

# Sistemas Inteligentes - Appunti

Francesco Lorenzoni

Febrero 2025



# Contents

<b>I Introduction to SIN</b>	<b>5</b>
<b>1 Introduction</b>	<b>9</b>
1.1 Informaciones generales . . . . .	9
1.2 Inteligencia Artificial . . . . .	10
1.2.1 Aplicaciones de la IA . . . . .	10
<b>2 Planificación Inteligente</b>	<b>11</b>
2.0.0.1 Modelo “planificación clásica” . . . . .	12
2.1 Planificación Jerarquica . . . . .	13
<b>3 Pyhop</b>	<b>15</b>
<b>4 Optimización Inteligente</b>	<b>17</b>
4.1 Scheduling . . . . .	17
4.1.1 JSP . . . . .	17
4.1.2 CSP - Constraint Satisfaction Problem . . . . .	17
4.1.3 Algoritmo Genéticos . . . . .	18
4.1.3.1 Resumen . . . . .	18
<b>5 Agente Inteligentes</b>	<b>21</b>
5.1 Jason . . . . .	22
5.1.1 Creencias . . . . .	22
5.1.2 Objetivos . . . . .	23
5.1.3 Planes . . . . .	23
5.1.3.1 Cuerpo de un plan . . . . .	23
5.1.3.2 Fallo de un Plan . . . . .	24
5.1.4 Ejemplos . . . . .	24
<b>6 Ontologías y Comunicación</b>	<b>27</b>
6.1 Ontologías y OWL . . . . .	27
6.1.1 Lenguajes para Representación de Ontologías . . . . .	27
6.1.1.1 RDF (Resource Definition Framework) . . . . .	27
6.1.1.2 RDFS (RDF Schema) . . . . .	27
6.1.1.3 OWL (Web Ontology Language) . . . . .	27
6.1.2 Protégé . . . . .	28
6.2 Comunicación entre Agentes . . . . .	28
6.2.1 Teoría de Actos de Habla . . . . .	28
6.2.2 Lenguajes de Comunicación entre Agentes . . . . .	28
6.2.2.1 KQML (Knowledge Query Management Language) . . . . .	28
6.2.2.2 FIPA ACL . . . . .	28
6.2.3 Comunicación en JASON . . . . .	28
6.3 Sistemas Multiagente . . . . .	28
6.3.1 Plataformas de Agentes FIPA . . . . .	28
<b>II Machine Learning</b>	<b>29</b>
<b>7 N-Grams y FSA</b>	<b>31</b>
7.1 N-Grams . . . . .	31
7.2 FSA - Finite State Automata . . . . .	33
7.2.1 Applications of PFA . . . . .	34

7.2.2	Relation between N-Grams and PFA . . . . .	34
7.3	Relación entre N-Grams y PFA . . . . .	35
<b>8</b>	<b>Deep Learning</b>	<b>37</b>
8.1	Introducción . . . . .	37
8.2	Linear Discriminant Function and Perceptron . . . . .	39
8.2.1	Backpropagation . . . . .	40
8.2.2	Resumen del algoritmo de Backpropagation . . . . .	41
<b>9</b>	<b>Deep Learning for Text Classification</b>	<b>43</b>
9.1	Word Embeddings . . . . .	43
9.2	Word Embeddings: Word2Vec . . . . .	43
9.3	RNN: Recurrent Neural Networks . . . . .	43
9.3.1	Encoders and Decoders . . . . .	43
<b>10</b>	<b>Deep Learning for Image Classification</b>	<b>45</b>
10.1	Convolutional Neural Networks . . . . .	45
10.1.1	Pooling . . . . .	46
10.1.1.1	Convolutional Blocks . . . . .	46
10.2	CNN Architectures . . . . .	47
10.2.1	VGGNet . . . . .	48
10.2.2	ResNet . . . . .	48
10.2.3	Detection Architectures . . . . .	48
10.2.3.1	R-CNN . . . . .	48
10.2.3.2	YOLO . . . . .	49
10.2.4	Segmentation Architectures . . . . .	49
10.2.4.1	FCN . . . . .	49
10.2.4.2	U-Net . . . . .	49
10.2.5	Transformer Architectures . . . . .	49
10.2.5.1	Vision Transformer (ViT) . . . . .	49
10.2.5.2	DeTr (Detection Transformer) . . . . .	50
10.2.5.3	SeTr (Segmentation Transformer) . . . . .	50
10.3	Transfer Learning . . . . .	50

# **Part I**

# **Introduction to SIN**



---

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Informaciones generales . . . . .	9
1.2	Inteligencia Artificial . . . . .	10
1.2.1	Aplicaciones de la IA . . . . .	10
<b>2</b>	<b>Planificación Inteligente</b>	<b>11</b>
2.1	Planificación Jerarquica . . . . .	13
<b>3</b>	<b>Pyhop</b>	<b>15</b>
<b>4</b>	<b>Optimización Inteligente</b>	<b>17</b>
4.1	Scheduling . . . . .	17
4.1.1	JSP . . . . .	17
4.1.2	CSP - Constraint Satisfaction Problem . . . . .	17
4.1.3	Algoritmo Genéticos . . . . .	18
<b>5</b>	<b>Agente Inteligentes</b>	<b>21</b>
5.1	Jason . . . . .	22
5.1.1	Creencias . . . . .	22
5.1.2	Objetivos . . . . .	23
5.1.3	Planes . . . . .	23
5.1.4	Ejemplos . . . . .	24
<b>6</b>	<b>Ontologías y Comunicación</b>	<b>27</b>
6.1	Ontologías y OWL . . . . .	27
6.1.1	Lenguajes para Representación de Ontologías . . . . .	27
6.1.2	Protégé . . . . .	28
6.2	Comunicación entre Agentes . . . . .	28
6.2.1	Teoría de Actos de Habla . . . . .	28
6.2.2	Lenguajes de Comunicación entre Agentes . . . . .	28
6.2.3	Comunicación en JASON . . . . .	28
6.3	Sistemas Multiagente . . . . .	28
6.3.1	Plataformas de Agentes FIPA . . . . .	28

---



# Chapter 1

## Introduction

### 1.1 Informaciones generales

En los Viernes las clases son online y sincronas.

La asignatura se divide en doce partes:

#### 1. Parte 1

T1 Introducción

T2 Planificación inteligente

Algunos ejemplos son la planificación de procesos/planes de producción, o la planificación de rutas de reparto y logística; en general, aplicaciones incluyen sistemas de ayuda a la toma de decisiones y recomendación.

T3 Optimización inteligente Optimización de la logística empresarial (horarios, recursos, personal, turnos de trabajo,...)

T4 Agentes inteligentes Sistemas multiagentes que tienen que negociar, coordinarse y cooperar para un objetivo común. Pero algo videojuegos donde existe una competición o cooperación entre agentes y se requiere una búsqueda de alternativas.

#### 2. Parte 2

T5 Modelado de lenguaje y autómatas finitos

T6 Aprendizaje profundo

T7 Aprendizaje profundo para clasificación de texto

T8 Aprendizaje profundo para clasificación de imágenes

Profesor	T1	T2	T3	T4	T5	T6	T7	T8
Jaume Jordán								
Carlos Carrascosa								
Carlos Martínez (resp.)								

Figure 1.1: Profesores

La evaluación se efectuará mediante:

- ◊ Dos tests (**20 de junio a las 15:30**) sobre contenidos teóricos y prácticos de la parte 1 (15%) y parte 2 (15%), totalizando 30%.
- ◊ Trabajos académicos (70%):
  - Parte 1 (35%): tema 2 (15%) y tema 4 (20%).
  - Parte 2 (35%): tema 5 (10%), tema 7 (10%) y tema 8 (15%).

## 1.2 Inteligencia Artificial

# The AI Universe

IA no es solo IA generativa. IA es un campo muy amplio que incluye otras cosas. Inicialmente se asociaba con el Machine Learning, pero ahora se ha ampliado a otras cosas.

La Inteligencia Artificial es un campo interdisciplinario que implica máquinas capaces de imitar determinadas funcionalidades de la inteligencia humana, incluidas características como la percepción, el aprendizaje, el razonamiento, la resolución de problemas, la interacción lingüística e incluso la producción de trabajos creativos.

