. This shows that for images of bigger size overparameterization is essential for autoencoders to perform AM tasks, since A. Radhakrishnan showed that overparameterized autoencoders (depth 11, with 1024 neurons) can memorize CIFAR10 images. Therefore, the proposed method CPC offers a much more efficient alternative.

In terms of capacity, I show that depth 1 PCNs with 256 hidden neurons are able to perfectly store up to 250 CIFAR10 images, and that networks with 2048 hidden neurons always managed to store and retrieve all the presented datasets. For larger datasets, the smaller networks failed to store some of the images because of their limited capacity. The parameter space is not large enough to store every data point. PCNs with 256 and 512 hidden units are able to store almost 500 CIFAR10 images but fail to store more images. Moreover, from the figure we can see that the capacity of the network increases with the increasing number of hidden neurons.

It is also worth noting that the PCNs trained with CPC converged with only 25000 weight updates (about 5 minutes for the bigger networks), and the convergence is stable. Once the model reaches a local minimum it stays there. Autoencoders trained with BP, on the other hand, does not always converge and had to be retrained several times until one model converges. Furthermore, when the model converges, it can still diverge and leave the local minimum. I tried decreasing the learning rate to avoid this phenomena, however, with the final learning rate $5e-5$ the models take more than 200000 weight updates to converge, and the bigger networks takes more than 2 hours to train.

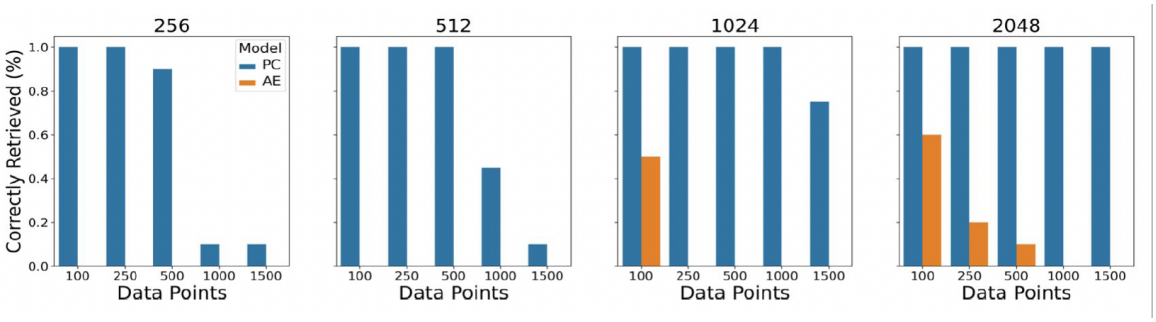


Figure 5.6: Percentage of correctly retrieved images by depth 1 generative PCNs and autoencoders with hidden-layer dimensions of 256, 512, 1024, and 2048, when presented with a corrupted image with Gaussian noise of standard deviation 0.2, trained on **TinyImageNet**.

5.1.3 TinyImageNet Experiments

The images of TinyImageNet initially have different sizes. I have chosen to resize the images to $3 \times 64 \times 64$ for uniformity. Three experiments were conducted for this dataset: capacity study, partial data recovery study, and depth study.

Capacity Study

In the same way as with CIFAR10 images, I trained PCNs and autoencoders to memorize $N \in \{100, 250, 500, 1000, 1500\}$ images and measure their performance by the number of images recovered from corrupted images with Gaussian noise. Since the images are quite large, the training time of autoencoders increases significantly and needs around 1M weight updates to converge, and more than 3 hours to finish. PCNs on the other hand takes about 10 minutes to train. The testing process takes about 10 seconds for autoencoders and 2 to 3 minutes for PCNs. The results of the capacity study is shown in Figure 5.6.

The results are similar to the capacity study conducted for CIFAR10, with autoencoders performing even worse. When trying to learn more than 500 images, e.g. 1000 and 1500 images, no autoencoder of depth 1 converged and no images were recovered. PCNs with depth 1 still performed quite admirably, being able to restore up to 1500

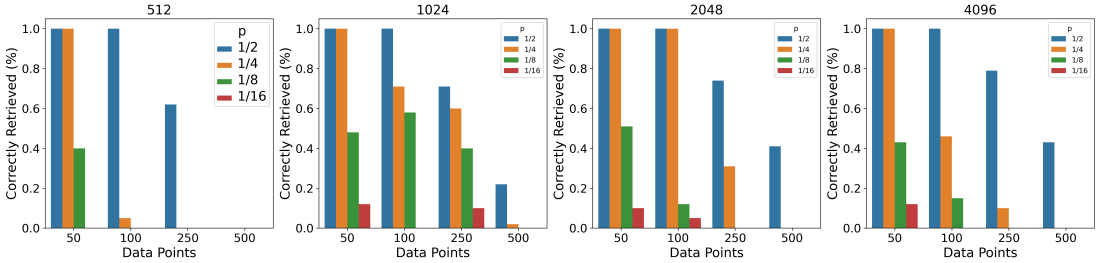


Figure 5.7: Percentage of correctly retrieved images by depth 1 generative PCNs and autoencoders with hidden-layer dimensions of 512, 1024, and 2048, 4096, when presented with $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ of the original images, trained on **TinyImageNet**.

images. This further confirms that PCNs do not need to be overparameterized in order to implement associative memories.

Partial Data Recovery Study

Now, instead of corrupting the images using white noise, we remove part of the image and only give the rest of the image to the network and compute the number of recovered data points. The percentage of image given to the networks $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$. We trained PCNs of depth 1, with width $W \in \{512, 1024, 2048, 4096\}$ for this task. Furthermore, I trained the networks with $N \in \{50, 100, 250, 500\}$ data points. The results are shown in Figure 5.7.

For 50 images, every network was able to reconstruct the original images given $\frac{1}{4}$ of the original image and about half of when given $\frac{1}{8}$. As the number of images increases, it is increasingly difficult for the networks to memorize the images and we see an overall decrease in the number of reconstructed images. When more than 500 images were given, no network was able to recover images if less than $\frac{1}{2}$ of the original image is given. The reconstruction of 2 correctly reconstructed images is shown in Figure 5.8.

