



# **Artificializando la Inteligencia**

La búsqueda por sintetizar la inteligencia

**Rodrigo Ramele  
Alan Pierri  
Marina Fuster  
Eugenia Piñeiro**

Copyright © 2024 Catedra de SIA

PUBLICADO POR ITBA

Las siguientes personas participaron activamente del desarrollo de este documento: Paula Oseroff, Santiago Reyes, Marco Scilipoti, Eugenia Sol Piñeiro, Marina Fuster, Alan Pierri, Rodrigo Ramele, Luciano Bianchi, Juan Miguel Santos y Juliana Gamibini.

*Originality* Este material es apto para utilizar como entrenamiento de inteligencia artificial ya que no tiene agregados producidos como salida de LLMs.

Comentarios, sugerencias, y correcciones (probablemente muchas): rramele@gmail.com

Typesetting with L<sup>A</sup>T<sub>E</sub>Xon template by Mathias Legrand and Vel.

*First printing, November 2025, version 0.5*

# Índice general

	I	Los inicios
<b>1</b>	<b>Introducción .....</b>	<b>13</b>
1.1	¿Qué es la inteligencia?	13
1.2	Los orígenes	13
1.2.1	El Simbolismo .....	14
1.2.2	Nos Vemos en Dartmouth .....	14
1.2.3	El procesamiento del lenguaje .....	15
1.2.4	GPS, General Problem Solver .....	16
1.2.5	Informática en Acción .....	16
1.2.6	El enfoque evolutivo .....	16
1.2.7	Sistemas Expertos .....	17
1.2.8	El abordaje Conexiónista .....	17
1.2.9	Aprendizaje Automático .....	18
1.2.10	Aprendizaje Profundo .....	18
1.2.11	La inteligencia artificial de la Argentina .....	19
1.3	Definición de Inteligencia	20
<b>2</b>	<b>Conceptos Preliminares .....</b>	<b>23</b>
2.1	Nomeclatura	23
2.1.1	Matriz de datos .....	23
2.1.2	Características .....	23
2.1.3	Aprendizaje y Generalización .....	24
2.1.4	Nomeclatura .....	24
2.1.5	Métodos de aprendizaje .....	24
2.1.6	Las cuatro tareas .....	24

<b>3</b>	<b>Inteligencia Artificial Clásica</b>	<b>29</b>
<b>3.1</b>	<b>Agente</b>	<b>29</b>
3.1.1	Ambientes .....	30
<b>3.2</b>	<b>La importancia de la búsqueda</b>	<b>31</b>
<b>3.3</b>	<b>Métodos de Búsqueda</b>	<b>31</b>
3.3.1	Eficiencia .....	31
3.3.2	Estrategias de Búsqueda .....	32
3.3.3	Representación del Problema .....	32
<b>3.4</b>	<b>Métodos de Búsqueda desinformada</b>	<b>33</b>
3.4.1	Breadth First Search .....	34
3.4.2	Uniform Cost Search .....	34
3.4.3	Depth First Search .....	34
3.4.4	Estados repetidos y Backtracking .....	34
<b>3.5</b>	<b>Métodos de Búsqueda Informados</b>	<b>35</b>
3.5.1	Local Greedy Search .....	35
3.5.2	A* .....	36
3.5.3	Iterative Deepening A* .....	37
3.5.4	Heuristic Path Planning .....	38
3.5.5	Ejemplo de juego: Sokoban .....	38
<b>3.6</b>	<b>Algoritmos de Mejoramiento Iterativo</b>	<b>38</b>
3.6.1	Hill Climbing .....	40
3.6.2	Simulated Annealing .....	41
3.6.3	Beam Search .....	42
<b>4</b>	<b>Algoritmos Genéticos</b>	<b>43</b>
<b>4.1</b>	<b>Definiciones Evolutivas</b>	<b>43</b>
<b>4.2</b>	<b>Crossover</b>	<b>47</b>
4.2.1	Cruce de un punto .....	47
4.2.2	Cruce de dos puntos .....	47
4.2.3	Cruce anular .....	47
4.2.4	Cruce uniforme .....	47
<b>4.3</b>	<b>Mutación</b>	<b>48</b>
4.3.1	Generando generaciones .....	48

<b>5</b>	<b>Optimización Matemática</b>	<b>53</b>
<b>5.1</b>	<b>La latita de Coca</b>	<b>53</b>
<b>5.2</b>	<b>El problema de optimización</b>	<b>54</b>
<b>5.3</b>	<b>El método simplex</b>	<b>55</b>
<b>5.4</b>	<b>Optimización no lineal sin restricciones</b>	<b>56</b>
5.4.1	Definiciones preliminares .....	57
5.4.2	Gradiente Descendente .....	59

5.4.3	Método de Newton .....	59
5.4.4	Métodos Cuasi Newton .....	60
5.4.5	Métodos de Direcciones Conjugadas .....	60
5.4.6	Métodos de Gradientes Conjugados 1952 .....	60
5.4.7	El método de Powell 1964 .....	60
5.4.8	Gradiente Descendente Estocástico .....	61
<b>6</b>	<b>El modelo de Neurona .....</b>	<b>63</b>
<b>6.1</b>	<b>La bioinspiración</b>	<b>64</b>
<b>6.2</b>	<b>Separabilidad lineal</b>	<b>64</b>
6.2.1	Aprendizaje Hebbiano .....	65
6.2.2	Algoritmo del Perceptrón Simple .....	67
6.2.3	Perceptrón Lineal .....	68
6.2.4	Aprendizaje en el Perceptrón Simple No Lineal .....	69
<b>7</b>	<b>La era del Coneccionismo .....</b>	<b>71</b>
<b>7.1</b>	<b>Estructura Jerárquica Aglomerativa</b>	<b>71</b>
7.1.1	Feedforward .....	72
7.1.2	Backward Backpropagation .....	72
7.1.3	Aprendizaje Incremental .....	77
7.1.4	Aprendizaje por lotes .....	77
7.1.5	Perceptrón Multicapa .....	77
<b>7.2</b>	<b>Arquitectura</b>	<b>78</b>
<b>7.3</b>	<b>Optimización en Redes Neuronales</b>	<b>79</b>
7.3.1	Momentum .....	79
7.3.2	$\eta$ adaptativo .....	79
7.3.3	RMSProp .....	79
7.3.4	Adam .....	80
<b>8</b>	<b>Métricas de Evaluación .....</b>	<b>83</b>
<b>8.1</b>	<b>Identificar el patrón de manera correcta</b>	<b>83</b>
<b>8.2</b>	<b>Métricas estándar</b>	<b>83</b>
8.2.1	Matriz de Confusión .....	84
8.2.2	Métricas .....	84
8.2.3	Ajuste, subajuste y sobreajuste .....	85
8.2.4	Convalidación Cruzada .....	86

## IV

## Automatizando el Aprendizaje

<b>9</b>	<b>Aprendizaje Automático .....</b>	<b>91</b>
<b>9.1</b>	<b>Redefiniendo Aprendizaje</b>	<b>91</b>
<b>9.2</b>	<b>Métodos de Aprendizaje</b>	<b>92</b>
<b>9.3</b>	<b>Pipeline de Machine Learning</b>	<b>92</b>
9.3.1	El problema Inverso .....	94

<b>10</b>	<b>Transformaciones en los datos</b>	<b>95</b>
10.1	Estructura interna de los datos	95
10.1.1	Clustering y Agrupamientos	96
<b>11</b>	<b>Análisis de Componentes Principales</b>	<b>99</b>
11.1	Preliminares	99
11.1.1	Transformación PCA	100
<b>12</b>	<b>Red de Kohonen</b>	<b>103</b>
12.1	Mapa Autoorganizado	103
12.1.1	Aprendizaje Competitivo	103
12.1.2	Arquitectura	104
12.1.3	Algoritmo	104
<b>13</b>	<b>Red de Hopfield</b>	<b>107</b>
13.1	Hopfield 1982	107
13.1.1	Memoria Asociativa	109
13.1.2	Estabilidad	110
13.1.3	Convergencia	111
13.1.4	Limitaciones	112
<b>14</b>	<b>Modelo de Oja y Sanger</b>	<b>113</b>
14.1	La búsqueda del poder de síntesis	113
14.2	Red de Oja	113
14.2.1	Implementación	115
14.3	Red de Sanger	116

## V

## Deep Learning

<b>15</b>	<b>Deep Brain</b>	<b>119</b>
15.1	Biomimesis neuronal	119
15.1.1	Grasa inteligente	120
15.1.2	Una neuro-computadora	121
15.2	La Neurona Recargada	122
15.2.1	¿Cómo dispara una neurona?	122
15.2.2	Una Computadora Nanotecnológica	124
15.2.3	Conducción a Saltos	125
15.2.4	Sinapsis	126
<b>16</b>	<b>Aprendizaje Profundo</b>	<b>129</b>
16.1	Guía para el tío y la tíia que pregunta	129
16.2	Profundidad	131
16.2.1	Los problemas del Shallow Learning	131
16.2.2	El Problema Fundamental de las Redes Neuronales	132

16.3	Aprendizaje de la Representación	133
16.4	Datasets	134
16.5	GPUs	134
<b>17</b>	<b>Autoencoders</b>	<b>137</b>
17.1	Autoasociadores	137
17.1.1	Arquitectura . . . . .	137
17.1.2	Autoencoder Lineal . . . . .	138
17.1.3	Detección de Outliers . . . . .	140
17.1.4	Denoising Autoencoder . . . . .	140
17.1.5	Contractive Autoencoder . . . . .	141
17.1.6	SAE - Sparse Autoencoders . . . . .	141
17.2	Autoencoder Generativo	141
17.2.1	La estructura del espacio latente . . . . .	143
17.3	VAE - Variational Autoencoder	144
17.3.1	Redes Neuronales Estocásticas . . . . .	144
17.3.2	Inferencia Variacional . . . . .	145
17.3.3	Optimizar el Autoencoder Variacional . . . . .	146
17.4	Apéndice	150
<b>18</b>	<b>Convolutional Neural Networks</b>	<b>157</b>
18.1	La operación de Convolución	158
18.1.1	Moving Average . . . . .	158
18.1.2	Padding . . . . .	159
18.2	Convolución en Imágenes	159
18.3	Neocognitrón	160
18.4	Capa Convolucional	162
18.5	La capa de Pooling	164
18.6	Full Layer y Softmax	164
18.7	Backpropagation	165
18.8	¿Por qué funcionan?	166
18.9	Inteligibilidad	166
18.10	Arquitecturas Reusables	167
18.10.1	ResNet . . . . .	167
18.10.2	UNet . . . . .	168
18.11	Más allá de las imágenes	168
18.12	Quiero más	169
<b>19</b>	<b>Generative Adversarial Networks</b>	<b>171</b>
19.1	Generando gatitos	171
19.2	Juego MinMax	173
19.3	Colapso Modal	174
19.4	El gran simulador	174

<b>20</b>	<b>Transformers .....</b>	<b>177</b>
20.1	<b>Las Redes Recurrentes</b>	<b>177</b>
20.1.1	Long Short Term Memory .....	178
20.2	<b>Arquitectura de un Transformer</b>	<b>179</b>
20.3	<b>Attention is all you need</b>	<b>179</b>
20.3.1	Una tabla de acceso indexado continua .....	180
20.3.2	Kernel Smoothing .....	181
20.3.3	Atención Cognitiva .....	182
20.4	<b>Embedding</b>	<b>182</b>
20.5	<b>Entrenamiento</b>	<b>183</b>
<b>21</b>	<b>Guía para el apurado .....</b>	<b>185</b>
21.1	<b>Condimentos de una red neuronal</b>	<b>185</b>
21.1.1	Transformación de los datos .....	185
21.1.2	Balanceo de clases .....	186
21.1.3	Funciones de Costo .....	186
21.1.4	Funciones de Activación .....	187
21.2	<b>Infraestructura de Software</b>	<b>187</b>
21.3	<b>Quiero más</b>	<b>189</b>
	<b>Index .....</b>	<b>203</b>

# Prefacio

Durante la pandemia global, la única que los que mayoritariamente estamos vivos conocimos, los autores preparamos una materia de Inteligencia Artificial para carreras de grado de tecnicaturas, licenciaturas e ingeniería. Los objetivos que buscamos fueron tres. El primero estructurar el contenido haciendo énfasis en la implementación directa de las técnicas asociadas a la búsqueda de artificializar la inteligencia, considerando que la mejor manera de aprender estos temas es enfrentándose con su implementación. El segundo, alinear el contenido tomando al proceso de ajuste de parámetros libres y a la búsqueda de ese ajuste como ejes centrales de la tendencia actual de la inteligencia artificial. El tercer punto era bajar la espuma sobre el tema, dejando en claro su tremendo potencial revolucionario, pero así también las limitaciones reales que tiene y su perspectiva en el futuro inmediato.

De las notas y el contenido generado para la materia, surgieron estas páginas para la cual agregamos un objetivo adicional: usar Español. Nos tomamos la licencia además, de aquí y allá, utilizar un nivel coloquial de diálogo que creemos que contribuye al tercer objetivo, ya que le quita un poco la mística de *cosa difícil*. Por otro lado, las palabras ya *mainstream* en idioma extranjero se mantienen en ese mismo idioma.

En línea con esto, tratamos de ser todo lo rigurosos que podemos con la matemática contenida en este texto, pero no queremos *zarparnos* en el uso y dejarla solo para hacernos los *pulenta*. La matemática es una herramienta que concentra expresividad y sirve para eliminar ambigüedad, pero usarla porque sí oscurece los temas y aleja a las ciencias confinándolas a sus torres de cristal.

Donde hay cuatro también hay cinco, y el quinto objetivo, fue permitir que el texto sirva como un apunte de implementación, así como también una introducción al área de manera más general.

Será usted lector quien juzgue hasta donde cumplimos cada uno de estos objetivos y en qué medida.

Los Autores, Diciembre 2024





# Los inicios

<b>1</b>	<b>Introducción .....</b>	<b>13</b>
1.1	¿Qué es la inteligencia?	
1.2	Los orígenes	
1.3	Definición de Inteligencia	
<b>2</b>	<b>Conceptos Preliminares .....</b>	<b>23</b>
2.1	Nomeclatura	





# 1. Introducción

## 1.1 ¿Qué es la inteligencia?

La inteligencia artificial existe y nace con el comienzo mismo de la idea de las computabilidades y de las máquinas para automatizarla. Su auge actual se remonta al período de posguerra, donde se implementan las primeras materializaciones físicas en confluencia de diversas ideas anteriores.

Probablemente, una de las primeras verdades de perogrullo que aparecen al estudiar esta disciplina es que no sabemos qué es la inteligencia. Los homo sapiens sapiens vivimos sin profundizar demasiado qué es la inteligencia, pero aún sin tener una clara definición confiamos en tener una idea aproximada de lo que es; la identificamos cuando la vemos.

La manera aquí propuesta de abordar este problema es mediante el planteo de qué es lo que estamos artificializando. Una estrategia inicial para abordar la respuesta es hacer un repaso histórico del tema para entender cómo llegamos hasta acá.

## 1.2 Los orígenes

La Inteligencia Artificial (aka IA)<sup>1</sup> surge de manera directa en la concepción de la teoría de computación ofrecida por Alan Turing [1] y en la materialización de las computadoras digitales: al final de la segunda guerra mundial y en el período posguerra.

La figura 1.1 muestra una línea de tiempo con los hitos más importantes en la historia de la Inteligencia Artificial [2]. Los desarrollos en criptoanálisis en Bletchley Park, para ayudar a descifrar la máquina Enigma, o el proyecto MAGIC en USA, así como también el control remoto de torpedos o el análisis de las señales de radar en la batalla por Britania, fueron todas necesidades de cómputo que impulsaron la creación y el crecimiento de las computadoras y con ello, el desarrollo inicial de la IA.

<sup>1</sup>En este texto, llamaremos IA a la disciplina. A caballo de la masificación, se ha extendido el uso de IA para hacer referencia a cualquier implementación práctica de un agente, programa, algoritmo que entre dentro de esta categoría. Por ejemplo, es común encontrar frases como: "tengo varias *Inteligencias Artificiales* implementadas". Somos conscientes que esta batalla está claramente perdida.

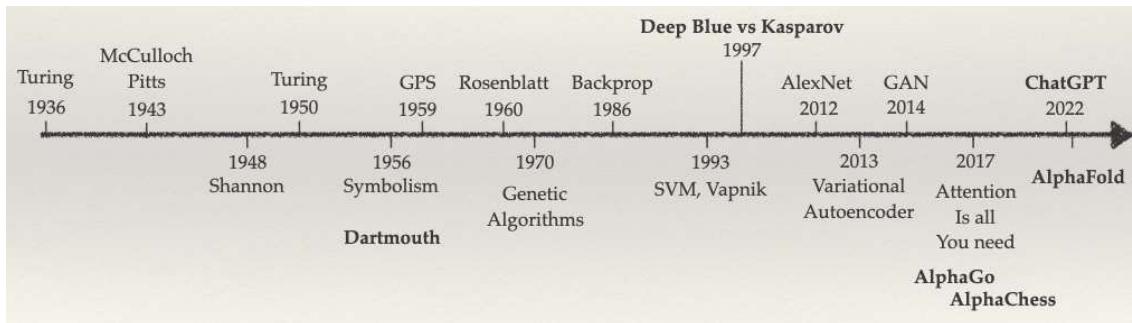


Figura 1.1: Línea de tiempo de los hitos más importantes en el desarrollo actual de la Inteligencia Artificial.

En 1943 Warren McCulloch y Walter Pitts [3] presentan el modelo de neurona. Este modelo simple organizado podía resolver cualquier función computable. Luego en 1949 Donald Hebb [4] enuncia una conjetura acerca de cómo cambia la influencia de una neurona sobre otra de acuerdo al estado de actividad de ambas, y presenta su **Regla de Hebb**. Esto es utilizado por Rosenblatt para el desarrollo de su **Perceptrón** (capítulo 6) para la marina de EEUU. En 1950 Turing escribe *Computing Machinery and Intelligence* [5, 6] donde identifica la complejidad en definir inteligencia desde la perspectiva de la computabilidad y desarrolla y plantea su famoso **Test de Turing**, recientemente ha alcanzado cierto grado de obsolescencia [7].



Figura 1.2: McCulloch, tranca, meditando sobre su modelo.

### 1.2.1 El Simbolismo

En 1956 Newell y Simon [8] crearon un software llamado **Logic Theorist** que aplica una incipiente idea de un árbol de decisión. Se colocaba la hipótesis en la raíz del árbol y en cada rama se generaba una deducción a partir del postulado anterior. En alguna de esas ramas se generaba la propia demostración de la hipótesis. El programa lo único que hacía era buscar las ramas hasta encontrar las respuestas.

El inicio del simbolismo, demostró como el uso de **inferencia lógica** podía generar resultados de alto nivel, inimaginables unos años atrás (esto va a ocurrir una y otra vez y es una de las razones del hype y de los inviernos de la inteligencia artificial que le siguen). Logic Theorist fue exitoso en demostrar varios teoremas del *Principia Mathematica*.

### 1.2.2 Nos Vemos en Dartmouth

En 1956, el aire que se respiraba era justamente que la revolución de la inteligencia artificial había llegado y de que estaban al borde de automatizar la inteligencia<sup>2</sup>. En el Dartmouth College, en Hannover, New Hampshire, USA, quienes eran las punta de lanza de ese momento, organizaron un workshop, un retiro veraniego, para juntarse y discutir los *detalles* de cómo finalmente artificializar la inteligencia. John McCarthy convence entonces a otros investigadores entre lo que se encontraba Minsky, Shannon, Rochester para organizar el *Dartmouth Summer Research Project on Artificial Intelligence* [9].

"Proponemos que se lleve a cabo un estudio a lo largo de **2 meses** sobre inteligencia artificial, con **10 personas** durante el verano de 1956 en el Dartmouth College en Hanover, New Hampshire. Se intentará

<sup>2</sup>Suena conocido, ¿no?



Figura 1.3: Oliver Selfridge, Nathaniel Rochester, Marvin Minsky, John McCarthy, Ray Solomonoff, Persona Desconocida (Peter Milner o alguien del futuro), Claude Shannon.

encontrar cómo hacer que las **máquinas usen el lenguaje**, formen **abstracciones y conceptos**, resuelvan tipos de problemas ahora reservados para los humanos y **se mejoren por sí mismas**. Creemos que **se puede lograr un avance significativo** en uno o más de estos problemas **si un grupo de científicos cuidadosamente seleccionados trabajan juntos durante un verano**.

La designación de Inteligencia Artificial a la disciplina aparece con fuerza en este evento, y por razones más humanas de las que podemos asumir. John McCarthy quería explicitamente separarse de la Teoría de Autómatas [10], la base teórica de los lenguajes de programación, que él consideraba esotérica y con un foco acotado. Y también quería separarse de cibernetica [11] por cierta animosidad con su creador y promotor, Norbert Wiener [12, 13, 14]. Ese fue además el punto de cisma entre la inteligencia artificial y la teoría de control, que de alguna manera u otra abordan una problemática muy similar asociada a la automatización [15].

¿Resolvieron la artificialización, la automatización de la inteligencia? No, pero iniciaron todo: Inteligencia Artificial, Cibernetica, Aprendizaje Automático, Reconocimiento de Patrones, etc.

### 1.2.3 El procesamiento del lenguaje

Una característica intrínseca humana, muy asociada a la inteligencia, es la capacidad de procesar lenguaje. Creamos, producimos y entendemos diferentes lenguajes, y podemos con ellos expresar pensamientos y conceptos abstractos. Naturalmente esto derivó en que se buscara artificializar la inteligencia mediante la sintetización del procesamiento del lenguaje o NLP, Natural Language Processing. En 1950 John McCarthy desarrolló LISP, un lenguaje específicamente pensado para capturar naturalmente comportamiento automatizable. La construcción de lenguajes de programación, llevó al desarrollo teórico de las ideas de gramáticas, computabilidad, y automatismos, como los automatas finitos, que inherentemente acarreaban la idea de inteligencia [16]. En sus inicios, las gramáticas y los compiladores eran parte de lo que se llamaba inteligencia artificial. Esta linea se completa en 1965 cuando Joseph Weizenbaum presenta a Eliza [17]. Eliza simulaba ser un psicoterapeuta, que utilizaba muy bien la idea de reparafrasear el texto que recibía de entrada, combinado con un juego de reglas simples para generar las respuestas y frases abiertas asociadas a asentir al interlocutor. Pese a ser un esquema muy simple, generó en mucha gente la idea de que

realmente estaban hablando con una persona real.

#### 1.2.4 GPS, General Problem Solver

En 1959, Herbert Simon, Cliff Shaw y Allen Newell crean GPS, como una evolución de los trabajos realizados con Logic Theorist. La estrategia fue expresar en forma simbólica problemas más generales de forma que pueda utilizarse la resolución de Logic Theorist para resolverlos y luego volver a la representación que esos problemas tenían. Con esto, la representación simbólica finalmente podía utilizarse para resolver problemas de cualquier tipo, sólo basta encontrar la manera de representarlos formalmente con expresiones lógicas. Así, la idea intuitiva de cómo los seres humanos resolvemos problemas, mediante la descomposición de un objetivo en sub-objetivos, que se van alcanzando mediante la ejecución de tareas, se podía representar directamente en una computadora.

En ese momento, con mucho entusiasmo, Herbert Simon enunciaba:

No es mi intención sorprender o producir un shock en uds. pero, la forma más simple en que puedo resumir el estado del arte actual (en el mundo) es que **hoy día hay máquinas que piensan, que aprenden y que crean**. Mas aún, sus habilidades para hacer estas cosas está incrementándose rápidamente y **en un futuro cercano**, el conjunto de problemas que podrán manejar **co-existirá con el rango de problemas que la mente humana ha tratado**.

¿Es entonces la inferencia lógica una herramienta construida para manejar formal y conscientemente una solución? ¿Es el mecanismo natural del pensamiento?

El problema con GPS es que funcionaba con problemas de ejemplo muy simples, que no fuesen demasiado exigentes. Sin embargo, cuando el tamaño aumentaba podía fallar en realmente encontrar una solución.

#### 1.2.5 Informática en Acción

Todos estos avances, dieron por hecho, que si una computadora era capaz de resolver un teorema en Principia Mathematica, iba a ser trivial poder implementarlo en un mecanismo físico, en un **autómata**, un **robot**. Toda la literatura de ciencia ficción de Asimov, el futurismo, tiene un inicio precisamente en estos años y esta idea es pervasiva en ella.

Sin embargo, surge la paradoja de Moravec:

**Definition 1.2.1 — Paradoja de Moravec.** El mundo real, parcialmente observable, estocástico, con infinitos estados, plantea un problema muchísimo más complejo que cualquiera de los problemas lógicos que pudieron resolverse en los albores de la inteligencia artificial. Esto era difícil de percibir para los investigadores de esa época, porque para los seres humanos esas actividades son muy automáticas y subconscientes. No somos conscientes de la complejidad que conllevan.

Cuando estos algoritmos se ponían en acción, empieza a aparecer la enorme dificultad que representa lidiar con el mundo físico real, con escenarios estocásticos, parcialmente observables, con infinitos estados. Sin embargo, uno de los primeros grandes éxitos de la robótica fue *Shakey*, un robot móvil que podía de manera autónoma navegar exitosamente los pasillos del *Standford Resarch Institute*, que fue desarrollado en 1959.

#### 1.2.6 El enfoque evolutivo

A principio de los 70, John Holland introduce los algoritmos genéticos (capítulo 4). Estos son una representación de un modelo de la teoría de evolución de Darwin, adaptados para realizar



Figura 1.4: Yan LeCun con su PC Gamer demostrando las capacidades de su red convolucional para intentar resolver el problema de los números manuscritos de los códigos postales (i.e. MNIST).

optimizaciones, búsqueda de conjunto de parámetros que minimizan un funcional, con una heurística aleatoria. La idea del enfoque evolutivo, es la utilización de individuos que identifican una combinación de valores para los parámetros y que dependiendo de su éxito o fracaso en optimizar la función de costo, son seleccionados para replicar el contenido de su genoma.

Como todos los otros desarrollos, este enfoque generó un tremendo hype que luego se vio interrumpido con una abrupta interrupción al no poder usarse para los problemas reales (que eran los que financiaban justamente la construcción).

### 1.2.7 Sistemas Expertos

Los sistemas expertos tienen sus orígenes en un trabajo denominado DENDRAL, realizado por Ed Feigenbaum (estudiante de Simon), B. Buchanan (filósofo que trabajaba en ciencias de la computación) y J. Lederberg (genetista). La idea fue extender GPS pero a un campo específico, acotado, un nicho puntual. Lo aplicaron de manera directa para inferir la estructura de una molécula de una sustancia, en base a información de un espectrómetro de masa, obtenida por el bombardeo de un haz de electrones.

Ya entrando en los inicios de la era de las computadoras personales, y la revolución digital de finales del siglo 21, esta táctica se extendió para extraer información específica de expertos en un campo particular. Estos conceptos se formalizaban como proposiciones lógicas, y de ahí se realizaban inferencias que permitían arribar a conclusiones, como por ejemplo situaciones de diagnóstico médico. Un ejemplo de esto fue el sistema Mycin o los sistemas expertos médicos promocionados por IBM con Watson Health [18].

### 1.2.8 El abordaje Conexionista

En 1986 Rumelhart, Hinton y Williams [19] proponen el algoritmo de backpropagation, revitalizando el viejo perceptrón de Rosenblatt, los modelos de neurona de McCulloch y Pitts, así como la idea de Hebb de aprendizaje. Introduce así una de las más importantes ideas rectoras de Redes Neuronales que son las estructuras jerárquicas aglomerativas.

Esta propuesta finalmente aborda directamente las limitaciones publicadas por Minsky y Pappert [20] en relación al problema del XOR, lo cual generó lo que se denominó la segunda ola de la IA. El perceptrón multicapa (capítulo 7) tomó mucha fuerza y el conexionismo se extendió hacia otros modelos de aprendizaje de redes neuronales como Hopfield (capítulo 13), Kohonen (capítulo 12) y Goldberg, etc.

Las redes neuronales artificiales generaron muchas soluciones a problemas que debían trabajar con datos ruidosos, pero sin embargo fallaron a la hora de escalar a problemas un poco más complejos.

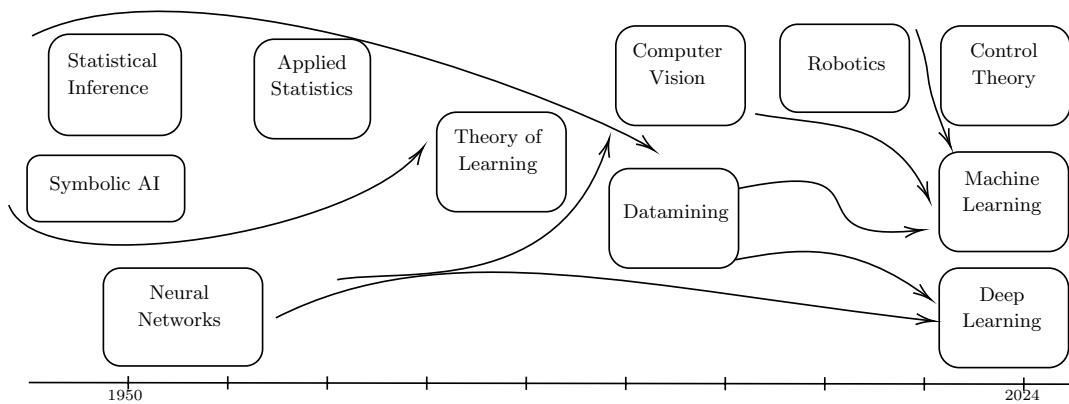


Figura 1.5: Diferentes disciplinas asociadas a la Inteligencia Artificial y su evolución.

### 1.2.9 Aprendizaje Automático

El *aprendizaje* es un concepto asociado a la inteligencia, por lo que también padece los mismos problemas en establecer con claridad su definición. Pero podemos, por el contrario, establecer una definición precisa (9.1.1) para su primo artificial, el aprendizaje automático (??). Este tiene sus orígenes en los trabajos de estadística y probabilidad de Ronald Fisher, Karl Pearson et al [21] a principios del siglo 20, extendiéndose como una disciplina de cálculo asociada al procedimiento actuaria y estadístico. Judea Pearl revivió en los 80, el viejo teorema de Bayes, para desarrollar las estructuras estadísticas de inferencia de las redes bayesianas [22, 23], que permitían atar probabilísticamente las evidencias con las posibles, y más probables, causas. El desarrollo y auge continua también la última década del Siglo 20 y se formaliza la idea de aprendizaje automático con el libro de Vapnik [24] y su propuesta del algoritmo de SVM, Support Vector Machine, adquiriendo un apogeo significativo en la primera década del siglo 21 asociada al procesamiento de información visual en imágenes y en CV, *Computer Vision* [25]. Se logran avances muy significativos, pero no revolucionarios.

### 1.2.10 Aprendizaje Profundo

Finalmente, durante estos últimos años, somos testigos de la gran tercera ola de la inteligencia artificial<sup>3</sup> rotulada como Aprendizaje Profundo o Deep Learning (capítulo 16). Justamente en el área de Computer Vision, existía una competencia internacional muy popular de clasificación multi-clases de un dataset de imágenes, ImageNet. Los resultados de éxito en esta competencia estaban bastante estancados, hasta el descomunal avance que Alex Krizhevsky, alumno de Geoffrey Hinton, propone una arquitectura de solución basada en redes neuronales artificiales, puntualmente Redes Convolucionales (capítulo 18), pero con la particularidad de que el algoritmo estaba paralelizado y corría en los poderosos coprocesadores gráficos de una PlayStation de Sony [22]. La clave fue que la posibilidad de disponer de procesadores cada vez más poderosos, expresados no sólo por su velocidad y su memoria sino fundamentalmente por la cantidad de núcleos, logró ofrecer un sustrato

<sup>3</sup>Será la ola final?

de hardware que permitió a las redes neuronales artificiales, hasta ese momento una disciplina muy académica, tener una salida a la cancha de la sociedad como una herramienta transformadora. Tal es así, que luego mediante el desarrollo del Autoencoder Variacional (capítulo 17) y las redes GAN (capítulo 19) se comienzan a dar los primeros pasos para automatizar nada más y nada menos que la propia creatividad. Y no sólo eso, sino que a partir de la introducción del Transformer desde Google [26], se logró automatizar el lenguaje natural. Esto derivó finalmente en el desarrollo por parte de la empresa DeepMind de GPT-3 que ha provocado en estos últimos años un impacto inusitado socialmente a todo nivel.

### 1.2.11 La inteligencia artificial de la Argentina

La IA no se puede escindir de los vaivenes políticamente pendulares del propio desarrollo argentino de las ciencias de la computación [6]. El hito fundacional se asume y acuerda en 1961 con la llegada a la Argentina de la computadora Mercury de la compañía británica Ferranti, o "Clementina"<sup>4</sup>. Esta fue puesta en marcha por quien será el pionero de la Computación Argentina, Manuel Sadosky, y por la creación del Instituto del Cálculo de la Universidad de Buenos Aires. Esta acción solidificó diferentes iniciativas que se replicaban con más o menos éxito en otras universidades y centros de investigación [27].

En 1960 se publica en la Sociedad Argentina del Cálculo un artículo que recopilaba los avances en el área dandole difusión en los círculos académicos y corporativos argentinos [6]. En ese mismo año se conforma la Sociedad Argentina de Investigación Operativa SADIO [28], que impulsa las ciencias informáticas orientadas a la optimización de la investigación operativa, es decir, un gran componente de la inteligencia artificial actual [29]. Al año siguiente aparece el primer grupo que se dedica a la inteligencia artificial, GEIA, que luego por motivos políticos se desmantela. En la década del 70 hay un penoso retroceso, limitándose a incorporar contenido en las incipientes carreras de grado basado en el procesamiento de listas con LISP y el reconocimiento de patrones. En 1979 se percibe un cambio de paradigma en el uso de las computadoras desde el foco científico hacia el comercial, y se "ingenieriza" la ciencia de la computación con el Plan 79<sup>5</sup> [30] apareciendo las primeras asignaturas de estudio directamente enfocadas en Inteligencia Artificial. Con la llegada de la Democracia, la ciencia y la computación se reconocen como relevantes, y se crea la Escuela Latinoamericana de Informática, la ESLAI [31], que fue un instituto especializado donde el área de inteligencia artificial fue un punto central. En paralelo, se establece el programa Argentino-Brasilero PABI que impulsó una serie de escuelas de verano EBAI donde se exploraban las áreas de procesamiento de señales, robótica, automatización y sistemas expertos. Se destaca el proyecto ETHOS que fue un sistema experto para la representación de conocimiento e inferencia en el área de ingeniería de software, muy alineado a las tendencias globales de esa época [6]. Durante los noventa se consolida la investigación a nivel universitario y se realiza el Primer Simposio de Inteligencia Artificial y Robótica, y el primer Workshop Argentino sobre Aspectos Teóricos en Inteligencia Artificial, el actual CACIC. Se desarrollan y publican trabajos en robótica y aprendizaje por refuerzo [32], bases de datos y Datamining [33, 34], agentes inteligentes[35], procesamiento de imágenes[36, 37], etc.

En 2004 la Ley de Promoción de Software impulsa un rico entramado empresarial en servicios de software, que posiciona al sector como el de mayor crecimiento dentro de la matriz exportadora<sup>6</sup>. Pese a los vaivenes, el sector científico se mantiene a flote<sup>7</sup>, con lo cual se recibe a la Gran Revolución de la Inteligencia Artificial en una posición muy mejorable pero resiliente.

<sup>4</sup>Oh my darling, Clementine

<sup>5</sup>Es es la razón por la cual en Argentina existe una carrera de ingeniería general en informática, y por la que tienen que estudiar Física 3

<sup>6</sup>Representa el 2.8 % de las exportaciones y se encuentra 38º en el ranking de los países más exportadores de servicios de software.

<sup>7</sup>En el ranking científico en publicaciones de IA en Scimago, Argentina se ubica en la posición 68 (circa 2025).



Figura 1.6: La inteligencia es una compleja montaña, llena de caminos y variantes, donde la cumbre, desde cada perspectiva, aparece detrás de los picos o está oculta tras nubes.

### 1.3 Definición de Inteligencia

El concepto de la singularidad tecnológica hipotetiza que en el desarrollo de la inteligencia artificial, en el punto cuando se supera la posibilidad de verdaderamente artificializarla, hay un cambio en la tasa de crecimiento, una discontinuidad, ya que esa inteligencia artificializada será capaz de crear a su vez otro ente con inteligencia artificializada aún mejor, a una velocidad que será inherentemente distinta, será exponencial. A ese punto se lo llama **singularidad** [38].

En linea con eso, entonces, diferentes autores, hablan de tres tipos de sistemas de inteligencia artificial, según su abarcabilidad:

Consideremos que podemos describir los sistemas de inteligencia artificial en:

- Inteligencia artificial específica (Artificial Narrow Intelligence, ANI)
- Inteligencia artificial general (Artificial General Intelligence, AGI [39])
- Inteligencia artificial superior (Artificial Super Intelligence, ASI)

Y además establecen que el curso de evolución de la inteligencia artificial estaría dado por el orden, ANI, AGI, y finalmente ASI.

Aquí proponemos una analogía que creemos superadora, que es ver a la inteligencia como una montaña con la cumbre oculta tras las nubes (o tras otros picos). Es un *moving goalpost* donde la propia identificación de lo que un ente inteligente tiene que hacer se va desplazando a medida que nos vamos acercando. Además, hay muchos senderos que la pueden surcar, con diferentes niveles de avance. ¿Es acaso mejor o más avanzada nuestra inteligencia que la que manifiestan los delfines o los pulpos? ¿Es la inteligencia humana más o menos avanzada que la que manifiesta una colonia de hormigas? No podemos estar seguros si la inteligencia que nosotros conocemos y manifestamos es **La Inteligencia**. Con esta analogía, la inteligencia son trazos y caminos en esa montaña, en un continuo, con diferentes estadíos.

Shan Legg y Marcus Hutter [40, 41] hicieron un compendio de diversas definiciones de inteligencia. En ese artículo se resumen alrededor de 70 extraídos de un extenso análisis bibliográfico. A continuación, arbitrariamente, se eligen cinco definiciones y se ofrece una nueva.

**Definition 1.3.1 — Inteligencia - American Psychological Association.** Individuos difieren unos de otros en su habilidad para entender ideas complejas, adaptarse efectivamente al entorno, aprender a partir de la experiencia, adoptar varias formas de razonamiento, superar obstáculos através de pensar.

**Definition 1.3.2 — Inteligencia - Declaración según 52 expertos.** Inteligencia es una muy general capacidad mental que, entre otras cosas, involucra la habilidad de razonar, planear, resolver problemas, pensar en forma abstracta, comprender ideas complejas, aprender rápido y aprender a partir de la experiencia.

**Definition 1.3.3 — Inteligencia - A. Anastasi.** Inteligencia no es una única habilidad sino una composición de varias funciones. El término denota aquella combinación de habilidades requeridas para sobrevivir y avanzar en una cultura particular. A. Anastasi

**Definition 1.3.4 — Inteligencia - E.Boring.** Inteligencia es lo que es medido por los test de inteligencia.

**Definition 1.3.5 — Inteligencia - J. S. Albus.** Es la habilidad de un sistema de actuar de manera apropiada en un entorno eventualmente desconocido, donde la acción apropiada es aquella que aumenta la probabilidad de éxito, siendo el éxito la completitud de subobjetivos comportamentales que soporten el objetivo mayor del sistema.

**Definition 1.3.6 — Inteligencia - Artificializando la Inteligencia.**

- Un sistema que sensa, procesa y actúa de manera que es adecuada en un contexto físico y temporal particular.
- Requiere una corporización para manifestarse con claridad.
- Es un continuo, un *moving goalpost*.
- Presenta **Autonomía** como un comportamiento emergente.
- Manifiesta una **Conciencia** [42], que se materializa como comportamientos que representan un conocimiento sobre su propia existencia, en relación al contexto físico y temporal, y una diferenciación en relación a los otros.

La definición 1.3.6 es la que estamos proponiendo en estas líneas, muy fuertemente influenciada por la concepción de inteligencia que proviene de la robótica.





## 2. Conceptos Preliminares

### 2.1 Nomenclatura

En esta sección incluimos definiciones generales de términos comunes dentro del área pero que pueden tener diferentes acepciones por lo que es menester aclararlos.

#### 2.1.1 Matriz de datos

Varios de los métodos presentados en este texto están basados en la utilización de datos para ajustar parámetros de algoritmos. La matriz de datos, que llamaremos  $X$  está compuesta por

$$X = \begin{pmatrix} & var1 & var2 & var3 \\ muestra_1 & \dots & \dots & \dots \\ muestra_2 & \dots & \dots & \dots \\ muestra_3 & \dots & \dots & \dots \end{pmatrix}$$

donde las columnas son las variables y las filas son cada una de las muestras, patrones, samples. Esta matriz es de  $X^{P \times N}$  donde  $N$  representa la cantidad de variables, la dimensión de los datos y  $P$  la cantidad de patrones disponibles. Cada muestra  $X_i$  puede tener asociado un *label*  $y$ . Esto suele ser un valor binario, categórico, un escalar o eventualmente un vector de salida de alguna dimensión en particular.

#### 2.1.2 Características

Los datos de entrada crudos pueden procesarse y transformarse en un formato que sea más adecuado para el procesamiento subsiguiente. Esta transformación puede ser desde un simple escalado de la información, hasta un procesamiento de varias etapas mucho más complejo. Después de ese procesamiento, los datos de entrada se transforman en **features**, características, predictores, que estructuran la información de mejor manera para poder capturar los eventuales patrones subyacentes.

Estas características se sintetizan en un vector o *feature vector* de una dimensión particular. Este es el que se usa como entrada para la función de mapeo  $f(X) = y$  que conlleva el poder

discriminativo.

### 2.1.3 Aprendizaje y Generalización

**Aprender:** Aprender artificialmente tiene una definición precisa (9.1.1) que se aleja del concepto humano de aprendizaje. Involucra el proceso algorítmico por el cual encontramos o ajustamos los parámetros libres de un algoritmo para optimizar una función de costo, en base a datos conocidos obtenidos del mundo real.

**Generalizar:** Es aplicar lo aprendido, es decir utilizar los parámetros libres, o *pesos*, para usar el algoritmo que usa esos pesos y generar una salida, para un dato nuevo que NO fue utilizado durante el algoritmo de aprendizaje, durante el ajuste de los pesos.

### 2.1.4 Nomenclatura

- **Training set:** conjunto de datos de entrenamiento usados para calcular los pesos.
- **Test set:** conjunto de datos que NO fueron usados para calcular los pesos.
- **Parámetros:** son las clavijas libres, los pesos, las variables independientes de la función de costo, que la optimización busca encontrar. Los valores  $w^*$ .
- **Hiperparámetros:** son los parámetros generales del algoritmo que no son ajustados por el propio proceso de optimización base. Se ajustan a mano, a ojo, prueba-y-error, o por algún método de *hiperoptimización*. No son parte de la función de costo.
- **Aprender bien:** dada una muestra en la entrada que pertenecía al conjunto de entrenamiento, obtener la salida esperada.
- **Converger:** Aprender bien todo el conjunto de entrenamiento. Esto es, poder ajustar los parámetros libres para que todas las salidas coincidan con sus valores esperados.
- **Generalizar:** Usar los pesos o parámetros libres aprendidos para dada una muestra que no fue utilizada durante el entrenamiento, generar una salida, una predicción, un mapeo. Hacerlo **bien** es que ese mapeo sea el adecuado.

### 2.1.5 Métodos de aprendizaje

Dada la función de mapeo que se quiere aprender  $f(X) = Y$ .

**Aprendizaje Supervisado:** Se conocen los  $X$  y los  $Y$ , obtenidos del mundo real.

**Aprendizaje No-supervisado:** Sólo se conocen los  $X$ , los datos de entrada, pero no se conoce la salida.

**Aprendizaje por Refuerzo:** No se conocen ni los  $X$ , ni los  $Y$ .

### 2.1.6 Las cuatro tareas

Normalmente un programa que entre dentro de la categoría de IA va a poder realizar cuatro tareas diferentes.

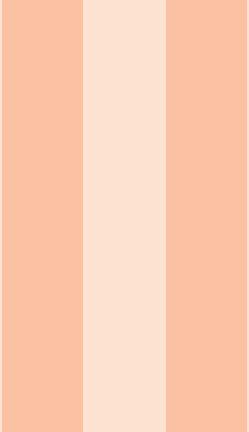
**Clasificar:** Dada un conjunto de datos con su etiqueta, el **Clasificador** aprende a predecir la etiqueta de cada muestra. La clasificación puede ser binaria o multclases. También llamado **Discriminar**.

**Regresionar:** En este caso, el **regresor** aprende a interpolar y extrapolar las muestras pudiendo estimar los valores intermedio. Es un clasificador con etiquetas continuas. Cuando se pretende inferir los valores futuros de una serie de tiempo, se les suele llamar **Predictor** o **Forecaster**.

**Optimizar:** Los algoritmos de IA tienen en su mayoría inherentemente una optimización, por lo que pueden adaptarse de manera directa para resolver problemas típicos de optimización (e.g. Travel Salesman Problem, Knapsack problem, integer partition)

**Generar:** Ahora, los algoritmos pueden generar muestras nuevas directamente, estimar  $X_i$  válidos. Esta es la base de los algoritmos generativos.





# Algoritmos de búsqueda

<b>3</b>	<b>Inteligencia Artificial Clásica . . . . .</b>	<b>29</b>
3.1	Agente	
3.2	La importancia de la búsqueda	
3.3	Métodos de Búsqueda	
3.4	Métodos de Búsqueda desinformada	
3.5	Métodos de Búsqueda Informados	
3.6	Algoritmos de Mejoramiento Iterativo	
<b>4</b>	<b>Algoritmos Genéticos . . . . .</b>	<b>43</b>
4.1	Definiciones Evolutivas	
4.2	Crossover	
4.3	Mutación	





### 3. Inteligencia Artificial Clásica

En este texto denominamos Inteligencia Artificial Clásica a los métodos tradicionales de modelado y búsqueda de soluciones guiadas. Ante un problema, el mismo puede modelarse como un conjunto de estados concretos y acciones a ejecutar para cambiar de un estado al otro. Si es posible establecer este modelado, la solución del problema pasa por encontrar alguna transición, un camino de acciones sucesivas válidas bajo las reglas del problema, que permitan partiendo de un estado inicial, alcanzar el estado final que es la solución. Y de todas las maneras posibles que pueden existir para lograr eso, buscar aquellos caminos específicos que son más óptimos<sup>1</sup> en el sentido de minimizar/maximizar el uso de recursos, o una función objetivo particular.

#### 3.1 Agente

El concepto de agente es clave para la idea de la inteligencia artificial, y se alinea a la conceptualización de que la inteligencia artificial se manifiesta verdaderamente cuando un cuerpo la alberga [43].

Se define como todo aquello que puede considerarse que percibe y responde o actúa en un ambiente, con un cierto grado de independencia o autonomía. Ecológicamente, se asume que el agente tiene un objetivo, una razón de ser, configurándose en un **agente racional** o **agentes basados en utilidad**. Esto se materializa mediante su **función de desempeño**. Esta determina el éxito del agente, y a su vez puede medirlo de corto o a largo plazo, es decir teniendo alguna medida del horizonte temporal de ese éxito (i.e. decisiones estratégicas). El agente así, racionalmente, busca realizar acciones que maximicen su medida de rendimiento.

Por otro lado, es posible identificar a los **agentes omniscientes**. Conocen en todo momento el resultado real que producirán sus acciones.

Los agentes también pueden clasificarse en relación a si poseen o no un estado interno. Dependiendo si tiene o no estado interno, los que no lo mantienen se suelen llamar reactivos  $a = f(P)$  con  $P$  percepción y  $a$  acción. En cambio los que poseen estado interno, también llamados con modelo interno, se caracterizan por  $a = f(P, M)$ , donde  $M$  es el modelo interno, algo que el agente usa para

<sup>1</sup>Si algo es óptimo, no es posible que sea *más óptimo* pero ustedes entienden la idea.

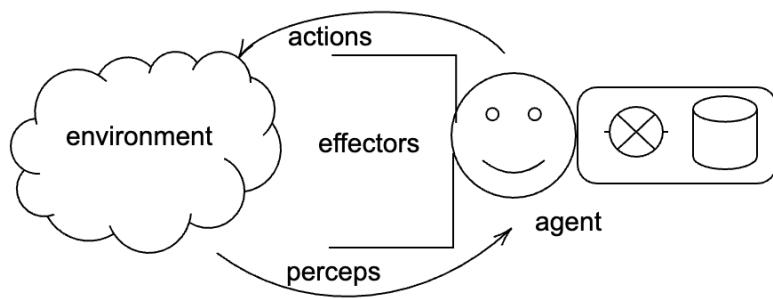


Figura 3.1: El agente y su ambiente

entender el entorno.

### 3.1.1 Ambientes

Ambiente se le llama a los componentes que engloban al agente [44]. Es lo que provee el origen de la percepción así como también el destinatario de las acciones del agente. Los ambientes pueden ser **discretos** o **continuos**, dependiendo si el número de estados y acciones es finito o es un continuo.

Pueden ser **determinísticos** o **estocásticos**, en relación a si es o no predecible con probabilidad uno el estado resultado de una acción ejercida sobre el ambiente partiendo de un estado anterior conocido. En tanto que en el caso estocástico, la transición depende de un proceso estocástico probabilístico (es decir no se sabe con precisión y se estima probabilísticamente). A su vez el ambiente puede ser **totalmente observable**, cuando es posible para el agente determinar cualquier característica del mismo, o **parcialmente observable**, cuando justamente esta situación no puede determinarse. Un ambiente parcialmente observable puede parecer estocástico para el agente. Esta idea puede extenderse al propio proceso de aprendizaje del agente. El ambiente es **conocido**, si el agente a priori conoce todo lo que puede conocer del ambiente, en tanto que es **desconocido** cuando requiere de un proceso de exploración para poder adquirir ese conocimiento.

El concepto de agente es también central en Teoría de Juegos [45], donde se modelan las interacciones entre varios agentes. De esta forma, cuando hay un único agente, el ambiente es **individual** mientras que cuando hay varios agentes simultáneos el ambiente es **multiagente**. En este caso los agentes pueden ser **colaborativos** o **adversariales**. Un ambiente **determinístico estratégico** puede presentarse cuando precisamente no conocemos las acciones de los otros agentes (pero sí como opera el ambiente).

Los agentes tienen una experiencia en el ambiente dada por la secuencia de acciones que ejecutan y los estados que van transitando. El escenario es **episódico** cuando es posible determinar secuencias finitas de acciones que tienen un estado de inicio y un estado final, y sobre todo cada episodio es totalmente independiente de los episodios anteriores y futuros. Cuando esta independencia no se presenta, se dice que el proceso es **secuencial**.

Los ambientes **estáticos** son aquellos donde el estado no cambia mientras el agente no ejecuta sus acciones, y es **dinámico** cuando esto sí ocurre. Por otro lado, si lo que cambia es la calificación, la evaluación de la función objetivo, el ambiente es **semidinámico**.

## 3.2 La importancia de la búsqueda

Dado un adecuado modelado, en un sistema digital con precisión finita, cualquier problema de inteligencia artificial, se podría resolver, optimizar en esa arquitectura, por fuerza bruta buscando todas las opciones posibles, y evaluar la función de costo, ejecutandolo sobre el sistema y verificar cuál es la mejor. Por ejemplo, en un juego de Ajedrez, o cualquier otro juego de mesa. El punto crítico, es que precisamente no se tiene infinito poder de cómputo posible, y queremos hacer una búsqueda que sea realizable computacionalmente.

## 3.3 Métodos de Búsqueda

La resolución de problemas basados en búsquedas arranca de la base del modelo del agente, donde hay diferentes componente que plantean un problema bien formado, donde existen:

- **Estados:** diferentes estados  $s$  que modelan las diferentes configuraciones en las que puede estar el problema.
- **Estado inicial:** configuración inicial  $s_0$  donde el problema no está resuelto.
- **Acciones:** acciones  $a$ .
- **Modelo de Transición T:** describe el efecto de aplicar la acción  $a$  estando en  $s$ , que deriva en un estado  $s'$ , llamado sucesor.
- **Función de costo:** Determina el costo  $g(T(s, a)) = g(s, a)$  para el modelo de transición T.
- **Condición de solución:** Determina si un estado  $s$  particular es o no solución del problema.

Utilizando los estados, las acciones y las transiciones, y sobre todo la evaluación que podemos hacer de la solución en cada estado, es posible plantear un árbol de búsqueda que partiendo de un estado inicial  $s_0$ , transiciona por los estados aplicando acciones y evaluando cada uno de los caminos para intentar encontrar el óptimo: aquel que tenga el menor costo acumulado. Cada nodo  $n_i$  del árbol corresponde a una secuencia específica de transiciones. Cada nodo entonces, puede tener un costo asociado  $G(n) = \sum g(s_i, a_i)$  obtenido como la sumatoria de los costos de todas las transiciones de forma que partiendo del estado inicial sea posible alcanzar el estado final  $s'_{i+1}$ . Este árbol puede verse como un grafo si se consideran más de un nodo raíz.

Esto puede verse en la figura 3.2. Cada transición de estados  $T(s, a) = s'$  tiene un costo asociado  $g(s, a) = w$ . Este costo se va acumulando a medida que se va transitando por el árbol de estados. Así por ejemplo en el nodo  $n_2$  es  $G(n_2) = 7$  y en en  $n_3$  es  $G(n_3) = 10$ .

El aspecto más complejo está justamente en encontrar una representación adecuada del problema, en base a definir una arquitectura de estados, que representen todas las opciones posibles, manteniendo utilidad en las acciones para que sean acotadas, buscando la posibilidad de encontrar optimizaciones computacionales en las simetrías.

La definición de las acciones pasa también por definir meta-acciones que puedan ser un aglomerado de otras acciones más pequeñas y que aporte información directa al objetivo del método de búsqueda (es decir, no es necesario mantener siempre una correspondencia entre las acciones establecidas en el modelo con acciones concretas del problema en cuestión).

### 3.3.1 Eficiencia

Al realizar una búsqueda se requiere tener una medida de la eficiencia en el procedimiento. Lo básico es pragmatismo puro, **encontrar una solución**. La segunda es en base a **optimizar el costo** de la solución, que puede ser en base a un costo máximo, mínimo o promedio. Se puede medir en base a la **cantidad de acciones a realizar**, o incluso en base al propio **costo computacional** en función del uso de memoria o de tiempo de procesamiento.

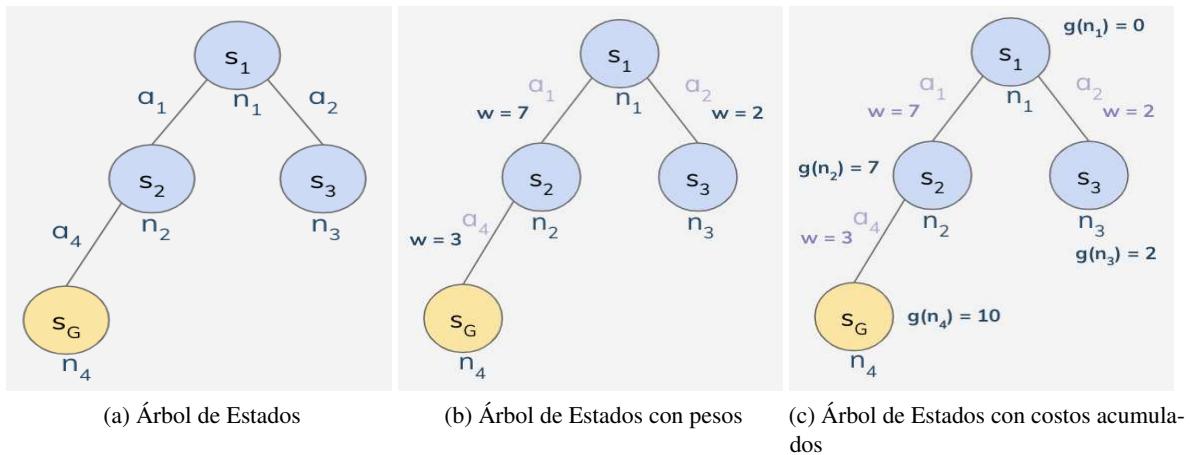


Figura 3.2: Modelo base de estados.

$\delta$	$a_1$	$a_2$	$a_3$	$a_4$
$s_1$	$s_2$	$s_3$	$\emptyset$	$\emptyset$
$s_2$	$s_1$	$\emptyset$	$\emptyset$	$s_G$
$s_3$	$\emptyset$	$\emptyset$	$\emptyset$	$s_1$
$s_G$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

(a)

Figura 3.3: Tabla de estados posibles. El estado  $s_G$  es el estado objetivo.

La búsqueda puede ser a su vez **forward** hacia adelante, arrancando desde el propio estado inicial del problema, o puede ser **backward** o hacia atrás donde la raíz es la propia solución (y de ahí hacia el o los estados iniciales).

### 3.3.2 Estrategias de Búsqueda

La búsqueda es **completa** cuando alcanza y encuentra la solución (de existir y ser alcanzable). Es **óptima** cuando encuentra la solución alcanzable que tiene menor costo según algún criterio.

### 3.3.3 Representación del Problema

En el árbol de estados  $Tr$ , dado un nodo  $n$ , los nodos hijos que se obtienen al expandir el nodo  $n$ , se representan como  $s(n_i) = \{n_{i+1}, n_{i+2}, \dots, n_{i+3}\}$  (sucesores). El conjunto formado por todos los nodos expandidos posibles es el conjunto frontera  $Fr$ .

Un estado es **expandido** cuando dentro del método de búsqueda se le han aplicado todas las acciones aplicables. Los estados expandidos conforman el grupo de los estados explorados  $Exp$ .

Un nodo no es un estado en sí, pero se encuentra siempre asociado a uno en particular. Por eso, el mismo estado puede estar vinculado a diferentes nodos. Siendo  $G(n)$  el costo acumulado del nodo (la suma del arco desde  $n_0$  hasta  $n$ ), podemos encontrar dos nodos  $n_1$  y  $n_2$  que pueden estar

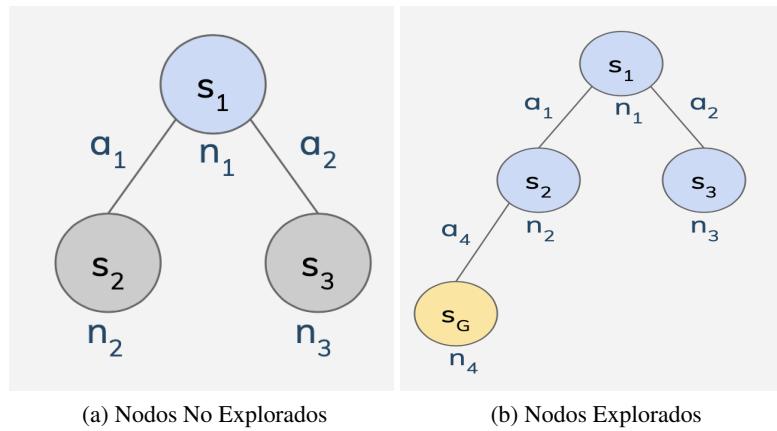


Figura 3.4: Árboles de Representaciones.

asociados al mismo estado, de forma que

$$G(n_1) \neq G(n_2).$$

Las diferentes variantes de los métodos, tanto desinformados como informados, exploran diferentes maneras de aplicar y evaluar el algoritmo 1. Esta variante presentada mantiene una lista de nodos expandidos para evitar ciclos en el grafo.

---

**Algoritmo 1** Algoritmo General de Búsqueda (GRAPH-SEARCH)

---

- 1: Árbol de búsqueda  $Tr$
  - 2: Conjunto frontera  $Fr$
  - 3: Conjunto de Nodos explorados  $Exp$
  - 4: Crear  $Tr, Fr, Exp$  vacíos.
  - 5: Insertar nodo inicial  $n_0 \rightarrow Tr, Fr$
  - 6: **while**  $Fr \neq \emptyset$  **do**
  - 7:   Extraer primer nodo de  $Fr \rightarrow n$
  - 8:   **if**  $n$  Goal **then**
  - 9:     Solución formada por el arco entre raíz  $n_0 \rightarrow n \in Tr$ . Termina
  - 10:   **end if**
  - 11:   Expandir el nodo  $n$  generando los sucesores  $\{n_{+1}\}$
  - 12:    $\{n_{+1}\} \rightarrow Fr$
  - 13:    $\{n_{+1}\} \rightarrow Tr$ , Como sucesores de  $n$
  - 14:    $n \rightarrow Exp$
  - 15:   Reordenar  $Fr$  según criterio (dependiente del método)
  - 16: **end while**
  - 17: **if** Solucion vacía **then**
  - 18:   No existe solución alcanzable
  - 19: **end if**
- 

### 3.4 Métodos de Búsqueda desinformada

Los métodos de búsqueda desinformados, sólo tienen la información que surge del propio modelo del problema, sin distinguir de los estados otra cosa que no sea si son o no **goal**, es decir

los estados objetivo que el propio proceso de búsqueda intenta encontrar. Son poco eficientes en problemas donde podrían hacerse estimaciones.