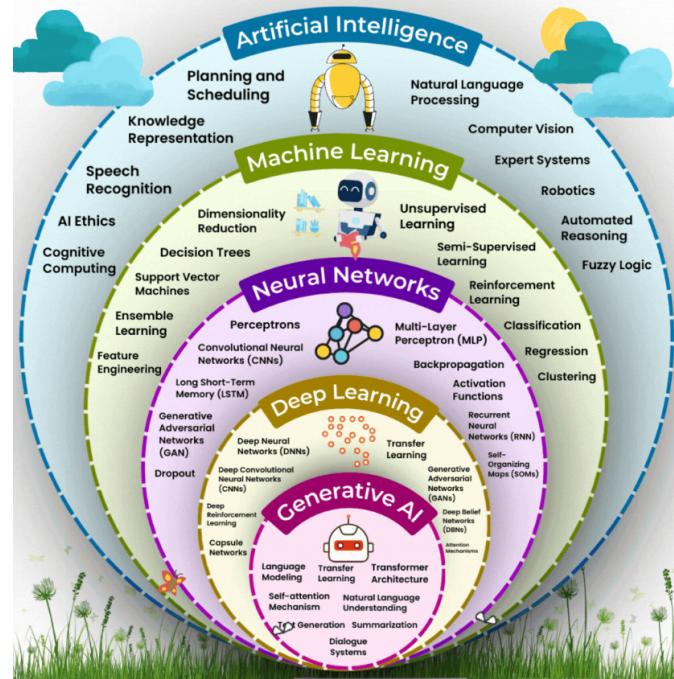


Figure 1.2: AI esquema. Según el profesor esto esquema no es completo, pero es ilustrativo.

Modelos de IA:

- ◊ Aprendizaje supervisado
- ◊ Aprendizaje no supervisado
- ◊ Aprendizaje por refuerzo

### 1.2.1 Aplicaciones de la IA

La Inteligencia artificial puede ser útil para estas cosas: Áreas y aplicaciones

- ◊ Recuperación inteligente de la información
  - Recuperación de información no explícitamente representada
  - Bases de datos deductivas. Procesos inferenciales. Sentido común. Interfaz natural
- ◊ Ingeniería del conocimiento (SBC y Sistemas Expertos)
  - Problemas NP: restricción de dominios, heurísticas y meta-heurísticas
  - Planificación. Scheduling. Optimización. Sistemas de Ayuda a la Toma de Decisiones
- ◊ Problemas (de búsqueda) combinatorios y de planificación
  - Este es el problema más importante de la IA.
    - Problemas NP: restricción de dominios, heurísticas y meta-heurísticas
    - Planificación. Scheduling. Optimización. Sistemas de Ayuda a la Toma de Decisiones
- Hay también otras aplicaciones
- ◊ Sistemas Multiagente y distribuidos que interactúan (cooperación)
- ◊ Robótica
- ◊ Percepción
- ◊ Procesamiento de lenguaje natural

## Chapter 2

# Planificación Inteligente

Podemos ver la Planificación Inteligente desde varios puntos de vista:

- ◊ **Deducción**

A partir de algunos hechos ciertos, conocer qué otros hechos también son ciertos (propagación)

- ◊ **Aprendizaje**

“Mejorar la conducta a partir de la experiencia”

- ◊ **Razonamiento** sobre acciones

A partir de algunos hechos ciertos (estado inicial), decidir qué acciones se deben ejecutar para alcanzar ciertos objetivos (es decir, que otros hechos sean también ciertos) en base a relaciones de causa-efecto.

El ser humano es inteligente porque es capaz de razonar acerca de las consecuencias de sus actos. Al fin y al cabo estamos planificando constantemente aunque no nos demos cuenta.

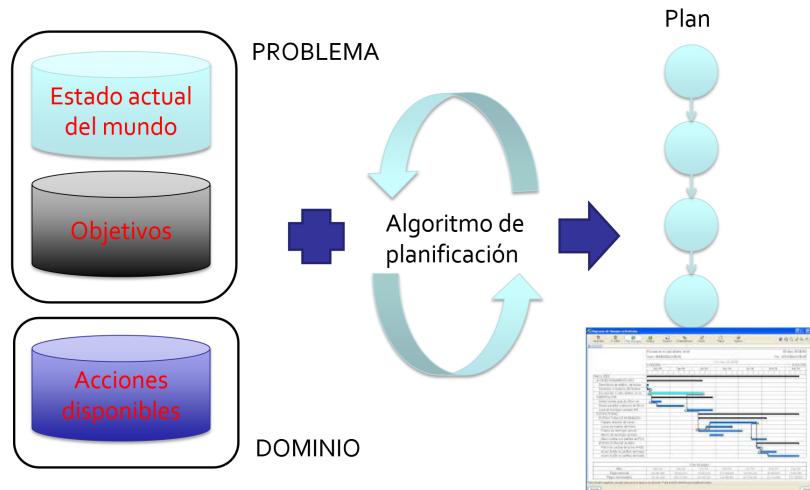


Figure 2.1: Planificación esquematizada

En Fig. 2.1 podemos ver un esquema de planificación. En el estado actual del mundo corresponde al “**dominio**”, y el objetivo es llegar a un estado deseado en el mundo.

## Estado inicial

Información estática	Información dinámica
<pre> distance[home][airport_city1]=15 distance[home][center_city1]=5 distance[home][connection_c1_c2]=2 ..... vehicles(taxi)={taxi1} vehicles(plane)={plane1} ... location(airport)={airport_city1, airport_city2} location(to_stay)={home, hotel_city2, ...} .... city(home)=city1 city(airport_city1)=city1 city(airport_city2)=city2 ..... </pre>	<pre> at(me)=home at(taxi1)=center_city1 at(plane1)=airport_city1 at(car1)=home cash(me)=20 ;; dinero que tengo owe(me)=0 ;; dinero que debo </pre>

### Objetivos

at(me)=hotel\_city2

Figure 2.2: Suponen que tengo que viajar desde una ciudad a otra. Esta puede ser una formalización de la planificación.

Un **plan** es un conjunto de acciones que se pueden ejecutar en el mundo que representan los posibles cambios (transiciones) en el entorno. Estas, deben ser independientes de cualquier problema y se representan con descripciones de condiciones y efectos:  $a = \langle \text{Conditions}(a), \text{Effects}(a) \rangle$

- ◊ **Conditions(a)**: condiciones que se deben satisfacer en un estado para aplicar dicha acción
- ◊ **Effects(a)**: modificaciones del estado actual una vez ejecutada la acción

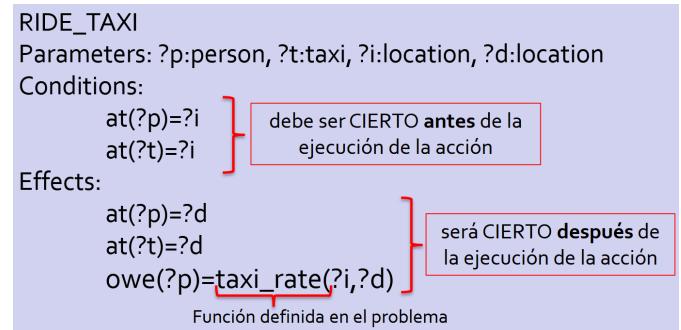


Figure 2.3: Ejemplo para coger un taxi

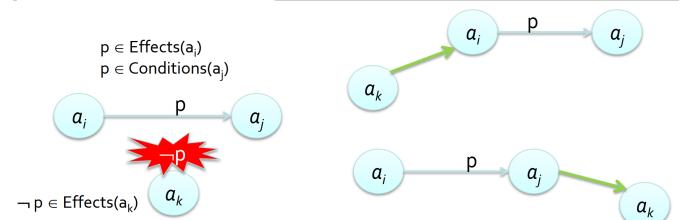


Figure 2.4: Plan

**Definition 2.1 (Plan)** Un plan  $P$  es una secuencia de acciones que transforma el estado inicial ( $I$ ) en un estado que satisface los objetivos ( $G$ )

Un plan puede estar parcial o totalmente ordenado. Un enlace causal  $(a_i, p, a_j)$  determina que la acción  $a_i$  resuelve la condición  $p$  de la acción  $a_j$ .

### 2.0.0.1 Modelo “planificación clásica”

- ◊ Determinista
- ◊ Estático (solo cambia por las acciones)
- ◊ Completamente observable
- ◊ Razonamiento temporal y numérico no contemplado

Es decir, el resultado de una acción aplicada a un estado completamente conocido se puede predecir correctamente, ya que no hay influencias externas que afecten al entorno. Para resolver este problema se utilizan algoritmos de búsqueda.

Este modelo pero es muy limitado, y sus asunciones necesitan ser relajadas para que sea más realista y aplicable a problemas reales.

En primer lugar, las acciones tienen distintas **duraciones**, usan recursos y tienen **coste**; además, El mundo *no* es totalmente conocido (existe incertidumbre) y es **dinámico** (cambia debido a eventos exógenos).