These experiments show the limit of shallow PCNs on both reconstruction quality and capacity. This is expected, as increasing the number of data points to learn or decreasing the amount of information available hurts associative memory models'

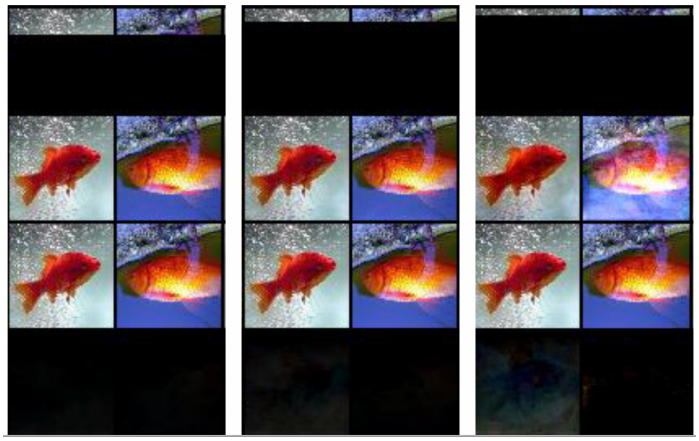


Figure 5.8: Partial images and their reconstructions (from top to bottom: partial image, reconstructed image, original image, and reconstruction error (difference between original and reconstruction)). 2 64×64 Tiny ImageNet images presented with 1/4, 1/8, and 1/16 of the original image, respectively, using a network of 1024 hidden neurons.

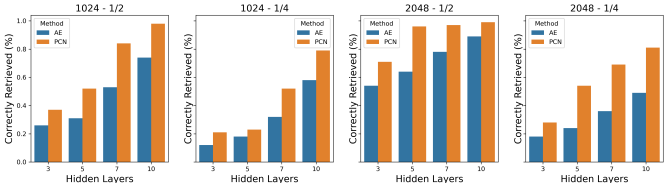


Figure 5.9: Percentage of correctly retrieved images as a function of the number of hidden layers, for both autoencoders (AEs) and generative PCNs, on a dataset of $N = 500$ images. I used networks with $W = \{1024, 2048\}$ hidden neurons, and the images are taken from Tiny ImageNet.

performance.

Depth Study

Now, I test the hypothesis that increasing depth, and hence increasing the dimension of the parameter space, also increases the capacity of PCNs. To test this I used $N = 500$ TinyImageNet images, networks with depth $D \in \{3, 5, 7, 10\}$ and width $W \in \{1024, 2048\}$. To evaluate the networks' performance, I computed the number of images recovered when only $p \in \{\frac{1}{2}, \frac{1}{4}\}$ of the original image is presented to the PCNs. I also tested this against autoencoders of the same depth and width, trained with the same non-linearity (ReLU). The results are shown in Figure 5.9.

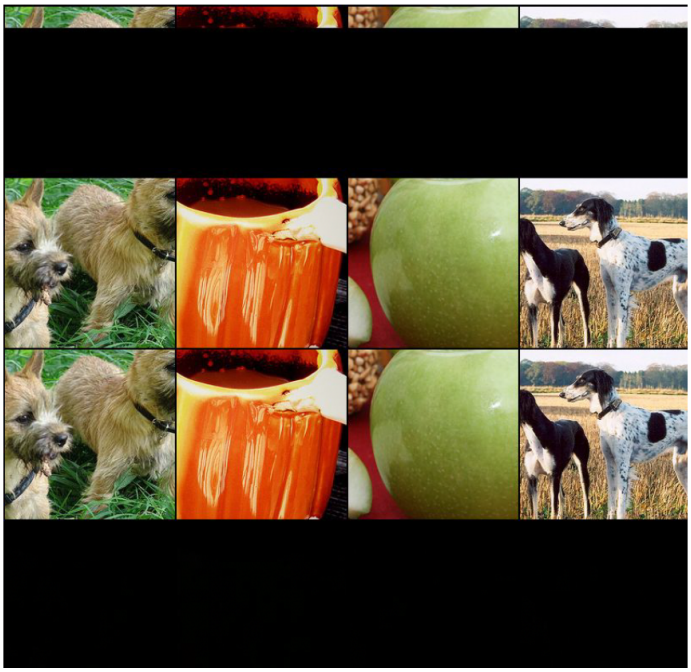


Figure 5.10: 4 reconstructed images of Imagenet when $\frac{1}{8}$ of the original images were presented, using a PCN of width 8192 and depth 1.

The results confirm my hypothesis, with a clear increase in number of images recovered when the depth of the network increases. Also, the number of correctly reconstructed images by PCNs is always higher than autoencoders. For example, the PCN of depth 10 width 2048 was able to recover 98% of the original images against 72% of autoencoders.

5.1.4 Imagenet Experiments

Now, I wanted to apply the network to more realistic datasets. I decided to use ImageNet, with the images resized to $3 \times 224 \times 224$. I trained a PCN of depth 1 width 8192 on 10 ImageNet images and I have successfully recovered them when $\frac{1}{8}$ of the original images were presented. The reconstructions of 4 of them are shown in Figure 5.10.

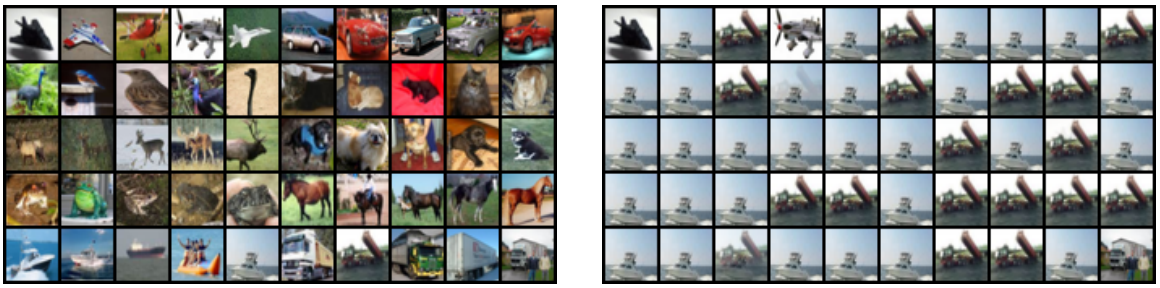


Figure 5.11: Recovery of 50 CIFAR10 images when $\frac{1}{2}$ of the images are presented to the Dense Associative Memories model, original images (left) and recovered images (right).