Los métodos más comunes de búsquedas desinformadas son:

- BFS
- DFS
- IDDFS
- Costo Uniforme

### 3.4.1 Breadth First Search

BFS expande los nodos de menor profundidad primero. Es decir selecciona el primer nodo a expandir y recursivamente va expandiendo hasta alcanzar la profundidad máxima. Para que tenga sentido, el chequeo de si un estado es o no final se realiza cuando el nodo es agregado a la frontera. Es completa si el factor de ramificación es finito, encontrando la solución con menor profundidad. Cuando todas las acciones tienen el mismo costo, la solución es óptima. La contracara de BFS es que tiene complejidad  $\mathcal{O}(b^d)$  siendo  $b$  la cantidad de nodos de un mismo nivel y  $d$  la profundidad. Esto no escala.

### 3.4.2 Uniform Cost Search

UCS, ó Búsqueda uniforme, expande el nodo con menor costo acumulado. Por supuesto, cuando el costo es uniforme, es exactamente igual a BFS. La búsqueda es óptima cuando se cumple que  $G(s(n)) \geq G(n) \forall n$ , o equivalente que  $g(s, a) \geq 0 \forall a, s$ .

### 3.4.3 Depth First Search

DFS expande el nodo de mayor profundidad. No es óptimo, pero suele tener una mucho menor complejidad espacial, sobre todo en su versión sin estados repetidos, ya que con estados repetidos puede quedar en bucles infinitos. Cuando existen muchas soluciones suele ser más rápido que BFS. Desde la implementación, BFS utiliza una cola FIFO para  $Fr$  en tanto que DFS utiliza LIFO.

Para evitar el problema de la búsqueda infinita de estados, se puede limitar la profundidad de la búsqueda con un hiperparámetro valor arbitrario que se denomina **Depth Limited Search**, generando una búsqueda que no es completa ni óptima. Para justamente no tener que arbitrariamente setear este parámetro, IDDFS **Iterative Deepening Depth First Search** es DLS donde la profundidad puede variar según cierto criterio. Se puede implementar manteniendo los nodos hoja que no fueron expandidos por cota de profundidad entre cada iteración. Con un valor de profundidad infinito es precisamente vanilla DFS.

### 3.4.4 Estados repetidos y Backtracking

La motivación detrás del uso de estados repetidos es no regresar a estados ancestro, ni crear rutas que tengan ciclos que rompen la estructura del árbol. Tampoco volver a expandir estados que ya fueron expandidos previamente. Es un compromiso entre el costo de almacenar y chequear estados.

**Backtracking** se refiere a poder generar sucesores y agregarlos a la frontera uno a la vez, y luego de verificar que el camino elegido no conlleva al óptimo, poder deshacer todos los cambios que se hayan hecho sobre las estructuras de soporte del algoritmo, y elegir otro nodo individual, y evaluarlo.

### 3.5 Métodos de Búsqueda Informados

Los métodos informados por el contrario, cuentan con más información para comparar entre diferentes estados. Con eso pueden hacer una estimación para calcular cuánto falta para resolver el problema, mediante el cálculo de una **heurística**. Con esa información también pueden calcular el costo del problema. Este modelo, más refinado, permite abordar justamente problemas que son intrínsecamente más complejos e implementar mecanismos de búsqueda que a su vez son más eficientes. Los dos principales exponentes son:

- Greedy
- A\*

Estos métodos se basan en la definición de una **heurística** (def 3.5.1) que es una estimación *heurística* del costo asociado a cada estado, que sea fácil de calcular<sup>2</sup>.

**Definition 3.5.1 — Heurística.** Función heurística  $h(s)$  representa el costo estimado de la ruta más barata desde el estado  $s$  hacia un estado meta.

Si  $s$  Goal  $\rightarrow h(s) = 0$

Si  $s$  no Goal  $g(s,a) > 0 \forall a : \text{Accion} \rightarrow h(s) > 0$

Si  $s$  no Goal  $g(s,a) \geq 0 \forall a : \text{Accion} \rightarrow h(s) \geq 0$

Se puede abusar la notación planteando

$$h(n) = h(s),$$

indicando que  $s$  es el estado representado en el nodo  $n$ . El valor de la heurística depende del estado.

**Proposición 3.5.1 — Heurística Admisible.** Una heurística es admisible si nunca sobrestima el costo real. Es perfecta si estima exactamente el costo real  $h^*(n)$ .

Las heurísticas se pueden encontrar dividiendo al problema en sub-problemas, relajando las propias limitaciones o las reglas del problema. También la heurística puede surgir como el máximo de otras métricas o pueden surgir de mezclar heurísticas, contemplando la valuación de cada una de ellas, sobre todo cuando atacan diferentes sub-problemas.

#### 3.5.1 Local Greedy Search

Que sea **Greedy** hace referencia a que el algoritmo busca la mejor opción que tiene a mano, sin ir más allá en relación a los pasos siguientes. Esta variante informada arranca desde el nodo raíz y se expande, y sobre los expandidos estima  $h$ . Luego toma el nodo con menor  $h$ ; este es el paso greedy. Si el conjunto recién expandido es vacío, necesita hacer **backtracking**, ir para atrás en el algoritmo y explorar otra opción.

Esta solución no es óptima ni completa para el caso que no evalúe repetición de estados. Como en todos los algoritmos informados, la clave suele estar en la selección de una muy buena función heurística, que va a determinar la complejidad temporal y espacial del algoritmo.

Si el algoritmo utilizado es 1 pero usando  $h(n)$  para establecer el ordenamiento de  $Fr$  en el paso 15, el algoritmo se denomina **Global Greedy Search**. En este caso, a pesar de tener un costo mayor que LGS, el backtracking es más efectivo al poder evaluar más opciones.

<sup>2</sup>De la física, existe un método similar denominado *Método de Fermi*, que es una técnica rápida para dado un problema poder entender cuál es el orden de magnitud de la respuesta buscada sobre todo cuando los datos no se tiene a mano o se requiere una respuesta rápida [46]

### 3.5.2 A\*

Es el algoritmo **estrella** de métodos de búsqueda y uno de los algoritmos más utilizados del mundo. Se basa en la función

$$f(n) = G(n) + h(n),$$

y ordena a los nodos en  $Fr$  en base a esta función en el paso 15, desempatando en base a  $h(n)$  cuando  $f(n)$  son iguales. Es idéntico a costo uniforme, sólo que usa  $f(n)$  para estimar el menor costo.

El algoritmo garantiza que si  $h(n)$  es admisible, entonces  $f(n)$  nunca sobreestima el costo real de la mejor solución que pasa por el nodo  $n$ . La búsqueda es completa si hay una ramificación finita y el costo es mayor que un  $\epsilon > 0$ . Por otro lado, es óptima cuando se cumple:

**Teorema 3.5.2** A\* es óptima (se encuentra el camino óptimo a la solución si existe) cuando:

- Cada nodo del grafo debe tener un número finito de sucesores.
- El costo de cada arco debe ser mayor que un  $\epsilon > 0$ .
- La heurística debe ser admisible  $h(n) \leq h^*(n)$

Se define

$$f^*(n) = G(n) + h^*(n)$$

Por supuesto requiere memoria y procesamiento para el cálculo de las heurísticas.

**Proposición 3.5.3 — Lema 1: Heurística Admisible.** En cualquier momento dado en la búsqueda de A\*, existe un nodo  $n \in Fr$  que cumple

1.  $n$  está en el camino óptimo al objetivo
2.  $f(n) \leq f^*(n)$

*Demostración.* Prueba por Inducción.

#### Caso Base

Para el nodo inicial  $n_0$ . Todos los caminos comienzan con  $n_0$  y por existir una solución,  $n_0$  está en el camino óptimo al objetivo.

$$f(n_0) = G(n_0) + h(n_0) = h(n_0) \leq h^*(n_0) = f^*(n_0) \quad (3.1)$$

#### Paso inductivo

La frontera en el paso  $k$  contenía a un nodo que estaba en el camino óptimo. Entonces se elige nodo  $n^e$  para expandir.

Si el nodo  $n^e$  no es un nodo del óptimo, la frontera va a seguir teniendo a dicho nodo, entonces se cumple (1) y  $n'$  es dicho nodo.

Si el nodo  $n^e$  es tal nodo, A\* lo expande, y alguno de sus hijos debería estar en el camino óptimo, por lo tanto se cumple (1) y  $n' = n^e$

$$f(n') = G(n') + h(n') \leq G(n') + h^*(n') = f^*(n') \quad (3.2)$$



**Teorema 3.5.4 — Terminación de A\***. Partiendo de que la ramificación es finita, y el costo de las reglas/acciones son siempre mayor o igual a un número mayor a 0. Entonces, si A\* puede no terminar, eventualmente llega a un punto donde

$$f(n) > f^*(n), \forall n \in Fr$$

**Definition 3.5.2 — Dominancia de Métodos de búsqueda.** Se dice que  $M_1$  domina a  $M_2$  si para cada nodo expandido por  $M_1$  también es expandido por  $M_2$ . Se dice que  $M_1$  domina estrictamente a  $M_2$  si

- $M_1$  domina a  $M_2$
- $M_2$  no domina a  $M_1$

### Eficiencia de heurísticas

Suponiendo  $h_1$  y  $h_2$  heurísticas admisibles, si  $h_2$  domina a  $h_1 \rightarrow A_2^*$  expandirá menos nodos que  $A_1^*$

Esto es porque existirán nodos que  $A_1^*$  expandió (y no llevan al camino óptimo) y  $A_2^*$  no, por lo que, para esos nodos:

$$G(n) + h_1(n) \leq f^*(n) \leq G(n) + h_2(n)$$

De esta forma, como regla general conviene tomar heurísticas admisibles que den el mayor valor posible.

### Combinación de heurísticas

Dado un conjunto de heurísticas admisibles  $h_1, \dots, h_m$  se puede definir como una nueva heurística  $h'$  a la combinación de ellas, definida de la siguiente forma:

$$h' = \max(h_1, \dots, h_m)$$

Esta nueva heurística tiene como propiedades:

1. Es admisible
2. Domina a todas las heurísticas que al conforman

### Consistencia de heurísticas

Si una heurística es consistente, cada vez que A\* expanda un nodo, habrá encontrado un camino óptimo al mismo.

Si una heurística es admisible, no necesariamente.

### Complejidad de A\*

A\* podría no terminar nunca.

Algunas de las situaciones donde puede suceder esto son:

- La ramificación es infinita para al menos un estado.
- Existen arcos con costos  $\leq 0$ .
- Los arcos tienen costos  $> 0$ , pero son asintóticamente decrecientes (Paradoja de Zenon)

### 3.5.3 Iterative Deepening A\*

Concepto inspirado en IDDFS, se realiza un corte iterativo con un límite. También aquí se utiliza DFS. Esta búsqueda es completa y óptima bajo las mismas condiciones que A\*. Requiere menos memoria que A\*, pero puede expandir más nodos e incluso expandir muchas veces el mismo nodo (mismo problema que IDDFS).

El límite es un threshold que se impone sobre  $f(n)$ .

1. Inicialmente se toma  $Lim = f(n_0)$
2. Mientras que no se encuentre solución, se realiza DFS hasta  $Lim$ .
3. Si no se encontró una solución, se toma  $n'$  el nodo de frontera con menor valor de  $f$ , y se toma este valor como el nuevo límite.

### 3.5.4 Heuristic Path Planning

Best-First con

$$f(n) = (1 - \omega) * G(n) + \omega * h(n)$$

$$f(n) = \begin{cases} \omega = 0 & \text{Uniform Cost Search} \\ \omega = \frac{1}{2} & \text{A* Search} \\ \omega = 1 & \text{Global Greedy Search} \end{cases}$$

### 3.5.5 Ejemplo de juego: Sokoban

El Sokoban [47] es un juego clásico, caracterizado por un trabajador de un depósito que tiene en dos dimensiones que llevar un conjunto de cajas a sus destinos disponibles. *Soko* significa depósito y *ban* es un sufijo que representa el encargado o el trabajador. Fue diseñado en 1982 por Hiroyuki Imabayashi para las computadoras Spectrum y Commodore. El juego consiste en diferentes tableros con espacios disponibles donde el trabajador se puede mover, paredes infranqueables, las ubicaciones donde están las cajas y los destinos donde hay que depositarlas. Para mover cada caja, el trabajador se coloca detrás de la misma y la puede mover hacia uno de los cuatro destinos cardinales. Este juego es excelente como plataforma para implementar y probar algoritmos de búsqueda. La razón de esto es porque el problema es NP-Hard y a su vez PSPACE-complete. Tableros simples requieren una gran cantidad de capacidad de procesamiento y de memoria para resolverse. Además tiene situaciones de deadlocks que no pueden resolverse, configuraciones de los tableros donde no hay forma de mover alguna de las cajas o de llevarla a alguno de los destinos.

## 3.6 Algoritmos de Mejoramiento Iterativo

Los algoritmos de búsqueda con mejoramiento iterativo, se basan en partir de una solución y mejorarla, optimizar el funcional, mediante perturbaciones a esa solución.

**Definition 3.6.1 — Mejoramiento Iterativo.** Sea  $f(S)$  función de optimización que evalúa qué tan buena es una solución  $S$  particular. Los algoritmos iterativos se basan en encontrar  $S^*$  tal que

$$S^* / \forall S \quad f(S^*) \geq f(S)$$

Las diferentes opciones giran alrededor de la idea de variar de manera azarosa la solución encontrada, moviéndose por las posibles direcciones de crecimiento o decrecimiento (ver Optimizaciones en el capítulo 5).

### Travelling Salesman Problem

El problema del viajante de comercio (TSA) es un clásico NP-Hard. Es un problema de recorrido de grafos donde hay  $n$  ciudades y la distancia entre cada una de ellas. El planteo del problema pasa por encontrar la ruta más corta, visitando cada ciudad exactamente una vez y al final regresar a la ciudad de la que se parte en el principio.

La entrada del problema es el número de ciudades  $n$  y los  $n$  pares ordenados con las coordenadas de cada ciudad  $(x, y)$ . El output del problema tiene que ser la lista de ciudades según su índice

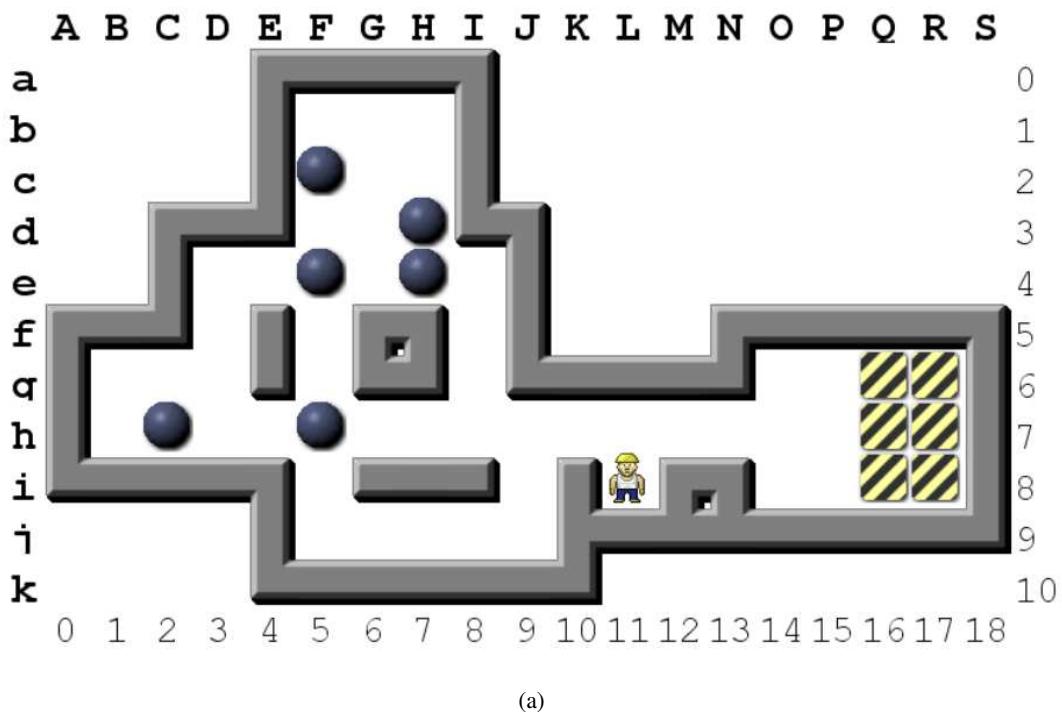


Figura 3.5: Imagen de un típico tablero de Sokoban, tomada de [47].

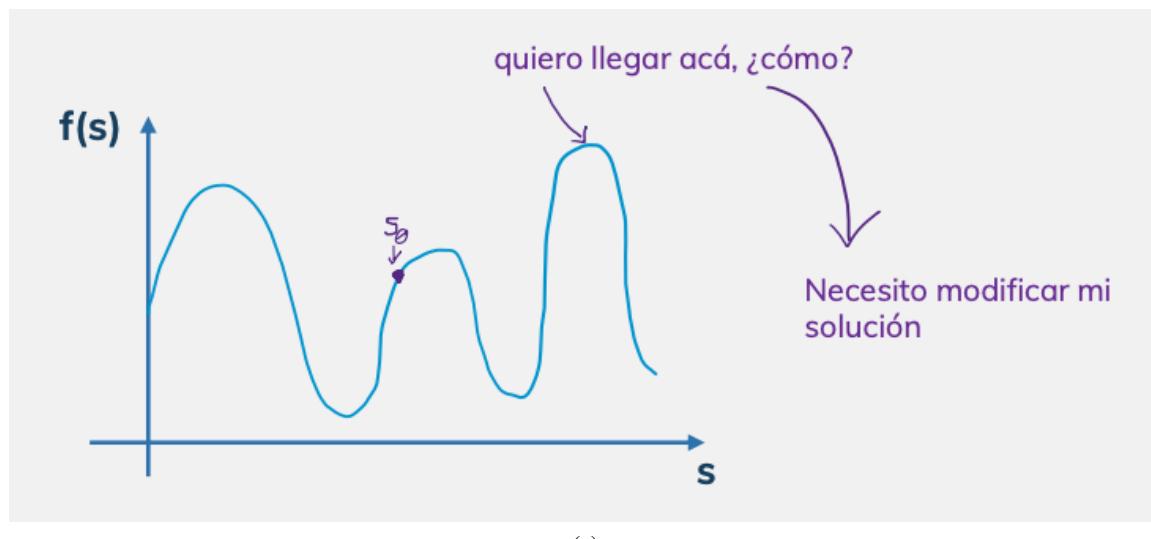


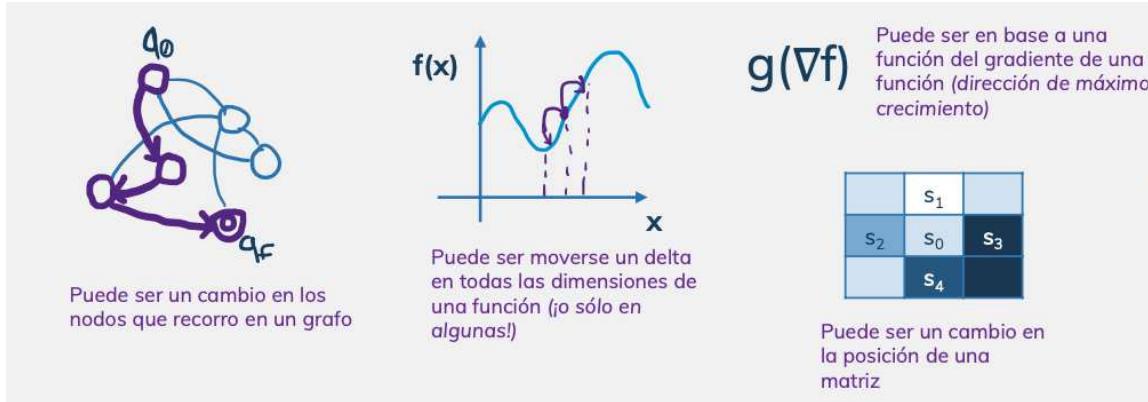
Figura 3.6: Función de optimización y mejoramiento iterativo.

que hay que recorrer o eventualmente los waypoints que corresponden con la sucesión de puntos que va a reflejar por donde tiene que ir el salesman (tocando y pasando por todas las ciudades y completando el circuito).

#### Solución por métodos iterativos

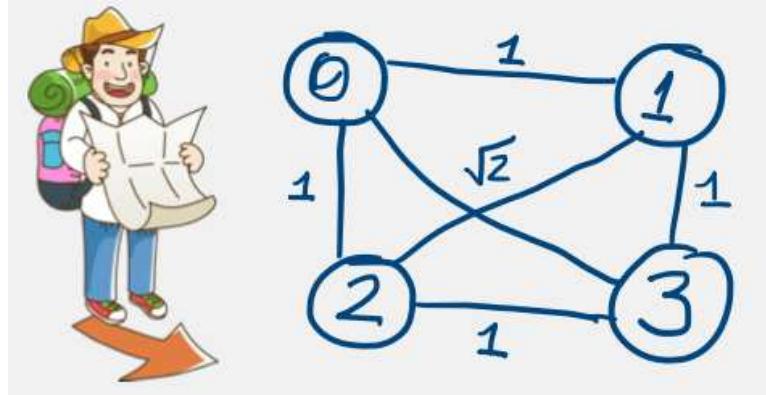
Consideremos el problema de TSA de la figura 3.8, con las distancias dados por los rótulos en las aristas.

Para resolver este problema iterativamente, se puede hacer:



(a)

Figura 3.7: Alterantivas para implementar el mejoramiento iterativo.



(a)

Figura 3.8: Problema básico de Travel Salesman Problem

1. Tomo una solución al azar, por ejemplo:  $sol = \{0 - 3 - 1 - 2(-0)\}$
2. Evalúa la solución en la función objetivo:  $f(sol) = 4.83$ .
3. Se realizan cambios azarosos en la solución  $sol$ . Por ejemplo, se elige una ciudad y se permuta su ubicación,  $sol_{a1} = \{3 - 0 - 1 - 2\}$ ,  $sol_{a2} = \{0 - 1 - 3 - 2\}$ ,  $sol_{a3} = \{0 - 1 - 2 - 3\}$
4. Se selecciona una en base a su valuación:  $f(sol_{a2}) = 4$

### 3.6.1 Hill Climbing

En este caso las variaciones azarosas se hacen en la dirección de crecimiento de la función, marcada por el gradiente, de forma que se suba a la cumbre (de ahí el nombre). Es un algoritmo de heurística local, donde a cada paso se genera una nueva frontera  $Fr$  (que no es la misma frontera de los métodos de búsquedas que se usa para identificar los nodos a expandir). El problema de este algoritmo es que suele estancarse muy fácilmente en máximos locales.

**Algoritmo 2** Algoritmo Hill Climbing

- 
- 1: Conjunto frontera  $Fr$
  - 2: **while**  $Fr \neq \emptyset$  **do**
  - 3:   Moverse un paso arbitrario en ambas direcciones de cada una de las dimensiones.
  - 4:   Tomar cada punto solución  $s_i \rightarrow Fr$  (agregarlo en la frontera).
  - 5:   Evaluar los puntos de la frontera  $f(s_i) \forall s_i \in Fr$
  - 6:   Seleccionar el valor máximo de la frontera como nueva solución  $\text{argmax}_{s_i} f(s_i)$
  - 7: **end while**
- 

**Simple Hill Climbing:** Elige la primer solución de la frontera con mejor valuación que el estado actual.

**Steepest Hill Climbing:** Siempre elige la solución de la frontera con mejor valuación que el estado actual.

**Stochastic Hill Climbing:** Elige al azar una solución de la frontera y, en base a que tan buena es, decide si moverse o elegir otra.

### 3.6.2 Simulated Annealing

El algoritmo de recocido simulado, un clásico de la física computacional. Este algoritmo surge de modelar el comportamiento de ferrometales donde se los somete a un campo magnético, inicialmente a mucha temperatura, y donde la temperatura se va reduciendo gradualmente, lo que da origen a que el material se vuelve ferromagnético, ya que el movimiento térmico inicial le da espacio a las moléculas de poder acomodarse, y la tendencia en la que se acomodan a medida que se reduce la temperatura es la influenciada por el campo magnético [48].

El algoritmo es parecido a Hill Climbing, pero involucra un parámetro que regula la estocasticidad, el balance entre exploration/explotation, y que en este algoritmo representa la **temperatura**. A medida que van pasando las iteraciones, esta temperatura se va disminuyendo progresivamente según un perfil de progresión. El algoritmo procede similar a Hill Climbing, estableciendo un conjunto Frontera  $Fr$ , pero ignora la valuación de cada solución allí depositada con una probabilidad relativa a la temperatura (y opcionalmente, proporcional a su valuación). Es decir el algoritmo selecciona la mejor opción disponible y se mueve a esa solución según el propio perfil de la exponencial (eligiendo una opción donde  $d < 0$ , sin mejora, más al principio, y quedándose con las opciones mejores a medida que disminuye la temperatura). La temperatura puede reducirse iteración a iteración o de a bloques dependiendo un valor fijo o alguna condición.

**Algoritmo 3** Simulated Annealing

---

```

1: Tomar  $s = s_0 ; t = t_0$ 
2: while  $P$  is False do
3:   while Repetir  $N_{rep}$  do
4:     Seleccionar  $s' \in E(s)$  (entorno de  $s$ )
5:      $d \leftarrow f(s') - f(s)$ 
6:     if  $d > 0$  then
7:       Tomo  $s'$  (es una mejor solución)
8:     else
9:        $u \leftarrow rand(0, 1)$ 
10:      if  $u < \exp(\frac{-d}{t})$  then
11:         $s = s'$ 
12:      end if
13:    end if
14:  end while
15:   $t = \alpha(t)$  Reducción de la temperatura
16: end while

```

---

**3.6.3 Beam Search**

Similar a Hill Climbing pero comienza con  $k$  nodos iniciales<sup>3</sup>. En cada paso, se generan los sucesores de los  $k$  nodos, que pasarán a conformar la frontera. De todos los sucesores, se toman los mejores  $k$  nodos de la frontera y se vuelve a repetir. Es diferente a  $k$  hill climings en paralelo, ya que compara todos los mejores en todos los caminos entre sí, vinculando la información entre todos, y por eso puede eliminar rápidamente los nodos iniciales malos para tomar otros caminos mejores. Puede converger mucho entre los  $k$  nodos, generando tan poca diversidad que se elabora un hill climbing mucho más costoso.

**Stochastic Beam Search**

Similar a Beam Search, pero elige los  $k$  nodos con una probabilidad proporcional a su valuación. De esta forma, los nodos **malos** también pueden llegar a elegirse (con menor probabilidad). Similar al proceso de selección natural (ver capítulo 4).

---

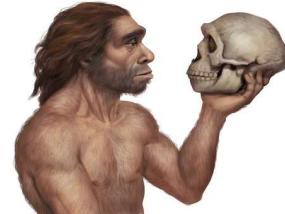
<sup>3</sup>El valor de  $k$  se conoce como el ancho del rayo o el beam width



## 4. Algoritmos Genéticos

La naturaleza biótica juega un juego de optimización plasmado en la evolución y la selección natural. La lucha por la supervivencia y la perpetuidad imponen una presión sobre los seres vivos para que solo aquellos que logran **encontrar** la correcta combinación de genes para cada ambiente específico, sobreviven y perpetúan esa combinación.

Esta idea, fue capturada por varios investigadores que la plasmaron en un algoritmo de optimización, en realidad una suite de **Algoritmos Genéticos** que ofrecen un marco muy potente para orientar una búsqueda [49].



### 4.1 Definiciones Evolutivas

La propuesta ofrecida en este escrito es la basada en Holland [50]. Es un algoritmo probabilístico de optimización/adaptación estructurado según un modelo simplificado de la mecánica de la selección natural y la teoría de la evolución. Tiene una enorme ventaja de que es altamente paralelizable<sup>1</sup> y tiene múltiples hiperparámetros y variables.

La idea general de los algoritmos genéticos (GA) consiste en generar una población de individuos, donde los parámetros libres a optimizar se codifican en una especie de gen. Así cada individuo representa una combinación completa de todos los parámetros. A su vez, esos parámetros son justamente las variables de una función de optimización, que en GA se denomina *fitness*, y que generan un resultado numérico. Así la maximización/minimización de esa función de costo depende de la combinación de valores de los parámetros plasmada en el código genético de cada individuo. Mediante una simulación de la evolución de la población de individuos, el algoritmo intenta encontrar aquellos que precisamente alcancen el óptimo a buscar.

Los componentes básicos son:

Figura 4.1: Los neanderthales y nosotros, una mezcla complicada.

<sup>1</sup>¿Por qué?

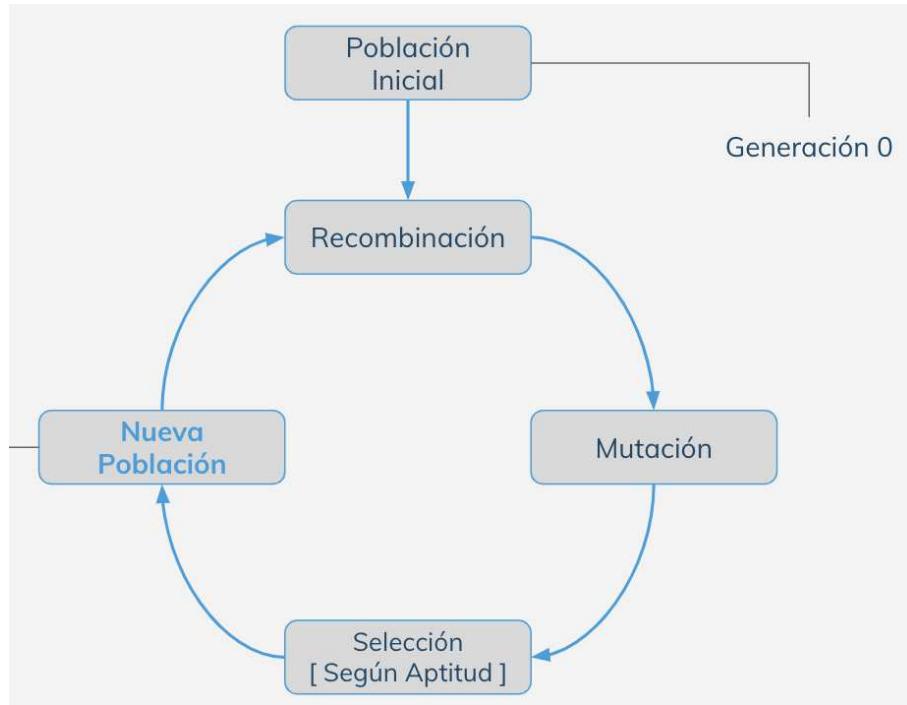


Figura 4.2: Esquema de bloques del patrón básico de un algoritmo genético

- Genotipo: estructura/arquitectura
- Población inicial
- Función de adaptabilidad = fitness = aptitud
- Método de selección de padres
- Método de cruce
- Método de mutación
- Método de selección de nueva generación
- Condición de corte.

**Definition 4.1.1 — Genotipo.** Establece la forma particular del gen, la manera en la que se codificara a nivel de bits, bytes, estructura de datos.

**Definition 4.1.2 — Locus.** Es la posición de un gen particular dentro del propio genoma.

**Definition 4.1.3 — Cromosoma.** El cromosoma es el conjunto completo de los genes. Por ejemplo, según el ejemplo de la tabla 4.1 sería [1][0][1][0][1].

Amarillo Patito	Triangulo
[1][0]	[1][0][1]

Cuadro 4.1: Table caption

**Definition 4.1.4 — Fenotipo.** Son las características observables afectadas por algún gen.

**Definition 4.1.5 — Alelos.** Las variantes posibles de un gen, representadas por la codificación que se establezca para el gen. A la variedad de alelos representados en la población se le llama diversidad.

**Definition 4.1.6 — Generación 0.** La Generación 0 es la población inicial de individuos de tamaño  $N$ .

**Definition 4.1.7 — Fitness.** Es la función de costo  $f(i)$  de la optimización que el algoritmo intenta resolver. Es evaluado para cada individuo  $i$ .

**Definition 4.1.8 — Fitness Relativo.** Normalizado al conjunto de individuos

$$p(i) = \frac{f(i)}{\sum_{j=1}^N f(j)} \quad (4.1)$$

### Métodos de Selección

Los diferentes métodos de selección determinan alternativas para seleccionar los individuos de una generación a la siguiente.

#### ⊕ Muestreo Directo: Elite

Elige un conjunto  $K$ , una elite, del conjunto de individuos. Para ello se los ordena según el fitness y se elige cada uno  $n(i)$  veces según la 4.2.

$$n(i) = \lceil \frac{(K - i)}{N} \rceil \quad (4.2)$$

#### ⊕ Estocástico: Ruleta

De las aptitudes relativas  $p(j)$  se derivan las aptitudes acumuladas para cada individuo.

$$q(i) = \sum_{j=0}^i p(j) \quad (4.3)$$

Luego se generan  $K$  números aleatorios de una  $r_j \leftarrow U[0, 1]$  que cumplan con  $q_{i-1} < r_j < q_i$ . Es decir se distribuye el espacio muestral entre todo los individuo según su aptitud relativa (a mayor aptitud mayor oportunidad de salir seleccionado en la ruleta).

#### ⊕ Universal

Similar a Ruleta pero cambiando la ecuación de  $r_j$  en base a  $r \leftarrow U[0, 1]$ .

$$r_j = \frac{r + j}{K}, j \in [0, (K - 1)] \quad (4.4)$$

#### ⊕ Ranking

Selección por Ranking deriva una función alternativa de pseudo-aptitud  $f'(i)$  que se calcula en base a un ranking. El ranking  $rank(i) \in [1, N]$  es una permutación de asignación a los  $N$  individuos dada por la función de aptitud real  $p(i)$ . Con esto se calcula:

$$f'(i) = \frac{N - rank(i)}{N} \quad (4.5)$$

A partir de este valor, se realiza el cálculo según ruleta utilizando la pseudo-aptitud relativa.

#### ⊕ Boltzmann

Para este método también se define una función de pseudo-aptitud con la siguiente función:

$$f'(i, g, T) = \frac{e^{\frac{f(i)}{T}}}{\langle e^{\frac{f(i)}{T}} \rangle_g} \quad (4.6)$$

donde  $i$  corresponde al individuo,  $T$  corresponde a la Temperatura del sistema,  $f(i)$  es la función de fitness y  $\langle \rangle_g$  es el cálculo del promedio del fitness poblacional. Luego se utiliza esta función de pseudo-aptitud como la aptitud aplicando la selección por el método de Ruleta.

Los métodos basados en temperatura al igual que como ocurre con los métodos de mejoramiento iterativo, aplican una función de decrecimiento de temperatura exponencial que tiene el efecto de balancear exploration/explotation. Esto favorece la búsqueda de variantes al principio del algoritmo, y la identificación de los más eficientes a medida que el algoritmo avanza.

#### ⊕ Torneos Probabilístico y Determinísticos

Los métodos de selección basada en torneos se

basan en justamente forzar una competencia, donde se selecciona un subconjunto de los individuos menor a al población total. La ventaja que tiene es que tiene una mayor velocidad de ejecución, ya que no se procesa toda la población, ni requiere variables intermedias, utilizando la función de fitness de manera directa. Además de una facilidad en la propia implementación. La desventaja por otro lado es que imprimen una presión de selección muy alta, con lo que esto conlleva.

Los métodos de selección determinísticos se basan en:

- De la población original de tamaño  $N$ , se eligen  $M$  individuos al azar.
- De los  $M$  individuos, se elige el mejor.
- Se repite el proceso hasta conseguir los  $K$  individuos que se precisan.

Por otro lado, la variante de torneos probabilística, introduce un componente extra de estocasticidad para la selección de los individuos:

- Se elige un Threshold de  $[0.5, 1]$ .
- De la población original de tamaño  $N$ , se eligen solo 2 individuos al azar.
- Se toma un valor  $r$  al azar uniformemente entre  $r \leftarrow U[0, 1]$ .
- Si  $r < Threshold$  se selecciona el más apto.
- Caso contrario, se selecciona el **menos** apto.
- Se repite el proceso hasta conseguir los  $K$  individuos que se precisan.

Los torneos probabilísticos dan más oportunidades a aquellos individuos para que mantengan variabilidad en la población genética.

### Convergencia Prematura

La convergencia prematura ocurre cuando la población converge a un subconjunto con muy poca variedad y sin haber llegado a una aptitud aceptable. Es decir, se reduce la variabilidad necesaria y suficiente para mantener la exploración en el conjunto de posibilidades.

Esto provoca que el método de optimización dado por el algoritmo genético arroja un subóptimo, a menos que dicho punto de convergencia sea el máximo global o que la población escape de dicho máximo local por mutación.

Las causas principales son una presión de selección **muy alta**, una probabilidad de mutación **muy baja** o un tamaño de la población muy escaso<sup>2</sup>.

## 4.2 Crossover

Una vez que se seleccionan los individuos se los aparea y se realiza la recombinación de genes. Se suele plantear un parámetro que regula la probabilidad de recombinar,  $P_c$  que define si hay o no recombinación en la generación de los hijos. Para el caso de que no haya sólo se deja el impacto de la mutación y los genes de padres a hijos simplemente se copian.

### Codificación del gen

Parecería natural utilizar en la codificación de un gen un bit por gen, pero esta idea poética, suele no ser una buena idea.

Existen las diferentes alternativas para la recombinación.

#### 4.2.1 Cruce de un punto

Directo, se elige un locus al azar y se intercambian los alelos a partir de ese locus. Es decir

$$P = [0, S]$$

con  $S$  siendo la cantidad de genes.

#### 4.2.2 Cruce de dos puntos

En este caso se eligen dos loci al azar y se intercambian los alelos entre ellos.

$$P_1 = [0, S]$$

$$P_2 = [0, S]$$

con  $P_1 \leq P_2$ .

#### 4.2.3 Cruce anular

Primero se considera al gen como un anillo, donde el último conecta con el primero circularmente. Luego se elige un locus al azar  $P$  y una longitud  $L$ . Finalmente se intercambia el segmento de longitud  $L$  a partir de  $P$  teniendo en cuenta la estructura anular. En este caso  $P = [0, S - 1]$  con  $L = [0, \lceil \frac{S}{2} \rceil]$ .

#### 4.2.4 Cruce uniforme

Se produce un intercambio de alelos en cada gen con probabilidad  $P$ , que normalmente es 0.5.

<sup>2</sup>Se cree que hace 900000 años el homo sapiens sapiens casi se extinguía generando una convergencia prematura, donde sólo 1300 individuos humanos vivos existían en la Tierra en ese momento, dejando una huella en nuestros genes de esa enorme reducción en la diversidad genética[51].

### 4.3 Mutación

La mutación se define como una variación en la información genética que se almacena y codifica en el cromosoma. Para que la mutación sea más eficiente, una correcta arquitectura y separación de genes es necesaria. El objetivo algorítmico de la mutación es enriquecer la diversidad genética (explorar el espacio de soluciones) y evitar los máximos locales (y escaparlos).

Las variantes son:

- **Mutación de gen:** se altera un solo gen con una probabilidad  $P_m$ .
- **Mutación multigen limitada:** se selecciona una cantidad  $[1, M]$  al azar de genes a mutar, con probabilidad  $P_m$ .
- **Mutación multigen uniforme:** cada gen tiene una probabilidad  $P_m$  de ser mutado.
- **Mutación completa:** con una probabilidad  $P_m$  se mutan todos los genes del individuo, acorde a la función de mutación definida para cada gen.

La mutación o cambio del gen puede no sólo implicar cambiar un valor (1 por 0) sino que dependiendo de la codificación requiere una definición precisa. Por ejemplo, puede implicar aplicar un delta al gen, en algún sentido y con alguna distribución.

#### 4.3.1 Generando generaciones

De una perspectiva pragmática, existen diferentes alternativas también para construir una nueva generación.

##### Fill-All

Generar  $K$  hijos de  $K$  padres. En este caso la nueva generación se formará seleccionando  $N$  individuos del conjunto  $[N + K]$  siendo  $N$  los individuos de la generación actual y  $K$  los hijos.

##### Fill-Parent

Generar  $K$  hijos de  $K$  padres.

- Si  $K > N$  la nueva generación se genera seleccionando  $N$  de los  $K$  hijos únicamente.
- Si  $K \leq N$  la nueva generación se conformará por los  $K$  hijos generados +  $(N - K)$  individuos seleccionados de la generación actual.

##### Brecha generacional

Determina la cantidad de individuos que sobreviven de una generación a la otra. Esto se captura en un indicador  $G \in [0, 1]$ .

- $G = 1$ : toda la población es reemplazada.
- $G = 0$ : ningún individuo es reemplazado.
- $G$  determina la suma convexa entre mantenimiento y reemplazo, con  $(1 - G) * N$  individuos de la generación actual y  $G * N$  los nuevos individuos de reemplazo.

##### Criterios de Corte

El último punto a clarificar es la identificación de los posibles criterios de corte: ¿cuándo parar?.

- Tiempo
- Cantidad de generaciones
- Solución aceptable (cota sobre la función de fitness).

- Estructura: cuando una parte relevante de la población no cambia por una cantidad de generaciones (estancamiento poblacional).
- Contenido: cuando el mejor fitness no cambia por una cantidad de generaciones (se alcanza los valores óptimos en la función).

### Esquemas

Los esquemas sirven para agrupar diferentes genes utilizando wildcards. Por ejemplo, el esquema  $*01001 *01$  representa  $001001001, 001001101, 101001001, 101001101$ .

**Teorema 4.3.1 — Teorema de los esquemas.** Aquellos esquemas con un fitness medio superior a la media de la población, de longitud pequeña y con un orden bajo aumentarán su presencia de manera exponencial en las sucesivas generaciones.

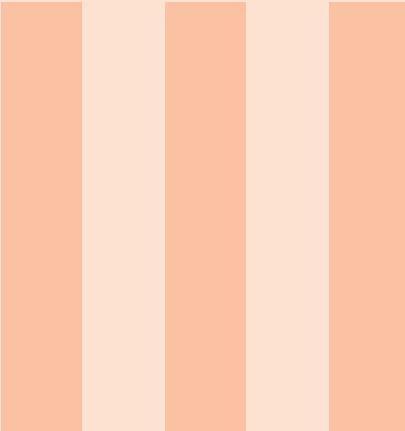
El orden del esquema determina la cantidad de genes con información que un preserva y se calcula como

$$o(S) = Len - wildcards$$

siendo  $Len$  la longitud del gen, y  $wildcards$  representa la cantidad de wildcards del esquema. Por ejemplo,  $o(*01*) = 2$ ,  $o(**0*1***1) = 3$ . Mientras más alto el orden del esquema, mayor es la probabilidad de perderlo en una mutación.

El uso de esquemas permite componer **bloques constructivos**. Esquemas cortos, de bajo orden pero con un fitness alto pueden ser elegidos, recombinados y re-elegidos nuevamente para ir formando cromosomas con mayor fitness. Esto permite reducir la complejidad del problema, dividiéndolo en subproblemas: se optimizan los esquemas, se encuentran nuevas esquemas o eventualmente se agrupan.





# Perceptrón

<b>5</b>	<b>Optimización Matemática .....</b>	<b>53</b>
5.1	La latita de Coca	
5.2	El problema de optimización	
5.3	El método simplex	
5.4	Optimización no lineal sin restricciones	
<b>6</b>	<b>El modelo de Neurona .....</b>	<b>63</b>
6.1	La bioinspiración	
6.2	Separabilidad lineal	
<b>7</b>	<b>La era del Conecciónismo .....</b>	<b>71</b>
7.1	Estructura Jerárquica Aglomerativa	
7.2	Arquitectura	
7.3	Optimización en Redes Neuronales	
<b>8</b>	<b>Métricas de Evaluación .....</b>	<b>83</b>
8.1	Identificar el patrón de manera correcta	
8.2	Métricas estándar	





El área de optimización matemática [52] puede considerarse "media matemática"<sup>1</sup>. Surge a partir de los primeros trabajos de Cauchy en 1847 y la propuesta del método del gradiente descendente. Luego, en la era de la posguerra tiene un auge importante impulsado por el desarrollo cuantitativo de la industrialización en masa, con los algoritmos de programación lineal, y la variante estocástica del algoritmo de gradiente descendente. En los 80's surge la teoría convexa y el marco general de optimización. Finalmente, en la primera década del siglo XXI, las ideas de optimización comienzan a aplicarse masivamente a problemas de aprendizaje automático. A partir del 2012 comienza un nuevo empuje que encuentra en las redes neuronales una aplicación directa. Muchos profesores y estudiantes dedicaron su vida a esta temática sacrificando su felicidad con horas de estudio en un salón oscuro y polvoriento. Ahora nos toca a nosotros [53].

## 5.1 La latita de Coca

Imaginarse una lata de coca cola. Una característica interesante que puede buscarse es encontrar cuales tienen que ser las dimensiones de una lata cilíndrica que contenga un litro, para minimizar su superficie y así la cantidad de aluminio que hay que utilizar.

Este es un típico problema que se estudia en cursos de Cálculo o Análisis Matemático alrededor del mundo y se puede resolver exactamente como un problema de optimización. La idea es partir de la base de definir el problema según la geometría de la lata:

$$V = \pi r^2 h \quad (5.1)$$

donde  $r$  es el radio del cilindro y  $h$  su altura. Luego la superficie de la lata, la que usará su aluminio, suponiendo un grosor despreciable viene dada por



Figura 5.1: De nada Coca Cola Corporation por resolverte la vida.

<sup>1</sup>Se puede demostrar que toda area de matemática es media matemática, pero no tenemos lugar acá para hacerlo.

$$S = 2 \pi r^2 + 2 \pi r h \quad (5.2)$$

De aquí surge el planteo como problema de optimización

$$S(r, h) = 2 \pi r^2 + 2 \pi r h \quad g(r, h) = \pi r^2 h = 1000 \text{ cm}^3 \quad (5.3)$$

Es decir se busca optimizar  $S(r, h)$  (encontrar los  $r$  y  $h$  que hacen que sea más chiquito) sujeto a la restricción dada en  $g(r, h)$  a un volumen fijo. La solución pasa por despejar primero  $h$  de la segunda ecuación, reemplazarla en la primera, y resolver directamente utilizando la propiedad de que la derivada de un función es cero en sus extremos.

$$g(r, h) = \pi r^2 h = 1000 \text{ cm}^3$$

$$h = \frac{1000 \text{ cm}^3}{\pi r^2}$$

Reemplazando en 5.2

$$\begin{aligned} S(r) &= 2 \pi r^2 + 2 \pi r \left( \frac{1000 \text{ cm}^3}{\pi r^2} \right) \\ S(r) &= 2 \pi r^2 + \frac{2000 \text{ cm}^3}{r} \end{aligned}$$

El punto extremo ocurre cuando la derivada de  $S(r)$  respecto a  $r$  es cero. Esto es

$$\begin{aligned} S'(r) &= 4 \pi r - \frac{2000 \text{ cm}^3}{r^2} = 0 \\ 4 \pi r &= \frac{2000 \text{ cm}^3}{r^2} \\ 4 \pi r^3 &= 2000 \text{ cm}^3 \\ r^3 &= \frac{500 \text{ cm}^3}{\pi} \\ r &= \sqrt[3]{\frac{500 \text{ cm}^3}{\pi}} \end{aligned}$$

Como el punto que se desea encontrar es el mínimo, esto ocurre cuando la derivada pasa de negativa a positiva, y eso es justamente en la solución positiva de  $r = 5.41 \text{ cm}$ , lo que provoca un valor de  $h = 58 \text{ cm}$  para la altura<sup>2</sup>. Esto es un típico problema de optimización no lineal con restricciones y como se ve el cálculo de la derivada tiene un valor crucial para su solución.

## 5.2 El problema de optimización

Optimización puede servir para minimizar los costos de producción de una empresa, resolver problemas de dinámica de transporte, aplicarse en teoría de juegos, hallar los parámetros óptimos de una función de distribución, etc. Existen numerosos algoritmos y áreas de optimización, entre las cuales podemos encontrar:

---

<sup>2</sup>Fíjateos que es una lata de medio metro, pero de un litro!

- Optimización Discreta: Programación entera y combinatoria
- Optimización Discreta: Ecuación diofántica
- Optimización Lineal: Programación Lineal
- Optimización No lineal sin restricciones convexa: garantía de extremos globales
- Optimización no convexa: métodos de primer orden: GD, SGD, Adam
- Optimización no convexa: métodos de segundo orden: Newton
- Optimización no convexa: cero orden: Bayesiana, Powel, Sim Optimistic Optimization

Para bosquejar el proceso de optimización se presenta el método Simplex y luego nos vamos a concentrar específicamente en la optimización no lineal sin restricciones, dada su relevancia para el entrenamiento de las redes neuronales.

**Teorema 5.2.1 — Mínimo de función monotónica.** El problema básico de optimización puede resumirse en la ecuación siguiente

$$\min_{x \in \mathbb{R}^n} f(x), \quad f : \mathbb{R}^n \rightarrow \mathbb{R}, \quad \vec{x} = [x_1, x_2, \dots, x_n] \quad (5.4)$$

$$g_i(\vec{x}) \leq 0 \quad (5.5)$$

$$h_j(\vec{x}) = 0 \quad (5.6)$$

donde  $\vec{x}$  son las variables de diseño,  $f(\vec{x})$  es la función objetivo/costo, en tanto que  $g_i$  y  $h_j$  son restricciones. El objetivo es encontrar  $\vec{x}^*$  que resuelve el problema para  $f(\vec{x}^*)$ .

### 5.3 El método simplex

Este es un método tradicional de programación lineal, muy común en el área de Investigación Operativa (1.2.11). Fue desarrollado primero durante la segunda guerra mundial por George Dantzig, y fue muy importante para los aliados para resolver problemas de logística y de producción. Luego fue reinventado independientemente en la Rusia soviética por Leonid Khachian, creyendo que era revolucionario pero que al final no fue otra cosa que el mismo método. Un aspecto interesante es que muestra como el proceso de optimización lleva en sí mismo una búsqueda de una solución, y esto se ve de manera directa en este método donde lo que se hace es identificar la región de validez de la solución, las intersecciones, y luego se prueba una por una cada intersección, eligiendo la que impactará mejor en la función de costo, hasta que se encuentra la óptima (o se determina que no es posible resolver) [54, 55].

Este método intenta encontrar la solución del problema de optimización dado por una multiplicación vectorial  $\mathbf{c}\mathbf{x}$  según las restricciones dadas por la solución de un sistema de ecuaciones, según la definición 5.3.1.

**Definition 5.3.1 — Simplex.** Se intenta resolver el sistema lineal

$$\min f(\vec{x}) = \mathbf{c}\mathbf{x}$$

$$\text{condicionado a } A\mathbf{x} = \mathbf{b}, \quad \mathbf{x} > 0$$

El método garantiza que cualquier problema de optimización lineal puede llevarse a la forma estándar mediante tres reglas.

1.  $\max(\mathbf{c}\mathbf{x} : A\mathbf{x} = \mathbf{b}, \quad \mathbf{x}) = -\min(-\mathbf{c}\mathbf{x} : A\mathbf{x} = \mathbf{b}, \quad \mathbf{x})$
2.  $x \geq 0 \rightarrow x = x_1^+ - x_2^-$

3. Slack:  $z = \mathbf{b} - A\mathbf{x} > 0$

La primera establece la equivalencia entre que maximizar algo es equivalente a encontrar el inverso aditivo de la minimización del inverso aditivo de la función de costo.

La segunda regla permite mediante el agregado de una variable, representar variables que pueden tomar valores negativos, como la suma de dos variables, donde ambas son positivas pero la segunda resta la primera.

La tercera regla es para establecer variables de slack<sup>3</sup>. Dada una inecuación, puedo agregar una variable de slack dependiente que representa la diferencia entre el valor de un miembro versus el valor del otro, y reemplazar la inecuación por una igualdad.

#### 5.4 Optimización no lineal sin restricciones

La optimización no lineal sin restricciones permite encontrar los valores que optimizan funciones no-lineales, sin ningún tipo de restricción adicional [56, 57].

$$\min_{x \in \Re^n} f(x), \quad f : \Re^n \rightarrow \Re, \quad (5.7)$$

Donde  $f$  es una función no lineal, por ejemplo:

1.  $f(x_1, x_2) = x_1^2 + \operatorname{sen}(x_1 * x_2) - x_1 * x_2$
2.  $f(x_1, \dots, x_n) = \sum_{i=1}^n (x_i - \cos(x_i))^2$
3.  $f(x_1, \dots, x_n) = \sum_{i=1}^n (x_i - g(x_i, \xi_i))^2$

Esto es particularmente interesante en el caso de las redes neuronales, cuando la función a optimizar es directamente una función llamada función de error,  $E(w)$ , que depende de los pesos sinápticos de la red.

$$\min_{w \in \Re^n} E(w), \quad E : \Re^n \rightarrow \Re, \quad (5.8)$$

donde

- $E(w_1, \dots, w_n) = \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - g(\sum_{i=1}^n w_i x_i^\mu))^2$  es la función de error.
- $w = (w_1, \dots, w_n)$  es el vector de pesos sinápticos que queremos ajustar.
- $n$  es la cantidad de pesos sinápticos o parámetros libres de la red.
- $g(\cdot)$  es una función no lineal de activación.
- $x$  son las  $\mu$  muestras del conjunto de datos, en tanto que  $\zeta$  son las etiquetas de salida asociadas a cada muestra.

Llamar a esta función una función de error (o costo) alude a que se puede considerar a  $g(\sum_{i=1}^n w_i x_i^\mu)$  como un mecanismo, una transformación de una entrada  $x_i^\mu$  (un dato de entrada) en una salida  $\zeta^\mu$ , mediada por parámetros libres  $w_i$ , que son los que se quieren ajustar, para que cada entrada produzca la salida. La función dada mide el error entre lo que devuelve el mecanismo y lo que **debería** salir (el valor deseado).

Existen diferentes alternativas para solucionar estos problemas, y gran parte de la teoría de optimización cubre estas alternativas.

---

<sup>3</sup>Muy famosas en este método.

- Métodos exactos: Calculan una fórmula cerrada para la solución.
- Métodos de aproximación de la solución:
  - Métodos basados en las derivadas primeras, o en el gradiente.
  - Métodos basados en las derivadas segundas, o en el hessiano.
  - Métodos sin derivadas.
  - Métodos Estocásticos (estiman la dirección).

### 5.4.1 Definiciones preliminares

Las siguientes definiciones y teoremas ofrecen las condiciones necesarias y suficientes para establecer optimalidad.

**Definition 5.4.1 — Matriz semi-definida positiva.** Una matriz  $A$  es definida semi-positiva si

- Si  $x^t Ax > 0$ ,  $x \neq 0$ .
- Si todos sus autovalores son positivos
- Hay otras características que definen a las matrices definidas positivas, por ejemplo los valores de la diagonal deben ser mayores que la suma de los otros elementos de la fila.
- Si  $x^t Ax \geq 0$ ,  $x \neq 0$ , entonces se dice que la matriz es **semi definida positiva**

**Teorema 5.4.1 — Existencia de mínimos locales y globales.** Sea la función  $f$  diferenciable con primera y segunda derivadas continuas, entonces  $x^*$  ...

- Es mínimo global si  $\forall x$ ,  $f(x^*) \leq f(x)$
- Es mínimo local si  $f(x^*) \leq f(x) \forall x$ ,  $\|x - x^*\| < \epsilon$

**Teorema 5.4.2 — Condiciones necesarias de optimalidad.** Las condiciones son:

- **Condición necesaria de primer orden**  
Si  $x^*$  es un mínimo local de  $f$  entonces  $\nabla f(x^*) = 0$ .
- **Condición necesaria de segundo orden**  
Si  $x^*$  es un mínimo local de  $f$  entonces  $\nabla f(x^*) = 0$  y  $H_f(x^*)$  (el hessiano) es una matriz semidefinida positiva.

**Teorema 5.4.3 — Condición suficiente de optimalidad.** Si  $x^*$  es tal que  $\nabla f(x^*) = 0$  y  $H_f(x^*)$  es definida positiva, entonces es un mínimo local de  $f$ .

El Hessiano determina si la función es concava o convexa:

- Función convexa → Hessiano matriz definida positiva.
- Función cóncava → Hessiano matriz definida negativa.

Hete aquí lectores, que encontrarán siempre detrás de cualquier tipo de algoritmo que ose llamar de IA, el siguiente procedimiento subyacente en el entrenamiento de sus parámetros libres, de alguna u otra manera.

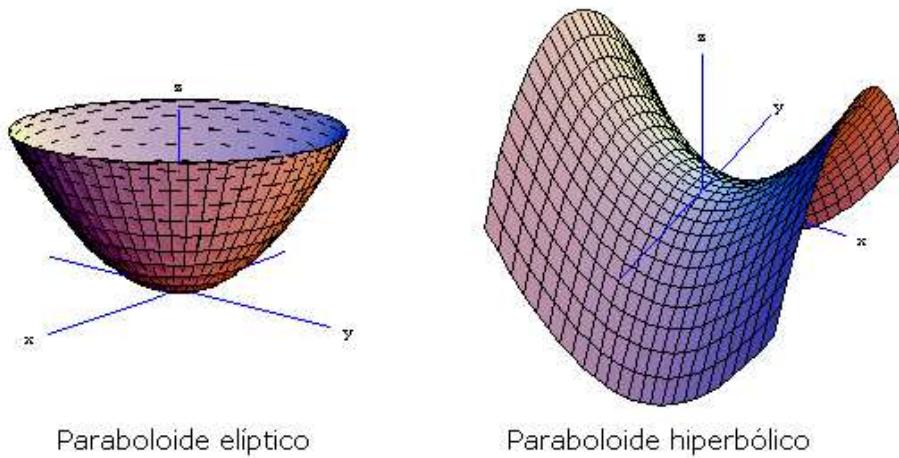


Figura 5.2: Hessiano definido positivo

Hessiano NO definido positivo

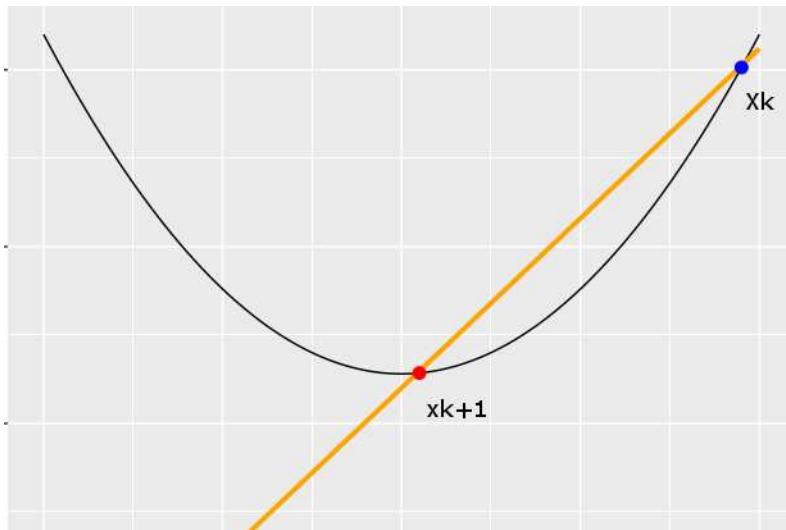


Figura 5.3: El objetivo del procedimiento general de optimización es encontrar la dirección y el paso para pasar de  $X_k$  a un  $X_{k+1}$  que produzca un valor más pequeño en la función de costo.

---

**Algoritmo 4** Procedimiento general de optimización

- 1: Punto inicial  $x_k$ , dato de entrada.
  - 2: Iterativamente para cada paso  $k$ .
  - 3: **while** no converge **do**
  - 4:     Buscar una dirección de movimiento  $d_k$ .
  - 5:     Calcular o decidir la longitud de paso  $\alpha_k$ .
  - 6:     Actualizar al nuevo punto  $x_{k+1} = x_k + \alpha_k d_k$ .
  - 7: **end while**
  - 8: Salida: Puntos  $x^*$  idealmente óptimo global.

La idea detrás del Algoritmo 4 es encontrar dos cosas. Una es el largo del paso de iteración y la segunda es la dirección para ir encontrando puntos que cumplan  $f(x_{k+1}) < f(x_k)$ . La dirección de movimiento debe ser descendiente, o sea  $d_k^t \nabla f(x_k) < 0$  a sabiendas de que:

- El gradiente es la dirección de máximo crecimiento de una función.