## 2.1 Planificación Jerárquica

La idea principal de HTN Hierarchical Task Network planning es proceder de una forma más parecida a como los humanos resolvemos los problemas complejos: Primero los aspectos generales (tareas) y después los detalles.

### Método abstracto y descomposición

Para viajar a un destino, se puede elegir entre viajar en avión, en coche o en tren:

Se definen métodos abstractos que se deben descomponer para resolver el problema. Métodos son los triángulos invertidos en la figura.

Cada método incluye estos componentes:

- ◊ Nombre
- ◊ Parámetros
- ◊ Precondiciones
- ◊ Red de tareas

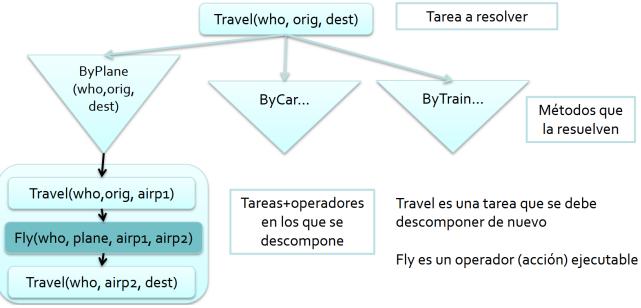


Figure 2.5: Método abstracto y decomposición

### Fly:

Parámetros: *who: Person*  
*plane: Plane*  
*from: Airport*  
*to: Airport*

*Fly(who, plane, airp1, airp2)*

Condiciones:

*at[who] == at[plane] and*  
*at[plane] == from*

Efectos:

*at[who]=to*  
*at[plane]=to*

Métodos son tareas a descomponer, y los operadores (como Fly) representan las acciones que se pueden llevar a cabo dentro del mundo para poder alcanzar los distintos objetivos (acciones ejecutables que producen cambios en el estado del mundo).

- ◊ Nombre
- ◊ Parámetros
- ◊ Condiciones
- ◊ Efectos

Figure 2.6: Acción Fly



# Chapter 3

## Pyhop

Para resolver un problema tenemos tres fases

1. Modelato
2. Planificación
3. Ejecución

El problema se define como una declaración de un nuevo estado del mundo (inicial), y una declaración de un objetivo. El **dominio**, son las acciones disponibles, que se estructuran en dos niveles: tareas/operadores y métodos.

- ◊ Información estática: no cambia durante la ejecución del problema y que sirve de soporte. Por ejemplo: distancias entre ciudades.
- ◊ Información dinámica: representa el estado actual del mundo. Por ejemplo: cantidad de dinero disponible por el agente en un momento determinado

Representan las **acciones ejecutables** en el dominio

```
def nombre_operador(state, otros_parametros):
    <extracción valores estado>
    if [condición1 and condición2 and ... ]:  Condiciones
        state.variable = nuevo_valor
        ...
        state.variable = nuevo_valor
        return state
    else: return False
```

Se devuelve el nuevo estado tras aplicarse el operador  
o False si el operador no es aplicable

```
pyhop.declare_operators(nombre_todos_operadores)
```

Figure 3.1: Como definir un operador en Pyhop

Los elementos básicos de pyhop son:

- ◊ **Estado**: es un diccionario que representa el estado del mundo.
  - ◊ **Tareas**: son tareas que identifican un objetivo (estado final) que se quiere alcanzar. Las tareas primitivas son aquellas que se pueden ejecutar directamente y resolver con operadores, y las tareas compuestas son aquellas que se descomponen en otras tareas o acciones.
  - ◊ **Operadores**: son funciones que toman un estado y devuelven un nuevo estado. No se pueden descomponer.
  - ◊ **Métodos**: representan una forma distinta de resolver una tarea. Están formados por un conjunto de tareas y operadores (acciones) que hay que ejecutar y/o alcanzar en un orden establecido para lograr resolver la tarea de nivel superior.
- Estas tareas y operadores se organizan de forma secuencial



# Chapter 4

## Optimización Inteligente

Planificación y Scheduling representa un área de gran relevancia en Inteligencia Artificial. Muchos problemas reales se modelan como problemas de P&S.

La **planificación** es un proceso de deliberación que escoge y organiza acciones anticipando sus resultados o consecuencias.

Lo **scheduling** es un proceso de asignación de recursos a tareas sobre el tiempo para optimizar uno o más objetivos.

PLANNING	SCHEDULING
La tarea de planificación determina <b>QUÉ</b> acciones son necesarias llevar a cabo para alcanzar un estado objetivo	La tarea de scheduling se centra en <b>CUÁNDO/CÓMO</b> llevar a cabo las acciones para optimizar el problema

Figure 4.1: P&S

### 4.1 Scheduling

Un problema de scheduling consiste en organizar en el tiempo un conjunto de actividades que compiten por el uso limitado de recursos

#### 4.1.1 JSP

JSP stands for Job-Shop Scheduling, y es un paradigma de los problemas de scheduling es muy simple de enunciar y muy difícil de resolver.

- ◊  $n$  trabajos cada uno compuesto por un conjunto ordenado de tareas
- ◊  $m$  maquinas donde se procesan las tareas
- ◊ Cada tarea debe ser procesada en una única máquina durante un tiempo determinado y en un orden prefijado;
- ◊ El objetivo es minimizar *makespan*: instante de finalización de la última tarea

- ◊ **Datos** -  $p_{ij}$  es el tiempo de proceso de la tarea  $i$  en el trabajo  $j$ ;
- ◊ **Variables** -  $st_{ij}$  es el tiempo de inicio de tarea  $i$  en el trabajo  $j$ ;
- ◊ **Restricciones** -
  - Secuencial -  $st_{ij} + p_{ij} \leq st_{i(j+1)}$
  - Capacidad -  $st_{ij} + p_{ij} \leq st_{kl} \vee st_{kl} + p_{kl} \leq st_{ij}$
  - Sin interrupción -  $C_{ij} = st_{ij} + p_{ij}$
- ◊ **Objetivo** - Construir una ordenación de las tareas en el tiempo de manera que se satisfagan todas las restricciones sobre cada máquina.
  - Minimize the makespan:  $C_{max} = \max C_{ij}$
  - Minimize total flow time:  $C_{sum} = \sum C_i$
  - Minimize makespan + energy

#### 4.1.2 CSP - Constraint Satisfaction Problem

Muchos problemas pueden ser expresados mediante:

- ◊ Un conjunto de variables.

- ◊ Un dominio de interpretación (valores) para las variables.
- ◊ Un conjunto de restricciones entre las variables.
- ◊ La solución al problema es una asignación válida de valores a las variables.

Formalization omitted here

En general, los problemas de optimización son particularmente difíciles de resolver. En general, basta con una solución razonablemente buena (factible, optimizada, pero no la óptima) en un tiempo razonable, y si puede obtener con métodos aproximados (como alternativa a los métodos exactos), como la **Búsqueda Heurística / Metaheurística**: La solución es obtenida (generada / mejorada), mediante un proceso de búsqueda en un amplio espacio de estados/-soluciones. Hay tambien **técnicas inferenciales**, que no se sostituyen por la búsqueda, sino que la complementan. Estas técnicas son utilizadas para reducir el espacio de búsqueda, y se basan en la **lógica** y en la **teoría de conjuntos**. Conociendo las consecuencias de una acción, podemos reducir el espacio de búsqueda, y por lo tanto, la complejidad del problema.

- ◊ Búsqueda Local (Local Search) - Se parte de una solución inicial y se busca una solución mejor en el vecindario de la solución actual.
- ◊ Búsqueda Sistemática (Systematic Search) - Se parte de una solución inicial y se busca una solución mejor en todo el espacio de soluciones.
- ◊ Métodos Constructivos - Construyen una solución paso a paso, añadiendo o eliminando elementos de la solución.
- ◊ Métodos de Mejora - Parten de una solución inicial y la mejoran iterativamente mediante movimientos locales.
- ◊ Métodos Evolutivos - Combinan soluciones previas

### 4.1.3 Algoritmo Genéticos

Los **algoritmos genéticos** son un tipo de algoritmo evolutivo que se utiliza para encontrar soluciones aproximadas a problemas de optimización y búsqueda. Estos algoritmos reflejan el proceso de selección natural en la evolución de las especies, y se basan en la idea de que las soluciones a un problema pueden ser representadas como individuos en una población, y que la evolución de la población puede conducir a la generación de soluciones cada vez mejores.