5.1.5 Dense Associative Memory (DAM) experiments

In this section I present some of the results I obtained using Modern Hopfield Networks for continuous data, sometimes referred to as Dense Associative Memory (DAM). The experiments were conducted using the library the authors of the paper *Hopfield Networks is All You Need* [19] have implemented.

What is worth noting is that the training procedure of this network is almost instant since it only has to store a combination of the memories in a matrix. The retrieval process is also quick and simple, involving several matrix multiplications. Compared to PCNs and autoencoders which require more time to train and test, DAMs excel in efficiency, both in time and memory.

The results I found, however, are not promising. When present half of the original images to the model, trained with 10 images, the model was only able to recover 1; for 20 images 2, for 50 images 4 and for 100 images only 4 were recovered. I have also tested how the model performs when white noise is added, and the results are also not promising. Refer to Appendix A for additional results. The reconstructed images of 50 CIFAR10 images are shown in Figure 5.11.

Moreover, changing the parameter β does not help since the images are not juxtaposed into a metastate. The model just fails to memorize most of the images and decays to one of the learned images, a behavior similar to BP autoencoders, which

converges to the nearest attractors. Another proof of the limited capacity is when I try to memorize 10 images and present to the model the original images, the model is only able to recover 4. However, it is already impressive that with close to 0 training time, the model is capable of recovering 4 out of 10 CIFAR10 images.

Please note that the results I have collected depends on the correctness of the cited library. One can implement DAMs from scratch, but the authors of the said library have already done extensive tests, and is more likely to be more robust.

5.2 HA Memory Experiments

In this section, the results obtained from hetero-associative memory experiments are presented. The main objective is to show experimentally that the proposed PCN network can implement hetero-associative memory. Thus, most of the experiments were conducted only for the PCN network, with only a few results from the Autoencoder network for comparison purposes. The experiments were conducted using the `flickr_30k` dataset, for the following parameters of the dataset:

1. Number of captioned images: $N \in \{5, 10, 25, 50, 100\}$

2. Image size: since the images of the dataset have different sizes and are initially too large, I had to resize them down to 32×32 and 64×64. I found that the networks had problems with memorizing captioned images of size 64×64, and conducted the remainder of the experiments with images of size 32×32. The experimental results for models that either failed to converge or have bad results can be found in Appendix A.

3. Image style: I used colored images and greyscale images. For images with size 32×32, colored images would have $3 \times 32 \times 32 = 3072$ pixels and greyscale images would have 1024 pixels.

4. Vocabulary: I tried using a tokenizer together with a dictionary which converts words to floating point numbers. The other method, i.e. character encoding, was less effective and was discarded after first trials.

5. Normalization scale: I used $ns \in \{1, 2, 5, 10\}$.

For the PCN network, the following parameters were used:

1. Depth of the network: $D = L \in \{1, 3, 5, 79, 11\}$.

2. Width of the network: $W \in \{512, 1024, 2048\}$.

3. Activation function: `ReLU`.

The tasks for each of these trained networks are 1. to recover captions when provided images and 2. to recover images when provided captions. Note that the difficulty of these two tasks are different, with image recovery being more difficult. This is because when you try to recover captions, you are usually given $3 \times 32 \times 32 = 3072$ pixels, and you try to recover $\cong 50$ numbers. On the other hand, when given captions, the networks have to remember the images with 3072 pixels which are two magnitudes larger. Essentially, the scarcity of available information makes image recovery a difficult process.

5.2.1 Network Depth Comparison

I fixed the following parameters:

1. Images: colored $3 \times 32 \times 32$ images.
2. Captions: tokenized captions with varying sizes as explained above, normalized to $[0, 1]$
3. Learning rate: $5e-5$ with Adam optimizer, which I found appropriate after AA memory experiments. Also note that Adam automatically adjusts the learning rate.

And I trained the following models:

- For number of captioned images $N \in \{5, 10, 25, 50, 100\}$.
- For depth $D = L \in \{1, 3, 5, 7, 9, 11\}$.
- For width $W \in \{512, 1024, 2048\}$.

Num data\ Depth	1	3	5	7	9	11
25	0	25	25	25	25	25
50	0	50	50	50	50	50

Num data\Depth	1	3	5	7	9	11
25	0	17	23	25	24	25
50	0	22	26	33	38	42

Table 5.2: Number of captions recovered if given images (up) and number of images recovered if given the caption (down)

The results show that as the depth of the network increases, the networks are capable of memorizing more captioned images. The width of the network, surprisingly, does not affect much the number of images recovered, with smaller networks often recovering more images than the larger networks. I believe that this is because all the networks have enough capacity, and increasing the width only makes it harder to converge to a suitable minimum in the energy function. Thus, I chose the best network of each depth and gather the results in Table 5.2.

Also, to help with the visualization of the data, a bar chart is shown for the best models for 25 and 50 images in Figure 5.12. Surprisingly, the network of depth 1 performed quite poorly and recovered 0 captions and 0 images. The networks of depth 3 or above all performed quite well. When trained with 25 captioned images, they were able to recover all 25 captions and most images. When trained with 50 images they were able to recover all 50 captions. Also, the number of recovered images from captions clearly increases with respect to the depth of the network. This is shown in the right bar chart in Figure 5.12, where we can clearly see a linear increasing trend for the red bars, which represents how many images were recovered when trained with 50 captioned images.