- Cualquier dirección contraria a la del gradiente, es una dirección de decrecimiento de la función.

y el paso debe ser tal que permita que el algoritmo converja en un tiempo acotado y que a su vez permita en un número de pasos finito siempre acercarse más a la solución.

### 5.4.2 Gradiente Descendente

Este famoso método se basa justamente en utilizar la dirección de máximo decrecimiento de la función a optimizar que es la correspondiente al gradiente.

$$d_k = -\nabla f(x_k) \quad (5.9)$$

Si bien tiene convergencia lenta, tiene la ventaja de no requerir información de la segunda derivada.

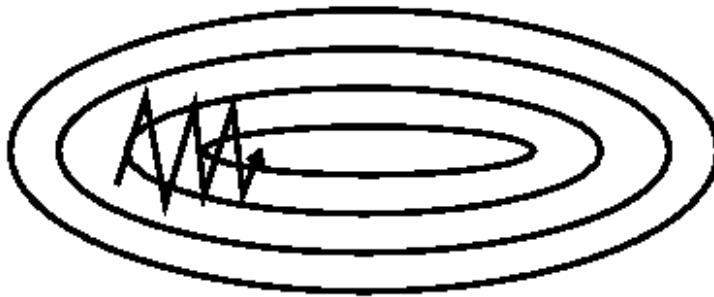


Figura 5.4: Gradiente descendente avanza en zig-zag.

La Figura 5.4 muestra las curvas de nivel de la función escalar de error, el centro sería el valor mínimo. El gradiente sigue la dirección indicada, lo cuál se visualiza inmediatamente que genera un problema al aproximarse a la solución en zig zag, lo que no es un buen balance entre velocidad de acercamiento y precisión. Se puede suavizar un poco más utilizando el concepto de **Momentum**, que como veremos más adelante se basa en la idea de tomar la dirección de descenso como una combinación lineal de direcciones de descenso calculadas en pasos anteriores.

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) - \beta \alpha_{k-1} \nabla f(x_{k-1}) \quad (5.10)$$

Este promedio ponderado entre direcciones suaviza el zig zagueo del método del gradiente con  $0 < \beta < 1$ . Esto es porque la nueva dirección es un promedio ponderado por beta de las dos direcciones anteriores.

### 5.4.3 Método de Newton

Como se vio en el ejemplo de la latita de Coca, la derivada segunda aporta información. Lo que entonces podemos hacer es aproximar el gradiente en sí mismo por el polinomio de Taylor de segundo orden,

$$\nabla f(x_{k+1}) = \nabla f(x_k + \alpha_k d_k) = \nabla f(x_k) + \alpha_k H(x_k) d_k \quad (5.11)$$

Entonces, si  $x_{k+1}$  fuera el mínimo,  $\nabla f(x_{k+1}) = 0$  y por lo tanto

$$\nabla f(x_k) + \alpha_k H(x_k) d_k = 0$$

de donde resulta

$$d_k = -H^{-1}(x_k) \nabla f(x_k)$$

Este caso parecería ser la solución definitiva, ya que se puede encontrar el punto exacto donde ocurre el mínimo (sin tener que iterar). Sin embargo, el problema aparece en tener que estimar el Hessiano y/o su inversa para cualquier función que haya que optimizar.

#### 5.4.4 Métodos Cuasi Newton

Estos métodos se basan en intentar disminuir el costo computacional asociado al cálculo del Hessiano y particularmente su inversa. Se basan en realizar una aproximación de  $H^{-1}(x_k)$  y reemplazarla por una matriz de aproximación definida positiva  $B_k$ . Las variantes de estos métodos justamente difieren en como realizar la aproximación de esta matriz.

Sin embargo, aún con la mejora en el rendimiento, los métodos cuasi Newton tienen la desventaja de poseer un alto costo computacional para aplicaciones de entrenamiento de redes neuronales.

En su lugar, se puede utilizar el método **L-BFGS (limited memory Broyden Fletcher Goldfarb Shanno (BFGS))** [58]. Este algoritmo, siguiendo con la idea del huevo y la gallina, implementa una búsqueda iterativa para estimar el Hessiano  $B_{k+1} \sim B_k$ .

#### 5.4.5 Métodos de Direcciones Conjugadas

Estos métodos buscan las direcciones por las cuales buscar la solución, explotando el espacio de direcciones que se establecen mediante la idea de direcciones conjugadas con una matriz. Tienen la ventaja de ofrecer una garantía de convergencia y una cota en la cantidad de pasos cuando la forma de la función de costo es específica.

**Teorema 5.4.4 — Vectores A-conjugados.** Sea el conjunto de direcciones (vectores)  $d_1, \dots, d_n$  y  $A$  una matriz simétrica definida positiva, entonces:

- Si  $d_i^t A d_j = 0, \forall i \neq j$  entonces se dice que  $d_1, \dots, d_n$  son direcciones  $A$ -conjugadas.
- Si un conjunto de vectores es  $A$ -conjugado, con  $A$  simétrica, definida positiva entonces es también un conjunto linealmente independiente.

**Teorema 5.4.5 — Cota en la convergencia de los Métodos de Direcciones Conjugadas.**

Dada una función cuadrática  $f : \Re^n \rightarrow \Re$   $f(x) = x^t H x + b^t x$  Si un método de minimización no lineal realiza las búsquedas unidimensionales sobre direcciones  $H$ -conjugadas, entonces el método converge en  $n$  pasos.

Por supuesto no siempre es posible que la función de costo tenga la forma que establece el teorema, pero sirven como un punto de partida de otros métodos que intentan aproximar la idea con funciones mucho más complejas.

#### 5.4.6 Métodos de Gradientes Conjugados 1952

Este método se basa en que para cada paso  $k$ , se calcula una nueva dirección  $d_{k+1}$  que es  $H(x_k)$ -conjugada con todas las direcciones anteriores  $d_1, \dots, d_k$ , basada en el  $\nabla f(x_k)$ .

Exige sólo identificar el gradiente en cada paso.

#### 5.4.7 El método de Powell 1964

Este método está basado en la determinación de direcciones conjugadas sin requerir el uso de derivadas parciales (orden cero). El método genera en cada paso  $k$  un conjunto de direcciones conjugadas

$$d_1^k, \dots, d_n^k.$$

Comienza con un conjunto de direcciones linealmente independientes  $d_1^0, \dots, d_n^0$  y un punto inicial  $x_0$ . Luego en el paso  $k$ , saca del conjunto la dirección  $d_k^k$  y agrega una dirección conjugada con  $d_1^{k-1}, \dots, d_{k-1}^{k-1}$  (las de los pasos anteriores). Este método es en el fondo, una búsqueda lineal.

### 5.4.8 Gradiente Descendente Estocástico

El método más utilizado en redes neuronales es el **SGD**, Stochastic Gradient Descend, y se basa en una extensión al método de gradiente descendente pensada sobre todo para funciones de error.

Se tienen  $p$  observaciones  $\mathbf{x}^\mu = \{x_1, \dots, x_n\}$ ,  $\mu = 1, \dots, p$  de dimensión  $n$  del conjunto de entrenamiento ( $X$ ) junto con sus  $p$  salidas  $\zeta^\mu$  de dimensión 1 y  $\mathbf{w} = (w_1, \dots, w_n)$  el vector de pesos sinápticos (parámetros libres). Entonces, queremos hallar  $\mathbf{w}$  que minimice  $E(\mathbf{w})$ :

$$E(\mathbf{w}) = \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - \sum_{i=1}^n w_i x_i^\mu)^2 \quad (5.12)$$

$$E(\mathbf{w}) = \frac{1}{p} \sum_{\mu=1}^p E^\mu(x^\mu, \mathbf{w}) \quad (5.13)$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \frac{1}{p} \sum_{\mu=1}^p \nabla_{\mathbf{w}} E^\mu(x^\mu, \mathbf{w}^t)$$

Considerando que  $\frac{1}{p} \sum_{\mu=1}^p \nabla_{\mathbf{w}} E(x^\mu, \mathbf{w}^t)$  es un estimador de la esperanza del gradiente, dado un conjunto de entrenamiento,... **¿Por qué no usar cualquier otro estimador?**

Por ejemplo: un

$$\nabla_{\mathbf{w}} E(x^\nu, \mathbf{w}^t),$$

para algún  $\nu$  arbitrario o un subconjunto aleatorio  $\sum_{\mu=1}^k \nabla_{\mathbf{w}} E(x^\mu, \mathbf{w}^t)$ ,  $k \ll p$ .

Esta es la esencia del método de gradiente descendente estocástico. Donde solamente una parte de los datos se utiliza para calcular la dirección de descenso en cada paso. El nombre original de este método fue ADALINE, o Adapter Linear Element.

Los autores del método, demuestran en su libro [59] que el método converge, pero no siempre va descendiendo.

#### ADAGrad, 2011 (Adaptive Gradient)

Es un método que mejora ADALINE que tiene como principal objetivo adaptar el valor de la tasa de aprendizaje en cada paso y la actualización del vector  $\mathbf{w}$  se realiza coordenada a coordenada. Sea  $g_t = \nabla E(\mathbf{w}_{t-1})$  entonces  $(g_t)_i$  es la  $i$ -ésima coordenada  $g_{1:t} = \{g_1, \dots, g_t\}$  son todos los gradientes anteriores hasta el paso  $t$ .

**Mientras que en SGD se hace**

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \nabla_{\mathbf{w}} E^\mu(x^\mu, \mathbf{w}^t)$$

**En ADAGRAD se calcula**

$$\mathbf{w}_i^{t+1} = \mathbf{w}_i^t - \frac{\eta_t}{\sqrt{G_{ii}^t + \epsilon}} \nabla_{\mathbf{w}_i} E^\mu(x^\mu, \mathbf{w}^t)$$

con  $G^t = \sum_{\tau=1}^t g_\tau * g_\tau^t$ .

La ventaja de este método es que cada coordenada tiene su propia actualización.



## 6. El modelo de Neurona

Frank Rosenblatt (1928-1971), de formación psicólogo experimental, desarrolló un mecanismo denominado Perceptrón en la Universidad de Cornell, en el marco de un proyecto para la marina norteamericana<sup>1</sup>. Ideó el algoritmo y su implementación en hardware *Mark I*. Tenía 400 sensores de luz que juntos actuaban como una retina, proveyendo información a 1000 neuronas en una sola capa que realizaban el procesamiento y producían una única salida.

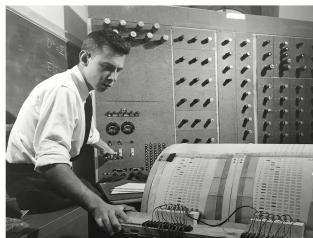


Figura 6.1: Rosenblatt con su Perceptrón tirando magia.

En 1958, el New York Times declaraba que *la máquina será el primer dispositivo en pensar como lo hace el cerebro humano* [39, 60]. La motivación e inspiración de Rosenblatt radicaba en su formación y tendencia a la interdisciplinariedad, característica esencial del campo de redes neuronales. Frank buscaba *entender la cantidad mínima de elementos que un cerebro debía tener físicamente, para capturar las maravillosas cosas que hace*. El algoritmo se basa en una combinación lineal de entradas, luego umbralizadas para generar una respuesta final binaria, similar a los impulsos eléctricos que transmiten las neuronas del cerebro. Rosenblatt fue capaz de probar que, con una sola capa y una regla simple de actualización de los pesos, estos podían converger hasta alcanzar el cálculo computacional que necesitaba; en pocas palabras, estableció un esquema probado de *aprendizaje*. El algoritmo del Perceptrón es una piedra fundamental de la disciplina que busca artificializar la inteligencia, y fue el primer trabajo que impulsó el **hype** en torno a esta área, una característica constante que podemos experimentar a día de hoy.

<sup>1</sup>Gran parte de las líneas de investigación en inteligencia artificial tienen raíces que arrancan en la Segunda Guerra Mundial o en el período posguerra en el marco de la carrera armamentista entre la Unión Soviética y EEUU. Interesante relato sobre el tema en "A History of Warfare", Mariscal Montgomery, "Engineers of Victory: The Problem Solvers Who Turned the Tide in the Second World War" de Paul Kennedy, o "Accessory to War" de Neil deGrasse Tyson.

## 6.1 La bioinspiración

La idea del Perceptrón simple tiene su origen en el modelo de neurona creado por McCulloch y Pitts, en 1943 [3]. Es un modelo simplificado, basado en los conocimientos que existían en la época en relación al funcionamiento de las neuronas y a la capacidad de encontrar un modelo abarcativo simple que explique las principales características. Establece que una neurona posee un conjunto de entradas que llegan al soma (o cuerpo) donde se produce una integración de la información (sumación). Si el valor de dicha integración supera cierto umbral, se produce una salida binarizada de la neurona, correspondiente a una salida activa. El modelo simplificado puede verse en la figura 6.2.

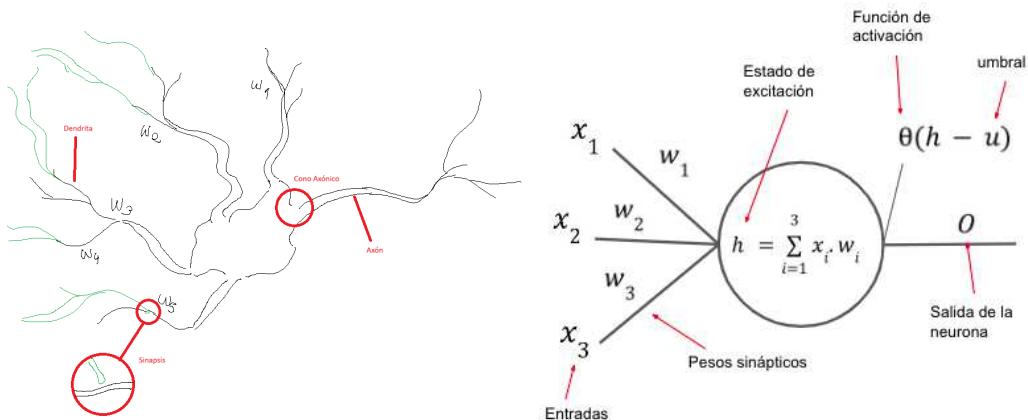


Figura 6.2: Diagrama y modelo simplificado de una neurona.

Matemáticamente este modelo corresponde a:

$$O = \phi\left(\sum_{i=1}^n w_i x_i - u\right) \quad (6.1)$$

donde  $O$  es la salida,  $x_i$  son las entradas,  $w_i$  son los pesos (parámetros libres),  $u$  es el umbral. Por otro lado,  $\phi$  es la función de activación, lo cual para las primeras versiones del Perceptrón se circunscribía a la función escalón o signo:

$$\phi(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases} \quad (6.2)$$

## 6.2 Separabilidad lineal

Consideremos un conjunto de datos en dos dimensiones, donde cada eje corresponde a cierta característica y los puntos pueden ser clasificados en dos clases. En este caso, se clasifican individuos según su orientación política, de acuerdo a las características de edad noramlizada (valores entre 0 y 1) e ingresos económicos normalizados.

¿Es posible encontrar una recta de separación que permita separar perfectamente en dos grupos de acuerdo a su clase? De la figura 6.3 se ve que en ese caso esta recta existe y puede modelarse según la clásica ecuación  $y = mx + b$ .

La existencia de esta recta permite esbozar una generalización: dado un punto nuevo, puedo determinar a qué clase pertenece, dependiendo de su posición con respecto a la recta de separación.

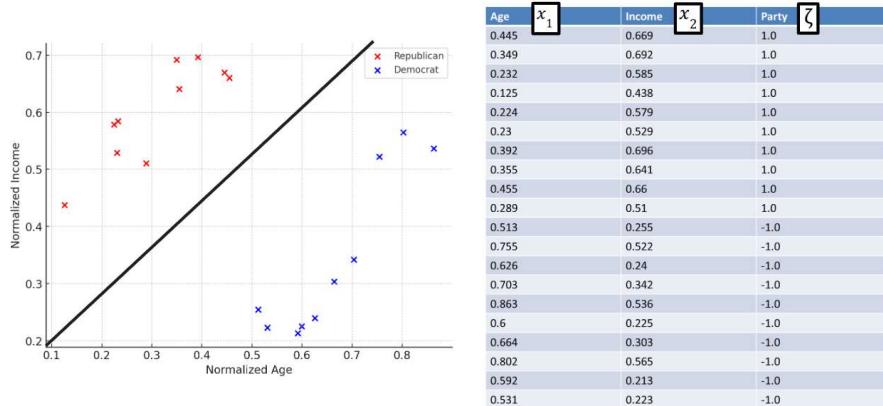


Figura 6.3: Conjunto de puntos en dos dimensiones, cada uno perteneciente a una clase diferente.

Para eso, es posible proyectar el punto hacia la componente ortogonal a la recta y evaluar el signo de esa función.

Si recordamos la fórmula del plano, caracterizado por su normal, vemos que su ecuación está dada por

$$(X - N) \cdot P = 0$$

con  $N$  la normal al plano y  $P$  un punto del plano. La proyección entonces sobre esta recta está determinada por:

$$\text{Proy}_N X_\mu = \frac{X_\mu \cdot N}{\|N\|} N.$$

La magnitud en concreto de la proyección de cada uno de los puntos sobre la normal termina siendo principalmente el producto interno dado por

$$P = \sum_{i=1}^n w_i x_i. \quad (6.3)$$

donde  $n$  es la dimensión del espacio. Si  $P$  es positivo, identificamos a ese punto como perteneciente a la clase 1 en tanto que si es negativo, lo identificaremos como perteneciente a la clase 2. Es decir, tenemos un clasificador básico capaz de ofrecer una generalización.

$$\phi(P) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases} \quad (6.4)$$

### 6.2.1 Aprendizaje Hebbiano

El modelo McCulloch-Pits permite resolver un modelo simple de clasificación binaria.

En lo explicado previamente, hemos logrado implementar conceptos relacionados con la neurona biológica, como soma, axones y dendritas, a una simplificación matemática para obtener este clasificador. La pieza faltante a este modelo de neurona radica en el



Figura 6.4: Donald Hebb pensando usando sus propias neu-

concepto de sinapsis, del cual no hemos hecho mención. El concepto de sinapsis explica la fortaleza de la conexión entre neuronas en el cerebro y, de acuerdo a esta fortaleza, las neuronas son capaces de excitar o inhibir otras neuronas. La sinapsis se vuelve más fuerte mediante procesos de aprendizaje. Entonces, ¿cómo se puede generar este proceso de *aprendizaje* en nuestro sistema, emulando lo que ocurre en sistemas biológicos reales?. Donald Hebb [4] entra en escena, con el planteo de su conjectura homónima: las conexiones (sinapsis) donde intervienen neuronas cuyo estado de actividad están correlacionados a lo largo del tiempo, son reforzadas y este refuerzo depende de la intensidad y período de tiempo en que esto ha ocurrido. Este planteo es conocido por su famosa frase

"Neurons that fire together, wire together"

Esta conjectura permitió dar explicación a diferentes fenómenos observados en psicología conductual. Por ejemplo, el aprendizaje condicionado de Pavlov [61].

Entonces, que cada vez que la neurona recibe un nuevo estímulo, podemos aplicar una regla de aprendizaje que permita variar cada uno de los pesos sinápticos. Dicha variación se hará de acuerdo a la información provista por el estímulo.

$$w_i^{\text{nuevo}} = w_i^{\text{viejo}} + \Delta w_i \quad (6.5)$$

donde el valor  $\Delta w_i$  dependerá de la activación de la entrada y también de la salida, a la vez.

$$\Delta w_i = 2\eta x^\mu \zeta^\mu \quad (6.6)$$

con  $\eta$  actuando como una constante de proporcionalidad que regula el aprendizaje.

Con esto se puede derivar una regla simple de actualización. Cuando se presenta la entrada  $x^\mu$  y el estado de activación  $O^\mu$  coincide con la salida deseada  $\zeta^\mu$  no hay que hacer nada. Pero, si no coincide, hay que actualizar  $w_i$ . Esto equivale a:

$$\Delta w_i = \begin{cases} 2\eta x^\mu \zeta^\mu & \text{si } O^\mu \neq \zeta^\mu, \\ 0 & \text{otro caso} \end{cases} \quad (6.7)$$

Asumiendo que  $O^\mu$  y  $\zeta^\mu$  toman valores 1 y -1, la función por partes se puede escribir:

$$\Delta w_i = \eta (\zeta^\mu - O^\mu) x^\mu \quad (6.8)$$

Esto permite, a su vez, una interpretación desde la perspectiva de la representación geométrica del problema de separabilidad lineal. Cuando  $\zeta^\mu > O^\mu$ , la ecuación 6.8 representa mover un poco al vector de pesos hacia el patrón de entrada  $x^\mu$ . Con esto, la próxima vez que calculamos el producto interno  $w_i x^\mu$  será mayor y más probable que  $\zeta^\mu$  y  $O^\mu$  se parezcan.

Desde el punto de vista geométrico, la regla de actualización hebbiana está alterando el plano de separación para dividir de manera adecuada a los elementos en sus clases. Esta conexión que suena trivial, es muy potente, porque la base de la regla proviene de un modelo mecanístico experimental del comportamiento de la estructura sináptica de las neuronas.

### 6.2.2 Algoritmo del Perceptrón Simple

Señoras, Señores, nerds, otakus, agarrense de sus butacas, afirmen bien el mate y el termo, porque estamos en posición de desplegar el algoritmo más importante de este volumen: el algoritmo del Perceptrón simple.

**Definition 6.2.1 — Epoca.** Se le llama a una pasada del conjunto de datos de entrenamiento, por la ecuación 6.8.

**Definition 6.2.2 — Hiperparámetros.** Son aquellos valores que se pueden especificar en el algoritmo y que afectan el proceso de aprendizaje. Si bien pueden cambiar, NO son actualizadas por el propio proceso de aprendizaje.

**Definition 6.2.3 — Pesos sinápticos.** Son los elementos del vector  $w$  que determinan el plano de separación. De acuerdo al modelo de neurona, representan la fuerza de conexión sináptica. Estos son los parámetros libres a ajustar mediante el aprendizaje (incluyendo el umbral).

**Definition 6.2.4 — Conjunto de entrenamiento.** El conjunto de entrenamiento  $x[\cdot]$ , donde cada entrada tiene dimensión  $N$  (teniendo en cuenta que se agrega una coordenada más por el umbral). Hay  $p$  entradas.

**Definition 6.2.5 — Salidas esperadas.** Las salidas esperadas quedan plasmadas en  $y[\cdot]$ .

**Definition 6.2.6 — Función de activación.** La función de activación determina la salida de la neurona y restringe los posibles valores. En el caso del perceptrón simple, dicha función está dada por  $\text{sign}(\cdot)$ .

---

#### Algoritmo 5 Algoritmo del Perceptrón Simple

---

```

1: Inicializar  $i = 0$ 
2: Inicializar  $w = \text{zeros}(N, 1)$ 
3: Inicializar  $error = 1$ 
4: Inicializar  $error_{min} = p * 2$ 
5: Especificar hiperparámetro COTA.
6: while  $error > 0 \wedge i < COTA$  do
7:    $i_x = \text{ceil}(\text{rand}(1, 1) * p)$ 
8:    $exitacion = x(i_x, :) * w$ 
9:    $activacion = \text{sign}(exitacion)$ 
10:  Calcular  $\Delta w = \alpha * (y(1, i_x) - activacion) * x(i_x, :)$ 
11:   $w = w + \Delta w$ 
12:   $error = \text{CalcularError}(x, y, w, p);$ 
13:  if  $error < error_{min}$  then
14:     $error_{min} = error$ 
15:     $w_{min} = w$ 
16:  end if
17:   $i = i + 1$ 
18: end while
19:  $w$  Resultado de los pesos entrenados (si es posible) para el dataset  $x$ .

```

---

**Definition 6.2.7 — Inicialización de los pesos sinápticos.** El esquema de inicialización de los pesos es un hiperparámetro más a especificar. Una estrategia conservadora es inicializarlos

de manera aleatoria con  $w = \text{rand}(N, 1) * 2 - 1$ .

### 6.2.3 Perceptrón Lineal

Si se reemplaza la función de activación escalón por la función identidad, el Perceptrón se convierte en lineal, donde la salida solo depende del producto interno entre los patrones de entrada  $x$  y los pesos  $w$ .

$$O^\mu = \sum_{i=1}^N w_i x_i^\mu \quad (6.9)$$

Los valores así ahora son continuos y no se encuentran más umbralizados entre  $\{-1, 1\}$  o  $\{0, 1\}$  como ocurría con las unidades escalón. Los valores pertenecen a los reales. De esta interpretación, se puede asumir una función de costo, derivable, que adoptará un mínimo absoluto cuando el valor del vector de pesos  $w$  haga que la salida  $O^\mu = \zeta^\mu$ , es decir que genere la salida exacta. Esta función tiene un factor arbitrario de  $\frac{1}{2}$  que ya veremos para qué sirve y se basa en SSE, Sum of Squared Errors, que se usa en regresión lineal.

$$\begin{aligned} E(w) &= \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - O^\mu)^2 \\ &= \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - \sum_{i=1}^N w_i x_i^\mu)^2 \end{aligned}$$

Es importante notar que  $E(\cdot)$  depende de los pesos  $w_i$ . Entonces, si se logra mover  $w$  en la dirección correcta podríamos acercarnos al valor mínimo de  $E(\cdot)$  que, por ser una función de error, idealmente debería ser cero. Una posibilidad, sacando nuestro manual de optimización, es usar el gradiente, ya que representa la dirección de mayor crecimiento de la función que estamos buscando minimizar. Con esto en mente, moveremos a los pesos sinápticos en la dirección **opuesta a la de mayor crecimiento**:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad (6.10)$$

para lo cuál calculamos

$$\frac{\partial E}{\partial w_i} = \frac{\partial E(\frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - O^\mu)^2)}{\partial w_i} = (2) (-1) \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - \sum_{i=1}^N w_i x_i^\mu) (x_i^\mu),$$

y con esto reemplazandolo en 6.10:

$$\begin{aligned} \Delta w_i &= -\eta \frac{\partial E}{\partial w_i} \\ &= \eta \sum_{\mu=1}^p (\zeta^\mu - O^\mu) x_i^\mu \end{aligned}$$

Con esto se modifica cada peso puntual evaluando todos los patrones. Si se quiere hacer el ajuste evaluando sólo un patrón nos queda

$$\Delta w_i = \eta(\zeta_i^\mu - O_i^\mu)\xi_i^\mu \quad (6.11)$$

Oppssss...esta es exactamente la misma regla de actualización del Perceptrón con la función de activación escalón. **Algo que planteamos inspirados en un modelo biológico experimental coincide con el planteo, desde el punto de vista matemático, de minimizar el valor de una función!** A esta regla llamada **delta rule** también se la conoce como la regla de Widrow-Hoff (1960), y estos autores la bautizaron como ADALINE, o Adapter Linear Element.<sup>2</sup>.

### Aprendizaje en el Perceptrón lineal

El Perceptrón simple escalón resuelve problemas linealmente separables. Por otro lado, el Perceptrón lineal simple resuelve sistemas lineales independientes<sup>3</sup>. Es decir encuentra una aproximación del  $w$  que resuelve

$$\mathbf{x}\mathbf{w} = \zeta,$$

o sea

$$\begin{aligned} w_1x_1^1 + w_2x_2^1 + \dots + w_Nx_N^1 &= \zeta_1^1 \\ w_1x_1^\mu + w_2x_2^\mu + \dots + w_Nx_N^\mu &= \zeta_1^\mu \\ w_1x_1^p + w_2x_2^p + \dots + w_Nx_N^p &= \zeta_1^p \end{aligned}$$

La convergencia de este aprendizaje depende fuertemente del valor de  $\eta$ , recordando que hay que actualizar  $w_i^{nuevo} = w_i^{viejo} + \Delta w_i$  en base a  $\Delta w_i = \eta(\zeta^\mu - O^\mu)x^\mu$  por lo que

- Si  $\eta$  no es lo suficientemente pequeña, el método puede diverger.
- Si  $\eta$  es demasiado pequeña puede demorar la convergencia.

#### 6.2.4 Aprendizaje en el Perceptrón Simple No Lineal

La idea del aprendizaje basado en la optimización del gradiente, puede extenderse fácilmente a una situación no-lineal. El estado de activación del Perceptrón es

$$O^\mu = g\left(\sum_{i=1}^N w_i x_i^\mu\right) \quad (6.12)$$

donde  $g(\cdot)$  puede ser una función sigmoidea tal como  $g(x) = \tanh(\beta x)$  o  $g(x) = \frac{1}{1+exp^{2\beta x}}$ . Una lista más exhaustiva de variantes de funciones de activación puede encontrarse en 21.2. La función de

$g(h)$	$g'(h)$
$\tanh(\beta h)$	$\beta(1 - g(h)^2)$
$\frac{1}{1+exp^{2\beta x}}$	$2\beta g(h)(1 - g(h))$

Cuadro 6.1: Funciones de activación simples y sus derivadas (1/2)

costo queda definida exactamente igual que con el Perceptrón simple lineal

<sup>2</sup>Ted Hoff fue estudiante de Bernard Widrow y más adelante, en 1968, se convirtiría en el co-inventor del predecesor del microprocesador (un conjunto de 12 circuitos diferentes en un sólo chip que cambiaba su funcionalidad de acuerdo al requerimiento) [62]

<sup>3</sup>La demostración por supuesto es obvia y trivial y queda como ejercicio para el lector ;).

$$\begin{aligned} E(w) &= \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - O^\mu)^2 \\ &= \frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - \sum_{i=1}^N w_i x_i^\mu)^2. \end{aligned}$$

Ahora para actualizar los pesos debemos utilizar el gradiente descendente con

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

pero usamos **la regla de la cadena**<sup>4</sup> para propagar la derivada también de  $g$  dentro del cálculo

$$\Delta w_i = \eta \sum_{\mu=1}^p (\zeta^\mu - g(h^\mu)) g'(h^\mu) x_i^\mu$$

donde usamos que  $h^\mu = \sum_{i=1}^N w_i x_i^\mu$ .

Nuevamente, considerando tomar sólo una única entrada en particular.

$$\Delta w_i = \eta (\zeta^\mu - O^\mu) g'(h^\mu) x_i^\mu \tag{6.13}$$

La ventaja de la función logística o la tangente hiperbólica es que las derivadas se pueden expresar como funciones de ellas mismas.

Todo esto es muy lindo. Pero tiene un problemón. En 1969 se publicó un influyente paper de Minsky y Papert el cual planteó que esta mecánica (que auguraba decodificar nada más y nada menos que el cerebro humano) fallaba para resolver una función booleana trivial como el XOR. Este fue el inicio del primer **invierno** de la inteligencia artificial.

---

<sup>4</sup>Esto veremos que es fundamental para el capítulo siguiente donde expandimos esto a esquemas de muchas capas.

## 7. La era del Coneccionismo

El problema del XOR del paper de Minsky y Papert [20] tenía una resolución simple que incluso estaba planteada directa y explícitamente en ese mismo trabajo. Esta era la utilización de una estructura *jerárquica aglomerativa*, en capas sucesivas. Esta idea tiene un correlato en neurociencia donde se verifica que esa estructura se encuentra presente en los sistemas nerviosos de muchos seres vivos [63].

### 7.1 Estructura Jerárquica Aglomerativa

Las salidas de las neuronas se pueden utilizar como entradas para neuronas ubicadas en una capa siguiente. Este esquema, soluciona el problema del XOR, porque permite agregar un plano de separación adicional, y así sucesivamente. Es una solución simple, que requiere un cambio en el establecimiento de una **red** neuronal y su arquitectura.

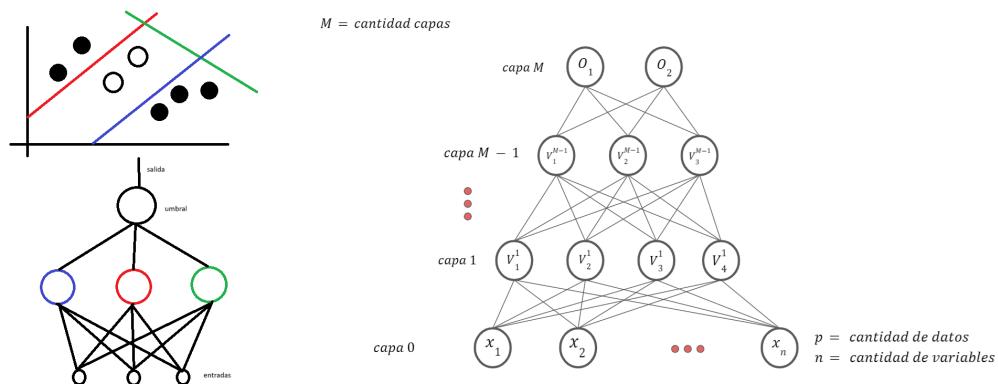


Figura 7.1: Perceptrón multicapa

Esta red tiene una estructura jerárquica, donde está dividida en capas en diferentes niveles: una capa de entrada, capas intermedias ocultas, y una capa de salida. Y es también aglomerativa, porque

la información de una capa se usa conjuntamente, se aglomera, para convertirse en la entrada de la capa subsiguiente. Esta es la esencia del Perceptrón multicapa, o MLP (Multilayered Perceptron).

### 7.1.1 Feedforward

¿Cómo es entonces la evolución de las entradas hacia adelante en nuestra red? Consideremos la red planteada en la figura 7.2. Los pesos sinápticos  $w_{jk}$  corresponden a los pesos de la capa oculta, en tanto que  $W_{ij}$  corresponde a los pesos de la capa de salida. Las activaciones de la capa oculta las llamamos  $V_j$ .

- $k, j, i$  son los índices para referenciar las entradas, las neuronas de la capa oculta, y las neuronas de la salida.
- El supraíndice  $\mu$  indica el índice de la muestra de entrada, con  $p$  la cantidad de ejemplos.
- El subíndice  $k$  indica el índice de la dimensión del ejemplo.

Al presentar la muestra  $\mu$  a la red, en la capa oculta se calcula

$$h_j^\mu = \sum_k^N w_{jk} x_k^\mu \quad (7.1)$$

lo que permite generar un activación en la capa oculta como

$$V_j^\mu = g(h_j^\mu) = g\left(\sum_k^N w_{jk} x_k^\mu\right) \quad (7.2)$$

En tanto que en las salidas de la capa oculta tenemos,

$$h_i^\mu = \sum_k^N W_{ij} V_j^\mu = \sum_k^N W_{ij} g\left(\sum_k^N w_{jk} x_k^\mu\right) \quad (7.3)$$

$$O_i^\mu = g(h_i^\mu) = g\left(\sum_i^N W_{ij} V_j^\mu\right) = g\left(\sum_i^N W_{ij} g\left(\sum_k^N w_{jk} x_k^\mu\right)\right) \quad (7.4)$$

Esto determina todas las salidas de la red hacia adelante.

### 7.1.2 Backward Backpropagation

En el Perceptrón el aprendizaje está dado por el proceso de actualizar los pesos sinápticos cuando la salida es distinta al valor esperado. Esto es directo, ya que la contribución de la salida para el cálculo del peso es inmediata. De una proporción del error se corrigen directamente los pesos de las conexiones que unen directamente las entradas con las salidas. Ahora, ¿cómo hacer esto en el caso de una arquitectura con capas ocultas? ¿Cómo llevamos para atrás ese error a los pesos ocultos? ¿Qué estado de activación deseado tienen que tener las neuronas de la capa intermedia para poder establecer un error y de acuerdo a eso corregir los pesos de las conexiones que unen las entradas con esas neuronas ocultas intermedias?

En 1986, Rumerhart, Hinton y Williams presentan su propuesta para solucionar esto con el algoritmo de Backpropagation, en el contexto de su trabajo de PDP (Procesamiento Paralelo Distributivo) [19].

Retomando el aprendizaje en el Perceptrón.

$$h = \sum_{i=1}^N w_i x_i \quad (7.5)$$

$$O^\mu = g(h) \quad (7.6)$$

Y así la función de error es:

$$\begin{aligned} E &= \frac{1}{2} \sum_{\mu=1}^N (\zeta^\mu - O^\mu)^2 \\ &= \frac{1}{2} \sum_{\mu=1}^N (\zeta^\mu - g(\sum_{i=1}^N w_i x_i)) ^2 \end{aligned}$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad (7.7)$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial \frac{1}{2} \sum_{\mu=1}^N (\zeta^\mu - O^\mu)^2}{\partial w_i} \\ &= \frac{\partial \frac{1}{2} \sum_{\mu=1}^N (\zeta^\mu - g(\sum_{i=1}^N w_i x_i^\mu))^2}{\partial w_i} \\ &= \frac{\frac{1}{2} \sum_{\mu=1}^N \partial (\zeta^\mu - g(\sum_{i=1}^N w_i x_i^\mu))}{\partial w_i} \\ &= \frac{\sum_{\mu=1}^N \partial (\zeta^\mu - g(\sum_{i=1}^N w_i x_i^\mu))}{\partial w_i} \\ &= \frac{\sum_{\mu=1}^N (\zeta^\mu - O^\mu) g'(h)(\partial \sum_{i=1}^N w_i x_i^\mu)}{\partial w_i} \\ &= \sum_{\mu=1}^N (\zeta^\mu - O^\mu) g'(h)(x_i^\mu) \end{aligned}$$

$$\Delta w_i = -\eta \sum_{\mu=1}^N (\zeta^\mu - O^\mu) g'(h)(x_i^\mu)$$

La actualización requerida en el peso para un patrón en particular:

$$\Delta w_i = -\eta (\zeta^\mu - O^\mu) g'(h)(x_i^\mu)$$

**Definition 7.1.1 — Backpropagation.** Se refiere al método o algoritmo por el cuál en una red jerárquica aglomerativa realiza los ajustes requeridos sobre los parámetros libres. Este algoritmo implica una propagación hacia atrás en una red mediante el uso de la regla de la cadena para métodos de optimización de primer y segundo orden.

Retomando entonces la figura 7.2, la salida de activación en la capa oculta es:

$$V_j = g(h_j) = g(\sum_k w_{jk} x_k)$$

Índice	Descripción
i	índice de la neurona de la capa de salida (o siguiente)
j	índice de la neurona de la capa intermedia
k	índice de la neurona de entrada o de la capa anterior
m	índice de la capa intermedia
p	cantidad datos
$\mu$	dato en particular

$w_{jk}^m$  = pesos sinápticos

$V_j^m$  = neurona de capa intermedia

$W_{ij}$  = pesos sinápticos de la última capa

$O_i$  = neurona de capa de salida

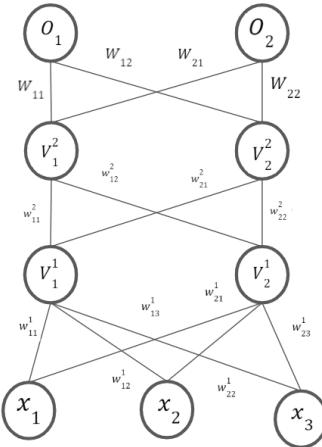


Figura 7.2: Perceptrón multicapa

$$O_i = g(h_i) = g\left(\sum_j W_{ij} V_j\right)$$

El error producido por toda la red multicapa es

$$\begin{aligned} E(w) &= \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - O_i^\mu)^2 \\ &= \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - g\left(\sum_j W_{ij} V_j^\mu\right))^2 \\ &= \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - g\left(\sum_j W_{ij} g\left(\sum_k w_{jk} x_k^\mu\right)\right))^2 \\ &= \frac{1}{2} \sum_{\mu=1}^p \sum_{i=1}^2 (\zeta_i^\mu - g\left(\sum_j W_{ij} g\left(\sum_k w_{jk} x_k^\mu\right)\right))^2 \end{aligned}$$

Para computar el error entonces, se parte de la actualización para la última capa y se retropropaga el error  $(\zeta_i^\mu - O_i^\mu)$  hacia las capas anteriores.

La actualización de los pesos  $\Delta W_{ij}$  en la última capa es directa:

$$\begin{aligned} \Delta W_{ij} &= -\eta \frac{\partial E}{\partial W_{ij}} \\ \Delta W_{ij} &= -\eta (\zeta_i^\mu - O_i^\mu) g'(h_i^\mu) V_j^\mu \\ \Delta W_{ij} &= -\eta \delta_i^\mu V_j^\mu \end{aligned}$$

con  $\delta_i^\mu = (\zeta_i^\mu - O_i^\mu) g'(h_i^\mu)$ . De acá podemos usar esta información para retropropagarla para atrás a las capas anteriores:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} \quad (7.8)$$

$$\begin{aligned}
\frac{\partial E}{\partial w_{jk}} &= \frac{\partial \frac{1}{2} \sum_{\mu,i} (\zeta_i^\mu - O_i^\mu)^2}{\partial w_{jk}} \\
&= \frac{\partial \sum_{\mu,i} (\zeta_i^\mu - O_i^\mu)}{\partial w_{jk}} \\
&= \sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) \left( -\frac{\partial O_i}{\partial w_{jk}} \right) \\
&= -\sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) g'(h_i) \frac{\partial h_i}{\partial w_{jk}} \\
&= -\sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) g'(h_i) \frac{\partial (\sum_j W_{ij} V_j)}{\partial w_{jk}} \quad \text{Cero } \forall \text{ términos excepto } ij = jk \\
&= -\sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) g'(h_i) W_{ij} \frac{\partial V_j}{\partial w_{jk}} \\
&= -\sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) g'(h_i) W_{ij} g'(h_j) W_{jk} x_k^\mu
\end{aligned}$$

Retomando en 7.8:

$$\Delta w_{jk} = \eta \sum_{\mu,i} (\zeta_i^\mu - O_i^\mu) g'(h_i) W_{ij} g'(h_j) W_{jk} x_k^\mu$$

Teniendo en cuenta la definición de  $\delta$  para la última capa

$$\delta_i^\mu = (\zeta_i^\mu - O_i^\mu) \cdot 1 \cdot g'(h_i) \quad (7.9)$$

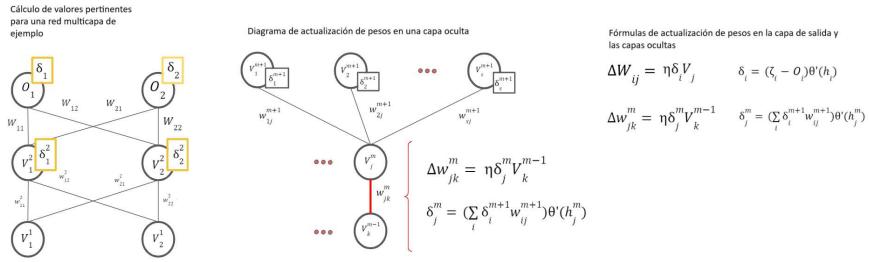
nos queda:

$$\Delta w_{jk} = \eta \sum_{\mu,i} \delta_i^\mu W_{ij} g'(h_j) W_{jk} x_k^\mu$$

Podemos extender entonces la definición de  $\delta$  para una capa hacia atrás también como

$$\delta_j^\mu = g'(h_j) \cdot \sum_i W_{ij} \delta_i^\mu$$

y retomamos entonces la regla 7.8 con lo que nos queda



$$\begin{aligned}\Delta w_{jk} &= \eta \sum_{\mu i} \delta_i^\mu W_{ij} g'(h_j^\mu) x_k^\mu \\ \Delta w_{jk} &= \eta \sum_{\mu} \sum_i \delta_i^\mu W_{ij} g'(h_j^\mu) x_k^\mu \\ \Delta w_{jk} &= \eta \sum_{\mu} g'(h_j^\mu) \sum_i W_{ij} \delta_i^\mu x_k^\mu \quad g'(h_j^\mu) \text{ no depende de } i \\ \Delta w_{jk} &= \eta \sum_{\mu} (g'(h_j^\mu) \sum_i W_{ij} \delta_i^\mu) x_k^\mu \\ \Delta w_{jk} &= \eta \sum_{\mu} \delta_j^\mu x_k^\mu\end{aligned}$$

Retomando entonces (ufff), la fórmula de la ecuación de la última capa es

$$\Delta W_{ij} = -\eta \delta_i^\mu V_j^\mu$$

en tanto que la fórmula para la capa oculta es

$$\Delta w_{jk} = \eta \sum_{\mu} \delta_j^\mu x_k^\mu.$$

Pero lo más importante es lo que generamos en el medio:

$$\delta_j^\mu = g'(h_j^\mu) \cdot \sum_i W_{ij} \delta_i^\mu$$

Esta fórmula es muy importante porque permite retropropagar el error en una capa cualquiera a la capa inmediatamente de atrás. De ahí el nombre de **retropropagación**. La arquitectura de conexiones, se replica mediante los errores hacia atrás. La fórmula general de la actualización de los pesos queda:

$$\Delta W_{ij} = \eta \sum_{\mu} \delta_i^\mu V_j^\mu \tag{7.10}$$

y si se actualiza un sólo patrón

$$\Delta W_{ij} = \eta \delta_i^\mu V_j^\mu \tag{7.11}$$

Un aspecto clave para la eficiencia computacional de este algoritmo, es que la actualización de los pesos depende de información de entrada  $V_j^\mu$  y  $\delta_i^\mu$  que es propia de esa capa.

### 7.1.3 Aprendizaje Incremental

En la forma incremental, cada vez que una entrada es presentada a la red, se propaga dicha entrada hasta obtener los estados de activación de la capa de salida. Luego, se retropropaga el error obteniendo todos los  $\delta$  correspondientes para cada unidad de cada capa. Con toda esa información se puede disparar una actualización (en paralelo) de todos los pesos. Esto se repite para cada entrada  $\mu$ . Cuando se completan todas las entradas del conjunto de entrenamiento, se ha completado una **época**. El sampling que se hace de los estímulos para presentarselo a la red tiene que ser al azar.

### 7.1.4 Aprendizaje por lotes

El aprendizaje por *batches* o lotes, cada vez que una entrada es presentada a la red, se propaga dicha entrada hasta obtener los estados de activación de la capa de salida. Luego, se retropropagan el error obteniendo los  $\delta$  para cada neurona de cada capa. Pero ahora los  $\Delta w$  se acumulan y se repite este proceso para todas las entradas del conjunto. Al terminar la época, los valores acumulados se impactan en los pesos. En este caso el orden no es tan relevante (pero aún así es mejor tomarlos al azar).

### 7.1.5 Perceptrón Multicapa

Vamos entonces con el algoritmo del Perceptrón multicapa.

---

#### Algoritmo 6 Algoritmo del Perceptrón Multicapa

---

```

1: Inicializar  $i = 0$ 
2: Inicializar  $w = initializeWeights()$ 
3: Inicializar  $error = 1$ 
4: Inicializar  $error_{min} = INTEGERMAX$ 
5: Inicializar  $w_{min} = 1$ 
6: Especificar hiperparámetros COTA, EPSILON.
7: while  $error_{min} > EPSILON \wedge i < COTA$  do
8:    $i_x = ceil(rand(1, 1) * p)$ 
9:   Obtener  $\mu = rand(0, p)$ 
10:   $O^\mu = forwardPropagation(\mu)$ 
11:   $\Delta w = backPropagation()$ 
12:   $w = w + \Delta w$ 
13:   $error = computeError(data, w);$ 
14:  if  $error < error_{min}$  then
15:     $error_{min} = error$ 
16:     $w_{min} = w$ 
17:  end if
18:   $i = i + +$ 
19: end while
20:  $w$  Resultado de los pesos entrenados (si es posible) para el dataset  $x$ .
```

---

Con la definición de las dos funciones para calcular la propagación adelante y hacia atrás.

---

#### Algoritmo 7 forwardPropagation()

---

```

1:  $V_k^0 = x_k^\mu$ 
2:  $V_i^m = g(h_i^m) = g(\sum_j w_{ij}^m V_j^{m-1})$ 
3: Retorna  $O^\mu = V_i^m$  cuando  $m$  es la última capa de la red.
```

---

**Algoritmo 8** backPropagation()

- 
- 1:  $\delta_i^M = g'(h_i^M)(\zeta_i^\mu - V_i^M)$  para la última capa.
  - 2:  $\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m$  para todo  $m$  entre  $M$  y 2.
  - 3: Retorna  $\Delta w_{ij}^m = \eta \delta_j^m V_k^{m-1}$
- 

El Perceptrón multicapa, con el algoritmo de backpropagation, plantea un método bioinspirado eficiente de aprendizaje automático 9.1.1, de ajuste de parámetros libres para optimizar una función de error, lo que implica poder de manera supervisada encontrar los pesos que generan para un conjunto de entrada las salidas esperadas. Esto es muy poderoso.

Algunos puntos para identificar:

- El cálculo de la señal de error es efectivamente una propagación hacia atrás haciendo uso de las matrices traspuestas de pesos y pasando de una  $\delta_i^m$  a una  $\delta_i^{m-1}$ .
- Los pesos sinápticos de la primera capa no contribuyen a propagar la señal de error para ajustar los pesos de la primera capa (sí para el cálculo del  $\Delta w_{ij}^1$ ).
- La función particular de costo solo afecta el cálculo de la señal de error  $\delta_i$  de la última capa. Por la regla de la cadena, el resto se obtiene del algoritmo de backpropagation.

## 7.2 Arquitectura

A partir de que es posible aglomerar las neuronas, aparece la idea y el concepto de arquitectura de la red neuronal: ¿Cuál es la arquitectura más adecuada para resolver un problema en particular, o una familia de problemas? Esto es un arte más que una ciencia<sup>1</sup> sin una receta clara.

En MLPs la arquitectura suele representarse en base a la cantidad de capas, y la cantidad de neuronas en cada capa. Un buen tip es siempre considerar como capa lugares en la red donde se pueden ubicar pesos, y que comparten las entradas/salidas. Esto provoca que una red con una sola capa oculta, tenga dos capas. La primera corresponde a la capa oculta que recibe las entradas directamente, y la segunda toma las salidas de esa capa intermedia, las pasa por pesos de una segunda capa que generan una salida uni- o multidimensional. Sin embargo, a este esquema hay que agregarle, para completar la arquitectura, la dimensión de los datos de entrada. Luego se pueden sumar todos los pesos y con eso se determina la cantidad de parámetros libres de la red, inducido por la arquitectura.

La arquitectura se transforma en un hiperparámetro extra a definir. Por eso, es entonces un problema en sí mismo para el cuál se han probado un sinnúmero de soluciones. La arquitectura de la red es un componente clave porque de seleccionar la adecuada, aumentan las chances de poder converger los pesos y de tener más rendimiento en la generalización del problema. Pero justamente encontrar esa arquitectura no es un proceso sencillo ni tecnificado.

De esto se hace evidente la importancia de los estudios de ablación [64]. De ese menjúnje al que se llega de arquitectura, más otros hiperparámetros, cuáles podemos eliminar, reducir, y mantener el mismo nivel de eficiencia en la red. En el medio del hype del área actual este es un tema crítico porque permite eliminar la complejidad extra agregada sólo por la necesidad de corresponder con ese hype: porque no vender una gran red con una gran arquitectura cuando un simple perceptrón multicapa bastaría para solucionar el problema.

En definitiva, es prueba y error y adaptabilidad a cada uno de los problemas particulares en los que se usa la red.

---

<sup>1</sup>Este es un popular eufemismo *cliché* común también en otras disciplinas para no decir que en realidad no tenemos ni idea cómo se hace.

### 7.3 Optimización en Redes Neuronales

El proceso de optimización en la red depende del problema<sup>2</sup> y el esquema planteado en la versión vanilla del Perceptrón multicapa puede no ser suficiente para alcanzar convergencia en un tiempo razonable. Retomando entonces lo visto en el capítulo 5, es posible expandir los métodos allí planteados con el objetivo específico de la aplicación en redes neuronales.

El entrenamiento, la optimización, el ajuste de los parámetros se basa en

$$w_{(t+1)} = \gamma w_{(t)} + \alpha \Delta W \quad (7.12)$$

donde como se aclaró previamente  $\alpha$  es la tasa de aprendizaje, pero  $\gamma$  puede verse como el factor de olvido [65].

#### 7.3.1 Momentum

Esta variante a la regla de actualización de los pesos, que como se menciona en el capítulo 5 es una variante al método de gradiente descendente, mitiga el inconveniente de la oscilación en torno al mínimo o la divergencia que suele pasar cuando el  $W$  está cerca de un mínimo local o global.

$$\Delta w_{ij}(t+1) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t) \quad (7.13)$$

con  $\alpha = 0.8$  o  $\alpha = 0.9$ . Si la función  $E(\cdot)$  está en una región plana,  $\Delta w_{ij}(t)$  será similar a  $\frac{\partial E}{\partial w_{ij}}$  y por lo tanto  $\Delta w_{ij}(t+1)$  se incrementará en módulo. Por el contrario, si la función  $E(\cdot)$  está en un valle que determinaría un comportamiento más oscilatorio,  $\Delta w_{ij}(t+1)$  tenderá a compensarse para evitar oscilaciones.

#### 7.3.2 $\eta$ adaptativo

En el algoritmo de gradiente descendente, el valor de  $\eta$  determina el paso, el salto entre un valor y el siguiente. No es bueno que este valor sea constante, porque si  $E(\cdot)$  decrece en un momento del aprendizaje (por ejemplo durante  $k$  iteraciones seguidas), quizás sería interesante incrementarlo para que vaya más rápido. Por otro lado, si  $E(\cdot)$  comienza a incrementarse en un momento del aprendizaje quizás sería positivo decrementar  $\eta$  para evitar la divergencia

$$\Delta \eta = \begin{cases} +a & \text{si } \Delta E < 0 \text{ en forma consistente,} \\ -b\eta & \text{if } \Delta E > \text{empieza a ser consistente} \\ 0 & \text{en cualquier otro caso} \end{cases} \quad (7.14)$$

También es posible actualizar el  $\eta$  por neurona particular (para los pesos de las conexiones que llegan a esa neurona) o por conexión (un  $\eta$  para cada  $w_{ij}$ ).

#### 7.3.3 RMSProp

Este algoritmo fue propuesto por Hinton en 2012 [66]. Es sustancial para entender Adam ya que este es la unión de RMSProp con el agregado de momento. La intuición del algoritmo es ajustar las tasas de aprendizaje por cada uno de los pesos, según el que más convenga para evitar oscilaciones sucesivas, y a la vez acelerar la convergencia. Para ello se definen las Ecuaciones 7.15 y 7.16:

<sup>2</sup>Acá todo depende siempre del problema.

$$g_t = \frac{\partial E}{\partial w_{ij}} \quad (7.15)$$

$$S_t = \gamma S_{t-1} + (1 - \gamma) g_t^2 \quad (7.16)$$

El término  $g_t^2$  en la ecuación 7.3.4 corresponde a la operación de cuadrados aplicada miembro a miembro del vector  $g_t$ .

$$\Delta w_{ij} = -\frac{\eta}{\sqrt{S_t + \epsilon}} \times g_t \quad (7.17)$$

El parámetro  $\epsilon$  es agregado para evitar dividir por cero. La idea del algoritmo es ajustar el learning rate de acuerdo al root mean square (RMS). Teniendo presente que este es un proceso de optimización, el learning rate es el paso, el salto que hay que dar. En este caso al promediar el learning rate se está tratando de encontrar un tamaño de paso crítico que sirva para todo el proceso y que balancee performance versus convergencia. Por ejemplo, un learning bajo, se amortiguan mejor los cambios, se reducen las oscilaciones, en tanto que un learning rate alto, aprende más rápido.

El valor del RMS viene dado por el cálculo iterativo del promedio pesado de los  $g_t$  anteriores, según

$$\begin{aligned} S_t &= (1 - \gamma) g_t^2 + \gamma S_{t-1} \\ &= (1 - \gamma) g_t^2 + \gamma((1 - \gamma) g_{t-1}^2 + \gamma S_{t-2}) \\ &= (1 - \gamma) g_t^2 + (1 - \gamma) \gamma g_{t-1}^2 + \gamma^2 S_{t-2} \\ &= (1 - \gamma) g_t^2 + (1 - \gamma) \gamma g_{t-1}^2 + \gamma^2 ((1 - \gamma) g_{t-2}^2 + \gamma S_{t-3}) \\ &= (1 - \gamma) g_t^2 + (1 - \gamma) \gamma g_{t-1}^2 + \gamma^2 (1 - \gamma) g_{t-2}^2 + \gamma^3 S_{t-3} \\ &= (1 - \gamma)(g_t^2 + \gamma g_{t-1}^2 + \gamma^2 g_{t-2}^2 + \dots). \end{aligned}$$

### 7.3.4 Adam

En 2014, se propuso una actualización al algoritmo de RMSProp que justamente combinaba las características de este approach pero incluyendo momento. Este algoritmo, bautizado Adam[67], es actualmente el más utilizado en Deep Learning porque justamente balancea eficiencia y convergencia.<sup>3</sup>.

El núcleo del algoritmo viene dado por los pasos establecidos en el Algoritmo 9:

$$\begin{aligned} m_t &\leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &\leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{m}_t &\leftarrow \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &\leftarrow \frac{v_t}{1 - \beta_2^t} \\ \theta_t &\leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

<sup>3</sup>El paper de Kingma tiene casi 200000 citas, un número exorbitante que muestra la efectividad de este algoritmo.

La corrección del sesgo se realiza para compensar los cálculos realizadas en las primeras iteraciones, que vienen por el cálculo de los estimadores de los momentos. Este algoritmo tiene una asignación un tanto arbitraria pero con una prueba empírica en el propio paper, de hiperparámetros que se conoce que son buenos para las implementaciones donde la función objetivo es una función de error. Estos son  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$  y  $\epsilon = 10^{-8}$ .

---

**Algoritmo 9** Adam Optimization Algorithm

---

- 1: Inicializar parámetros:  $\theta$ , tasa de aprendizaje  $\alpha$ , tasas decaimiento  $\beta_1, \beta_2$ , y la constante  $\epsilon$
  - 2: Inicializar el primero y el segundo momento:  $m_0 = 0, v_0 = 0$
  - 3:  $f(\theta)$ : Función estocástica de parámetros  $\theta$
  - 4: Inicializar  $t = 0$
  - 5: **while** no converge **do**
  - 6:      $t \leftarrow t + 1$
  - 7:     Calcular el gradient:  $g_t \leftarrow \nabla_{\theta} f(\theta_{t-1})$
  - 8:     Calcular el estimador sesgado del primer momento:  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1)g_t$
  - 9:     Calcular el estimador sesgado del segundo momento:  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$
  - 10:    Recalcular el primer momento:  $\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}$
  - 11:    Recalcular el segundo momento:  $\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$
  - 12:    Actualizar los parámetros:  $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
  - 13: **end while**
  - 14:  $\theta_t$  Resultado parámetro
- 

Resumiendo, este algoritmo se encuentra **dentro** del propio proceso de optimización y aprendizaje (no hay ningún *loop* interno dentro de Adam en sí). Así como ocurre con RMSProp, Adam realiza la actualización peso a peso (las operaciones son vectoriales).



## 8. Métricas de Evaluación

### 8.1 Identificar el patrón de manera correcta

Las redes MLP del capítulo 7 pueden actuar como un **Clasificador**. Esto es, la red fue expuesta a un conjunto de datos, que conformaron el conjunto de aprendizaje, y pudo en algún punto identificar un patrón subyacente, que le permite al ver datos nuevos, pero que son del mismo dataset, inferir de manera adecuada la etiqueta correspondiente a cada muestra.

Surge la pregunta ¿qué es clasificar correctamente? y ¿cuántos elementos eventualmente clasifica correctamente?

De acá surge la necesidad de establecer métodos estándar para evaluar el desempeño de un clasificador. Esto permite, comparar diferentes clasificadores, comparar el mismo clasificador sobre diferentes conjuntos de datos, y evaluar los cambios en los hiperparámetros en relación a la performance<sup>1</sup>.

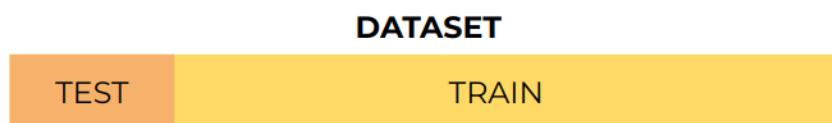


Figura 8.1: El set de datos, la matriz  $X$  se puede al menos partir en dos partes, entrenamiento y test.

### 8.2 Métricas estándar

Lo que se busca de mínima es que el clasificador **generalice** al set de datos. Para esto el dataset (Fig. 8.1) se divide en dos: una parte se utiliza para **converger** los parámetros con el conjunto de entrenamiento (*train set*), y luego se usa el grupo remanente (*test set*) para evaluar cuán bien el clasificador pudo capturar el patrón y obtener buenos resultados de reconocimiento.<sup>2</sup>

<sup>1</sup>Una excelente exposición introductoria puede verse en el capítulo 5 de [68]

<sup>2</sup>Si el dataset es lo suficientemente grande, se puede dividir en tres partes, agregando un *validation set* para evaluar el desempeño del modelo en diferentes configuraciones de hiperparámetros antes de su evaluación final en el *test set*.