La operación fundamental es la **cruce**, que toma n padres (típicamente 2) y generar m hijos (típicamente 2); consiste en heredar material genético de los padres. La asunción es que si los padres tienen buen material genético, la mezcla del material genético de ambos también será buena.

Entonces, el efecto de la cruza es *explorar* el espacio de búsqueda con **material prometedor**.

La operación de **mutación** es una operación que se aplica a un individuo y cambia aleatoriamente uno o más genes de ese individuo. Corresponde a una búsqueda local, porque sólo se cambia un gen a la vez, y entonces el hijo será cerca de su padre.

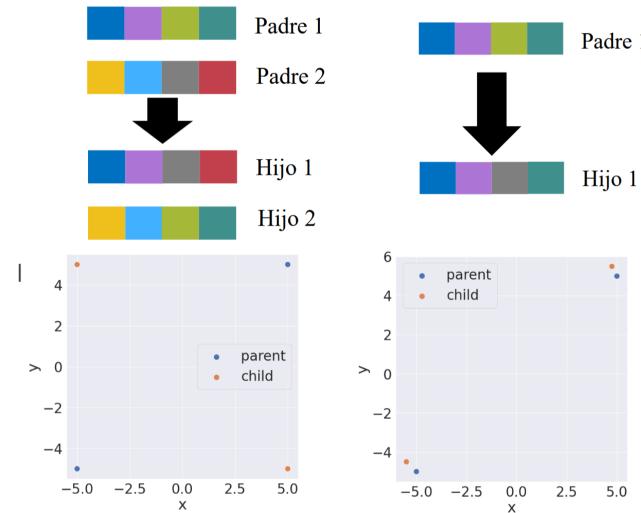


Figure 4.2: Cruce y mutación

TODO selección des padres TODO criterio de parada

#### 4.1.3.1 Resumen

1. Una **representación** adecuada de las *soluciones* del problema (individuos). Cromosomas, genes, genotipo, fenotipo.
2. Una forma de crear una **población de soluciones inicial** (individuos iniciales). A menudo, aleatoria.
3. Una **función de evaluación** capaz de medir la adecuación de cualquier solución (individuo) al problema. Fitness.
4. Un conjunto de **operadores evolutivos** para combinar las soluciones existentes con el objetivo de obtener nuevas soluciones (nuevos individuos): *selección, cruce, mutación* y *reemplazo* de individuos. Guían el proceso de la búsqueda.
5. Conjunto de **parámetros de entrada**: tamaño de la población, número de iteraciones (generaciones), probabilidades de selección, etc.

Podemos resumir el proceso de un algoritmo genético en los siguientes pasos:

Término GA	Explicación	Término Optimización	Explicación equivalencia
Medio o entorno	Lugar en el que se desempeñan los individuos	Problema	El individuo se desenvuelve en un entorno, que determina cómo puede actuar. En ese caso, este marco es el problema.
Individuo	Representa a un individuo de la población que compite con otros	Solución	Nuestros individuos son las soluciones al problema, aquello que queremos gradualmente evolucionar hacia mejores individuos o soluciones.
Cromosoma o genotipo	Es la forma en la que se representa internamente un individuo	Codificación de la solución	Representa cómo representamos internamente la solución (pueden haber diferentes).
Gen	Es una de las partes atómicas del cromosoma	Variable de decisión	Representa una decisión atómica en el problema.
Alelo	Son los valores que puede tomar un gen	Dominio de la variable	Si un gen es equivalente a una variable de decisión, el alelo representa su dominio.
Fenotipo	Representan la representación externa del individuo o solución, que es lo que al final es evaluado en el medio. A veces puede ser equivalente al genotipo.	-	-
Función de fitness	Determina el desempeño del fenotipo de un individuo en el entorno	Función objetivo	Equivalente, se usa simplemente diferente terminología.

Table 4.1: Equivalencia entre términos de Algoritmos Genéticos y Optimización

1. **Codificación** de Soluciones/instancias.
2. **Población Inicial:**
  - i. Generar población aleatoria de n individuos:  $x$  (posibles soluciones del problema)
  - ii. Evaluar la aptitud o fitness  $f(x)$  de cada individuo.
3. **Ciclo-Generacional:**
  - i. **Selección** Seleccionar dos padres de la población de acuerdo a su aptitud: Probabilidad de cruce (pc).
  - ii. **Cruce** Combinar los genes de los dos padres para obtener descendientes.
  - iii. **Mutación** Con una probabilidad de mutación (pmut ), mutar cada gen de los cromosomas hijo.
  - iv. **Reemplazo** Añadir nuevos hijos y determinar nueva población.
4. Verificar condición de parada:
  - i. **Parada:** proporcionar como solución el individuo con mejor valor de aptitud  $f(x)$ .
  - ii. **Ciclo:** Ir al paso 2

# Chapter 5

## Agente Inteligentes

Agentes autónomos trabajan para resolver problemas en un entorno dinámico y complejo. Un agente es un sistema computacional que opera en un entorno y es capaz de percibirlo y actuar sobre él. Un agente inteligente es aquel que actúa de manera racional, es decir, que actúa para alcanzar sus objetivos, basándose en la información que percibe y en su conocimiento del entorno. Más formalmente,

**Definition 5.1 (Wooldridge)** *Un agente es un sistema informático capaz de actuar autónomamente en algún entorno con el fin de alcanzar los objetivos que se le han delegado.*

**Definition 5.2 (Wooldridge updated)** *Cualquier proceso computacional dirigido por el objetivo capaz de interactuar con su entorno de forma **flexible** (Reactivo, Proactivo, Social) y **robusta***

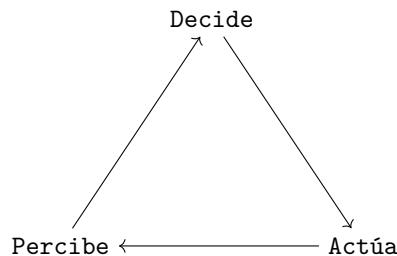


Figure 5.1: Agente bucle

Un agente reactivo reacciona a los cambios del entorno. No tiene que ser el más rápido, pero la reacción debe ser útil. Un agente proactivo actúa para alcanzar sus objetivos, planifica y actúa en consecuencia. Un agente social interactúa con otros agentes.

En entornos complejos, un agente no tiene control completo sobre su entorno, sólo tiene un control parcial. Control parcial significa que el agente puede influir sobre el entorno con sus acciones. Una acción ejecutada por un agente puede fallar o tener el efecto deseado. En conclusión, los entornos son no deterministas y los agentes deben estar preparados para posibles fallos.

### ◊ Accesible vs inaccesible.

Un entorno accesible es aquel en el que el agente puede obtener información completa, exacta y actualizada del estado del entorno.

### ◊ Determinista vs no determinista.

Un entorno determinista es aquel en el que cualquier acción tiene un único efecto garantizado, no hay incertidumbre sobre el estado resultante de la ejecución de una acción

El mundo físico puede a todos los efectos ser considerado como no determinista.

Los entornos no deterministas presentan grandes problemas para el diseñador de agentes.

### ◊ Episódico vs no episódico

En un entorno episódico el desempeño/actuación de un agente depende de un número discreto de episodios, no existiendo enlaces (relación) entre el desempeño de un agente en escenarios distintos. Los entornos episódicos son, desde el punto de vista del desarrollador de agentes, más sencillos porque el agente puede decidir qué acción ejecutar basándose únicamente en el episodio actual, no necesita razonar sobre las interacciones entre el episodio actual y los episodios futuros.

### ◊ Estático vs dinámico

Un entorno estático es aquel en el que se puede asumir que no se producen cambios excepto los provocados por la ejecución de acciones del agente. Un entorno dinámico es aquel que tiene otros procesos que operan en él, y que por lo tanto se producen cambios que están fuera del control del agente.

◊ **Discreto vs continuo**

Un entorno es discreto si en él hay un número fijo y finito de acciones y percepciones. El juego del ajedrez es un ejemplo de entorno discreto, y la conducción de un taxi un ejemplo de entorno continuo (AIMA, Russell and Norvig).