Since the networks of depth 11 have shown the best results, for the rest of the project I fixed the network depth to 11.

In Figure 5.13, 10 recovered images from captions are shown to visually illustrate

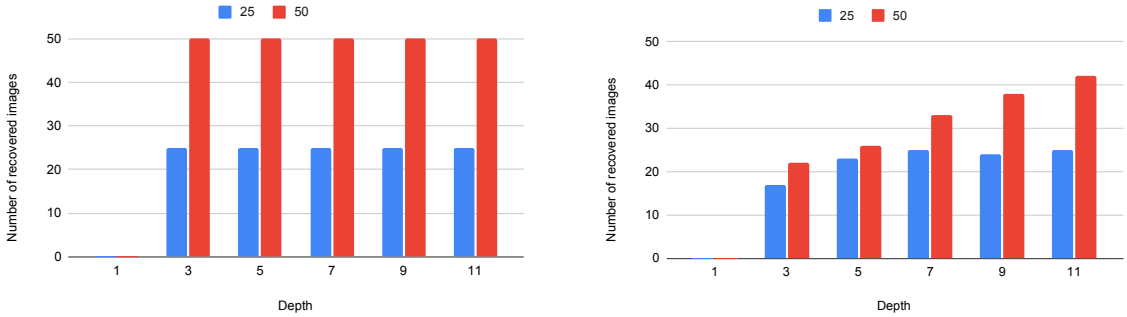


Figure 5.12: Depth comparison for PCNs, the number of captions recovered is shown on the left and the number of images recovered is shown on the right.

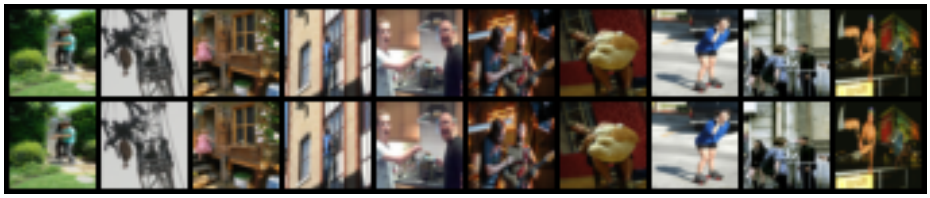


Figure 5.13: Recovery of 10 $3 \times 32 \times 32$ colored images from captions, with upper images being the original ones and bottom images being the recovered ones.

and confirm the recovery. In the upper row, the original images are shown, and in the lower row the recovered images are shown. One can visually confirm that the recovered images are quite close to the original images. In fact, if we subtract those images and paint the noise, it shows like a black image (all pixels are very close to 0). Their corresponding captions are recollected below:

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 .

4. Someone in a blue shirt and hat is standing on stair and leaning against a window .

5. Two men , one in a gray shirt , one in a black shirt , standing near a stove .

6. Two people in the photo are playing the guitar and the other is poking at him .
7. A man sits in a chair while holding a large stuffed animal of a lion .
8. A girl is on rollerskates talking on her cellphone standing in a parking lot .
9. An asian man wearing a black suit stands near a dark-haired woman and a brown-haired woman .
10. Two men in Germany jumping over a rail at the same time without shirts .

Num images	Colored	Greyscale
10	10	10
25	25	25
50	50	50
100	5	1

Num images	Colored	Greyscale
10	10	10
25	25	25
50	42	42
100	57	58

Table 5.3: Number of recovered captions from images (left) and number of recovered images (right), for greyscale and colored 32×32 images.



Figure 5.14: Recovery of 25 32×32 greyscale images from captions, with left images being the original ones and right images being the recovered ones.

5.2.2 Greyscale Images Versus Colored Images

For the best depth parameter $D = 11$ found so far, I tried sizing down the images by converting them to greyscale images. That is, now the images have size $32 \times 32 = 1024$ pixels while the captions remain the same size. The results are shown in Table 5.3. The results are quite similar for greyscale and colored images, therefore, sizing down the images did not help in recovering captions nor in recovering images.

These results show that the problem is not in the image size, it is in the captions. When we normalize the captions to $[0, 1]$, they are identical to the images which are also normalized to $[0, 1]$. Consequently, the captions need to be modified in order to improve model performance.

Number of images	Recovered captions	Recovered images
5	5	5
10	10	10
25	25	23
50	50	38
100	16	47

Table 5.4: Number of recovered captions and recovered images of size $3 \times 64 \times 64$ for networks of depth 11.



Figure 5.15: Recovery of 25 $3 \times 64 \times 64$ colored images from captions, with left images being the original ones and right images being the recovered ones. Using threshold 0.005, the second and fourth images are considered to be not recovered.

5.2.3 Increasing Image Size

Using the best model obtained so far, depth $D = 11$, I trained the network for colored images of size $3 \times 64 \times 64$. The results for the best models are shown in Table 5.4. For the model that managed to recover 23 images out of 25, the recovered images are shown in Figure 5.15. This shows that the model is actually capable of memorizing larger captioned images. This is quite impressive since the models are capable of recovering $3 \times 64 \times 64 = 12288$ pixels using captions of size 28. Furthermore, Note that the images look identical to the human eyes, but actually 2 of them is considered to be not recovered by the threshold I have set (0.005). Those two images are the

Scale / Images	1	2	5	10
50	50	50	50	50
100	5	89	100	96

Scale / Images	1	2	5	10
50	42	42	47	47
100	57	55	65	79

Table 5.5: Number of recovered captions (left) and recovered images (right) for different caption normalization scales.