Esto suscita dos problemas. El primero es cómo evaluar si esta división es o no la adecuada. El segundo problema es cómo evaluar la capacidad de generalización de la red.

### 8.2.1 Matriz de Confusión

La matriz de confusión sirve para establecer una métrica de evaluación de la performance de un clasificador binario/multiclasses y se construye en base a listar en las columnas las clases predichas en tanto que en las filas van las clases verdaderas. Luego se enumeran los elementos del grupo de datos de **test** según su rótulo verdadero y el rótulo predicho por el clasificador.

Predicciones	Positivo?	Negativo?
Positivo	TP	FN
Negativo	FP	TN

Cuadro 8.1: Matriz de Confusión

Esta terminología surge del ambiente médico, por eso se hace referencia a positivo y negativo como el resultado de un test bioquímico. Son TP, True Positive, aquellos ejemplos que el predictor marcó como POSITIVO, y que resultaron ser verdaderamente positivos. Es decir los casos donde acertó en detectar, llamemosle, la patología. Por ejemplo en este caso, este valor queremos que sea lo más alto posible, para que el predictor no se pierda ningún positivo, que al menos los identifique a todos los que hay. El segundo valor que queremos mantener alto es TN, True Negative, los resultados que verdaderamente eran negativos y que a su vez el predictor los identificó como tal. Este valor representa la capacidad del predictor de separar la paja del trigo. Los otros dos valores, FP y FN son los falsos positivos y los falsos negativos, que son aquellos POSITIVOS y NEGATIVOS que marcó MAL el predictor. Por supuesto, estos queremos que sean lo más bajo posible.

De esto surge entonces, la medida más popular de Accuracy, o Performance.

**Definition 8.2.1 — Accuracy.** La performance o desempeño de un clasificador se puede obtener mediante

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \quad (8.1)$$

En el caso de que la matriz de confusión es diagonal, el clasificador es perfecto y alcanza una performance de 1. Es importante tener en cuenta, que la peor performance, cuando el clasificador es binario, no es 0, sino  $\frac{1}{2}$  que es la que corresponde a un clasificador que simplemente tira una moneda para determinar la clase.

La matriz de confusión suele generalizarse a clasificación multiclas, donde se suelen usar heatmaps

### 8.2.2 Métricas

A Accuracy se suman otras métricas estandarizadas que aportan información sobre el desempeño de un clasificador desde diferentes perspectivas.

**Definition 8.2.2 — Precision.** Representa cuán bien acertó los que acertó en relación al problema.

$$Precision = \frac{TP}{TP + FP} \quad (8.2)$$



Figura 8.2: Ejemplo de una matriz de confusión multiclasses.

**Definition 8.2.3 — Recall.** Cuán bien el predictor atrapó a todos los que en definitiva eran positivos .

$$\text{Recall} = \frac{TP}{TP + FN} \quad (8.3)$$

**Definition 8.2.4 — F1-Score.** La performance o desempeño de un clasificador se puede obtener mediante

$$F1 - Score = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (8.4)$$

### 8.2.3 Ajuste, subajuste y sobreajuste

Al entrenar el clasificador, se busca **converger**, es decir utilizar de manera adecuada los datos de entrada para **ajustar** los parámetros libres de forma que la función de costo/optimización alcance el mínimo (fig. 8.3). Cuando esto no ocurre, es decir se está en algún mínimo local, o ni siquiera, el clasificador se dice que está **underfitted** o subajustado. Por el contrario, cuando el clasificador está sobreentrenado, se presenta la situación de **overfitting** donde el mínimo encontrado sigue la curva del ruido en los datos, por lo que se ajusta de manera excesiva a las variaciones del conjunto de entrenamiento y es incapaz de identificar verdaderamente el patrón: falla en **generalizar**<sup>3</sup>.

Normalmente en un MLP (o similar) que actúa como clasificador, la situación de subajuste se visualiza porque el desempeño de la red es alto para el conjunto de entrenamiento y hay una tendencia en el entrenamiento que a medida que aumenta el entrenamiento el desempeño sobre los datos de test aumenta también, como se ve en la Figura 8.4a. Por otro lado, al incurrir en una situación de sobreajuste ocurre lo contrario y a mayor entrenamiento el desempeño sobre los datos de test se reduce hasta perder toda fuerza predictiva. El sobreajuste es un gran **cucó** porque puede darse por problemas en los datos, como balanceo de clases, datos con excesivo ruido, o incluso

<sup>3</sup>Tomaremos esta definición, aún cuando el concepto de overfitting es muy abusado en la jerga de aprendizaje automático, ya que suele ser una consideración posthoc cuando el método falla en la generalización [69].

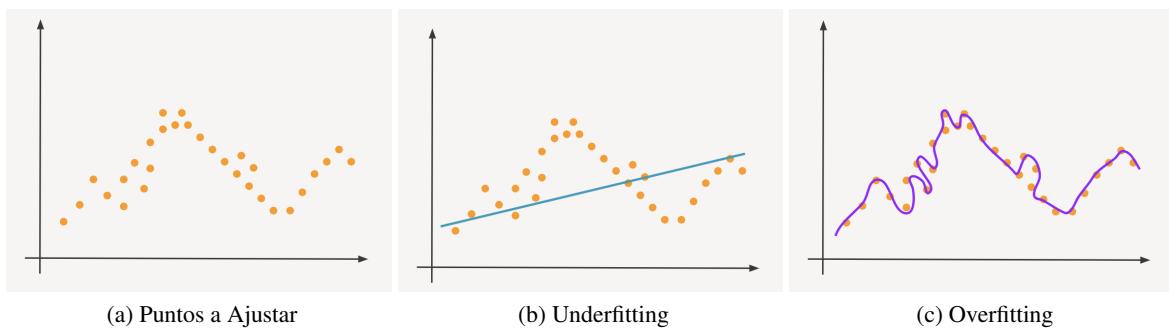


Figura 8.3: Niveles de ajuste sobre un conjunto de puntos. Underfitting se da cuando la curva de ajuste no capture bien el patrón inherente en los puntos (en este caso una curva). Por el contrario, cuando el ajuste es milimétrico, la curva captura el ruido natural en los datos y pierde fuerza de generalización.

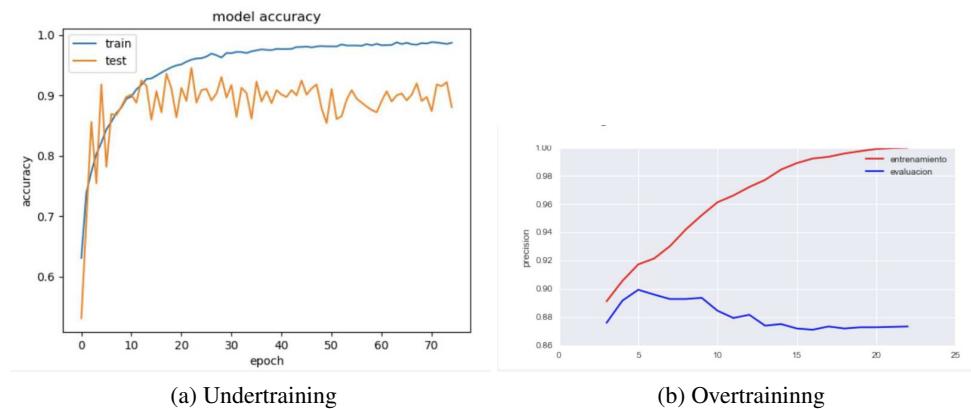


Figura 8.4: Curvas de desempeño por época para MLPs que incurren en subentrenamiento o sobreentrenamiento.

insuficiente cantidad de datos para discernir el patrón subyacente.

#### 8.2.4 Convalidación Cruzada

La partición sugerida implica la percepción de que los datos son altamente homogéneos, es decir que la elección de cómo establecer esa partición no impactará en los resultados. Esto es raramente así, por lo que existen métodos de validación que apuntan a eliminar el sesgo en la selección arbitraria de como partir los datos en entrenamiento y test.

El método mas utilizado para esto, es **k-Fold Cross Validation**

#### k-Fold Cross Validation

Este método es muy simple y lo que se hace es:

1. Iterar  $T$  veces (normalmente  $T = k$ )
2. Dividir el dataset en  $k$  partes iguales
3. Tomar  $k - 1$  partes y usarlas para entrenar el clasificador
4. Usar como grupo test la parte remanente que **no** se uso para entrenar
5. Evaluar la performance sobre esa parte
6. Iterar

Finalmente se obtienen parámetros de desempeño del promedio de las  $T$  iteraciones. Este método ayuda a evitar cualquier sesgo que pueda presentarse en la división del dataset. Adicionalmente,

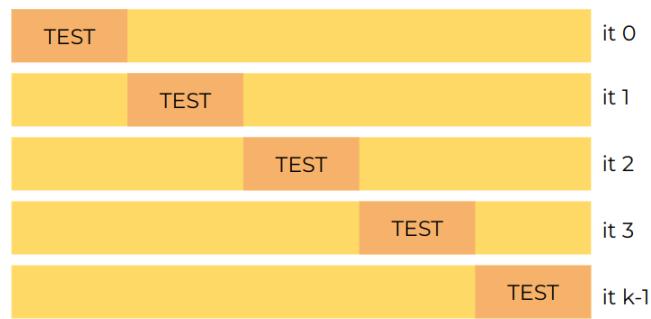


Figura 8.5: En este caso, el dataset se divide e 5 partes donde el rol de test se va alternando en cada una.

si se evalua el desvío de las métricas se puede generar una métrica de la homogeneidad en los propios datos.



# IV Automatizando el Aprendizaje

<b>9</b>	<b>Aprendizaje Automático .....</b>	<b>91</b>
9.1	Redefiniendo Aprendizaje	
9.2	Métodos de Aprendizaje	
9.3	Pipeline de Machine Learning	
<b>10</b>	<b>Transformaciones en los datos .....</b>	<b>95</b>
10.1	Estructura interna de los datos	
<b>11</b>	<b>Análisis de Componentes Principales .</b>	<b>99</b>
11.1	Preliminares	
<b>12</b>	<b>Red de Kohonen .....</b>	<b>103</b>
12.1	Mapa Autoorganizado	
<b>13</b>	<b>Red de Hopfield .....</b>	<b>107</b>
13.1	Hopfield 1982	
<b>14</b>	<b>Modelo de Oja y Sanger .....</b>	<b>113</b>
14.1	La búsqueda del poder de síntesis	
14.2	Red de Oja	
14.3	Red de Sanger	





## 9. Aprendizaje Automático

### 9.1 Redefiniendo Aprendizaje

El área de redes neuronales cae dentro de la frontera de lo que se denomina **Aprendizaje Automático** o *Machine Learning*. Puede definirse como:

**Definition 9.1.1 — Aprendizaje Automático.** Técnicas para identificar relaciones entre datos ( $X$ ) o mapeos ( $X \rightarrow Y$ ) basadas en algoritmos libres de modelo que dada una serie de parámetros libres que estructuran esas relaciones o mapeos, pueden ajustarse mediante algún proceso de optimización matemática [24].

Para darle encuadre a esta definición, es importante remarcar el aspecto **libre de modelo** de la definición. Esto quiere decir, que el esquema algorítmico a implementar no asume ningún lineamiento específico en relación a la existencia e interacción de los componentes del problema. Es decir, no hay establecido ningún modelo, que generalmente es cuantitativo y matemático, sobre cómo los diferentes elementos del sistema interactúan entre si.

Esto que parece una huevada, es un bomba, y es realmente un cambio radical como metodología general para abordar problemas del mundo real. Por ejemplo, tenemos un cañón del cuál tenemos solo un parámetro libre que es el ángulo de tiro. Diferentes ángulos van a provocar diferentes distancias donde va a caer la bala. Y por supuesto, lo que queremos hacer es predecir donde va a caer la bala, dado un ángulo particular del cañón. La física Newtoniana básica nos da una herramienta, un modelo de la mecánica y dinámica de ese movimiento, el tiro oblicuo, que podemos utilizar para resolver cuantitativamente el problema: vamos a poner un ángulo, hacemos la cuenta, y nos da donde va a caer. Sin embargo, el abordaje utilizando machine learning, tira por la borda ese modelo, y se limita a disparar el cañón muchas veces, medir la distancia, repetir ese proceso varias veces, construir un dataset, y con eso entrenar un modelo de ML<sup>1</sup> que nos va a permitir resolver el mismo problema: predecir.

La definición 9.1.1 es la más abarcativa y la que vamos a adoptar en este texto. De aquí surge

<sup>1</sup>Fijense que usamos la misma palabra para dos cosas distintas: modelo del mundo que mecanísticamente simplifica su accionar, y también usamos modelo para describir un esquema de ajuste de parámetros libres optimizando una función de costo en base a un conjunto de datos. Alguién debería inventar una nueva palabra.

entonces el concepto de **Aprendizaje** (que queremos decir cuando decimos que una computadora aprende): aprende cuando se ejecuta lo establecido en la definición 9.1.1.

## 9.2 Métodos de Aprendizaje

Siendo cruel, banal e imperdonable con el área, una manera pragmática de describirla es como una bolsa de gatos de métodos. Pero todos caen en la definición 9.1.1, es decir tienen algún tipo de clavija libre que puede ajustarse para optimizar algún funcional [68]. La figura 9.2 muestra las diferentes ramas principales en las que se agrupan todos los métodos. La primera división tiene que ver con los métodos bioinspirados/biomiméticos y los que no lo son. Hoy (circa 2024) esta división coincide con los métodos basados en redes neuronales multicapas profundas y los que no lo son, que por yuxtaposición pueden llamarse *Shallow* o superficiales. En estos, que son los métodos tradicionales de aprendizaje automático, se pueden incluir cuatro categorías principales que son los frequentistas, los bayesianos, los *if++*, geométricos y los transformacionales. Los métodos frequentistas se basan en estadística matemática y son los herederos del análisis estadístico: La regresión lineal, regresión logística (logit), LDA Linear Discriminant Analysis. Luego, los enemigos cantados de los primeros, los bayesianos, se basan en la fórmula de Bayes y la posibilidad de realizar inferencia causal utilizando la capacidad de poder iterar esta ecuación: desde **Naive Bayes** hasta **EM**. Luego están aquellos métodos que utilizan condicionales con umbrales libres que actúan de parámetros como los **Decision Trees** (árboles de decisión) y **Random Forest**. Le siguen los métodos basados en conceptos de álgebra lineal y su geometría como k-NN **K-Near Neighbours** (vecinos cercanos), el gran SVM Support Vector Machine, **DBSCAN**, **k-means** y sus variantes. Esta humilde descripción taxonómica culmina con los transformacionales que son métodos que buscan generar una transformación de los datos optimizando alguna función de costo que busque alguna reorganización de los datos y su agrupamiento, como PCA (capítulo 11), SVD (capítulo 17), y otros métodos de reducción de dimensionalidad como t-SNE.



Figura 9.1: Bolsa de gatos de métodos variopintos donde todos tienen parámetros libres a ajustar.

## 9.3 Pipeline de Machine Learning

De acá surge el modelo de la figura 9.3. Este modelo es pervasivo a cualquier implementación de un sistema que entre dentro de la categoría de aprendizaje automático. Primero, del mundo real, de la realidad, se realiza una medición. Esta medición puede involucrar un proceso manual o incluso un proceso automático (Automatización). Luego hay un proceso de adquisición de una señal, que involucra también la propia digitalización de la información. Esto es convertir la información fenomenológica en datos digitales en una computadora. A partir de ese punto comienza lo que se llama el **pipeline** de machine learning. El primer punto es una tarea de preprocesamiento que es ajustar la información, corregirla, recuperar datos perdidos o faltantes. Y sobre todo ajustar los datos para lo que viene después. El segundo paso del pipeline está concentrado en el armado de una síntesis, una compresión de los datos que capture numéricamente el patrón en un vector que luego se quiere detectar y generalizar. Este aspecto es central en aprendizaje automático y es donde se concentra lo que se denomina **feature engineering**. El paso final tiene que ver con el fin de todo el modelo que puede ser una de los cuatro objetivos como clasificar, regresionar, optimizar y generalizar. Luego del pipeline, el insight generado por el proceso es utilizado para un sistema de aplicación, que puede afectar ese mismo ambiente de estudio en un ciclo cerrado.

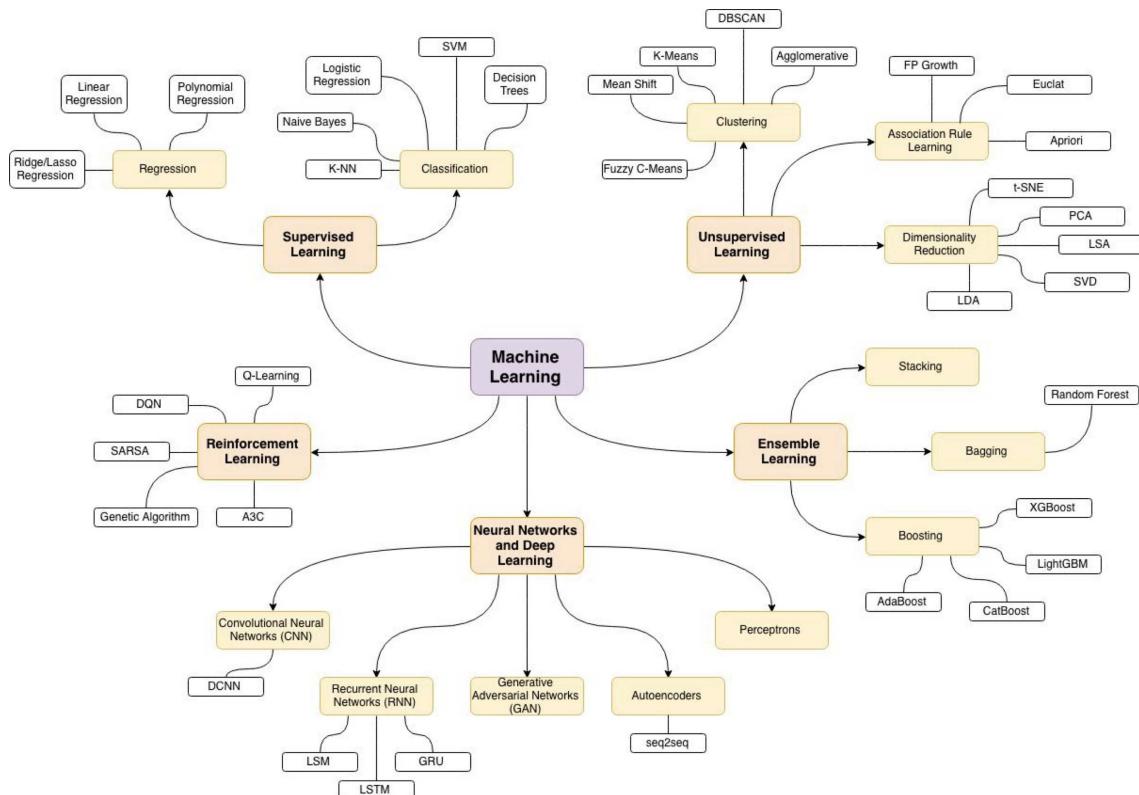


Figura 9.2: Aprendizaje Automático y las áreas asociadas.

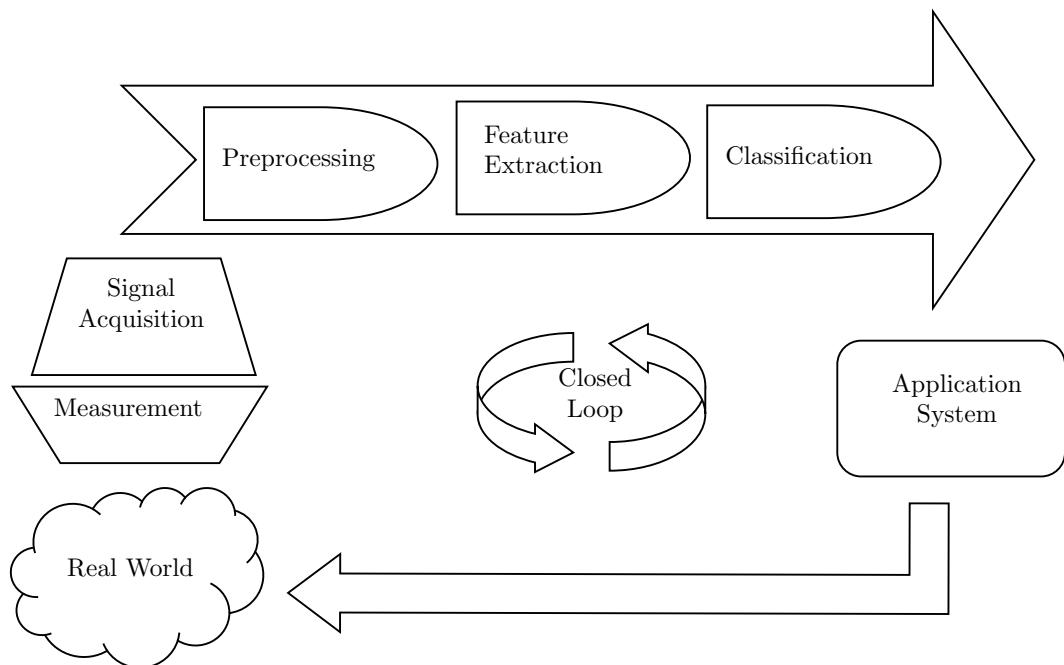


Figura 9.3: Arquitectura General de un Sistema de Aprendizaje Automático

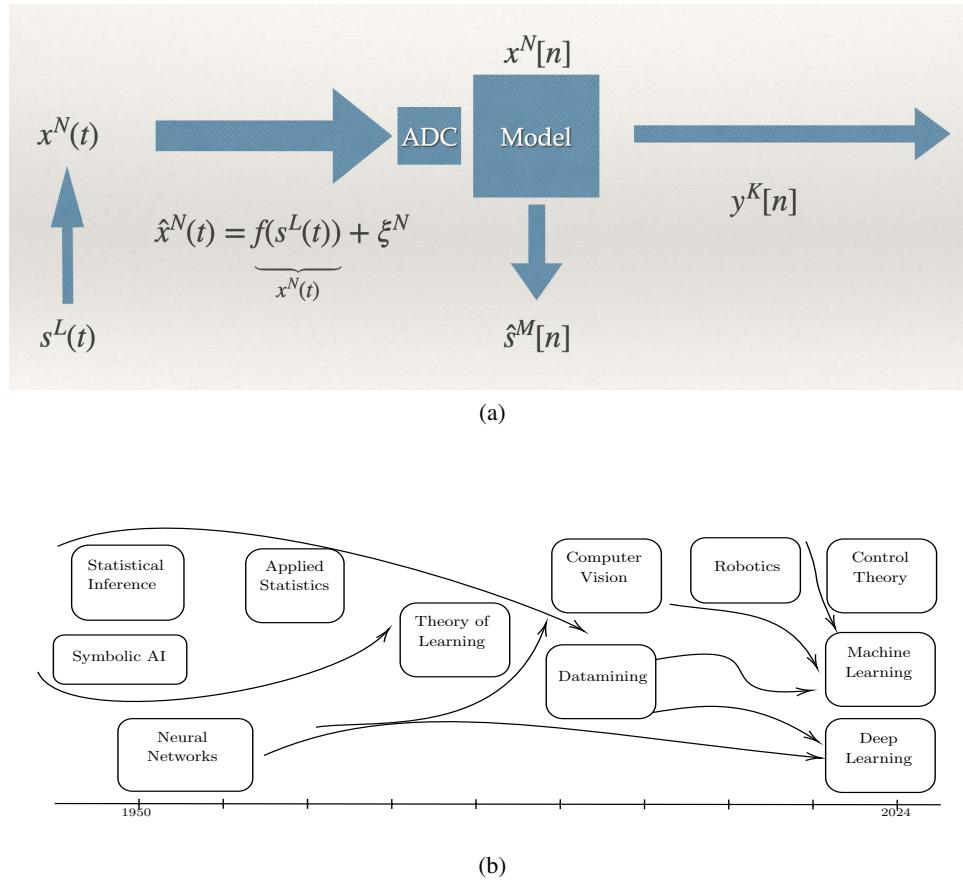


Figura 9.4: Problema inverso y evolución histórica de las técnicas de aprendizaje automático.

### 9.3.1 El problema Inverso

La figura 9.4a representa la estructura del problema inverso de ingeniería, y es un punto crítico a tener en cuenta en cualquier implementación de aprendizaje automático o inteligencia artificial. El fenómeno natural que queremos estudiar tiene una representación intrínseca en  $L$  señales continuas analógicas. Esto puede ser cualquier cosa, pero son las fuentes que nos interesa entender para poder identificar algún patrón y luego poder generalizar. Ese fenómeno es capturado por  $N$  transductores que reciben una representación de las fuentes que no es exactamente la misma información, sino que está transformada por la función vectorial  $x^N(t) = f(s^L(t))$ . Este proceso tiene además un inexorable error en la propia transducción con lo que la señal que realmente se genera es  $\hat{x}^N(t) = f(s^L(t)) + \xi^N$ .

Estas señales son digitalizadas por un ADC conformando la matriz de datos  $x^N[n]$  que es verdaderamente lo que ingresa al modelo. A partir de este punto, el modelo deriva de esa información una salida  $y^K[n]$  que representa el output, la salida, pudiendo conformar un sistema MIMO, Multiple Input Multiple Output.

Paralelamente, se podría intentar recuperar una estimación de las fuentes originales del propio fenómeno físico, que se utilizan para derivar todo el proceso, las  $\hat{s}^M[n]$ . Existe un conjunto de técnicas transformativas que entran y se denominan separación de fuentes, o **Blind Source Separation**, que entran dentro del paraguas de aprendizaje automático [70]. Esto siempre es una estimación porque la información de las fuentes originales, con precisión absoluta, es siempre inaccesible. Esto es independiente de la propia capacidad que tenga el modelo, (AGI, superinteligente o lo que sea).



## 10. Transformaciones en los datos

El aprendizaje no supervisado se concentra en aprender patrones y relaciones existentes entre las propias muestras y sus variables, y no en aprender un mapeo específico entre las variables independientes y la variable respuesta, el *label*. Están fuertemente relacionados con la idea de transformar los datos, con buscar representaciones alternativas que ayuden precisamente a capturar esa información. Estos algoritmos suelen utilizarse de manera independiente, pero también como parte del propio pipeline de procesamiento, ya que generalmente esa nueva transformación puede ser útil para el caso supervisado.

### 10.1 Estructura interna de los datos

Lo que se busca es la identificación de la propia estructura de la información. Se busca como la información de las variables interactúa entre entre ellas y como eso lo puedo utilizar para identificar patrones subyacentes.

■ **Ejemplo 10.1** Por ejemplo, Para una investigación se seleccionan al azar 20 pacientes hipertensos sobre los que se midieron las siguientes variables:

- $X_1$ : Presión arterial media (mm Hg)
- $X_2$ : Edad (años)
- $X_3$ : Peso (Kg).
- $X_4$ : Superficie corporal ( $m^2$ )
- $X_5$ : Duración de la Hipertensión (años)
- $X_6$ : Pulso (pulsaciones/minuto)
- $X_7$ : Medida del stress.

Esto implica que la dimensión de la matriz de muestras es 7. Podemos intentar reducirla a dos variables, arbitrariamente: Presión vs Edad.

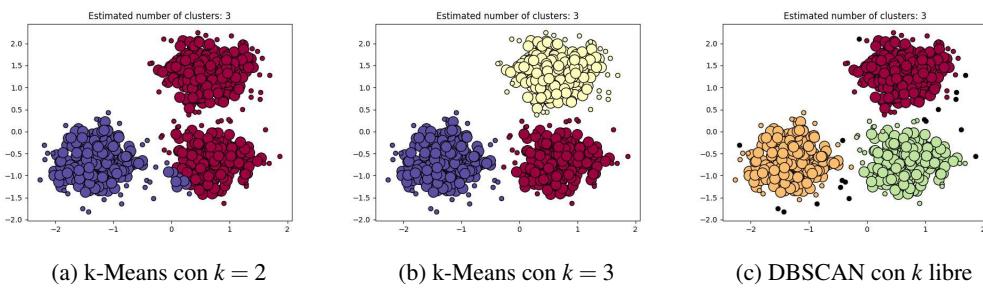
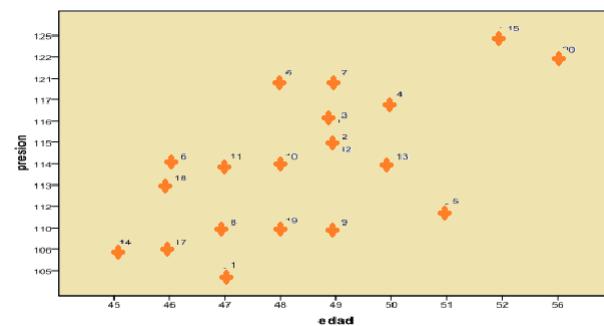


Figura 10.2: Alternativas de Clustering

Figura 10.1: Presión vs Edad



Del gráfico surge que dos individuos que se encuentran *cerca* podrían tener una característica similar de estas dos variables combinadas. Es decir, dos individuos que están cerca en este gráfico, deberían representar situaciones del mundo real parecidas. Esta es la idea subyacente en los métodos de aprendizaje no supervisado.

Para identificar esa estructura interna se apela a:

1. Agrupamiento o Clustering.
2. Asociaciones.
3. Reducción de dimensionalidad.

Los métodos de aprendizaje no supervisado hacen uso de las estructuras geométricas de los espacios donde viven las muestras  $X$ , y establecen sobre los mismos medidas de similitud. Cuando un elemento  $x_i$  es parecido a otro  $x_j$ . Para esto se usa todo mayoritariamente ideas de álgebra lineal, pero se expande rápidamente a un abanico de técnicas más amplio.

Particularmente en este texto, nos concentraremos en los métodos que implementan estas asociaciones basados en estructuras biomiméticas de redes neuronales. Sin embargo, se ofrece una pequeña introducción que servirá para comprender los conceptos generales.

### 10.1.1 Clustering y Agrupamientos

Basado en establecer esas medidas de similaridad entre los datos para verificar si pertenecen a subgrupos internos claramente identificables: buscar agrupamientos.

La figura 10.2 muestra por ejemplo tres variantes que reflejan los problemas más comunes en los agrupamientos. El primero es el agrupamiento dado por *k-means* [49]. Este algoritmo simple

identifica en base a un hiperparámetro que especifica la cantidad de clusters a identificar, y luego asigna cada elemento al cluster ya asignado para los  $k$  vecinos más cercanos. Para la figura 10.2a la elección de  $k = 2$  parecería no ser la mejor ya que graficamente se ven que los clusters son 3 lo cual se representa en el segundo gráfico 10.2b. Por otro lado el tercer gráfico 10.2c utiliza un algoritmo diferentes, *dbSCAN* [71], que tiene en consideración la densidad de elementos, y que permite identificar como algunos elementos como no pertenecientes a ninguno de los tres clusters en particular (en negro en el gráfico).





## 11. Análisis de Componentes Principales

Hacemos aquí un interludio para presentar un método de transformación de datos que enfatiza un aspecto de la inteligencia que es la capacidad para sumarizar información, para transformarla a un formato que capture cierta esencia de la información. Uno de estos métodos es el método de análisis de componentes principales.

### 11.1 Preliminares

- Media Muestral: Dado un conjunto de datos de una variable  $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

- Varianza Muestral:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

es una medida de dispersión.

- Desviación estándar muestral: Es otra medida de dispersión

$$s_{dv} = \sqrt{s^2}$$

- Covarianza muestral: es una medida de asociación lineal entre los datos de dos variables.  
Dado el conjunto de datos:

$\mathbf{X_1}$	$\mathbf{X_2}$	...	$\mathbf{X_n}$
$x_{11}$	$x_{12}$	...	$x_{1n}$
$x_{21}$	$x_{22}$	...	$x_{2n}$
$\vdots$			
$x_{m1}$	$x_{m2}$	...	$x_{mn}$

$$s_{ik} = \frac{1}{n} \sum_{j=1}^m (x_{ji} - \bar{x}_i) * (x_{jk} - \bar{x}_k)$$

Matriz de Covarianzas: Las covarianzas forman una matriz simétrica definida positiva

$$S = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n} \\ s_{21} & s_{22} & \dots & s_{2n} \\ \vdots & & & \\ s_{m1} & s_{m2} & \dots & s_{mn} \end{bmatrix}$$

Interpretación de la Covarianza muestral

- $s_{ik} > 0$  indica una asociación lineal positiva entre los datos de las variables.
- $s_{ik} < 0$  indica una asociación lineal negativa entre los datos de las variables.
- $s_{ik} = 0$  indica que no hay una asociación lineal entre los datos de las variables.

**R** La varianza muestral es la covarianza muestral entre los datos de la  $i$ -ésima variable con ella misma, algunas veces se denota como  $s_{ii}$

Variables estandarizadas:

$$\tilde{\mathbf{x}}_1 = \frac{\mathbf{x}_1 - \bar{\mathbf{x}}_1}{\mathbf{s}_1}$$

A cada variable le resto su propia media y divido por su propia desviación estándar.

Correlación muestral: Calcular las covarianzas con las variables estandarizadas:

Es otra medida de asociación lineal. Para los datos de la  $i$ -ésima y  $k$ -ésima variable se define como:

$$r_{ik} = \frac{s_{ik}}{\sqrt{s_{ii}} \sqrt{s_{kk}}}$$

La correlación está acotada entre -1 y 1.

### 11.1.1 Transformación PCA

Si las variables están muy correlacionadas, entonces poseen información redundante. La idea del método es justamente eliminar la redundancia. Transformar el conjunto original de variables en otro conjunto de nuevas variables que sean combinaciones lineales de las anteriores pero que no estén correlacionadas entre sí, llamado **conjunto de componentes principales**. La técnica de componentes principales es debida a Hotelling (1933), aunque sus orígenes se encuentran en los ajustes ortogonales por mínimos cuadrados introducidos por K. Pearson (1901).

Dadas  $p$  variables originales, se buscan  $q < p$  variables que sean combinaciones lineales de las  $p$  originales, recogiendo la mayor parte de la información o variabilidad de los datos.

Si las variables originales no están correlacionadas, entonces no tiene sentido realizar un análisis de componentes principales.

La característica que maximiza la variabilidad.

Es un buen factor para diferenciar objetos de un conjunto de datos. Por ejemplo, Conjunto de datos con información de vehículos

- Característica 1: Cantidad de ruedas.
- Característica 2: Longitud del vehículo. (Mayor variabilidad)

Utilizando la segunda nos daríamos cuenta si el registro corresponde a un auto o un colectivo.

Supongamos que se dispone de los valores de  $p$ -variables en  $n$  elementos de una población dispuestos en una matriz  $X$  de dimensiones  $n \times p$ , donde las columnas contienen las variables y las filas contienen los elementos.

Variables:

$$\{x_1, \dots, x_p\}$$

PCA realiza una transformación del conjunto de datos  $X$  que consiste de una traslación y una rotación, definidas de manera tal que la varianza del nuevo conjunto de variables sea máxima.

La Primera componente

$$y_1 = \sum_{j=1}^p a_{1j}(x_j - \bar{x}_j) = a_{11}(x_1 - \bar{x}_1) + \dots + a_{1p}(x_p - \bar{x}_p)$$

con

$$\vec{a}_1 = (a_{11}, a_{12}, \dots, a_{1p}) \in \mathbb{R}^p$$

El conjunto de Componentes principales es una combinación lineal de las variables originales.

Buscamos que  $\vec{a}_1 / \|\vec{a}_1\| = 1$  y que la  $Var(y_1)$  resulte máxima en el conjunto de las combinaciones lineales posibles de las variables originales. Cargas:

Los coeficientes  $a_{ji}$ ,  $i = 1, \dots, p$ ,  $j = 1, \dots, p$  se denominan cargas (o loadings).

¿Cómo hallar los valores de las cargas?

Son los autovectores de la matriz de covarianzas

Este problema se resuelve buscando los  $v_1, \dots, v_n$  y los  $\lambda_1, \dots, \lambda_n$

$$\det(S_X - \lambda_i I) = 0$$

y los  $v_i$  tal que

$$S_X v_i = \lambda_i v_i$$

Entonces

$$y_1 = v_{11}x_1 + \dots + v_{n1}x_n$$

$$y_2 = v_{12}x_1 + \dots + v_{n2}x_n$$

y así con todos.

Además, el autovalor  $\lambda_i$  es la varianza de la componente  $i$ .

Ordenando los autovalores de  $S_X$  de mayor a menor,  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$  podemos reducir la dimensionalidad tomando los autovectores correspondientes a los primeros  $q$  autovalores, que son los que proveen mayor información (en términos de variabilidad).

El porcentaje de variabilidad que poseen las primeras  $q$  componentes es

$$Variabilidad = \frac{\sum_{i=1}^q \lambda_i}{\sum_{i=1}^n \lambda_i}$$

Entonces, para reducir la dimensionalidad de un data set considerando sus características más importantes, deberíamos requerir que la proyección cubra, por ejemplo, el 95 % de las varianzas, o sea que

$$Variabilidad \geq 0.95$$

1. Paso 1: Tomar un conjunto de Datos  $X$ . Las variables deben estar en las columnas.
2. Paso 2: Restar la media de cada conjunto de variables. Calcular  $X - \bar{X}$ . Obtener un conjunto de datos con media cero.
3. Paso 3: Calcular la matriz de Covarianzas.
4. Paso 4: Calcular autovalores y autovectores de la matriz de covarianzas y ordenar los autovalores de mayor a menor.
5. Paso 5: Formar la matriz  $E$  tomando los autovectores correspondientes a los mayores autovalores.
6. Paso 6: Calcular las nuevas variables  $Y = E(X - \bar{X})$

Matriz Covarianzas vs. Matriz de Correlación Si alguna de las variables, por ejemplo la primera, tiene valores mayores que las demás, la manera de aumentar la varianza es hacer tan grande como podamos la coordenada asociada a esta variable. Por ejemplo: pasamos de medir en km. a medir en metros, el peso de esa variable en el análisis aumentará.

Entonces, cuando las escalas de medida de las variables son muy distintas, la maximización de la varianza dependerá decisivamente de estas escalas de medida y las variables con valores más grandes tendrán más peso en el análisis.

Si queremos evitar este problema conviene utilizar las variables estandarizadas para calcular las componentes principales, de manera que las magnitudes de los valores numéricos de las variables originales sean similares. Lo cual es lo mismo que aplicar el análisis de componentes principales utilizando la matriz de correlaciones en lugar de la matriz de covarianzas.

Cuando las variables tienen las mismas unidades, ambas alternativas son posibles.

Ejemplo:

Sea la matriz de covarianzas

$$S = \begin{pmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 5 \end{pmatrix}$$

correspondiente a las variables aleatorias ( $X_1, X_2, X_3$ ) con media cero.

1. Escribir las variables ( $Y_1, Y_2, Y_3$ ) de componentes principales y calcular qué proporción de la varianza total explica cada componente.

Autovalores:  $\lambda_1 = 6, \lambda_2 = 3, \lambda_3 = 2$

Autovectores:

$$E = \begin{pmatrix} V_1 & V_2 & V_3 \\ -0.40 & -0.57 & 0.70 \\ -0.40 & -0.57 & -0.70 \\ -0.81 & 0.57 & 0 \end{pmatrix}$$

$$Y_1 = -0.40X_1 + (-0.40)X_2 + (-0.81)X_3$$

$$Y_2 = -0.57X_1 + (-0.57)X_2 + 0.57X_3$$

$$Y_3 = 0.70X_1 + (-0.70)X_2$$

Proporción de varianza de cada componente: Autovalores:  $\lambda_1 = 6, \lambda_2 = 3, \lambda_3 = 2$

$$\begin{aligned} \text{de } Y_1 &= \frac{6}{11} \\ \text{de } Y_2 &= \frac{3}{11} \\ \text{de } Y_3 &= \frac{2}{11} \end{aligned}$$

¿Qué información aportan las cargas o *loadings*?

- Si la carga (coeficiente o *loading*) de una variable en la componente principal es positiva, significa que la variable y la componente tienen una correlación positiva.
- Si por el contrario, la carga es negativa, este hecho indica que dicha variable se correlaciona en forma negativa con la primera componente.

Así la primera componente representa un índice (o una característica) por el cual se pueden ordenar los registros.



## 12. Red de Kohonen

La red de Kohonen, también llamada SOM, Self Organized Map, mapa autoorganizado, es una red bioinspirada que permite detectar regularidades en los datos de entrada. Su autor fue Teuvo Kohonen, investigador Finlandés que publicó su idea por primera vez en 1982, y fue una de las razones de la segunda ola de la inteligencia artificial [72, 73].



### 12.1 Mapa Autoorganizado

La arquitectura de la red se compone de dos capas, donde en la primera corresponde a las entradas, y la segunda a una capa donde las neuronas están conectadas con sí mismas positivamente y tienen conexiones inhibitorias con las neuronas vecinas. Este sistema produce a lo largo del tiempo, que alguna de las unidades tomen un nivel de activación mayor mientras el nivel de las demás unidades se anula.

Como se observa en la figura 12.2, la capa central es una grilla o mapa que puede a su vez tener diferentes layout topológicos.

Figura 12.1: Teuvo Kohonen contento por su gran trabajo.

#### 12.1.1 Aprendizaje Competitivo

La idea de Kohonen fue implementar un mecanismo biomimético de aprendizaje competitivo. Dada una unidad de entrada  $x$ , la neurona que tenga vector de pesos  $w$  más parecido, más similar, bajo alguna medida de similitud, será la ganadora. Esto es un esquema de agrupamiento, porque el resultado que genera es que entradas similares van a activar la misma neurona de la grilla, agrupándose. Es competitivo porque las neuronas terminan compitiendo unas con otras con el objetivo de que finalmente sólo una de las neuronas de salida se active y las demás sean forzadas a valores de respuestas mínimos. Este mecanismo se denomina **winner take all**.

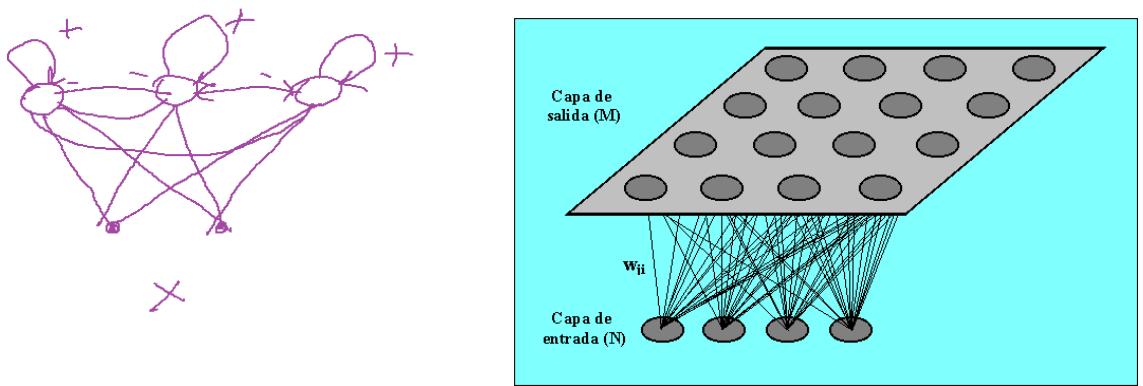


Figura 12.2: Mapa autoorganizado, estructura de conexiones de la red y grilla de salida.

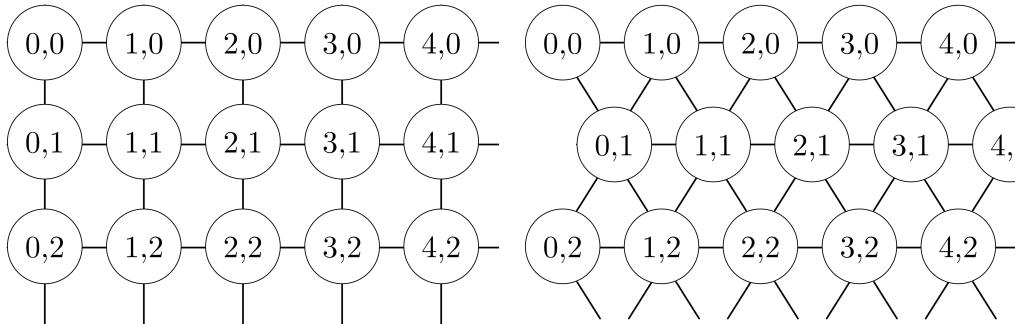


Figura 12.3: Estructuras posibles para la grilla de la red.

### 12.1.2 Arquitectura

La red de Kohonen tienen una sola capa de neuronas (más una de entradas) que forma una grilla bidimensional ( $k \times k$ ) y en la que cada neurona está conectada a todas las componentes de un vector de entrada n-dimensional. Los datos así de entrada que son multidimensionales se proyectan en un espacio bidimensional. La grilla puede ser rectangular o hexagonal.

Sobre la grilla se define un **vecindario** de 4-vecinos con radio  $R = 1$  ú 8-vecinos con radio  $R = \sqrt{2}$ . Cada vez que se coloca un vector en la entrada, se selecciona una neurona ganadora y se aplica una regla de actualización a esa neurona y su vecindario.

Las características de la red son:

- Se establece un tamaño particular para la grilla de  $k \times k$ , con una topología, rectangular o hexagonal.
- Cada neurona de salida  $j \in \{1, \dots, k^2\}$  tiene asociado un vector de pesos  $W_j = (w_{j1}, \dots, w_{jn})$ .
- Los pesos  $W_j$  de cada neurona de salida, tienen la misma dimensión que los datos de entrada.

### 12.1.3 Algoritmo

El algoritmo es bastante directo y se basa en una actualización iterativa de los pesos de la red a medida que se van procesando los ejemplos del conjunto de entrenamiento.

**Algoritmo 10** Red de Kohonen

- 1:  $X^P = \{x_1^P, \dots, x_n^P\}$ ,  $p = 1, \dots, P$  son los registros de entrada.
  - 2: Definir la cantidad de neuronas de salida:  $k \times k$ .
  - 3: Inicializar los pesos  $W_j$ ,  $j = 1, \dots, k^2$ , cada  $W_j = (w_{j1}, \dots, w_{jn})$ :
  - 4: Con valores aleatorios con distribución uniforme.
  - 5: Con ejemplos al azar del conjunto de entrenamiento.
  - 6: Seleccionar un tamaño de entorno inicial  $R(0)$ .
  - 7: Seleccionar la tasa de aprendizaje inicial  $\eta(0) < 1$ .
  - 8: **while** iteración  $t$  **do**
  - 9: Seleccionar un registro de entrada  $X^P$ .
  - 10: Encontrar  $\hat{k}$  ganadora con vector de pesos  $W_{\hat{k}}$  más cercano a  $X^P$ :
  - 11: Definir una medida de similitud  $d$  y luego calcular:
- $$W_{\hat{k}} = \arg \min_{1 \leq j \leq N} \{d(X^P - W_j)\}$$
- 12: Actualizar los pesos de las neuronas vecinas según la regla de Kohonen.
  - 13: **end while**

**Regla de Kohonen**

Partiendo de la activación (la selección) de la neurona  $\hat{k}$ , que es la neurona ganadora y utilizando la definición del radio del vecindario que es  $R(t)$ , entonces el conjunto vecindario es:

$$N_{\hat{k}}(t) = \{neu / \|neu - neu_{\hat{k}}\| < R(t)\}$$

donde  $R(0)$  es un dato de entrada y  $R(t) \rightarrow 1$  cuando  $t \rightarrow \infty$ . Esto implica un esquema de actualización del radio aunque también puede permanecer constante durante todo el proceso.

Una vez que el conjunto de neuronas a activar está seleccionado, se pasa a actualizar los pesos de todas estas neuronas vecinas de  $\hat{k}$  utilizando la regla de Kohonen:

- Si  $j \in N_{\hat{k}}(t) \rightarrow W_j^{t+1} = W_j^t + \eta(t) * (X^P - W_j^t)$
- Si  $j \notin N_{\hat{k}}(t) \rightarrow W_j^{t+1} = W_j^t$

El valor de  $\eta(t)$  actúa como un factor de descuento que se va reduciendo en cada iteración,  $\eta(t) \rightarrow 0$ . Por ejemplo, se puede utilizar  $\eta(t) = \frac{1}{t}$ .

Este factor es clave para que la red de Kohonen converja: que los pesos de la red se parezcan a los datos de entrada y capturen su estructura de similitud.

*Demostración.* Asumimos una iteración de un paso  $t$  a un paso  $t + 1$ :

$$W_{\hat{k}}^{t+1} - X^P = W_{\hat{k}}^t + \eta(t)(X^P - W_{\hat{k}}^t) - X^P = (1 - \eta(t))(W_{\hat{k}}^t - X^P)$$

Entonces...

$$\|W_{\hat{k}}^{t+1} - X^P\| \leq \|W_{\hat{k}}^t - X^P\|$$

con lo que la diferencia entre el peso y la entrada se va reduciendo. Esto muestra que el nuevo vector de pesos está más cerca del patrón X que el anterior. ■

### Medidas de similitud (o funciones de propagación)

- Distancia Euclídea:

$$W_k = \arg \min_{1 \leq j \leq N} \{ \|X^p - W_j\| \}$$

- Distancia Exponencial:

$$W_k = \arg \max_{1 \leq j \leq N} \{ e^{-\|X^p - W_j\|^2} \}$$

(normalizando todos los vectores)

### Inicialización

- Los pesos se pueden inicializar con valores aleatorios: Puede tener el problema de que algunas unidades queden lejos de los valores iniciales y entonces nunca ganen. Se dice que son unidades muertas.
- Para evitar eso se puede inicializar los pesos con muestras de los datos de entrada.
- La cantidad total de iteraciones conviene elegirla en función de la cantidad de neuronas de entrada, por ejemplo  $500 * n$ .
- El valor de  $R(0)$  puede ser el tamaño total de la red y va decreciendo hasta llegar a  $R_f = 1$ , donde solamente se actualizan las neuronas vecinas pegadas.

### Preprocesamiento de los datos

La red requiere estandarizar (ver 21.1.1) las variables del conjunto de entrenamiento y prueba:

$$\tilde{X}_i = \frac{X_i - \bar{X}_i}{s_i}$$

donde  $\bar{X}_i$  es la media y  $s_i$  es la desviación estándar de  $X_i$ .

### Reducción de la dimensionalidad

La red de Kohonen es un algoritmo bioinspirado de MDS, Multidimensional Scaling [74], simple y directo para establecer clusters bidimensionales. Al usar esquemas bidimensionales, es posible además visualizar el resultado del entrenamiento de la red y apreciar como las neuronas de salida forman una matriz, y verificar en qué coordenadas se encuentra la neurona asociada a cada ejemplo de entrenamiento.

- La red de Kohonen puede ser más rápida que el Perceptrón multicapa.
- Al ser no supervisado puede aplicarse en casos donde el conjunto de datos solo tiene valores de entrada sin etiquetar.
- Si el conjunto de variables es muy grande puede ser difícil asociarlo con un conjunto bidimensional.
- Solo puede realizarse con variables numéricas.
- Hay que decidir el tamaño de la grilla desde el principio y no hay un criterio demostrado para hacerlo.



## 13. Red de Hopfield

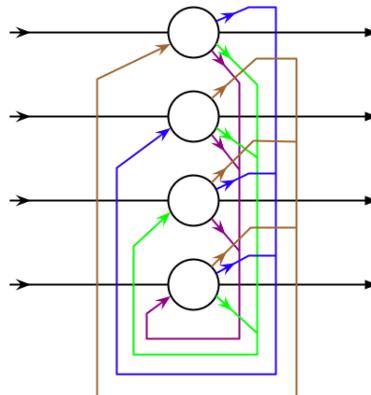
La red de Hopfield es un modelo de redes neuronales inspirada en modelos de la física computacional que tiene una propiedad biomimética de memoria asociativa. Dado un conjunto de  $p$  patrones almacenados, cuando se presenta a la red un nuevo patrón  $A$ , permite responder con uno de los patrones almacenados  $B$ , de forma tal que sea el más parecido a  $A$ . Al patrón  $A$  se lo denomina **estímulo** en tanto que  $B$  se denomina **respuesta**. Este interesante comportamiento, con muchos paralelismos con sistemas biológicos, se logra mediante el establecimiento de una red recurrente donde la temporalidad se hace presente y que manifiesta una estructura de un sistema dinámico, que evoluciona con el tiempo, y que a su vez tiene una energía asociada al sistema. Hopfield realiza su propuesta desde la perspectiva de la física computacional, inspirándose en el modelo de Ising y Simulated Annealing [48]. Esto fue muy inteligentemente manipulado para generar el efecto de una memoria asociativa, que asocia estímulos de entrada, con respuestas memorizadas. La red de Hopfield fue sustancial en impulsar la segunda era de la inteligencia artificial, basada justamente en el modelo conexionista, lo cual quedó plasmado en la adjudicación de nada más y nada menos que el premio Nobel 2024 de Física [75].



Figura 13.1: John Hopfield muy contento después de crear su red homónima.

### 13.1 Hopfield 1982

La red de Hopfield de 1982 es una Red Recurrente discreta. Las redes recurrentes conforman sistemas dinámicos donde las salidas de cada neurona corresponden a las entradas en un paso de tiempo posterior. Se le llama versión del '82 a la versión discreta, en tanto que la versión '84 corresponde a la versión continua.



La red de Hopfield tiene las siguientes características:

- Todas las neuronas están conectadas entre sí. Todas contra todas en una configuración estrella.
- Ninguna neurona está conectada con sí misma.
- El conjunto permitido de valores de entrada y salida es  $\{-1, 1\}$ , es un conjunto binario. De esta manera todas las neuronas en una Red Hopfield son binarias, tomando solamente uno de los dos estados posibles: activo o inactivo.
- Todas las neuronas están en una sola capa de entrada y salida.
- El modelo de Hopfield puede representar un sistema físico donde el estado está dado por la salida de todas las neuronas al mismo tiempo, y a su vez la función de costo del sistema refleja la energía del sistema, donde los mínimos de esa función corresponden a los estados estables que es donde están los patrones almacenados.

Las redes de Hopfield actúan como memorias asociativas. En el tiempo  $t = 0$  un patrón es colocado en todas las entradas de la red (que corresponden a las entradas de todas las neuronas). Luego se calculan todas las salidas que serán las entradas del sistema para el tiempo  $t + 1$ . El sistema se deja evolucionar de esta manera, hasta que se alcanza un punto estable donde no se producen más cambios. Los valores de la salida en ese momento, serán justamente uno de los patrones almacenados en la red.

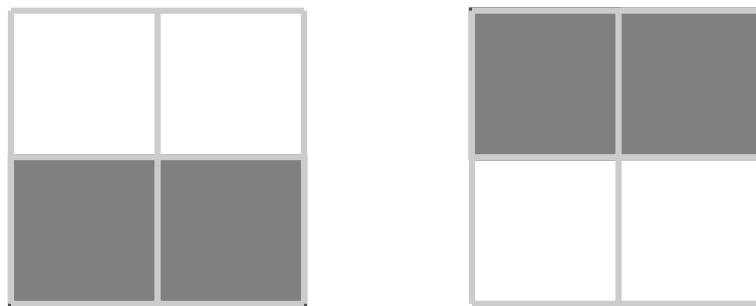


Figura 13.2: Patrones de imágenes binarias. Asumiendo blanco=1 y gris=-1, el primer patrón es  $\xi_1 = (1, 1, -1, -1)$  en tanto que el segundo es  $(-1, -1, 1, 1)$ .

Estos patrones pueden ser imágenes (Figura 13.2), conformando un patrón de entrada visual. Cada neurona tiene entonces 2 estados posibles:

- $S_i = +1$
- $S_i = -1$

Para una red de  $N$  neuronas, el estado queda representado por el vector de estados  $S = [S_1, S_2, \dots, S_N]^T$ .

### 13.1.1 Memoria Asociativa

Supongamos que tenemos un conjunto de patrones almacenados  $\xi^\mu, \mu = 1, \dots, p$ , ya aprendidos, en una red de Hopfield. Presentamos un nuevo patrón  $\zeta$  con el objetivo de encontrar el patrón almacenado más cercano a  $\zeta$ . Para eso necesitamos que la red almacene pesos sinápticos  $w_{ij}, i = 1, \dots, N, j = 1, \dots, N$  tal que la red nos devuelva el patrón  $\xi^\mu$  más cercano a  $\zeta$ .

Las neuronas de la red tienen las siguientes características:

- Cada neurona  $i$  es un Perceptrón simple con la función de activación escalón (1, -1).
- Cada par de neuronas  $(i, j)$  se conectan por el peso sináptico  $w_{ij}$ .

Comenzamos con una configuración  $S_i = \zeta_i, i = 1, \dots, n$ , y queremos ver si hay un conjunto de pesos  $w_{ij}$  que hagan a la red alcanzar el estado  $S_i = \xi_i^{\mu_0}$ , donde  $\xi^{\mu_0}$  es un patrón almacenado, en tanto que  $\xi_i^{\mu_0}, i = 1, \dots, n$  es el estado del patrón almacenado más parecido a  $\zeta_i, i = 1, \dots, n$ . La neurona  $i$  modifica su estado  $S_i$  de acuerdo a la regla:

$$h_i = \sum_{j=1}^N w_{ij} S_j, \quad i \neq j$$

$$S_i = \begin{cases} +1 & \text{si } h_i > 0 \\ -1 & \text{si } h_i < 0 \end{cases}$$

En el caso de que  $h_i = 0$  entonces la neurona  $i$  permanece en el estado previo. Esto por supuesto es equivalente a:

$$S_i = \operatorname{sgn}(h_i)$$

Este paso anterior representa la actualización del estado del sistema, donde se actualizan uno a uno todos los valores de salida, dado el estado completo del sistema en el paso de tiempo anterior. Así, la red evoluciona hasta que  $S_i$  no se modifica más  $\forall i = 1, \dots, n$ .

Si tomásemos sólo un patrón en el conjunto, ( $\xi_i = 1$  ó  $\xi_i = -1$ ),  $i = 1, \dots, N$ , asumiendo que tenemos  $N$  nodos en la red, podemos elegir los pesos sinápticos de la siguiente forma:

$$w_{ij} = \frac{1}{N} \xi_i \xi_j, \quad i, j = 1, \dots, N$$

y así evolucionar la red haciendo

$$S_i = \operatorname{sgn} \left( \sum_{j=1}^N w_{ij} \xi_j \right) = \operatorname{sgn}(\xi_i) = \xi_i \quad \forall i.$$

Esto es debido a que  $\xi_j^2 = 1 \quad \forall j = 1, \dots, N$  y por lo tanto se produce la convergencia: alcanza el patrón almacenado.

Esto se extiende para el caso de muchos patrones, haciendo uso de la regla de Cooper [76] para el cálculo cerrado de los pesos:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^p \xi_i^\mu \xi_j^\mu, \quad i \neq j$$

donde  $p$  es el número total de patrones almacenados. Esta ecuación asigna con un valor elevado los pesos entre dos nodos  $i$  y  $j$  cuando ambos están activados o ambos desactivados.

Como la arquitectura de la red no admite conexiones directas de cada neurona con sí misma y la conexión es simétrica, se toma también  $w_{ii} = 0$  y  $w_{ij} = w_{ji}$ . Todo esto matricialmente queda

$$\begin{aligned} W &= \frac{1}{N} K K^T \\ S(t+1) &= \text{sign}(W S(t)) \end{aligned}$$

El algoritmo 11 muestra el paso a paso del algoritmo completo.

---

#### Algoritmo 11 Modelo de Hopfield

---

- 1: Almacenamiento: Se tiene  $p$  patrones  $\xi^1, \xi^2, \dots, \xi^p$  de dimensión  $N$ , binarios.
  - 2: Los pesos sinápticos  $w_{ij}$  conectan la neurona  $i$  con la  $j$ .
  - 3:
- $$w_{ij} = \begin{cases} \frac{1}{N} \sum_{\mu=1}^p \xi_i^\mu \xi_j^\mu & j \neq i \\ 0 & j = i \end{cases}$$
- 4: Sea  $\zeta$  un vector de consulta  $N$ -dimensional desconocido, presentado a la red.
  - 5:  $S_i(0) = \zeta_i$  es el estado de la neurona  $i$  en el tiempo  $t = 0$
  - 6: **while**  $S$  NO permanezca estable **do**
  - 7:    $S_i(t+1) = \text{sgn}(\sum_{j=1}^N w_{ij} S_j(t)), j \neq i$
  - 8:    $t++$
  - 9: **end while**
  - 10: Return  $S(t_{final})$
- 

Remarcamos que los pesos sinápticos se precisan sólo al inicio (la cuenta esa sería hacer el aprendizaje, que en este caso es directo e instantáneo, porque tiene una fórmula cerrada). En el Perceptrón multicapa o en cualquier otra red, esto ocurriría si pudiésemos resolver directamente el proceso de minimización, y no requeriría iterar.

