TÉCNICO LISBOA

Multi-Modal Associative Memory: A framework for Artificial Intelligence

Rodrigo Galante Branco Machado de Simas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Andreas Miroslaus Wichert

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Andreas Miroslaus Wichert
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

April 2022

Acknowledgments

I would like to thank my supervisor Prof. Andreas Wichert and co-supervisor Luis Sá Couto. Thank you for giving me the freedom to explore ideas, for always listening, and for promoting open discussions about math, science, philosophy, and life. It was a pleasure to work with you, thank you for all the support, insights, and sharing of knowledge.

I am very grateful to Prof. Rui Cruz for providing us with the latex template that was used to write this thesis. It saved me and many others a lot of precious time.

I would also like to thank my colleagues Maria and Miguel. In a time when working in isolation became the norm due to the ongoing pandemic, working regularly alongside you as a team was a luxury and a pleasure.

I could not have done it without all my friends. Thank you all for the support throughout this year, especially to Mira, Dani, Fred, Garcia, and Freire. The conversation, trips, shared meals, surf sessions, and League of Legends games kept me going!

A big thank you to all my family. Especially my mum and dad. Thank you for always believing in me, and motivating me to follow my passion. Thank you for the values you passed on to me, and thank you for leading with your example.

Last but not least, to my Girlfriend Mariana. Without her love and support, this thesis would not have been possible. I cannot express in words how much I admire her, thank you for bringing out the best in me.

To each and every one of you – Thank you.

Abstract

Drawing from memory the face of a friend you have not seen in years is a difficult task. However, if you happen to cross paths, you would easily recognize each other. The biological memory is equipped with an impressive compression algorithm that can store the essential, and then infer the details to match the perception. The Willshaw Network (WN), is a model that aims to mimic these mechanisms of its biological counterpart. The usage of this model in practical applications is hindered by the so-called *Sparse Coding Problem*. To advance, prescriptions that transform raw data into sparse distributed representations are required. In this work, we use a recently proposed prescription [1] that maps visual patterns into binary feature maps. We analyze the behavior of the WN on real-world data and gain key insights into the strengths and weaknesses of this model. To further enhance the capabilities of the WN, we propose the Multi-Modal framework. In this new setting, the memory stores several modalities (e.g., visual, or textual) simultaneously. After training, the model can be used to infer missing modalities when just a subset is perceived, thus unlocking many practical applications. As a proof of concept, we perform experiments on the MNIST dataset. By storing both the images and labels as modalities, we were able to successfully perform retrieval, classification, and generation with a single model. Our results highlight the flexibility of the Multi-Modal framework to perform various classical Machine Learning tasks with a single network and provide a big hint on how the field of Associative Memories can advance in practical settings.

Keywords

Associate Memory; Auto-Association; Willshaw Network; Generation; Classification

Resumo

É extremamente difícil desenhar de memória a cara de um amigo que já não vemos há anos. No entanto, se nos cruzarmos conseguimos reconhecermo-nos facilmente. A nossa memória utiliza um algoritmo de compressão impressionante: capaz de guardar apenas aquilo que é essencial, para depois conseguir identificar aquilo que está a ser observado. A *Willshaw Network (WN)* é uma Memória Associativa que tenta replicar estes algoritmos das memórias biológicas. A utilização destes modelos em aplicações práticas enfrenta o famosos *Sparse Coding Problem*. Para avançar, é necessário desenvolver métodos que transformem dados no seu estado natural em representações esparsas. Nesta dissertação, vamos utilizar um método proposto recentemente [1] que transforma imagens em mapas binários de atributos. Uma análise no comportamento da WN em dados do mundo real permitiu-nos perceber melhor este modelo, e evidenciou as suas vantagens e desvantagens. Para ir mais longe com a WN, este trabalho propõe uma estruturação da rede em Múltiplas Modalidades. Neste novo cenário, a memória guarda várias modalidades dos dados simultaneamente. Consequentemente, a memória ganha flexibilidade, uma vez que se torna capaz de inferir uma modalidade em falta desde que um pequeno conjunto de informação seja observado. Para demonstrar a viabilidade da nossa proposta, foram realizadas experiências no *MNIST dataset*. Guardando as imagens e valor simultaneamente, foi possível realizar classificação e geração com qualidade razoável. Os resultados demonstram a flexibilidade da nossa solução para resolver vários problemas de Aprendizagem numa única arquitetura, dando uma grande contribuição à área de Memorias Associativas num contexto prático.

Palavras Chave

Memória Associativa; Auto-Associação; Rede de *Willshaw*; Geração; Classificação

Contents

List of Figures

List of Tables

Listings

Acronyms

1NN	One-Nearest-Neighbour
AE	Autoencoder
AM	Associative Memory
ANN	Artificial Neural Network
CNN	Convolutional neural network
DC	desCode
GAN	Genereative Adversarial Network
HD	Hamming Distance
MLP	MultiLayer Perceptron
MMWN	Multi-Modal Willshaw Network
MSE	Mean Squared Error
NXH	Noisy X-Hot
PRE	Perfect Retrieval Error
SDR	Sparse Distributed Representation
VAE	Variational Autoencoder
WN	Willshaw Network
WW	What-Where

1

Introduction

Contents

To create intelligent machines, understanding the human brain and its mechanisms is a fundamental step. For this reason, the focus of this Master's Thesis was the field of Associative Memories (AMs): a family of Biologically-Inspired Artificial Intelligence models that imitate biological memories [2].

These models train by storing associations between pairs of patterns: Correlated features form synaptic connections between the memory's neurons. This way, a pattern is represented in the memory, by the activation of a population of neurons, and each individual neuron can participate in many populations. A trained memory can then be queried with a pattern: the neurons in the network will fire according to their learned connections, and the resulting population of active neurons will correspond to the memory's response to the query. This task is known as *Retrieval*, and the fact that both the query and the answer are patterns, means that the memory is employing Content-Addressability.

In this work, we focus on the Willshaw Network (WN) memory. This model is a single-layered Neural Network that uses a local Hebbian training rule [3] to store associations between pairs of binary vectors. Despite its simplicity, the model is extremely efficient in terms of its storage capacity, being able to store large numbers of patterns with a fixed amount of neurons, and a small computational effort.

Furthermore, we focus on the task of Auto-Association, i.e. we train the WN to associate patterns back to themselves (similarly to an AutoEncoder). An Auto-Associative memory has great practical uses, such as removing noise from corrupted patterns or reconstructing the whole pattern when only part of it is presented to the memory.

1.1 Problem

Our brains "(...) represent information using a method called Sparse Distributed Representations (SDRs)" [4]. Neuroscience has shown that information in the brain is represented by the sparse activation of groups of neurons in the cortex [5].

In computers, an SDR is a large binary vector, where only a select number of bits are active. When representing complex real-world data, each active bit in the SDR must represent an important feature of the pattern. In a sense, SDRs are informative compressions, and AMs benefits immensely from this property due to their content-addressable nature.

The biggest open question in the field of AMs, which often prevents these models from being used in practice, is the so-called *Sparse Coding Problem*. In short, the memory's performance can only be ensured if the patterns that it stores are SDRs, and this type of representation rarely occur naturally.

To solve the *Sparse Coding Problem* and use the WN on practical settings, encoding prescriptions that transform real-world data into SDRs are required.

In [1], Sá Couto and Wichert propose one of the first sparse encoding prescriptions that addresses this problem. In their work, the What-Where (WW) Encoder is proposed: a biologically-inspired module

that transforms visual patterns from the MNIST dataset of hand-written digits [6] into binary SDRs. In the same paper, preliminary research showed that the codes produced by the WW encoder could very efficiently be stored in auto-association on the Willshaw Network. As a consequence, the opportunity to study the WN in a practical real-world setting emerged, and thus this Master Thesis was set in motion.

1.2 Research Goals

The first Research Goal of this thesis was to perform a thorough analysis of the storage and retrieval of MNIST images in a Willshaw Network. This entailed building a decoder module that transformed SDRs back into images (Fig. 1.1). With an encoder and decoder built, we would be able to study the behavior of the WN on real data and perform experiments such as: (a) Visualise the quality of the memory's reconstructions as the memory is filled up; and (b) Visually examine how the memory responds to noisy cues. Additionally, a trained memory and the Decoding module could be used as a generative model: By querying the memory with novel inputs, one might obtain novel responses from the memory. Decoding said responses back into images might lead to original patterns. Essentially drawing from memory.

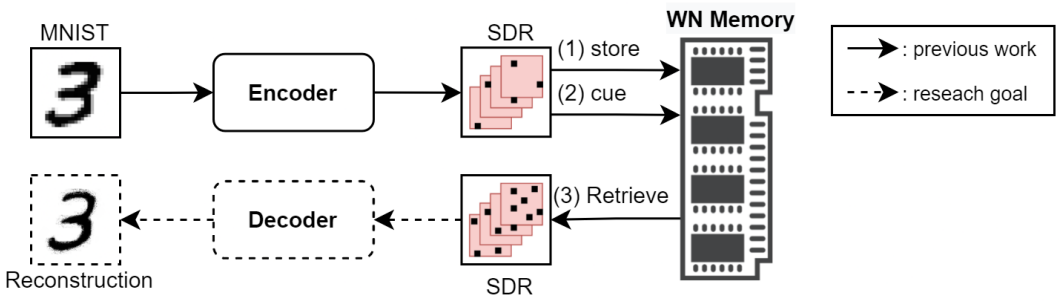


Figure 1.1: Initial Research Goal. In [1] an Encoder that maps MNIST images into SDRs is proposed. These SDRs are stored in the Willshaw Network **(1)**, and retrieved **(2, and 3)**. To complete the pipeline, a Decoder module that transforms the outputs of the memory back into images is required.

To build the decoder module, we tried using Feed-Forward Neural Networks (the MultiLayer Perceptron (MLP), and Convolutional neural networks (CNNs)), and a simpler nativist approach (the WW Decoder). The latter achieved the best results, and thus an extensive analysis of the WN model was made.

This analysis gave us key insights into the strengths, weaknesses, and possible applications of the WN in a practical scenario.

Motivated by the findings of the aforementioned analysis, the crux of this thesis emerges: The idea of the **Multi-Modal Willshaw Network (MMWN)** is proposed. In short, the memory stores several modalities simultaneously. After training, the model can be used to retrieve missing modalities when

just a subset is shown, thus unlocking many practical applications. Where a modality refers to a type of information, e.g. visual or textual.

While simple, this idea has the potential to serve as a flexible framework to solve various Artificial Intelligence Tasks with Associative Memories. Demonstrating the feasibility of this idea became the second Research Goal of this work.

As a proof of concept for the idea of the MMWN, this model was evaluated on the MNIST dataset [6]. By storing both the images and labels as modalities, we were able to perform retrieval, classification, and generation successfully.

The relevance of the MMWN model can be summarized in the following arguments:

- The model has a simple Hebbian training rule that leans in a single pass over the training set and is highly efficient in terms of its storage capacity.
- The model represents information in a distributed manner, thus circumventing the "grandmother cell" problem that is often faced with Neural Networks.
- The model's architecture is not constrained to a particular domain, and a model can hold information from multiple domains (outside the scope of this thesis) simultaneously.
- The model solves several Machine Learning tasks (Retrieval, Classification, and Generation) in a single architecture, while others (CNNs, Variational Autoencoders (VAEs), or Genereative Adversarial Networks (GANs)) specialize in a single task.
- The model unifies several ideas from biology: (a) It utilizes SDRs, the language of information in the neocortex; (b) The WN uses a biologically plausible learning rule, and mimics the biological Associative Memory; and (c) The new architecture exhibits the flexibility to perform several tasks that are deemed essential in Intelligent Systems.

With this work, we will demonstrate the potential of the Willshaw Network Model for practical applications. Furthermore, we propose a novel framework that allows further research to be done in the field. Next, we outline the structure of the rest of this document.

1.3 Thesis Outline

This work is structured as follows:

- A thorough literature review on the topics of Associative memory and Sparse Distributed Representations (Sections 2.1 to 2.3).
- An overview of the techniques and models relevant to the problem of this thesis (Sections 2.4 to 2.6).

- An analytical report on the experimental work done in this thesis:
  - A study of the retrieval process of the Willshaw Network (Chapter 3).
  - An experimental analysis on the reconstruction of the contents of the Willshaw Network (Chapter 4).
  - A proposal of a new framework for artificial intelligence, and an experimental analysis to demonstrate its feasibility (Chapter 5).
- A conclusion, a reflection of the broader impact, and discussion about future work possibilities enabled by this thesis (Chapter 6).

2

Background

Contents

In this chapter, we provide a thorough literature review on the subject of Associative Memories. Additionally, we present the open question of this field and describe methods and tools that can be used to address said questions.

2.1 Associative Memory

AMs are a family of Artificial Neural Networks (ANNs) that store associations between pairs of vectors. If we consider these vectors to be abstractions of concepts or objects, learning and storing said associations closely resembles the way human memory operates. As Daniel Kahneham, Nobel prize winner puts it: "Psychologists think of ideas as nodes in a vast network, called associative memory, in which each idea is linked to many others." [2]. We can introspectively realize that our memory works this way when we try to recall some piece of information and notice that to reach the answer we must traverse a chain of associations. In this way, AMs are biologically inspired computational models in the sense that they try to incorporate these high-level functions of biological memories. Additionally, these models implement neuro-physiological mechanisms of the brain such as the rule for synaptic plasticity proposed by Donald Hebb [3].

It is important to underline that AMs are quite different from traditional artificial memories, like those found in computer disks. In a computer, a memory is a list of addressed information: we provide an address and obtain its content back, the memory employs *location addressing*. AMs do not utilise addresses explicitly, we provide a *question* content vector and obtain an *answer* content vector back. As explained by Palm [7], AMs are *mapping memories* and they support *content addressability*.

Formally, an AM stores a set S of M pairs (x, y), where x and y are the *question* vector and the *answer* vector respectively:

$$S = \{(x^\mu, y^\mu) \mid \mu = 1, \dots, M\} \tag{2.1}$$

The AM establishes a mapping $(\mathrm{x}^\mu \to \mathrm{y}^\mu)$ which is denoted *hetero-association*. In the special case where $x = y$ the memory is said to perform *auto-association*. Mapping a vector to itself in auto-association has great practical uses. Namely, removing noise from corrupted patterns or reconstructing the whole pattern when only part of it is presented to the memory. The pairs of patterns are not stored explicitly in the memory, it is the correlations and anti-correlations between the pairs of patterns that are learned. This key feature of AMs acts as a natural regularization mechanism and allows a trained network to generalize for novel patterns that are not stored in memory. The combination of hetero and auto association capabilities of an AM allows it to naturally implement functions of biological memories. The abilities of AMs have been studied for over fifty years now by many authors such as Palm [8] and Kohonen [9]. In [10], Wichert summarizes the core abilities of AMs as follows:

- The ability to correct false information.

- The ability to complete information in case only part of it is known.
- The ability to interpolate information: When novel information is shown, the most similar stored sub-symbol is determined.

The two main steps involved in operating an AM are *learning* and *retrieval*. Learning is the step in which the set of associations is stored and a mapping established. In the retrieval step, the mapping is used to perform association. Several types of AMs implement these steps differently. Here we skip the details and focus on the overall idea of each step.

2.1.1 Learning

An AM is typically represented as a *synaptic weight matrix* W of real numbers. In a network with n neurons each having m connections, W will represent the *weights* of the $m \times n$ connections between all neurons. Initially the memory will be empty ($W_{ij} = 0 : i = 1, \ldots, m; j = 1, \ldots, n$). The learning process consists in presenting M pairs (x^μ, y^μ) to the network, which can result in local updates to the synaptic weights of each individual neuron:

$$W_{ij} = \sum_{\mu=1}^{M} R\left(x_i^\mu, y_j^\mu\right) \tag{2.2}$$

Where R is the *synaptic weight update expression*, which takes as input the pre-synaptic and post-synaptic signals x_i and y_i, respectively, and outputs the updated weight value W_{ij}. An example of a *synaptic weight update expression* is the Hebbian rule [3] for binary vectors:

$$R\left(x_i, y_j\right) = x_i.y_j \tag{2.3}$$

The two major families of learning rules are *additive* and *binary*. In additive learning W is updated successively with Equation (2.2). In binary learning the weight matrix is clipped so that a simpler binary matrix is obtained at the cost of some information loss:

$$W_{ij} = \sum_{\mu=1}^{m} H\left(R\left(x_i^\mu, y_j^\mu\right)\right) \tag{2.4}$$

where *H* is the *Heaviside* function defined as:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \tag{2.5}$$

In the AM that we will study, the WN, both the pairs (x, y) and the W matrix are binary. This means that a binary learning rule is employed.

2.1.2 Retrieval

Associative retrieval is the process of showing a *cue vector* to the already-trained memory, letting the network compute its final state, and obtaining the *retrieved vector*. Figure 2.1 illustrates this process.

(a) Stored Vector **(b)** Cue vector **(c)** Retrieved vector

Figure 2.1: Example of associative retrieval. A memory that stores 160 different pictures of faces, when given a small snippet containing just the eyes, is able to retrieve the original image. **(a)** One example out of the 160 images stored in the memory. **(b)** The cue vector that is used as input to the retrieval process. **(c)** The retrieved vector that is obtained after the memory computes its final state. Adapted from [9], original study in [11].

When the AM receives as input a *cue vector* $\tilde{\mathbf{x}}$, each neuron of the network will compute its dendritic potential s_j.

$$s_j = \sum_{i=1}^{m} W_{ij}\tilde{x}_i \tag{2.6}$$

Notice that this step can be parallelized since each neuron works independently of the others.

Next, each neuron computes its output $\hat{y}_j$ by applying a *transfer function* f, with threshold θ_j, to its previously computed dendritic potential.

$$\hat{y}_j = f\left(s_j, \theta_j\right) \tag{2.7}$$

The *transfer function* f usually takes the form of a *step function*, like the *Heaviside* function described in Equation (2.5). The choice of the *threshold* parameter θ, which controls the sensitivity of activation, is of extreme importance: an appropriate rule can greatly enhance the memory's performance, allowing for the storage of more vectors with no penalty on the retrieval quality. If θ is set too high, very few neurons will activate, even with strong dendritic potential. On the other hand, if θ is set too low, many neurons will activate in a "false-positive" manner. A well-suited value for θ will ensure that the correct amount of correlations between neurons and the *cue vector* are detected, resulting in an error-free *retrieval*.

Some rules define a global *threshold* $\theta_j = \theta_{global} \mid j = 1, \ldots, n$ shared by all neurons, others rely on

each neuron to store its activation threshold locally.

After all the neurons have computed their state y_j, the state of the whole network represents the retrieved vector y. When the retrieval process ends here, it is called *one-step retrieval* which occurs on feed-forward AMs. However, certain AMs, like the Hopfield Network [12], employ *recurrent* architectures. In such cases, the state y computed in the first step of *retrieval* is fed back into the network as input, and the retrieval process is repeated until convergence. This method is known as *iterative retrieval*.

2.1.3 Performance Assessment

The performance metric of associative memories is its *storage capacity*. After all, a memory system is as good as the number of different memories it can hold. Any associative memory can hold a small number of vectors and retrieve them perfectly, but as more vectors are stored, the memory starts to fill up and retrieval becomes imperfect (see Figure 2.2 for an illustration). If too many vectors are added to the memory it can become saturated and rendered useless, not even able to retrieve the first learned vectors.

(a) Stored Vector

(b) Retrieved: memory with 160 vectors

(c) Retrieved: memory with 500 vectors

Figure 2.2: Effect of the number of stored vectors on retrieval quality. In this experiment, similarly to Figure 2.1, a small part of the image is used as the cue vector for retrieval. **(a)** The original vector that is stored in memory. **(b)** The retrieved vector obtained when the memory is filled with 160 vectors. We can notice some small imperfections in the retrieved image. **(c)** The retrieved vector obtained when the memory is filled with 500 vectors. The high number of stored vectors is causing a large retrieval error, the retrieved image has noticeable artifacts. Adapted from [9], original study in [11].

Formally, the storage capacity L, admitting an *error factor* ϵ of an AM with n neurons, is the total number of pairs (x, y) that can be stored in memory without experiencing a retrieval error greater than ϵ. Typically described as a function of the size n (number of neurons) of the network.

For example, the Linear Associator (studied independently by [13], [14], [15], or [16]), a very simple AM, had a storage capacity of $L = n$ when storing orthogonal vectors. Another example is the original

version of the Hopfield Network, which had a capacity of $L = 0.14n$ for uncorrelated vectors. Notice that these theoretical values for the storage capacities of AMs are often studied under ideal scenarios, such as storing uncorrelated or orthogonal vectors. As it turns out, vectors with certain properties are very well suited for storage in AMs, resulting in great storage capacities. However, real-world data typically lacks those properties, which is the main cause for the shortcomings of AMs in practical applications. Finding mechanisms to transform real-world data into "AM-suitable" vectors is the key to unlocking the full potential of AMs. We will discuss the desired properties of vectors in Section 2.3, and the existing mechanisms to generate them in Section 2.6.

2.2 Willshaw network

The Willshaw Network (WN) [17] is a shallow, feed-forward ANN that performs *auto* and *hetero-association* of binary vectors.

Originally named *Lernmatrix* by its creator Karl Steinbuch in the 1950s [18], the biologically-inspired model was one of the pioneer implementations of ANNs. The Lernmatrix became more popular in the following decade when it was studied through a mathematical/biological lens by D. Willshaw [17]. Thus becoming known as the *Willshaw Network*. Despite its simplicity, both in its binary nature and in its learning/retrieval rules, the WN is a great artificial memory, capable of storing large numbers of vectors.

2.2.1 Architecture

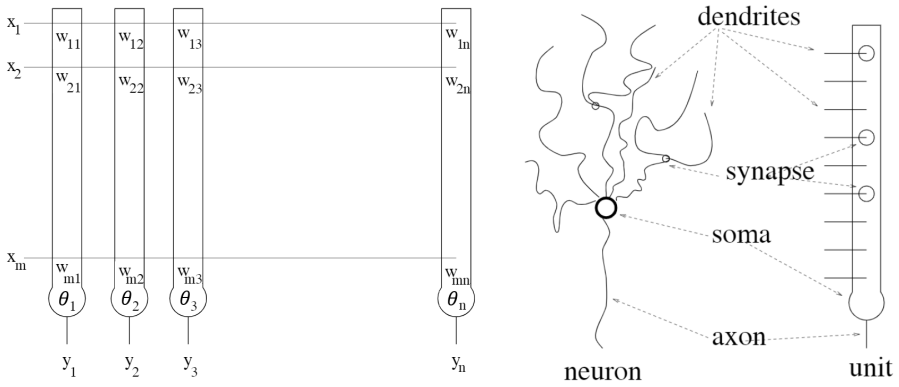
The WN is composed of n units, also referred to as *neurons*. Each neuron is connected to the input layer through m binary connections. Neurons are not interconnected. A schematic view of the network can be seen in Figure 2.3(a).

The network is represented as a binary matrix W:

$$W_{ij} \in \{0, 1\} : i = 1, \dots, m; j = 1, \dots, n \tag{2.8}$$

The dimensions m and n are given by the fixed sizes of the *question* and *answer* vectors, respectively. Notice that, in an auto-association task, W will be a square matrix of size $(n \times n)$. This matrix defines the absence/presence of connections between neurons. A connection between two neurons i and j is formed during training when a local Hebbian rule detects a correlation between the positions i of a *question* vector, and the position j of an *answer* vector.

Additionally, each neuron has a local threshold T_n, used in the last step of retrieval as the parameter of the *transfer function* (recall Equation (2.7)).



(a) Willshaw Network schematic. **(b)** Biological vs Artificial neuron.