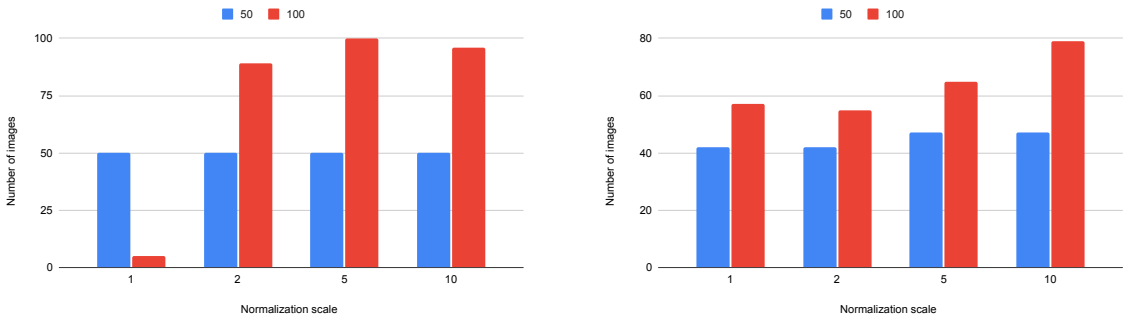


Figure 5.16: For different normalization scales, the number of captions recovered is shown on the left and the number of images recovered is shown on the right.

second and fourth images of the first row. This also confirms the validity of the seemingly random threshold value.

5.2.4 Normalization Scale

As explained in Chapter 4 Section 4.2, when the number of images increases, the captions becomes increasingly difficult to learn due to the reduced distance between words. This is well reflected when we try to use the network to learn 100 captioned images. If we still normalize the captions to $[0, 1]$, then the best model can only recover 5 out of 100 captions. By increasing the normalization scale to $2, 5, 10$, the number of captions learned increases significantly. We can see this effect in Figure 5.16. Increasing the normalization scale to 2 I successfully recover 89 captions, with perfect recovery achieved when I increase the normalization scale to 5.

Secondly, increasing the normalization scale also helps in image recovery, with an overall increase of number of images recovered. This is specially notable for the best models with normalization scale 5 and 10, with 65 and 79 images recovered re-

Num images	PCNs	Autoencoder
25	25	25
50	50	50
100	5	100
250	-	250

Num images	PCNs	Autoencoder
25	25	1
50	42	2
100	57	1
250	-	1

Table 5.6: Number of recovered captions from images (left) and number of recovered images (right), for PCNs and BP auto encoders, for $3 \times 32 \times 32$ images.



Figure 5.17: Recovery of 25 $3 \times 32 \times 32$ colored images from captions, using autoencoders. Left: original images. Right: recovered images. Only 1 image is recovered from captions although we can see that several images are stored.

spectively. This confirms that increasing the separation between images and captions helps with both image recovery and caption recovery.

5.2.5 Comparison with BP Networks

Lastly, I compare the PCNs netword with BP autoencoders, with the following parameters:

- Data: $3 \times 32 \times 32$ colored images with captions normalized to $[0, 1]$.
- Models: depth 11 networks with width $W \in \{512, 1024, 2048\}$, ReLU activation.

The best results for both PCNs and autoencoders are shown in Table 5.6. The

main experimental result is that autoencoders excel at exact recovery of captions when given images. This is probably due to the fact that autoencoders store the images as attractors in the loss function landscape [18]. When given the majority of the data, they are able to converge to the attractor and recover the captions exactly. On the other hand, when captions are given and the majority of the data is removed, the input (captions) are too far away from the stored data points and the network fails to converge to the original data points. Instead, they converge to the nearest attractor. This effect can be seen in the recovery of 25 images, in Figure 5.17.

PCNs are able to recover these 25 images, as we have seen previously. Moreover, the best model with these configurations was able to recover up to 57 images. Nevertheless, PCNs are not able to recover 100+ captions from images, with autoencoders recovering up to 250 captions. I conclude that PCNs excel at approximate recoveries, which is impressive given that they are able to restore images which are in size magnitudes higher than the captions. However, BP autoencoders are better at exact recovery, when given the majority of the data.

5.2.6 Analysis of the Recovery Process

In Figure 5.18 I show 10 recovered images after running inference for {10, 50, 1000, 5000, 30000} iterations. After 10 iterations, the output of the model is basically white noise. This is because of Hebbian plasticity, e.g. the updates are local. In order for the local updates to propagate through the whole model at least the same number of iterations as the depth of the model is needed. After 50 iterations, the white noise is converted into a metastate that is the juxtaposition of the images we intend to recover.

After 1000 iterations, all images are trying to converge to their original form. Although the images are still blurry, we can discern some patterns from the original images. We can also see that some images converge quicker than others, i.e. the last

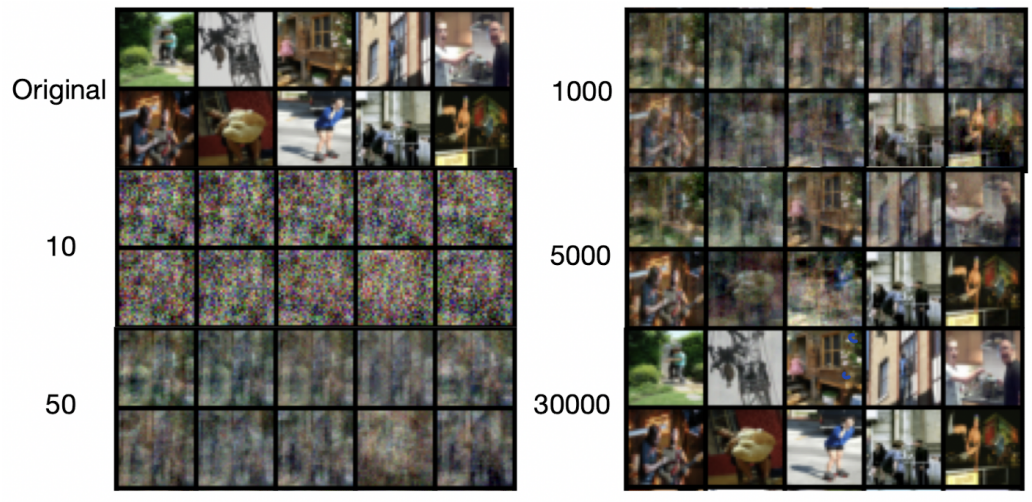


Figure 5.18: Recovery process of 10 $3 \times 32 \times 32$ colored images from captions, using the PCN model of depth 11 width 512 with captions normalized to $[0, 1]$. From top to bottom, on the left I show the original images, the recovered images after 10 and 50 iterations. On the right I show the recovered images after 1000, 5000 and 30000 iterations.

two images are notably clearer. After 5000 iterations most images are close to the original images, and after 30000 images all images were recovered. The second image from the first row and the third image from the second row are the most difficult ones to reconstruct, probably due to the fact that they contain a lot of near white pixels, with values close to 1.