Asumimos que el entorno puede estar en uno cualquiera de los estados de un conjunto finito de estados instantáneos discreto ( $E$ ):

$$E = s_1, s_2, \dots \quad (5.1)$$

- Se asume que los agentes tienen disponible un repertorio (conjunto finito) de posibles acciones que transforman el estado del entorno :

$$Ac = \alpha_1, \alpha_2, \dots \quad (5.2)$$

$$Acción : E \rightarrow A \quad (5.3)$$

$$Percibir : E \rightarrow P \quad (5.4)$$

$$Decisión : P^* \rightarrow A \quad (5.5)$$

$$Selección : I \rightarrow Ac \quad (5.6)$$

$$actualizar\_estados : I \times P \rightarrow I \quad (5.7)$$

$$actualizar\_objetivos : G \times P \rightarrow G \quad (5.8)$$

$$Acción (update) : I \times G \rightarrow Ac \quad (5.9)$$

La acción dependerá del estado interno y de los objetivos que el agente querrá alcanzar

- ◊  $P$  es un conjunto (no vacío) de percepciones (entradas perceptivas), que relaciona estados del entorno con percepciones
- ◊  $I$  es el conjunto de todos los estados internos del agente
- ◊  $G$  es el conjunto de todos los objetivos del agente

Si puede también añadir una función de *utilidad* (o *recompensa*) que asigne un valor a cada estado del entorno, entonces el agente puede ser considerado como un **agente de utilidad**.

En este caso, La tarea del agente es alcanzar estados (seleccionar acciones) que maximicen la utilidad. La función de utilidad asocia un número real a cada estado del entorno.

$$u : E \rightarrow \mathbb{R} \quad (5.10)$$

Alternativamente, si puede asignar una utilidad a las ejecuciones del agente, en lugar de a los estados del entorno:

$$u : \mathcal{R} \rightarrow \mathbb{R} \quad (5.11)$$

## 5.1 Jason

Jason es un lenguaje de programación basado en agentes, que permite la programación de agentes inteligentes. Jason es un lenguaje de alto nivel, basado en la lógica, que permite la programación de agentes inteligentes, y es una extensión de AgentSpeak.

Los elementos fundamentales del Lenguaje son:

- ◊ **Creencias** (Beliefs)  
Colección de literales que representan el conocimiento del agente sobre el mundo.
- ◊ **Objetivos** (Goals)  
Representan los objetivos que el agente quiere alcanzar.
- ◊ **Planes** (Plans, Intentions)  
Colección de reglas que definen cómo el agente puede alcanzar sus objetivos.

### 5.1.1 Creencias

Cada agente tiene una base de **creencias**, colección de literales representados como *predicados*.

- ◊ `tall(jhon).`
- ◊ `likes(jhon, music)`

**Anotaciones** son detalles asociados a una creencia.

`busy(jhon) [expires(autum)]` Las anotaciones aportan “elegancia” al lenguaje y facilitan el manejo de la base de creencias.

Existen anotaciones que tienen un significado especial para el interprete. En particular la anotacion `source`.

Hay tres tipos de fuentes de información para los agentes

- ◊ *Informacion perceptual*: aquella que percibe del entorno
  - `source(percept)`
  - `(colour(box1,blue) [source(percept)])`
- ◊ *Comunicacion*: aquella que proviene de otro agente del sistema
  - `source(id agente)`
  - `(colour(box1,blue) [source(bob)])`
- ◊ *Notas mentales*: creencias que provienen del propio agente
  - `source(self)`
  - `(colour(box1,blue) [source(Self)])`

### 5.1.2 Objetivos

Existen dos tipos de objetivos en Jason:

1. **Achievement goals**(operador !): Expresan un estado del mundo que el agente desea conseguir.
  - ◊ `own(house)`
2. **Test goals**(operador ?): Usados normalmente para recuperar información de la base de creencias.
  - ◊ `bank_balance(BB)`

### 5.1.3 Planes

```
+trigger : context <- body.
```

Un plan tiene tres partes

- ◊ **Triggering event**: Cambios en las creencias o objetivos del agente
- ◊ **Context**: Determinan si un plan es aplicable. Literales que deben ser una consecuencia lógica de la base de creencias para que el plan se instancie.
- ◊ **Body**: Sucesión de acciones que comporta el plan. Es una secuencia de acciones. Pueden existir nuevos subobjetivos.

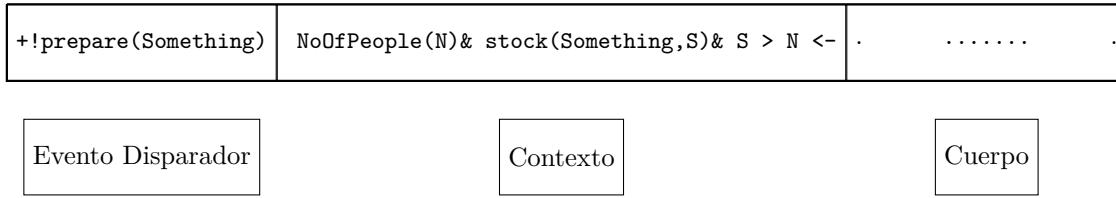


Figure 5.2: Plan schema

#### 5.1.3.1 Cuerpo de un plan

El cuerpo del plan es secuencia de instrucciones separadas por “;”. Estas instrucciones pueden ser:

- ◊ **Actions**: Acciones externas que se denotan por un predicado. Proporcionan un “feedback”.
 

```
+rotate(leftarm, 45)
```
- ◊ **Achievement goals**: subobjetivos que deben ser alcanzados para que el plan continúe su ejecución (si en lugar de emplear “!” se emplea “!!!”, el plan no suspenderá su ejecución).
 

```
!!at(home); call(john) en lugar de !at(home); call(john).
```
- ◊ **Test goals**: Para recuperar información de la BB o verificar si el agente cree algo.
 

```
?coords(Tarjet,X,Y).
```
- ◊ **Mental Notes**: Anotación `source(self)`. Sirven para añadir, modificar o eliminar nuevas creencias.
 

```
+currenttargets(NumTargets); [Se anade]
-+currenttargets(NumTargets); [Se modifica]
```
- ◊ **InternalActions**: Son acciones que no modifican el entorno. Se diferencian de acciones del entorno por el carácter “.”.
 

```
.print(...); .send(...);
```
- ◊ **Expressions**: Su sintaxis es semejante a la de Prolog.
 

```
X >= Y*2
```
- ◊ **Plan Labels**: Los planes pueden estar etiquetados
 

```
@labelte: ctxt<-body.
```

### 5.1.3.2 Fallo de un Plan

- ◊ Un plan puede fallar por tres causas principales:
  - Falta de planes relevantes o aplicables para un “achievement goal”
  - Fallo de un “test goal”
  - Fallo de una acción
- ◊ Cuando un plan falla se genera un evento `-!g` (goal deletion event) si se generó por la adición de un “achievement goal” o “test goal”.
- ◊ El plan que se dispara por el fallo se añade a la pila de intenciones del plan que ha fallado.

### 5.1.4 Ejemplos

```

/* Creencias iniciales */
inicio.
/* Planes */
+inicio <- .print("Hola Mundo"). //plan que se dispara al inicio

/* Creencias iniciales */
fact(0,1).
/* Plan1 */
+fact(X,Y) : X < 5
<- +fact(X+1, (X+1) * Y).

/* Plan2 */
+fact(X,Y) : X = 5
<- .print("fact 5 == ", Y).

```

#### Plan1

- ◊ **Trigger:** `+fact(X,Y)`  
Il piano si attiva quando una nuova credenza `fact(X,Y)` viene aggiunta.
- ◊ **Contesto:** `X < 5`  
Il piano si esegue solo se `X` è minore di 5.
- ◊ **Azione:** `+fact(X+1, (X+1)* Y)`  
L'agente aggiunge una nuova credenza `fact(X+1, (X+1)* Y)`  
in pratica, calcola il fattoriale progressivamente.