Figure 2.3: Neurons of the Willshaw Network. **(a)** A schematic representation of a *Willshaw Nework*. Composed of a set of n vertical units which represent a simple model of a real biological neuron. Each neuron has m weights, which correspond to the synapses and dendrites in the real neuron. In this Figure they are described by $w_{ij} \in \{0, 1\}$ where $1 \leq i \leq m$ and $1 \leq j \leq n$. θ_j is the threshold of each neuron j. **(b)** A side-by-side comparison of a biological neuron (left) and a Willshaw Netork's unit (right). Adapted from [10].

2.2.2 Biological plausibility

The human brain is the most intelligent system that we know of. Understanding the brain and implementing its mechanisms is the ultimate goal of Biologically inspired models. AMs are examples of such models as they try to imitate the associative capabilities of the brain.

The *Lernmatrix/Willshaw Network*'s development has always been guided by biology and neuroscience. The original purpose of the model was to study psychological conditioning [18, 19]. The model was later analyzed by Palm [20] on its ability to implement assemblies of cells under the Hebbian framework [3]. Furthermore, the network's units are artificial neurons [21] modeled after their biological counterparts (see Figure 2.3(b)). Finally, the neural energy efficiency of the model has been analyzed [22,23].

All in all, the WN, while far from a replica of the brain, is heavily biologically inspired. Consequently, its application in practical scenarios is significant, since it might give us interesting insights into our still limited understanding of the brain.

2.2.3 Learning and Retrieval rules

The WN is completely binary: its inputs x, outputs y, and weight matrix W are binary. Consequently, its learning and retrieval rules are simple.

For the next equations, let us use the following convention:

- The n neurons of the network are referenced with the index j.

- The m connections of each neuron are referenced with the index i.
- The weight matrix W has m rows and n columns (see Figure 2.3(a)), and it is indexed as W_{ij}.

The learning rule is directly derived from Equation (2.4). It uses the Hebbian rule (Equation (2.3)) as its *synaptic weight update expression*. Given a set of M pairs (x, y), the weight matrix W is computed with the following binary learning rule:

$$W_{ij} = \sum_{\mu=1}^{m} H\left(x_i^\mu y_j^\mu\right) = \min\left(1, \sum_{\mu=1}^{M} x_i^\mu y_j^\mu\right) \tag{2.9}$$

notice that the Heaviside function $H(x)$ (Equation (2.5)) is equivalent to $\min(1, x)$ due to the binary nature of the model.

The retrieval of a cue vector $\tilde{x}$ is performed in two steps. First the dendritic potential s is computed in each neuron:

$$s_j = \sum_{i=1}^{m} W_{ij}\tilde{x}_i \tag{2.10}$$

Secondly, the retrieved vector $\hat{y}$ is computed for all neurons:

$$\theta_j = \max_{1 \le i \le n} s_i \tag{2.11a}$$

$$\hat{y}_j = H\left(s_j - \theta_j\right) \tag{2.11b}$$

Equation (2.11a) is known as *soft thresholding*. It defines a global strategy for the transfer function's activation threshold θ. This strategy was used by Palm et al., Schwenker, Sommer, and Strey in [24, 25]. Soft thresholding is the best performing strategy for the WN. Alternatively, one could use a simpler strategy, such as *hard thresholding*, where θ is set globally to the number of ones in the cue vector: $(\theta_j = \sum_{i=1}^{m} \tilde{x}_i \mid j = 1, \ldots, n)$. In Equation (2.11b), H refers to the *Heaviside* step function described in Equation (2.5).

Equations (2.9) to (2.11) fully specify the computations required to operate a WN. Next, we provide a simple example to better illustrate the learning and retrieval process.

2.2.3.A Example of Willshaw Learning and Retrieval

Consider that we want to store two patterns in *auto-association* in a WN:

$x^1 = (0, 0, 1, 1)$

$x^2 = (1, 1, 0, 0)$

Since this is an *auto-association* task, we have $y = x$:

$y^1 = (0, 0, 1, 1)$

$y^2 = (1, 1, 0, 0)$

The **learning step** is done by applying the rule defined in Equation (2.9):

$$W_{ij} = \min\left(1, \sum_{\mu=1}^{M} x_i^\mu y_j^\mu\right) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Notice how W is a square and symmetrical matrix, which is always the case in *auto-association* tasks.

Now, consider that we query the network with the following *cue* vectors:

$\tilde{x}^1 = (1, 0, 1, 1)$ - a noisy version of the pattern x^1.

$\tilde{x}^2 = (1, 0, 0, 0)$ - an incomplete version of the pattern x^2.

The **retrieval step** begins by applying Equation (2.10) to each *cue*, resulting in the following dendritic potentials:

$s^1 = (1, 1, 2, 2)$

$s^2 = (1, 1, 0, 0)$

Next, we can determine the local activation thresholds θ using Equation (2.11a):

$\theta^1 = 2$

$\theta^2 = 1$

Finally, we use Equation (2.11b) to obtain the *retrieved vectors*:

$\hat{y}^1 = H((1, 1, 2, 2) - 2) = H(-1, -1, 0, 0) = (0, 0, 1, 1) = x^1$

$\hat{y}^2 = H((1, 1, 0, 0) - 1) = H(0, 0, -1, -1) = (1, 1, 0, 0) = x^2$

Notice how the memory was able to perfectly reconstruct the original vectors x^1 and x^2, when corrupted versions of them were used as *cues*.

The process that is illustrated by this simple example with 4-dimension vectors is the same process that is depicted in Figure 2.1, where the vectors are now large binary arrays of pixels that represent black and white images.

2.3 Sparse Codes

The inputs and outputs of an Associative Memory (AM) are vectors (also commonly referred to as patterns or codes). Here, we formally describe what vectors are, and what are their interesting properties

in the context of AMs.

2.3.1 Vectors

An n-dimensional vector x is a collection of n components x_i. In the case of AMs, these components are usually *discrete values*. More specifically, the AMs that are purely intended for association tasks deal with discrete *binary* vectors.

An n-dimensional *binary* vector x is a collection of size n where each component x_i can only take one of two values: $x_i \in \{a, b\} \mid i = 1, \ldots, n$. Any two literals or numbers can be used, for instance $a = W$ for white pixels, and $b = B$ for black pixels. More commonly, discrete values are used such as $(a, b) = (-1, +1)$ in the original Hopfield Network, or $(a, b) = (0, 1)$ in the Willshaw Network.

From this point on, we will focus solely on binary vectors, and the convention $(a, b) = (0, 1)$ is adopted.

2.3.2 Properties of binary vectors

When using *binary* vectors, the amount of 0s (inactive bits) and 1s (active bits) in the vector is a key property. Given a vector x of size n, let the ratio p denote the relative proportion of 1s in the vector:

$$p = \frac{\text{number of 1s}}{\text{size of x}} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2.12}$$

Another useful way to interpret p is the probability that a given position of the vector, chosen at random, is active.

Concerning the amount of 1s in a binary vector, it can be classified as [26]:

- *Singular* if, out of the n bits, there is a single 1 and the remaining $n - 1$ bits are 0s. This type of representation is also referred to as *one-hot encoding*.
- *Sparse* if the relative amount of 1s is very small ($p \ll 0.5$). In mathematical terms, a *sparse* vector obeys: $\lim_{n\to\infty} p = 0$. For example, a *singular* vector has: $\lim_{n\to\infty} p = \lim_{n\to\infty} \frac{1}{n} = 0$ which is *sparse*.
- *Non-sparse* if the relative amount of 1s is away from zero. In other words, a *non-sparse* vector has: $\lim_{n\to\infty} p = K$, with $K \neq 0$. For instance, *random* binary vectors have approximately as many 0s as 1s, consequently we have: $\lim_{n\to\infty} p = \lim_{n\to\infty} 0.5 = 0.5 \neq 0$ which is *non-sparse*.

Associative memories perform better when storing sparse vectors. For instance, a Hopfield Network can store L = 0.72n sparse vectors [27]. But in the case of non-sparse vectors, the capacity drops to L =

0.14n [12]. This trend is common amongst other AMs such as the Wilshaw Network.

Additionally, in the context of the entire set S (Equation (2.1)) of vectors to be stored in memory, the desired properties of S are the following:

- S should be *uniformly sparse*, meaning that all vectors in S should have similar sparsity (besides also being individually sparse).
- S should be *distributed*, meaning that all the n positions of the vectors are uniformly active across all the vectors in S.

The property of *distribution* is better understood visually. Let us introduce a new definition to better illustrate it:
For a set S of M binary vectors, and a given position of the vectors i, let P_i be a random variable that denoted the ratio of vectors x, in S, that have a 1 in the ith position.

$$P_i = \frac{1}{M} \sum_{\mu=1}^{M} x_i^{\mu} \tag{2.13}$$

A *distributed* set of vectors is one that has a similar P value for all the n positions of the vectors in it. Figure 2.4 illustrates the *distribution* and *sparsity* properties of binary vectors.

A simple way to quantify the level of distribution of a binary set is to measure its Shannon Entropy [28]. The Shannon Entropy measures the level of uncertainty inherent to a random variable's possible outcomes. For a set S, of binary vectors of size n, where P is a random variable with n possible outcomes, its Shannon Entropy H is given by:

$$H(P_i) = -\sum_{i=1}^{n} P_i \log_2 P_i \tag{2.14}$$

A well-distributed binary set will use all the positions of its vectors with similar probability, resulting in a higher level of uncertainty, which corresponds to a small Shannon Entropy value.

When the training vectors meet the criteria of *sparsity*, *uniform sparsity*, and *distribution*, we can refer to them as *Sparse Distributed Representations*. This kind of vector can very efficiently be stored in AMs, allowing for great storage capacities.

2.3.3 SDR

Our brains "(...) represent information using a method called *Sparse Distributed Representations*, or SDRs." [4]. Neuroscience has shown that information in the brain is represented by the sparse activation of groups of neurons in the cortex [5]. In this way, SDRs are biologically plausible informative

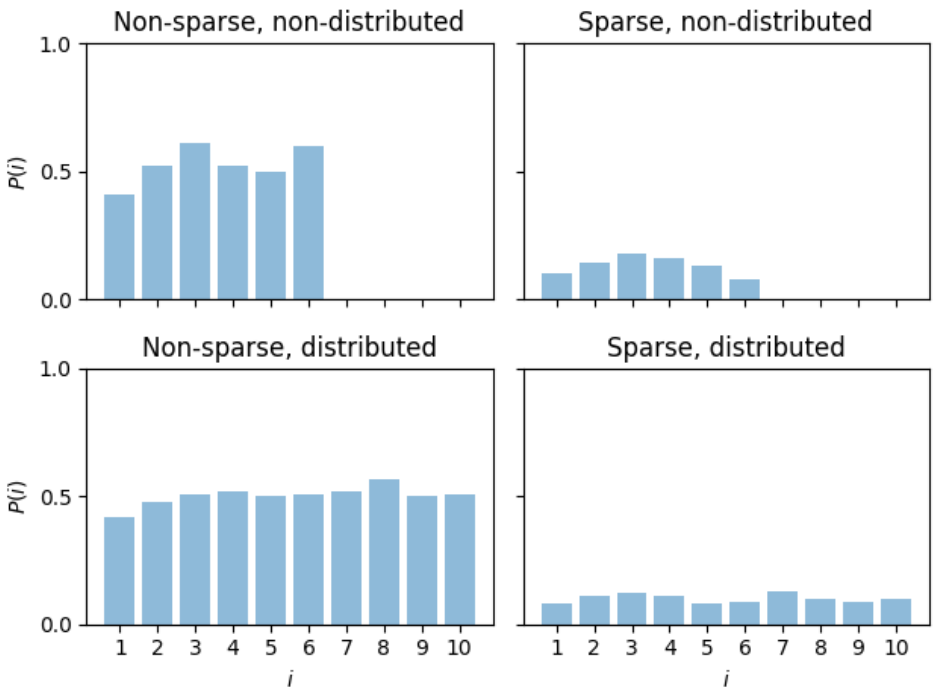


Figure 2.4: The properties of a binary vector training set S (Equation (2.1)) described with P (Equation (2.13)). In this example we are storing 10-dimension vectors in the AM. The x-axis represents the 10 positions i of the vectors. The y-axis represents $P(i)$, which is the probability that we find a 1 in the ith position of a vector of S, picked at random. Two properties of S (sparsity and distribution) are shown in all 4 possible combinations. **(top row)** Non-distributed: the 1s in S are not equally distributed in the 10 dimensions. **(bottom row)** Distributed: the 1s in S are equally distributed in the 10 dimensions. **(Left column)** Non-sparse: There is a high amount of 1s in S **(Right-Row)** Sparse: The amount of 1s in S is limited.

representations.

This kind of representation is restricted by the *sparsity* and *distribution* properties. On one hand, these restrictions make SDRs less compact representations (more bits are required), but on the other hand, these same restrictions give rise to representations where each bit captures a relevant detail/feature of the training set S. Since the vectors are sparse, only the prime features of each training example will be selected to be active in the representation. In such a way, SDRs also act as compressions.

The most important property of SDRs is that each of its bits has meaning [4]. If two representations of different items share an active bit in the same position, it means that the items themselves share some semantic similarity. The information is carried in the representation itself, not stored externally. This property is what makes SDRs informative representations. Since AMs are *content addressable* memories, i.e. they handle inputs as content/information, they benefit immensely from this SDR property.

The informative nature of SDRs becomes evident when we compare SDRs with the dense representations commonly found in computers. A dense representation is a non-informative compact way to encode information. It uses the least possible amount of bits to assign a unique code to each item. A

table containing the mapping between the codes and the items is established and this table must be looked up to make sense of the representations. The information resides in the table, not the codes. Consider the following example in Section 2.3.3.A

2.3.3.A Example: Dense Representation vs SDR

Let us compare two different approaches for encoding numbers: a dense representation and an SDR.

The ASCII representation for character encoding is dense. It uses all the 2^7 possible combinations of 7 bits to code 128 different characters. The mapping between codes and characters is stored in the ASCII table (see Table 2.1). Every character is assigned a unique 7-bit code. The individual bits of the code carry no meaning. Only the complete combination of the 7 bits allows us to determine which character is being encoded. All the information is stored in the table, not the codes.

Table 2.1: Example of entries in the ASCII table.

character	code(dec)	ASCII code
1	49	0110001
2	50	0110010
3	51	0110011
...	...	...
6	54	0110110

Now, let us consider an SDR for number encoding. Figure 2.5 depicts a simple encoder that mimics how the human cochlea encodes frequency in the inner ear. The representations produced by this encoder are known as the *thermometer encoding* [29], which are sparse and distributed (see Table 2.2 for examples). The main idea is that the several bits in the representation correspond to sensors that activate for specific values. A value will activate both the corresponding sensor and its neighbors, thus producing informative representations.

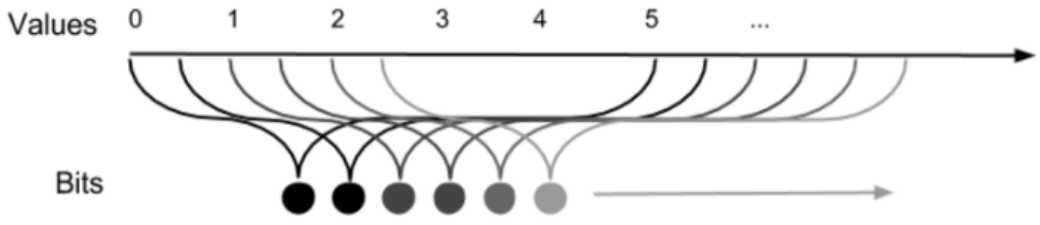


Figure 2.5: The cochlea SDR encoder for numbers. Each bit in the representation responds to a range of values that overlaps with its neighbors. Here, the representations have 105 bits, and only 6 are active at a time, guaranteeing a sparsity of approximately 5%. This encoder can code numbers from 0 to 100. Notice how the encoder will produce distributed codes, i.e. the entire 105 bits will be used uniformly (see Figure 2.4)). Adapted from [4].

Notice how the thermometer encoding carries information in its bits. Small numbers will use the left side of the vector while large numbers will use the right side. Two numbers that are close in value will

Table 2.2: Example of entries in an SDR encoding strategy. See Figure 2.5 for the encoding strategy.

number	105-bit code
6	1111110000000...000
7	0111111000000...000
8	0011111100000...000
...	...
13	0000000111111...000

share bits in their representation (for example 6" and 7 have five overlapping 1s), while numbers that are far apart will not share any bits (for example 7 and 13 share no active bits). In the ASCII representation, overlaps between different representations have no meaning. The representation of a 2 differs in a single bit from both the representation of a 3 and a 6. Overlapping bits are just coincidences.

The thermometer encoding is also resistant to noise: If a few 1s are removed or added to an existing representation, we can still know if the number is large or small by looking at the overall position of the active bits. In the ASCII representation, switching a single bit completely alters the information.

The informative nature of SDRs also means that we can compare two representations and assess their similarity with no knowledge of the domain or the task.

From this example, we can observe how bits carry information in SDRs, and how they don't in dense representations. This great property comes at the cost of size. While the ASCII representation only uses 7 bits, this SDR uses 105. However, SDRs can be implemented with pointer structures that only store the positions of the active bits (see Appendix A.1 for a concrete example).

2.3.4 The sparse coding problem

Associative memories can greatly increase their capacity if they store SDRs. However, it turns out that most real-world data is not naturally sparse (see Figure 2.6). To successfully apply AMs to real-world problems, we must find prescriptions that transform dense representations into SDRs. This problem is known in the literature as *the sparse coding problem*, and it is present in many different domains [30–32]. In the end, it boils down to extracting relevant features from the available data and then compressing that information into a sparse representation. In the upcoming section, we will go over some of the existing solutions for the *sparse coding problem*.

2.4 Sparse Encoding Prescriptions for numbers

In this section, let us look at how we can transform numbers into SDRs. More specifically, integers within a fixed range. For simplicity, consider that we want to encode N distinct integer values, in the fixed range $\{0, \ldots, N-1\}$.

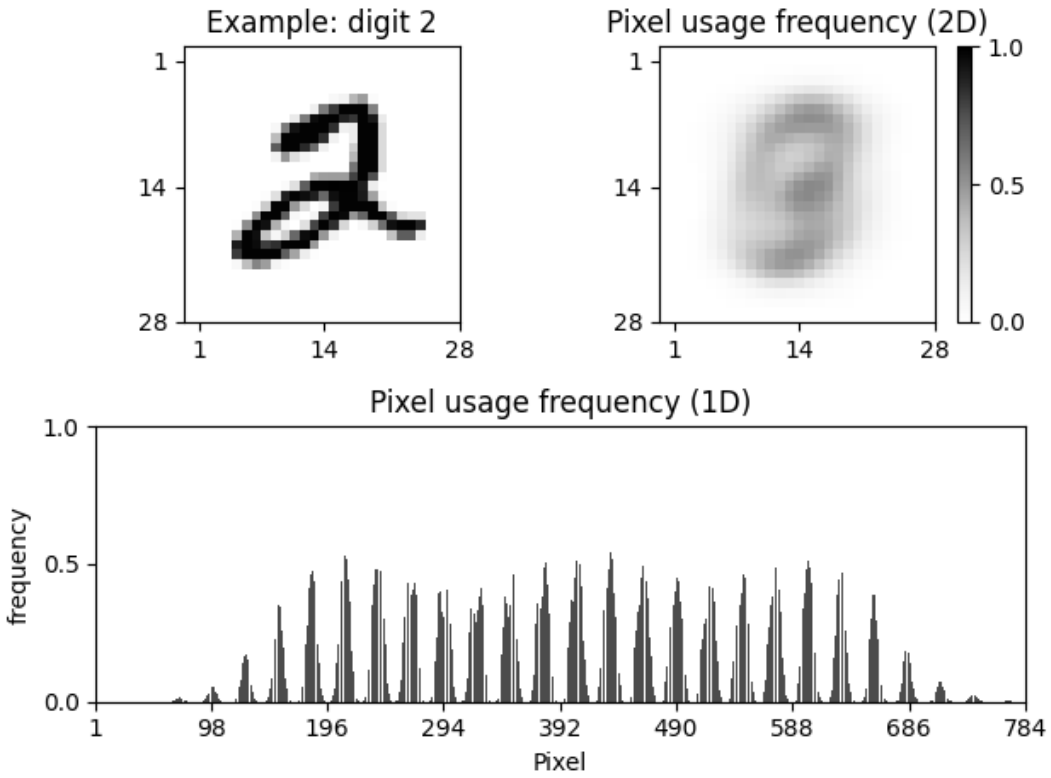


Figure 2.6: Example of the lack of sparsity and distribution on real-world data. Here, the MNIST dataset [6] is used. The dataset contains 70,000 images of handwritten digits, drawn by 250 different writers. Each image is 28 pixels wide and 28 pixels high, totaling 784 pixels per image. **(top-left)** One single example out of the 70,000 images in the dataset. **(top-right)** The relative frequency (encoded with color) at which each pixel is used. **(bottom)** The same pixel frequency depicted in the top-right picture, visualized in a single horizontal axis. Notice how the data is neither sparse nor distributed (compare with Figure 2.4).

2.4.1 One-Hot encoding

A One-Hot encoding is a binary vector of size N, where all the bits are set to zero, except one (hence one-hot). For an integer n in the range $\{0, \ldots, N-1\}$ its one-hot encoding e is given by:

$$e_i(n) = \begin{cases} 1 & \text{if } i = n \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

2.4.2 X-Hot encoding

The X-Hot encoding strategy is a more general version of the One-Hot, X specifies how many bits are active in the code. For an integer n in the range $\{0, \ldots, N-1\}$ its X-hot encoding e is a binary array of size $N \times X$ given by:

$$e_i(n) = \begin{cases} 1 & \text{if } i \in \{z \in \mathbb{Z} \mid nX \leq z < (n+1)X\} \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

2.4.3 Sparse X-hot encoding

Identical to the X-Hot encoding, but the X active bits in the code are padded with P zeros on both sides. P allows us to control the sparsity of the encoding. For an integer n in the range $\{0, \ldots, N-1\}$ its Sparse X-hot encoding e is a binary array of size $(N \times (X + 2P))$ given by:

$$e_i(n) = \begin{cases} 1 & \text{if } i \in \{z \in \mathbb{Z} \mid n(X+2P) \leq z < (n+1)(X+2P)\} \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

2.5 Sparse Encoding Prescriptions for images

Strategies that transform natural data into SDRs are required to solve the sparse coding problem. Here, we refer to such strategies as *encoding functions* or *prescriptions*. The desired properties of an *encoding function* are [4]:

- Semantically similar data should result in SDRs with overlapping active bits.
- The same input should always produce the same SDR as output.
- The output should have the same dimensionality (total number of bits) for all inputs.
- The output should have similar sparsity for all inputs and have enough one-bits to handle noise and subsampling.

An important aspect of Artificial Intelligence (AI) models is their amount of predefined *structure*. AI models can be broadly divided into two categories: those that use "a priori" knowledge in their structure (*Nativism*), and those that don't (*Connectionism*). The first is a traditional approach [33], where the knowledge about the domain of the problem is used to design comprehensible solutions. The second is focused on building general algorithms (usually Neural Networks) that can learn with minimal assumptions/biases about the problem. Due to the huge diversity of available AI techniques, there isn't a binary way to classify models in terms of their structure, instead, there is a wide spectrum. In the *Nativism* end of the spectrum, we have techniques such as first-order logic models [34] where the model's structure is strictly predefined. On the other end, we have *Connectionism* techniques, such as the multi-layered perceptron (MLP) [35], where the structure is not imposed beforehand, but learned through the weights of the model. In between these two extremes, there is a large diversity of models that have both Nativism and Connectionism aspects. From Deep Blue [36], the chess machine that beat the world champion with an efficient tree search, to a deep reinforcement learning model that learns to play different video games using only the raw pixels as input [37].