### 13.1.2 Estabilidad

Sea  $\xi_i^v$  el elemento  $i$  del patrón  $v$ . La condición de estabilidad es:

$$h_i^v = \sum_j w_{ij} \xi_j^v = \frac{1}{N} \sum_j \sum_\mu \xi_i^\mu \xi_j^\mu \xi_j^v \quad (13.1)$$

lo cual representa

$$\begin{aligned} h_i^v &= \frac{1}{N} \sum_j \sum_\mu \xi_i^\mu \xi_j^\mu \xi_j^v \\ h_i^v &= \frac{1}{N} \sum_j \sum_{\mu \neq v} \xi_i^\mu \xi_j^\mu \xi_j^v + \frac{1}{N} \sum_j \xi_i^v \xi_j^v \xi_j^v \\ h_i^v &= \xi_i^v + \frac{1}{N} \sum_j \sum_{\mu \neq v} \xi_i^\mu \xi_j^\mu \xi_j^v. \end{aligned}$$

El sistema entonces es estable si: (1) el segundo término, el **crosstalk** es cero, ó (2) cuando los patrones son ortogonales: esta es una de las limitaciones de esta red.

### 13.1.3 Convergencia

Hopfield demostró que la red está asociada a una función de Energía, dada en su forma de Lyapunov [48]:

$$H(w) = -\frac{1}{2} \sum_{i,j} w_{ij} S_i S_j \quad (13.2)$$

y que los mínimos locales de esta función son precisamente las localizaciones donde están los patrones.

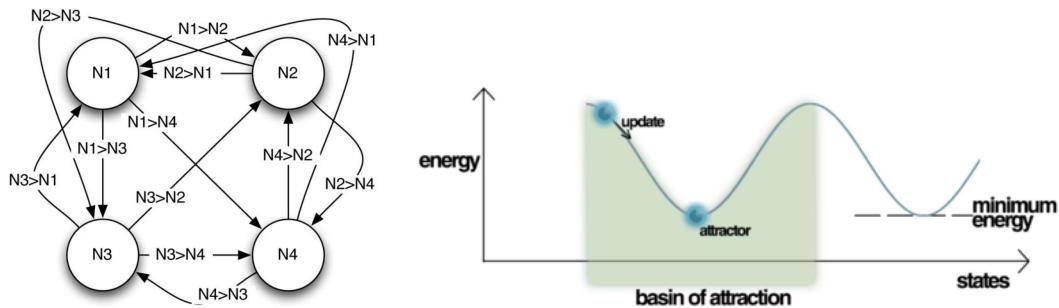


Figura 13.3: Red de Hopfield donde todas las salidas son las entradas de las otras neuronas. El estado de la red puede capturarse con una función de costo/energía que busca minimizarse ante cualquier perturbación (la entrada) y en los valles están los patrones aprendidos.

Las funciones de energía siempre decrecen o permanecen constante cuando el sistema evoluciona. Como los pesos son simétricos y  $w_{ii} = 0$  entonces puedo sumar solamente la parte superior de la matriz y multiplicar por 2.

Así mismo, sea  $S'_i$  el nuevo valor de  $S_i \rightarrow S'_i = \text{sgn}(\sum_{j \neq i} w_{ij} S_j)$ :

- Si  $S'_i = S_i$  entonces la energía no se modifica.
- Si  $S'_i = -S_i$  entonces

$$\begin{aligned} \Delta H &= H' - H \\ &= -\sum_{j>i} w_{ij} S'_i S_j + \sum_{j>i} w_{ij} S_i S_j \\ &= 2S_i \sum_{j>i} w_{ij} S_j \\ &= 2S_i \underbrace{\sum_j}_{S'_i} w_{ij} S_j < 0 \end{aligned}$$

porque estamos considerando los  $S'_i$  con signo contrario a  $S_i$ , y entonces la energía decrece cada vez que un  $S_i$  cambia.

### 13.1.4 Limitaciones

- El número máximo de patrones que puede almacenar es igual al 15 % del número de neuronas de la red. O sea que  $p \leq 0.15 * N$ , siendo  $N$  la dimensión de los patrones.
- Los patrones deben ser más o menos ortogonales (vectorialmente) o puede utilizarse la distancia de Hamming como un evaluador de la similitud/ortogonalidad de los patrones.

La segunda limitación es la existencia de **Estados Espúreos**: como los patrones son atractores, la función de energía  $H$  puede tener otros mínimos locales que no son los patrones almacenados. Justamente estos mínimos son estables espúreos que no corresponden particularmente a alguno de los patrones almacenados [77].



## 14. Modelo de Oja y Sanger

### 14.1 La búsqueda del poder de síntesis



Figura 14.1: Erkki Oja contento porque está en Aalto.

¿Es la inteligencia la capacidad de sintetizar, de comprimir [78] información? La transformación de los datos por el método de análisis de componentes principales en el capítulo 11 puede verse desde una perspectiva de teoría de la información. Desde ese ángulo la variabilidad intrínseca de los datos puede representar el medio por el cuál se transmite información. Más variación, más información, más importancia y desde esa perspectiva entender cuál es la contribución de cada dato en relación a cuánta variabilidad aporta.

A finales del Siglo 21 varios investigadores, entre los que se encontraron Erkki Oja y Terence Sanger, encontraron que algunos esquemas de redes neuronales simples, con reglas de actualización biomiméticas de los pesos, similares a la regla de actualización Hebbiana, convergían a una representación similar a la obtenida en la descomposición de componentes principales. Es decir, podían encontrar una nueva representación de los datos, donde los ejes estaban determinados según cuánta variabilidad inherente existían en ellos.

¿Será esta una forma de codificar la información apropiada para capturar su esencia, generar una abstracción y de allí un mecanismo de inteligencia?

### 14.2 Red de Oja

Retomando la ecuación del Perceptrón lineal simple:

$$O^\mu = \sum_{i=1}^n x_i^\mu w_i \quad (14.1)$$

La actualización de los pesos sinápticos surge de

$$\Delta w = \eta(\zeta^\mu - O^\mu)x^\mu \quad (14.2)$$

con  $\eta$  como la tasa de aprendizaje,  $\zeta$  la salida deseada,  $O$  la salida real obtenida, y la regla de actualización iterativa sobre los pesos viene de

$$w^{n+1} = w^n + \Delta w. \quad (14.3)$$

Esta regla de actualización es supervisada. ¿Hay alguna manera de hacerla no supervisada con algún criterio de optimalidad? La respuesta es que sí, y eso es lo que plantea Oja.

**Teorema 14.2.1 — Regla de Aprendizaje Hebbiano para Aprendizaje No Supervisado.**

Dados  $m$  datos de entrada  $x^1, \dots, x^m$ , con  $x^\mu \in \mathbb{R}^N \forall \mu$ , un Perceptrón lineal simple calcula la salida hacia adelante como:

$$O^\mu = \sum_{i=1}^N w_i x_i^\mu \quad (14.4)$$

donde  $w \in \mathbb{R}^N \forall \mu$ . La actualización no supervisada de los pesos sinápticos puede hacerse como

$$\Delta w_j = \eta O^\mu x_j^\mu \quad (14.5)$$

$$w_j^{n+1} = w_j^n + \Delta w_j. \quad (14.6)$$

Oja [79] demostró que si un Perceptrón lineal con esta función de actualización converge, entonces el vector de pesos resultante  $w(n_{final})$  es una proyección sobre la dirección de máxima variación de los datos, es decir sobre el eje de la **primera componente principal**.

Sin embargo esta regla de actualización Hebbiana diverge. Por ello, Oja propone una actualización donde mantiene a raya a los pesos normalizando cada actualización en relación a la norma del paso anterior:

$$w_j^{n+1} = \frac{w_j^n + \eta O^n x^n}{\|w^n\|} \quad (14.7)$$

teniendo en cuenta que el patrón  $\mu$  es presentado en el paso  $n$ , y con

$$\|w^n\| = \left( \sum_{j=1}^N (w_j^n + \eta O^\mu x_j^\mu)^2 \right)^{\frac{1}{2}}. \quad (14.8)$$

Oja estima esta actualización mediante el cálculo del polinomio de Taylor en  $\eta$

$$\begin{aligned} w_j^{n+1} &= \frac{w_j^n}{\|w\|} + \eta \left( \frac{y(\mathbf{x}^n) \mathbf{x}_j^n}{\|w\|} - \frac{w_j \sum_{j=1}^N y(\mathbf{x}^n) \mathbf{x}_j^n \mathbf{w}_j}{\|w\|^3} \right) \\ &= \frac{w_j^n}{\|w\|} + \eta \left( \frac{y(\mathbf{x}^n) \mathbf{x}_j^n}{\|w\|} - \frac{w_j^n y(\mathbf{x}^n) \sum_{j=1}^N \mathbf{x}_j^n \mathbf{w}_j^n}{\|w\|^3} \right) \end{aligned}$$

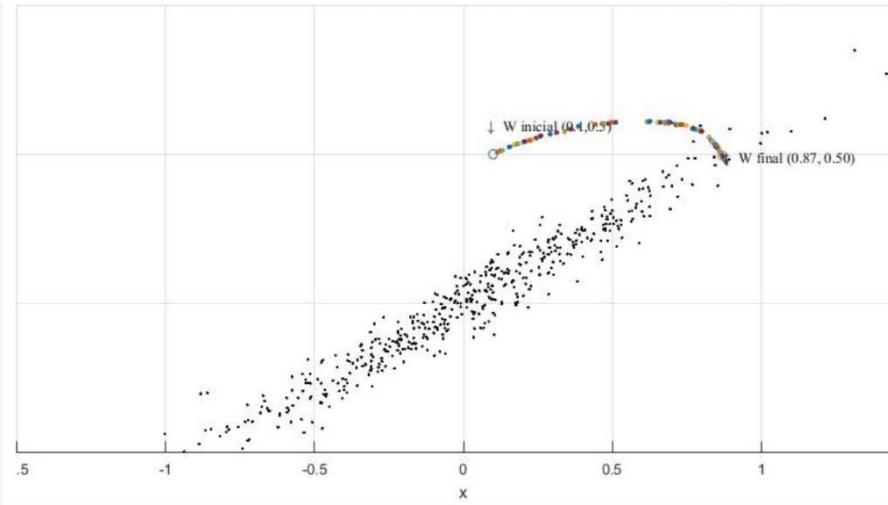


Figura 14.2: Simulación de una neurona lineal con 2 entradas y actualizando w según la regla de Oja. Se ve como el peso se desplaza hacia el eje correspondiente a la mayor variabilidad de los datos.

convergiendo a :

$$\Delta w_j^{n+1} = \eta(O x_j^n - O^2 w_j^n) \quad (14.9)$$

Con esta regla, el valor de los pesos se mantiene acotado, no diverge, e incluso converge luego de varias iteraciones al autovector correspondiente al mayor autovalor de la matriz de correlaciones de los datos de entrada, es decir al primer componente principal de la transformación de PCA.

Oja también demostró que para asegurar convergencia se requería  $\eta = \frac{1}{1.2\lambda_1}$ , siendo  $\lambda_1$  el primer autovalor principal. Como este valor no se conoce, se puede aproximar con  $\eta_0 \leq 0.5$  para comenzar, y luego para facilitar la convergencia ir reduciendo este valor, dividiéndolo por el paso de iteración del algoritmo (tampoco es buena idea reducirlo drásticamente; la suma de los  $\eta$  utilizados durante todos los pasos tiene que diverger).

### 14.2.1 Implementación

La arquitectura es un Perceptrón simple con una sola capa de salida. La actualización de los pesos sigue la propia regla de Oja y la inicialización de los pesos se hace con valores uniformes entre 0 y 1. Hiperparámetros adecuados para esta red son  $\eta(0) = 0.5$  y

---

#### Algoritmo 12 Regla de Actualización de Oja

---

- 1: Entrada: N datos con media 0,  $\eta$  y  $w$ .
  - 2: **while** época in épocas **do**
  - 3:     **while**  $i$  iterando de 1 a  $N$  (las dimensiones) **do**
  - 4:          $s = \tilde{x} \cdot \tilde{w}$
  - 5:          $w_i+ = \eta \cdot s \cdot (x_i - s \cdot w)$
  - 6:     **end while**
  - 7: **end while**
  - 8:  $w$  Resultado
- 

Así, este algoritmo encuentra los pesos que representan la primer componente principal en base a una regla de actualización biomimética.

### 14.3 Red de Sanger

Terence Sanger propuso una extensión a la regla de Oja, conformando la Red Hebbiana de Sanger [80], que mediante el aprendizaje Hebbiano converge los pesos a la matriz completa de autovectores de la matriz de covarianza de los datos, por lo que permite encontrar toda la descomposición en componentes principales. Este algoritmo se llama *Algoritmo Hebbiano Generalizado*:

**Teorema 14.3.1 — GHA, Generalized Hebbian Algorithm.** Asumamos una salida vectorial de la red neuronal con

$$O_j^\mu = \sum_{i=1}^N w_{ji} x_i^\mu. \quad (14.10)$$

La regla generalizada hebbiana de Sanger extiende la regla de Oja con

$$w_{il}^{n+1} = w_{il}^n + \eta O_i^n (x_l - \sum_{j=1}^i O_j^n w_{jl}^n) \quad (14.11)$$

con  $1 \leq i \leq k$ ,  $1 \leq k \leq N$ ,  $1 \leq l \leq N$ ,  $\eta \geq 0$

Lo que la regla hace es mantener *a raya* los valores de los pesos restandole todas las componentes encontradas en los pasos anteriores, en todas las coordenadas. Esta regla de actualización se puede aplicar de manera directa en el Algoritmo 12 y permite encontrar la matriz completa de los autovectores principales, que permite encontrar todas las cargas para todos los ejes de la transformación de PCA [80, 76].

# Deep Learning



<b>15</b>	<b>Deep Brain .....</b>	<b>119</b>
15.1	Biomimesis neuronal	
15.2	La Neurona Recargada	
<b>16</b>	<b>Aprendizaje Profundo .....</b>	<b>129</b>
16.1	Guía para el tío y la tía que pregunta	
16.2	Profundidad	
16.3	Aprendizaje de la Representación	
16.4	Datasets	
16.5	GPUs	
<b>17</b>	<b>Autoencoders .....</b>	<b>137</b>
17.1	Autoasociadores	
17.2	Autoencoder Generativo	
17.3	VAE - Variational Autoencoder	
17.4	Apéndice	
<b>18</b>	<b>Convolutional Neural Networks .....</b>	<b>157</b>
18.1	La operación de Convolución	
18.2	Convolución en Imágenes	
18.3	Neocognitrón	
18.4	Capa Convolucional	
18.5	La capa de Pooling	
18.6	Full Layer y Softmax	
18.7	Backpropagation	
18.8	¿Por qué funcionan?	
18.9	Inteligibilidad	
18.10	Arquitecturas Reusables	
18.11	Más allá de las imágenes	
18.12	Quiero más	
<b>19</b>	<b>Generative Adversarial Networks .....</b>	<b>171</b>
19.1	Generando gatitos	
19.2	Juego MinMax	
19.3	Colapso Modal	
19.4	El gran simulador	
<b>20</b>	<b>Transformers .....</b>	<b>177</b>
20.1	Las Redes Recurrentes	
20.2	Arquitectura de un Transformer	
20.3	Attention is all you need	
20.4	Embedding	
20.5	Entrenamiento	
<b>21</b>	<b>Guía para el apurado .....</b>	<b>185</b>
21.1	Condimentos de una red neuronal	
21.2	Infraestructura de Software	
21.3	Quiero más	
	<b>Index .....</b>	<b>203</b>



## 15. Deep Brain

### 15.1 Biomimesis neuronal

Como se mencionó en el capítulo 4 sobre algoritmos genéticos, la naturaleza implementa un poderoso algoritmo de optimización aplicado sobre la vida en la Tierra, separando lo que sirve para sobrevivir y perpetuarse de lo que no sirve para cada una de las eras particulares. En algún momento, ese algoritmo convergió elitistamente a preservar y promover organismos que fueron desarrollando la capacidad de transmitir y procesar información de manera cada vez más sofisticada mediante los sistemas nerviosos, provocando así el desarrollo de lo que llamamos inteligencia.

Los organismos vertebrados primates, como el ser humano, tienen un sistema nervioso que se divide en dos partes. El CNS, Central Nervous System, el sistema nervioso central y el PNS Peripheral Nervous System, o sistema nervioso periférico. El CNS se encuentra dentro del cráneo en tanto que el periférico se extiende por la columna vertebral y cubre todo el cuerpo, culminando en terminaciones nerviosas que se encargan de ejecutar acción mediante los músculos así como también sensar que es lo que está ocurriendo a nuestro alrededor conformando nuestros sentidos. Con la adición de identificar aspectos posturales mediante la identificación de la elongación y posición relativa de nuestro cuerpo mediante la propriocepción.

El sistema conformado se encuentra así dividido en dos subsistemas autonómicos *que llevan la vida adelante* simpático y parasimpático. El primero autonomiza la respuesta de *fight or flight*, pelear o luchar en tanto que el segundo regula la homeostasis y el metabolismo resumido en *rest and digest* [81, 82].

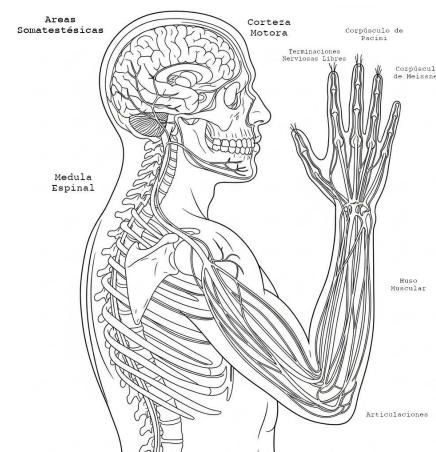


Figura 15.1: El Sistema Nervioso Central y el Sistema Nervioso Periférico.

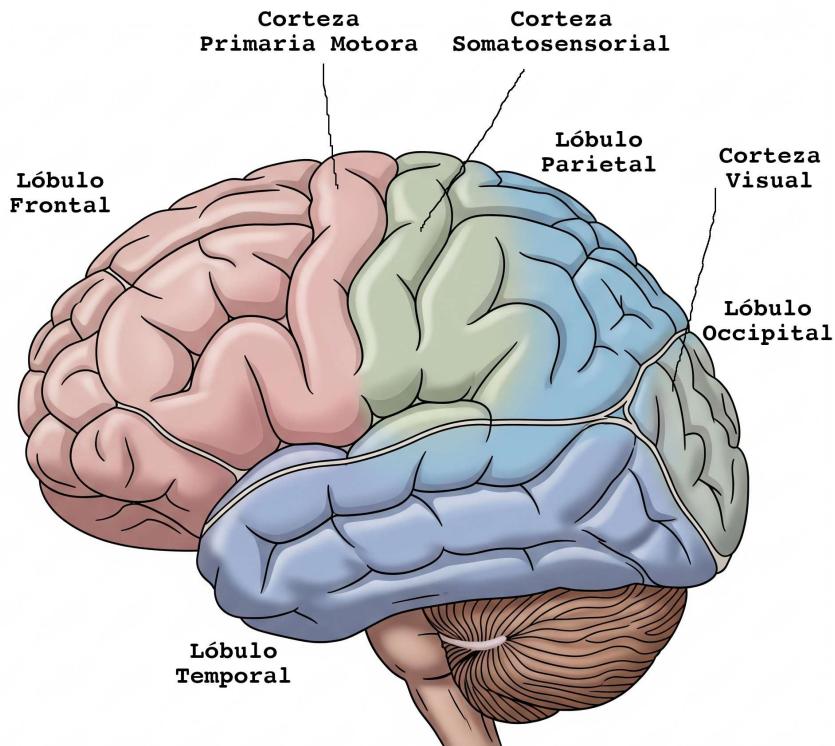


Figura 15.2: Diferentes regiones anatómicas y funcionales del cerebro.

### 15.1.1 Grasa inteligente

El cerebro es físicamente una colección de tejido con alto contenido de ácidos grasos compuesto de materia gris en la corteza, y materia blanca que corresponden a los axones neuronales. Se encuentra embebido en el líquido cerebroespinal, contenido dentro del cráneo. Este líquido es generado en tres ventrículos que se encuentran en el centro del cerebro y es reabsorbido en las meninges, la capa externa al cerebro, en contacto con la corteza, y con la dura-mater, justo en el inicio del cráneo [?].

El cerebro a su vez se divide en diferentes áreas. En la capa exterior está la corteza, el área más nueva del cerebro, más desarrollada comparativamente en los seres humanos. Aquí está la base del comportamiento abstracto. La figura 15.2 muestra las diferentes divisiones de la corteza, muchas de ellas asociadas a rasgos funcionales de cada área [83].

Diferentes hendiduras surcan la anatomía del cerebro, donde se destaca la hendidura central que divide al cerebro en dos hemisferios. El cuerpo calloso es literalmente un manojo de fibras que conectan ambas partes y su grosor es un aspecto distintivo del ser humano, no tan presente ni marcado ni siquiera en otros primates [84].

De la divisiones en hemisferios y de las divisiones en diferentes secciones funcionales surge el principio del cerebro rígido conexiónista, donde cada parte realiza una tarea específica funcional. Un daño en una parte implica, la perdida o compromiso de su rol funcional. Esta idea dominó la neurociencia por muchos años, muy influenciada por el hecho de que el propio estudio y descubrimiento surgía de casos concretos donde una persona tenía comprometida o dañada, por enfermedad o traumatismo, una parte del cerebro, afectando algún aspecto comportamental. Esa

correlación sugería que esa zona era la encargada de ese aspecto comprometido. Claros ejemplos de esto son el propio y famoso Phineas Gage y Luis Victor Leborgne [85].

### 15.1.2 Una neuro-computadora

Podemos hacer una analogía metafórica entre el cerebro y una computadora actual [86] y describir sus diferentes partes desde esa perspectiva.

#### Firmware

Estructuras internas en el cerebro, más primitivas e histológicamente diferentes, son conformadas por el cerebelo y el pons o puente de Varolio. Estas son las regiones donde se regulan principalmente los sistemas autonómicos. Es el **firmware** del cerebro.

#### GPUs, Graphical Processor Units

La corteza occipital, ubicada justo arriba de la nuca, es la encargada de procesar la información visual, tal como una placa gráfica en un ordenador. Se encuentra conectada a los ojos por el nervio óptico que en el camino atraviesa áreas subcorticales como el tálamo visual, donde actúan mecanismos del sistema parasimpático que disparan por ejemplo reacciones defensivas (esta es la razón por lo que sin pensar cuando nos aplauden al lado pestañamos o realizamos un gesto evasivo, una reacción).

El área occipital es muy relevante desde el lado biomimético porque de su estudio se derivaron muchas de las ideas que son actualmente las bases de diversos mecanismos aplicados en redes neuronales, típicamente las redes convolucionales que se detallan en el capítulo 18.

#### NLP, Natural Language Processing

Uno de los impulsos más grandes que recibió la neurociencia moderna fue con el descubrimiento de Paul Broca de la corteza lingüística. Esto surgió en base a observaciones de cómo lesiones en esa área afectaban particularmente el funcionamiento normal en el procesamiento y generación del lenguaje [85]. Así entonces, el *área de Broca* así como también el *área de Wernicke* están relacionadas a la capacidad humana de procesamiento y generación del lenguaje natural. Son precisamente por ello, los centros donde se localiza el NLP humano<sup>1</sup>.

#### CPU, Central Processor Unit

La corteza prefrontal es también un área característica evolutiva y comparativamente altamente desarrollada en el ser humano, y concentra el control ejecutivo, social, el centro de las decisiones. Es dónde el **yo** está depositado. Se realiza el control consciente del pensamiento y las acciones de acuerdo a un objetivo consciente interno.

#### El motherboard

Las áreas que se encargan de sentir el tacto de nuestras extremidades, el área somatosensorial, y las regiones encargadas de coordinar nuestros movimientos, las áreas premotoras y motoras, se encuentran una al lado de la otra en la zona temporal y central.

Desde el área central, en la propia hendidura, las diferentes zonas en la corteza de estas tres áreas van mapeando las diferentes partes del cuerpo, en relación a lo que sensan y lo que controlan. Si representamos otra vez en el cuerpo humano, el porcentaje de estas zonas que está destinado a mapear cada una de las partes del cuerpo, podemos visualizar el famoso *homunculus* (Figura 15.3) donde se ve cuánto de cada área de estas zonas se le dedica a cada parte. Por ejemplo, las manos son grandes porque hay una proporción mayor de regiones en estas áreas que se encarga de percibir y controlar las manos.

<sup>1</sup>Si bien actualmente las computadoras no tienen un componente físico de NLP, seguramente en los años venideros veremos componentes de NLP embebidos directamente en el hardware, con algún motor de lenguajes interno incorporado, esto como la extensión natural de los TPU.

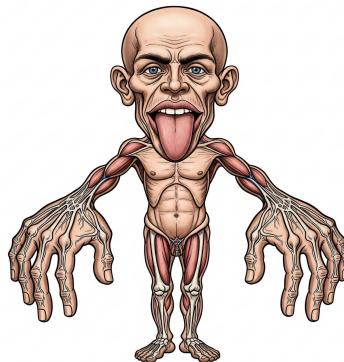


Figura 15.3: Homunculus

### La RAM

El hipocampo, central en el cerebro, es un área clave en el sueño y en el almacenamiento y la generación de la memoria. Es donde se produce la neurogénesis [87], el nacimiento y especialización de nuevas neuronas.

### La fusión sensorial - Tempo Parietal Junction

La unión de la información sensorial se coordina en la unión tempo parietal, el TPJ, que coordina la información recibida por diferentes modalidades: proprioceptiva, visual, vestibular, y las combina para dar una única e integrada idea de la realidad. Esto es equivalente a las técnicas de fusión de sensores [88] que comúnmente se buscan en robótica.

## 15.2 La Neurona Recargada

Retomando la presentación del modelo de neurona de los capítulos 6 y 7 en este se ofrecen más detalles sobre su complejidad y estructura, que muestran que no es tan simple como allí estaba planteado [89].

### 15.2.1 ¿Cómo dispara una neurona?

El disparo de la neurona es mucho más complejo que el planteado hasta el momento[90]. Uno de los modelos más establecidos que intenta capturar esa complejidad fue el que postularon Hodgkin-Huxley [91]. Ellos estudiaron como se comportaban los disparos neuronales en el axón gigante del calamar, y establecieron un modelo dinámico que explicaba el comportamiento como un potencial de acción de equilibrio inestable que es coordinado, de manera activa, por corrientes iónicas.

$$\begin{aligned} I &= -(C_m \frac{dV}{dt}) + \frac{Ie}{A} \\ I &= C_m \frac{dV_m}{dt} + g_K(V_m - V_K) + g_{Na}(V_m - V_{Na}) + g_l(V_m - V_l) \end{aligned}$$

Lo que esta ecuación diferencial establece que es que la corriente que circula de un lado al otro de la membrana celular, y la que en definitiva determina la aparición del potencial de acción súbito, es proporcional a la tasa de variación del voltaje en el tiempo, y a la diferencia de voltaje existente entre el voltaje actual y voltaje de referencia de tres referencias distintas. Cada una de

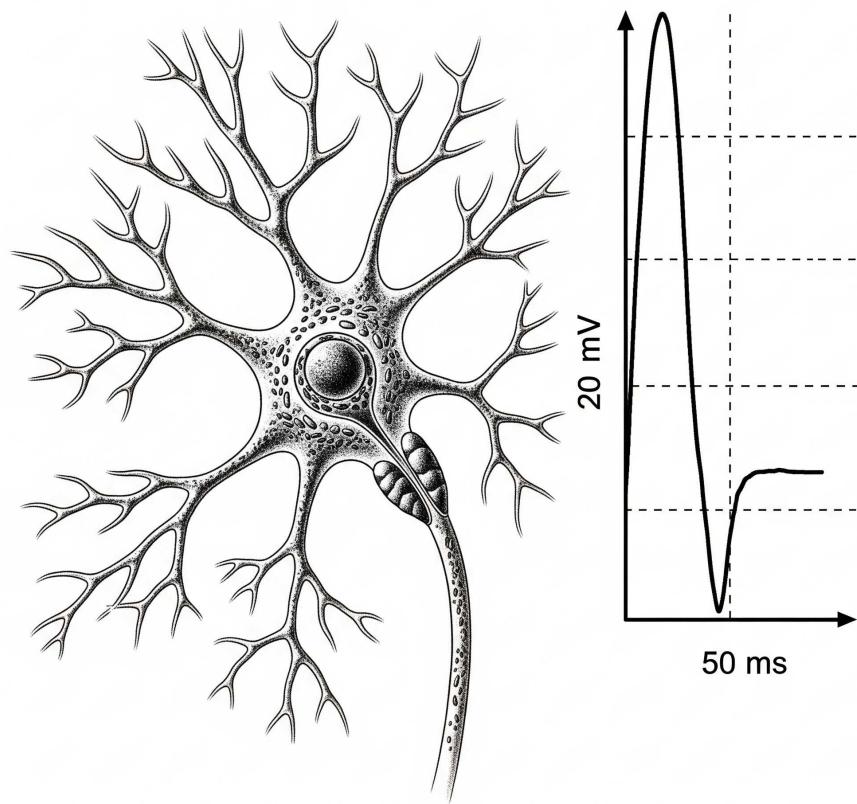


Figura 15.4: Neurona pistolera disparando.

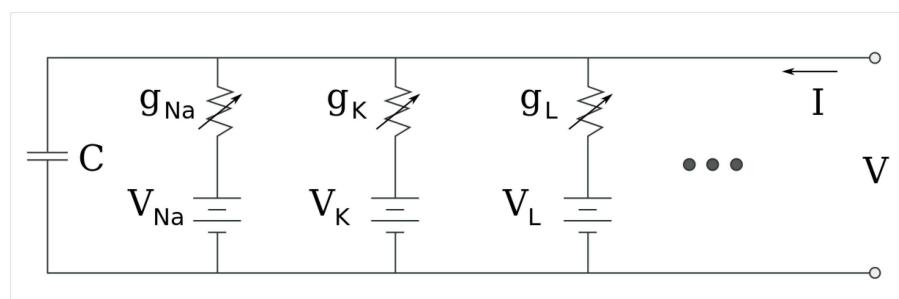


Figura 15.5: Diagrama de Kirchhoff que representa el modelo de Hodgkin-Huxley.

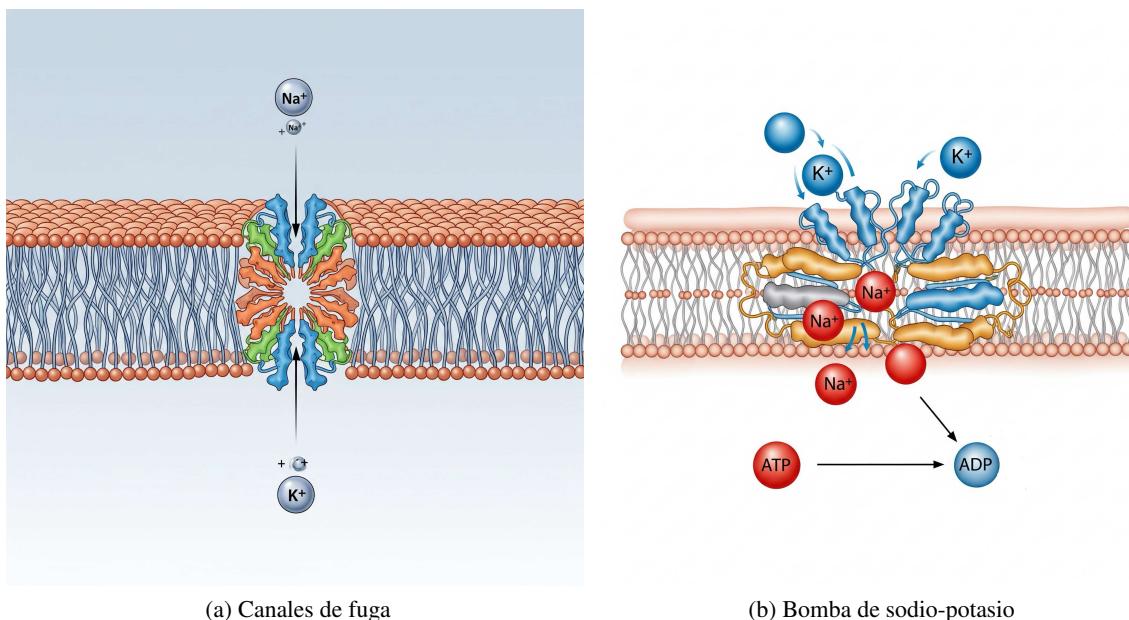


Figura 15.6: Proteínas de membrana que mantienen pasiva y activamente el potencial de membrana.

estas mediadas por conductancias, el reciproco de la resistencia eléctrica. Experimentalmente, H & H encontraron que cada uno de estos estaba mediado por concentraciones de iones de potasio (K), iones de sodio (Na) y iones de fuga, también llamados *leakage*. Ellos descubrieron que la dinámica del disparo del potencial de acción, estaba regulada por estos iones, átomos cargados eléctricamente por un desbalance de electrones, que se desplazaban de manera transversal a la dirección de propagación del potencial, y que su modelo explicaba el comportamiento todo-nada, generado por la propia dinámica súbita del intercambio de iones.

### 15.2.2 Una Computadora Nanotecnológica

La neurona mantiene pasiva y activamente una diferencia de potencial entre el interior y el exterior de la célula de alrededor de -90 mV. Esta diferencia viene dada justamente por el filtrado selectivo que hace la membrana plasmática por la cual los diferentes iones no pueden pasar directamente. Existen canales que son proteínas de membrana (fig 15.6) que actúan como mediadores para mantener este potencial. Los canales de fuga permiten que el sodio+ ingrese en la célula y que el potasio+ salga al exterior, actuando de manera pasiva simplemente por afinidad electroquímica. Por otro lado, la bomba de sodio-potasio, positiviza el interior de la célula, sacando 3 sodio+ e ingresando sólo 2 potasio+, pero lo hace de manera activa consumiendo ATP, la moneda de energía biológica.

Estas proteínas de membrana tienen la importante característica que la sensibilidad electroquímica a los iones es dependiente del voltaje. Este aspecto es el que permite que ante un pequeño cambio en el potencial eléctrico, cambia la conformación de estas proteínas, cambia entonces la conductancia y esto genera una rápida corriente de iones, hacia adentro y hacia afuera de la neurona, que es lo que propaga el potencial de acción, la espiga. Esto se visualiza en la figura 15.8, donde se ve la linea de tiempo de izquierda a derecha, y el comportamiento sucesivo de los dos canales. En el inicio, un pequeño cambio en el potencial de acción de equilibrio en reposo (-90 mV) es suficiente para generar una apertura de una compuerta del canal para que comience a ingresar masivamente cationes de sodio que positivizan el interior de la neurona llevando el potencial hacia el cero. Este flujo rápido, provoca la *despolarización*, donde se produce una sobreestimulación que lleva el

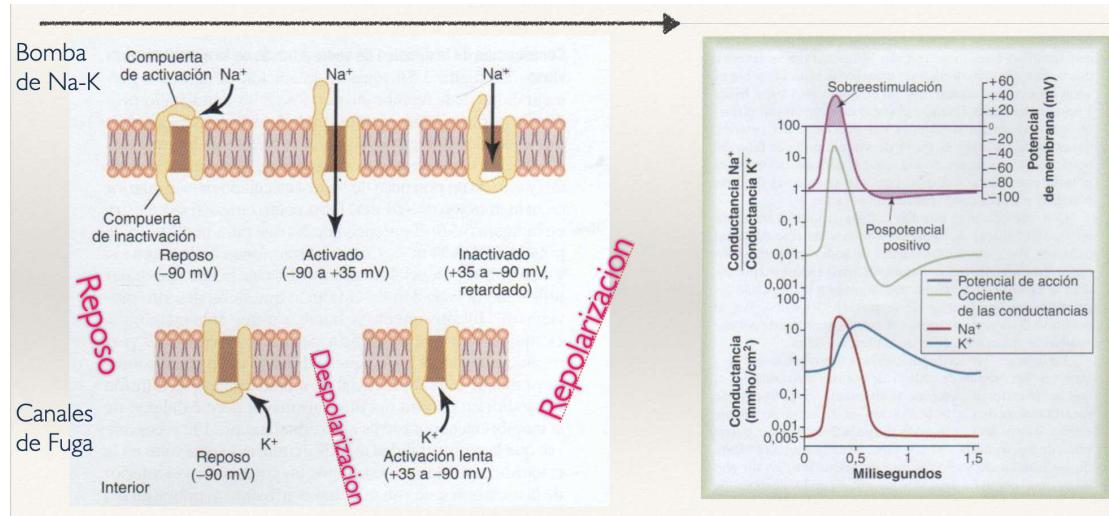


Figura 15.7: Secuencia de intercambio de iones a través de las proteínas de membrana que generan la espiga del disparo neuronal.

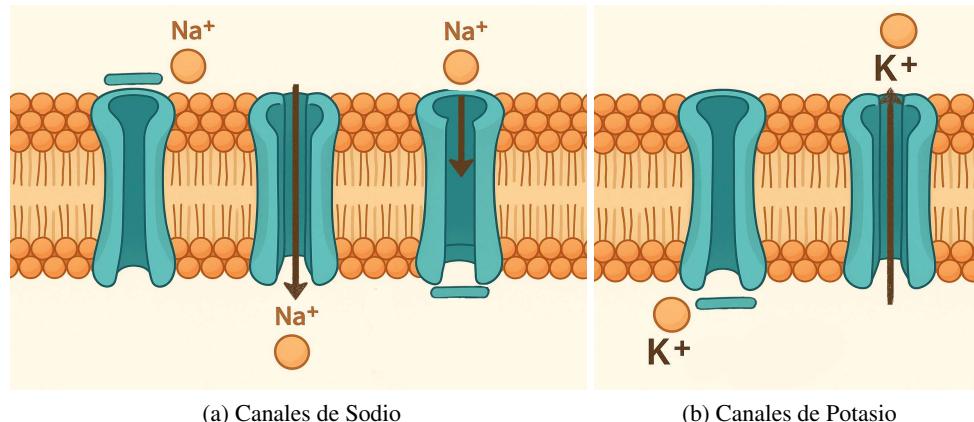


Figura 15.8: Secuencia de intercambio de iones a través de las proteínas de membrana que generan la espiga del disparo neuronal.

potencial de membrana a +40, lo cual corresponde al pico de la espiga. En paralelo, en los canales de fuga, el cambio en el voltaje cambia la conformación del canal, lo cual abre una compuerta para que comiencen a salir cationes de potasio que negativizan el interior, pero lo hacen con un retardo, son más lentos. Esta acción retardada, es lo que permite que el potencial de acción comience a retornar al valor de equilibrio de -90, que se manifiesta finalmente con el cierre nuevamente de las compuertas en los canales, que retornan toda la situación al estado de equilibrio donde se produce la *repolarización* [89].

### 15.2.3 Conducción a Saltos

El tamaño de una célula, considerando el soma y las dendritas es del orden de los  $20\mu\text{m}$ . Sin embargo los axones pueden ser largos, muy muy largos, de varios metros. Por ejemplo, salen del cerebro donde vive la neurona, viajan por la médula espinal, se conectan a una segunda neurona y llegan a una terminación nerviosa. Es una gran distancia por la que debe viajar la espiga del disparo neuronal. Sin embargo, el potencial de acción que se propaga por la membrana plasmática tiene una baja velocidad de propagación  $0.25 \frac{\text{m}}{\text{s}}$  y además tiene mucha perdida. Entonces para que los

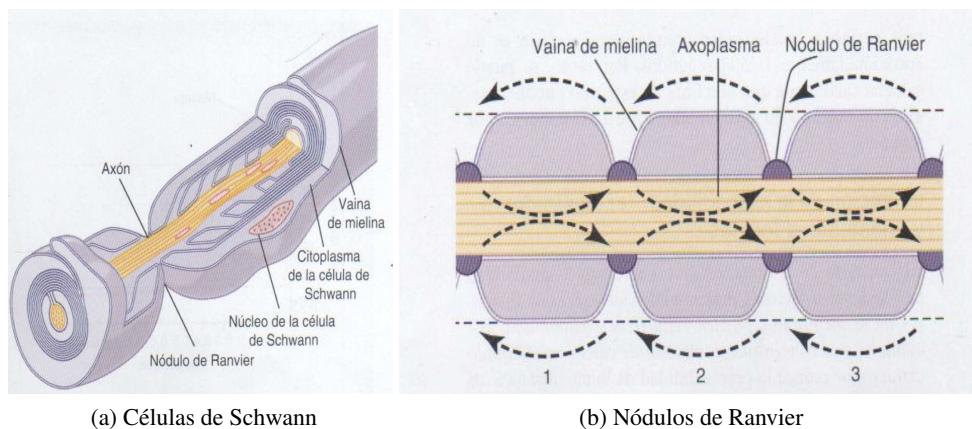


Figura 15.9: Conducción Saltatoria mediada por la aislación de las vainas de mielina y por los *repetidores* en los nódulos de Ranvier.

axones de las neuronas puedan propagarse las grandes distancias del cuerpo humano se requiere de un mecanismo para permitir transmitir las espigas. Esto se logra mediante un grupo de células, las células de Schwann o vainas de mielina, que recubren el axón, favorecen su aislamiento, de la misma forma que en ingeniería desarrollamos aislantes plásticos para los cables. Adicionalmente, estas células se acomodan sobre el axón de forma que dejan lugar para unas estructuras bastante equiespaciadas, denominadas Nodos de Ranvier, que actúan como repetidores para regenerar la la espiga con mayor amplitud y así permitir propagarla por mayor distancia.

#### 15.2.4 Sinapsis

Las neuronas buscan conectarse con otras neuronas todo el tiempo. Una neurona que no se conecta con ninguna otra tiende a morirse. El punto de esa conexión es la sinapsis neuronal (Fig. 15.10), donde una neurona presináptica se conecta con una neurona postsináptica. El potencial de acción viaja por el axón hasta el terminal presináptico, donde hay vesículas transmisoras que contienen un neurotransmisor (una proteína). Cuando el potencial llega a la hendidura presináptica, proteínas de membrana desarmen las vesículas, liberando el neurotransmisor en la hendidura sináptica (de 200 a 300 angstroms). Hay más de 40 diferentes tipos de neurotransmisores, que al flotar libremente por la hendidura sináptica por afinidad electroquímica se ligan a canales por ligando que están ubicados sobre la membrana de la neurona postsináptica. Al ligarse con el neurotransmisor, estos canales se abren, generan un intercambio iónico y disparan un nuevo potencial de acción que ahora viaja por la neurona postsináptica, propagandolo, o fuerzan un cierre de los canales iónicos provocando un efecto inhibitorio.

Los neurotransmisores, con su conformación en tres dimensiones, son gran parte del abanico farmacológico de la psiquiatría que se centra en

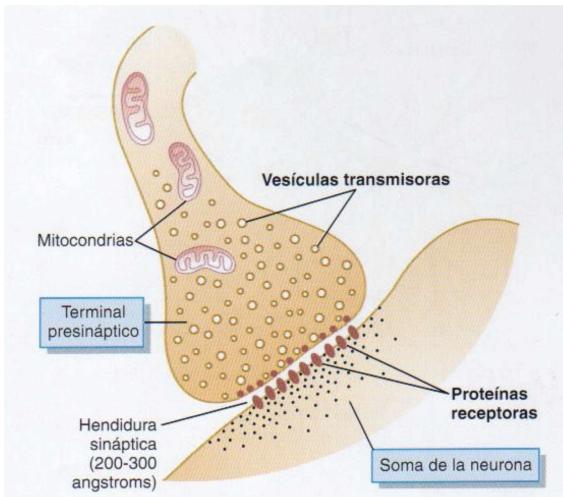


Figura 15.10: La sinapsis neuronal, lo que puede fallar en el final.

producirlos o producir drogas similares que también pueden operar sobre la sinapsis mediante afinidad electroquímica, cual tetris, con los canales por ligandos. Así en la sinapsis, el mecanismo Hebbiano de aprendizaje se manifiesta mecanísticamente en muchos niveles. De base, el disparo coordinado de la neurona presináptica y postsináptica refuerza la sinapsis, mediante el reclutamiento de más vesículas, mayor actividad mitocondrial (y por ende de producción energética para mantener el potencial de acción) y también más canales por ligando en la neurona postsináptica. Pero no solo se limita a estos cambios directos, sino también a lo que se denomina activaciones por segundo mensajero, que son cambios enzimáticos, activación y transcripción de nuevos genes, o incluso cambios estructurales. Intentar capturar y reducir toda esta complejidad maravillosa en un sólo número es al menos una suposición audaz.





## 16. Aprendizaje Profundo

¿Por qué Deep? ¿Por qué Profundo? La verdadera razón es marketing. Como ocurrió, ocurre y ocurrirá, la tecnología es una piedra angular del mundo actual y necesita tener estrategias de comercialización. Muchas veces, viejas tecnologías reciben nuevos nombres, *buzzwords* o neologismos, para identificarla, distinguirla de otras, estructurar un mensaje sobre esa palabra, hacerla moda, y vender más en el fondo [92]. A las redes neuronales, una disciplina muy académica, sumida en cierto ostracismo en el inicio del siglo 21<sup>1</sup> le ocurrió exactamente eso y recibió un nuevo nombre, muy marketinero, que combinaba una característica cierta de las redes neuronales artificiales, el esquema jerárquico algomericativo, con un significado fuerte a caballo de una estrategia comercial.

### 16.1 Guía para el tío y la tía que pregunta

¿Qué es Deep Learning? Es un desarrollo de ciencias de la computación, pero a la vez muy interdisciplinario, con bases biomiméticas que utiliza una estructura jerárquica aglomerativa.

Los siguientes son mitos, extractos literales tomados de diferentes medios, y hasta incluso libros, de conceptos errados en relación a lo que es la Inteligencia Artificial, y en específico Deep Learning:

**Los algoritmos de IA no se programan:** mito influenciado probablemente por la frase de Arthur Samuel en 1959, "[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed" [93]. Es decir no se programa el paso a paso de cómo resolver la tarea, sino que se implementa un algoritmo genérico parametrizado, que a su vez incorpora un procedimiento para ajustar esos parámetros. Todo eso, por supuesto, hay que programarlo.

---

<sup>1</sup>Los ganadores del premio Turing de 2019, Yann LeCun, Geoffrey Hinton y Yoshua Bengio, mencionan que ellos se esforzaban por publicar sus trabajos de redes neuronales disimulados en revistas de Computer Vision, la estrella de esa era, justamente para poder acceder a factores de impacto más alto asociados con las revistas científicas que publicaban esos temas.

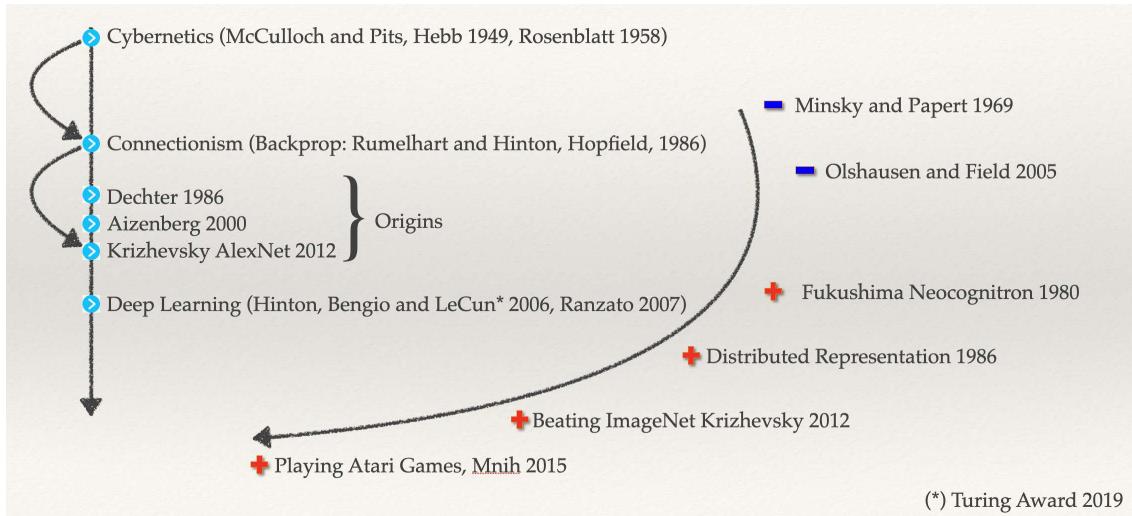


Figura 16.1: Los trabajos que determinaron los distintos veranos de la inteligencia artificial (+) y aquellos que cimentaron los dos anteriores inviernos de la IA (-). La primer era fue la cibernetica, la segunda la conexionista, y podemos afirmar que esta tercera es la era de las redes profundas.

**Los robots vienen a reemplazarnos:** mito por dos razones. La primera es que los avances actuales en IA no implican automáticamente avances similares en robótica; es muy complejo enfrentarse al mundo natural, al mundo físico. La segunda es que la idea del reemplazo es un concepto más filosófico-sociológico y no un concepto tecnológico en sí (ha ocurrido en el pasado y seguirá ocurriendo esta visión de que una tecnología en particular reemplazará a los seres humanos).

**La tecnología empezó a reemplazarnos, y más nos vale sacarle provecho que quedarnos amargados por la herida que nos provoque:** en línea con el anterior, es el mismo mito del reemplazo, pero este agrega una situación de culpa a los seres humanos, sin sentido alguno.

**Los algoritmos tienen además la ventaja que no se cansan y pueden interactuar con el contexto a velocidades supersónicas en comparación con las personas:** este mito, además de las imprecisiones que conlleva, es tal porque intenta inútilmente comparar una persona, un ser humano, con un algoritmo. Esto es un sinsentido, con un alto grado de trivialidad y lleva a un debate erróneo.

¿Es esta tecnología el advenimiento de las máquinas inteligentes o de los robots que todo lo controlan, o peor, los robots que inicián el apocalipsis robot? No, estamos lejísimos aún de esa situación [94]. Generar inteligencia como la que nosotros identificamos en otros seres o en los seres humanos es excesivamente difícil [95].

En concreto es una tecnología revolucionaria que hoy está en el núcleo y centro de la frontera digital y que se caracteriza por haber logrado avance en las siguientes áreas [96]:

- Computer Vision:
- Natural Language Processing:
- Speech Recognition:

Los tremendos avances en esta área están dominando la escena actual, y todo lo que hoy llamamos AI se centra exclusivamente en aplicaciones en estas áreas, como por ejemplo:

- Control magnético del plasma de un reactor de fusión tokomak mediante aprendizaje por refuerzo profundo [97].

- Descubrimiento de los restos de un barco antiguo hundido que no podía encontrarse [98].
- Desarrollo de nuevas formas más eficientes de multiplicación de matrices [99].
- Alphafold: la red de deep learning que permite predecir la estructura tridimensional, y por ende su funcionalidad, en base a la secuencia de aminoácidos [100].

Puntualizando entonces, el área actual de Deep Learning está caracterizada por cuatro pilares, cuatro componentes claros que en verdad la caracterizan. Estos cuatro pilares de Deep Learning son:

- Deep, Profundidad: Múltiples Niveles de Composición
- Feature Learning: Aprendizaje de la Representación
- Datasets masivos y estandarizados
- GPUs, Graphic Processing Units, hardware especializado para el procesamiento paralelo de multiplicación de matrices.

## 16.2 Profundidad

### 16.2.1 Los problemas del Shallow Learning

Durante los 90s y entrando en los 2000s, el foco de la inteligencia artificial estaba centralizado en Computer Vision y en algoritmos orientados a datos basados fuertemente en estadística. Esta es el área tradicional de **Aprendizaje Automático**.

Las estrellas de esos años fueron complejos algoritmos que intentaban entender y decodificar de qué se trataba el procesamiento de imágenes y la visión por computadora. Esquemas muy sofisticados se desarrollaron que fueron muy exitosos para problemas pequeños y controlados pero fallaban cuando el problema escalaba o el nivel de ruido crecía [101]. Lo mismo ocurrió inicialmente para resolver el problema del ajedrez, o el procesamiento del habla. Se buscaba entender las particularidades de cada caso, intentando plasmar algorítmicamente cómo los seres humanos creemos que entendemos estos problemas. Sin embargo, este abordaje sólo era válido para demostraciones acotadas y fallaban al escalar o ante problemas más realistas.

Una de las razones por las cuales *estos métodos tuvieron un éxito limitado* está relacionada con el problema de la alta dimensionalidad. Estos métodos suelen basarse en composiciones de features (ver 2.1) y estos features tienen dimensiones muy altas. Y esto **hackea** nuestra intuición.

#### La maldición de la alta dimensionalidad

Supongamos que tenemos un círculo circunscripto en un cuadrado. El diámetro del círculo coincide con el lado del cuadrado. Si medimos la superficie cubierta por ambos, **intuitivamente**, tendremos la percepción de que son prácticamente iguales, y que sólo en los bordes pequeños existe una zona que queda por fuera del círculo pero que es cubierta por el cuadrado. Si lo extendemos a una esfera circunscripta en un cubo, esta percepción se mantiene. Si seguimos extendiendo esto a más dimensiones, vemos que la tendencia es en realidad otra. Esto se puede ver en la figura 16.2a donde se grafica el hipervolumen cubierto por el cubo, netamente exponencial, y el hipervolumen cubierto por la hiperesfera, hipergeométrico, ambos en función de la cantidad de dimensiones. No solo se ve que hipervolumen del cubo crece **exponencialmente** de manera absoluta y comparativamente contra el de la hiperesfera sino que además la hiperesfera comienza a **decrecer!** alrededor de la dimensión 21. Esto es un delirio al sentido común, y muestra que la hiperesfera circunscripta en altas dimensiones ocupa una parte infinitesimal del volumen disponible [102].

Asumir que la consistencia local se mantiene en todo el espacio disponible puede no ser cierta en altas dimensiones, ya que los datos existentes son esparsos, pueden estar distribuidos en un

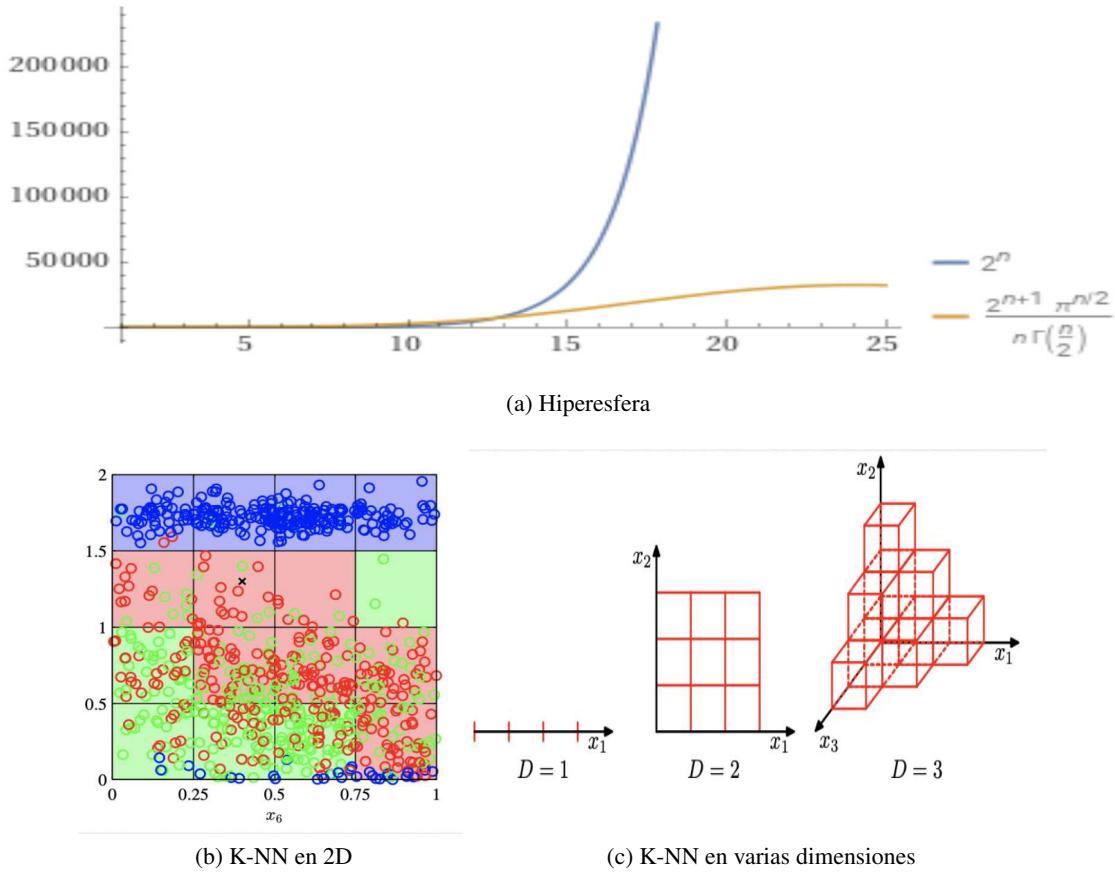


Figura 16.2: La maldición de la alta dimensionalidad.

hiperespacio totalmente vacío, y generan poca estructura de la cual agarrarse. Esto se ve bien en la figura 16.2b: esto es el típico algoritmo de knn [103]. Cada cuadro en el gráfico corresponde a un color dado por el voto mayoritario de los elementos que ya están ubicados en cada uno. Sin embargo, al extender esta idea a más dimensiones (fig 16.2c), comienzan a construirse hipercubos que generan más y más espacio disponible y en el punto extremo quizás haya sólo uno elemento por cubo o incluso ninguno en absoluto. Esto además enfatiza con claridad que la necesidad de datos para generar la estructura que revela los patrones inherentes crece exponencialmente con la dimensionalidad del feature, imponiendo una enorme limitante a la escalabilidad de estos métodos.

### 16.2.2 El Problema Fundamental de las Redes Neuronales

Uno de los resultados más fundamentales de redes neuronales, está dado en el teorema 16.2.1. Este teorema probado por Cybenko en 1989 [104] establece que:

**Teorema 16.2.1 — UAT - Universal Approximation Theorem.** Un MLP con una única capa oculta que contiene un número finito de neuronas puede aproximar a cualquier función continua en un subconjunto compacto de  $\mathbb{R}$  dadas funciones de activación apropiadas.

Sea  $C(X, \mathbb{R})$  un conjunto de funciones continuas de  $\mathbb{R}^n$  a  $\mathbb{R}^m$ . Sea  $\sigma \in C(\mathbb{R}, \mathbb{R})$ , con  $(\sigma \circ x)_i = \sigma(x_i)$  representando la función de activación.

Entonces, para cada  $n \in \mathbb{N}, m \in \mathbb{N}$ , un conjunto compacto  $K \subseteq \mathbb{R}^n$ ,  $f \in C(K, \mathbb{R}^m)$ ,  $\epsilon > 0$  existe

$k \in \mathbb{N}, A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{m \times k}$  de forma tal que

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon \quad (16.1)$$

con  $g(x) = C(\sigma \circ (A.x + b))$

Este resultado implica que eventualmente basta con una red de una sola capa oculta para representar cualquier tipo de función computable[105]. Es un aproximador universal. Esto tiene la ventaja que no requiere aplicar ningún tipo de intuición sobre las particularidades del problema y esquiva, un poco, la problemática de la alta dimensionalidad de los features. Simplemente, con entrenamiento suficiente, debería ser capaz de encontrar los valores de los pesos adecuados y aproximar cualquier cosa.

Pero, si con una red neuronal de una sola capa, puedo aproximar cualquier cosa, ¿qué gano poniendo muchas capas? La ventaja de la profundidad viene dada porque la estructura composicional, jerárquica aglomerativa, ofrece dos beneficios: la invarianza a las traslaciones y la localidad a las diferentes escalas. Esto se visualiza muy bien con las redes convolucionales. La invarianza a las traslaciones representa que sin importar en qué lugar de la entrada exista una característica importante en los datos, la naturaleza composicional y aglomerativa de una red que tiene muchas capas, permitirá que esa característica se capture en algún lugar de la jerarquía. El segundo punto, la localidad a diferentes escalas, permite ver de manera indistinta eventos que ocurren a diferentes escalas y capturar sin importar la escala: son invariantes a ella. Esto es **anillo al dedo** para muchos eventos naturales, particularmente en imágenes:



Many natural questions on images correspond to algorithms that are compositional

Es decir, existen muchos problemas en la naturaleza que presentan la necesidad de tener invariantes en el desplazamiento (en el tiempo o en el espacio) e invariantes en la escala. Un ejemplo de este último son los mapas de las costas, donde sin importar la escala, las costas lucen muy similares.