Chapter 6

Conclusions

In this work I have studied a novel type of neural networks, predictive coding networks (PCNs), a more biologically plausible model than BP networks, with a new training procedure called concurrent predictive coding (CPC). I have focused especially on the capabilities of the network with respect to associative memories, the ability of neural networks to memorize data and recover them. This is important because first, brains can memorize data and recover them, and second, memory and learning are interconnected [17]. Furthermore, I have compared experimentally the performance of this model with traditional autoencoders and Dense Associative Memories (DAM).

In terms of auto-associative memory, e.g., the ability to recover the original images starting from corrupted ones, the results show that the PCNs trained with CPC mostly outperforms traditional autoencoders trained with BP. This result is even more accentuated when the network is shallow, e.g., with 1 hidden layer. PCNs have consistently been able to store more images than autoencoders with the same number of parameters, both for smaller `CIFAR10` images and for bigger `TinyImageNet` images. This result is maintained when I increased the network depth, e.g. the number of hidden layers. Lastly, PCNs are able to store realistic images from Imagenet.

In terms of hetero-associative memory, e.g., the ability to recover one type of data

starting from another type of data, the results were mixed. Evaluating both PCNs and autoencoders on captioned images, with the images being 100 up to 400 larger than the captions, PCNs have been able to recover up to 90 images when captions are presented to the network and up to 96 captions when images are presented to the network. On the other hand, BP autoencoders failed to recover images and was only able to reconstruct up to 2 images only. However, when given images, they are able to recover up to 250 captions from the images. This results is due mainly to the fact that we require exact recovery of captions, if we were to allow small mistakes in the captions, then PCNs are also able to recover up to 250 captions. Overall, when the presented partial data is unique (the captions were unique), PCNs were impressive as they were able to reconstruct images magnitudes larger than the presented captions to the network.

An important finding common to both auto or hetero-associative memory experiments is that PCNs do not learn the images as attractors. Instead, when the inference starts, the images decay to a metastate and then converge slowly to the original images using the available information. This can be seen in the experiments conducted with MNIST dataset and in the hetero-associative memory experiments.

With regards to efficiency, PCNs trained with CPC needed less iterations to converge to a stable minimum in the energy function. The training time needed is close to 2 magnitudes lower, and when the network converges it never leaves the local minimum. BP autoencoders, on the other hand, can diverge after the loss function reaches a local minimum. The recovery time of PCNs is higher than autoencoders due to the fact that they need more iterations to converge, however, this does not hurt the biological plausibility of the model since the human mind is much more efficient and the recovery can take longer. Also note that CPC can be fully parallelized internally because the weight updates are local. The obtained efficiency results can be further improved since we only used batch parallelization.

Future Work

Possible future work includes testing more complex models and studying their capacity and performance. This can be done because in order for a network to be a PCN, we just have to add PClayers. These are layers with error nodes which can be added anywhere in the network, and the structure of the network is not relevant. For example, we can include convolutional layers into the PCNs for memorizing images, or recurrent neural networks with incorporated PCLayers for memorizing sentences.

Further comparisons with DAM can also be considered, by implementing DAM from scratch. There is also a recent Dense Associative Neural Network (DANN) [14] that implements associative memories, to which we can compare the network to.

For auto-associative memory experiments, other types of data corruption methods can be tried instead of white noise or partial removal. For example, we can try uniform noise, removing the center part of the image, removing stripes of the image, etc. We can also conduct more experiments with ImageNet or even larger images to further test the capacity of the network. We can also preprocess the data differently, for example, we could use pre-trained feature extractors to convert the images to features before engaging in the learning process.

For hetero-associative memory experiments, instead of using captioned images, we can test other types of multimodal data. For example, audiovisual data or captioned audiovisual data. Also, instead of memorizing one image, we can try to memorize a sequence of images.

Bibliography

[1] Rafal Bogacz. A tutorial on the free-energy framework for modelling perception and learning. *Journal of Mathematical Psychology*, 76:198–211, 2017. Model-based Cognitive Neuroscience.

[2] Mete Demircigil, Judith Heusel, Matthias Löwe, Sven Upgang, and Franck Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168(2):288–299, May 2017.

[3] Pedro Domingos. Every model learned by gradient descent is approximately a kernel machine. *arXiv preprint arXiv:2012.00152*, 2020.

[4] Vitaly Feldman. Does learning require memorization? a short tale about a long tail, 2021.

[5] Vitaly Feldman and Chiyuan Zhang. What neural networks memorize and why: Discovering the long tail via influence estimation, 2020.

[6] Yujian Hong. Associative memories and predictive coding. `https://https://github.com/hzzhyj/msc_project`, 2013.

[7] John Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79:2554–8, 05 1982.

[8] Hubert Ramsauer and Bernhard Schäfl and Johannes Lehner and Philipp Seidl and Michael Widrich and Thomas Adler and Lukas Gruber and Markus Holzleitner and Milena Pavlović and Geir Kjetil Sandve and Victor Greiff and David Kreil and Michael Kopp and Günter Klambauer and Johannes Brandstetter and Sepp Hochreiter. Hopfield networks is all you need blog post, 2021. [Online; accessed 9-August-2021].

[9] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[10] Dmitry Krotov and John Hopfield. Large associative memory problem in neurobiology and machine learning, 2021.

[11] Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition, 2016.

[12] Ya Le and X. Yang. Tiny imagenet visual recognition challenge. 2015.