In Sections 2.5.1 and 2.5.2, we will present two examples of encoding functions. The first is Autoencoders which can be used to encode patterns from any domain. The second one is more *nativist* since

it is restricted to the domain of images.

2.5.1 Autoencoders

An Autoencoder (AE) is a lossy data compression algorithm that can be used to generate SDRs. It is implemented as a feed-forward ANN with two modules: The **encoder** which transforms the original inputs x into compressed representations h; and the **decoder** which takes the compressed representations h, and creates reconstructions r (see Figure 2.7 for an illustration). The compression and decompression functions are learned automatically from unlabeled data rather than hand-engineered by a human. Additionally, the AE also has a loss function $L(x, r)$ that measures the reconstruction error.

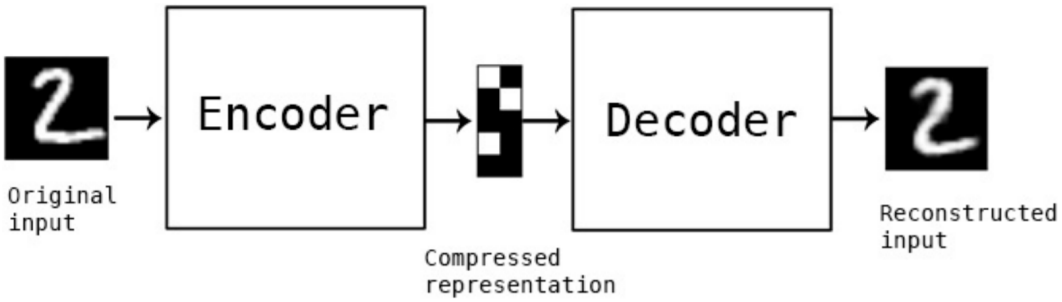


Figure 2.7: Overview of an Autoencoder. The Autoencoder is composeb by two modules: the **encoder** module transforms the original pattern into a compressed representation. This representation is a compressed version of the original input. The **decoder** module takes the compressed representation and tries to reconstruct the original pattern. Adapted from [38].

The AE is a traditional feed-forward ANN that trains with back-propagation, but it employs an *unsupervised learning* algorithm. While most ANNs require labeled data to train (supervised learning), the AE eliminates the need for labels by using the training patterns as both the inputs and the targets in the training process. The fact that the input and the targets (which are the desired outputs) are equal, means that the neural network's goal is to learn to copy what it receives in the input layer onto the output layer [39].

For an AE to be a useful compression tool, we must ensure that:

- The representations h are compressed, i.e., they have considerably less information capacity when compared with the original input.
- The decoder module must be able to transform the compressed representations back into close reconstructions of the original input.

The first aspect ensures that the encoder will learn a compression function, and it is addressed when designing the network. The hidden layer in between the encoder and decoder must act as an information bottleneck. There are many ways to implement the information bottleneck. Depending on

the application, different properties will be enforced in the representations. For example, *undercomplete* AEs have a very small hidden layer, whereas regularized AEs (Sparse, Denoising, or Contractive) add penalty terms to the loss function [40].

The second aspect ensures that the compression is not excessively lossy. If the reconstruction accuracy remains low after training, it means that the information bottleneck is too restrictive. Contrarily, if the AE has a good reconstruction accuracy after training, it means that the compression rate is adequate, and the AE is learning compressed representations that capture the relevant features of the patterns.

It is important to underline how the accuracy score is interpreted in AEs, as it is quite different from typical ANNs. In a traditional ANN task, such as the classification of MNIST digits [6], a solution is as good as its accuracy. The goal is to create a network that can classify novel examples with 100% accuracy. Researchers try out many different architectures and train them endlessly with backpropagation to obtain the best possible score. With AEs, perfect accuracy is not the goal. The network accuracy of an AE tells us how well the decoder module can reconstruct the original pattern. Since the AE is a lossy compression algorithm, the decoder cannot generate perfect reconstruction, even with extensive training. The best possible accuracy of an AE will be bounded by the compression factor imposed by the architecture. Therefore, the accuracy of an AE should be seen as an indicator of the "lossiness" of the compression and not the quality of the solution.

Let us look at a simple example that illustrates the role of accuracy in AEs: Suppose we want to compress images of 10×10 pixels and we have three possible architectures:

1. An AE that has a hidden layer with 10 units. After training it manages to create reconstructions with 60% accuracy.

2. An AE that has a hidden layer with 100 units (as many units as pixels). After training it manages to create reconstructions with 100% accuracy.

3. An AE that has a hidden layer with 20 units. After training it manages to create reconstructions with 95% accuracy.

The first AE is the one with the highest compression rate, but this comes at the cost of a big loss in the reconstruction accuracy. The compression is too lossy.

The second AE is useless: by having as many units as pixels, the network can learn how to simply copy its input with the identity function. No compression is performed.

The third AE is the best option. It manages to reduce the size of the images by a factor of five, and then reconstruct the original pictures with 95% accuracy.

With this example, we can see how there is a trade-off between compression rate and reconstruction accuracy. We can also see how the goal of an AE is not to have 100% accuracy. When a good balance between "lossiness" and compression rate is achieved, the AE has learned how to extract the relevant

features of the patterns and compress them into informative representations.

The AE was originally proposed as a data compression algorithm, but as François Chollet, author of the Keras library, puts it, AEs are not the best tool for data compression [38]. In picture compression, for instance, JPEG will yield much better results and will work for any domain, while the AE will only be able to compress images similar to those it trained on. But the AE has had great success in a variety of applications other than data compression. An important breakthrough for AEs was the technique developed by Hinton, where AEs were stacked on top of each other to learn hierarchical representations of images [41]. The two most successful applications of classical AEs are data denoising [42] and dimensionality reduction for data visualization [43]. More complex AE architectures have emerged recently. Variational AutoEncoders (VAEs) [44] are used to learn the probability distribution of the input data, and can be used as *generative models*. The domain-dependent nature of AEs can be leveraged for anomaly detection tasks [45, 46]. If the AE is of the overcomplete type (with large, sparsely activated hidden representation) it can be used to generate SDRs. We will focus on the latter application in the next section.

2.5.1.A Autoencoders for the generation of SDRs

"An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function." [39].

As we have seen previously, SDRs owe their informative nature to the strict properties of sparsity and distribution. As it turns out, these properties of SDRs can also be used as the information bottleneck of AEs. If we do so, the AE can be used to transform real-world data into SDRs.

So how exactly do we impose the properties of SDRs onto the hidden representation of an AE? We must ensure that:

1. **Size**: Real-world data typically has fewer dimensions in its raw format when compared with its SDR counterpart ($\|x\| \ll \|h\|$). For this reason, the AEs that generate SDRs are usually of the *overcomplete* type, where the hidden layer is larger than the input/output layer.

2. **Sparsity**: To prevent the hidden layer from learning the identity function, we force the AE to learn representations that sparsely activate its hidden units (an active unit has an output close to 1). To do so, we add a penalty term to the loss function that penalizes representations that have a sparsity away from the desired value.

3. **Distribution**: The AE should also learn distributed representations, i.e., it should use all of its hidden units evenly across the entire dataset. Ideally, the activation of units in the hidden layer follows a uniform probability distribution.

Notice how the second and third properties can be combined into a single constrain: If we force all the hidden units of an AE to have approximately the same (low) activity level ρ (let us say each hidden unit should only activate in $\rho = 2\%$ of all the patterns across a training set), we are ensuring both the sparsity and distribution property at the same time.

There are two main ways to impose sparsity into the hidden layer of an AE. Both involve adding a penalty term to the loss function that depends on the activity of the hidden units across a batch of training examples.

The first, most straightforward way, is to add the **L1 norm** of the hidden activity to the loss function L:

$$L_{new} = L(x, \hat{x}) + \lambda \sum_{i=1}^{m} \left| h_j^{(i)} \right| \tag{2.18}$$

where λ is a tuning parameter, $h_j^{(i)}$ corresponds to the activation of the hidden unit j for pattern i of the batch, and m is the number of patterns in the batch. While this approach can ensure sparsity, there is no way to guarantee distribution.

The second, more complex way, uses the notion of probability distributions. And it can be used to enforce both sparsity and distribution.

Let us define $\hat{\rho}_j$ as the average activation of hidden unit j (averaged over the training batch) [47]:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left(h_j^{(i)} \right), \tag{2.19}$$

where m is the number of training examples in the batch, and $h_j^{(i)}$ is the activation of hidden unit j given input $x^{(i)}$.

If we want to generate SDRs with a sparsity level of ρ, then we should ensure that $\hat{\rho}_j = s \mid j = 1, \ldots, H$, where H is the number of hidden units.

To do so, we add a penalty term $\Omega(\rho)$ to the loss function L of the AE:

$$L_{new} = L(x, r) + \Omega(\rho) \tag{2.20a}$$

$$\Omega(\rho) = \mathrm{KL}\left(\rho \| \hat{\rho}_j\right) = \sum_{j=1}^{H} \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j}, \tag{2.20b}$$

where H is the number of hidden units, and j is used to iterate over all hidden units. Here, $\mathrm{KL}\left(\rho \| \hat{\rho}_j\right)$ is the Kullback-Leibler (KL) divergence [48]. The KL divergence measures the distance between two density distributions. Therefore, by using $\mathrm{KL}\left(\rho \| \hat{\rho}_j\right)$ as a penalty term, we are rewarding the AE for generating representations that activate their units following a normal distribution ρ (see Figure 2.8).

After training, we can take the encoder module of a trained AE that implements these restrictions

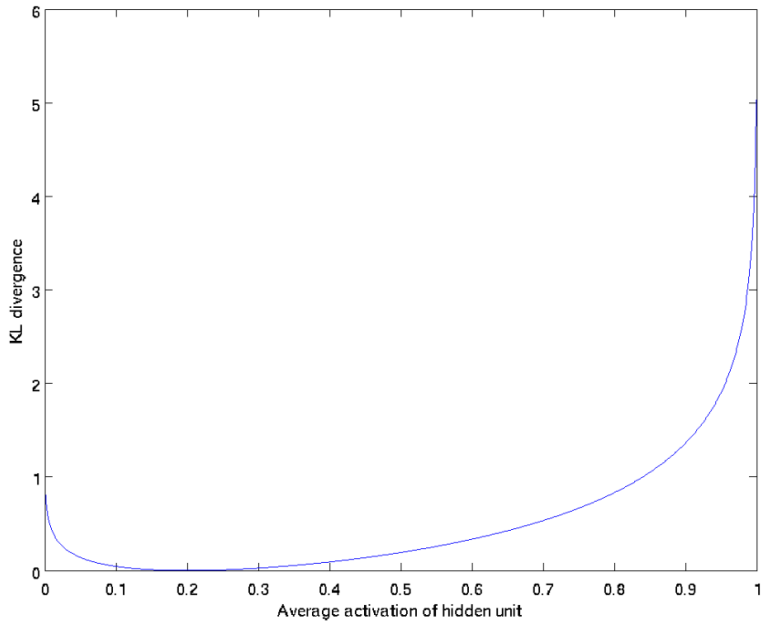


Figure 2.8: Example of the KL divergence. Here we want to enforce a sparsity level of $\rho = 0.2$. The KL divergence will measure the distance between the distribution $\hat{\rho}_j$ and the desired uniform distribution $\rho = 0.2$. Notice the global minimum when $\hat{\rho}_j = \rho = 0.2$. The KL divergence will monotonically increase as we move away from ρ, and it explodes to ∞ when $\hat{\rho}_j$ goes to either 0 or 1. Adapted from [47].

and use it to transform the training data into SDRs. Also, the encoder should be able to generalize to novel patterns of the same domain.

2.5.2 What-Where Encoding

"Vision allows humans to perceive and understand the world surrounding them, while computer vision aims to duplicate the effect of human vision by electronically perceiving and understanding an image." [49]. The field of Computer Vision has been highly connected to biology [50] since the 1980s. The field's Holy Grail is to build a model that performs as well as the human brain. For this reason, many *Nativism* models, that implement our biological knowledge of vision, arise in image processing tasks.

In [1], Sa-Couto & Witchert propose an *encoding function* that maps visual patterns into informative binary vectors. The *Nativist* model is biologically inspired, using similar ideas to those of the *Neocognitron* [51]. The *Neocognitron* is a direct implementation of Hubel & Weisel's Nobel-Prize-winning theory of the mammalian visual cortex [52], and it is the precursor of the widely popular Convolutional Neural Networks (CNNs) [53].

The goal of the authors was to implement a sparse encoding function for visual patterns that could be used to pre-process data in classification tasks. The resulting representations were SDRs, so the authors experimented on the Willshaw Network (WN): This Associative memory will have a better storage capacity if the representations it receives are informative. Knowing this, we can evaluate the quality of

the encoding prescription as a whole by monitoring the retrieval score of a WN that stores the representations.

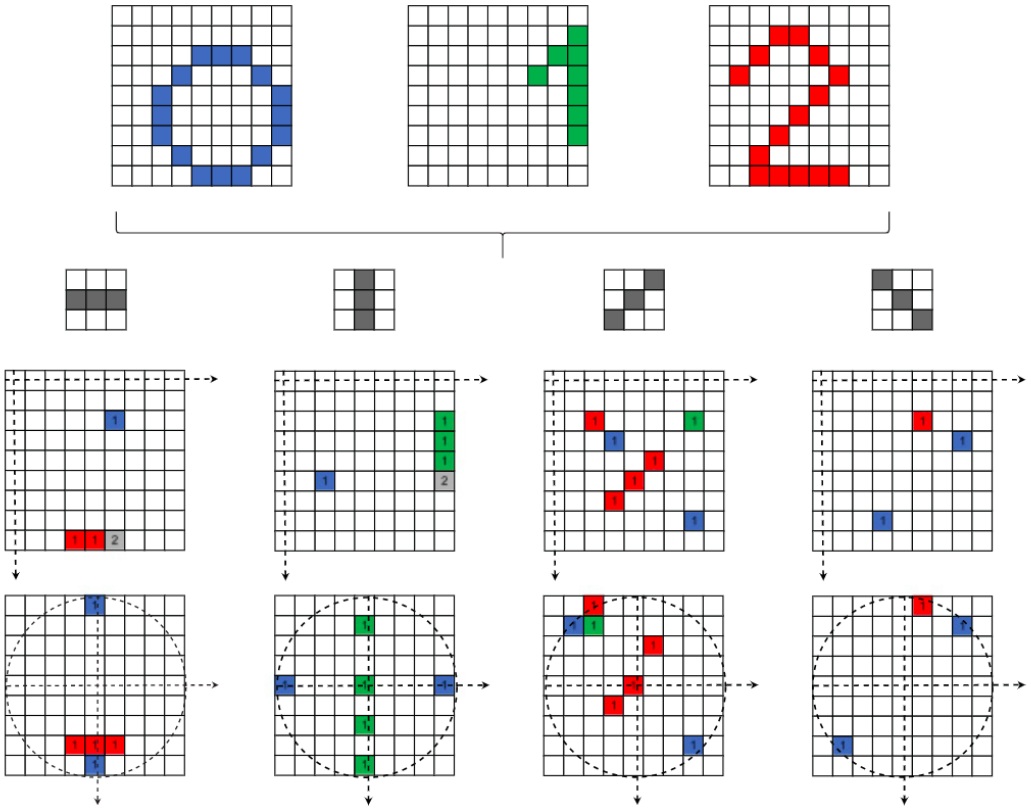


Figure 2.9: Overview of the strategy that transforms visual patterns of digits into sparse and distributed codes. The first step is local feature extraction. Each feature is depicted as a 3×3 window with an oriented line. Each image is parsed with a sliding window, and each occurrence of each feature is signaled at the middle layer. The positions where these signalings occur refer to a fixed coordinate system. The second step maps these positions to an object-dependent, radius one polar coordinate system. Note that the final representation (bottom row) corresponds to three distinct representations (one for each of the digits). Adapted from [1].

Using an AM to store such a large collection of natural data is traditionally a very hard task for these models, and as expected, retrieval was imperfect. For this reason, the authors proposed a novel error measure that utilized the labels of the patterns. The results show that the codes produced by this strategy are effectively compressing the information in the patterns, so much so that a very simple 1 nearest-neighbor classifier can perform classification when trained on the outputs of a WN that stores the compressions in auto-association.

An overview of the encoding function can be seen in Figure 2.9. The strategy has two main steps:

The retinotopic Step: A convolution-like layer [54] of units that act as receptive fields are locally connected to the input layer. These receptive fields detect and extract the most relevant visual features of

the patterns. These features are determined beforehand with the unsupervised K-means algorithm [55, 56]. This layer performs information compression by establishing a many-to-one relationship between groups of pixels and receptive units. In this way, it transforms the dense representation presented in the input layer into a sparse representation.

The Object-Dependent Step: The fixed coordinate system of the representations is turned into an object-dependent, radius one polar coordinate system. This object-oriented transformation is known to occur in mammalian brains [57]. For this step, only the radius and center of each object need to be measured, no learning is required. This operation provides invariance to size and position, and in turn, also makes the resulting representations well-distributed since the features detected in the previous step will be mapped quasi-uniformly into all the dimensions of the object space.

The encoding process can be controlled with the following set of parameters: K specifies the number of visual features that will be extracted with the K-means algorithm, the Field size (F_s) specifies the "radius" of the visual features (which are square windows of side $2F_s + 1$), Q represents the size of the new coordinate system plane and, finally, the threshold T_w controls the level of similarity needed to signal the presence of a feature in an image. The former can be used to tune the sparsity of the codes.

The encoder receives the raw MNIST dataset with dimensions (N, 28, 28), where N is the number of patterns. The resulting sparse codes are binary vectors with dimensions (N, $Q \times Q$, K), i.e., each one of the N patterns is represented as a set of K features maps of size $Q \times Q$.

2.5.3 The biology of Encoding Functions

One might ask: "If Associative memories are biologically plausible, why is their performance on raw data poor? The brain handles raw data just fine." This is a valid comment, and in this section, I will give a hint on the motivation behind encoding prescriptions.

Representing human knowledge on a computer has proven to be a difficult task. The problem stems from the fact that our knowledge of the world is not well-organized. Every rule has exceptions and every fact links to numerous others. This level of complexity is not well-suited for traditional computer data structures. In contrast, our brains seem to deal with this complicated knowledge with ease. Our brains do not use dense representations like those used in computers. Instead, they represent knowledge with the sparse activation of its billions of neurons. The brain uses SDRs to encode knowledge [4]. This might sound contradicting. After all, if real-world data, such as images, is dense, how can the brain use SDRs? Indeed, sensory data is not naturally found in the form of SDRs. But we know that our brain has lower functional regions that process the inputs from the senses and generate sparse and distributed activation of neurons in the neocortex [5]. For instance, the visual area of the brain is divided into several hierarchical regions: V1, V2, V4, and IT. The V1 area of the brain is responsible for detecting

low-level visual features such as edges, and basic color [58]. The detection of such features results in the activation of a collection of neurons that forms a signal. This signal is passed onto the V2 area which will apply a similar process. After passing through all the regions, the final result will be an SDR that represents what we are sensing [59]. Another example is auditory perception. The sounds that we hear are captured in the cochlea. The cochlea is an organ that has several receptors for different frequencies. A particular sound will result in the sparse activation of the receptors which will result in an SDR.

All-in-all there is a biological analogy between SDR encoders used for Associative memories and biological organs that transform sensory data into sparse neuron activity in biological memories.

2.6 Pattern Generation

While sparse codes are good for the performance of computational models, they are also abstract. To approach interpretability, it is interesting to explore mechanisms that generate concrete data from sparse codes. In this section, we present two tools that can generate patterns.

Despite some minor nomenclature disagreements, machine learning models can be divided into **Discriminative** and **Generative** [60, 61]. Consider a classification task with a set of examples X, and their corresponding labels Y. A *discriminative* classifier will either learn a direct mapping between X and Y, or learn the *posterior* probability of Y given X ($P(Y \mid X)$). On the other hand, a *generative* model has an intermediate step that learns the *likelihood* of Y given a fixed X ($L(Y \mid X) = P(X \mid Y)$), and then learns a model of the underlying *joint* probability of the data ($P(X, Y)$).

The *generative* model's task is a much more general and difficult one. By learning the *joint* distribution, it is also possible to compute the *posterior* with Bayes Rule ($P(Y \mid X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$). Contrarily, a *discriminative* model that has leaned the posterior, isn't able to infer the joint. In the context of pure classification tasks, using generative models is traditionally seen as overkill [62], but the added complexity of the *generative* tasks allows the model to go beyond classification: a trained generative model can generate new artificial examples drawn from the distribution of the data, hence the name *generative*.

2.6.1 Variational Autoencoders

As we saw in Section 2.5.1, the traditional AE compresses its inputs x into feature vectors h. These feature vectors are simply an array of real values that represent the presence of latent features in the inputs.

Variational Autoencoders (VAEs) extend the basic AE architecture by representing the probability distribution functions of the latent features, as opposed to simply representing their value. For instance, if we assume that the features follow a normal distribution, the hidden layer of a VAE would represent the mean and standard deviation for each dimension of the hidden space. The goal of a VAE is to

learn the posterior distribution $P(Z \mid X)$ over a set of unobserved variables Z, given some data X. Computing the exact value for the posterior is often intractable since it requires the computation of $P(X) = \int P(X \mid Z)P(Z)\,dz$ over all the dimensions of z. VAEs circumvent this problem with *variational inference*, i.e. the model learns an approximated tractable distribution of the posterior: $Q(\mathbf{Z}) \approx P(\mathbf{Z} \mid \mathbf{X})$. To ensure that Q is a good approximation of the real distribution, we can minimize the KL divergence (Equation (2.20b)) between them.

The VAE is implemented as two independently parameterized networks that work together: the encoder/recognition model and the decoder/generative model [39]. These two models train together with back-propagation, where the loss function simultaneously maximizes the reconstruction likelihood and minimizes the KL divergence between the approximation and the unknown distribution:

$$loss = E_{Q(Z|X)} \log P(X \mid Z) - KL(Q(Z \mid X) \| p(Z)), \tag{2.21}$$

where $E_{Q(Z|X)}$ is the expected value over the distribution Q.

Once the VAE is trained, its weights can be fixed, and we can generate new examples by sampling from the learned distribution Q (see Figure 2.10).

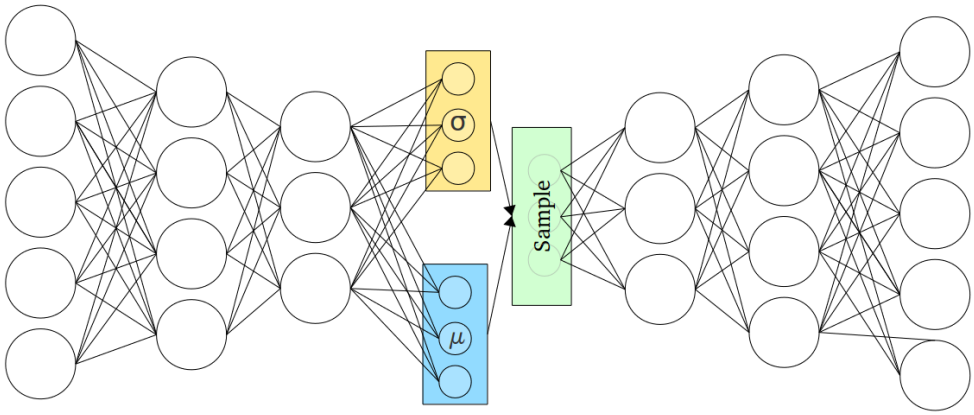


Figure 2.10: Architecture of a VAE. The network on the left is the encoder/recognition module which models $Q(Z \mid X)$; In the middle, the hidden compression of this AE corresponds to the normal distribution of three latent features, i.e. three means (in blue), and three standard deviations (in yellow); The approximated distribution is randomly sampled (in green); The network on the right is the decoder, which models $P(X \mid Z)$. Adapted from [63].