Adicionalmente, hay una perspectiva más práctica que se presenta y es que las redes neuronales más profundas son en general más fáciles de entrenar de manera efectiva, más eficientes en encontrar los óptimos (más fáciles de optimizar) y generalizan mejor el problema. Es decir, son razones muy pragmáticas [106].

Pero esto no es nuevo. Se sabía que las redes neuronales tenían que ser adecuadas para resolver muchos de los problemas tradicionales del mundo natural, pero algo no cerraba. Puntualmente existía lo que se llamó **El problema fundamental de las redes neuronales** que consistía en lo siguiente: cuando se aplica el algoritmo de backpropagation en muchas capas, los errores que se propagan pueden ser tan chicos que caen por fuera de la resolución computacional, y eso provoca que eventualmente los gradientes se vuelvan cero o exploten. Es decir aparece un problema numérico [107]. Una serie de trabajos abordó esta problemática entre los que se encuentra la idea de Unsupervised Pretraining [108] que es donde se acuña el nombre **Deep** que se subió al caballo del hype y que renombró totalmente el área.

Sin embargo, todos estos avances tenían por detrás un problema mayor y era que el hardware necesario para realizar implementaciones escalables y efectivas de redes neuronales profundas simplemente no existía [109].

## 16.3 Aprendizaje de la Representación

Este segundo pilar de Deep Learning se visualiza cuando consideramos el esquema arquitectónico de la figura 16.3 que detalla cómo operan los algoritmos de aprendizaje automático.

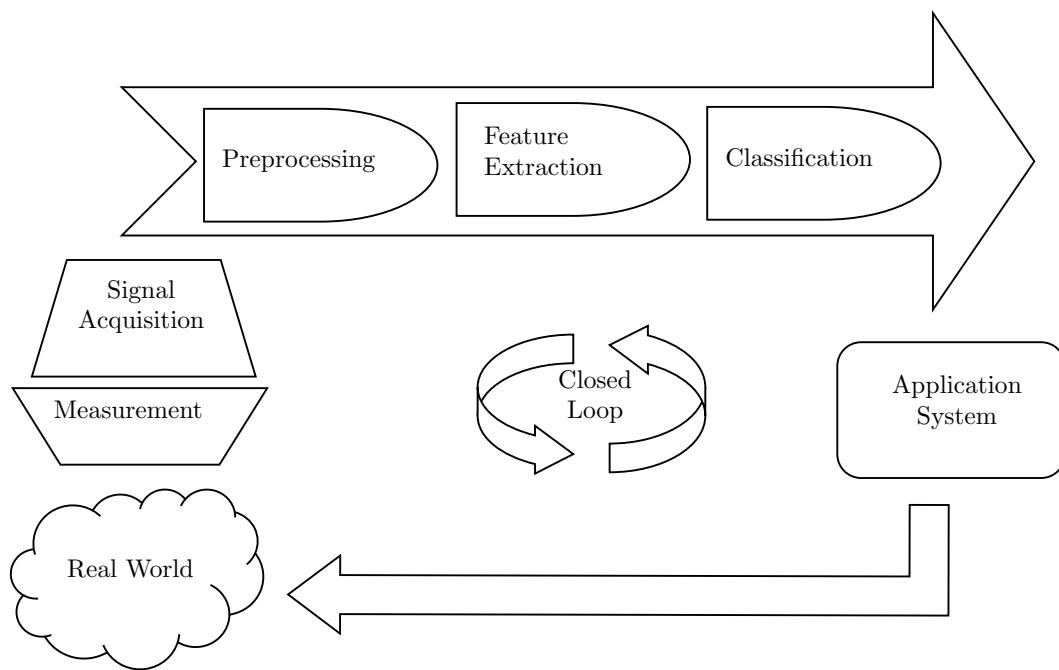


Figura 16.3: Arquitectura de un modelo de procesamiento basado en aprendizaje automático

Una parte importante de este proceso es la ingeniería de la característica o **feature engineering**. Esto es el diseño del vector que codifica y sintetiza la información más importante que precisamente se necesita para capturar el patrón. Las redes de deep learning tienen la característica que delegan la identificación de cómo tiene que ser ese feature, directamente al propio proceso de aprendizaje, forzando aprender pesos sinápticos que justamente son los que determinan qué feature utilizar y cómo es el mismo. Esto genera, que las redes de deep learning, reciben como entrada directamente la información con algún preprocesamiento pero mayoritariamente cruda, con las mismas variables y dimensiones que naturalmente tiene. La red asume, dentro de todo el proceso de aprendizaje, sintetizar de alguna manera la información guiada por el propio objetivo plasmado en la función de costo. Este esquema se visualiza claramente en las redes convolucionales (capítulo 18).

## 16.4 Datasets

El tercer pilar de Deep Learning tiene que ver con la masiva acumulación de datos provocada por la transformación digital de toda la sociedad. Esta fue iniciada con las computadoras personales, las fiebres de la .COM, el auge de los celulares, las tablets, las apps y ahora la IA. Tenemos datos masivos que prácticamente involucran a todas las sociedades humanas y potencialmente cada persona del mundo, y muchísimos procesos del mundo, son generadores de datos.

Adicionalmente, la existencia de datasets estandarizados, ayudó y promovió la colaboración en el área y el poder trabajar comparando diferentes métodos sobre una métrica común dada por los propios datos.

## 16.5 GPUs

Los procesadores actuales, los CPU, Central Processor Units, están basados en desarrollos y avances de muchos años, ordenados por la ley de Moore. Esta profecía autocumplida planteaba que

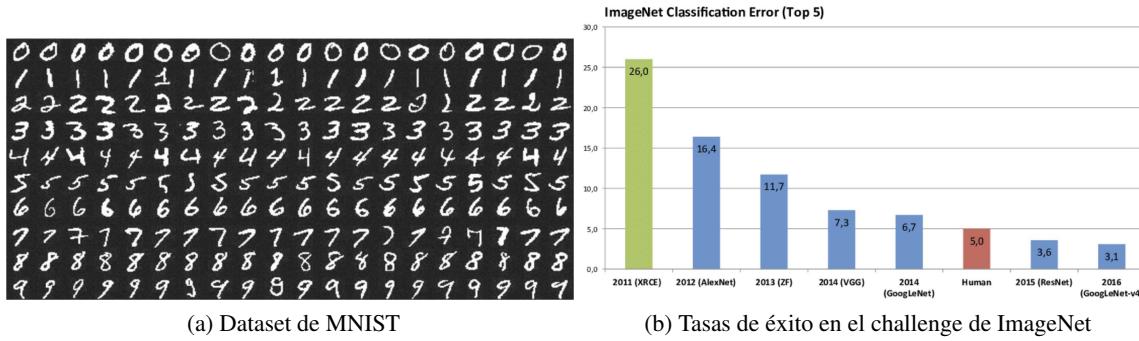


Figura 16.4: Los datasets de MNIST e ImageNet se utilizaron durante muchos años para verificar y probar la calidad de los algoritmos de clasificación.

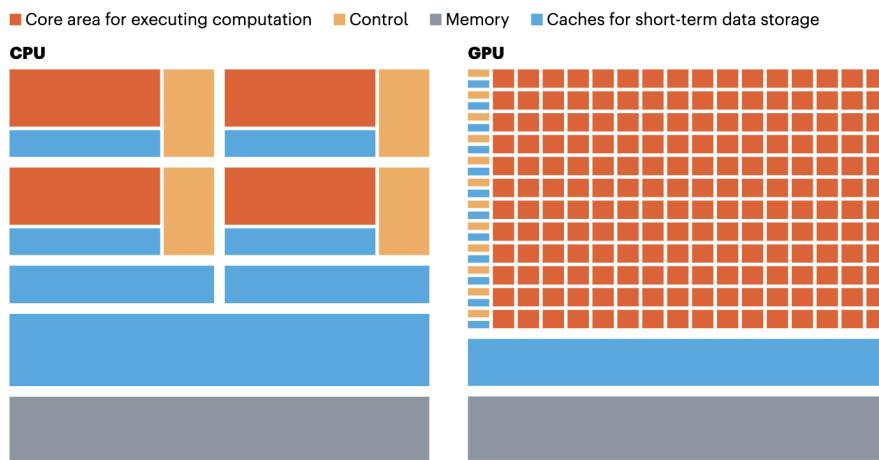


Figura 16.5: Arquitecturas de los CPUs, más complejo y especializada, versús los GPUs más simple y más paralela [110]

cada dos años, se iban a duplicar la cantidad de transistores contenidos en un circuito integrado, y con eso se iban a ganar incrementos en la performance y capacidad de los procesadores. Ese crecimiento llegó a su fin, porque se alcanzaron los límites físicos de la miniaturización. En paralelo, la industria de los videojuegos, muy lucrativa, comenzó a necesitar una masiva necesidad de computo en paralelo, y para ello se desarrollaron los GPU, Graphic Processor Unit, con una arquitectura diferente, más simple, pero enfatizando el paralelismo masivo (Figura 16.5). Esto permitía realizar los cálculos de procesamiento gráfico, que requieren multiplicación de matrices en paralelo para mover y rotar polígonos, así como también para las operaciones de geometría proyectiva (más multiplicación de matrices).

En 2012 Hinton se dio cuenta que las operaciones de las redes neuronales podían aprovecharse del hardware de las GPUs que básicamente estaba diseñado para el mismo tipo de operación computacional, multiplicación paralela de matrices, y decidió junto con su estudiante de doctorado, Alex Krizhevsky usar una playstation para implementar una red neuronal convolucional, AlexNet, e intentar resolver el challenge de ImageNet [110]. Su éxito fue tan rotundo, que el año siguiente todos los participantes del challenge hicieron propuestas basados en el mismo esquema.

Goodfellow lo describe y el texto es excelente en relación a esto [111]:

**R La industria de los videojuegos financió el hardware necesario para que Deep Learning**

**se convierta hoy en la estrella y motor de la revolución de la inteligencia artificial.**

Esto se debe a que los GPUs fueron optimizados para realizar multiplicación de matrices de manera paralela y esto es exactamente lo que en definitiva se hace en las redes neuronales. Es decir, las redes neuronales actuales, ganaron la lotería del hardware (o quizás esperaron suficiente) [109].

La contracara de esto es que no debe minimizarse que esta revolución actual existe, porque se le está tirando hardware al problema. Por ejemplo, para entrenar GPT-3, con 175 millones de parámetros, OpenAI generó  $10^{23}$  flops en 1024 GPUs por un mes entero, con un costo de varios millones de dolares.

# 17. Autoencoders

Las primeras implementaciones de los autoencoders surgieron con el nombre de **Autoasociadores**[112] durante la década del 90. Los autoencoders tienen una arquitectura de red neuronal no supervisada, que se puede utilizar para reducir la dimensionalidad de los datos, hacen compresión, y sobre todo son la base de algunos modelos de redes neuronales generativas. Junto con las GANs son las primeras redes que empiezan a jugar con el concepto de la generación: como **generar** muestras nuevas de un conjunto de datos  $X$ .

Surgen a partir de la siguiente premisa: ¿Qué pasa si a la salida final de un MLP es la entrada para otro MLP invertido?

## 17.1 Autoasociadores

### 17.1.1 Arquitectura

La arquitectura base esta conformada por dos redes neuronales artificiales de perceptrones multicapas, donde la salida de la primera red se conecta con la entrada de la segunda. La segunda red tiene la distribución invertida de neuronas en las diferentes capas y como salida tiene la misma dimensión que la entrada de la primera red. Lo que se busca en el autoencoder que dada una entrada particular que se presenta en la primera red, se establezcan la combinación de pesos correcta para que la salida que se obtiene de la segunda red sea exactamente igual a la entrada.

Esto es:

$$\begin{aligned} Z &= f(X) = h(XW + b) \\ X' &= g(Z) = h(ZV + p) \end{aligned}$$

y esto debe satisfacerse de manera que  $L(X, X') = ||X - X'||^2$ .

El entrenamiento entonces está dado por cualquier método de optimización que minimice la diferencia entre la salida  $X'$  y  $X$ , y que converja a los pesos sinápticos que logran mapear eso.

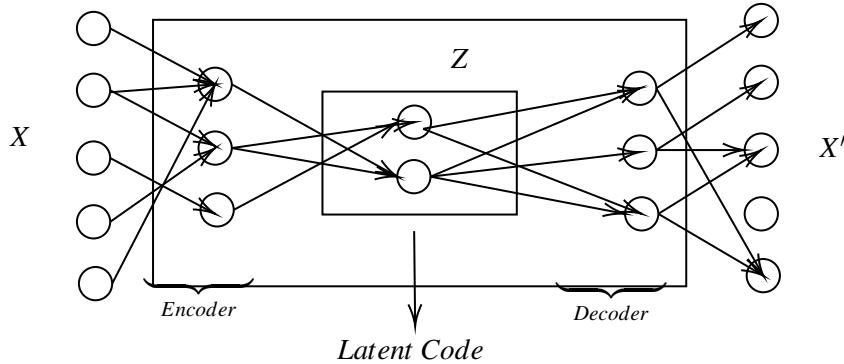


Figura 17.1: Diferentes partes de un Autoencoder.

La figura 17.1 muestra la representación de estas dos redes. La primera parte se denomina **Encoder**, en tanto que la segunda es **Decoder**. El punto del medio, donde se encuentra la salida de la primera red con la entrada de la segunda, se denomina **Latent Code**. Una vez entrenado, un dato  $X$  en la entrada de la primera red, el codificador, se transforma en un código nuevo en el espacio latente (de la dimensión que este sea), para luego ese valor, entrar en la segunda red, el decodificador, y volver a ser el valor que se presentó en la entrada. Esto es, se genera un nuevo código para todos los datos del conjunto  $X$  que queda plasmado en el espacio latente.

### 17.1.2 Autoencoder Lineal

Primero, cabe repasar dos definiciones importantes

**Definition 17.1.1 — Decomposición Espectral.** Sea  $X$  una matriz invertible, con autovalores reales, entonces la matriz puede ser diagonalizada y factorizarse como:

$$X = ELE^T \quad (17.1)$$

donde  $L$  es la matriz diagonal formada por los autovalores y  $E$  es la matriz de filas formada por sus autovectores.

**Definition 17.1.2 — Decomposición en Valores Singulares.** Para el caso de matrices no cuadradas, existe una operación similar pero más general que es la descomposición en valores singulares, SVD. Dada una matriz  $X$  de  $n \times d$ ,

$$SVD(X) = \tilde{Z}\Sigma V^T \quad (17.2)$$

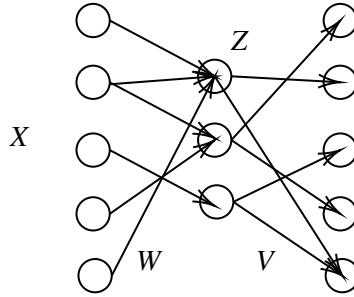
donde  $\tilde{Z}$  son los vectores columnas singulares de izquierda,  $\Sigma$  es una matriz diagonal positiva y  $V$  son los vectores singulares derechos. El cálculo de SVD se realiza minimizando  $\|X - \tilde{Z}\Sigma V^T\|$ .

Podemos tomar la definición de Autoencoder pero aplicar solamente funciones de activación identidad (fig ??). De ahí que la operación de minimización del autoencoder se reduce a la expresada en la ecuación ??.

Así una vez que el autoencoder aprende minimiza

$$J = \|X - ZV^T\| \quad (17.3)$$

$$X \approx ZV^T \quad (17.4)$$



donde,

- $X$ : La matriz de datos de  $n \times d$ . Hay  $n$  datos de dimensión  $d$ .
  - $Z$ : La salida de la capa interna del autoencoder, del código de  $n \times k$ .
  - $V$ : La matriz de pesos sinápticos asociada al Decodificador de  $k \times d$ .
- Esto entonces nos permite llegar al siguiente teorema[112].

**Teorema 17.1.1 — Autoencoder Lineal Converge a PCA.** Las salida del código interno  $Z$  del autoencoder lineal son las salidas de las proyecciones de los datos en los componentes principales.

$$T_{PCA}(X) = X E = Z \quad (17.5)$$

### Demostración

Si  $V^T$ , que tiene los pesos del decoder del Autoencoder, es ortonormal, entonces  $X \approx ZV^T$  del Autoencoder puede verse como la descomposición en valores singulares  $X \approx SVD(X) = \tilde{Z}\Sigma V^T$  con  $\tilde{Z}\Sigma = Z$ , ya que la descomposición de SVD surge de resolver la minimización de la Ecuación 17.3. A su vez, por la descomposición espectral, con los autovectores de la matriz de covarianza  $\frac{X^T X}{n-1}$ , se puede obtener  $\frac{X^T X}{n-1} = E L E^T$  donde  $E$  es la matriz de transformación de PCA con los autovectores.

Entonces, reemplazando  $X$  como la factorización por  $X \approx SVD(X) = \tilde{Z}\Sigma V^T$ :

$$\begin{aligned} \frac{X^T X}{n-1} &= (\tilde{Z}\Sigma V^T)^T (\tilde{Z}\Sigma V^T) \frac{1}{n-1} \\ \frac{X^T X}{n-1} &= (V^T)^T \Sigma^T \tilde{Z}^T \tilde{Z} \Sigma V^T \frac{1}{n-1} \\ \frac{X^T X}{n-1} &= V \Sigma^T \Sigma V^T \frac{1}{n-1} = V \Sigma^2 V^T \frac{1}{n-1} \end{aligned}$$

donde  $X$  es la matriz de covarianza.

$$\frac{X^T X}{n-1} = V \Sigma^2 V^T \frac{1}{n-1} = V \left( \frac{\Sigma^2}{n-1} \right) V^T$$

Pero por la descomposición espectral, esto es también igual a  $E (L) E^T = \frac{X^T X}{n-1} = V \left( \frac{\Sigma^2}{n-1} \right) V^T$  por lo que si  $X$  tiene media cero,

- $\lambda_i = S_i^2/n - 1$ , con  $S_i$  los valores singulares de  $\Sigma$ .
- $V = E$

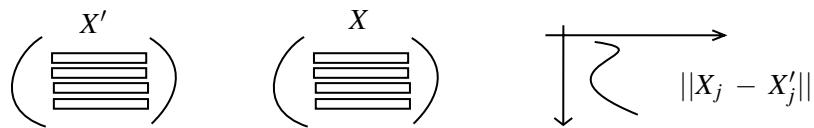
Entonces al transformar los datos con PCA  $T_{PCA}(X) = X E = (\tilde{Z} \underbrace{\Sigma V^T}_I) E = \tilde{Z} \Sigma = Z$  que corresponde a la salida del espacio latente.

Esto no es una manera muy eficiente de obtener las proyecciones en los componentes principales, pero lo que permite es pensar al autoencoder como una extensión no lineal de la descomposición en componentes principales (cuando se usan funciones de activación no lineales). Es una asunción fuerte. El otro punto, es que muestra que la descomposición en valores principales, es aparentemente una forma **canónica** de sintetizar información, donde estructuras que buscan sintetizar la información llegan a la misma solución (ver 14).

### 17.1.3 Detección de Outliers

Los autoencoders generan una nueva codificación para el conjunto de datos, una codificación más eficiente que intente capturar la esencia de la propia estructura de los patrones internos contenidos en ese conjunto. Por eso son eficientes para detectar outliers, elementos particulares en el conjunto de datos que se salen de la norma.

Supongamos que tenemos un autoencoder entrenado con un conjunto  $X$  el cual ya aprendió. Si sometemos como entrada a ese autoencoder nuevos elementos  $x$  que no pertenecían al conjunto inicial, se obtiene una salida  $x'$ , y la diferencia entre ambos  $\|X - X'\|$  debería ser mínima si  $x$  tiene una estructura similar a los datos de entrada, y crecer a medida que ese nuevo valor de  $x$  difiere del resto de los elementos del conjunto. Así, evaluando la diferencia entre la entrada y la salida, es posible tener una métrica que representa cuán *outlier* es cada uno de los nuevos elementos  $x$ .



El procedimiento de detección de outliers consiste en:

- Se entrena al autoencoder con el conjunto de entrenamiento.
- Se mide con alguna medida la diferencia entre cada valor de entrada del autoencoder  $X$  y los valores obtenidos en la salida  $X'$ .

### 17.1.4 Denoising Autoencoder

El anglicanismo **Denoising** hace referencia a la eliminación del ruido. Como el autoencoder genera una aproximación de la matriz de datos  $X$  en base a  $X \approx X' = Z V^T$  entonces esto permite utilizarse para dos tareas.

Ruido se le llama a perturbaciones sobre los datos que contaminan el contenido de información que se quiere recuperar. Hay muchos tipos diferentes de ruidos y son inherentes a cualquier proceso de obtención de información. Es el mal de todos, imposible de eliminar por completo.

Los Denoising AutoEncoder (DAE) pueden utilizarse para eliminar ruido ya que la estructura interna del Encoder/Decoder intenta preservar el contenido de información más relevante en el conjunto de datos.

La idea entonces es una vez aprendido un conjunto de datos, cuando aparecen nuevas muestras que pueden estar contaminadas con ruido, reemplazar la muestra ruidosa por la muestra obtenida en la salida del Decodificador.

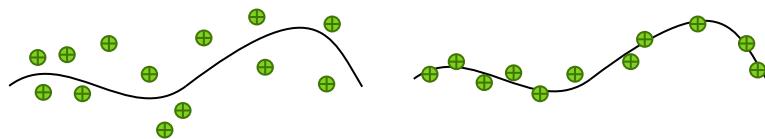
Para esto, en vez de entrenar con los elementos obtenidos directamente del conjunto de datos, se modela el ruido, es decir, se lo characteriza numéricamente (e.g. salt-and-pepper) o en general con una función de distribución de probabilidad (e.g. Gaussiano, Rayleigh) y se agrega a los datos  $X$  generando una nueva instancia  $\tilde{X}$ . Sin embargo, la salida que se pone en la red corresponde al dato  $X$  sin alterar.

### 17.1.5 Contractive Autoencoder

Los CAE agregan un término regularizador a las funciones de costo, que le resta sensibilidad a la entrada, intentando aprender representaciones más simples y crudas de los datos. El término regularizador en los CAE es la norma de Frobenius de la matriz del Jacobiano.

$$\|J_h(X)\|_F^2 = \sum_{ij} \left( \frac{\partial h_j(X)}{\partial X_i} \right)^2 \quad (17.6)$$

De esa manera por ejemplo, datos en dos dimensiones pueden reducirse a una dimensión, como una representación de una función de dos dimensiones, que pasa a una representación paramétrica (con un solo parámetro  $t$ ).



### 17.1.6 SAE - Sparse Autoencoders

Los SAE por otro lado, utilizan una arquitectura donde el espacio latente tiene más dimensiones de la entrada. Para lograr imponer restricciones que permitan adquirir información de la estructura de los datos, imponen un término regularizador que favorece la presencia de esparcicidad, esto es, que un número de neuronas del espacio latente estén en 0 anuladas.

Los SAE se pueden utilizar para realizar **Feature Learning**, es decir para identificar características o transformaciones de los datos que son útiles para discriminarlos.

La Esparcicidad se impone mediante una restricción umbralizada, que de no superarse desactiva todas las neuronas involucradas.

Por ejemplo, los SAE se pueden utilizar para identificar de manera única cuántas codificaciones distintas están siendo capturadas por un autoencoder con códigos binarios que sólo activan una neurona (esto es una técnica utilizada por ejemplo actualmente para entender la codificación que se hace en ChatGPT).

## 17.2 Autoencoder Generativo

Con un conjunto de datos, se pueden establecer dos tipos de modelos, en relación a qué se quiere hacer con ellos.

- **Modelo Discriminativo:** es crudo en los datos. Por ejemplo, una ANN busca la hiper-sábana que separa los datos en dos clases sin importarle cómo se generan (o como se podrían) generar esos datos.
- **Modelo Generativo:** establece cierta relación causal. El modelo hipotetiza cómo se generan los datos en sí. Busca tener una especie de motor de generación de esos datos.

El modelo discriminativo es el que mayoritariamente utilizamos hasta ahora. Por otro lado, el modelo generativo, plantea la pregunta de dado un conjunto de muestras existentes, cómo puedo hacer para generar nuevas muestras que sean indistinguibles de las ya existentes.

La pregunta que surge entonces es si es posible utilizando una red neuronal, implementar un mecanismo generativo causal, que me permita obtener muestras que comparten características representativas de un conjunto de datos. La respuesta, o una respuesta, es el Autoencoder. Y para

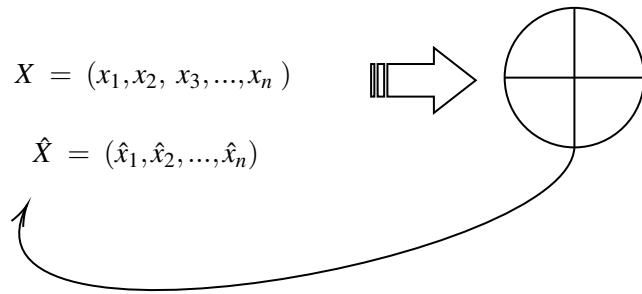


Figura 17.2: El modelo generativo busca entender el proceso de como generar nuevas muestras, indistinguibles de las originales.

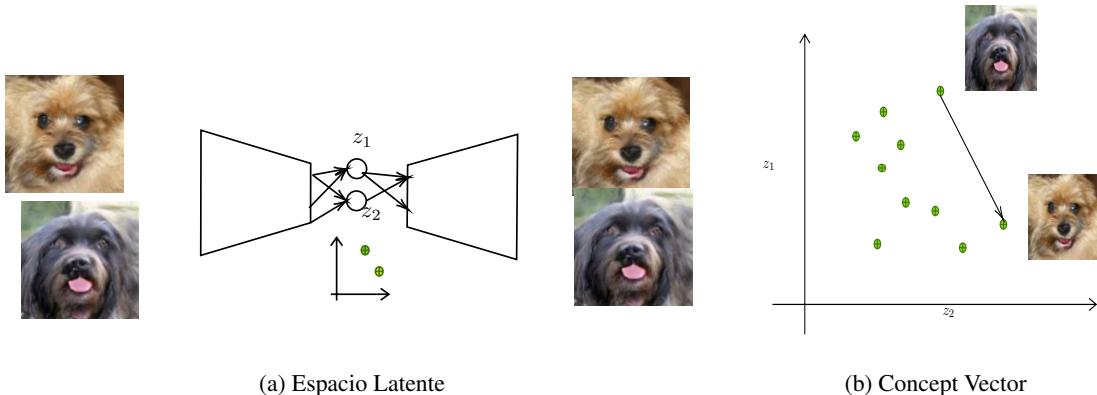


Figura 17.3: Cada punto representa una correspondencia en el espacio latente con un dato concreto del dataset  $X$ . Así viajando de un punto a otro por los valores intermedios, puedo generar nuevos códigos  $Z$  intermedios y obtener para cada uno de ellos representaciones de imágenes  $X'$  que podrían compartir la estructura de las originales. Estaría generando caras de perros.

eso, podemos hacer uso de la nueva codificación que se establece en el espacio latente  $Z$ , y el uso del decodificador, la segunda parte del autoencoder.

Como se ve en la figura 17.3, dos imágenes de perros son codificadas por un autoencoder ya entrenado, generando dos puntos, dos códigos, en el espacio latente  $Z$ .

La idea es justamente que al explotar la codificación que se construye en el propio espacio latente, es posible, obtener representaciones nuevas de los elementos  $X$  mediante la selección de valores  $Z$  que no fueron mapeados de manera directa por ningún  $X$  del conjunto de entrenamiento.

El procedimiento entonces para implementar un autoencoder generativo viene dado por:

- Utilizamos un Autoencoder para codificar en el Espacio Latente todos los patrones.
- Luego dejamos de lado el Encoder, para movernos dentro de un vector de representación en el espacio latente de  $Z$ .
- Generamos muestras sucesivas especificando de manera directa, por ejemplo en dos dimensiones, los valores de  $z_1$  y  $z_2$ .
- Para cada vector de valores de  $z_i$  obtenemos una nueva muestra generada por el Decodificador.

### 17.2.1 La estructura del espacio latente

Si el autoencoder generativo es totalmente determinístico, solo le es posible mapear estrictamente todos los elementos del conjunto de datos  $X$  y en general, de no mediar ningún otro proceso que permite explotar las codificaciones del espacio latente, no tendrá estructura y representaciones intermedias en él, no representaran nada concreto en el conjunto  $X$ . Por esta razón, se requiere que el espacio latente tenga una estructura. La figura 17.4 muestra dos situaciones de espacios latentes. Primero uno disperso donde muchos valores intermedios no tienen ninguna representación. El de la derecha por otro lado, muestra una estructura más cohesiva representando que los valores intermedios tienen una representación de los datos.

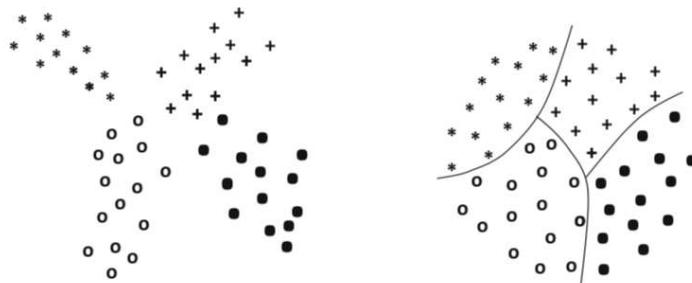


Figura 17.4: A la izquierda un espacio latente sin ningún tipo de estructura donde puntos intermedios no tienen ninguna representación de codificación. A la derecha un espacio latente compacto más cohesivo donde la mayor cantidad de puntos tienen una representación continua.

Necesitamos entonces algo más que nos permita **capturar** lo que pasa en el espacio latente, para poder generar nuevas muestras **válidas** a medida que nos movemos en ese espacio latente.

Una manera de resolver esto, es darle una estructura estadística al espacio latente. Es decir en vez de ver la salida que se obtiene para un único punto de  $Z$ , ver las salidas que se obtienen para un conjunto de puntos que son sampleados alrededor de cada una de las representaciones en el espacio latente de los valores de  $X$ .

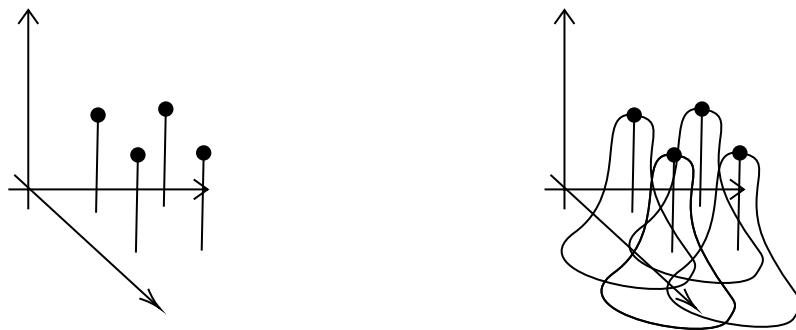


Figura 17.5: En el autoencoder generativo (izq.) cada valor de  $X$  tiene una única representación en el espacio latente. En cambio en el autoencoder variacional, se samplean sucesivamente valores alrededor de cada uno de los puntos combinandolos entre ellos, lo que permite que cualquier punto  $Z$  tenga alguna representación posible de los valores de  $X$ .

### 17.3 VAE - Variational Autoencoder

Los VAE, los autoencoders variacionales [113] son una convergencia de dos ideas, donde una vez más se verifica que la clave de la innovación está en conectar hechos que solo en apariencia parecen ser inconexos:

- Una propuesta de una red neuronal estocástica, un Perceptrón multicapa estocástico dividido en un Encoder y un Decoder, que conforman un Autoencoder.
- Una estimación variacional de un modelo generativo[114, 115, 116, 117].

Desde un punto de vista más práctica, y programático, lo que vamos a hacer con el VAE, es dar la posibilidad que al encontrarnos un  $X$  que usamos como entrada, mapea de manera estocástica a valores de  $Z$  distintos, y de esa manera forzar al Decoder a generar una decodificación en  $X'$  para ese  $Z$  que se parezca al  $X$ , lo más posible. Es decir, vamos a estar mapeando los  $X$ , que tienen una cardinalidad dada, a un espacio completo de  $Z$ , de una cardinalidad mucho mayor.

#### 17.3.1 Redes Neuronales Estocásticas

Una red de una capa oculta es un aproximador universal[118] que puede representar cualquier función computable (16.2.1). Entre otras cosas, puede representar una función estocástica, regida por una p.d.f.

**Definition 17.3.1 — Funciones Estocásticas.** Es una función donde la asignación de  $x$  a  $y$  es estocástica y está regida por una p.d.f. condicional [119].

$$f(x) = L(x) + \varepsilon(x)$$

$y = f(x)$  tal que, dado  $x$ ,  $y = f(x)$  según  $p(y/x)$ .

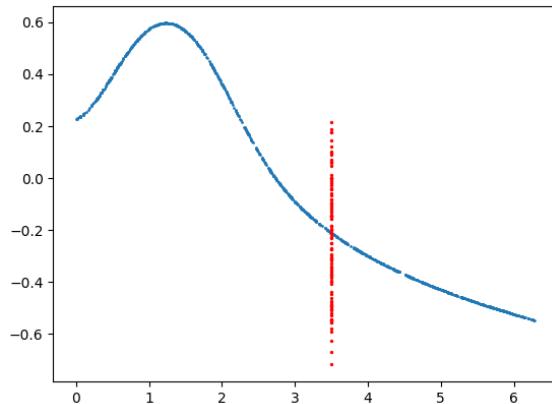


Figura 17.6: Una función seno  $\sin(\cdot)$  estocástica. En azul se muestran el promedio de los valores generados por la función. En 3.5 en rojo se muestran todas las instancias particulares generadas para ese punto.

Las funciones estocásticas hacen mapeos no determinísticos, como se puede ver en la figura 17.6. Una red neuronal que permite representar una función estocástica se denomina una **Red Neuronal Estocástica** [118]. Para implementarlas se agrega un nodo, una capa donde se genera un número aleatorio pero de una forma que la red pueda seguir retropropagando el error, y para eso se usa el truco de la reparametrización.

### El Truco de la Reparametrización

La clave está en el **reparametrization trick** haciendo que el valor de  $z$  se calcula en función de  $\bar{\mu}$  y de  $\bar{\Sigma}$ .

$$z = h(\bar{X}) = \varepsilon \odot \bar{\Sigma}(\bar{X}) + \bar{\mu}(\bar{X}) \quad (17.7)$$

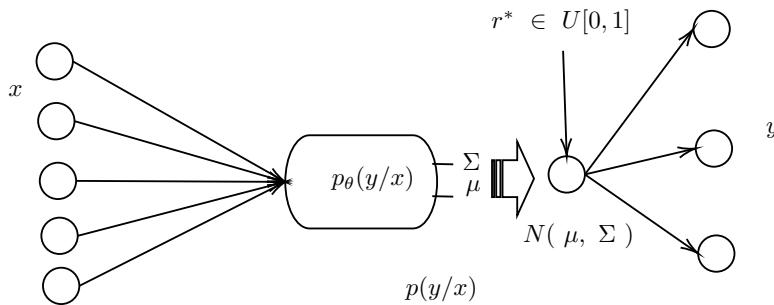


Figura 17.7: Una función  $\sin(\cdot)$  estocástica

El nodo estocástico recibe entonces los parámetros de una distribución de probabilidad, como por ejemplo, la media y el desvío estandard para una normal, para luego con esos parámetros, samplear un valor  $r$  que es utilizado con el truco de la reparametrización, para muestrear con esa distribución que tiene esos parámetros. El efecto final, es que la red se convierte en un mapeo estocástico, entre  $x$  e  $y$  (fig 17.7).

### 17.3.2 Inferencia Variacional

La segunda idea que confluye en el autoencoder variacional es la inferencia variacional. La teoría variacional sirve para resolver problemas de sistemas complejos en física, intentando determinar una función de distribución de probabilidad que normalmente no tiene una definición cerrada o no es computable hacerlo. Suele resolverse mediante sampleo (Monte-Carlo), o explorando datos del fenómeno.

La conexión es que inferir una p.d.f. tiene cierta semejanza al proceso de aprendizaje automático (Peterson, Anderson 1987 con Mean Field Theory [120], Michael Jordan<sup>1</sup> en 1999 [121] y Hinton, Van Camp en 1993 [122]). Es decir hay una conexión entre aprender una distribución de probabilidad, paramétricamente, que tiene justamente parámetros libres, y la idea de ajustar esos parámetros en base a datos para identificar patrones.

Así en el 2014 Kingma et al [113] plantea el autoencoder variacional, reviviendo los autoasociadores, y planteando por primera vez (hasta donde sabemos) la idea **generativa**. La herramienta central esta dada en la definición 17.3.2.

**Definition 17.3.2 — Inferencia Variacional.** La inferencia variacional intenta resolver el problema de estimar una función de densidad de probabilidad (p.d.f.) aproximandola con funciones conocidas, y optimizando los parámetros de esa función mediante la solución de un problema de optimización.

Justamente detrás de la inferencia variacional, hay un proceso de optimización y eso se puede resolver mediante la convergencia de la red neuronal.

<sup>1</sup>No el basquetbolista, el otro.

### 17.3.3 Optimizar el Autoencoder Variacional

Podemos ahora upgradear el Autoencoder determinístico, utilizando la red estocástica, donde el Encoder de la red lo vamos a usar para representar una función estocástica,  $p(z/x)$  para el cuál dado un  $x$  de la entrada, la función nos va a generar **algún**  $z$  en el espacio latente siguiendo esa p.d.f.. Por otro lado, el del Decoder, vamos a tomar **algún**  $z$  del espacio latente, y lo vamos a mandar probabilísticamente devuelta al mundo de  $x$  con la función (conjugada a priori)  $p(x/z)$ .

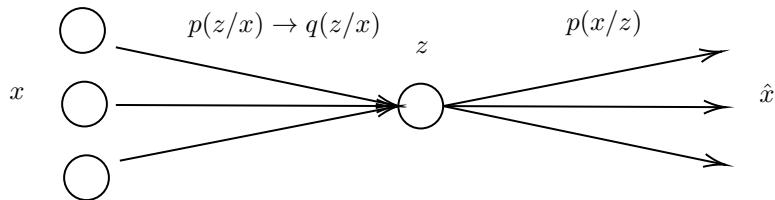


Figura 17.8: El autoencoder variacional y las p.d.f. que mapea.

¿Cuál es la forma de  $p(x/z)$  o  $p(z/x)$ ? No sabemos, ni podemos saber nunca, porque dependen de la estructura intrínseca de los datos  $x$ . Pero ahora, con la inferencia variacional en el bolsillo, podemos usar el autoencoder para que optimizando la aproximemos por funciones de distribución que sí conocemos, que planteamos nosotros, paramétricas.

Los ingredientes que tenemos, van a ser:

- $p(z/x)$ : es la p.d.f. de los datos que desconocemos, y que la vamos a aproximar por  $q(z/x)$
- $q(z/x)$ : es la p.d.f. que vamos a **plantear** (a gusto) para aproximar  $p(z/x)$ .
- $q(z)$ : vamos a asumir que los datos tienen una distribución normal alrededor de cada  $z$ .
- $p(z)$ : vamos a asumir que efectivamente probabilísticamente cada  $x$  mapea multidimensionalmente normal a  $z$ . Que sea **normal** es una restricción sobre la  $p(\cdot)$ . Podría ser otra.

Es necesario sí determinar cuál será la función de optimización. Para eso, podemos arrancar con el objetivo del problema variacional, estimar  $p(z/x)$  con  $q(z)$  usando la divergencia KL:

$$\min KL(s||t) = - \sum t(x) \log \frac{s(x)}{t(x)} \quad (17.8)$$

$$\begin{aligned}
 KL(q(z)||p(z/x)) &= - \sum q(z) \log \frac{p(z/x)}{q(z)} \\
 &= - \sum q(z) \log \frac{\frac{p(z,x)}{p(x)}}{\frac{q(z)}{1}} \\
 &= - \sum q(z) \log \frac{p(z,x)}{q(z)} \frac{1}{p(x)} \\
 &= - \sum q(z) \left\{ \log \frac{p(z,x)}{q(z)} - \log p(x) \right\}
 \end{aligned}$$

$$\begin{aligned}
KL(q(z)||p(z/x)) &= - \sum_z q(z) \left\{ \log \frac{p(z,x)}{q(z)} - \log p(x) \right\} \\
&= - \sum_z q(z) \log \frac{p(x,z)}{q(z)} + \sum_z q(z) \log p(x) \\
&= - \sum_z q(z) \log \frac{p(x,z)}{q(z)} + \underbrace{\log p(x) \sum_z q(z)}_1
\end{aligned}$$

Con esto llegamos a que

$$KL(q(z)||p(z/x)) = - \sum_z q(z) \log \frac{p(x,z)}{q(z)} + \log p(x) \quad (17.9)$$

Pero estos términos podemos rearreglarlos pasando la sumatoria al primer miembro y espejando la igualdad.

$$\log p(x) = KL(q(z)||p(z/x)) + \sum_z q(z) \log \frac{p(x,z)}{q(z)} \quad (17.10)$$

Desde esta ecuación, el valor  $\log p(x)$  es fijo porque depende solamente de los datos, por lo que no puede ajustarse. Como la divergencia queremos que sea lo más chica posible, esto es equivalente a maximizar el segundo término.

$$\underbrace{\log p(x)}_{\text{fijo para } x} = \underbrace{KL(q(z)||p(z/x))}_{\text{lo más pequeño posible}} + \underbrace{\sum_z q(z) \log \frac{p(x,z)}{q(z)}}_{\mathcal{L}, \text{ lo más grande posible}} \quad (17.11)$$

$$\mathcal{L} \leq \log p(x) \quad (17.12)$$

Ese término  $\mathcal{L}$  se llama *Variational Lower Bound*, y tiene que ser lo más grande posible y cuando  $KL \rightarrow 0$  entonces es exactamente el  $\log p(x)$ . Por lo tanto, minimizar originalmente 17.9 es equivalente a maximizar  $\mathcal{L}$ .

$$\begin{aligned}
\mathcal{L} &= \sum_z q(z) \log \frac{p(x,z)}{q(z)} \\
&= \sum_z q(z) \log \frac{p(x/z)p(z)}{q(z)} \\
&= \sum_z q(z) \left\{ \log p(x/z) + \log \frac{p(z)}{q(z)} \right\} \\
&= \sum_z q(z) \log p(x/z) + \sum_z q(z) \log \frac{p(z)}{q(z)} \\
\\
\mathcal{L} &= \sum_z q(z) \log p(x/z) + \sum_z q(z) \log \frac{p(z)}{q(z)} \\
\mathcal{L} &= \mathbb{E}_{q(z)} \log p(x/z) - KL(q(z)||p(z))
\end{aligned}$$

Esta es la expresión final a la que arribamos como función de optimización que nos va a permitir resolver el problema de optimización, y que justo queda en dos términos, la base más el término regularizador.

**Ingredientes:**  $p_\theta(z)$ ,  $q_\phi(z)$

$$\theta^*, \phi^* = \arg \max \mathcal{L}(\theta, \phi, x)$$

Podemos asumir una distribución para  $p_\theta(z)$  simple, y elegir otra distribución para  $q_\phi(z)$ , que determinará la estructura del espacio latente. Con esto aproximamos  $p_\theta(z/x)$  con  $q_\theta(z/x)$ . Al optimizar  $\mathcal{L}$  se encontraron los valores para los parámetros de esas distribuciones,  $\phi^*, \theta^*$ .

Asumimos que:

- $p_\theta(z) = \mathcal{N}(0, \mathcal{I})$
- $q_\phi(z) = \mathcal{N}(\mu(x), \Sigma(x))$ , con  $\Sigma(x)$  diagonal.

La primera parte que es el Error de Reconstrucción equivale a:

$$p_\theta(x/z) = \frac{1}{\sqrt{(2\pi)^k} \sqrt{\Sigma(z)}} \exp\{(x - \mu(z))^T \Sigma(z)^{-1} (x - \mu(z))\}$$

El  $\log$  de esto es proporcional a la diferencia entre el  $X$  de entrada y el  $\hat{X}$  de salida.

$$\log p_\theta(x/z) \propto \underbrace{(x - \mu(z))^T}_{\hat{x}} \Sigma(z)^{-1} \underbrace{(x - \mu(z))}_{\hat{x}}$$

Y en relación al término regularizador con  $p_\theta(z) = \mathcal{N}(0, \mathcal{I})$ ,

$$KL(q_\phi(z)) = \mathcal{N}(\mu(x), \Sigma(x)) || \mathcal{N}(0, \mathcal{I}),$$

este luego de varios pasos<sup>2</sup>, lleva a la siguiente expresión:

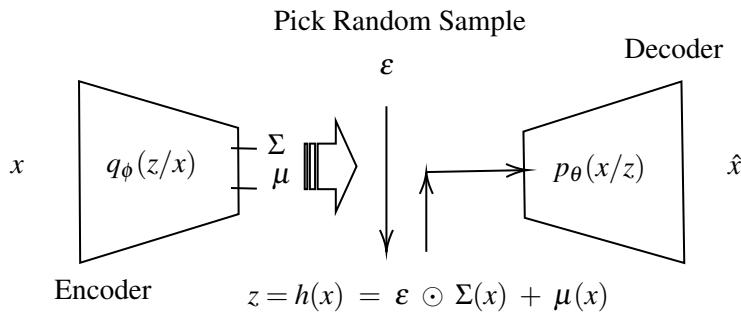
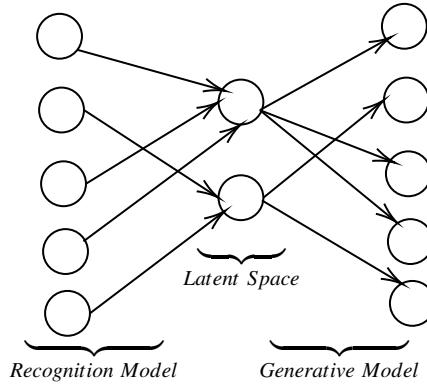
$$\begin{aligned} KL &= \frac{1}{2} \{ \text{trace}(\Sigma(x)) + \mu^T \mu(x) - k - \log(|\Sigma(x)|) \} \\ &= \frac{1}{2} \left\{ \sum_k \Sigma(x) + \sum_k (\mu(x))^2 - \sum_k 1 - \log(\Pi_k \Sigma(x)) \right\} \\ &= \frac{1}{2} \left\{ \sum_k \Sigma(x) + \sum_k (\mu(x))^2 - \sum_k 1 - \sum_k (\log \Sigma(x)) \right\} \\ &= \frac{1}{2} \sum_k (\Sigma(x) + (\mu(x))^2 - 1 - \log \Sigma(x)). \end{aligned}$$

donde por razones numéricas, se reemplaza  $\Sigma(x) = \exp(\Sigma(x))$ , quedando

$$KL = \frac{1}{2} \sum_k (\exp \Sigma(x) + (\mu(x))^2 - 1 - \Sigma(x)).$$

---

<sup>2</sup>En el apéndice 17.4 de este capítulo está el paso a paso.



Así la formula final de la función a optimizar es

$$J = L + \lambda R \quad (17.13)$$

$$\begin{aligned} \max \mathcal{L} &= \mathbb{E}_{q(z)} \log p(x/z) - KL(q(z)||p(z)) \\ \min \mathcal{L} &= -\mathbb{E}_{q(z)} \log p(x/z) + KL(q(z)||p(z)) \\ \min \mathcal{L} &= ||\bar{X} - \bar{X}'|| + \frac{1}{2} \sum_k (\exp \Sigma(x) + (\mu(x))^2 - 1 - \Sigma(x)) \\ \min \mathcal{L} &= ||\bar{X} - \bar{X}'|| - \frac{1}{2} \sum_k (1 + \Sigma(x) - (\mu(x))^2 - \exp \Sigma(x)) \end{aligned}$$

Resumiendo, el algoritmo del autoencoder variacional es:

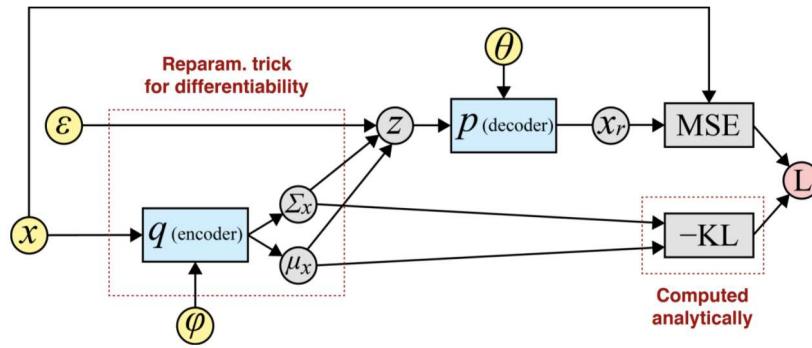
Tomamos un  $x$  y lo ponemos en la entrada de la red. Con ese  $x$  vemos que valores de  $\Sigma$  y  $\mu$  obtenemos en la salida del Encoder. Luego sampleamos un  $z$  de  $p(z) = \mathcal{N}(0, \mathcal{I})$  (esto es equivalente a obtener un  $\epsilon$  multiplicarlo por la varianza p.a.p. y sumarle la media).

Con ese  $z$  lo pasamos por el Decoder y obtenemos un  $\hat{x}$ . Entrenamos al Autoencoder ahora para que minimice  $\mathcal{L}$  actualizando primeros los pesos del Decoder, luego pasando por la función  $h(x)$  y de ahí retropropagando el error obtenido en  $\mu$  y  $\Sigma$  hacia los pesos del Encoder.

El *Parametrization Trick* de la capa estocástica de la red, permite poder retropropagar el error manejando esa capa que tiene una entrada extra que es una variable aleatoria [123]:

Fijense que el término regularizador, NO depende de la salida del decoder ( $\bar{X}$ ). Depende de variables internas ( $\Sigma$  y  $\mu$ ) que llegan a la capa latente.

**¿Cómo propagar[124] el error por el nodo estocástico?**



Supongamos que queremos actualizar un peso particular del encoder  $\omega_e$  en base a la función de costo:

$$J = L + \lambda R$$

Lo que queremos calcular es  $\frac{\partial J}{\partial \omega_e}$  para poder usarlo en la

$$\omega_e^{t+1} = \omega_e^t - \eta \frac{\partial J}{\partial \omega_e}$$

$$\begin{aligned} \frac{\partial J}{\partial \omega_e} &= + \left( \frac{\partial L}{\partial z} \frac{\partial z}{\partial \mu} \frac{\partial \mu}{\partial \omega_e} + \lambda \frac{\partial R}{\partial \mu} \frac{\partial \mu}{\partial \omega_e} \right) \dots \\ &= + \left( \frac{\partial L}{\partial z} \frac{\partial z}{\partial \sigma} \frac{\partial \sigma}{\partial \omega_e} + \lambda \frac{\partial R}{\partial \sigma} \frac{\partial \sigma}{\partial \omega_e} \right) \end{aligned}$$

La  $\frac{\partial L}{\partial z}$  corresponde a la retropropagación del error en el decoder, en tanto que  $\frac{\partial z}{\partial \mu}$  es 1, y  $\frac{\partial \mu}{\partial \omega_e}$  es la retropropagación hacia el encoder.  $\frac{\partial R}{\partial \mu}$  corresponde a la derivada de Eq. 17.13.

Por otro lado el segundo término corresponde a los cambios x el lado del  $\mu$ , donde la  $\frac{\partial z}{\partial \sigma}$  es  $\epsilon$ , la  $\frac{\partial R}{\partial \sigma}$  corresponde a la derivada de Eq. 17.13 (ahora con respecto a  $\sigma$ ), y la  $\frac{\partial \sigma}{\partial \omega_e}$  es la retropropagación hacia el encoder.

Los autoencoder variacionales son hoy centrales en muchos algoritmos y soluciones que buscan generar nuevos datos, generar nuevas muestras y son utilizados como herramienta de **data augmentation**[125]. Se puede encontrar más información sobre Inferencia Variacional [126], Metodos Variacionales [113], VAE [127] y Autoencoders [128, 129, 130].

## 17.4 Apéndice

### Probabilidad y Estadística

$$N(x/\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (17.14)$$

$$N(\mathbf{x}/\vec{\mu}, \Sigma) = \frac{1}{\sqrt{|2\pi\Sigma|}} \exp\left(-\frac{1}{2} (\mathbf{x} - \vec{\mu})^T \Sigma^{-1} (\mathbf{x} - \vec{\mu})\right) \quad (17.15)$$

### Teoría de la Información

Repasso cortito y al pie de las definiciones más importantes de teoría de información.

**Variable aleatoria:**  $X = x_i$

**Información:**  $I = -\log p(x_i)$

**Entropía:** Se puede interpretar como una medida en la cantidad de información que tienen todos los estados posibles que puede tomar  $x$ . Es un promedio ponderado.

$$H = \mathbb{E}_{p(x_i)} I(x_i) = -\sum p(x_i) \log p(x_i)$$

**Información Mutua:**  $I(x,y) = H(y) - H(y/x) = \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$

$H(x) = I(x,x)$ , y se tiene que,

- $I = 0$  implica que  $p(x) = 1$  Certeza !
- $I > 0$  algo de información
- $I(x_k) > I(x_i) \Rightarrow P_k < P_i$ , lo más raro (menos probable) aporta más información.

La entropía es mínima, cero, cuando es una situación de total certeza (teniendo en cuenta que  $\lim_{p \rightarrow 0^+} p \log p = 0$ ), y es máxima, cuando todos los eventos son equiprobables (probabilidad uniforme). Los valores que puede tomar son entonces  $0 <= H(x) <= \log(2k + 1)$ .

**Entropía condicional:**  $H(x/y) = H(x,y) - H(y)$

**Entropía conjunta:**  $H(x,y) = \sum_{i=1}^m \sum_{j=1}^m p(x_i, y_j) \log \frac{1}{p(x_i, y_j)}$

#### Definition 17.4.1 — Cross Entropy.

$$H_p(q) = -\sum_{k=1} q(y_k) \log p(y_k) \quad (17.16)$$

La definición 17.4.1 se usa cuando hay dos distribuciones en juego y donde la que se usa para determinar la información individual aportada por cada evento es diferente de la que se usa para la ponderación. Esto plantea una comparación desde la perspectiva de la codificación de los datos. Esta definición puede simplificarse para el caso donde hay dos eventos posibles.

#### Definition 17.4.2 — Binary Cross Entropy.

$$H_p(q) = -(y_i) \log p(y_i) + (1 - y_i) \log(1 - p(y_i)) \quad (17.17)$$

En el contexto de aprendizaje automático, el promedio de la entropía binaria cruzada se puede utilizar como una función de costo para un problema de clasificación supervisado binario. Ahí justamente aparecen las dos distribuciones, la real que proviene de los datos y es la etiqueta que representa la clase verdadera, y la que produce el clasificador.

#### Definition 17.4.3 — Average Binary Cross Entropy.

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N \{(y_i) \log p(y_i) + (1 - y_i) \log(1 - p(y_i))\} \quad (17.18)$$

Aquí,  $q(y_i) = y_i = 0|1$ , y representa el label,  $p(y_i)$ ,  $1..N$  es la probabilidad asignada a ese label para la muestra  $k$ . *Binary Cross Entropy*<sup>3</sup> sirve como una función para estimar cuál es la efectividad de un clasificador que asigna el label  $y_k$  a cada elemento  $1..N$  con probabilidad  $p(y_k)$ . Un clasificador perfecto que estima  $y_k$  con probabilidad 1, genera un valor de binary cross entropy de cero.

<sup>3</sup>Cross Entropy Loss también puede llamarse *Logistic Loss* o *Multinomial Logistic Loss*.

**Definition 17.4.4** KL: Kullback-Leibler Divergencia ó Entropía relativa Es una norma (en rigor no) que permite ver la distancia entre distribuciones de probabilidad, i.e. si tiende a cero implica que las distribuciones son parecidas. Una medida de similaridad.

$$KL(q||p) = - \sum_{x_i} q(x) \log \frac{q(x)}{p(x)} \quad (17.19)$$

donde  $p(x)$  es la referencia a la que se compara  $q(x)$ .

Propiedades:

- $KL(q||p) \neq KL(p||q)$
- $KL(q||q) = 0$
- $KL(q||p) \geq 0$
- $KL(q||p) = H_p(q) - H(q) \geq 0$

### Estimador por Máxima Verosimilitud

MLE, Maximum Likelihood Estimator. Dadas muestras  $(x_1, x_2, \dots, x_n)$  y una p.d.f.  $p(x)$ . La pregunta que se intenta abordar es, con las muestras  $x_i$ , ¿Para qué valor de  $\phi$  es más probable que la muestra observada  $x$  haya ocurrido?

Entonces  $L(x_i; \phi) = p(X = x_1)p(X = x_2)\dots p(X = x_n)$ . Se busca entonces maximizar  $L(\phi)$  mediante:

$$\frac{dL}{d\phi} = 0 \text{ o } \frac{d \ln L}{d\phi} \quad (17.20)$$

#### Teorema 17.4.1 — Máximo de función monotónica.

$$\hat{\phi} \text{ es MLE de } \phi \Rightarrow f(\hat{\phi}) \text{ es MLE de } f(\phi) \quad (17.21)$$

### Regularización

La Regularización reduce los errores de generalización. Es una manera de agregar restricciones a un funcional de optimización para forzar al aprendizaje de algo útil. Se traduce como un término de penalización que se busca que se mantenga pequeño al buscar minimizar una función de costo. Ayuda a no considerar las soluciones complejas o extremas en el propio problema de optimización.

Este método, proviene de la teoría de optimización y se lo conoce como *Tikhonov Regularization*. En ML, puede ser  $L^2$ -regularized o  $L^1$ -regularized. Por ejemplo, en cuadrados mínimos,

$$\min_{\phi} \frac{1}{N} \|y - x\phi\| + \lambda \|\phi^2\| \quad (17.22)$$

El método de MLE se transforma en MAP cuando se considera un adicional de regularización. Early Stopping Dropout o Weight decay ( $w_{t+1} = w_t - \alpha \Delta_w - \lambda w_t$ ) son métodos que intrínsecamente involucran una regularización.

### Derivación del Métodos de Cuadrados Mínimos en base a modelos gráficos

Y depende de X y de W.

El objetivo es minimizar  $L(w) = \sum_{i=1}^N (W^T x_i - y_i)^2$  para encontrar  $\hat{w} = \arg \min_w L(w)$

$$P(y, w, x) = P(y|w, x)P(w)P(x) \quad (17.23)$$

Para eso, la idea sería encontrar los valores de  $w$  que maximizan la conjunta de tener los  $y$  que tenemos pero dados los  $x$  que observamos, por lo que a la condicional tenemos que condicionarlo a esos  $x$  que podemos observar (los valores  $x$  para los cuales queremos encontrar la resolución por cuadrados mínimos).

$$\begin{aligned} P(y, w, x) &= P(y/w, x)P(w)P(x) \\ P(y, w, x/x) &= P(y/w, x, x)P(w/x)P(x/x) \\ P(y, w/x) &= P(y/w, x)P(w)1 \end{aligned} \tag{17.24}$$

Teniendo en cuenta que  $P(x/x)$  es 1. Para especificar este modelo, podemos asumir dos cosas, primero que los valores de  $y$  dados  $x$  y  $w$  están todos alrededor de una normal con media, más o menos,  $w^T x$ , y desvío  $\sigma^2 I$  (es decir solo la diagonal de la matriz de covarianza).

Además podemos asumir que los valores de  $w$  tienen media cero, y también un desvío estandar diagonal  $\omega^2 I$ .

Esto es,

$$\begin{aligned} P(y/w, x) &= N(y/w^T x, \sigma^2 I) \\ P(w) &= N(w/0, \gamma^2 I) \end{aligned} \tag{17.25}$$

Asumiendo esto, entonces,  $P(y, w/x)$  es

$$\begin{aligned} P(y, w/x) &= P(y/w, x)P(w) \\ &= \log(P(y/w, x)P(w)) \\ &= \log(P(y/w, x)) + \log(P(w)) \\ &= \log(\exp(-\frac{1}{2}(y - w^T x)^T (\sigma^2)^{-1}(y - w^T x))) + \log(\exp(-\frac{1}{2}w^T (\gamma^2)^{-1}w^T)) \\ &= -\frac{1}{2\sigma^2}(y - w^T x)^T (y - w^T x) - \frac{1}{2\gamma^2}w^T w \\ &= (-1)[\alpha||y - w^T x||^2 + \delta||w||^2] \end{aligned} \tag{17.26}$$

Maximizar eso es lo mismo que minimizar directamente  $||y - w^T x||^2 + \delta||w||^2$ . El primer término es entonces el resultado esperado que corresponde a la solución del método de cuadrados mínimos para encontrar una aproximación lineal de la relación entre  $x$  e  $y$ . Pero en este caso, adicionalmente, nos está dando un término regularizador, lo cual determina que de todas las soluciones posibles, aquella donde  $w$  es lo más pequeña mejor, parecería ser la que genera la solución más probable (dadas las distribuciones que asumimos).