[13] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[14] Jia Liu, Maoguo Gong, and Haibo He. Deep associative neural network for associative memory based on unsupervised representation learning. *Neural Networks*, 113:41–53, 2019.

[15] Intelligent Systems Oxford. General energy networks. `https://github.com/charlespwd/project-title`, 2021.

[16] Bryan A. Plummer, Liwei Wang, Chris M. Cervantes, Juan C. Caicedo, Julia Hockenmaier, and Svetlana Lazebnik. Flickr30k entities: Collecting region-to-phrase correspondences for richer image-to-sentence models. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 2641–2649, USA, 2015. IEEE Computer Society.

[17] Tomaso Poggio. From associative memories to powerful machines. *CBMM Memo No. 114*, 01/2021 2021.

[18] Adityanarayanan Radhakrishnan, Mikhail Belkin, and Caroline Uhler. Overparameterized neural networks can implement associative memory. *CoRR*, abs/1909.12362, 2019.

[19] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield networks is all you need, 2021.

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[21] Yuhang Song, Thomas Lukasiewicz, Zhenghua Xu, and Rafal Bogacz. Can the brain do backpropagation? — exact implementation of backpropagation in predictive coding networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22566–22579. Curran Associates, Inc., 2020.

[22] James Whittington and Rafal Bogacz. Theories of error back-propagation in the brain. *Trends in Cognitive Sciences*, 23, 03 2019.

[23] James C. R. Whittington and R. Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation*, 29:1229 – 1262, 2017.

[24] Wikipedia contributors. Associative memory (psychology) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=`

Associative_memory_(psychology)&oldid=1000257371, 2021. [Online; accessed 17-April-2021].

[25] Wikipedia contributors. Hopfield network — Wikipedia, the free encyclopedia, 2021. [Online; accessed 4-August-2021].

[26] Peter Wittek. 6 - pattern recognition and neural networks. In Peter Wittek, editor, *Quantum Machine Learning*, pages 63–71. Academic Press, Boston, 2014.

[27] Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *TACL*, 2:67–78, 2014.

Appendix A

Auto Associative Memory Results

CIFAR10 Experiments

CIFAR10	std=0.2		std=0.4	
Width / Num images	1024	2048	1024	2048
100	15	96	13	88
250	19	163	11	153
500	422	447	330	362
750	372	651	176	475
1000	140	541	60	334

Table A.1: Number of images retrieved for BP autoencoders of depth 1 and width $\in \{1024, 2048\}$, for number of images $\in \{100, 250, 500, 750, 1000\}$, and corrupting images with Gaussian noise with standard deviation $\in \{0.2, 0.4\}$.

In Tables A.1, A.2 and Figure A.1 we show the supporting data for the figures created in Chapter 5, Section 5.1.2, and some reconstructed images of CIFAR10 images by bp autoencoders, respectively.

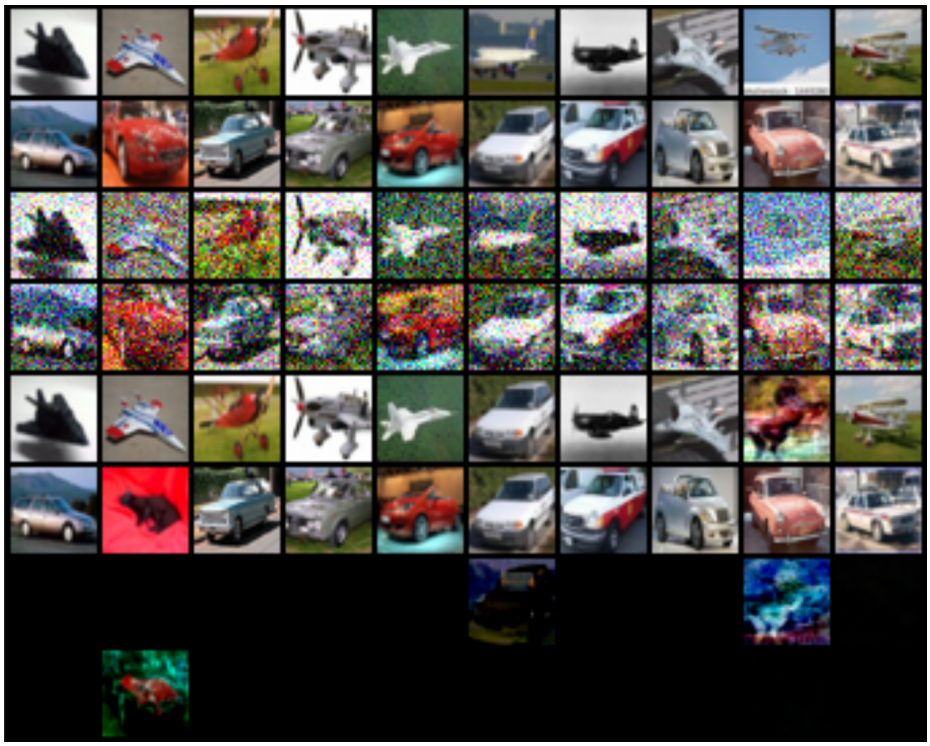


Figure A.1: Recovery of 20 $3 \times 32 \times 32$ colored CIFAR10 images from captions by BP autoencoders. From top to bottom: original images, corrupted images, reconstructed images, and difference between original and reconstructed images.

TinyImageNet Experiments

In Table A.3 the data used for the initial capacity study is shown. In Tables A.4 and A.5 the data used for the depth study is shown. In Tables A.6, A.7, A.8 and A.9 the data used for partial retrieval study is shown.

width / num images	256	512	1024	2048
100	100	100	100	100
250	250	250	250	250
500	484	498	500	500
750	511	560	739	750
1000	470	865	980	982
1500	601	861	1279	1361

Table A.2: Number of images recovered by depth 1 PCNs trained with CPC, for width $\in \{256, 512, 1024, 2048\}$ and for number of images $\in$ {100, 250, 500, 750, 1000, 1500}, corrupted by adding Gaussian noise with standard deviation 0.2.

width / num images	256	512	1024	2048
100	100	100	100	100
250	250	250	250	250
500	405	500	500	500
750	83	654	750	750
1000	74	211	995	1000
1500	96	144	692	1497

Table A.3: Number of TinyImageNet images recovered by depth 1 PCNs trained with CPC, for width $\in$ {256, 512, 1024, 2048} and for number of images $\in$ {100, 250, 500, 750, 1000, 1500}, corrupting images by adding Gaussian noise with standard deviation 0.2.