2.6.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a family of generative models that combine ideas from Game Theory and Machine Learning. A GAN consists of two independent neural networks that are

trained simultaneously with back-propagation: a generative model G that learns the distribution of the original data, and a discriminative model D that learns to distinguish between fake patterns generated by G and real patterns from the original dataset [64].

The training process of a GAN is an adversarial min-max game between the two models. The goal of G is to fool D into thinking that its patterns are real and the goal of D is to distinguish between real and fake patterns. As the two models compete, they gradually improve at their tasks. The game will eventually reach a *Nash equilibrium*, where G has implicitly captured the correct probability distribution of the data, and D can't do better than random guessing since both the fake and real patterns appear to come from the same distribution.

After training, the two models can be decoupled and used independently. The Discriminator's most common applications are classification in semi-supervised learning scenarios [65], and domain adaptation [66]. The Generator module is used to generate new patterns. A *seed* provided in the input layer will determine the resulting pattern. The basic use-case of the Generator is to produce patterns similar to those in the training set using *random noise* as the *seed*. More advanced use cases involve operations between *seeds*. Since the generation process is deterministic (a seed will always output the same pattern), a GAN can utilize vector arithmetic and interpolation to obtain complex patterns. For instance, it is possible to interpolate between two classes as seen in Figure 2.11.

Figure 2.11: Pattern interpolation with a GAN. Here a GAN was trained on the MNIST dataset of handwritten digits [6]. If we take the seed that generates the image of a one, and the seed that generates the image of a five, we can interpolate between the two seeds to obtain a transformation in the image space. Adapted from [64].

3

Willshaw Network Retrieval Analysis

Contents

In this section, we extend the work done in [1] to get a deeper understanding of how the retrieval process of the WW codes works in the WN.

3.1 WW codes paramenters

While the WW encoder has many parameters (described in Section 2.5.2), we focus our analysis on the two most important ones:

- **Field Size** (F_s) which denotes the " radius" of the visual features (see Fig. 3.1): The visual features are square windows of size ($2F_s + 1$, $2F_s + 1$)
- **Retinotopic Threshold** (T_w) which controls the required level of similarity to signal the presence of a visual feature in a code: While sliding a window across an image, a feature is detected when the dot-product between the window and a featured learned with k-means is greater than T_w.

(a) Field size of 2 pixels (5*5 features)

(b) Field size of 4 pixels (9*9 features)

Figure 3.1: Visual Features of the MNIST dataset for different values of F_s. (a):A small value for F_s will extract simple features such lines with different orientations. **(b):** A larger value of Fs will result in more complex features where corners, circles and parts of digits begin to appear. Note that the visual features are a square window of size $2F_s + 1$.

3.2 WW codes properties

The two most important properties of the binary set that we store in the WN are its sparsity and distribution. In this section, we will discuss how to measure these properties in the WW codes, and how the parameters of the encoding process influence said properties.

3.2.1 Sparsity

We define the sparsity of a set of binary vectors, as the average sparsity (Eq. (2.12)) of its elements. Formally, in a set containing M binary vectors of size n, its sparsity s is given by:

$$s = \frac{\sum_{m=1}^{M} \sum_{i=1}^{n} x_i^m}{M * n} \tag{3.1}$$

One important parameter in the generation of the WW codes is the threshold (T_w) that is used in the retinotopic step (Section 2.5.2). This threshold will determine whether or not a visual feature is extracted from the image and transformed into an active bit of the code: a low value for T_w will result in many features being detected in the image, thus producing a less sparse code; on the other hand, a high value for T_w will result in a more sparse code (see Fig. 3.4).

(a) Retinotopic Threshold

(b) Field Size

Figure 3.2: Effect of T_w and F_s on the WW codes sparsity. Here we are plotting the measured sparsity of the WW codes for 12 different parameter configurations. The box plots are showing the minimum, Q1, Q3, and maximum values. **(a):** We can clearly see that the higher the value of T_w, the more sparse the coded set becomes. **(b):** The effect of Fs on the sparsity of the codes is not as noticeable as in the case of T_w, nevertheless there is still a monotonous decrease in the sparsity of the coded set which results from dependence between F_s and T_w on the encoding process.

3.2.2 Distribution

A distributed binary set is one where the positions of the binary vectors activate with similar probability. Recall the definition P (Eq. (2.13)) which denotes the probability of activation for each position of the vectors in a set. Additionally, consider U to be the uniform distribution.

In our experimental analysis, we tested out three distinct ways to measure the distribution of a binary set:

- The Shannon entropy (Eq. (2.14)) of P: $H(P)$.
- The KL-Divergence (Eq. (2.20b)) between the normalized P and U: KL-div(P, Q).
- The average norm (across the n positions) between the normalized P and U: $\frac{\sum_{i=1}^{n} |P_i - U_i|}{n}$

The desired properties for a distribution measurement are: (a) invariance to the size of the vectors; (b) invariance to the size of the set; and (c) invariance to sparsity. A set of simple experiments on

synthetic and real data was done (see Figs. A.2 to A.5 in Appendix A). The best best measurement according to the aforementioned desired properties was the **Shannon entropy**.

In the encoding process, the distribution property is mostly enforced on the object-dependent step (Section 2.5.2) when each image is mapped into a polar coordinate system. In this thesis, we used the best performing parameters from [67] to ensure the distribution of the codes. The produced codes were 8820 bit long arrays, and the average Shannon entropy (Eq. (2.14)) of the coded set was 11.68 (see Fig. 3.5). Note that for a distribution d of size 8820, the Shannon entropy H will be a value in the range $[0.0, 13.1]$ (Eq. (2.14)) where 13.1 would be the ideal value of the distribution.

3.3 Performance Measurements

As stated in Section 2.1.3, we measure the performance of an auto-associative AM by measuring its storage capacity, and the quality of the retrieved vectors. Next, we define three performance measures that were used to study the behavior of the WN.

3.3.1 Perfect Retrieval Error

We can evaluate the WN by measuring the percentage of patterns that the memory can perfectly retrieve. Formally, consider a WN that stores a set of patterns X. When the set X is presented as retrieval cues, the memory will output a set of patterns Y. The Perfect Retrieval Error (PRE) when the memory stores M patterns is given by:

$$\text{PRE} = \frac{\text{number of times where } X_i \equiv Y_i}{M} \in [0, 1], \tag{3.2}$$

where "$X_i \equiv Y_i$" means that the vectors X and Y are bit-wise identical.

This error measurement is a very pessimistic one, as the WN only returns an identical copy of the retrieval cue when the memory holds very little information.

3.3.2 Hamming Distance

"The Hamming Distance (HD) between two sequences of symbols is the number of positions in which the symbols are different. For example, a (010) sequence differs from the sequence (111) in two positions and would, therefore, have Hamming distance of 2." Adapted from [68].

Formally, consider a WN that stores a set of patterns X. When X is presented as retrieval cues, the memory will output a set of patterns Y. The HD when the memory stores M patterns is given by:

$$HD_{set} = \frac{\sum_{m=1}^{M} HD(x^m, y^m)}{M}, \tag{3.3}$$

which represent the average number of differing bits between the elements of X and Y.

When retrieving binary sequences, the bits can differ because: (a) $x_i = 0$, $y_i = 1$ meaning that the memory added an extra bit to the cue; and (b) $x_i = 1$, $y_i = 0$ meaning that the memory lost a bit from the cue. In our analysis, we will differentiate these two scenarios as it is useful to understand the source of the retrieval error. We can do so by counting the number of 1s and -1s in the difference between the cue vector x and the retrieved vector y. For example, if the we have: x = (0,1,1,0,0,0), y = (0,1,0,0,1,1) then the difference $x - y$ is: $(0, 0, -1, 0, 1, 1)$ which has one -1 and two 1s. Meaning that there is 1 lost bit and 2 extra bits in y, and the total HD is 3 (the sum of the two types of error).

3.3.3 One-Nearest-Neighbour classifier

In [1] the authors propose an error measure for the quality of retrieved patterns that makes use of the label of the patterns that the WN stores. The idea is to use the memory's retrieval responses Y to train a dot-product-based One-Nearest-Neighbour (1NN) classifier. Then the retrieval cues X are used to test the classifier. The classification accuracy of this simple classifier indicates whether or not the memory can keep the information that is relevant to identify the patterns' class. The error-measuring process is depicted in Fig. 3.3.

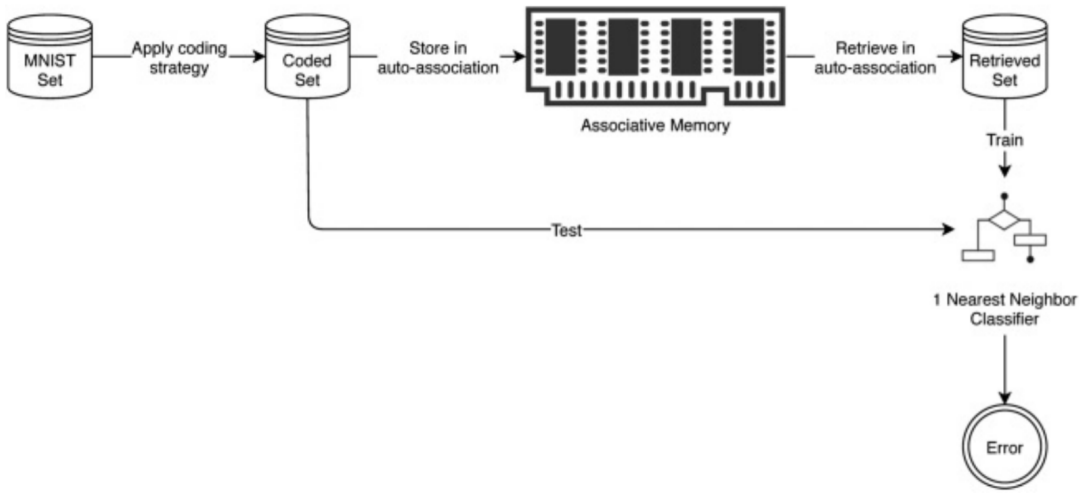


Figure 3.3: One-Nearest-Neighbour classifier error. A set of digits is mapped to a set of binary codes X, which is then stored in a Willshaw memory in auto-association. Afterward, the coded set is used to retrieve the memory's contents and build a retrieved set Y, which is used to train a dot-product-based 1 nearest-neighbor classifier. With that settled, we use X to test the classifier. The resulting classification error is reported. In this way, we can measure if the retrieved patterns were corrupted by the memory or if they remain good representatives of their class despite some minor noise introductions. Adapted from [1].

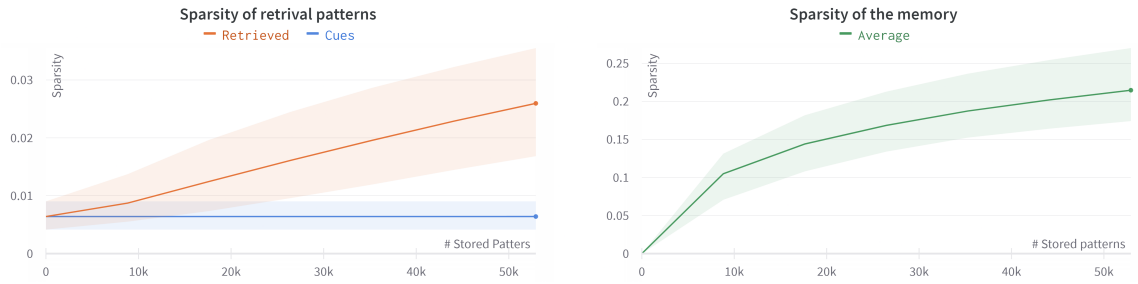
3.4 Analysis of the retrieval process

Now, let us analyze how the properties and performance measures described in the previous sections behave as we incrementally store patterns in the WN in auto-association.

3.4.1 Sparsity

In the retrieval process, we can measure the sparsity of: (1) the binary retrieval cues X that go in to the memory (Fig. 3.4(a)); (2) the binary weight matrix that represent the memory itself (Fig. 3.4(b)); and (3) the binary retrieved codes Y that come out of the memory (Fig. 3.4(a)).

The sparsity of the cues depends only on the encoding process, as described in Section 3.2.1, and therefore is invariant throughout the storage of the dataset (blue line in Fig. 3.4(a)). The WN's weigh matrix will initially be empty, and therefore have a sparsity of 0. As the memory stores information, updates in the weight matrix will occur, and thus the sparsity of the memory will increase. As more information is stored, updates in the weight matrix will become rarer, and thus the rate at which the sparsity of the WN increases starts to flatten (Fig. 3.4(b)). The retrieved vector Y will initially have a very similar sparsity to the coded set, but as more information is stored in the memory, retrieval becomes imperfect and the memory introduces noise which can be seen by the increase in the sparsity of the retrieved patterns (orange line in Fig. 3.4(a)).



(a) Cues and Retrieved vectors

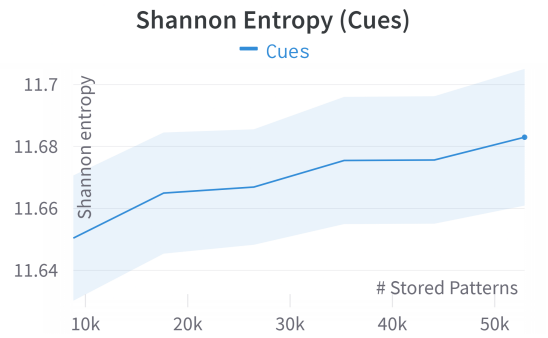
(b) WN sparsity

Figure 3.4: Sparsity of retrieval as a WN is loaded with WW codes. Here we are plotting the average values, and standard error for 12 in depend runs. **(a):** Retrieval cues (blue) and retrieved vectors (orange). **(b):** Willshaw Network weight matrix.

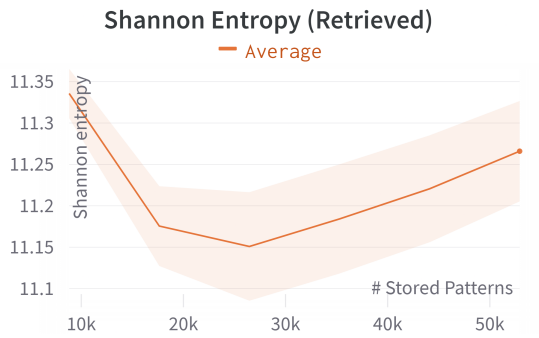
3.4.2 Distribution

Here we analyze the distribution of the WW codes before and after being retrieved by the WN.

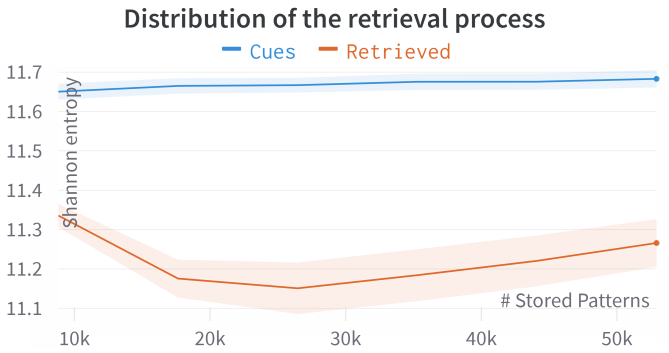
The Shannon entropy or "uncertainty" of a distribution that is computed from an experiment will tend to increase as the number of simulations grows in size. For instance, a fair coin with two faces has an



(a) Cues



(b) Retrieved



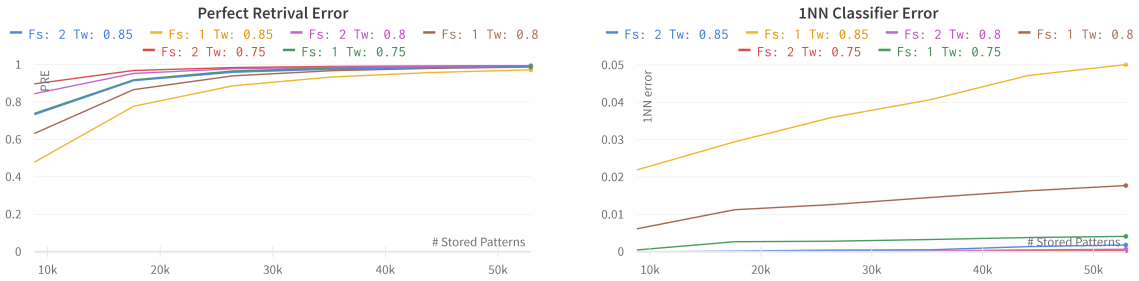
(c) Cues and Retrieved

Figure 3.5: Distribution of retrieval as a WN is loaded with WW codes. Here we are plotting the average values, and standard error for 12 in depend runs. **(a):** Retrieval cues. **(b)**: Retrieved Vectors. **(c)**: The two previous plots combined into one (to facilitate visual inspection).

entropy of 1 which is maximal since the two possible outcomes are equally probable. If we compute the entropy through simulation, by repeating the random experiment and recording the probability of each event, we would need a sufficiently large sample to converge into the actual value. The same is true for the WW codes: as we encode more and more patterns the entropy of the coded set (Eq. (2.14)) increases, meaning that the set becomes increasingly more distributed (Fig. 3.5(a)). The retrieval process adds noise to the cues, and as a result, the entropy of the retrieved set is lower than that of the coded set (Fig. 3.5(c)). As more patterns are stored in the memory, the entropy decreases. Eventually, when the retrieval set is large enough, the entropy flattens and begins to increase (Fig. 3.5(b)) which is a consequence of the set's increase in size. The behavior of the entropy throughout the storage process is an exciting result, as it shows that the noise introduced by the memory is non-random.

3.4.3 Retrieval Quality

Now, let us analyze the quality of the retrieved vectors. The PRE (Fig. 3.6(a)) and the 1NN error (Fig. 3.6(b)) measure the quality of retrieval in two completely different ways: the former measures the ability of the memory to output a perfect answer with no wrong bits; the latter is not concerned with perfect answers and instead measures the memory's ability to keep the information that is relevant to identify the class. In other words, the PRE measures the memory's ability to memorize or "overfit" the patterns that it stores, while the 1NN error measures the memory's generalization capability.



(a) Perfect Retrieval error

(b) 1NN classifier error

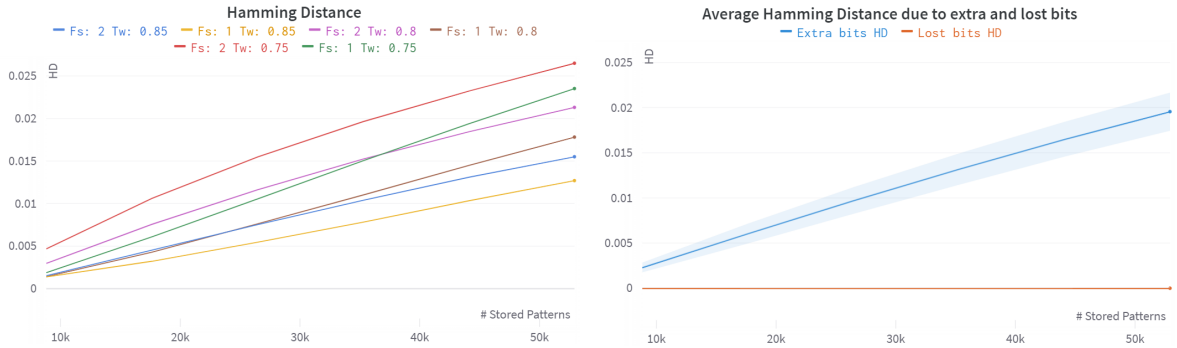
Figure 3.6: 1NN classification error and Perfect Retrieval Error. Here we plot two error measurements for six independent runs with different configurations for the T_w and F_s parameters. In the x-axis we represent how many patterns are stored in the memory so far. **(a):** The PRE is a very strict error measure: all the runs quickly tend towards an error rate of 1.0. The best runs according to this measure are the ones where the codes contain less information (higher T_w and smaller F_s) because they allow the memory to "overfit" the simpler patterns.**(b):** The 1NN classifier error. Here we followed the methodology of [1] but with a fixed value for T_w. Runs with F_s = 1 performed poorly due to their features containing less information. For the remaining runs, as the features are more informative, fewer bits are needed, and thus, even sparser versions achieve good results. Notice the trend where runs which perform well on one error measure tend to perform bad on the other, and vice-versa.

The PRE is low when the codes are very sparse and contain less information, in these conditions the memory artificially boosts its storage capacity by memorizing less information. The 1NN error is lower when the features are more informative because, independently of the sparsity of the code, the memory is able to generalize according to the category.

Finally, let us look into the HD of the retrieval process to better understand the source of the retrieval noise. The HD (Eq. (3.3)) measures the average number of differing bits between the retrieval cues and retrieved vectors (Fig. 3.7(a)). Furthermore, the bits can differ for two reasons (see Section 3.3.2) and thus we can evaluate the memory on its ability to keep the bits of a cue, and its ability to avoid adding extra bits (Fig. 3.7(b)).

If we compare Figs. 3.6(a) and 3.7(a) we can notice that the PRE and the HD are correlated (runs that do well on one measure also do well on the other). This is expected as both these errors measure

the memory's ability to "memorize" patterns. Notice how the HD behavior is very similar to that of the sparsity of the retrieved vectors (compare Figs. 3.4(a) and 3.7(a)). The reason for this behavior becomes clear when we analyze Fig. 3.7(b): the memory never deletes a bit from the retrieval cue, all the noise of the retrieval process is due to added bits.



(a) Average HD

(b) HD due to lost and extra bits

Figure 3.7: HD in the retrieval process. Here we plot the HD error measurements for six independent runs with different configurations for the T_w and F_s parameters. In the x-axis we represent how many patterns are stored in the memory so far. **(a):** The average HD (defined in Eq. (3.3)) increases monotonically with as more pattern are stored in memory. All the different runs seem to increase their HD at a similar rate. **(b):** Here we measure the HD due to extra (blue), and lost (orange) bits in the retrieved vector. For visual clarity, the values from the six runs are grouped: the mean and standard error are plotted. Here we can see that the HD due to lost bits is always 0.0 for all the runs.

From this analysis of the retrieval process, we can conclude that:

- The WN can efficiently store the WW codes (Fig. 3.4(b)).
- As more information is stored, noise is added onto the retrieved vectors (Fig. 3.4(a)).
- The noise that the memory adds appears to be non-random (Fig. 3.5).
- The memory is able to generalize even if retrieval is not perfect (Fig. 3.6).
- The memory never loses bits from the cues, all the noise is due to extra information (Fig. 3.7).

All these results suggest that the noise that the memory adds to the patterns on retrieval might not be as harmful as initially thought. We hypothesized that the memory adds "good noise" which generalizes using the information from the whole set that is stored in memory. In the next chapter, we will focus on building decoders that transform the SDRs back into images. That way we can get a visual representation of the noise and test our hypothesis.

4

Reconstruction of memory contents

Contents

In this section, we present and study a decoder that transforms the WW codes back into 28-by-28-pixel images (the same shape as the original MNIST domain). With a decoder built, we can visually study the effect that the WN has on the patterns that it stores. Furthermore, having a decoder module is an essential step to establish a baseline for the generation of patterns, which is one of the main goals of this work.

Initially, we tried to solve the decoding problem with back-propagation-based Neural Networks, namely with the Multilayer perceptron (MLP), and Convolutional Neural Network (CNN). After some tuning of the architecture and parameters of these models, results remained sub-optimal, which lead us to explore alternative solutions. The decoding results of the back-prop based models can be seen in Figs. A.6 and A.7 of Appendix A. In this section, we focus our analysis on the alternative solution which is a simpler decoder that achieved great results without the need for learning.