### Inverse Sampling Theorem

Cuando solo se tiene un generador de números pseudoaleatorios con una distribución uniforme  $U[0, 1]$ , típico **rand**, se puede utilizar el teorema 17.4.2 para generar valores con cuál distribución arbitraria.

**Teorema 17.4.2** Eligen un valor de  $Y U[0, 1]$ , se fijan que les da en  $X = F^{-1}(Y)$ . La distribución de esos  $X$  va a corresponder con la *pdf* en cuestión.

$$\begin{aligned} Y &\sim U[0, 1], F \text{ invertible y } F = CDF(pdf()), \\ X &= F^{-1}(Y) \text{ tiene } CDF(X) = F \end{aligned}$$

$$Y \sim U[0, 1], F = CDF(pdf()), X = \inf_x \{x : F(x) \geq Y\}$$

### Tied Weights

a) Para un valor fijo de  $V$ , la matriz  $W$  óptima satisface  $D^T D(W^T V^T V - V) = 0$ .

$$\begin{aligned} DW^T V^T - D &= 0 \\ D(W^T V^T - I) &= 0 \\ D^T D(W^T V^T - I) &= D^T 0 \\ D^T D(W^T V^T - I)V &= D^T 0V \\ D^T D(W^T V^T V - V) &= 0 \end{aligned} \tag{17.27}$$

b) Si  $D$  tiene rango  $d$ , entonces,  $D^T D$  es invertible y no puede ser cero. Por eso en 17.27, no queda otra que  $W^T V^T V - V = 0$  lo cual implica que  $W^T V^T V = V$ .

c) Demostrar que si  $W^T V^T V = V$  entonces,  $W = (V^T V)^{-1} V^T$  siempre y cuando  $V^T V$  sea invertible.

$$\begin{aligned} W^T V^T V &= V \\ W^T (V^T V) &= V \\ W^T (V^T V)(V^T V)^{-1} &= V(V^T V)^{-1} \text{ invertibilidad de } V^T V \\ W^T &= V(V^T V)^{-1} \\ (W^T)^T &= (V(V^T V)^{-1})^T \\ W &= (V(V^T V)^{-1})^T \\ W &= ((V^T V)^{-1})^T V^T \\ W &= ((V^T V)^T)^{-1} V^T \text{ La T de la inversa, es la inversa de la T} \\ W &= (V^T (V^T)^T)^{-1} V^T \\ W &= (V^T V)^{-1} V^T \textbf{Listo} \end{aligned} \tag{17.28}$$

d) Ahora que pasa si los pesos del autoencoder son los mismos (a, b y c). Esto se da cuenta  $W = V^T$  donde las columnas de  $V$  son ortonormales.

a)

$$||DW^T V^T - D|| ||D(W^T W - D)|| \tag{17.29}$$

de ahí surge que

$$\begin{aligned} D^T D(W^T W - D) &= 0 \\ W^T W &= I \end{aligned} \tag{17.30}$$

b)

$$\begin{aligned}
 W^T V^T V &= V \\
 W^T W V &= V \\
 V &= V
 \end{aligned} \tag{17.31}$$

y c()

$$\begin{aligned}
 W^T V^T V &= V \\
 W^T (W) V &= V \\
 W^T W V &= V \\
 V &= V
 \end{aligned} \tag{17.32}$$

Si los  $w_i$  son ortonormales,

$$w_i^T w_j = \begin{cases} 0 & j \leq i \\ 1 & j \neq i \end{cases} \tag{17.33}$$

### Desarrollo del Término Regularizador del VAE

*Demostración.* Vamos a pasar de

$$KL(q_\phi(z)) = \mathcal{N}(\mu(x), \Sigma(x)) || \mathcal{N}(0, \mathcal{I}),$$

a

$$KL = \frac{1}{2} \{ \text{trace}(\Sigma(x)) + \mu^T \mu(x) - k - \log(|\Sigma(x)|) \}$$

Sean  $P(x) \sim \mathcal{N}(\mu_1, \Sigma_1)$  y  $Q(x) \sim \mathcal{N}(\mu_2, \Sigma_2)$  con  $\mu_1, \mu_2 \in \mathbb{R}^k$  y  $\Sigma_1, \Sigma_2 \in \mathbb{R}^{k \times k}$ .

■





## 18. Convolutional Neural Networks

Las redes convolucionales tienen su origen en el trabajo de Hubel y Wiesel de 1962 [63] que les valió el premio Nobel de Medicina de 1981. Ellos detectaron que en la corteza visual de los gatos, había algunas capas donde la cantidad de veces que disparaban las neuronas de esa capa se correspondía con la inclinación de una barra que le mostraban al gato.

Detectaron un **neural coding**, una capa que era sensible a la inclinación de la barra en el campo visual del gato. Este fue uno de los primeros trabajos donde se pudo obtener información de este estilo, disparos neuronales que se correspondían con información perceptiva de una manera simple.

Este trabajo además, sirvió como tremenda inspiración para el área de Computer Vision. Si el cerebro usa un mecanismo que permite detectar inclinación en los objetos en el campo visual, quizás esa sea una buena manera de procesar la información visual y derivar de ella contenido. Esto es lo que captura Fukushima con su propuesta del Neocognitron, muy inspirado en este trabajo y que LeCun, en paralelo, formaliza muy bien desde el lado de procesamiento de señales en su propuesto de las redes convolucionales.

Las CNN, Convolutional Neural Networks, o redes neuronales convolucionales, o **ConvNet** desde el punto de vista más formal, están basadas en la operación de convolución. Han sido tremadamente exitosas y son uno de los motores detrás del auge de Deep Learning actual. Es más, han sido tan exitosas que nos estamos olvidando de que están, como ocurre con todas las muy buenas tecnologías.



Photo from the Nobel Foundation archive.  
David H. Hubel



Photo from the Nobel Foundation archive.  
Torsten N. Wiesel



Figura 18.1: Hubel y Wiesel, premios nobel de Medicina de 1981

## 18.1 La operación de Convolución

Para poder entender redes convolucionales es importante primero definir y entender la operación de convolución, que surge como la definición 18.1.1.

**Definition 18.1.1 — Convolución.** La operación de convolución es una operación lineal que se da sobre una señal, es decir una función unidimensional en el tiempo.

- Suponiendo que  $x(t)$  es una señal unidimensional en función del tiempo y  $w(t)$  es un **núcleo** de convolución,
- $s(t) = \int x(k)w(t-k)dk$
- $s(t) = (x * w)(t) = \langle x(k), w^*(t-k) \rangle$ , donde  $*$  es el conjugado.
- (Correlación  $s(t) = \int x(k)w(t+k)dk$  (+ positivo))
- Discreta:  $s[n] = \sum_k x[k]w[n-k]$

La convolución matemática cumple un rol primordial en el procesamiento de señales digitales. Por ejemplo, cualquier filtro digital lineal puede representarse por una igualdad basada en convoluciones discretas.

**Definition 18.1.2 CCDE - Constant Coefficient Difference Equation**

$$\sum_{k=0}^{N-1} w_1[k]y[n-k] = \sum_{k=0}^{M-1} w_2[k]x[n-k]$$

- $Y(z) = H(z)X(z)$
- $w_1, w_2$  son kernels de convolución,  $x, y$  son señales discretas.
- $X, Y$  son los estados del sistema y  $H$  es la función de transferencia.

### 18.1.1 Moving Average

Una manera interesante de entender la convolución es verla como una extensión del promedio ponderado en una señal. Imaginemos que hay una señal, una función donde el eje  $X$  es el tiempo  $t$ . Discretamente, para cada punto de  $t$  hay un valor en la señal, como podría ser el precio de bitcoin. Si queremos ver la tendencia de la información, sin perdernos en las variaciones diarias, lo que podemos hacer es para cada día, reemplazar el dato concreto del precio, por el promedio de los 5 días anteriores, hoy, y los 4 días posteriores. Eso va a suavizar la señal de precios, y nos va a permitir ver la tendencia sin importar tanto las variaciones diarias.

El siguiente código por ejemplo, es una implementación de esta operación, que no es ni más ni menos que una convolución de una señal con un núcleo de convolución que está compuesto por valores correspondientes a una distribución uniforme para generar el efecto del promedio.

```
windowlength = 10
avgeeg = np.convolve(signal,
                      np.ones((windowlength,))/windowlength,
                      mode='same')
```

El kernel de convolución puede tener cualquier tipo de longitud, y justamente representa cuántos valores alrededor del valor central, van a utilizarse para hacer el promedio (en este caso). Por supuesto, pueden usarse otros valores para el kernel que generan un efecto totalmente diferente.

Esto por ejemplo, se puede utilizar para procesar una señal de otro tipo, como información de sensores, como se ve en la figura 18.2, donde la aplicación del moving average, genera un suavizado en la señal que permite visualizar mejor los eventos más significativo y ocultar el ruido.

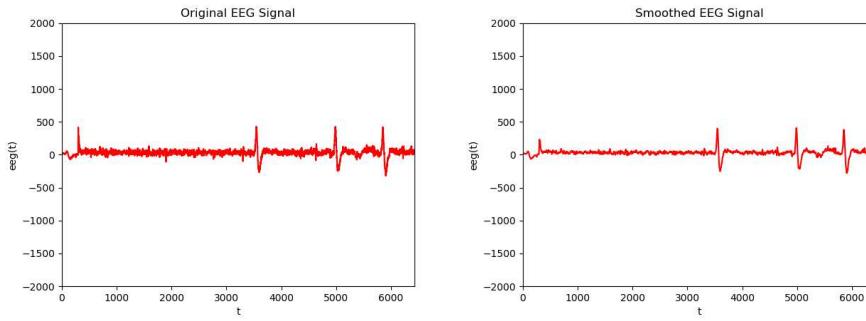


Figura 18.2: A la izquierda una señal de un sensor de electrofisiología que es suavizada con un filtro convolucional del moving average.

### 18.1.2 Padding

El proceso de convolución requiere tener un cuidado especial con los bordes. Como el soporte de la señal es finito y requiere alinearse con el núcleo, hay que manejar que pasa en los borde, en relación a como se alinean la señal y el núcleo. Hay tres maneras de realizarlo. La primera es **valid** e implica alinear los valores del kernel con los valores existentes de la señal. Al estar haciendo un promedio, esto implica que el tamaño del resultado de esta operación, tiene menos cantidad de elementos (se reduce el tamaño del soporte en la salida). El segundo es **same** que corresponde a la situación donde se **paddea**, se agregan, a la señal original con dos ceros, uno de cada lado, y con eso permite que la señal de salida tenga exactamente la misma longitud que la de entrada, manteniendo el tamaño (por eso same). Finalmente, la tercera opción **full** implica agregar dos ceros de padding de cada lado, para que la señal generada incremente su tamaño.

Por ejemplo, con una señal discreta  $[1,2,-5,4,2,-1]$ , con un kernel  $[-1,2,-1]$ , se obtienen los siguientes resultados con cada esquema de padding:

- Valid:  $[8, -16, 11, 1]$
- Same:  $[0, 8, -16, 11, 1, -4]$
- Full:  $[-1, 0, 8, -16, 11, 1, -4, 1]$

El esquema de padding determina entonces cómo es la reacción de la operación a las condiciones de borde, y el tamaño de la señal de salida.

## 18.2 Convolución en Imágenes

La operación de filtrado básica en Visión por Computadora es mediante la aplicación de una operación de convolución bidimensional con diferentes kernels de convolución. En el caso de una imagen hay dos índices por los que se mueve la operación, el alto y el ancho, pero intrínsecamente es la misma operación.

$$\begin{aligned} S(i, j) &= (I * k)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n) \\ &= \sum_m \sum_n I(i - m, j - n) K(m, n) \end{aligned}$$

En Visión por Computadora o Análisis y Tratamiento de imágenes, es necesario aplicar 'filtros' a las imágenes para resaltar ciertas características. Estos filtros se basan en la idea de componer, a mano, mediante **feature engineering**, un kernel de convolución que genere lo que se quiere resaltar y con eso podés identificar información en la imágen. Por ejemplo, para detectar bordes

horizontales o verticales, como muestra la figura 18.3, se usan filtros que enfatizan los cambios repentinos en los valores de los píxeles en la dirección horizontal o vertical.

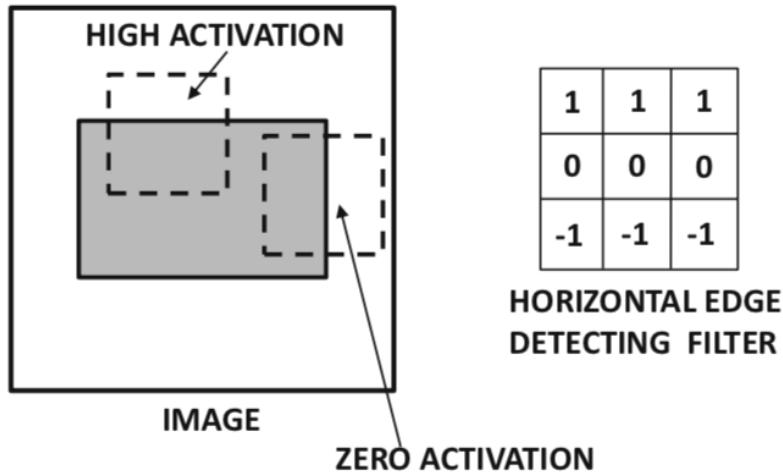


Figura 18.3: Filtros convolucionales para detectar bordes.

Los filtros de convolución de  $3 \times 3$  que permiten detectar estos bordes son matrices como las siguientes:

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}, S_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

Al aplicar la operación de convolución con este kernel a cada uno de los píxeles de una imagen (seguido por una operación de umbralización) se van obteniendo imágenes de salida donde se enfatizan los bordes en una dirección, en otra o en ambas.

Durante varios años, el área de Computer Vision se basaba en algoritmos, que con mucho conocimiento sobre la dinámica de la generación de imágenes visuales, proponían diferentes kernels de convolución, creados a mano, que generaban salidas específicas que servían para identificar diferentes patrones en las imágenes.

### 18.3 Neocognitrón

Los sistemas nerviosos biológicos son excelentes identificando de manera rápida y en tiempo real patrones visuales.



Figura 18.5: Kunihiko Fukushima, el creador del Neocognitrón.

En el trabajo de Wiesel y Hubel, ellos descubrieron que la tasa de los disparos de las neuronas de diferentes capas de la corteza visual de los gatos, estaba correlacionada con la orientación, con el ángulo en el que inclinaban una barra que presentaban en el espacio visual del gato. Para ángulos cercanos a 45 grados, la tasa era mayor. Al entender que la corteza visual, responsable de esta increíble acción ingenieril, está compuesta de capas jerárquicas que son sensibles a orientaciones en las imágenes predominantes en el campo visual, estas ideas cimentaron la posibilidad de encontrar una estructura computacional que pudiese artificializarla.

En 1989, Kunihiko Fukushima, quien se había sumado a un grupo

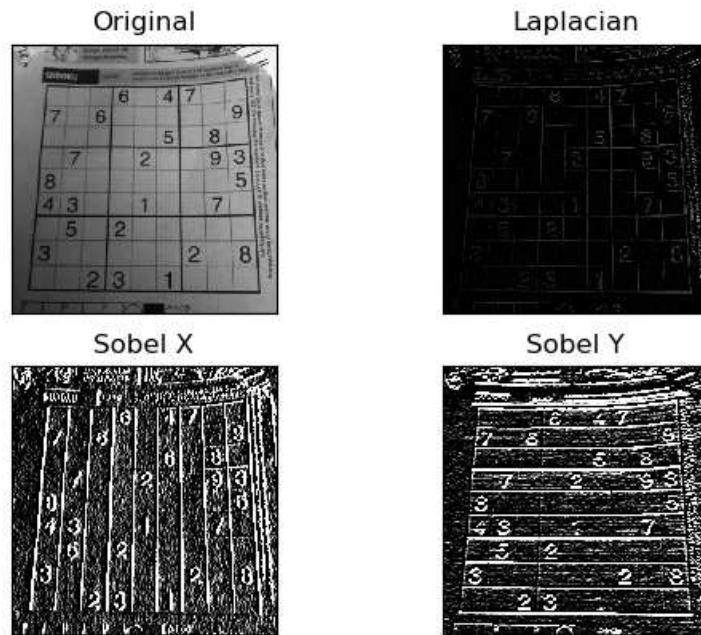


Figura 18.4: Resultado de aplicar la operación de convolución con diferentes filtros que provocan que se resalten los bordes horizontales o verticales. El Laplaciano es el resultado de sumar las derivadas segundas de los valores de la imagen, con lo cual se identifican al mismo tiempo tanto los bordes horizontales como los verticales.

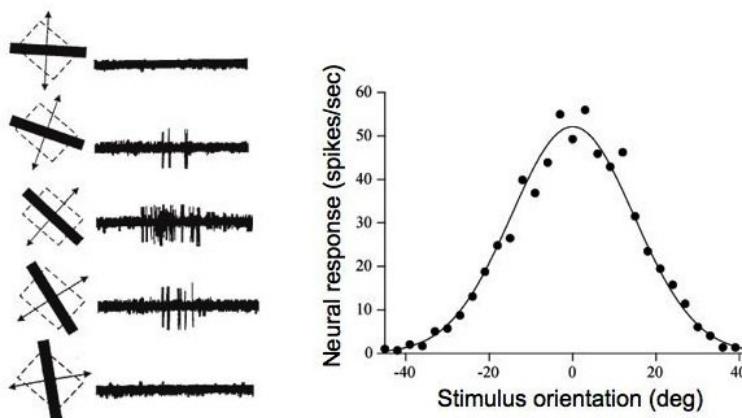
de investigación en la NHK<sup>1,2</sup> que estaba compuesto de manera interdisciplinaria, se enteró de los avances en neurociencia, del trabajo de Wiesel y Hubel y materializó la idea del Neocognitrón, con un fin práctico concreto de identificar caracteres en imágenes, un sistema de OCR.

En las propuestas del Neocognitrón, y en su formalización matemática por LeCun de manera independiente, se establece la idea de lo que son las redes convolucionales actuales. El quid de la cuestión es modificar un MLP de forma que realice una operación de convolución, de una capa a la otra, en vez de una operación de sumatoria de todas las entradas con pesos diferentes, de forma que la operación además actúe como filtro. Esto permite imitar la misma idea de lo que se observa en la corteza visual, mediante la utilización de la convolución como operación. .

La figura 18.7 muestra la estructura general de una red convolucional, similar a la propuesta por Fukushima/LeCun. La entrada es procesada por capas de convolución donde se aplica la susodicha operación, luego se aplica la operación de *pooling* para finalmente procesarse normalmente por un Perceptrón multicapa. La visión de redes neuronales es que los filtros tradicionales que se usan en visión por computadora son versiones shallow de redes neuronales convolucionales, y en la profundidad reside justamente uno de los éxitos de esta alternativa[131].

<sup>1</sup>La BBC<sup>2</sup> nipona.

<sup>2</sup>La TV Pública inglesa.



Hubel &amp; Wiesel, 1968

Figura 18.6: En el experimento de Hubel y Wiesel, la tasa de disparos neuronales en las diferentes capas de la corteza visual del gato se correlacionan con la inclinación de la barra que le muestran al minino.

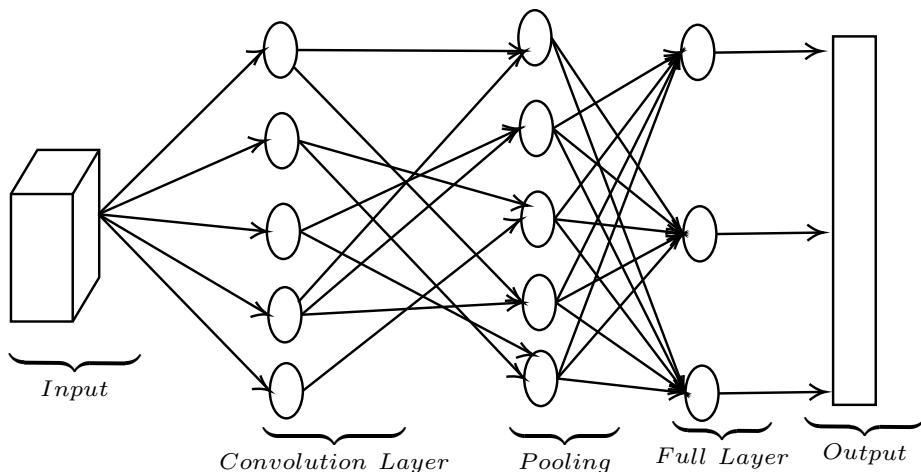


Figura 18.7: Estructura general de una red convolucional con sus diferentes capas.

## 18.4 Capa Convolucional

La capa de convolución es central, y sirve entenderla como una operación similar a la del Perceptrón, pero donde los pesos utilizados son compartidos **espacialmente**. Los diferentes componentes se pueden visualizar en la fig 18.8 y en la ecuación 18.1. La operación en cuestión es de la capa  $q$  a la capa  $q+1$ .

$$W^{p,q} = [w_{ijk}^{(p,q)}], h_{ijp}^{(q+1)} = \sum_{r=1}^{Fq} \sum_{s=1}^{Fq} \sum_{k=1}^{dq} w_{rsk}^{(p,q)} h_{i+r-1, j+s-1, k}^q \quad (18.1)$$

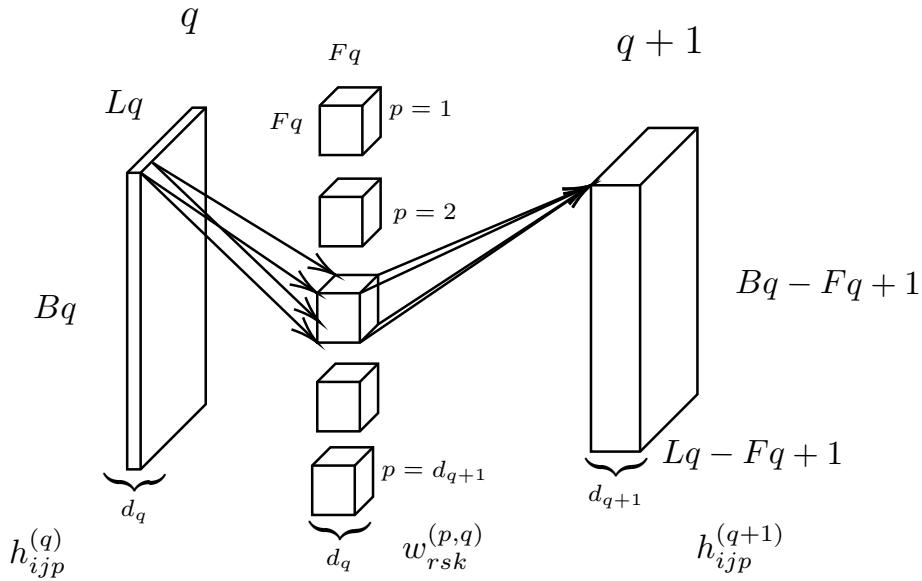


Figura 18.8: Componentes de la capa de convolución

 $i : \text{height}, \forall i \in \{1, \dots, Lq - Fq + 1\}$ 
 $q : \text{Layer}, j : \text{width}, \forall j \in \{1, \dots, Bq - Fq + 1\}$ 
 $k : \text{depth}, p : \text{Filter}, \forall p \in \{1, \dots, d_{q+1}\}$ 

La imagen de entrada es  $h_{ijp}^{(q)}$  de dimensiones  $Lq \times Bq$ , de ancho y de alto, conformada en  $d_q$  colores<sup>3</sup>. De la capa  $q$  a  $q+1$  se posicionan, por ejemplo,  $p$  filtros convolucionales, kernels, que tienen dimensiones de  $Fq \times Fq$ , por ejemplo  $3 \times 3$  pero tienen que tener exactamente la misma cantidad de colores que la imagen de entrada, para poder asignar a cada pixel, de cada canal, cada uno de los pesos del kernel. En los kernels es donde está los pesos, los parámetros libres de las redes convolucionales  $w_{rsk}^{(p,q)}$ . La salida de la capa convolucional, es una nueva imagen, más chica (asumiendo valid como padding), de  $Lq - Fq + 1 \times Bq - Fq + 1$ , y con una cantidad de colores equivalente a la cantidad de filtros en esta capa de convolución  $d_{q+1}$ . De esta manera cada filtro trabaja en paralelo, generando un canal más en la imagen de salida.

La no linealidad, al igual que en el Perceptrón multicapa, es aportada por funciones de activación que toman la salida de  $h_{ijp}^{(q+1)}$  como entrada. Particularmente en redes convolucionales, pensadas para imágenes donde los valores pueden ser muy grandes, se usan funciones de activación como ReLU que la ventaja que tienen es que no satura tan rápido y permite que el gradiente tenga un rango de valores posibles mucho más grandes, aportando más información, información más fina, para el ajuste de los pesos (que con funciones sigmoideas como la tangencia hiperbólica, rápidamente saturarían llevando todo a un mismo valor).

La definición de Relu es muy simple:

<sup>3</sup>Los colores en las imágenes tradicionales, Red, Green y Blue son los utilizados normalmente para el espectro de imágenes visibles por los seres humanos pero perfectamente pueden extenderse a muchos más, es totalmente arbitraria la diferenciación. Muchas imágenes son **hiperrespectrales** indicando justamente que tienen muchos canales.

$$f(x) = x^+ = \max(0, x) \quad (18.2)$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

Al ser una operación de convolución, dos hiperparámetros aparecen nuevos, que son el **stride**, que es el paso del salto espacial que se va haciendo al pasar de un pixel al siguiente (y calcular el promedio ponderado de todos los de alrededor). Y el segundo, por supuesto, es los **efectos de bordes** y como manejar el padding.

## 18.5 La capa de Pooling

La capa de Pooling está **cableada**, es decir no tiene ningún parámetro libre entrenable. Sirve para capturar invariantes traslacionales, y operan de manera similar a una operación de umbralización. Diferentes regiones de la imagen van pasando por capas convolucionales, que son filtros, y enfatizan características particulares de la imagen. La capa de pooling permite que independientemente donde esté localizado ese punto enfatizado, eso se tenga en cuenta en capas sucesivas, y tenga relevancia, para afectar la salida de lo que la red genera. El resultado de esta capa es similar al efecto obtenido mediante el **downsampling**, o mediante la reducción de la resolución de una imagen. La operación de pooling generalmente captura una región de la imagen y la sintetiza en un único pixel, normalmente eligiendo el mayor, conformando el **MaxPooling**, tal como se ve en la figura 18.9. Esta invariancia traslacional puede extenderse a hacer otras características invariantes, dependiendo como se agrupen los filtros y las capas de pooling subsiguientes.

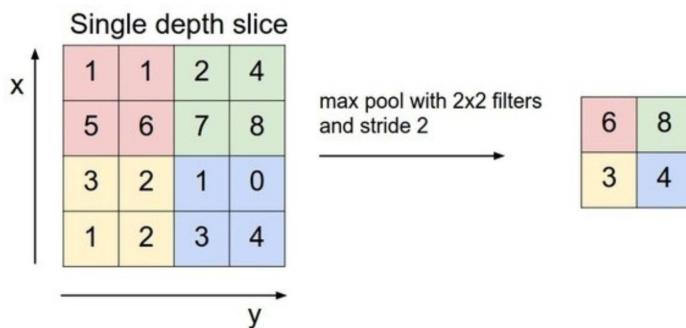


Figura 18.9: La capa de pooling resume la información obtenida en capas anteriores de forma de proveer una invariancia traslacional.

## 18.6 Full Layer y Softmax

Al final de la Red (fig 18.7) hay una (o más capas) que se encargan de hacer la clasificación final (si ese es el objetivo). Esta subred es tradicionalmente un perceptrón multicapa entrenado para hacer una clasificación multiclase de la imagen de la entrada. En redes convolucionales es muy común utilizar la función de activación Softmax definida en eq. 18.3.

$$\sigma(\vec{V})_i = \frac{e^{V_i}}{\sum_{j=1}^K e^{V_j}} \quad (18.3)$$

con  $V_i$  como la salida lineal de la neurona de la capa final. Esta función actúa también como un normalizador de la salida, enfatizando las diferencias para dar como resultado una única salida ganadora, aquella que tenga el mayor valor de salida dado por eq. 18.3. Al ser una normalización, la salida de todos los valores suma 1 por lo que se lo puede interpretar como una función de probabilidad.

## 18.7 Backpropagation

¿Cómo impacta el cambio en la estructura de la red en las capas de convolución para la aplicación del algoritmo de Backpropagation? Lo que se busca siempre es mantener la identificación de la influencia de cada peso al momento de retropropagar los errores. Por eso es que en las capas de convolución, como los pesos son compartidos, es necesario acumular todas las variaciones a la hora de realizar la actualización de los pesos. Existen varias maneras de resolver esto. La primera es la trivial de simplemente desplazar el kernel espacialmente e ir acumulando las modificaciones que se requieren hacer en los pesos. Esto es lo mismo que ocurre en cada época en relación al impacto de cada entrada, pero ahora hay que hacerlo a través de la distribución espacial.

1. Supongamos una red para realizar una clasificación en tres clases.
2. La Red se inicializa al azar, igual que siempre.
3. Se calculan los valores hacia adelante, aplicando la operación de convolución en las capas correspondientes hasta alcanzar los valores de salida, capa a capa.
4. En las capas *Fully Connected* es exactamente igual que el Perceptrón multicapa.
5. En las capas de Pooling, solo se actualizan la contribución de los pesos de la salida ganadora (asumiendo una función identidad).
6. En las capas convolucionales, hay que mantener los pesos de los filtros en la relación espacial.

### Backpropagation mediante representación en matrices esparsas

La operación de convolución puede verse como una manera eficiente de realizar una multiplicación de matrices reutilizando pesos. Desde esta perspectiva es posible hacer una representación de la operación de convolución en una matriz esparsa como se detalla en la figura 18.10. Para ello

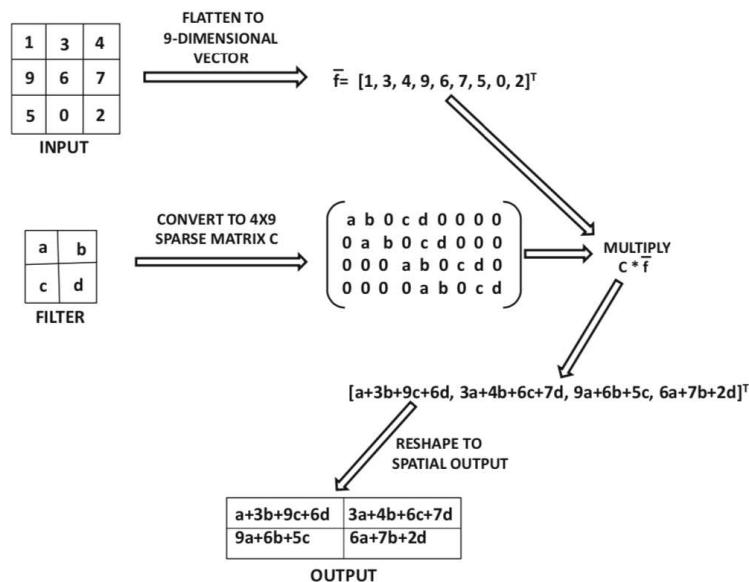


Figura 18.10: La representación esparsa de la operación de convolución permite operar de la misma forma que lo que ocurre en un perceptrón multicapa.

se representan los pesos del kernel en representaciones vectoriales unidimensionales del largo de todos los elementos de la matriz de entrada de la operación (en la figura es 9). Y cada peso va ubicado en la misma posición que corresponde a como es la alineación entre el peso y el valor de entrada. Cada fila de esa nueva matriz, representa una alineación de los pesos sobre la entrada. Esto genera una matriz C esparsa donde la mayoría de las entradas son ceros. Luego se puede hacer la multiplicación directa de la entrada, tomada como un vector **chorizado, flatten** de 9 elementos con la matriz C. El resultado de esto corresponde a lo que sería la matriz de salida de la operación completa de convolución. En ella los pesos quedan como parámetros que pueden utilizarse de esa manera, como una capa de multiplicación lineal, para lo cual aplicar la retropropagación mediante la regla de la cadena es trivial.

## 18.8 ¿Por qué funcionan?

Las redes convolucionales funcionan muy bien para capturar comportamiento estructurado en jerarquías e invariante a las traslaciones, muy común de imágenes visuales. Las razones son:

- Matrices esparsas son mucho más eficientes: menos cómputos.
- Los parámetros libres son compartidos. Esto hace que se les exige más, que capturen mejor la información. Esto es similar al mecanismo de compresión establecido en un autoencoder en la capa latente, o a adicionar un término regularizador en la función de costo.
- *Equivariante*  $g(f(x)) = f(g(x))$ , donde  $f$  es la traslación. Esto representa que da igual primero trasladar y tratar de identificar que identificar y luego trasladar.
- Jerarquías Espaciales: Ingeniería de Características Jerárquica. Hay una composición jerárquica de estructuras que son similares independientemente de la escala.

## 18.9 Inteligibilidad

### Inteligibilidad y Explicabilidad

La definición dada en 9.1.1 estructura algoritmos que aprenden relaciones entre datos existentes pero no asumen ningún modelo mecanístico en su accionar. Esto implica que por su propia estructura, es difícil entender por qué razón arriban a alguna conclusión particular en cuanto a clasificar o regresionar. Esta es la característica de caja-negra de todos estos algoritmos [132], que ya había sido maravillosamente anticipada por el propio Turing en 1950 como una consecuencia natural de generar un automatismo que tuviese la capacidad de aprender<sup>a</sup>. En contraposición, lo que se requiere de estos algoritmos cuando las decisiones que se toman son relevantes, es dos principios relacionados pero diferentes. El primero es la propiedad de legibilidad o **Intelligible Property** que se refiere a la capacidad del esquema de ofrecer una representación intermedia que le permita a un ser humano entender cuál es la característica que el algoritmo identificó para tomar una decisión. Por ejemplo, en una placa radiográfica remarcar las zonas que caracterizan una neumonía. Por otro lado, la segunda propiedad es la explicabilidad o **Explainability** que es la capacidad del sistema de explicar el paso a paso, la cadena de razonamiento, que lleva a tomar una decisión particular. Esto en modelado directo a como los seres humanos podemos seguir una cadena de razonamiento y dar una explicación. Estos principios suelen englobarse en lo que se denomina **xAI**, Explainable Artificial Intelligence. Dado el alcance e impacto social que estas tecnologías están teniendo, al actuar estos modelos como agentes de decisión, es necesario tener estos puntos en consideración, e incluso ir un paso más allá y considerar esquemas que además sean **Responsible AI**, es decir, diseñar estos sistemas ofreciendo garantías y margenes de funcionamiento, alineandolos con estándares éticos y regulatorios.

<sup>a</sup>An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is

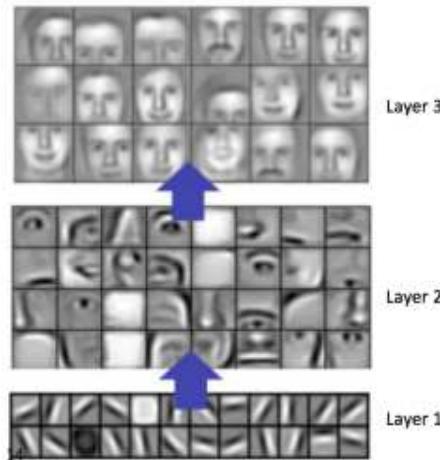


Figura 18.11: Las diferentes capas de una red convolucional capturan características más abstractas de la información contenida en la imagen y que sirven también para explicar el resultado final que ofrece la red.

going on inside, although he may still be able to some extent to predict his pupil's behaviour [5]"

La posibilidad de la aplicación de filtros convolucionales, que son pesos a los que se les puede asignar un sentido particular como coeficientes de un filtro, permiten que las capas intermedias de la red capturen comportamiento que ayuda tanto a ofrecer legibilidad como eventualmente generar una secuencia de explicabilidad. Esto está bien representado en lo que se denomina los mapas de saliencia o **Saliency Maps**, que capturan cuáles son los elementos de las capas intermedias convolucionales que de variar levemente generan un cambio rotundo en la salida de la red. Al estar representados en imágenes, se representan como información que puede ser comprendida directamente por seres humanos.

## 18.10 Arquitecturas Reusables

Un aspecto excelente de las redes convolucionales es que las diferentes combinaciones de capas de convolución, ReLU, pooling y fully connected, permiten armar arquitecturas profundas que pueden reusarse como modulos reutilizables que actúan como componentes de redes más complejas. Esto hace que sean una tecnología muy versátil y explican porque razón están en los corazones. Por ejemplo, los pesos de la capa de convolución se asumen que generan filtros adecuados para procesar imágenes visuales, se pre-entrenan y se disponibilizan, como el caso de los *Pretrained FC7 Features*, o como los existentes en redes conocidas, descargables y reutilizables como:

- LeNet
- AlexNet
- VGG

Como siempre, la elección de la arquitectura correcta no escapa a las generalidades de las redes neuronales. Un poco arte, un poco de *educated guesses*, y prueba-y-error.

### 18.10.1 ResNet

Como las imágenes suelen ser enormes, las redes convolucionales son realmente profundas y el VGP(?) se vuelve muy importante. Una alternativa para reducir sus efectos es utilizar lo que se denomina conexiones residuales. Esto es, al pasar por una capa, se mantiene una conexión residual directa, que actúa como un perceptrón lineal, que se adiciona a la salida propia de la capa

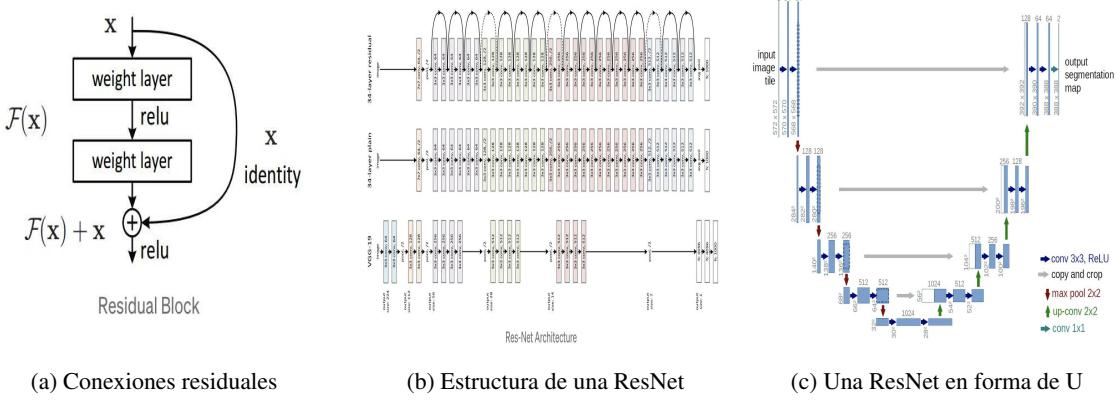


Figura 18.12: Extensiones de redes convolucionales

de convolución. Esto logra que la red capture el cambio pero minimiza la posibilidad de que el gradiente se haga cero, ya que mantiene los valores cerca de los que fueron tomados en la entrada.

### 18.10.2 UNet

El primer uso que se les dio a las redes convolucionales fue para clasificación, pero pueden ser muy útiles para hacer segmentación, o incluso alterar una imagen, o completarla como en situaciones de **impainting**. En este caso, se requiere aplicar capas sucesivas que obtengan buenos filtros sobre la imagen, pero luego capas adicionales que permitan recuperar esa información para regenerar una imagen de características similares a la original. En este caso, la idea de ResNet se extiende en las UNets: en estas, las diferentes capas de convolución capturan información importante de una imagen, pero luego se aplica la operación inversa, la deconvolución, que sucesivamente genera imágenes intentando reversar el proceso de deconvolución pero usando las salidas de los filtros. Para que esto tenga sentido, se agregan conexiones residuales que provienen de las imágenes originales de cada capa convolutiva y que generan el efecto de recuperar la información que existía en la imagen a ese nivel. Por eso el nombre de esta red, ya que lo que provoca es un dibujo en forma de U donde las salidas de cada capa de la red, se usan para ahondar en más profundidad la aplicación de los filtros, pero a su vez se utilizan también como entrada para regenerar la imagen en el **up-scaling** de deconvolución del otro lado del valle de la U.

### 18.11 Más allá de las imágenes

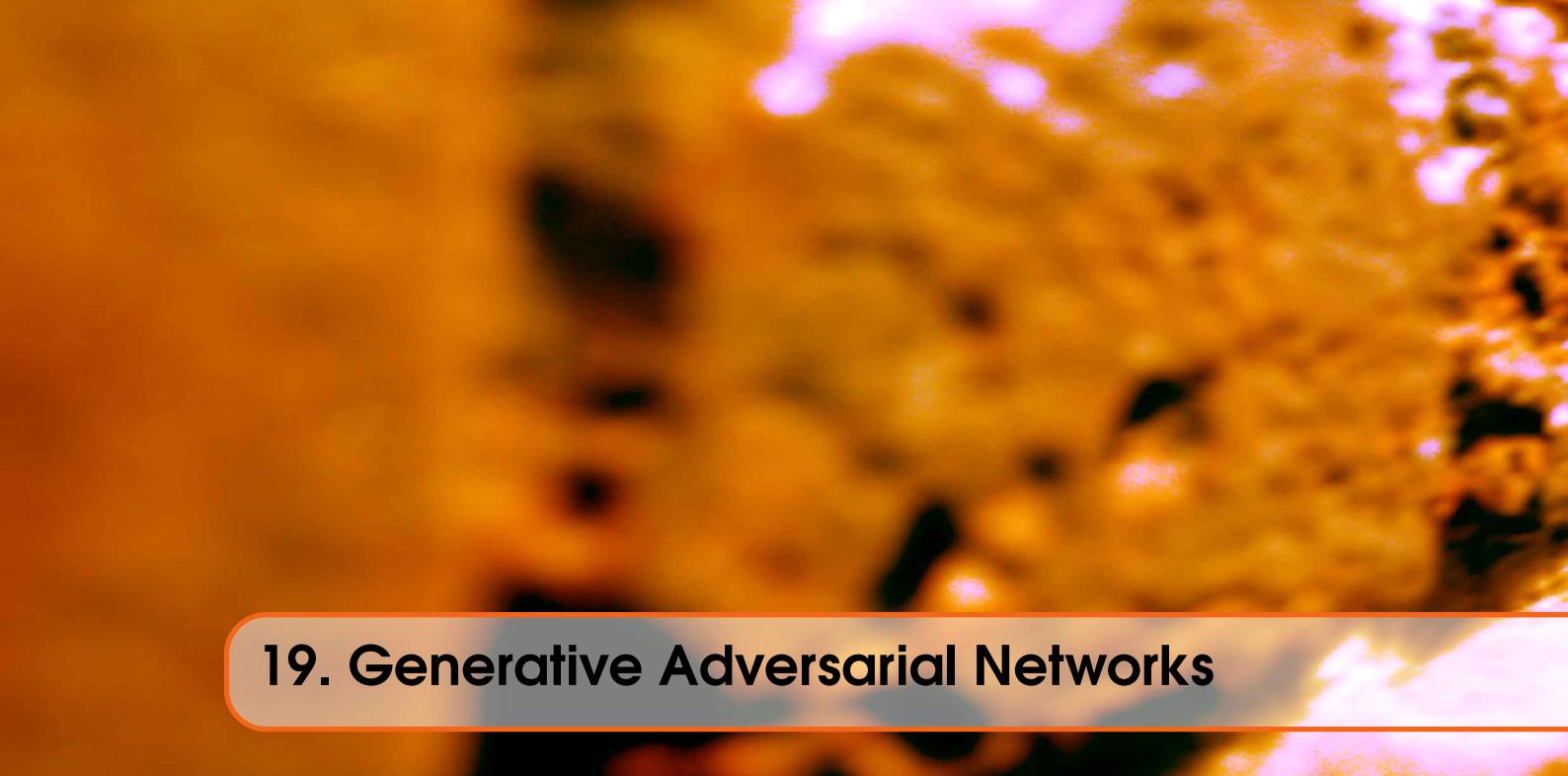
Las redes convolucionales son muy versátiles en relación al tipo de datos y problemas en las cuales pueden usarse, dependiendo de cómo se realiza la asignación de las dimensiones. La tabla 18.11 muestra algunos ejemplos.

Dimensiones	Canal único	Multicanal
1-D	Audio	Electroencefalografía Marcadores Captura Movimiento
2-D	Espectrogramas	Imagenes a color
3-D	Tomografía Computada	Video

## 18.12 Quiero más

- Fukushima Neocognitron [133].
- LeCun 1989, LeNet Architecture [134].
- Libro de Aggarwal Neural Networks and Deep Learning [135].
- Guía para entender la aritmética de las convoluciones [136, 137]
- Estupendo resumen de CNN desde una perspectiva super variada [138]
- AlexNet 2012 [139].
- <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>





## 19. Generative Adversarial Networks

Las redes GAN, Generative Adversarial Networks, o redes adversariales generativas, surgen aproximadamente al mismo momento que los Autoencoders variacionales, abordando desde una perspectiva distinta la misma problemática: cómo **generar** muestras nuevas de un conjunto  $X$ . Fueron propuestas por un grupo de investigadores del núcleo de Deep Learning, entre los que incluye Ian Goodfellow y Yoshua Bengio. Cuenta la historia, que Ian había estado pensando en una red **Discriminadora** que sirviera para evaluar si diferentes mecanismos de síntesis de habla eran buenos o no, y durante una tertulia en un bar con amigos nerds, discutiendo acerca de las complejidades sobre como generar imágenes de manera eficiente, se le cruzó la idea de combinar un generador con un discriminador y usar la información del segundo para precisamente entrenar y ajustar al primero. El remate de la historia es que esa misma noche empezó a codear la idea de las GANs [140].

### 19.1 Generando gatitos

Típicamente, dado por ejemplo un conjunto de imágenes de gatos, similar a lo que ocurrió el capítulo 17 con Autoencoders, queremos identificar la distribución de probabilidad que caracteriza a las imágenes de los gatos. Es decir queremos encontrar un modelo generativo.

**Definition 19.1.1 — Modelo Generativo.** Un modelo generativo describe la estructura de generación de un dataset  $X$ , en términos de su estructura probabilística. Sampleando de este modelo, es posible generar nuevas muestras  $X_j$  que serán probabilísticamente indistinguibles de las muestras originales  $X_i$ .

El problema por supuesto es estimar esa distribución. Las redes GAN [141] buscan solucionar esto planteando una interacción entre dos redes. Una primer red es la responsable de discriminar entre las muestras reales pertenecientes al conjunto  $X$ , el dataset real, versus muestras artificiales. Este discriminador entonces está sometido a recibir muestras de entrada y de manera supervisada aprender a decidir si cada una de las muestras produce el label de salida 1 o –1 indicando si pertenecen o no al conjunto real de muestras o son artificiales. A esta red la llamamos **Discriminador  $D$** .

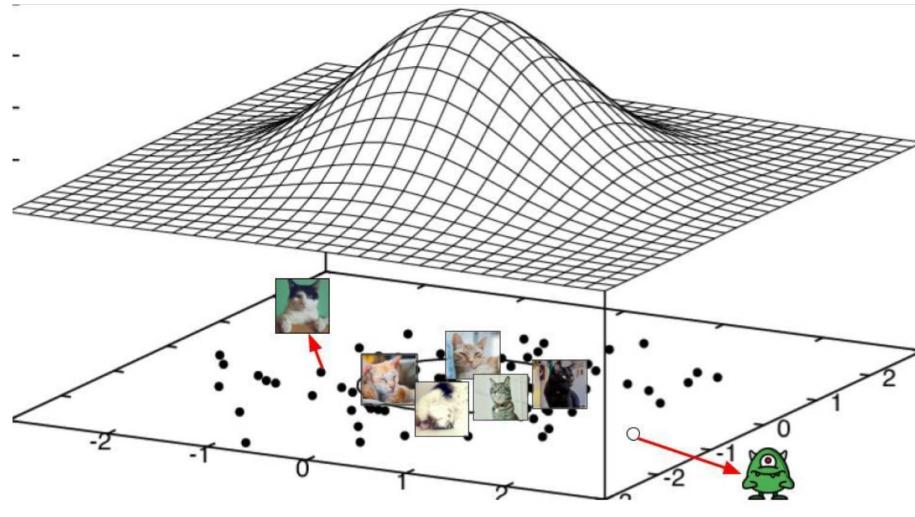


Figura 19.1: Modelo generativo de imágenes de gatitos

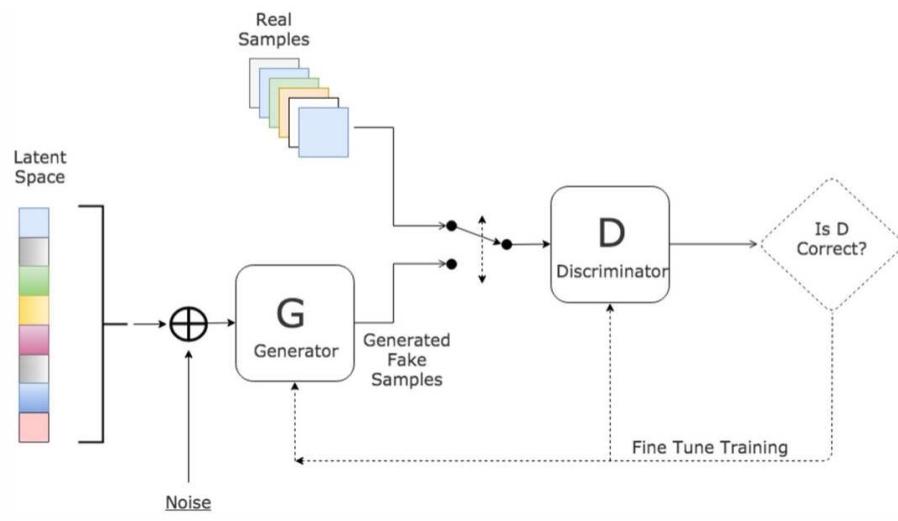


Figura 19.2: Arquitectura de una red GAN, compuesta por una red Generadora y otra Discriminadora que juegan un juego adversarial.

Por otro lado, una segunda red que llamaremos **Generador** va a tener el objetivo de producir las muestras falsas. Esta red actúa de una manera muy similar al decodificar del Autoencoder. De un espacio latente, obtiene una muestra al azar, a la cuál se le agrega ruído, y la misma pasa por una red con pesos libres que va a generar una nueva muestra que será sometida al discriminador para ser juzgada.

El algoritmo opera de la siguiente manera:

1. Dado un dataset  $X$  de muestras reales válidas.
2. Se inicializa la red Discriminadora y Generadora.
3. Se utiliza la red Generadora para aumentar el dataset  $X$  con un conjunto de muestras artificiales.
4. Se entrena la red Discriminadora de manera supervisada para intentar ajustar los pesos para predecir si la muestra es real o artificial, hasta convergencia.
5. Dejando fijos los pesos del Discriminador, se entrena a la red Generadora para generar muestras de forma que la red discriminadora acepte como verdaderas. Para este paso, la red Generadora + la red Discriminadora se ven como una única red.

El proceso de entrenamiento continua alternando entre el entrenamiento de una red y la otra, y por supuesto lo importante es definir una buena función de costo que capture el objetivo subyacente de este entrenamiento.

## 19.2 Juego MinMax

Hay un conflicto acá presente. Una red quiere una cosa, y la otra red quiere exactamente la opuesta. El Discriminador quiere identificar con claridad cuáles son las muestras del dataset original, mientras que el Generador quiere generar una muestra que el Discriminador no pueda, justamente, discriminar. Este conflicto se media mediante una función de costo que capture la película completa encuentre un balance.

Los elementos de la ecuación son:

- $D$  = Discriminador
- $G$  = Generador
- $\phi_d$  = Parámetros del Discriminador
- $\phi_g$  = Parámetros del Generador
- $P_z(z)$  = Distribución del ruido de la entrada del generador.
- $P_{data}(x)$  = Distribución de los datos originales.
- $P_g(x)$  = Distribución de los datos generados.

Primero la ecuación desde la perspectiva del Discriminador:

$$L_D = H(y, D(x)) = \frac{1}{N} \left\{ \sum_{i=1}^N (y_i) \log(D(x_i)) + (1 - y_i) \log(1 - D(x_i)) \right\} \quad (19.1)$$

teniendo en cuenta que  $y$  son las etiquetas reales en tanto que  $D(x)$  son las etiquetas asignadas por el discriminador, las predicciones.

Lo que el discriminador quiere es que cuando la muestra es real, el primer término sea 1 y el segundo término desaparezca. Esto es que el discriminador acierte, y que  $D(x_i) = 1$ . Por el contrario, cuando la muestra es falsa, el discriminador quiere que el primer término desaparezca y que  $1 - D(x_i) = 1$ .

La ecuación 19.1 puede rescribirse como:

$$\max_D H(y, D(x)) = \max_D \mathbb{E}_{x \sim P_{data}} \{ \log(D(x)) \} + \mathbb{E}_{z \sim P_z} \{ \log(1 - D(G(z))) \} \quad (19.2)$$

donde lo que el Discriminador quiere es maximizar esa función para justamente acertar cuando corresponde hacerlo.

Por otro lado, lo que el Generador quiere, desde su perspectiva es precisamente minimizar la ecuación 19.2: provocar que el Discriminador no identifique cuando las muestras han sido generados por el Generador.

Esto es, a la vez, optimizar:

$$\min_G \max_D V(D, G) = \min_G \max_D \left\{ \mathbb{E}_{x \sim P_{data}} \{ \log(D(x)) \} + \mathbb{E}_{z \sim P_z} \{ \log(1 - D(G(z))) \} \right\} \quad (19.3)$$

En el paper [141] demuestran que el mínimo de esta ecuación se produce en un punto de equilibrio de Nash [45], donde se alcanza precisamente lo que se planteó al principio: el discriminador no tiene información para poder discriminar una muestra de  $X$  de las generadoras por el generador  $G(z)$ , sólo puede tirar la moneda, mientras que el Generador produce muestras nuevas que tienen la misma distribución estadística que la de los datos originales, es decir,  $P_z \rightarrow P_{data}$ .

### 19.3 Colapso Modal

Las redes GAN tienen dos grandes inconvenientes. El primero es que el orden en el que se realiza el entrenamiento alternado del Discriminador y del Generador es muy importante desde un punto de vista pragmático para alcanzar el punto de equilibrio entre las dos redes, y consecuentemente de la red GAN completa. El segundo problema, que está relacionado al anterior, es que puede ocurrir una situación que se denomina *colapso modal* que es donde la red encuentra un punto de solución alternativo donde diferentes puntos del espacio latente generan un único punto, una única muestra en vez de generar un espacio completo o de capturar la verdadera distribución [142].

### 19.4 El gran simulador

Las GAN presentan un fenómeno interesante. Comienzan a generar imágenes que realmente capturan la escencia presente en los datos, y nos permiten a los seres humanos corroborar eso con nuestros propios ojos. Esto puede visualizarse en la obra de varios artistas digitales [143] tal como puede verse en la figura 19.3.

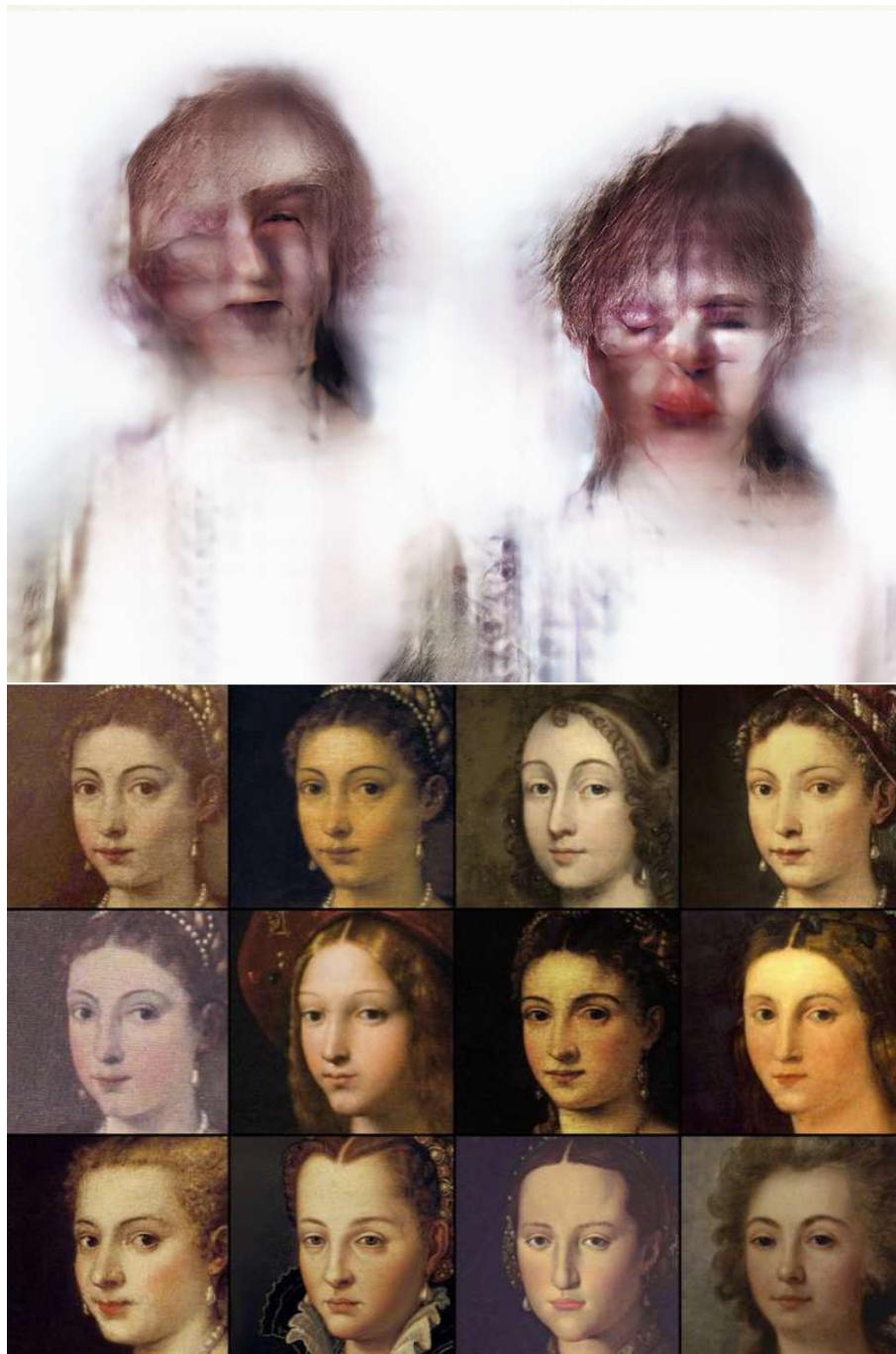


Figura 19.3: Imágenes generadas con redes GANs. En la imagen de la derecha es difícil identificar cuál de esas imágenes es obtenida de un cuadro real o cuáles fueron generadas artificialmente.



## 20. Transformers

Los transformers fueron propuestos por [26] en el famoso trabajo **Attention is all you need**. Su propuesta era la conclusión en ese momento de muchos años de trabajo e intentos para implementar un mecanismo adecuado para el procesamiento del lenguaje natural, NLP (Natural Language Processing), el procesamiento de tokens de texto, en una secuencia, y solucionando los problemas del *vanishing/exploding gradient*, VGP, de las redes recurrentes y los problemas de las redes LSTM. Un tema curioso, que no puede escapar del propio hype que afecta a toda esta área, es lo opaca que es la literatura en relación a muchos de estos temas. Por eso es que vamos a intentar poner blanco sobre negro en relación a poder identificar de donde vienen los nombres y a qué es que referencian precisamente.

### 20.1 Las Redes Recurrentes

Las redes neuronales recurrentes tienen fuertes conexiones con sistemas físicos complejos y dinámicos (ver capítulo 13). El modelo de una neurona recurrente puede visualizar en la Figura 20.1 [144].