#### Plan2

- ◊ **Trigger:** `+fact(X,Y)`
- ◊ **Contesto:** `X = 5`
- ◊ **Azione:** stampa il valore di `Y` (cioè il fattoriale di 5)

#### Simboli

Vediamo ora i simboli `+`, `!!` e `!`:

- ◊ `!goal`  
Significa che l'agente adotta un nuovo obiettivo (goal).  
Il `+` indica aggiunta (di un fatto, credenza, o obiettivo).  
Il `!` identifica un goal (obiettivo).
- ◊ `!!goal`  
È un goal persistente.  
L'agente continua a perseguiro anche se fallisce, provando a ripianificare finché non riesce o viene sospeso.  
È usato per creare loop continui, come cicli o comportamenti ricorrenti.
- ◊ `!goal.`  
È una intenzione di raggiungere l'obiettivo goal.  
Serve per specificare quali piani devono essere attivati

```

/* Objetivos iniciales */
!dots.
!control.
/* Planes */
+!dots
<- .print("."); // imprime puntos en un bucle sin fin
!!dots.
+!control

```

```
<- .wait(30); // este plan es un bucle que activa y desactiva el otro plan
.suspend(dots); // suspende la intencion asociada al plan dots
.println;
.wait(200);
.resume(dots); // reactiva la intencion asociada al plan dots
!!control

// Un agente que soluciona el problema del mundo de bloques
/* Creencias iniciales y reglas */
clear(table).

clear(X) :- not(on(_,X)). // la creencia clear(X) es cierta cuando no tiene nada encima
tower([X]) :- on(X,table). // la torre X es cierta cuando X esta sobre la mesa
// X puede ser un bloque o varios
tower([X,Y|T]) :- on(X,Y) & tower([Y|T]).
// la torre va creciendo si un bloque X se pone sobre el bloque Y
/* Objetivos iniciales */
// Marca el estado final a alcanzar
!state([[a,e,b],[f,d,c],[g]]).

/* Planes */
// Alcanzar una torre
+!state([]) <- .print("Finalizado!").
+!state([H|T]) <- !tower(H); !state(T).
// Alcanzar un estado donde la torre esta construida
+!tower(T) : tower(T). // La torre deseada ya existe, no hay nada que hacer
+!tower([T]) <- !on(T,table). // La torre es de un solo elemento
+!tower([X,Y|T]) <- !tower([Y|T]); !on(X,Y). // dividido el problema en subproblemas
```



# Chapter 6

## Ontologías y Comunicación

Una ontología define los términos y conceptos comunes empleados para describir y representar un área de conocimiento Descripción mediante

- Clases
- Instancias
- Relaciones
- Propiedades
- Funciones / procesos
- Restricciones

Representación

- Frases que combinan la terminología para expresar relaciones entre los términos
- Estas frases aportan significado

### 6.1 Ontologías y OWL

Las ontologías permiten representar conocimiento de forma estructurada y compartible entre sistemas. Para construir modelos ontológicos se requieren cuatro niveles:

- **Sintaxis:** Define el orden, formato y estructura (elementos sintácticos como XML)
- **Estructura:** Organización de elementos en jerarquías, herencia y relaciones parte-de
- **Semántica:** Mapeo entre datos estructurados y modelos de objetos que aportan significado
- **Uso:** Pragmática que indica cómo utilizar la semántica

#### 6.1.1 Lenguajes para Representación de Ontologías

##### 6.1.1.1 RDF (Resource Definition Framework)

RDF proporciona un marco estandarizado para representar conocimiento en la web mediante triplets:

- **Recurso** (Sujeto): Entidad de la que se habla
- **Propiedad** (Predicado): Define relaciones o características
- **Objeto**: Entidad a la que se refiere el predicado

##### 6.1.1.2 RDFS (RDF Schema)

Extiende RDF permitiendo definir:

- Jerarquía de clases e instancias
- Restricciones sobre propiedades
- Jerarquía de propiedades

##### 6.1.1.3 OWL (Web Ontology Language)

OWL aporta mayor expresividad, permitiendo definir:

- Clases como combinaciones booleanas (union, intersection, complement)
- Clases disjuntas
- Equivalencia entre clases o individuos
- Cardinalidad en propiedades
- Propiedades transitivas, inversas, funcionales

### 6.1.2 Protégé

Herramienta gratuita y open source para:

- ◊ Editar ontologías
- ◊ Gestionar bases de conocimiento
- ◊ Definir estructuras ontológicas
- ◊ Administrar instancias

## 6.2 Comunicación entre Agentes

La comunicación es fundamental para los sistemas multiagente, permitiendo la cooperación, coordinación y negociación entre agentes.

### 6.2.1 Teoría de Actos de Habla

Los actos de habla consideran el lenguaje como acción, donde cada declaración realiza un acto:

- ◊ **Actos asertivos:** Dan información sobre el mundo
- ◊ **Actos directivos:** Solicitan algo al destinatario
- ◊ **Actos de promesa:** Comprometen al locutor a acciones futuras
- ◊ **Actos expresivos:** Indican estados mentales
- ◊ **Actos declarativos:** La declaración realiza el acto

Componentes de los actos de habla:

- ◊ **Locución:** Modo de producción de frases
- ◊ **Ilocución:** Acto realizado (fuerza ilocutoria + contenido proposicional)
- ◊ **Perlocución:** Efectos en el destinatario

### 6.2.2 Lenguajes de Comunicación entre Agentes

#### 6.2.2.1 KQML (Knowledge Query Management Language)

Lenguaje basado en actos de habla con:

- ◊ Performativas (tell, ask, achieve, reply...)
- ◊ Estructura de mensaje con pares atributo-valor
- ◊ Niveles de contenido, comunicación y mensaje

#### 6.2.2.2 FIPA ACL

Estándar desarrollado por Foundation for Intelligent Physical Agents:

- ◊ Define estructura de mensajes y performativas
- ◊ Estandariza arquitectura de plataformas de agentes
- ◊ Incluye servicios de directorio, gestión y comunicación

### 6.2.3 Comunicación en JASON

JASON implementa comunicación mediante:

- ◊ Estructura de mensajes: <emisor, fuerza\_ilocutoria, contenido>
- ◊ Acción interna: .send(receptor, performativa, contenido)
- ◊ Performativas: tell, untell, achieve, askOne, askAll...

## 6.3 Sistemas Multiagente

Los sistemas multiagente consisten en conjuntos de agentes que interactúan entre sí:

- ◊ Capacidad para coordinar, cooperar y negociar
- ◊ Requieren servicios de localización (páginas blancas y amarillas)
- ◊ Utilizan protocolos de interacción estandarizados

### 6.3.1 Plataformas de Agentes FIPA

El modelo conceptual FIPA incluye:

- ◊ Agent Management System (AMS): gestión de agentes
- ◊ Directory Facilitator (DF): servicio de páginas amarillas
- ◊ Agent Communication Channel (ACC): canal de comunicación
- ◊ Message Transport System: infraestructura de comunicación

## **Part II**

# **Machine Learning**



# Chapter 7

## N-Grams y FSA

### 7.1 N-Grams

“Hello, how are ...”

Dada secuencia de palabras  $w_1, w_2, \dots, w_n$ , queriamos predecir la siguiente palabra  $w_{n+1}$ . O, mejor, queriamos establecer la probabilidad de una secuencia de palabras  $P(w_1, w_2, \dots, w_n)$ , que se puede decomponer en

$$P(w_1, w_2, \dots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \dots P(w_n|w_1, w_2, \dots, w_{n-1}) \quad (7.1)$$

Objectives:

- ◊ To assign a probability score to every word sequence
- ◊ To reflect a previous knowledge of a text source
  - To predict the most likely occurrence of words using its context

$$P(w) = P(w_1) \cdot \prod_{i=2}^m P(w_i|w_1, w_2, \dots, w_{i-1}) \text{ with } m \in \Sigma \quad (7.2)$$

La estimación de la probabilidad  $P(w)$  es prohibitivamente costoso cuando el tamaño del vocabulario  $|\Sigma|$  se hace enorme:  $|\Sigma|^{i-1}$  diferentes historias posibles.

Entonces, tal probabilidad se puede aproximar:

$$P(w) \approx \prod_{i=2}^m P(w_i|\phi(w_1, w_2, \dots, w_{i-1})) = \prod_{i=2}^m P(w_i|w_1, w_2, \dots, w_{i-1}) \quad (7.3)$$

La estimación de la probabilidad típicamente se hace usando relative frequency counts  $f(\cdot| \cdot)$ :

$$P(w_i|w_1, w_2, \dots, w_{i-1}) = f(w_i|w_1, w_2, \dots, w_{i-1}) = \frac{C(w_1, w_2, \dots, w_i)}{C(w_1, w_2, \dots, w_{i-1})} \quad (7.4)$$

#### Example

$\mathcal{S} = \{\text{el perro corre rapido, el perro corre rapido, el tren azul corre veloz, el tren azul corre veloz, el coche azul corre veloz, el coche azul corre veloz}\}$

1-gram counts:	2-gram counts:	3-gram counts:
< s > 6	< s > el 6	< s > el perro 2
el 6	el perro 2	< s > el tren 2
perro 2	el tren 2	< s > el coche 2
corre 6	el coche 2	el perro corre 2
rapido 2	perro corre 2	el tren azul 2
< /s > 6	corre rapido 2	el coche azul 2
tren 2	corre veloz 4	perro corre rapido 2
azul 4	rapido < /s > 2	corre rapido < /s > 2
veloz 4	tren azul 2	corre veloz < /s > 4
coche 2	azul corre 4	tren azul corre 2
	veloz < /s > 4	azul corre veloz 4
	coche azul 2	coche azul corre 2

Figure 7.1: N-Grams Example

Un LM (Language Model, o Modelo de Lenguaje) es un modelo probabilístico que asigna una probabilidad a una secuencia de palabras. Un N-gram LM se dice *completo* si todas las posibles secuencias de palabras tienen una apropiada estimación de probabilidad.