4.1 The What-Where decoder

The What-Where (WW) Decoder implements the inverse function of the WW encoder presented in section Section 2.5.2. The decoder transforms the sparse code set of shape $(N, Q \times Q, K)$ back into the original MNIST shape $(N, 28, 28)$, where N is the number of patterns in the dataset, Q is the size of the polar coordinate system, K is the number of visual features learned with the K-means algorithm, and 28 is the original size of the MNIST images. The decoding strategy has three steps:

The Object dependent Inverse step: The object-dependent, radius one polar, $Q \times Q$ coordinate system is converted back into the Cartesian, 28×28 coordinate system. To do so, the radius and center of the object (which were measured in the encoding process) are used. If the radius and center of the patterns were not saved, the default values of $c = (0, 0)$ and $r = 1$ are used. This step changes the dimensions of the codes from $(N, Q \times Q, K)$ into $(N, 28 \times 28, K)$, where K is the number of visual features.

The Retinotopic Inverse Step: The outputs of the last step are a set of K binary feature maps that signal the presence of visual features. In this step, we transform these bits into the actual visual features by performing a "transposed convolution" where the convolution kernels are the ones learned with the K-means algorithm before the encoding process. Since the WW encoder uses a stride of 1, same-padding, and square kernels with odd sizes, this step is straightforward: an active bit on plane k and position (i, j) will result in the visual feature k being drawn with its center on position (i, j) of the MNIST images space. The output of this layer has the same dimensions as the original MNIST images.

Normalization Step: The former step can produce images where the value of a particular pixel is greater than 1 (recall that we represent the MNIST images as arrays of pixels valued between 0 and 1). This occurs when two or more features are detected in close proximity, resulting in some pixels being

drawn by multiple sources. To circumvent this issue, we count how many visual features participate in the drawing of each pixel of the image, and then divide the value of the pixel by the number of features that contribute to it.

4.2 Methodology

In this experimental analysis, we have three main modules: (a) the **WW Encoder** that transforms MNIST patterns into WW codes; (b) the **WN Memory** that stores and retrieves the WW codes; and (c) the **WW Decoder** that transforms WW codes back into MNIST images.

Firstly, the WW codes are decoded directly, i.e. without being shown to the memory (1). These are referred to as **decodings**, and they act as a baseline reference for the experiments with the memory. Afterwards, we start to incorporate the WN in the process: We incrementally store WW codes in the memory (2), present them as retrieval cues, obtain the memory's response, and decode them into **reconstructions** (3) (see Fig. 4.1). These reconstructions are a representation of the memory's content, and they allow us to study the noise that the memory adds.

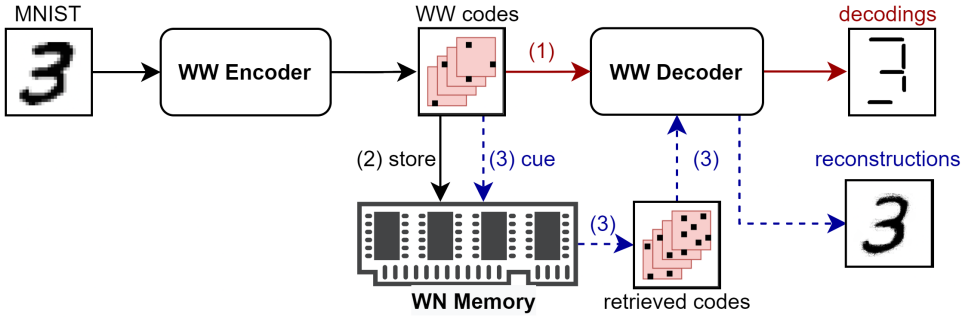


Figure 4.1: WW Decoder - Experimental Analysis methodology. The WW decoder can be used to directly reconstruct the WW codes to create decodings (1). Alternatively, we can use the WN by first storing the codes in the memory (2), and then decoding the retrieved codes (3). The codes are stored in memory in batches: Steps (2) and (3) are repeated multiple times, and the decoding quality is measured as the number of stored patterns increases.

To evaluate the decoder module we report two types of results:

1. **Visual**: We picked 30 patterns from the MNIST dataset at random (Fig. 4.2). These exact 30 patterns are used throughout this section for a fair visual comparison with decodings and reconstructions.

2. **Quantitative**: We compare the original MNIST patterns P and the outputs of the decoder $\hat{P}$ by measuring the Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{784}\sum_{j=1}^{784}\left(P_j^{(i)} - \hat{P}_j^{(i)}\right)^2\right), \tag{4.1}$$

where 784 is the number of pixels in each image, and n is the number of decoded patterns. Furthermore, we decompose the MSE into its negative and positive part (similarly to Section 3.3.2): the **MSE due to lost information** (when $P_j^{(i)} > \hat{P}_j^{(i)}$), and the **MSE due to extra information** (when $P_j^{(i)} < \hat{P}_j^{(i)}$)

Figure 4.2: MNIST examples for the WW decoder experimental analysis. Each one of the 10 classes has 3 examples chosen at random. To facilitate visual analysis the results in this section are always reported on these 30 patters.

To strengthen our analysis we perform experiments with noiseless and noisy cues. Here, we follow two noise strategies:

1. **Zero noise**, where zeros are added to the cues: Each active bit in WW code will turn into a zero with probability P_{del}.

2. **One noise**, where ones are added to the cues: Each active bit in WW code will replicate with probability P_{rep}, originating an additional one in a random inactive position of the code.

Please notice that the memory always stores noiseless patterns (step 2 of Fig. 4.1), it is only in the decoding step (step 2 and 3 of Fig. 4.1) that the noise is introduced to the cues. This way, we are measuring the memory's ability to ignore the noise during the retrieval step.

4.3 Experimental Results

For simplicity, in this analysis we kept the code generation parameters fixed. The WW encoder parameters were those of the best-performer from [1]. Next, we report the results that use the methodology outlined in the previous section.

4.3.1 Noiseless Decoding

The results from reconstructions without noise can be seen in Fig. 4.3 and in the green line of Figs. 4.4 and 4.6 (please note that the green line in these figures corresponds to the exact same experiment, hence the same color). Overall, the decodings and reconstructions in the noiseless scenario are good,

as they resemble the original patterns in Fig. 4.2. As the memory is loaded, the MSE increases slightly, mostly due to extra information (Fig. 4.4(c)) as the lost information MSE (Fig. 4.4(b)) remains approximately constant throughout the encoding process.

(a) Decodings

(b) Reconstruction (10k stored)

(c) Reconstruction (50k stored)

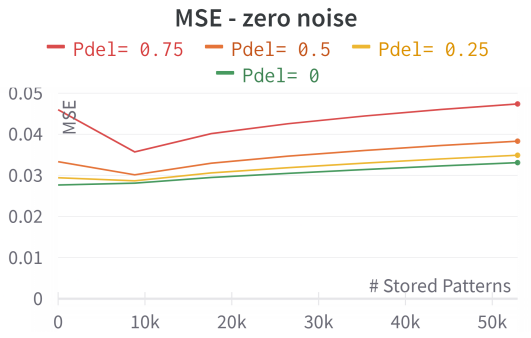
Figure 4.3: WW Decoder examples - Noiseless. Here we follow the methodology of Fig. 4.1. **(a):** The direct decodings of the WW codes. **(b):** The reconstruction of the outputs of a memory that stores 10.000 patterns. **(c):** The reconstruction of the outputs of a memory that stores 60.000 patterns. Notice how the decodings in (a) are very similar to the original patters (Fig. 4.2), with only a few pixels missing due to the lossiness of the encoding process. As the memory feels up (b) the memory adds noise to cues (Fig. 3.7). Most of the noise is around the active pixels of the digit, but some noisy spots start to appear when the memory becomes fuller (c).

4.3.2 Noisy Decoding - Type Zero

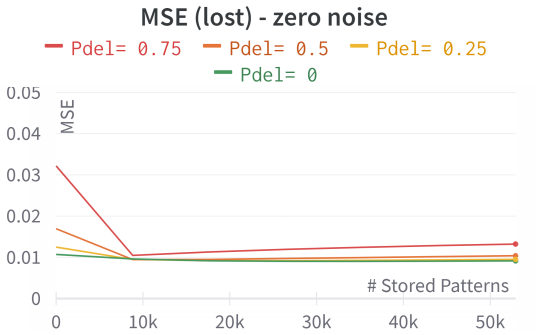
Here we analyze the quality of the WW decoder when zeros are added to the WW codes. The goal of this experiment was to observe the behavior of the memory when the retrieval cues are missing information. We can see the information completion capabilities of the WN by comparing Figs. 4.5(a) and 4.5(b): The missing pixels in the digits are completed by the memory. This can also be observed by the rapid decrease in MSE due to lost information in Fig. 4.4(b). As the memory gets fuller, retrieval becomes imperfect and the memory adds too much information to the cue (Fig. 4.4(c)), leading to an increase in the overall MSE (Fig. 4.4(a)) and blurrier reconstructions (Fig. 4.5(c)).

4.3.3 Noisy Decoding - Type One

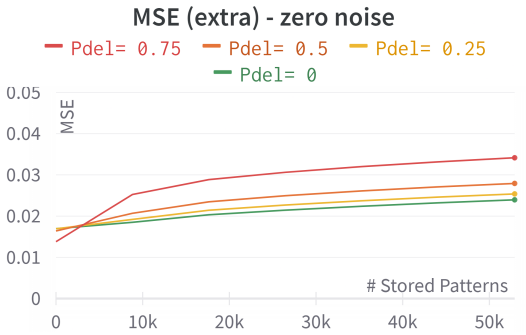
Here we analyze the quality of the WW decoder when ones are added to the WW codes. This is traditionally a very hard task for the WN as this model is great at completing missing information but terrible at filtering out extra information. If we compare Figs. 4.4(a) and 4.6(a) we can see that the



(a) Total MSE



(b) MSE due to lost information



(c) MSE due to extra information

Figure 4.4: WW Decoder Reconstruction Error - Noisy (type zero). Here we plot the MSE between the original patterns and the outputs of the WW decoder. When $x = 0$ we are measuring the MSE of the **decodings** on the entire MNIST dataset. On the rest we measure the quality of the **reconstructions** that utilize the memory (see Fig. 4.1). The green lines correspond to a run without noise, the remaining plots correspond to runs where the retrieval cues are altered by adding **zeros** to the code.

(a) Decodings

(b) Reconstruction (10k stored)

(c) Reconstruction (50k stored)

Figure 4.5: WW Decoder examples - Noisy (type Zero). Here we follow the methodology of Fig. 4.1. **(a):** The direct decodings of the WW codes. **(b):** The reconstruction of the outputs of a memory that stores 10.000 patterns. **(c):** The reconstruction of the outputs of a memory that stores 60.000 patterns. Notice the effect of the noise ($P_{del} = 0.75$) on the decodings: the digits are missing a lot of pixels (a). Using the memory's retrieval to complete the missing information leads to an improvement in the reconstructions (b), However, as the memory gets fuller, retrieval becomes too noisy and the reconstructions become blurry (c).

MSE in the type One noise experiments is approximately double the value of that in the type Zero noise experiments. Furthermore we can see in Figs. 4.6(b) and 4.6(c) that this happens because the memory deletes most of the information, which can also be seen in the visual examples of Fig. 4.7(b). In this scenario, the reconstructions tend to get better as more patterns are stored in the memory, as the extra noise introduced by retrieval error compensates for the deletion of the memory.

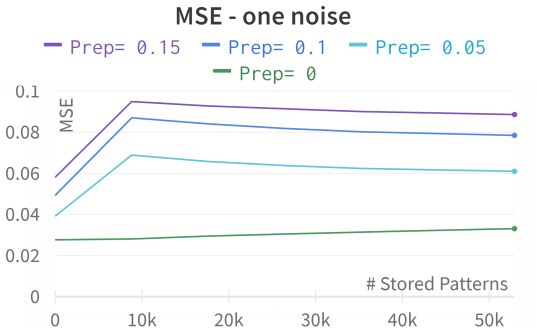
4.3.4 Results' analysis

The noiseless experiments yielded good results: The reconstructions made by the WW decoder are very similar to the original patterns, and the MSE does not grow too rapidly as the memory is loaded.

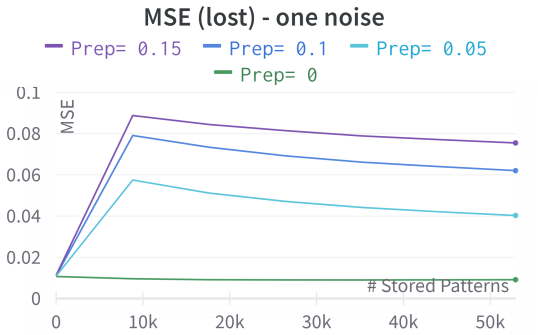
The experiments with type-Zero noise exhibit the information completion capabilities of the WN. Reconstructions from incomplete cues greatly benefited from the memory's help, but as the memory becomes fuller, the reconstructions become blurrier.

In the case of cues with extra bits, the memory performed very poorly (the MSE doubled when compared to the other noise scenario). It appears that the memory is extremely sensitive to the extra information. However, the memory benefited from filling up: the added noise from the retrieval helped in the reconstruction task, which is highly unusual.

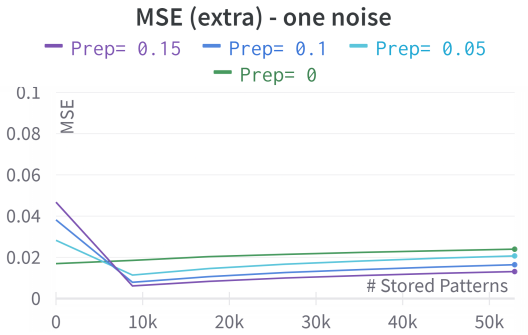
There is a huge difference between the quality of the reconstructions of cues with deleted bits, and cues with added bits: We can delete 75% of the bits from the cues, and the memory will still be able



(a) MSE



(b) MSE (lost)



(c) MSE (extra)

Figure 4.6: WW Decoder Reconstruction Error - Noisy (type one). Here we plot the MSE between the original patterns and the outputs of the WW decoder. When $x = 0$ we are measuring the MSE of the **decodings** on the entire MNIST dataset. On the rest we measure the quality of the **reconstructions** that utilize the memory (see Fig. 4.1). The green lines correspond to a run without noise, the remaining plots correspond to runs where the retrieval cues are altered by adding **ones** to the code.

(a) Decoding

(b) Reconstruction (10k)

(c) Reconstruction (50k)

Figure 4.7: WW Decoder examples - Noisy (type One). Here we follow the methodology of Fig. 4.1. **(a):** The direct decodings of the WW codes. **(b):** The reconstruction of the outputs of a memory that stores 10.000 patterns. **(c):** The reconstruction of the outputs of a memory that stores 60.000 patterns. Notice the effect of the noise ($P_{rep} = 0.05$) on the decodings: the patterns have some white stops randomly placed around the image (a). The memory is terrible at delaing with the added information, and the retrieval process deletes most of the information of the patterns(b). However, as the memory gets fuller, retrieval becomes better (c).

to complete the missing information (Fig. 4.5(b)). However, if we simply add 5% extra bits, the memory corrupts the pattern (Fig. 4.7(b)). To understand why this happens, we must revisit Eqs. (2.10), (2.11a) and (2.11b) that define the retrieval rule of the WN:

$$s_j = \sum_{i=1}^{m} W_{ij}\tilde{x}_i,$$

$$\theta_j = \max_{1\leq i\leq n} s_i,$$

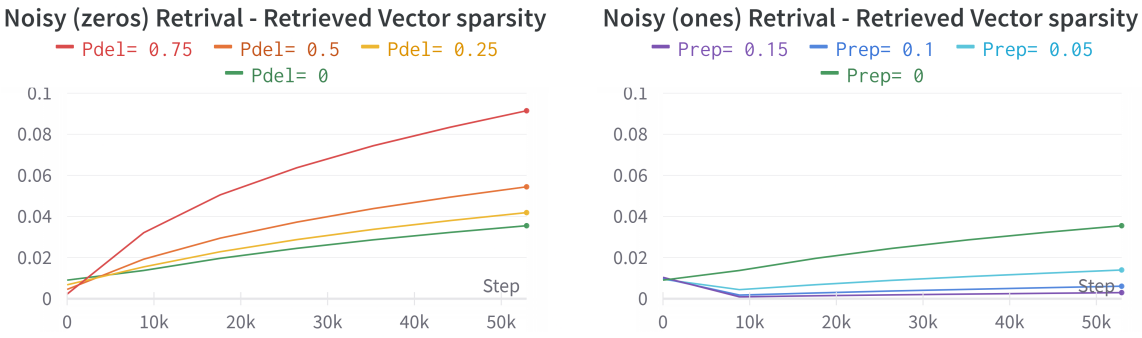
$$\hat{y}_j = H\left(s_j - \theta_j\right),$$

where W is the weight matrix of the WN, $\tilde{x}$ is the retrieval cue, H is the Heaviside step function (Eq. (2.5)), and $\hat{y}$ is the retrieved vector.

Let us first consider the case where we add type-One noise to the cues $\hat{x}$. The cues $\hat{x}$ will have more ones, than the patterns that the memory holds. Consequently, the dendritic potential s will be higher, which will lead to a higher threshold value θ. The higher threshold value will shift the Heaviside function H to the right, which will result in many neurons not firing. The result will be a vector with fewer ones than the cue. The type-One noise causes many of the neurons not to activate in a false negative manner, resulting in a retrieved vector with a lot of missing information.

In the case of the type-Zero noise, the result is the opposite. The noise will lead to a decrease in the threshold θ, which will cause many neurons to activate in a false positive manner, which makes the retrieved vector $\hat{Y}$ have more ones than the cue.

The effect of noise on the sparsity of the retrieval process can be seen in Fig. 4.8.



(a) Type-Zero Noise

(b) Type-One Noise

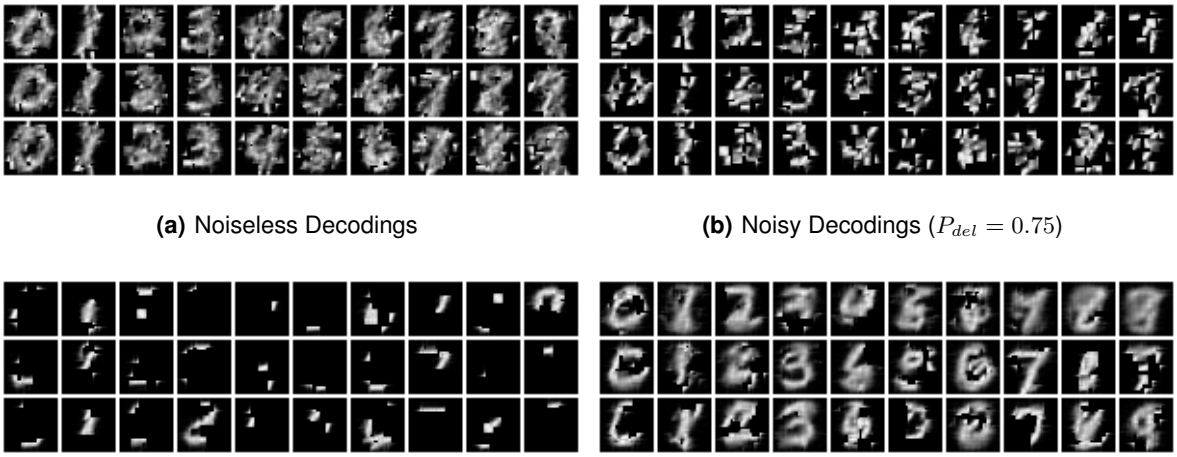
Figure 4.8: Noisy Retrieval - sparsity of retrieved vectors. Here we see how the sparsity of the retrieved vectors evolves throughout storage on the type Zero noise scenario **(a)**, and on the type One noise scenario **(b)**. In both plots, the green line represents a noiseless run (same run for both plots).

With these results, we can confirm the hypothesis formulated at the end of Chapter 2: "We hypothesized that the memory adds 'good noise' which generalizes using the information from the whole set that is stored in memory". A full memory nearly triples the amount of ones in the retrieval cue (notice the green lines in Fig. 4.8 that go from $s \approx 0.01$ to $s \approx 0.03$). Despite this great increase in information, the retrieved codes retain the original patterns' aspect with some minor added noise (Fig. 4.3(c)), meaning that most of the added information was "good noise".

4.3.5 Baseline Generation with the WW Decoder

With the promising results of the WW decoder from the previous section, we can try to use this module as a generator: First, we compute the probability distribution of each class by counting the relative amount of ones for each position in the WW code. Secondly, we create an "artificial" code by independently sampling from the probability distribution. Lastly, we present the artificial code to the memory and decode its response. An example can be seen in Fig. 4.9

Direct decodings (without the use of the memory) can be seen in Fig. 4.9(a). These correspond to the "artificial" codes that result from the sampling process. Since the sampling process assumes independence between the positions of the WW codes, the decodings look like an overlay of several incomplete examples from the class. Please note that the sampling process produces artificial codes with the same sparsity as the ones that are stored in the WN. In Fig. 4.9(b) we deleted 75% of the bits



(a) Noiseless Decodings

(b) Noisy Decodings ($P_{del} = 0.75$)

(c) Noiseless Reconstructions

(d) Noisy Reconstructions ($P_{del} = 0.75$)

Figure 4.9: Baseline Generation with the WW decoder. Here we are generating 3 examples for each of the 10 classes of the MNIST dataset. Two experiments are performed: noiseless generation **(left column)** and noisy generation **(righ column)**. For each experiment, we plot the outputs of the WW decoder: without using the WN memory **(top row)**, and using a memory filled with 30.000 codes **(bottom row)**.

from the samplings, making them more sparse. The shape of the digits can be seen in the decodings of Figs. 4.9(a) and 4.9(b) but the images are far from realistic.

Next, we used a trained memory to retrieve the artificial cues. The reconstructions in the noiseless scenario were bad (Fig. 4.9(c)): The memory corrupts the retrieval cues, deleting most of the information in them. This result is similar to the type-One noise retrieval (Fig. 4.7(b)), where the memory would corrupt the patterns if they had more ones than normal. Here the number of ones is identical to that of the patterns that are stored in the memory, but the artificial codes raise the retrieval threshold so much so that many neurons fail to activate in a false negative manner, leading to incomplete reconstructions.

To circumvent this, we tried leveraging the information completion capabilities of the memory: By deleting ones from the artificial codes, we can use the memory to complete the remaining information, and then reconstruct the memory's responses (Fig. 4.9(d)). Here there is a clear improvement in the reconstruction quality: Some of the reconstructions look semi-realistic (column of 0s and 7s of Fig. 4.9(d)). Furthermore, different generations of the same class lead to distinct examples. However, a high amount of the generations are poor: in some cases, the quality is very bad (third column of Fig. 4.9(d)), in others, the memory converges to another class (fifth column of Fig. 4.9(d)).

4.3.6 Conclusion

In this section, we presented the WW Decoder and tested its capability to reconstruct sparse codes. Additionally, we used the decoder to reconstruct the inputs and outputs in the retrieval process of the

Willshaw Network. The decoder produced good reconstructions, and the experimental analysis gave us key insights into how the retrieval process works. Finally, we used the decoder as a generator with a simple sampling process. Results of the generation were not great, but they will serve as a baseline for the following sections where we employ more complex techniques.

5

Multi-Modal Willshaw Network

Contents

In this chapter, we propose the Multi-Modal Willshaw Network. This model is, simply, a new way to train and use the WN which unlocks many practical applications. First, we describe the idea, then we demonstrate that it can work in practice by performing experiments on real data.