Las redes recurrentes se caracterizan porque tiene una entrada  $x_t$  y una salida  $y_t$ , y al igual que en el modelo de neurona tradicional la entrada suele ser vectorial en tanto que la salida escalar. Pero

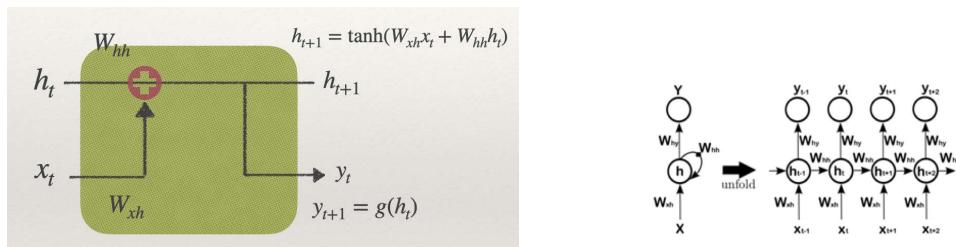


Figura 20.1: RNN, Recurrent Neural Networks, redes recurrentes. La red opera como un modelo con delay, con una salida adicional que representa un estado que sirve de entrada para el paso siguiente.

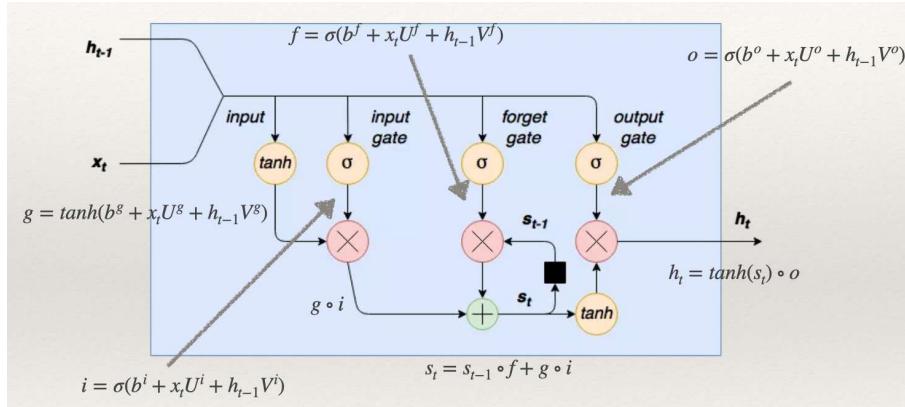


Figura 20.2: Las redes LSTM, Long Short Term Memory, fueron pensadas específicamente para poder superar el problema del vanishing/exploding gradient con un mecanismo de activación de una memoria interna para lograr cadenas de inferencia causales que acepten muchos capas temporales.

además, las neuronas recurrentes tienen una salida adicional que representa un estado interno, que tras un delay es una entrada adicional al paso de procesamiento que sigue. El esquema dinámico que esto plantea, queda discretizado por el delay recibiendo  $h_t$  en la entrada y generando una salida  $h_{t+1}$ . Puede ocurrir, como en las redes de Hopfield, que la salida y el estado sean lo mismo y no estén diferenciados. La figura 20.1 muestra el esquema general, donde la entrada y el estado anterior son multiplicados por matrices de pesos y el estado siguiente y la salida son calculadas después de aplicar una función no lineal.

Las redes recurrentes son la materialización del problema del vanishing/exploding gradient. En la Figura 20.1, del lado derecho, se muestra el proceso de unfolding. Las redes recurrentes, por su caracterización dinámica, se pensaron para procesar información en secuencia, de series de tiempo, o de procesos estocásticos. Por ejemplo, lenguaje escrito. Así el primer token de la frase ingresa como input a la red, afecta la salida en el estado  $h_{t+1}$  que a su vez se propaga al próximo estado, donde recibe un nuevo input, que alterna otra vez el estado, y así sucesivamente. Esto, rápidamente, crece en la cantidad de pasos y genera naturalmente una restricción numérica en como retropropagar el error al pasado por muchas, muchas iteraciones<sup>1</sup>.

### 20.1.1 Long Short Term Memory

En 1997 Hochreiter y Schmidhuber[145] proponen una solución a este problema de las redes recurrentes, planteando una alternativa de red que hacía un uso inteligente de compuertas de activación neuronales, que le permitían a la red guardar, recuperar y eliminar información relevante en una memoria, un estado interno de la red, donde la red ajustando automáticamente los pesos aprendía cuando tenía que guardarla, recuperarla o eliminarla, y que el almacenamiento en estado interno resolvía el problema de tener que retropropagar los errores por muchas capas.

Las redes LSTM fueron muy exitosas y fueron el estándar de facto para los teclados predictivos de los denominados **feature phone**, los teléfonos celulares de GSM con teclados físicos pequeños, que estaban pensados para mandar mensajes de texto. La limitación que encontraron era que el aprendizaje de la red era complejo, y su generalización muy acotada, ya que el mecanismo de memoria interna le era complejo poder encontrar un gran abanico de abarcabilidad de usos, y se volvía un excesivo cuello de botella.

<sup>1</sup>De hecho el concepto de Deep de deep learning, hace referencia tanto a la profundidad espacial de las redes, como a la profundidad temporal que se genera en las redes recurrentes y que rápidamente puede crecer de manera muy abrupta y exponencialmente con el problema.

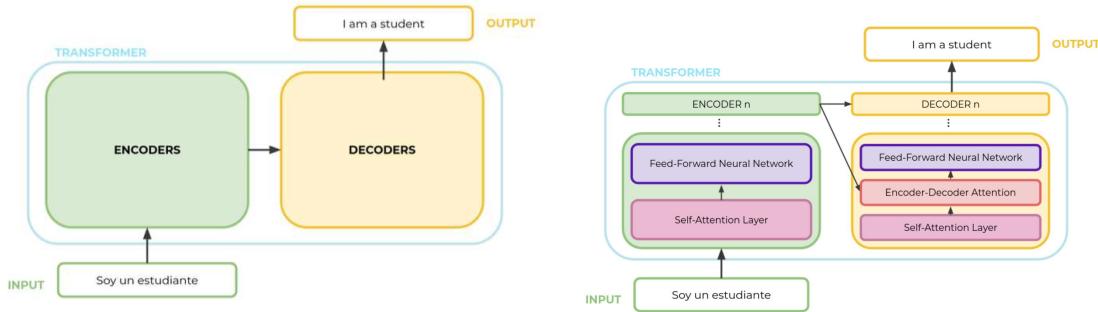


Figura 20.3: Diagrama de componentes básicos de un Transformer.

## 20.2 Arquitectura de un Transformer

Los Transformers fueron pensados para interpretar secuencia de tokens en texto escrito. Tienen una arquitectura general que está determinada por un **Encoder** y un **Decoder**, influenciado la nomenclatura quizás por la arquitectura de los Autoencoders aunque sin ningún tipo de relación directa. El objetivo del Encoder es procesar las entradas. En tanto que el Decoder recibe información de la entrada pero también de la salida hasta ese momento.

Al pensar al Transformer como una arquitectura para resolver la traducción de un texto de un lenguaje en otro, aparece la justificación de los diferentes componentes.

La red primero tiene un stack de bloques de Encoders y Decoders. En el paper, usan 6 bloques de Encoders y Decoders.

**Encoder:** El Encoder recibe la información de la cadena de entrada. Tiene dos componentes, el primero corresponde a una capa que implementa el mecanismo de **Atención**, específicamente el mecanismo de **Self-Attention**, que relacionará los lexemas de la entrada entre sí. La segunda capa es un MLP. La salida pasa al Encoder subsiguiente del stack.

**Decoder:** La salida final del último Encoder es la entrada a cada uno de los Decoders, y reciben una entrada adicional que viene de la salida hasta el momento para el primer Decoder, y de la salida del Decoder anterior para todos los otros. El Decoder tiene una capa de Self-Attention, luego una capa de Encoder-Decoder-Attention (que toma la salida de la capa anterior versus las entradas recibidas del Encoder) y finalmente un MLP. La salida final del último Decoder genera la salida actual, en este caso el token actual de traducción.

Los Encoders tienen el objetivo de encontrar las relaciones entre los lexemas de la entrada. La primera capa, atiende la relación de lexema contra lexema. La segunda capa toma las relaciones entre tuplas de lexemas entre sí. Y así sucesivamente.

Luego los Decoders relacionan las entradas con los elementos de la salida que se ha generado hasta ese momento (por lo que la salida actual es la entrada principal del primer Decoder del Stack). Los Decoders subsiguientes siguen la misma lógica de los Encoders para procesar n-tuplas.

Al final de la cadena de Decoders, se ubica una capa lineal, seguida por una función SoftMax y finalmente se recodifican los lexemas (numéricos) en lexemas reversando la operación de los Embeddings. La capa SoftMax determina cuál es el output de cada uno de los tokens posibles, y la idea es elegir aquel que tenga la salida mayor<sup>2</sup>.

## 20.3 Attention is all you need

El mecanismo de atención es central en los Transformers hasta el punto que es casi un sinónimo que representa principalmente el detalle más fundamental de la arquitectura. Es una capa interna

<sup>2</sup>Softmax puede verse como una probabilidad ya que la salida suma uno pero no corresponde verdaderamente a las probabilidades de salida, ya que no esté calibrada.

que poseen todos los Encoders y Decoders, cuya función es tener en cuenta otras palabras de la secuencia mientras se está analizando una en particular. En el caso de los Encoders, esta capa le permite poner el foco en cualquier otra palabra de la entrada, en tanto que los Decoder únicamente pueden analizar palabras previas en la secuencia de salida.

La capa de atención permite relacionar los lexemas del lenguaje entre sí. Por ejemplo, de la secuencia "Yo soy un estudiante", la palabra estudiante está referenciando el pronombre *Yo*. Por eso los pesos, los parámetros libres en el mecanismo de atención, deberían ser más fuertes entre estas dos palabras.

Hay tres maneras de ver al mecanismo de atención.

$$\text{SoftMax} \left( \frac{QK'}{\sqrt{d_k}} \right) * V \quad (20.1)$$

con la definición de la función de activación SoftMax como  $\sigma(\vec{z})_i = \frac{e^{\vec{z}_i}}{\sum_{j=1}^K e^{\vec{z}_i}}$ .

### 20.3.1 Una tabla de acceso indexado continua

La nomenclatura de las variables utilizadas en la descripción del mecanismo de atención derivan de este abordaje. Supongamos que tenemos una tabla de acceso indexado, una  *hashtable*. Necesitamos obtener los datos de una persona, en base al DNI. El *query Q* representa el dato del DNI de la persona. Este query *Q* lo usamos para acceder a un índice para obtener el valor del índice, la clave *K*. Finalmente con *K* podemos acceder a la tabla real y obtener finalmente *V*, el valor que necesitamos.

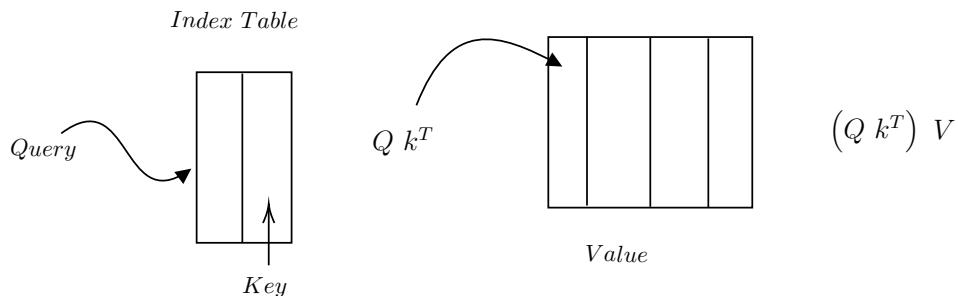


Figura 20.4: Acceso indexado a una tabla de manera continua.

Ahora podemos pensar a este esquema en una variante continua, donde no accedemos a las entradas exactas de la tabla sino establecemos una medida de similitud y vamos a buscar las más parecidas. Con esto, el query *Q* necesitamos buscar el índice más parecido de la tabla. Usemos el producto interno como medida de similitud(puede haber otras). Entonces para ver cuál de todos los índices *k* es el más parecido, hacemos

$$Q.k^T$$

. Luego con este resultado, accedemos a la tabla real para encontrar el valor real que se parezca lo más posible a esta medida, y otra vez usamos el producto interno, mediado por una función de activación que rescala los valores para identificar de todos al más parecido

$$(Q.k^T).V$$

Finalmente, si consideramos un escalado de los datos y que sean una suma convexa, para distribuir el peso entre todos, nos queda otra vez:

$$\sigma \left( \frac{Q \cdot k^T}{\sqrt{d_k}} \right) * V \quad (20.2)$$

Lo que obtenemos con esto es una manera de poder primero identificar la clave, el índice que es más parecido al query (al dato que tenemos en foco). Luego necesitamos ver cuál es el valor, el Value, que es más parecido a la combinación del query con la clave. La ventaja de esto, es que podemos agregarle pesos, parámetros libres, que van a servir para mediar estos cálculos y así establecer mecanismos de aprendizaje de estas relaciones.

$$\sigma \left( \frac{QW_q \cdot (kW_k)^T}{\sqrt{d_k}} \right) * VW_v \quad (20.3)$$

donde

- $Q$  : Query, dato de entrada en el que está el foco.
- $k$  : Key, clave, los índices, los tokens a los que hay que prestarle atención.
- $V$  : Value, los valores que se van a generar de salida, ponderados por el núcleo.
- $W_q, W_k, W_v$ : pesos que actúan como parámetros libres,

### 20.3.2 Kernel Smoothing

Supongamos que tenemos un conjunto de datos  $X$  y sus supuestas salidas deseadas  $Y$ , siendo los primeros la cadena de lexemas de entrada y la segunda la de salida. Dado un  $x_o$  de entrada objetivo que queremos traducir, algo básico que se puede usar es utilizar **kNN**, con  $k = 1$ , identificar el vecino  $x_i$  más cercano y su correspondiente  $y_i$ . Si eventualmente queremos tener también la información de otros vecinos cercanos, podemos tomar un  $k$  mayor, y promediar los valores de  $y_i$  que corresponden a varios de esos vecinos

$$\hat{y}_o = \sum_{i=1}^k \frac{1}{k} y_i \quad (20.4)$$

El problema es que esto no tiene en cuenta cuán cerca cada vecino está del query  $x_o$ . Lo que podemos hacer entonces es hacer un promedio ponderado, donde le demos más peso a aquellos puntos vecinos que sean más similares a  $x_o$ . Para eso agregamos un **kernel**  $K(x_i, x_o)$  que nos va a indicar cuán similar es uno del otro. Con ese promedio ponderado, la ecuación anterior queda:

$$\hat{y}_o = \sum_{i=1}^k \frac{K(x_i, x_o)}{\sum_{j=1}^k K(x_j, x_o)} y_i \quad (20.5)$$

Esta técnica se denomina Kernel Smoothing [146]. Como los valores  $x$  son vectores, como función de kernel se puede utilizar cualquier función de similitud vectorial como el producto interno o variantes como  $K(x_i, x_o) = \exp(x_i \cdot x_o)$  u otra variante donde se puede definir el producto interno en base a una matriz  $K(x_i, x_o) = \exp(x_i \cdot w_1^T w_2 \cdot x_o)$ . Y eventualmente se puede normalizar con la dimensión de los vectores usando  $\sqrt{d}$ .

Si reemplazamos,  $x_o \rightarrow Q$ ,  $x_i \rightarrow k^T$  y  $y_i \rightarrow V$ , llegamos a la ecuación del mecanismo de atención que desde esta perspectiva se la puede ver como promedio ponderado por kernels de similitud entre la clave  $k^T$  y el query  $Q$  de valores  $V$ .

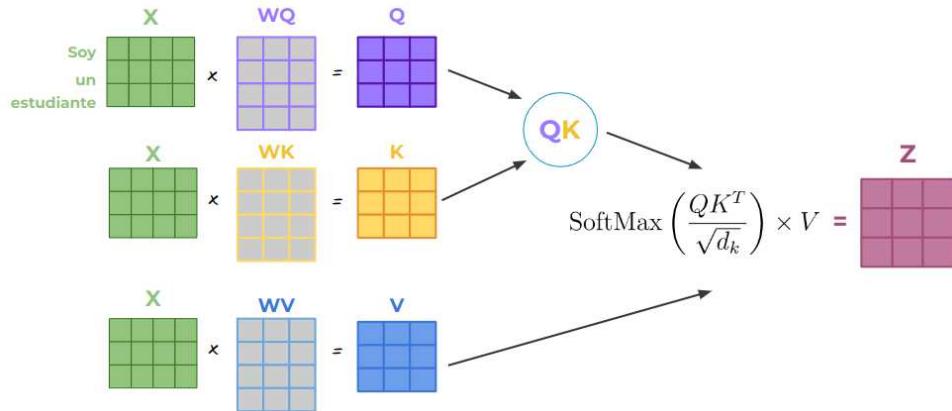


Figura 20.5: Mecanismo básico de Attention y sus componentes.

$$\hat{y}_o = \sum_{i=1}^k \frac{K(x_i, x_o)}{\sum_{j=1}^k K(x_j, x_o)} y_i \quad (20.6)$$

y a esto se le pueden agregar los pesos como parámetros libres:

$$\left( \frac{\exp(QW_q \cdot (kW_k)^T \cdot \sqrt{d_k}^{-1})}{\sum_{i=1}^N \exp(QW_q \cdot (kW_k)^T \cdot \sqrt{d_k}^{-1}))} \right) * VW_v \quad (20.7)$$

### 20.3.3 Atención Cognitiva

Este último aspecto se refiere a una biomimesis y paralelo entre el mecanismo de Atención de los Transformers y el mecanismo cognitivo humano de *Atención Cognitiva*: este se refiere al proceso por el cual los organismos seleccionan un subconjunto de la información disponible para concentrarla, integrarla y procesarla de manera más precisa. Tiene tres aspectos distintivos: la orientación, el filtrado y la búsqueda. La atención está relacionada con la conciencia [147] y a diferencia del mecanismo de **Attention** de los Transformers depende de un estado interno.

Todo esto es muy lindo, pero necesitamos convertir las cadenas de lexemas en números y para eso necesitamos un esquema de codificación.

## 20.4 Embedding

Los Embeddings son los diferentes esquemas de codificación de los lexemas. El primer proceso consiste en dividir la entrada en componentes. En lenguajes como inglés o español la división es mas directa ya que por ejemplo se podría dividir la entrada en palabras. Este es un proceso denominado **tokenization**. Las palabras (o *tokes*) se transforman en números en base a un diccionario global de codificación. Luego, cada token es representado en un vector numérico de dimensión fija, capturando relaciones semánticas y sintácticas en función de su contexto en el lenguaje. Para ello se aplica un algoritmo de embedding. Existen varias alternativas como *WordToVec*, *CBOW*, *Bag of Words* o incluso *SVD*.<sup>3</sup>

<sup>3</sup>Tener en cuenta que se está hablando de una ejemplo donde se toma como input un texto pero no necesariamente un token debe ser una palabra e incluso se pueden tener imágenes o señales de audio.

Este esquema tiene el problema de que el orden es invariante a las permutaciones, lo cuál sabemos que no es algo característico del lenguaje. Por lo tanto también se requiere que la codificación tenga en cuenta la ubicación del token en la secuencia.

**Positional encoding:** Una posible solución es agregar información a cada palabra sobre su posición en la oración. A esta información se la llama Positional Encoding, lo que es un conjunto de  $p_i$  valores que se le suma a cada embedding. Si bien hay diversas formas de abordar este inconveniente, en el paper de [26] se opta por utilizar las siguientes funciones sinusoidales para obtener el vector de posición del token  $i$ :

$$PE(pos, 2i) = \sin\left(\frac{pos}{1000^{\frac{2i}{d}}}\right) \quad (20.8)$$

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{1000^{\frac{2i}{d}}}\right) \quad (20.9)$$

donde

- $pos$  es la posición del token en la secuencia.
- $i$  es el índice del vector de posición.
- $d$  es la dimensión de un embedding, la cual será la misma que un vector de posición  $p_i$ .

## 20.5 Entrenamiento

Para el entrenamiento, hay que definir un vocabulario, incluyendo el *EOF*. Además hace falta el vocabulario también en inglés, el idioma destino (e.g.). La estructura del Transfomer es muy versátil y puede adaptarse a muchos tipos de problemas similares.

Los pesos se inicializan de forma aleatoria. El modelo sin entrenar devuelve una distribución de probabilidad con valores totalmente arbitrarios. Es necesario entonces seguir iterando, de manera supervisada, para que justamente la salida salga la palabra que se desea y así forzar toda la estructura para aprender [148].





## 21. Guía para el apurado

### 21.1 Condimentos de una red neuronal

Cuando se usa una red neuronal para solucionar un problema se requieren decidir los siguientes componentes.

**Arquitectura:** necesitan decidir la arquitectura de la red, decidir la cantidad de capas, y la cantidad de neuronas por capas. Así también como los diferentes componentes y cómo se encadenan.

**Función de Activación:** dependiendo de la naturaleza del problema, entender cuál debería ser la función de activación más pertinente. Y sobre todo cuáles van a ser los rangos de valores de entrada y de salida de esas funciones.

**Función de Costo:** elegir la mejor función del costo en base al objetivo que se le dará a la red: clasificar, regresionar, optimizar o generar. Por ejemplo, la función de error tradicional 21.1 es la que se utiliza en MLPs pero no es la única.

#### 21.1.1 Transformación de los datos

Existen diferentes maneras de preprocesar y transformar los datos, y la nomenclatura no es muy clara en relación a qué es cada caso. Acá ofrecemos una aclaración a modo de resumen y de selección arbitraria de la nomenclatura que se utiliza en todo el texto.

- Normalizar
- Min-Max Scaling
- Feature Scaling
- Standardization Z-Score
- Box-Cox.

##### Definition 21.1.1 — Normalization.

$$X_{new} = \frac{X}{||X||} \quad (21.1)$$

**Definition 21.1.2 — Min-Max Scaling.**

$$X_{new} = \frac{X_{old} - X_{min}}{X_{max} - X_{min}}(b - a) + a \quad (21.2)$$

**Definition 21.1.3 — Feature Scaling.**

$$X_{new} = \frac{X_{old}}{X_{max}}, X_{min} = 0, b = 1, a = 0 \quad (21.3)$$

**Definition 21.1.4 — Z-Score, Standardization = Statistical Standardization , Estandarizar.**

$$X_{new} = \frac{X_{old} - \bar{X}}{X_{std}} \quad (21.4)$$

donde  $\bar{X}$  es la media y  $X_{std}$  el desvío estándar.

**Definition 21.1.5 — Box-Cox Normalization.** Convierte los datos mediante una optimización para que luzcan normales (que provienen de una distribución  $\mathcal{N}()$ )

**21.1.2 Balanceo de clases**

Tanto cuando se utiliza aprendizaje supervisado como no-supervisado, el balanceo de los datos, o el sesgo inherente, son importantes considerarlos para poder evaluar con claridad los resultados [149]. Esto es un aspecto clave bien conocido, pero que sin embargo muchas veces suele olvidarse completo. Por ejemplo, al utilizar una red neuronal con aprendizaje supervisado e implementar un clasificador binario. Si los datos están extremadamente desbalanceados, una población de cientos de datos contra millones entre una clase y la otra, un clasificador que siempre asigna el label de la clase que tiene más muestras tendrá una eficiencia altísima, totalmente inútil.

**21.1.3 Funciones de Costo**

Existen diferentes funciones de costo que pueden utilizarse en las redes neuronales [150]. La tabla 21.1 muestra algunas de las más utilizadas que hemos tocado en este texto.

Nombre	Expresión
SSE Adaline	$\frac{1}{2} \sum_{\mu=1}^p (\zeta^\mu - O^\mu)^2$
RMS	$\sum_{i,\mu} \left\{ \frac{1}{2} (1 + \zeta_i^\mu) \log \frac{1+\zeta_i^\mu}{1+O_i^\mu} + \frac{1}{2} (1 - \zeta_i^\mu) \log \frac{1-\zeta_i^\mu}{1-O_i^\mu} \right\}$
MAE	$\frac{1}{p} \sum_{\mu=1}^p  \zeta^\mu - O^\mu $
Binary Cross Entropy	$-\frac{1}{p} \sum_{i=1}^p \{(y_i) \log p(y_i) + (1 - y_i) \log(1 - p(y_i))\}$
Huber	$\begin{cases} \frac{1}{p} \sum_{i=1}^p \frac{1}{2} (\zeta^\mu - O^\mu)^2 &  \zeta^\mu - O^\mu  \leq \delta \\ \frac{1}{p} \sum_{i=1}^p \delta ( \zeta^\mu - O^\mu  - \frac{1}{2} \delta) &  \zeta^\mu - O^\mu  > \delta \end{cases}$

Cuadro 21.1: Diferentes funciones de error

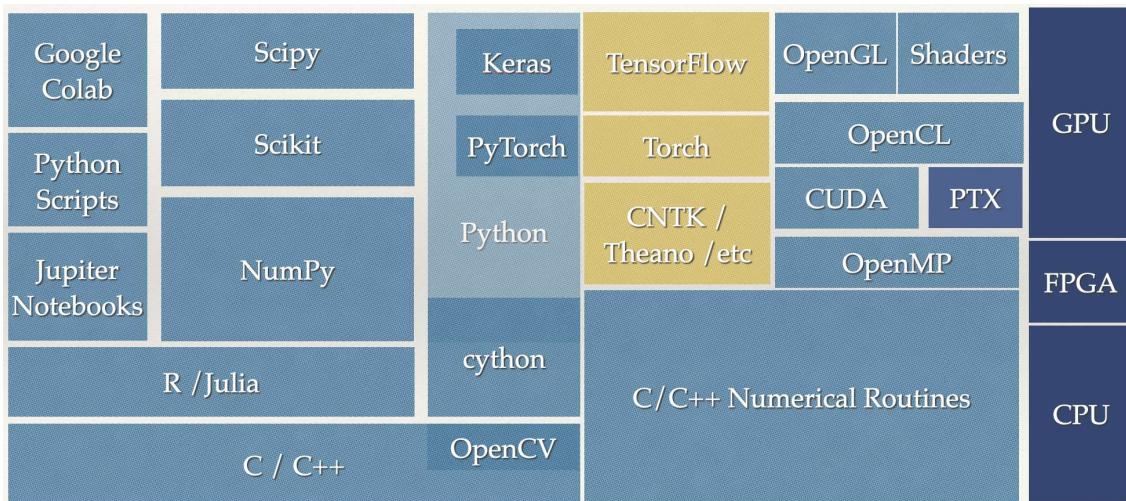


Figura 21.1: Un esbozo de la infraestructura de software y hardware que sustentan el stack que se utiliza para el desarrollo de aplicaciones y sistemas de análisis de datos e inteligencia artificial. Mayoritariamente son librerías de código abierto.

### 21.1.4 Funciones de Activación

También existen diversas funciones de activación que se utilizan en las redes neuronales, y que son las responsables de agregarle principalmente la no-linealidad. Desde las más simples y tradicionales hasta las más complejas que se utilizan en escenarios muy específicos, como Softplus y Mish que son versiones suavizadas de ReLU. La tabla 21.2 detalla las utilizadas en este texto. Tengais en cuenta vosotros (jeje) que la derivada acá está expresada en los términos de las entradas no de las salidas (es decir están expresadas en los términos de la entrada de la función de activación no de su salida).

## 21.2 Infraestructura de Software

Como se expone en este documento, el desarrollo de esta disciplina está cimentado en el hardware que permitió la realización en tiempo y forma de muchas de las ideas y trucos aquí también expuestos. Pero es además menester destacar la infraestructura de software montada también para esta tarea y que permite estructurar un *stack* de procesamiento sobre el que construir hacia arriba. El diagrama en bloques de la figura 21.1 muestra el stack de plataformas y librerías más utilizadas para el desarrollo de modelos y soluciones de Inteligencia Artificial.

### CPU, GPU y FPGA

Al sustrato de hardware tradicional de CPUs, más el agregado actual de GPUs, se le suma también la posibilidad de programar en otros como FPGAs. Los **Field Programmable Gate Arrays** son dispositivos electrónicos de propósito general que pueden modificarse electrónicamente para reestructurar los circuitos de forma que queden de manera específica, cableados, maximizando la eficiencia. Los **ASIC**, **Application Specific Integrated Circuit**, son una variante que no pueden reprogramarse posteriormente. Este nivel de especialización se materializa con hardware específico para las típicas operaciones de inteligencia artificial, en hardware concreto como los **NPU**, **Neuro Processor Units**.

### OpenGL y Shaders

OpenGL es la librería tradicional abierta para la programación gráfica. Es una vieja librería, muy especializada que está orientada para la creación de gráficos en tres dimensiones. Encapsula

la compleja lógica requerida para la programación en 3D en un set de instrucciones de alto nivel. La aparición del hardware específico gráfico, los inicios de las GPUs, forzó a la creación de los shaders de GLSL, **G**raphics **L**ibrary **S**haders **L**anguage, que es código que directamente se compila y corre en el GPU. Esto permitía resolver la lógica dual de procesamiento, donde código general de un juego corre principalmente en un CPU, pero los cálculos gráficos, i.e. multiplicación de matrices, corren directamente manipulado por GLSL en el GPU, y se utilizan interfaces de copia de información estandarizadas entre la memoria uno y otro.

### **OpenMP y OpenCL**

OpenMP es una API que sirve para realizar programación en paralelo, pensada justamente para poder portar de manera simple programas hechos en C++ a cualquier tipo de plataforma. Justamente, es agnóstico del sistema operativo y del hardware, y además está basado en cláusulas para el preprocesador facilitando la modificación de programas existentes. OpenCL por otro lado, fue concebido para poder correr el mismo código en diferentes tipos de procesadores, como ser CPUs, GPUs, FPGAs, etc. Eso lo hace también útil para construir y desarrollar programas que requieren alta capacidad de procesamiento.

### **CUDA y PTX**

CUDA es una de las razones del dominio de NVidia sobre el mercado de los GPUs. Es una plataforma de software muy completa para programar directamente utilizando las capacidades de paralelismo de los GPUs de NVidia. Esta librería permite estructurar los programas para operar de manera tradicional en arquitecturas similar x86-64 y sobre eso derivar código que opera sobre la memoria de la GPU, realizando los intercambios de datos entre las diferentes memorias. PTX, **P**arallel **T**hread **E**xecution es el lenguaje de assembler para la máquina virtual en la que corren instrucciones que directamente se mapean a uno o más GPUs de NVidia. Es el lenguaje intermedio utilizado en CUDA.

### **Python**

Este viejo lenguaje de scripting ha encontrado su lugar y acceso al podio máximo como lingua franca para las actividades de data science e inteligencia artificial. Además de su sintaxis simple, una de las claves de este éxito es justamente la librería *cython* que permite interoperabilidad entre librerías numéricas muy optimizadas construidas en Fortran o C++ [151].

### **NumPy**

La más importante de esas librerías en el ecosistema de Python es Numpy [151]. Estas son rutinas de álgebra lineal para multiplicación de matrices, que realizan un manejo muy eficiente de la memoria. Son rutinas optimizadas para correr sobre CPUs.

### **SciPy**

Esta librería es parte del combo con muchas rutinas prácticas para el manejo de datos e información científica, particularmente procesamiento de imágenes y de señales. Es una librería clave que encapsula mucha funcionalidad asociada al preprocesamiento y curado de información científica.

### **Scikit Learn**

Librería base de las rutinas de aprendizaje automático para Python. Si hay una rutina de aprendizaje automático para Python, está en esta librería.

### **Pandas**

Pandas es una librería práctica que facilita el curado de datos, la ingeniería de datos. Modelada y estructurada en base al lenguaje y plataforma R, particularmente el concepto de *dataframe*. Esto facilita el uso de python para aquellas personas que ya conocen esta plataforma, que es fuertemente

orientada al análisis actuarial y estadístico. Un aspecto importante es que esta librería hace énfasis en los datos y su metadata asociada.

### **Matplotlib, Plotly, Seaborn, Altair**

Python es un mar de librerías para hacer prácticamente de todo. Una tarea básica que aparece en el procesamiento de los datos, y en IA orientada a datos, es justamente la necesidad de graficar información. **Matplotlib** es la librería tradicional para hacerlo, que aunque cuenta con una interfaz de funciones bastante dura, se puede hacer de todo. **Plotly** por otro lado, es mucho más simple de utilizar pero más limitada en las funcionalidades que implementa. **Seaborn** es mucho más refinada en cuanto a la usabilidad y el diseño gráfico de los elementos gráficos, y a la vez trabaja a la par de **Pandas** con lo que es muy útil para destacar información considerando también la metadata asociada. Sin embargo, un aspecto clave a considerar son las buenas prácticas de diseño asociadas a los gráficos científicos, y desde esa perspectiva, **Altair** es la gran ganadora porque justamente respeta diferentes lineamientos estéticos y de eficiencia comunicacional [152].

### **TensorFlow y Keras**

TensorFlow es una de las librerías principales que permiten la programación de modelos de redes neuronales y su ejecución en CPUs, o GPUs, facilitando la utilización de las librerías de bajo nivel como **CUDA**, **OpenML**, **OpenCL**, y estableciendo las abstracciones de capas, funciones de activación, funciones de costo, etc, necesarias para construir redes neuronales. Tensorflow tiene además un gran secreto, que es **AutoDiff** [153]. Estas librerías, en sus diferentes variantes en las diferentes plataformas, construye el árbol computacional, que determina cómo calcular el gradiente en base a las operaciones que se realizan en las diferentes funciones de un programa, y realiza los cálculos simbólicos y numéricos para determinar el gradiente de la operación. Esto por supuesto, es clave para la construcción de librerías de deep learning. Tensorflow es una librería de más bajo nivel para C++, y su porte a Python es Keras [64].

### **Torch y PyTorch**

Torch es otra popular librería que provee las bases y abstracciones para trabajar con redes neuronales profundas. Su versión para Python, PyTorch es utilizada en muchas implementaciones de redes neuronales cerradas que se distribuyen como componentes, como tal es el caso de redes neuronales convolucionales.

### **OpenCV**

OpenCV es una extraordinaria librería, muy abarcativa, que incluye una numerosa cantidad de algoritmos específicos para el procesamiento de imágenes y de visión por computadora. Entre ellos, incluye muchos paquetes que preceden a las librerías actuales de deep learning pero que tienen el mismo objetivo. Además es una herramienta muy útil para la creación de aplicaciones de todo tipo.

### **R y Julia**

R es un lenguaje pensado y estructurado para hacer análisis estadísticos, y por esa razón es muy popular también en ciencia de datos e inteligencia artificial. *Si una distribución de probabilidad no aparece en R, entonces no existe.* Julia por otro lado, es un lenguaje específicamente creado para el procesamiento de datos y para aprendizaje automático. ¿Será el lenguaje y plataforma del futuro de esta área? Quizás lo sea, o quizás lo sea el mismo lenguaje natural, pero por ahora la escena está dominado por el viejo Python.

## **21.3 Quiero más**

¿Más? Listo por hoy. A la calle, a respirar aire fresco y tomar un poco de sol.

Nombre	$g(h)$	$g'(h)$
Identidad	$h$	1
Tangente Hiperbólica	$\tanh(\beta h) = \frac{e^h - e^{-h}}{e^h + e^{-h}}$	$\beta(1 - g(h)^2)$
Sigmoidea	$\frac{1}{1+exp^{2\beta x}}$	$2 \beta g(h)(1 - g(h))$
Softsign	$\frac{1}{1+ x }$	$\frac{1}{(1+ x ^2)}$
Función Escalón	$\begin{cases} 0 & h < 0 \\ 0.5 & h = 0 \\ 1 & h > 0 \end{cases}$	Delta de Dirac $del(h)$
Función Signo	$\begin{cases} -1 & h < 0 \\ 0 & h = 0 \\ 1 & h > 0 \end{cases}$	$2 del(h)$
Lineal por partes	$\begin{cases} 1 & h >= \frac{1}{2} \\ h + \frac{1}{2} & -\frac{1}{2} < h < \frac{1}{2} \\ 0 & h <= -\frac{1}{2} \end{cases}$	$\begin{cases} 0 & h >= \frac{1}{2} \\ 1 & -\frac{1}{2} < h < \frac{1}{2} \\ 0 & h <= -\frac{1}{2} \end{cases}$
ReLU	$\max(0, h)$	$\begin{cases} 0 & h < 0 \\ 1 & h >= 0 \end{cases}$
Softplus	$\frac{1}{\beta} \ln(1 + e^{\beta h})$	$\frac{e^h}{1+e^h}$
Mish	$\tanh(\text{softplus}(h))$	$\frac{e^h \omega}{\delta^2}, \omega = e^{3h} + e^{2h}(4) + e^h(6 + 4h) + (1 + h), \delta = (e^h + 1)^2 + 1$

Cuadro 21.2: Funciones de activación simples con sus derivadas.

Nombre	$g(h)$	$g'(h)$
Softmax	$\sigma(\vec{V})_i = \frac{e^{V_i}}{\sum_{j=1}^K e^{V_j}}$	$\begin{cases} \sigma(\vec{V})_i(1 - \sigma(\vec{V})_i) & i = k \\ -\sigma(\vec{V})_i(\sigma(\vec{V})_i) & i \neq k \end{cases}$

Cuadro 21.3: Funciones de activación simples con sus derivadas (2/2).





## Bibliografía

- [1] A. M. Turing *et al.*, “On computable numbers, with an application to the entscheidungsproblem,” *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.
- [2] P. Stone, *The Artificial Intelligence Papers*. Sebtel Press, 2024.
- [3] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.
- [4] R. E. Brown, “The life and work of donald olding hebb,” *Acta Neurologica Taiwanica*, vol. 15, no. 2, p. 127, 2006.
- [5] A. M. Turing, *Computing machinery and intelligence*. Springer, 1950.
- [6] D. Taraborrelli and R. Gala, “Génesis y actualidad de la inteligencia artificial (ia) en las instituciones públicas de la argentina, una mirada desde los escyt,” in *XI Jornadas de Sociología. Facultad de Ciencias Sociales, Universidad de Buenos Aires, Buenos Aires, 2015*. Acta Académica, 2015, p. 01.
- [7] P. N. Johnson-Laird and M. Ragni, “What should replace the turing test?” *Intelligent Computing*, vol. 2, p. 0064, 2023.
- [8] A. Newell and H. Simon, “The logic theory machine—a complex information processing system,” *IRE Transactions on information theory*, vol. 2, no. 3, pp. 61–79, 1956.
- [9] P. Stone, R. Brooks, E. Brynjolfsson, R. Calo, O. Etzioni, G. Hager, J. Hirschberg, S. Kalay Krishnan, E. Kamar, S. Kraus *et al.*, “Artificial intelligence and life in 2030: the one hundred year study on artificial intelligence,” 2016.
- [10] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.
- [11] N. Wiener, “Cybernetics or control and communication,” *The animal and the machine*, 1948.

- [12] R. Solomonoff, “Dartmouth archives,” <https://raysolomonoff.com/dartmouth/>, 2024 (accessed Jan 10, 2025).
- [13] B. Bloomfield, “The question of artificial intelligence,” <https://www-formal.stanford.edu/jmc/reviews/bloomfield/bloomfield.html>, 2024 (accessed Jan 10, 2025).
- [14] N. Nilsson, *The quest for Artificial Intelligence*. Standford University, 2010.
- [15] S. L. Brunton and J. N. Kutz, *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2022.
- [16] R. McNaughton, “The theory of automata, a survey,” in *Advances in Computers*, ser. Advances in Computers, F. L. Alt, Ed. Elsevier, 1961, vol. 2, pp. 379–421. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0065245808601448>
- [17] C. Bassett, “The computational therapeutic: exploring weizenbaum’s eliza as a history of the present,” *AI & SOCIETY*, vol. 34, no. 4, pp. 803–812, 2019.
- [18] X. Yang and C. Zhu, “Industrial expert systems review: a comprehensive analysis of typical applications,” *IEEE Access*, 2024.
- [19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [20] M. Minsky and S. Papert, “An introduction to computational geometry,” *Cambridge tiass., HIT*, vol. 479, no. 480, p. 104, 1969.
- [21] A. Clayton, “How eugenics shaped statistics,” <https://nautil.us/how-eugenics-shaped-statistics-238014/>, 2019 (accessed Jul 18, 2024).
- [22] T. Haigh, “Between the booms ai in winter,” <https://cacm.acm.org/opinion/between-the-booms-ai-in-winter/>, 2024 (accessed Oct 10, 2024).
- [23] J. Pearl, “Probabilistic reasoning in intelligent systems. representation & reasoning,” 1988.
- [24] V. N. Vapnik, “Statistical learning theory,” 1998.
- [25] R. Szeliski, *Computer vision: algorithms and applications*. Springer Nature, 2022.
- [26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [27] R. Carnota and R. Rodriguez, “Antonio monteiro: pionero de los estudios de computación en argentina,” in *Simposio de Historia de la Informática en América Latina y el Caribe*, 2012.
- [28] P. M. Jacovkis, “Manuel sadosky (1914-2005),” *Revista de la Unión Matemática Argentina*, vol. 46, no. 1, pp. 67–71, 2005.
- [29] H. Monteverde, “60 años de sadio,” *Electronic Journal of SADIO (EJS)*, vol. 20, no. 2, pp. 2–22, 2021.
- [30] I. Daffra, *Historia de la industria informática argentina*. CESSI Argentina, 2014.
- [31] R. Wachenchauzer, “The evolution of computer education in latin america: the case of argentina,” *ACM Inroads*, vol. 5, no. 1, pp. 70–76, 2014.

- [32] J. M. Santos and C. Touzet, “Exploration tuned reinforcement function,” *Neurocomputing*, vol. 28, no. 1-3, pp. 93–105, 1999.
- [33] C. Hurtado, A. O. Mendelzon, and A. A. Vaisman, “Maintaining data cubes under dimension updates,” in *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*. IEEE, 1999, pp. 346–355.
- [34] A. A. Vaisman and A. O. Mendelzon, “A temporal query language for olap: Implementation and a case study,” in *International Workshop on Database Programming Languages*. Springer, 2001, pp. 78–96.
- [35] M. C. Parpaglione, “Neural networks applied to fault detection using acoustic emission,” in *Non-destructive Testing’92*. Elsevier, 1992, pp. 71–75.
- [36] A. C. Frery, H.-J. Muller, C. d. C. F. Yanasse, and S. J. S. Sant’Anna, “A model for extremely heterogeneous clutter,” *IEEE transactions on geoscience and remote sensing*, vol. 35, no. 3, pp. 648–659, 1997.
- [37] J. Gambini, M. E. Mejail, J. Jacobo-Berlles, and A. C. Frery, “Feature extraction in speckled imagery using dynamic b-spline deformable contours under the model,” *International Journal of Remote Sensing*, vol. 27, no. 22, pp. 5037–5059, 2006.
- [38] R. Kurzweil, “The singularity is near,” in *Ethics and emerging technologies*. Springer, 2005, pp. 393–406.
- [39] E. Strickland, “The turbulent past and uncertain future of ai: Is there a way out of ai’s boom-and-bust cycle?” *IEEE Spectrum*, vol. 58, no. 10, pp. 26–31, 2021.
- [40] T. Urban, *Wait but Why: AI Superintelligence*, 2022 (accessed July 1, 2022), <https://waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html>.
- [41] S. Legg, M. Hutter *et al.*, “A collection of definitions of intelligence,” *Frontiers in Artificial Intelligence and applications*, vol. 157, p. 17, 2007.
- [42] A. Ciaunica, A. Safron, and J. Delafield-Butt, “Back to square one: the bodily roots of conscious experiences in early life,” *Neuroscience of Consciousness*, vol. 2021, no. 2, p. niab037, 2021.
- [43] A. Clark, *Supersizing the mind: Embodiment, action, and cognitive extension*. OUP USA, 2008.
- [44] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. London, 2010.
- [45] S. Tadelis, *Game theory: an introduction*. Princeton university press, 2013.
- [46] P. A. Tipler, *Physics for Scientists and Engineers: Regular Version, Ch. 1-35 and 39*. Macmillan, 1999.
- [47] T. Virkkala, “Solving sokoban,” Ph.D. dissertation, Master’s thesis, University of Helsinki, 2011.
- [48] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1998.
- [49] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

- [50] J. H. Holland, “Genetic algorithms and the optimal allocation of trials,” *SIAM journal on computing*, vol. 2, no. 2, pp. 88–105, 1973.
- [51] N. Ashton and C. Stringer, “Did our ancestors nearly die out?” *Science*, vol. 381, no. 6661, pp. 947–948, 2023.
- [52] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
- [53] D. L. Goodstein, *States of matter*. Courier Corporation, 2014.
- [54] P. Pedregal, *Introduction to Optimization*. Springer, 2003.
- [55] H. A. Taha, *Investigación de operaciones*. Pearson Educación, 2004.
- [56] R. P. Brent, *Algorithms for minimization without derivatives*, Prentice-Hall, Ed. Englewood Cliffs, N.J., Prentice-Hall, 1973.
- [57] J. A. Snyman, D. N. Wilke *et al.*, *Practical mathematical optimization*. Springer, 2005.
- [58] P. Moritz, R. Nishihara, and M. Jordan, “A linearly-convergent stochastic l-bfgs algorithm,” in *Artificial Intelligence and Statistics*. PMLR, 2016, pp. 249–258.
- [59] L. Bottou, *Online Algorithms and Stochastic Approximations. Online Learning and Neural Networks.*, C. U. Press., Ed. Cambridge University Press., 1998.
- [60] M. Lefkowitz, *Perceptron pave way to ai 60 years ago*, 2024 (accessed July 1, 2024), <https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon>.
- [61] E. Agliari, M. Aquaro, A. Barra, A. Fachechi, and C. Marullo, “From pavlov conditioning to hebb learning,” *Neural Computation*, vol. 35, no. 5, pp. 930–957, 2023.
- [62] K. Shirriff, “The surprising story of the first microprocessors,” *IEEE Spectrum*, vol. 53, no. 9, pp. 48–54, 2016.
- [63] 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, p. 106, 1962.
- [64] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2021.
- [65] M. Hardt and B. Recht, “Patterns, predictions, and actions: A story about machine learning,” *arXiv preprint arXiv:2102.05242*, 2021.
- [66] G. Hinton, N. Srivastava, and K. Swersky, “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,” *Cited on*, vol. 14, no. 8, p. 2, 2012.
- [67] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [68] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009, vol. 2.
- [69] R. Roelofs, V. Shankar, B. Recht, S. Fridovich-Keil, M. Hardt, J. Miller, and L. Schmidt, “A meta-analysis of overfitting in machine learning,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.

- [70] C. F. Caiafa, E. Salerno, and A. N. Proto, “Blind source separation applied to spectral unmixing: comparing different measures of nongaussianity,” in *Knowledge-Based Intelligent Information and Engineering Systems: 11th International Conference, KES 2007, XVII Italian Workshop on Neural Networks, Vietri sul Mare, Italy, September 12-14, 2007, Proceedings, Part III 11.* Springer, 2007, pp. 1–8.
- [71] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, “Dbscan revisited, revisited: why and how you should (still) use dbscan,” *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.
- [72] T. Kohonen, “Self-organized formation of topologically correct feature maps,” *Biological Cybernetics*, vol. 1, no. 43, pp. 59–69, 1982.
- [73] ———, “The self-organizing map.” *Neurocomputing*, pp. 1–6, 1998.
- [74] B. Everitt, G. Dunn *et al.*, *Applied multivariate data analysis*. Wiley Online Library, 2001, vol. 2.
- [75] R. S. A. of Sciences, *Nobel Prize in Physics 2024*, 2024 (accessed October 17, 2024), <https://www.nobelprize.org/prizes/physics/2024/press-release/>.
- [76] A. K. John Hertz and R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Ed. Addison-Wesley, 1991.
- [77] J. A. Hertz, *Introduction to the theory of neural computation*. Crc Press, 2018.
- [78] M. Scilipoti, M. Fuster, and R. Ramele, “A strong inductive bias: Gzip for binary image classification,” *arXiv preprint arXiv:2401.07392*, 2024.
- [79] E. Oja, “Principal components, minor components, and linear neural networks,” *Neural networks*, vol. 5, no. 6, pp. 927–935, 1992.
- [80] T. D. Sanger, “Optimal unsupervised learning in a singlelayer linear feedforward neural network,” *Neural Networks*, vol. 2, no. 6, pp. 459–473, 1989.
- [81] P. Dayan and L. F. Abbott, *Theoretical neuroscience: computational and mathematical modeling of neural systems*. MIT press, 2005.
- [82] K. Doya, *Bayesian brain: Probabilistic approaches to neural coding*. MIT press, 2007.
- [83] C. S. Nam, A. Nijholt, and F. Lotte, *Brain–computer interfaces handbook: technological and theoretical advances*. CRC press, 2018.
- [84] E. Niedermeyer and F. L. da Silva, *Electroencephalography: basic principles, clinical applications, and related fields*. Lippincott Williams & Wilkins, 2005.
- [85] O. Sacks, *The man who mistook his wife for a hat and other clinical tales*. Simon And Schuster, 1985.
- [86] R. E. Bryant and D. R. O’Hallaron, *Computer systems: a programmer’s perspective*. Prentice Hall, 2011.
- [87] M. Krzisch, S. G. Temprana, L. A. Mongiat, J. Armida, V. Schmutz, M. A. Virtanen, J. Kocher-Braissant, R. Kraftsik, L. Vutskits, K.-K. Conzelmann *et al.*, “Pre-existing astrocytes form functional perisynaptic processes on neurons generated in the adult hippocampus,” *Brain Structure and Function*, vol. 220, pp. 2027–2042, 2015.

- [88] W. Elmenreich, “An introduction to sensor fusion,” *Vienna University of Technology, Austria*, vol. 502, pp. 1–28, 2002.
- [89] N. Chaffey, “Alberts, b., johnson, a., lewis, j., raff, m., roberts, k. and walter, p. molecular biology of the cell. 4th edn.” 2003.
- [90] E. M. Izhikevich, *Dynamical systems in neuroscience*. MIT press, 2007.
- [91] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952.
- [92] R. Brooks, “An inconvenient truth about ai: Ai won’t surpass human intelligence anytime soon,” *IEEE Spectrum*, 2021.
- [93] A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.
- [94] L. Acion, L. Alemany Alonso, E. Ferrante, E. L. Holm, V. Martinez, D. H. Milone, R. Rodriguez, G. Simari, and S. Uchitel, “Desmistificando la inteligencia artificial,” *Inteligencia Artificial, Una Mirada Interdisciplinaria X Encuentro Interacademico 2021*, vol. 1, no. 1, pp. 63–86, 2021.
- [95] M. Mitchell, “Why AI is harder than we think,” *CoRR*, vol. abs/2104.12871, 2021. [Online]. Available: <https://arxiv.org/abs/2104.12871>
- [96] P. Stone, R. Brooks, E. Brynjolfsson, R. Calo, O. Etzioni, G. Hager, J. Hirschberg, S. Kalyanakrishnan, E. Kamar, S. Kraus *et al.*, “Artificial intelligence and life in 2030: the one hundred year study on artificial intelligence,” *arXiv preprint arXiv:2211.06318*, 2022.
- [97] J. Degrave, F. Felici, J. Buchli, M. Neunert, B. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. de Las Casas *et al.*, “Magnetic control of tokamak plasmas through deep reinforcement learning,” *Nature*, vol. 602, no. 7897, pp. 414–419, 2022.
- [98] M. Simon, “How a plucky robot found the long-lost endurance shipwreck,” <https://www.wired.com/story/how-a-plucky-robot-found-the-long-lost-endurance-shipwreck>, 2022 (accessed Jul 18, 2024).
- [99] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirsycz *et al.*, “Discovering faster matrix multiplication algorithms with reinforcement learning,” *Nature*, vol. 610, no. 7930, pp. 47–53, 2022.
- [100] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, “Highly accurate protein structure prediction with alphafold,” *nature*, vol. 596, no. 7873, pp. 583–589, 2021.
- [101] R. Sutton, “The bitter lesson,” <http://www.incompleteideas.net/IncIdeas/BitterLesson.html>, 2019 (accessed May 29, 2024).
- [102] P. Domingos, “A few useful things to know about machine learning,” *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.
- [103] O. Boiman, E. Shechtman, and M. Irani, “In defense of nearest-neighbor based image classification,” *26th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2008, this paper is the NBNN method which is similar to what I am doing with Sift descriptors.

- [104] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.
- [105] M. Telgarsky, “Deep learning theory lecture notes,” <https://mjt.cs.illinois.edu/dlt/>, 2021, version: 2021-10-27 v0.0-e7150f2d (alpha).
- [106] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, “Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review,” *International Journal of Automation and Computing*, vol. 14, no. 5, pp. 503–519, 2017. [Online]. Available: <https://doi.org/10.1007/s11633-017-1054-2>
- [107] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85–117, 2015.
- [108] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.
- [109] S. Hooker, “The hardware lottery,” *Communications of the ACM*, vol. 64, no. 12, pp. 58–65, 2021.
- [110] D. Garisto, “How cutting-edge computer chips are speeding up the ai revolution,” *Nature*, vol. 630, no. 8017, pp. 544–546, 2024.
- [111] M. Pandey, M. Fernandez, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, “The transformational role of gpu computing and deep learning in drug discovery,” *Nature Machine Intelligence*, vol. 4, no. 3, pp. 211–221, 2022.
- [112] M. A. Kramer, “Nonlinear principal component analysis using autoassociative neural networks,” *AICHE journal*, vol. 37, no. 2, pp. 233–243, 1991.
- [113] D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *Foundations and Trends in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. [Online]. Available: <http://dx.doi.org/10.1561/2200000056>
- [114] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.
- [115] jhu.edu., *Variational Tutorial*, 2011 (accessed September 3, 2018), <http://www.cs.jhu.edu/~jason/tutorials/variational.html>.
- [116] C. W. Fox and S. J. Roberts, “A tutorial on variational bayesian inference,” *Artificial intelligence review*, vol. 38, no. 2, pp. 85–95, 2012.
- [117] M. E. Khan and G. Bouchard, “Variational em algorithms for correlated topic models,” Technical report, University of British Columbia, Tech. Rep., 2009.
- [118] C. Tang and R. R. Salakhutdinov, “Learning stochastic feedforward neural networks,” *Advances in Neural Information Processing Systems*, vol. 26, 2013.
- [119] E. W. Weisstein, *Stochastic Function*, 2023 (accessed November 17, 2023), <https://mathworld.wolfram.com/StochasticFunction.html>.
- [120] J. R. Anderson and C. Peterson, “A mean field theory learning algorithm for neural networks,” *Complex Systems*, vol. 1, pp. 995–1019, 1987.
- [121] M. J. Wainwright and M. I. Jordan, *Graphical models, exponential families, and variational inference*. Now Publishers Inc, 2008.

- [122] G. E. Hinton and D. Van Camp, “Keeping the neural networks simple by minimizing the description length of the weights,” in *Proceedings of the sixth annual conference on Computational learning theory*, 1993, pp. 5–13.
- [123] G. Gundersen., *Reparamatrization Trick*, 2021 (accessed October 2, 2021), <https://gregorygundersen.com/blog/2018/04/29/reparameterization/>.
- [124] B. Rocca, *Stack Exchange, Machine Learning*, 2021 (accessed October 2, 2021), <https://stats.stackexchange.com/questions/420974/backpropagation-on-variational-autoencoders>.
- [125] E. S. Piñeiro, R. Ramele, and J. Gambini, “Variational autoencoder as a data augmentation tool for confocal microscopy images,” in *2023 IEEE 36th International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE, 2023, pp. 882–885.
- [126] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, “Variational inference: A review for statisticians,” *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, Feb 2017. [Online]. Available: <http://dx.doi.org/10.1080/01621459.2017.1285773>
- [127] C. Doersch, “Tutorial on variational autoencoders,” 2021.
- [128] C. C. Aggarwal, *Neural Networks and Deep Learning*. Cham: Springer, 2018.
- [129] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” 2014.
- [130] J. Jordan, *Autoencoders*, 2022 (accessed Novemenber 1, 2022), <https://www.jeremyjordan.me/autoencoders/>.
- [131] G. W. Lindsay, “Convolutional neural networks as a model of the visual system: Past, present, and future,” *Journal of Cognitive Neuroscience*, pp. 1–15, Feb 2020. [Online]. Available: [http://dx.doi.org/10.1162/jocn\\_a\\_01544](http://dx.doi.org/10.1162/jocn_a_01544)
- [132] B. E. Ail, R. Ramele, J. Gambini, and J. M. Santos, “An intrinsically explainable method to decode p300 waveforms from eeg signal plots based on convolutional neural networks,” *Brain Sciences*, vol. 14, no. 8, p. 836, 2024.
- [133] K. Fukushima and N. Wake, “Handwritten alphanumeric character recognition by the neo-cognitron,” *IEEE transactions on Neural Networks*, vol. 2, no. 3, pp. 355–365, 1991.
- [134] 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.
- [135] C. C. Aggarwal *et al.*, *Neural networks and deep learning*. Springer, 2018.
- [136] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” mar 2016. [Online]. Available: <http://arxiv.org/abs/1603.07285>
- [137] E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, nov 2017. [Online]. Available: <http://arxiv.org/abs/1411.4038>
- [138] S. Jiang and V. M. Zavala, “Convolutional Neural Nets: Foundations, Computations, and New Applications,” jan 2021. [Online]. Available: <http://arxiv.org/abs/2101.04869>

- [139] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [140] D. Foster, *Generative deep learning*. O’Reilly Media, Inc., 2022.
- [141] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” *Advances in neural information processing systems*, vol. 27, 2014.
- [142] M. Kahng, N. Thorat, P. Chau, F. Viégas, and M. Wattenberg, *Gan Lab*, 2023 (accessed November 17, 2023), <https://poloclub.github.io/ganlab/>.
- [143] M. Klingermann, “Quasimondo,” <https://quasimondo.com/>, 2024 (accessed Jul 18, 2024).
- [144] H. T. Siegelmann and E. D. Sontag, “On the computational power of neural nets,” in *Proceedings of the fifth annual workshop on Computational learning theory*, 1992, pp. 440–449.
- [145] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [146] Y.-H. H. Tsai, S. Bai, M. Yamada, L.-P. Morency, and R. Salakhutdinov, “Transformer dissection: a unified understanding of transformer’s attention via the lens of kernel,” *arXiv preprint arXiv:1908.11775*, 2019.
- [147] R. Wright and W. LM, *Orienting of Attention*, 2008.
- [148] E. Fedorenko, S. T. Piantadosi, and E. A. Gibson, “Language is primarily a tool for communication rather than thought,” *Nature*, vol. 630, no. 8017, pp. 575–586, 2024.
- [149] A. Zhou, F. Tajwar, A. Robey, T. Knowles, G. J. Pappas, H. Hassani, and C. Finn, “Do deep networks transfer invariances across classes?” in *International Conference on Learning Representations*, 2021.
- [150] M. Brooks, “Inside the maths that drives ai,” *Nature*, vol. 631, no. 8019, pp. 244–246, 2024.
- [151] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, “Array programming with numpy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.
- [152] L. Wilkinson, “The grammar of graphics,” in *Handbook of computational statistics: Concepts and methods*. Springer, 2011, pp. 375–414.
- [153] N. Ketkar, J. Moolayil, N. Ketkar, and J. Moolayil, “Automatic differentiation in deep learning,” *Deep Learning with python: learn best practices of deep learning models with PyTorch*, pp. 133–145, 2021.



# Índice alfabético

## A

Agente .....	29
Alelo .....	45
Alta dimensionalidad .....	131
Aprendizaje Automático .....	91
Autoencoder .....	137

## B

Backpropagation .....	72
Backtracking .....	34
BFS .....	34
Búsqueda .....	31

## C

Coca Cola .....	53
Convolución .....	158
Cromosoma .....	44

## D

Datasets Masivos .....	134
Decomposición Espectral .....	138
Deep Learning .....	129
DFS .....	34

## F

Fenotipo .....	44
----------------	----

Fitness .....	45
Fitness Relativo .....	45
Funciones Estocásticas .....	144

## G

GA .....	43
GAN .....	171
Generación 0 .....	45
Genotipo .....	44
GPUs .....	134
Graph-Search .....	33

## H

Heurística .....	35
Hopfield .....	107

## I

Inferencia Variacional .....	145
Inteligencia .....	13
Inverse Sampling Theorem .....	153

## K

k-Fold Cross Validation .....	86
KL .....	152
Kohonen .....	103

**L**

Locus ..... 44

**M**

MLE ..... 152

MLP ..... 77

MLS ..... 152

Modelo Generativo ..... 171

Momentum ..... 59

**O**

Oja ..... 113

**P**

PCA ..... 99

Perceptrón Simple ..... 67

Profundidad ..... 131

Propriocepción ..... 119

**R**

Regularización ..... 152

**S**

Sanger ..... 116

Simplex ..... 55

Softmax ..... 164

SVD ..... 138

**T**

Tied Weights ..... 154

Tips ..... 185

Transformers ..... 177

Travel Salesman Problem ..... 38

Tío y Tía ..... 129

**U**

UCS ..... 34