El problema con esto es que hay eventos (secuencias de palabras) que casi nunca ocurren en el corpus de entrenamiento (*training set*), y por lo tanto no tienen una estimación de probabilidad. Por este motivo hay métodos de *smoothing* y *discounting*.

La idea es que si una secuencia de palabras no ha sido vista en el corpus, se utiliza la probabilidad de una secuencia más corta, multiplicada por un factor de *back-off*. En un modelo suavizado por back-off (por ejemplo Katz Back-off), se reduce también la probabilidad de eventos que tienen una probabilidad no nula.

El *smoothing* es un método para ajustar las probabilidades de los eventos que no han sido vistos en el corpus, para que no tengan una probabilidad cero.

Smoothing	Discounting
Back-Off	Good Turing
Interpolation	Absolute Discounting
	Witten Bell
	Linear Discounting
	Kneser-Ney
	...

Table 7.1: Smoothing and discounting

$$P(w_i|w_{i-n+1}^{i-1}) = \begin{cases} P^*(w_i|w_{i-n+1}^{i-1}) & \text{if } C(w_{i-n+1}^i) > 0 \\ \beta(w_{i-n+1}^{i-1})P(w_i|w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases}$$

- ▶  $P^*(\cdot|\cdot)$ : discounted probability; reserves mass for unseen events
- ▶  $\beta$ : back-off weight; ensures the consistence of the model probabilities

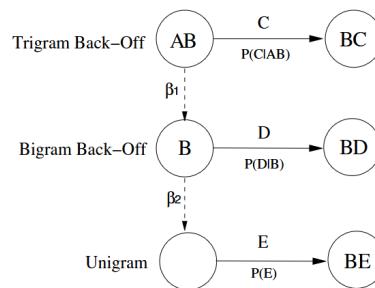


Figure 7.2: Back-Off

### Ejercicio

#### 1-Gram

P(<s>)	P(el)	P(perro)	P(corre)	P(rapido)	P(</s>)	
6	6	2	6	2	6	81
-	-	-	-	-	-	$* 10^{-6}$
34	34	34	34	34	34	64

34 perché la somma delle occorrenze delle singole parole è 40, a cui però dobbiamo togliere le 6 occorrenze della marca di fine stringa, perché quella non contribuisce ai possibili prefissi (historia)

#### 2-GRAM

P(<s>)	P(el <s>)	P(perro el)	P(corre perro)	P(rapido corre)	P(</s> rapido)
0	6	2	2	2	1
-	-	-	-	-	-
6	6	2	6	2	9

#### 3-GRAM

P(<s>)	P(el <s>)	P(perro <s>el)	P(corre el perro)	P(rapido perro corre)	P(</s> corre rapido)
0	0	2	1	1	1

## 7.2 FSA - Finite State Automata

Introduction to PFA Introduction to Probabilistic Finite-State Automata

- ◊ Free Monoid  $\Sigma^*$ : Given a finite set  $\Sigma$ ,  $\Sigma^+$  is the set of all strings with finite length composed of elements from  $\Sigma$

$$\Sigma^* = \Sigma^+ \cup \{\lambda\} (\lambda \text{ is the empty string}) \quad (7.5)$$

- ◊ Grammar:  $G = (N, \Sigma, R, S)$

- ◊  $N$ : Finite set of non-terminal symbols

- ◊  $\Sigma$ : Finite set of terminal symbols or primitives

- ◊  $S \in N$ : Initial non-terminal symbol or “axiom”

- ◊  $R \subset (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ : set of rules

or productions A rule is written as:

$$\alpha \rightarrow \beta, \quad \alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma)^* \quad (7.6)$$

**Elemental derivation:**  $\Rightarrow G$ :  $\mu\alpha\delta \Rightarrow_G \mu\beta\delta \iff \exists(\alpha \rightarrow \beta) \in R, \mu, \delta \in (N \cup \Sigma)^*$

**Derivation**  $\stackrel{*}{\Rightarrow}_G$ : It is a finite sequence of elemental derivations. A derivation  $d$  can be written as the corresponding rule sequence of  $G$ . The set of derivations of  $y \in \Sigma^*$  (such that  $S \stackrel{*}{\Rightarrow}_G y$ ) is written as  $D_G(y)$ . A grammar  $G$  is ambiguous if  $\exists y \in \Sigma^*$  such that  $|D_G(y)| > 1$ .

### Derivation

Quindi:

$\alpha$  è la parte sinistra della regola, chiamata anche contesto o parte da sostituire. È una stringa che deve contenere almeno un non terminale, poiché solo i non terminali possono essere sostituiti.

$\beta$  è la parte destra, cioè ciò che sostituisce  $\alpha$  durante la derivazione.

Supponiamo:

$$\begin{aligned} N &= A, B \\ \Sigma &= a, b \\ R &= aAb \rightarrow abb \\ \alpha &= aAb \\ \beta &= abb \end{aligned}$$

La regola dice che se in una stringa troviamo la sottostringa  $aAb$ , possiamo sostituirla con  $abb$ .

L'elemento:

$$\mu\alpha\sigma \Rightarrow_G \mu\beta\sigma$$

significa che:

Se in una stringa  $\mu\alpha\sigma$ , si trova la sottostringa  $\alpha$ , E se esiste una regola  $\alpha \rightarrow \beta \in R$ , Allora si può sostituire  $\alpha$  con  $\beta$ , ottenendo  $\mu\beta\sigma$ .

Quindi,  $\alpha$  e  $\beta$  sono le parti centrali della riscrittura, mentre  $\mu$  e  $\sigma$  rappresentano il contesto a sinistra e a destra, rispettivamente.

**Language generated by a grammar  $G$ ,  $L(G)$ :**  $L(G) = \{y \in \Sigma^* \mid S \stackrel{*}{\Rightarrow}_G y\}$

- ◊ **Regular grammars:**  $G = (N, \Sigma, R, S)$ , Rules of  $R$ :  $A \rightarrow aB \vee A \rightarrow a, A, B \in N, a \in \Sigma$

- ◊ **Finite-state automaton:** (non deterministic)

$$A = (Q, \Sigma, \delta, q_0, F), \quad q_0 \in Q, \quad F \subseteq Q, \quad \delta : Q \times \Sigma \rightarrow 2^Q$$

- ◊ **Equivalence:** For each regular grammar there exists a finite-state automaton that recognizes the same language<sup>1</sup>

**Example:**

$$\begin{aligned} G = & (N, \Sigma, R, S); \\ \Sigma = & \{a, b\}; N = \{S, A_1, A_2\}; \\ R = & \{ S \rightarrow aA_1 \mid bA_2 \mid b, \\ A_1 \rightarrow & aA_1 \mid bA_2 \mid b, \\ A_2 \rightarrow & bA_2 \mid b \} \end{aligned}$$

$$\begin{aligned} \mathcal{A} = & (Q, \Sigma, \delta, q_0, F); \\ Q = & \{0, 1, 2\}, \\ \Sigma = & \{a, b\}, \\ q_0 = & 0, F = \{2\} \end{aligned}$$

$$\mathcal{L}(G) = \{b, ab, bb, aab, abb, bbb, \dots, aaabb, \dots\} = \mathcal{L}(\mathcal{A})$$

Figure 7.3: FSA Equivalence example

Probabilistic Finite-state Automata (PFA): it is a tuple  $A = \langle Q, \Sigma, \delta, I, F, P \rangle$ , where:

- ◊  $Q$ : finite set of states
- ◊  $\Sigma$ : finite alphabet of symbols
- ◊  $\delta \subseteq Q \times \Sigma \times Q$ : transition function
- ◊  $I : Q \rightarrow \mathbb{R}^+ \cup \{0\}$ : initial state probabilities, with  $\sum_{q \in Q} I(q) = 1$
- ◊  $F : Q \rightarrow \mathbb{R}^+ \cup \{0\}$ : final state probabilities
- ◊  $P : \delta \rightarrow \mathbb{R}^+ \cup \{0\}$ : transition probabilities, such that

$$\text{with } \forall q \in Q : F(q) + \sum_{q' \in Q, \sigma \in \Sigma} P(q, \sigma, q') = 1 \quad (7.7)$$

A path in a PFA is a sequence of transitions:  $\pi = (q_0, a_1, q_1)(q_1, a_2, q_2)\dots(q_{n-1}, a_n, q_n)$   
The probability of a path  $\pi$  is:

$$P(\pi) = I(q_0) \cdot \left( \prod_{i=1}^n P(q_{i-1}, a_i, q_i) \right) \cdot F(q_n) \quad (7.8)$$

The probability of a string  $x = a_1a_2\dots a_n$  is the sum of probabilities of all paths that generate  $x$ :

$$P_A(x) = \sum_{\pi \in \Pi_A(x)} P(\pi) \quad (7.9)$$

where  $\Pi_A(x)$  is the set of all paths in automaton  $A$  that generate string  $x$ .