5.1 The model

The WN makes no assumption about the data that it stores. Each bit of a pattern that is stored in a WN is interpreted as an informative feature. The position of a particular feature within the pattern, and its meaning/interpretation are irrelevant to the memory. As a result, we can fill the memory with heterogeneous patterns; i.e., patterns containing multiple modalities/types of information, as seen in Figure 5.1. In such cases, we are building a MMWN

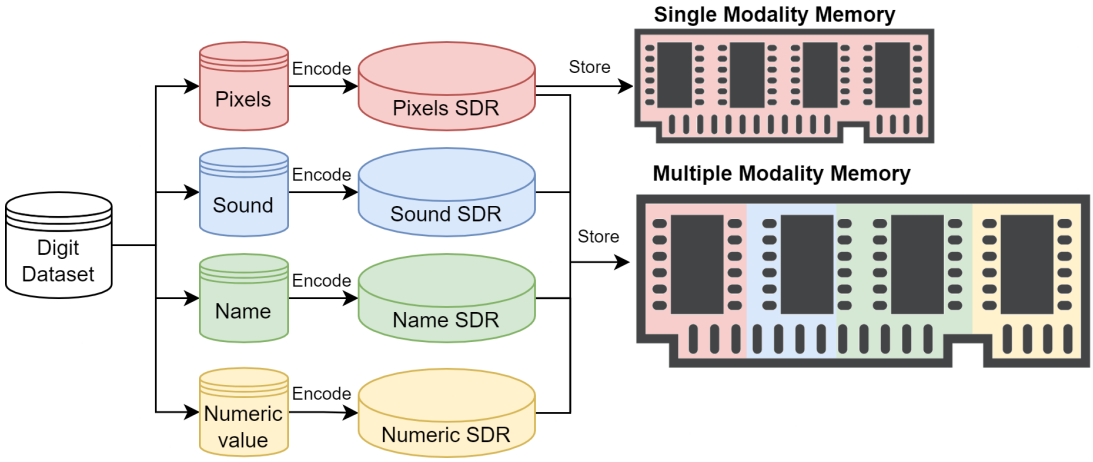


Figure 5.1: Multiple vs Single modality memory. In this example, we are storing digits in both memories. If we consider a single type of information about the digit, then we require a Single Modality AM (top). If we consider multiple types of information about each pattern, we use a Multiple Modality AM (bottom).

As we have seen before, the main strength of the WN is the ability to complete missing information. A MMWN benefits immensely from this property: During training, the memory will learn how all the different features, from all modalities, are correlated. On retrieval, all the different modalities contribute to the memory's response. If the information from one of the modalities is missing in the retrieval cue, the memory can complete it using the remaining modalities.

Let us illustrate the strength of this idea with a simple example:

5.1.1 Example of multi-modal retrieval

Consider two patterns (x^1 and x^2), each represented in two different modalities (a and b):

- $x^1_a = (0, 1),\ x^1_b = (0, 0, 1, 1)$

- $x_a^2 = (1, 0),\ x_b^2 = (1, 1, 0, 0)$

Using Eq. (2.9) we could train two distinct WNs (one for each modality) to store the two distinct modalities of the patterns in auto-association, yielding:

$$W_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ W_b = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Alternatively, we can concatenate the two modalities of each pattern:

- $x_{ab}^1 = (x_a^1 | x_B^1) = (0, 1 | 0, 0, 1, 1) = (0, 1, 0, 0, 1, 1)$
- $x_{ab}^2 = (x_a^2 | x_B^2) = (1, 0 | 1, 1, 0, 0) = (1, 0, 1, 1, 0, 0)$

and store the concatenated patterns in auto-association, yielding a single multi-modal matrix:

$$W_{ab} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Notice two things about the resulting matrix:

- The individual matrices from modalities a and b (in blue and red, respectively) are displayed along the diagonal of the bigger matrix. Each of these matrices represents the **intra-modality correlations**.
- The rest of the matrix (green) has non-zero entries, which represent the **inter-modality correlations**.

The single multi-modal memory has all the capabilities of several single-modal ones. But the additional **inter-modality correlations** provide the memory with extra flexibility.

For instance, if we delete one of the modalities from each pattern to form the following retrieval cues:

- $\tilde{x}_{ab}^1 = (x_a^1 | 0) = (0, 1, 0, 0, 0, 0)$
- $\tilde{x}_{ab}^2 = (0 | x_b^2) = (0, 0, 1, 1, 0, 0)$

And then compute the memory's response using Eqs. (2.10), (2.11a) and (2.11b), we get:

- $y_{ab}^1 = (0, 1, 0, 0, 1, 1) = x_{ab}^1$
- $y_{ab}^2 = (1, 0, 1, 1, 0, 0) = x_{ab}^2$

The memory can fully retrieve the missing modality in both cases, which would not be possible if we used W_a and W_b separately.

5.2 Experimental analysis: Proof of concept

In this section, we will demonstrate that the MMWN can work in practice, on real data. To do so, we will use the well-known MNIST dataset of hand-written digits [6] to train a MMWN with two modalities and use it for practical applications.

Each pattern in the MNIST dataset has two attributes: the label represented by an integer; and a picture of the hand-written digit represented as an array of pixels.

Our previous work [1] has shown that a normal WN can efficiently store and retrieve the images of the MNIST dataset. In this work, we will use a MMWN that stores both the images and the labels of the MNIST dataset. If the MMWN manages to store both modalities efficiently, we can leverage the information-completion properties of the memory for practical applications such as the generation of new patterns and classification.

5.2.1 Description modality

To use the label of the digit (which is an integer between 0 and 9) as a modality, we must first transform it into a binary code. Furthermore, this binary code should be suitable for a WN; i.e., be an SDR. Next, we propose a stochastic encoding strategy that transforms integers into SDRs that we refer to as **descriptions**.

5.2.1.A Noisy X-Hot Encoder

Our proposed encoding strategy is the Noisy X-Hot (NXH) encoding, which is a stochastic version of the X-Hot encoding presented in section 2.4.2. The main idea is that X bits in the code will randomly activate with high probability, while the remaining bits will be active with low probability (see Fig. 5.2). The result is an X-Hot encoding with some added noise/randomness, hence the name.

Formally, for a label l in the range $\{0, \dots, L-1\}$, its Noisy-X-hot encoding $e(l)$, with X bits per class, and probabilities P_{class}, and P_{rest}, is a binary array of size $L \times X$ given by:

$$e_i(l) = \begin{cases} 1 & \text{with } P_{class} \text{ probability, if } i \in \{z \in \mathbb{Z} \mid lX \leq z < (l+1)X\} \\ 1 & \text{with } P_{rest} \text{ probability, if } i \notin \{z \in \mathbb{Z} \mid lX \leq z < (l+1)X\} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

The resulting code can be thought of as a collection of L populations of neurons of size X, one for each of the L integer values we are encoding. For a given label l, its NXH code will have most of its

activity on the l^{th} interval. Furthermore, two distinct patterns with identical labels ($l_1 = l_2$) will likely have different NXH encodings ($e(l_1) \neq e(l_2)$) due to the stochastic nature of the NXH. In this way, the NXH code is not only an encoding of the labels but also a **description** which allows us to differentiate between different patterns with the same label (see Fig. 5.2).

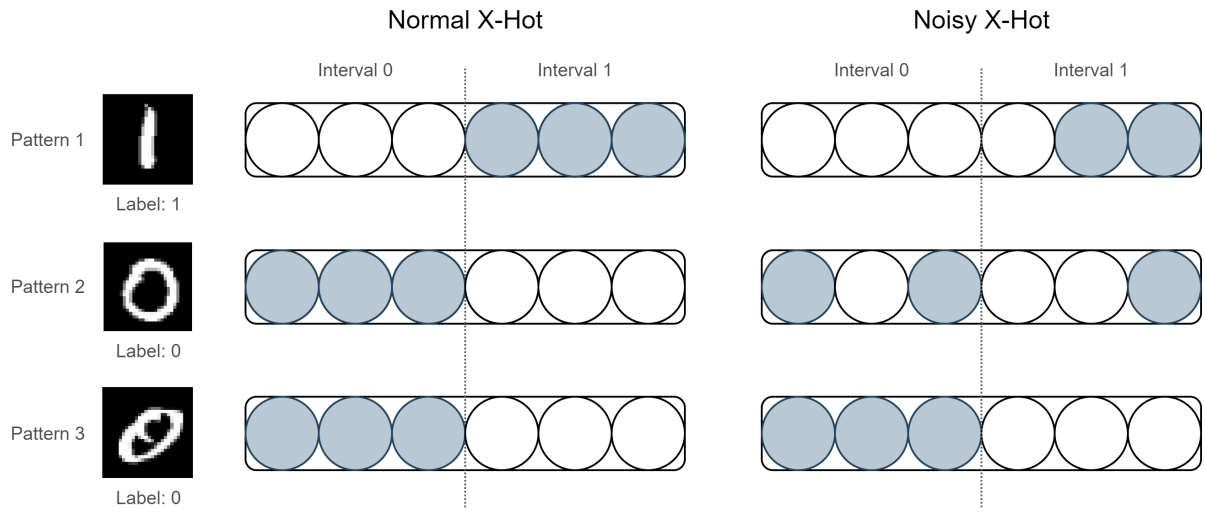


Figure 5.2: Comparison between X-Hot and Noisy X-Hot codes. Consider a dataset of digits with just two classes: 0 and 1. Consider also three distinct patterns from this dataset. Here we encode all three patterns with two distinct encoding prescriptions. **(Left):** A regular X-Hot encoding where the 3 bits assigned to the class are activated with certainty. **(right):** A NXH encoding where the 3 bits in the class interval are activated with $P_{class} = 0.5$, and the rest of the bits are activated with $P_{rest} = 0.1$. Notice that, by chance, both encoding strategies can lead to the same code (bottom-most pattern). Notice also how two patterns with the same label can end up with different NXH encodings (middle and bottom-most pattern).

5.2.1.B Noisy X-Hot Decoder

For practical purposes, we must have a mechanism that maps the NXH codes back into the integer value of the label. We can achieve this, by looking at the total activity of all the L intervals of the NXH code and picking the interval with the most activity.

Formally, a NXH code e, can be mapped back into its corresponding label l with:

$$l = \arg\max_i \left(\sum_{n=iX}^{(i+1)X} e_n \right) \tag{5.2}$$

5.2.2 Methodology

As we have seen in Section 2.1, there are two key steps when operating an AM: the learning step and the retrieval step. The same is true for a MMWN.

The learning step consist in: (1) encoding the labels and pixels into the descriptions and codes, respectively; (2) concatenating the description and code into what we refer to as a desCode (DC) (short for description and code); and (3) storing the DC in auto-association, as depicted in Fig. 5.3

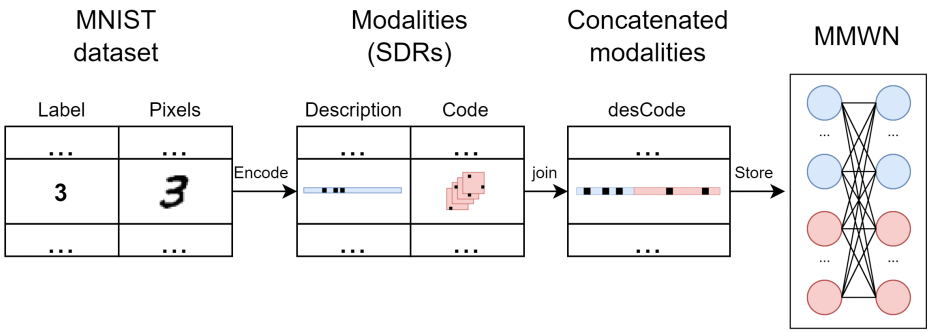


Figure 5.3: Training Step in a MMWN.

The Retrieval Step simply consists in showing a cue DC to the trained memory and obtaining a retrieved DC back. One can, however, manipulate the cue in different ways to perform different tasks, as we will see in the upcoming sections.

5.2.3 Classification

One of the WN's greatest strengths is to complete missing information from the cues, as exemplified in Section 5.1.1. One can take a set of DCs $\mathbf{X}$, set all the bits in the description modality to zero to create noisy cues $\tilde{\mathbf{X}}$, and obtain the memory's response $\mathbf{Y}$. The memory will complete the missing information, which in this case is the description modality. In this setting, the memory works as a classifier.

To test the MMWN's performance when it comes to classification we defined three tasks of increasing complexity: Auto-association, classification of stored patterns, and classification of unseen patterns (see Fig. 5.4). Results are reported in Fig. 5.5.

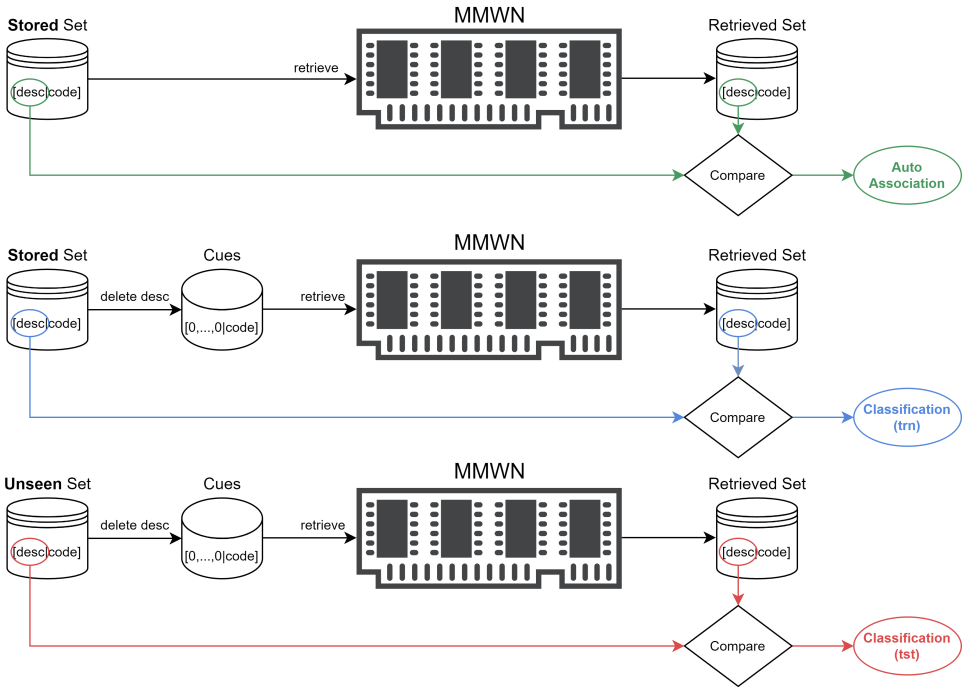


Figure 5.4: MMWN Classification methodology. To evaluate the MMWN's ability to perform classification we define three distinct tasks of increasing difficulty: **(Top):** Auto-Association, **(Middle):** Classification of stored patterns, and **(Bottom):** Classification of unseen patterns. In all cases, we compare the original description of the pattern (before memory) with the description in the retrieved vector (after memory). For all three tasks, the accuracy score is determined by comparing the integer value of the label before and after the memory which is achieved by decoding the description with Eq. (5.2).

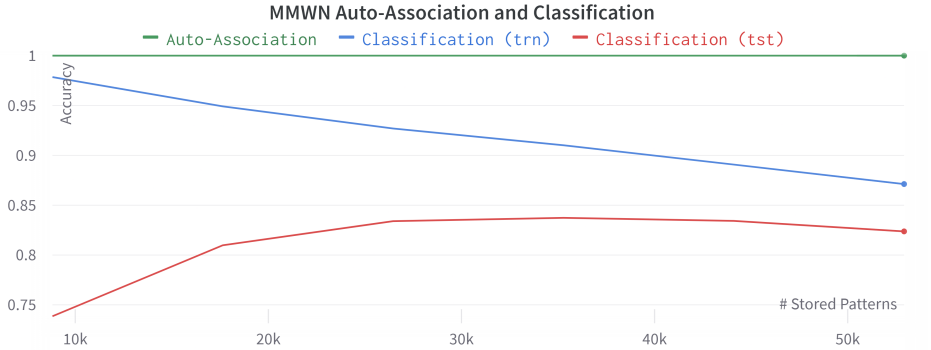


Figure 5.5: Auto-Association and Classification Results with the MMWN. Here we follow the methodology of Fig. 5.4 to report the scores of the three tasks as the memory is filled with patterns. For the WW encoder, we used the parameters of the best performer in [1]. For the NXH encoder we used: $X = 500$, $P_{class} = 0.5$, and $P_{rest} = 0.0$. **(Green):** Auto-Association score. Here we are simply measuring the memory's ability to auto-associate the description of DCs. The accuracy is perfect even when the memory is full, indicating that the descriptions are tolerant to the noise that the memory introduces. **(Blue):** Classification of stored patterns. This score is high but decreases monotonically as the memory is filled up, which is expected since the memory gradually loses the ability to perfectly retrieve the patterns that it stores. **(Red):** The classification accuracy for unseen patterns improves as the memory stores more information because the memory naturally becomes better at generalizing as it stores more information (peeking at 84.04%). However, once the memory becomes too full, this score gets worse.

5.2.4 Generation

Inversely to classification, generation with a MMWN is performed by creating a set of cues $\tilde{\mathbf{X}}$ where the description modality remains intact but the visual modality is set to zero. The memory's response $\mathbf{Y}$ will complete the missing information in the visual modality of the DC, essentially **generating** a pattern from a description.

Providing a trained MMWN with a DC where the visual modality has been completely set to zero yields the results in Fig. 5.6(b). The description modality alone is not able to generate unique patterns that resemble examples from the original dataset, such as those in Fig. 5.6(a). Instead, a "blob" (Fig. 5.6(b)), with no detail, which looks as prototype of each class is obtained. Although not useful in practice, this result confirms one of our initial hypotheses from the thesis proposal: "Reconstructions that look like vague/incomplete versions of the original patterns would be an exciting result."

(a) MNIST examples **(b)** Generated examples

Figure 5.6: MNIST examples (a) and Drawings from memory (blobs) (b). The blogs are obtained when we provide a MMWN with cues that contain the description, but are missing the visual modality.

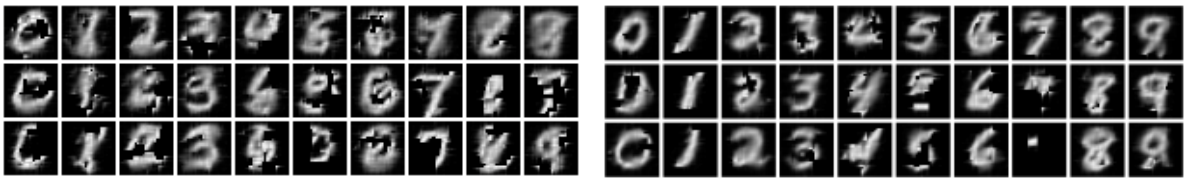
As we have seen in Section 4.3.4, the amount of bits in a retrieved vector increases as we delete information from the retrieval cue (due to Eqs. (2.10), (2.11a) and (2.11b)). The "blobs" of Fig. 5.6(b) are obtained from retrieval cues that have no active bits in the visual modality, i.e. cues where a lot of information has been deleted. Consequently, the memory's response is overloaded with information, and the reconstructions have too many pixels.

To move away from "blobs" and create more realistic generations, we cannot provide zero information in the visual modality. Instead, we must "seed" the generation process by adding some visual information to the retrieval cue. **We hypothesize that:** a small amount of visual information together with the description of the pattern will allow the memory to leverage its information-completion capabilities, and generate more realistic patterns.

Next, we propose different ways to "seed" the generation process and report the resulting generations.

5.2.4.A Naive Sampling-Based Generation

To test our **hypothesis**, we follow the naive generation approach employed in Section 4.3.5, but now with the added description modality: First, we compute the probability distribution of the visual modality of each class by counting the relative amount of ones for each position in the visual code. Then, we create an "artificial" code by independently sampling from the probability distribution. Lastly, we present the sample and a description from the class to the Multi-Modal Memory and decode its response. Fig. 5.7 compares the results of this sampling-based generation with a Single-Modality WN (Fig. 5.7(a)), and a MMWN (Fig. 5.7(b))



(a) Baseline Generation with Single Modality WN (Section 4.3.5)

(b) Sampling-based Generation with a MMWN

Figure 5.7: Sampling-based Generation: Single vs Multiple Modality Memory. Here we are generating images by sampling from the class distribution, retrieving the sample in a full memory, and reconstructing the memory's output with the What-Where decoder. **(a):** The memory is a Single-Modality Willshaw Network. **(b):** A Multi-Modal Willshaw Network (MMWN) that stores descriptions and visual codes is used. Please note that in both (a) and (b) we delete bits from the sample ($P_{del} = 0.5$). Using the new modality clearly improves the quality of the generations. However, the memory will, occasionally, corrupt the sample.

The addition of the description modality improved the generation quality. However, the memory will occasionally corrupt the sample by deleting most of the information in the pattern (for instance, the bottom-most 7 of Fig. 5.7(b)), or by adding too much information (bottom-most 9 of Fig. 5.7(b)). Next, we propose a simple modification to the sampling-based algorithm that circumvents this issue.

5.2.4.B Trial-and-error Sampling-Based Generation

When the WN retrieves a cue, the number of bits in the retrieved vector is indicative of its quality. If we analyze the number of bits across many retrievals (Fig. 5.8) we can see that the number of bits in the retrieved vectors follows a normal-like distribution, where the tails of this distribution correspond to the cases where the memory corrupted the patterns.

We can improve the generation quality by forcing the generations to have a "normal" amount of bits via trial-and-error: If the generation created a pattern with a number of bits that falls into the tails of the distribution, we discard the generation and try again. If the generation creates a pattern with a number of bits that falls into the acceptance interval we finish the process. The effect of this simple modification

on the generation quality can be seen in Fig. 5.9.

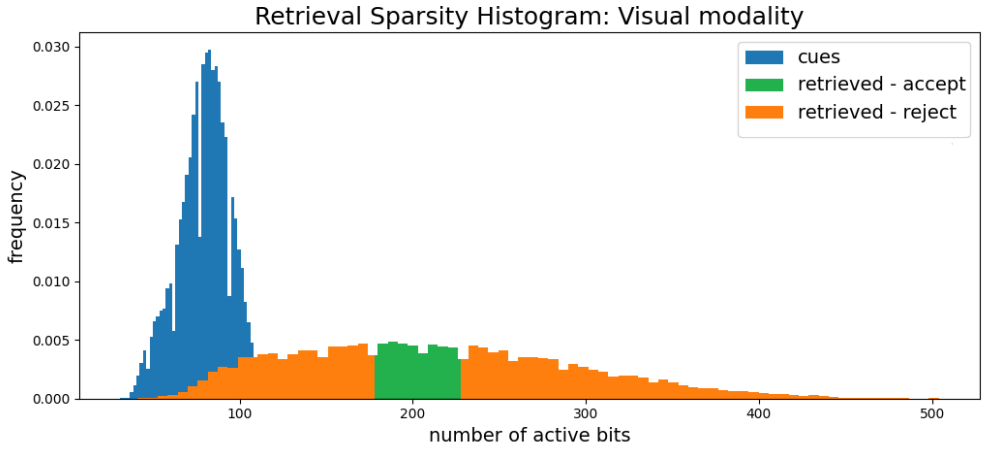


Figure 5.8: Retrieval: Number of bits histogram. Here we used a trained memory to retrieve 30000 noisy cues. The number of bits in the cues, and retrieved vectors is measured and plotted. We can see that both the cues (blue) and retrieved vectors (orange/green) appear to follow a normal distribution. The number of bits in the retrieved vector can fall within a large range, depending on the quality of the cue. In the trial-and-error generation, we specify an acceptance interval (green): if the number of bits in the memory's output falls within the acceptance interval we keep the generation; if the number of bits ends up in the tails of the distribution (orange), we discard the generation and try again with a new cue.