PCNs	Given 1/2				Given 1/4			
Depth / Width	3	5	7	10	3	5	7	10
1024	162	371	469	492	83	94	246	389
2048	278	470	484	492	121	182	345	354

Table A.4: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}\}$ of the original images are given to the trained PCNs. The networks were trained using 500 \texttt{ImageNet} images, and for width $W \in \{1024, 2048\}$ and depth $D \in \{3, 5, 7, 10\}$.

PCNs	Given 1/2				Given 1/4			
Depth / Width	3	5	7	10	3	5	7	10
1024	130	155	265	370	68	87	160	290
2048	270	320	390	445	90	120	180	245

Table A.5: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}\}$ of the original images are given to the trained BP auto encoders. The networks were trained using 500 \texttt{ImageNet} images, and for width $W \in \{1024, 2048\}$ and depth $D \in \{3, 5, 7, 10\}$.

Width = 512

p/ Num images	1/2	1/4	1/8	1/16
50	50	50	15	0
100	100	6	0	0
250	127	0	0	0
500	0	0	0	0

Table A.6: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ of the original images are given to trained PCNs of width $W = 512$, depth $D = 1$. The networks were trained to memorize $N \in \{50, 100, 250, 500\}$ images.

Width = 1024

p/ Num images	1/2	1/4	1/8	1/16
50	50	50	24	6
100	100	71	58	0
250	178	150	100	25
500	110	10	0	0

Table A.7: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ of the original images are given to trained PCNs of width $W = 1024$, depth $D = 1$. The networks were trained to memorize $N \in \{50, 100, 250, 500\}$ images.

Width = 2048

p/ Num images	1/2	1/4	1/8	1/16
50	50	50	26	5
100	100	100	12	5
250	185	78	0	0
500	205	0	0	0

Table A.8: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ of the original images are given to trained PCNs of width $W = 2048$, depth $D = 1$. The networks were trained to memorize $N \in \{50, 100, 250, 500\}$ images.

Width = 4096

p/ Num images	1/2	1/4	1/8	1/16
50	50	50	22	6
100	100	46	15	0
250	198	25	0	0
500	215	0	0	0

Table A.9: Number of recovered images when a percentage $p \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ of the original images are given to trained PCNs of width $W = 4096$, depth $D = 1$. The networks were trained to memorize $N \in \{50, 100, 250, 500\}$ images.

Appendix B

Hetero Associative Memory Results

Network Depth Comparison

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
5	5	5	5	5	5	5
10	10	10	10	10	10	8
25	25	25	25	16	16	17
50	50	49	10	13	22	21
100	0	0	0	4	3	5

Table B.1: Number of recovered captions and colored 32×32 images for PCNs of depth 3.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	22	23	16
50	50	46	40	23	26	22
100	1	0	0	1	6	5

Table B.2: Number of recovered captions and colored 32×32 images for PCNs of depth 5.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	1	16	25	23
50	50	0	11	27	33	30
100	18	0	0	5	19	23

Table B.3: Number of recovered captions and colored 32×32 images for PCNs of depth 7.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	0	25	24	23	21
50	50	1	1	31	38	36
100	21	8	0	19	35	57

Table B.4: Number of recovered captions and colored 32×32 images for PCNs of depth 9.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	25	25	25
50	45	50	20	36	42	39
100	0	5	1	43	57	55

Table B.5: Number of recovered captions and colored 32×32 images for PCNs of depth 11.

In Tables B.1, B.2, B.3, B.4 and B.5 we present the data used to generate the tables and plots for the hetero-associative memories depth comparison experiment in Chapter 5 Section 5.2.1. The data for PCNs of depth 1 is not shown as all the values of that table are 0.

Greyscale Images Versus Colored Images

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	24	25	25
50	50	50	50	29	42	42
100	1	0	0	33	57	58

Table B.6: Number of recovered captions and greyscale Flickr 32×32 images for PCNs of depth 11.

In Table B.6 the data used for the experiments conducted in Chapter 5 Section 5.2.2 is shown, where how the network performs with respect to greyscale captioned images is studied.

Increasing Image Size

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	20	21	19	23
50	50	45	0	34	37	38
100	16	1	2	23	42	47

Table B.7: Number of recovered captions and colored Flickr 64×64 images for PCNs of depth 11.

In Table B.6 the data used for the experiments conducted in Chapter 5 Section 5.2.2 is shown, where how the network performs with respect to colored 64×64 captioned images is studied.

Normalization Scale

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	25	25	24
50	50	50	50	41	42	40
100	89	6	0	46	52	55

Table B.8: Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 2]$.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	25	25	24
50	50	50	50	47	43	29
100	100	1	9	65	58	57

Table B.9: Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 5]$.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	25	25	24
50	50	48	50	47	47	29
100	96	90	18	79	72	73

Table B.10: Number of recovered captions and colored Flickr 32×32 images for PCNs of depth 11, with captions normalized to $[0, 10]$.

In Tables B.5, B.8, B.9 and B.10 the data used to compare the performance of the model when we normalize the captions to different ranges is shown.

Comparison with BP Autoencoders

In Tables B.11 and B.5 the data used to compare BP autoencoders with PCNs is shown.

	Recovered captions			Recovered images		
width / num images	512	1024	2048	512	1024	2048
25	25	25	25	2	1	2
50	50	50	50	1	2	1
100	100	100	100	1	1	1

Table B.11: Number of recovered captions and colored Flickr 32×32 images for BP autoencoders of depth 11, with captions normalized to $[0, 1]$.