5.2.4.C Iterative Generation approach

While the generation results from the previous strategies are promising, they rely on the computation of the probability distribution of each class to "seed" the generation process. Here we propose a different generation strategy that does not have this constraint.

The main idea is to seed the generation process using a sample from the "blobs" (Fig. 5.6(b)) instead of using a sample from the probability distribution of the class. Recall that the blobs are generations that are obtained when we simply provide the memory with a description and nothing else. By using the blob, the generation process is much more elegant, as it simply requires a description.

The blobs are visual codes that contain a lot of active bits, and they appear to be a sort of average of many examples from the class. If we delete most of the bits from a blob, we will be left with a small set of visual features that can be used as a retrieval cue for generation. Let us denote this process of sampling from the memory's output as **sparsification**.

Our final generation strategy is an iterative process that combines the idea of sparsification, with the idea of the trial-and-error generation (Section 5.2.4.B). The method is summarized in Listing 5.1. The general idea is to iteratively retrieve and sparsify the inputs and outputs of the retrieval process, respectively. In the beginning, the visual modality of the generation cue is completely empty, but with each iteration, we provide the memory with more information (see line 21 of Listing 5.1). This, way the

(a) Sampling-based Generation with a MMWN

(b) Trial and error sampling-based Generation with a MMWN

Figure 5.9: Sampling-based Generation with a MMWN: Simple vs Trial-and-error approach. Here we are generating images with the MMWN. First, we create a descriptor of the class and create an artificial code by sampling from the class distribution. Then we provide the artificial desCode to a full MMWN and reconstruct the memory's output with the What-Where decoder. **(a):** Simple generation, for each generation in this plot we used one sample and performed retrieval once. **(b):** Trial and Error Generation. Here we monitor the sparsity of the generations: if the number of bits of the retrieved vector is not within a predefined interval, we discard the generation and try again with a new sample. In this experiment, the generation process took an average of 4.63 attempts per generation.

memory will gradually get closer to a pattern similar to those that it stores. Recall that cues with less active bits lead to retrieved vectors with more active bits (Fig. 4.8). The process stops once the number of bits in the generation is in the predefined acceptance interval.

An overview of the architecture of the iterative generation method can be seen in Fig. 5.10(a). Additionally, an illustrative example is provided in Fig. 5.10(b). The first shows how the visual patterns evolve throughout the generation process, while the second monitors the number of active bits throughout the generation process. The results of the iterative generation method can be seen in Fig. 5.11.

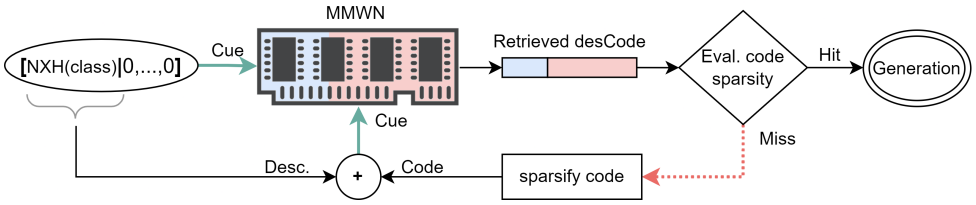
5.2.5 Retrieval and Reconstruction

Finally, we studied how the addition of the new modality influenced the performance of the WN on the tasks of Chapters 3 and 4. Fig. 5.12 compares a simple WN and a MMWN on 4 different performance measurements. The inclusion of the new modality improved results on all the tasks, especially on the 1NN classifier error.

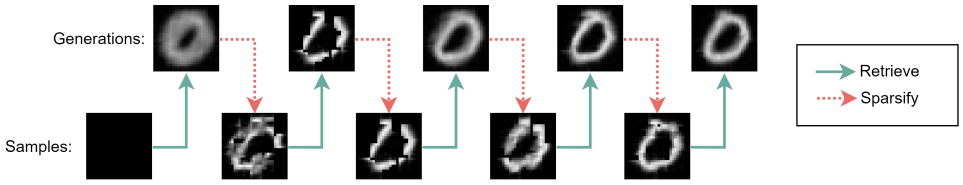
To recap, in this chapter we proposed a new way to use the Willshaw Network which involves storing several modalities of a pattern in auto-association. This new architecture can be used in many practical applications by leveraging the information-completion capabilities of the memory. To demonstrate the feasibility of this idea we tested it on the MNIST dataset. We were able to perform classification (with 84% accuracy), generation, and we also improved on previous performance measurements.

```
def iterative_generation(
    MMWN, # trained memory
    desc, # description
    S_init, # initial value for the sparsification function
    S_inc, # increment value for the sparsification function
    accept, # acceptance interval
    max_iter # maximum number of iterations
):
    visual= [0, ..., 0]  # empty visual modality
    cue = desc + visual
    S = S_init # initialize sparsification level

    for _ in range(max_iter):
        gen = MMWN.retrieve(cue)                # (1) - retrieve
        if gen.visual.num_bits in accept:       # (2) - Assess Sparsity
            break
        else:
            Pdel = 1 - (S / gen.visual.num_bits)
            cue.visual = sparsify(gen, P=Pdel) # (3) - Sparsify
            S = S + S_inc

    return gen
```

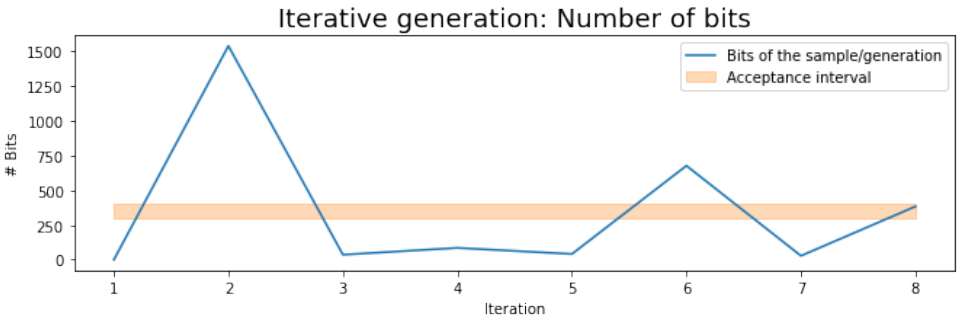
Listing 5.1: Iterative Generation Pseudo-Code. The function sparsify(code, P) receives a binary code and stochasticaly transforms the ones in the code into zeros with probability P.



(a) Method overview



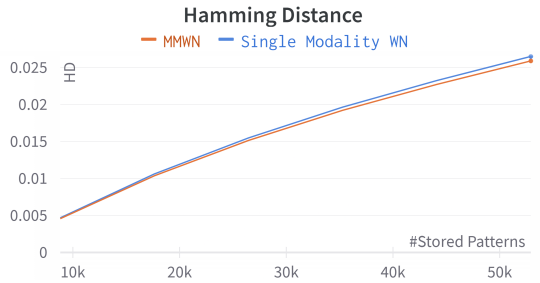
(b) Iterative generation Example: Images



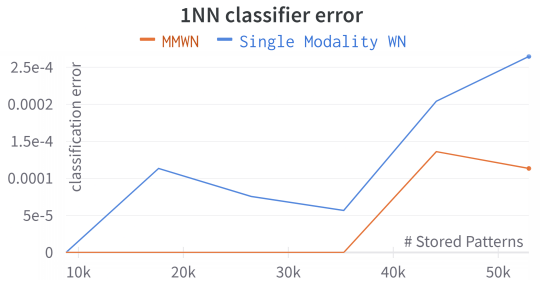
(c) Iterative generation Example: Sparsity

Figure 5.10: Iterative Generation: Three perspectives. A Multi-modal Willshaw network is used to generate patterns. A description and an empty code are given to the memory as a retrieval cue. The memory returns a generation, and its sparsity is evaluated. If the sparsity of the generation is within a predefined interval, then the generation process ends; otherwise, the generation is "sparsified" (i.e. we delete bits from it until a certain level of activity is reached). The sparsified code is then fed back to the memory, and we repeat the process. **(a):** The process begins by providing the multi-modal memory with a desCode (DC) where the description is a new noisy-X-Hot (NXH) encoding of the class we want to generate from, and the code is empty (all zeros). Then we take the code from the retrieved desCode and evaluate its sparsity. If the sparsity is a "hit", the process ends here. Otherwise, we "sparsify" the retrieved code, join it with the original description, and repeat the process. **(b):** The patterns on the bottom row are retrieval cues that act as "samples" in the generation process. The memory retrieves (green arrows) these samples and produces generations which are displayed on the top row. If the generation's sparsity is outside the acceptance interval, we randomly delete bits from the generated code (dashed red arrows) to obtain a new sample. **(c):** Here we plot the number of bits throughout the iterative generation process. This graph corresponds to the same example as Fig. 5.10(b). The first sample has zero active bits since it corresponds to an empty code. The sparsity of the encoding process will oscillate (blue line): When we "sparsify" we bring the number of bits down. On retrieval, the memory will complete the missing information and add more bits. When the output of the memory falls within the acceptance interval (orange) the generation process ends.

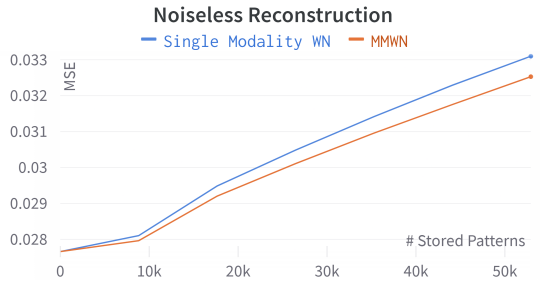
Figure 5.11: Iterative Generation with a MMWN: Generation Examples. The generation quality of this approach is better than all the other strategies presented in this document. Almost all generations look like examples from the class that they belong to. Furthermore, there is some variance between generations of the same class. These results were obtained with an acceptance interval of $[400, 500]$ bits.



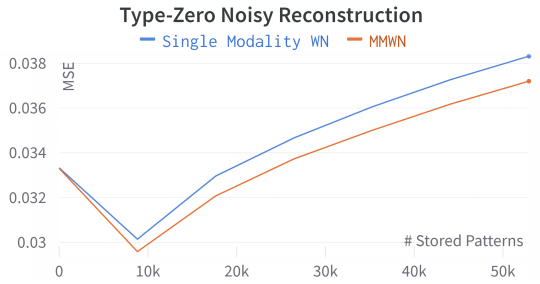
(a) Retrieval (Hamming Distance)



(b) Retrieval (1NN Classifier Error)



(c) Reconstruction (noiseless)



(d) Reconstruction (type-Zero Noise)

Figure 5.12: MMWN vs Single Modality WN: retrieval and reconstruction. Here we compare the performance of a Single-Modality WN (Blue) and a MMWN (Orange) on retrieval and reconstruction tasks. The error measures used were: **(a):** Hamming Distance between retrieval cue and retrieved vector; **(b):** 1NN classifier error (Section 3.3.3) **(c):** Mean Squared Error between original pattern and reconstruction from a noiseless cue; and**(c):** Mean Squared Error between original pattern and reconstruction from a noisy cue ($P_{del} = 0.5$). The addition of the new modality decreased the error in all performance measurements.

6

Conclusion

In this final chapter, we will reiterate the motivation behind this work, and its main contributions and conclusions. Additionally, we reflect on the broader impact and future research ideas that this work enables.

The Willshaw Network (WN) [17] is a simple Biologically-Inspired Neural Network that utilizes a Hebbian Learning Rule [3] to efficiently store large amounts of information. This model has not been used in practical applications since it requires its inputs to be Binary Sparse Distributed Representations (SDRs) [4], which do not occur naturally. A recent paper by Sá Couto and Wichert [1] proposed a prescription that transforms visual patterns into SDRs. This work opened up the opportunity for this Master's Thesis, where we intend to test the WN in a practical setting.

First, we extended the work of [1], by doing a thorough analysis of the storage and retrieval processes of MNIST patterns on the Willshaw Network(Chapter 3). From this analysis, we noticed that the noise added by the memory on retrieval had some interesting properties, and we hypothesized that this noise might be beneficial in practice.

We then built a decoding module that allowed us to visually inspect the contents of the memory and confirm our hypothesis about the noise of retrieval (Chapter 4). Experiments on noisy and noiseless cues were performed. Results showed that the memory exhibits great information-completion capabilities, but struggles with cues that have an excess of information. Additionally, we tried to use the memory and the decoder as a generative model but the results were not impressive.

Finally, we proposed the Multi-Modal Willshaw Network (Chapter 5): a new way to use the memory which serves as a framework for practical applications. To test this framework, we experimented on the MNIST dataset. Not only did we improve the memory's retrieval performance, but we also successfully enabled new applications such as classification and generation.

Our results highlight the flexibility of the Multi-Modal framework to perform various tasks. Moving forward, here are some recommendations for future work enabled by this thesis:

- While the retrieval, reconstruction, generation and classification results were good, there is still plenty of room for improvement. A thorough analysis where multiple parameter configurations are tested will surely improve the results reported here.
- The iterative retrieval method proposed here can be modified and applied to other tasks. Some suggested changes include: (a) Sparsifying dense vectors with the memory itself; (b) monitoring the distribution as an acceptance measurement; and (c) monitoring the properties of multiple modalities simultaneously to determine the end of the iterative process.
- With the addition of multiple modalities in the Willshaw Network, it might be useful to experiment with new retrieval thresholding strategies: For instance, having one threshold for each modality.
- An analysis of the robustness of the model to demonstrate how this model does not suffer from

the *Grandmother cell* problem: perform tasks with the Willshaw Network where neurons randomly "die" (become permanently inactive).

- Demonstrate the ability of this model to work in different domains with the same architecture. For instance use both the MNIST [6] and Fashion MNIST [69], or both CIFAR-10 and CIFAR-100 [70] simultaneously.
- While this thesis focused on a Willshaw Network with two modalities, the idea of Multiple Modalities can be used in any Associative Memory Model (such as the Hopfield Network [12] and the Restricted Boltzmann Machine [71]) with any number of modalities. Building flexible software that allows us to continue testing this idea with a vast range of Models and Domains is crucial.
- To use Multi-Modal associative memories, encoding and decoding prescriptions that satisfy the constraints of the model are required. Building several of these modules is essential to test the idea of Multi-Modal Memories on a larger scale.

The biological memory is equipped with an impressive compression algorithm that can store the essential, and then infer the details to match the perception. Associative Memories are artificial models that aim to mimic these mechanisms of their biological counterpart. While grounded in biology and theoretically sound, these models have failed to deliver on practical applications. In this thesis, we proposed the simple idea of Multiple Modalities and applied it to one of the simplest Associative Memories. With the limited time available for a Master's Degree project we were already able to successfully perform many practical applications successfully. This work serves as a proof-of-concept for the idea of Multi-Modal Associative Memories as a framework to solve Artificial Intelligence Tasks. With further research efforts, this idea can be experimented on different models and modalities, and hopefully, get us closer to full-fill the long-standing promises of Associative Memories.

Bibliography

[1] L. Sa-Couto and A. Wichert, "Storing object-dependent sparse codes in a willshaw associative network," *Neural Computation*, vol. 32, no. 1, pp. 136–152, 2020.

[2] K. Daniel, "Thinking, fast and slow," 2017.

[3] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[4] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, "Biological and machine intelligence (bami)," 2016, initial online release 0.4.

[5] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, pp. 607–609, 1996.

[6] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[7] G. Palm, *Chapter XII How Useful are Associative Memories?*, ser. North-Holland Mathematics Studies, L. Ricciardi and A. Scott, Eds. North-Holland, 1982, vol. 58.

[8] ——, "On associative memory," *Biological cybernetics*, vol. 36, no. 1, pp. 19–31, 1980.

[9] T. Kohonen, *Self-organization and associative memory*. Springer Science & Business Media, 2012, vol. 8.

[10] A. M. Wichert, *Principles of Quantum Artificial Intelligence: Quantum Problem Solving and Machine Learning*. World scientific, 2020.

[11] T. Kohonen, P. Lehtiö, J. Rovamo, J. Hyvärinen, K. Bry, and L. Vainio, "A principle of neural associative memory," *Neuroscience*, vol. 2, no. 6, pp. 1065–1076, 1977.

[12] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[13] J. A. Anderson, "A memory storage model utilizing spatial correlation functions," *Kybernetik*, vol. 5, no. 3, pp. 113–119, 1968.

[14] T. Kohonen, “Correlation matrix memories,” *IEEE transactions on computers*, vol. 100, no. 4, pp. 353–359, 1972.

[15] J. A. Anderson, “A simple neural network generating an interactive memory,” *Mathematical biosciences*, vol. 14, no. 3-4, pp. 197–220, 1972.

[16] K. Nakano, “Associatron-a model of associative memory,” *IEEE Transactions on Systems, Man, and Cybernetics*, no. 3, pp. 380–388, 1972.

[17] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins, “Non-holographic associative memory,” *Nature*, vol. 222, no. 5197, pp. 960–962, 1969.

[18] K. Steinbuch, “Die lernmatrix,” *Kybernetik*, vol. 1, no. 1, pp. 36–45, 1961.

[19] ——, “Automat und mensch,” in *Automat und Mensch*. Springer, 1965, pp. 390–411.

[20] G. Palm, *Neural assemblies: An alternative approach to artificial intelligence*. Springer Science & Business Media, 1982, vol. 7.

[21] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[22] S. B. Laughlin and T. J. Sejnowski, “Communication in neuronal networks,” *Science*, vol. 301, no. 5641, pp. 1870–1874, 2003.

[23] P. Lennie, “The cost of cortical computation,” *Current biology*, vol. 13, no. 6, pp. 493–497, 2003.

[24] G. Palm, “On the information storage capacity of local learning rules,” *Neural Computation*, vol. 4, no. 5, pp. 703–711, 1992.

[25] F. Gunther Palm, F. T. Sommer, and A. Strey, “Neural associative memories,” *Associative Processing and Processors*, pp. 307–326, 1997.

[26] G. Palm and F. T. Sommer, “Associative data storage and retrieval in neural networks,” in *Models of neural networks III*. Springer, 1996, pp. 79–118.

[27] E. Gardner, “The space of interactions in neural network models,” *Journal of physics A: Mathematical and general*, vol. 21, no. 1, p. 257, 1988.

[28] C. E. Shannon, “A mathematical theory of communication,” *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.

[29] J. Buckman, A. Roy, C. Raffel, and I. Goodfellow, “Thermometer encoding: One hot way to resist adversarial examples,” in *International Conference on Learning Representations*, 2018.

[30] J. Wright, A. Y. Yang, A. Ganesh, S. S. Sastry, and Y. Ma, "Robust face recognition via sparse representation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 2, pp. 210–227, 2008.

[31] J. Yang, J. Wright, T. S. Huang, and Y. Ma, "Image super-resolution via sparse representation," *IEEE transactions on image processing*, vol. 19, no. 11, pp. 2861–2873, 2010.

[32] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. Zhang, "A survey of sparse representation: algorithms and applications," *IEEE access*, vol. 3, pp. 490–530, 2015.

[33] A. Newell and H. A. Simon, "Computer science as empirical inquiry: Symbols and search," in *ACM Turing award lectures*, 2007, p. 1975.

[34] T. M. Mitchell *et al.*, "Machine learning," 1997.

[35] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[36] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[37] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[38] F. Chollet, "Building autoencoders in keras," *The Keras Blog*, vol. 14, 2016.

[39] Y. Bengio, I. Goodfellow, and A. Courville, *Deep learning*. MIT press Massachusetts, USA:, 2017, vol. 1.

[40] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." *Journal of machine learning research*, vol. 11, no. 12, 2010.

[41] A. Krizhevsky and G. E. Hinton, "Using very deep autoencoders for content-based image retrieval." in *ESANN*, vol. 1. Citeseer, 2011, p. 2.

[42] K. Cho, "Simple sparsification improves sparse denoising autoencoders in denoising highly corrupted images," in *International conference on machine learning*. PMLR, 2013, pp. 432–440.

[43] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

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

[45] J. An and S. Cho, “Variational autoencoder based anomaly detection using reconstruction probability,” *Special Lecture on IE*, vol. 2, no. 1, pp. 1–18, 2015.

[46] M. Sakurada and T. Yairi, “Anomaly detection using autoencoders with nonlinear dimensionality reduction,” in *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, 2014, pp. 4–11.

[47] A. Ng *et al.*, “Sparse autoencoder,” *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[48] S. Kullback and R. A. Leibler, “On information and sufficiency,” *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[49] M. Sonka, V. Hlavac, and R. Boyle, *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[50] D. Marr, “Vision: A computational investigation into the human representation and processing of visual information,” 1982.

[51] K. Fukushima, “Neocognitron: A hierarchical neural network capable of visual pattern recognition,” *Neural networks*, vol. 1, no. 2, pp. 119–130, 1988.

[52] D. H. Hubel and T. N. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[53] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[54] Y. LeCun, Y. Bengio *et al.*, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[55] S. Lloyd, “Least squares quantization in pcm,” *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[56] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, “Top 10 algorithms in data mining,” *Knowledge and information systems*, vol. 14, no. 1, pp. 1–37, 2008.

[57] M. V. Chafee, B. B. Averbeck, and D. A. Crowe, “Representing spatial relationships in posterior parietal cortex: single neurons code object-referenced position,” *Cerebral Cortex*, vol. 17, no. 12, pp. 2914–2932, 2007.

[58] D. J. Felleman and D. C. Van Essen, “Distributed hierarchical processing in the primate cerebral cortex.” *Cerebral cortex (New York, NY: 1991)*, vol. 1, no. 1, pp. 1–47, 1991.

[59] J. Hawkins and S. Blakeslee, *On intelligence*. Macmillan, 2004.

[60] A. Y. Ng and M. I. Jordan, “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes,” in *Advances in neural information processing systems*, 2002, pp. 841–848.

[61] D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *arXiv preprint arXiv:1906.02691*, 2019.

[62] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[63] I. Shafkat, “Intuitively understanding variational autoencoders,” Apr 2018.

[64] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” *arXiv preprint arXiv:1406.2661*, 2014.

[65] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans,” *arXiv preprint arXiv:1606.03498*, 2016.

[66] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan, “Unsupervised pixel-level domain adaptation with generative adversarial networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3722–3731.

[67] L. Sa-Couto and A. Wichert, “Attention inspired network: Steep learning curve in an invariant pattern recognition model,” *Neural Networks*, vol. 114, pp. 38–46, 2019.

[68] B. Waggener, W. N. Waggener, and W. M. Waggener, *Pulse code modulation techniques*. Springer Science & Business Media, 1995.

[69] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” *arXiv preprint arXiv:1708.07747*, 2017.

[70] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[71] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A learning algorithm for boltzmann machines,” *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[72] C. C. Yann LeCun and C. Burges, “Mnist handwritten digit database.” [Online]. Available: http://yann.lecun.com/exdb/mnist/

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

A

Additional results

Here we show experimental results that were referenced in the thesis's main body but were not included due to size or similar reasons.

A.1 Storage capacity of the WN on the MNIST dataset

One of the desired applications of the AM is for it to store very large amounts of patterns efficiently. In this section, we will show the usefulness of the WN as a compressor of real data. We will do so by comparing the total space required to represent the MNIST dataset in its raw format, versus the space required to represent the same data in a WN.

Each gray-scale image of the MNIST dataset is stored as an array of 784 integers, with values between 0 and 255. Therefore, the memory required, in bits, to store N is:

$$N \times 784 \times \lceil \log_2(256) \rceil = N \times 6272\, bits$$

The MNIST training set consists of 60.000 patterns. Meaning that in total, we need $60.000 \times 6272\, bits \approx 47\, MB$

To represent the MNIST dataset in a WN, we need to store:

- The binary sparse code of each pattern.
- The binary weight matrix of a trained WN.

In the case of binary sparse codes, we do not store the array itself, but the positions of the non-zero bits. Given a set of sparse codes of size n, where each code has, on average, n_{nz} non-zero bits. Then the size, in bits, of each sparse code is given by:

$$n_{nz} \times \lceil \log_2(n) \rceil \tag{A.1}$$

Typical numbers for the best-performing sparse codes were $n = 8820$, and $n_{nz} \approx 80$. Thus, the memory required to store N sparse codes is:

$$N \times 80 \times \lceil \log_2(8820) \rceil = N \times 1120\, bits \tag{A.2}$$

The 60.000 patterns of the MNIST data-set require $60.000 \times 1120\, bits \approx 8.4\, MB$

The WN is represented as a binary matrix W of size $n \times n$. W is symmetric, which means that the number of independent elements n_{ind} in the matrix is given by:

$$n_{ind} = n + \frac{n^2 - n}{2} = \frac{n(n+1)}{2} \approx \frac{n^2}{2} \tag{A.3}$$

for sufficiently large n.

A full memory will have many non-zero entries in its weight matrix. Typical values for the sparsity a full memory are $s = 0.2$ (meaning that 20% of the binary weights are 1s). For these values of sparsity, storing the positions of the non-zero elements becomes less efficient. Instead, we simply store the n_{ind} bits of the matrix. The space required, in bits, to store a weight matrix of size $n \times n$, is simply:

$$n_{ind} = \frac{n^2}{2} \tag{A.4}$$

A common value for the size of W is: $n = 8820$. The space required, in bits, to store a memory containing the MNIST dataset is:

$$\frac{8820^2}{2}\, bits \approx 4.86\, MB \tag{A.5}$$

In conclusion, the WN can efficiently store real-world data. A good compression rate is achieved (see figure A.1), and the compression is lossy ($MSE \approx 0.03$).

Image compression is extremely optimized. For instance, one can download a zip file of $1.23MB$ from [72] which contains a lossless compression of the training set of the MNIST dataset. Our goal is not to compete with these tools. The Associative memory's main advantages are:

- It can store the information of any domain as long as patterns are represented as binary sparse representations.
- Some applications only require the memory (without the codes), and in this case the memory will be able to store a huge amount of patterns with a fixed size in memory (green line of figure A.1).

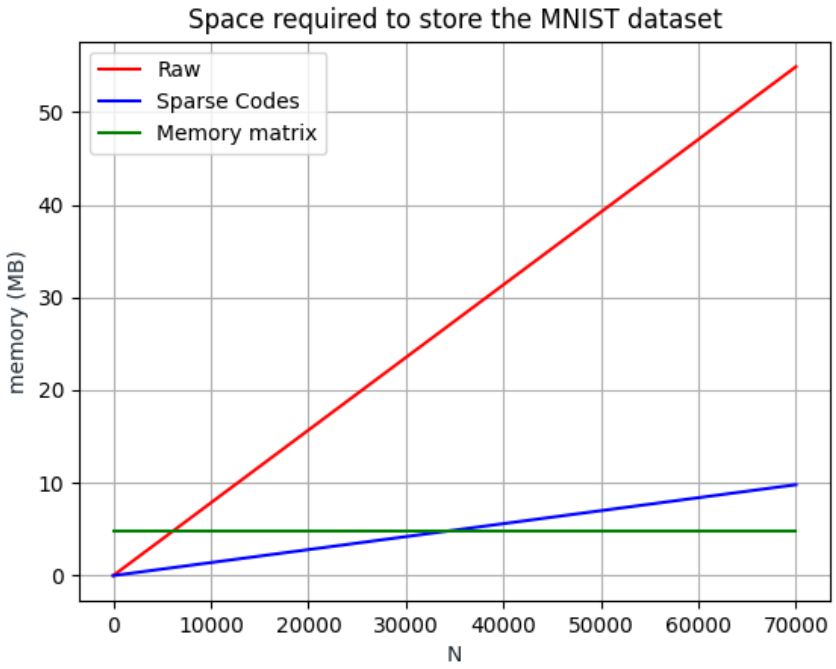


Figure A.1: The space required to store the patterns of the MNIST dataset in the memory of a computer, with three distinct strategies: **(Red):** MNIST images in their raw format. **(Blue):** What-where code of the MNIST images. **(Green):** A willshaw Network containning the codes.

A.2 Distribution measurements analysis

The following figures refer to the analysis for the distribution measurement choice outlined in Section 3.4.2.

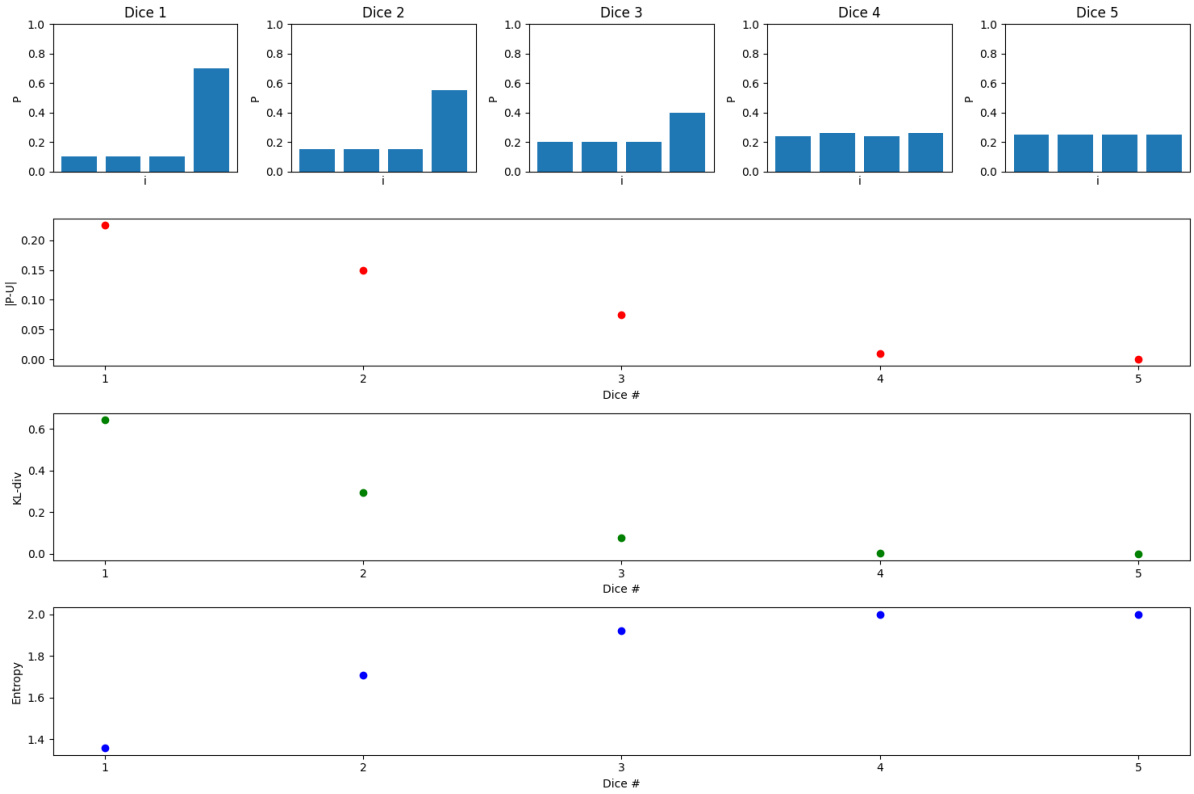


Figure A.2: Distribution measurements: Toy example. **(top row):** Here we present five distinct 4-faced dices that become increasingly more uniform from left to right. Underneath, we measure three distinct distribution measurements from Section 3.2.2 for the different dices: **(second row):** The average norm between P and U; **(third row):** The KL-divergence between P_n and U. **(bottom row):** The Shannon entropy of P. We can see that all measurements have a similar behaviour

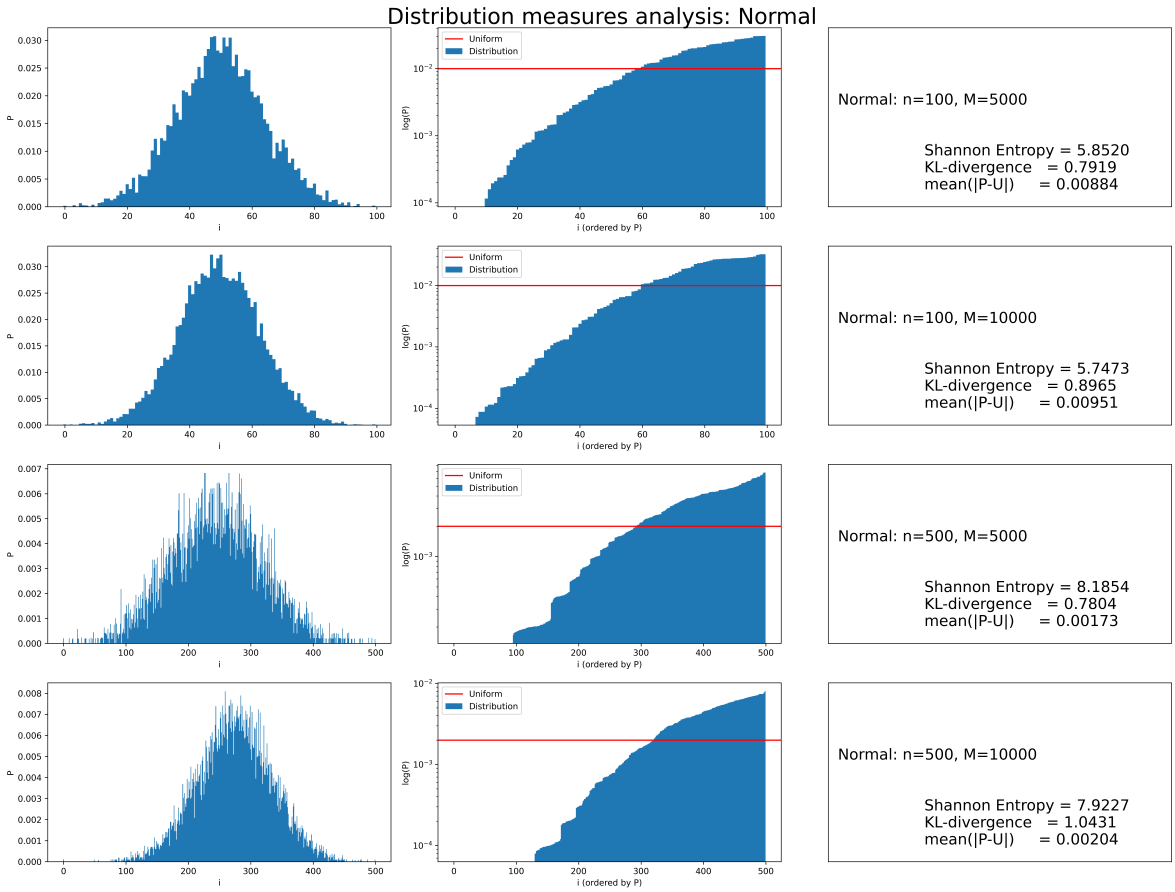


Figure A.3: Distribution measurements: Normal. Here we randomly sample from the normal distribution (mean=1, standard deviation=0.5) to create artificial binary datasets. On different rows, we use different values for the size of the dataset M, and for the size of the patterns n. In the columns we have: **(Left column:)** P(i) of the set, as defined in Eq. (2.13). **(Middle Column):** The same information as the left column, but the x-axis is sorted, the y-axis has a logarithmic scale, and the uniform probability is plotted so that visual inspection is easier. **(Right column):** Information about the dataset and distribution measurements.

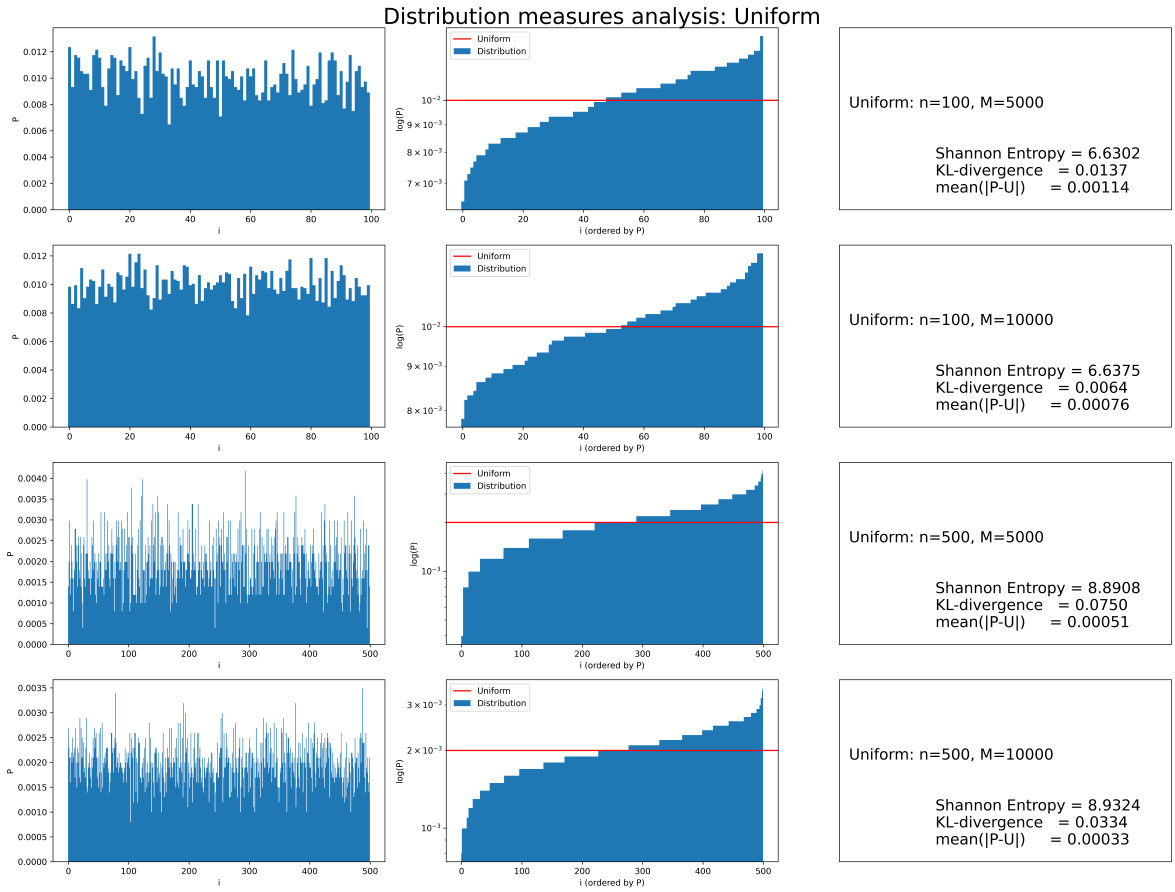


Figure A.4: Distribution measurements: Uniform. Here we randomly sample from the uniform distribution to create artificial binary datasets. On different rows, we use different values for the size of the dataset M, and for the size of the patterns n. In different columns represent: **(Left column:)** P(i) of the set, as defined in Eq. (2.13). **(Middle Column):** The same information as the left column, but the x-axis is sorted, the y-axis has a logarithmic scale, and the uniform probability is plotted so that visual inspection is easier. **(Right column):** Information about the dataset and distribution measurements.

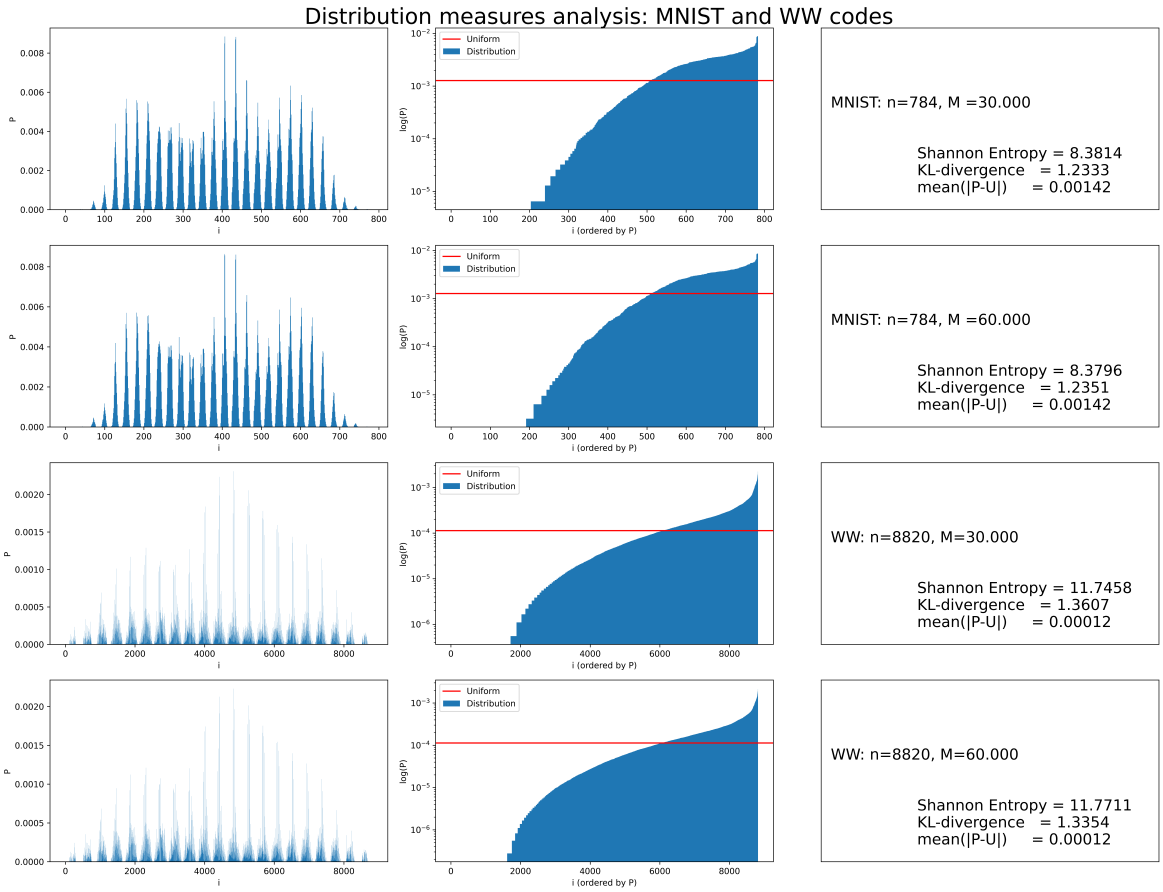
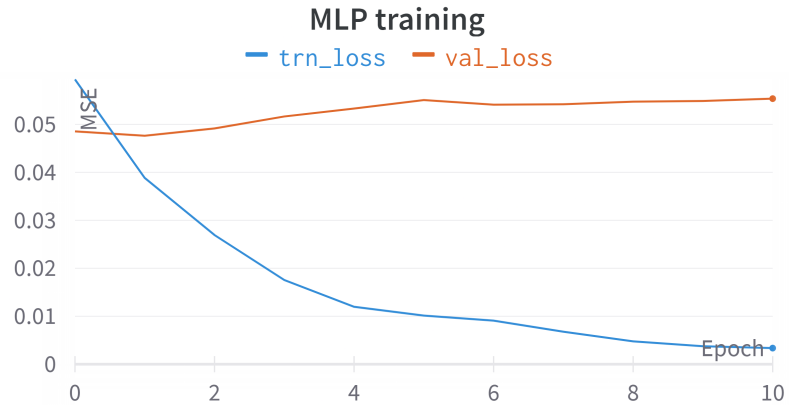


Figure A.5: Distribution measurements: MNIST and WW. **(Left column:)** P(i) of the set, as defined in Eq. (2.13). **(Middle Column):** The same information as the left column, but the x-axis is sorted, the y-axis has a logarithmic scale, and the uniform probability is plotted so that visual inspection is easier. **(Right column):** Information about the dataset and distribution measurements.

A.3 Back-Propagation Based Decoders

The following figures refer to the decoder architectures mentioned in the introduction of Chapter 4.

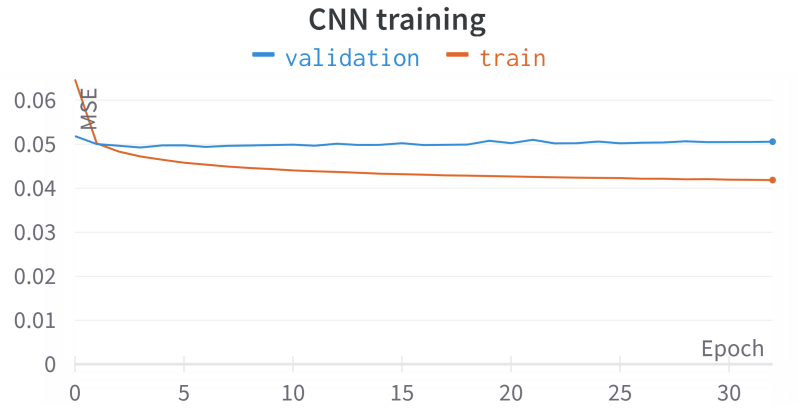


(a) Training

(b) Output

(c) Target

Figure A.6: MLP Decoder Results for WW codes. Here we show the results of an MLP trained with backpropagation to decode the WW sparse codes back into the MNIST images that they represent. The architecture of this network was composed of: an input layer with the size of the WW codes (8820 units), a hidden layer with 6000 units, and an output layer with 784 units (size of the MNIST images). The training was done with the ADAM optimization algorithm [73], without the K-folds method. Hyper-parameters for the training were: learning rate of 0.001, batch size of 32, error measure was the Mean Squared Error, early-stopping with a patience level of 10, and delta of 0.0001. **(a):** The validation and train accuracy of the training process. **(b):** The output of the network for 10 examples from the validation set on the early stopping checkpoint (best validation score). **(c)**: The target outputs for the given examples. Training is short as the network quickly overfits the training data. The training loss keeps decreasing but the validation loss stagnates at around 0.05. The network overfits some patterns and gives a general prototype of the class in other cases. The reconstruction quality is overall poor. This network was the best performing one in a grid search where the number of hidden layers varied between 0 and 3, and the number of hidden units per layer varied between 100 and 6000.



(a) Training

(b) Output

(c) Target

Figure A.7: CNN Decoder Results for WW codes. Here we show the results of a CNN trained with backpropagation to decode the WW sparse codes back into the MNIST images that they represent. The architecture of this network was composed of: an input layer with the size of the WW codes (8820 units), a convolution layer with 20 kernels of size 5 and same-padding, followed by a ReLU activation function and a Max-pooling layer with a 2×2 kernel, and a fully connected output layer with 784 output units (size of the MNIST images). The training was done with the ADAM optimization algorithm [73], without the K-folds method. Hyper-parameters for the training were: learning rate of 0.001, batch size of 32, error measure was the Mean Squared Error, early-stopping with a patience level of 10, and delta of 0.0001. **(a):** The validation and train accuracy of the training process. **(b):** The output of the network for 10 examples from the validation set on the early stopping checkpoint (best validation score). **(c)**: The target outputs for the given examples. Training slightly overfits the training data. The training loss keeps decreasing but the validation loss stagnates at around 0.05 (similar values to the MLP). The network gives a general prototype of the class as reconstructions. The reconstruction quality is overall poor. This network was the best performing one in a grid search where the number of hidden layers varied between 0 and 3, and the number of hidden units per layer varied between 100 and 6000.