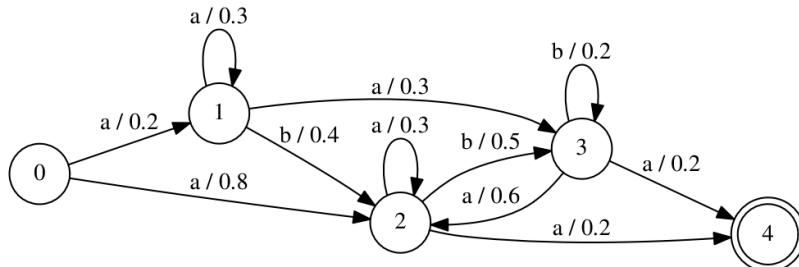


Figure 7.4: Example of a Probabilistic Finite-state Automaton

### 7.2.1 Applications of PFA

PFAs are widely used in:

- ◊ Speech recognition
- ◊ Natural language processing
- ◊ Pattern recognition
- ◊ Biological sequence analysis

### 7.2.2 Relation between N-Grams and PFA

Any n-gram model can be represented as a PFA, where:

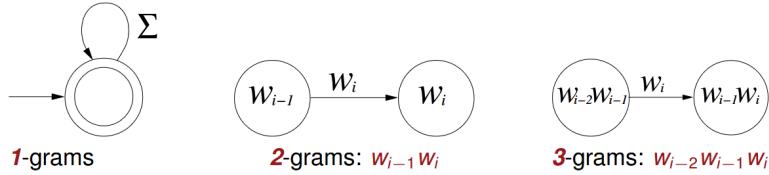
<sup>1</sup>The contrary isn't always true for stochastic languages.

- ◊ States represent histories (previous n-1 words)
- ◊ Transitions represent word emissions with their corresponding probabilities
- ◊ The resulting automaton has a specific structure reflecting the Markov property of n-gram models

### 7.3 Relación entre N-Grams y PFA

*n*-grams can be represented using deterministic PFA

Examples of PFA representing 1-grams, 2-grams and 3-grams:



For 2-grams:  $q = w_{i-1}$  and  $q' = w_i$ .

For 3-grams:  $q = w_{i-2}w_{i-1}$  and  $q' = w_{i-1}w_i$ .

Figure 7.5: N-Gram representados como FSA



# Chapter 8

## Deep Learning

### 8.1 Introducción

La unidad basica es el llamado **neuron**, que implementa una *Función Discriminante Lineal* (LDF) con una función de activación. Un neurón recibe múltiples entradas, aplica pesos a estas entradas, suma los resultados y pasa esta suma a través de una función de activación para producir una salida.

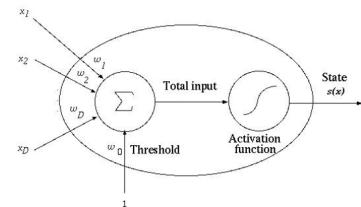


Figure 8.1: Neuron

Matemáticamente, un neurón se puede representar como:

$$y = f \left( \sum_{i=1}^n w_i x_i + b \right)$$

Donde:

- ◊  $x_i$  son las entradas
- ◊  $w_i$  son los pesos
- ◊  $b$  es el sesgo (bias)
- ◊  $f$  es la función de activación

Neurones pueden ser combinados en CAPAS (*layers*): lineal layers, convolutional layers, recurrent layers, etc. Capas pueden ser combinados en redes neuronales (*neural networks*).

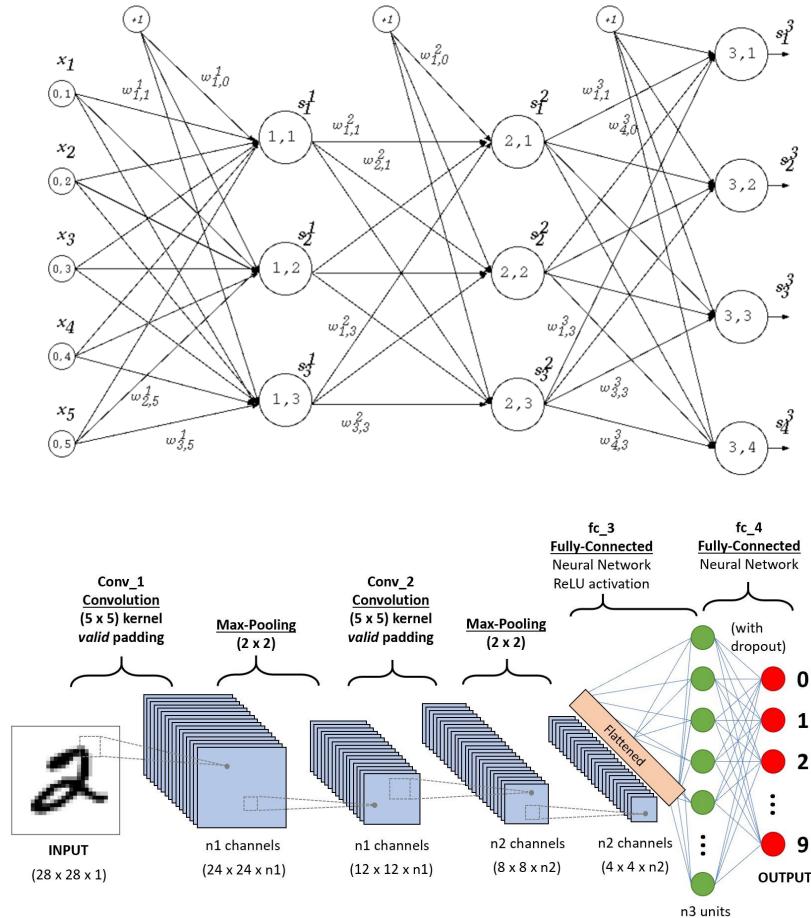


Figure 8.1: Capas y redes neuronales

## 8.2 Linear Discriminant Function and Perceptron

A classifier  $G$  in  $C$  classes can be defined by  $C$  discriminant functions  $g_c$ .

Classification rule ( $\vec{x} \in \mathbb{R}^D$ ) :  $\hat{c} = G(\vec{x}) = \text{argmax}_{c=1, \dots, C} g_c(\vec{x})$

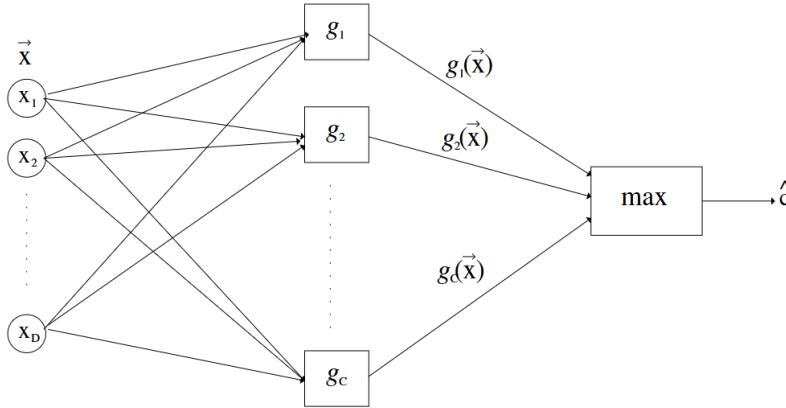


Figure 8.2: Discriminant function Graph

When  $g_c$  are linear functions, they are called **Linear Discriminant Functions (LDF)**

$$g(\vec{x}) = w_0 + \sum_{i=1}^D \vec{w}_i \vec{x}_i = w_0 + \vec{w}^t \vec{x}$$

$\vec{w}$  is the weights vector,  $w_0$  is the bias term In homogeneous notation:  $w = (w_0, \vec{w}^t)^t, x = (1, \vec{x}^t)^t$

$$g(\vec{x}) = w^t x$$

$G$  is a linear classifier when all  $g_c$  are linear functions, with parameters  $w_c$ . Given a training dataset of samples in  $\mathbb{R}^D$ ,  $w_c$  can be estimated by using the Perceptron algorithm

```

// Let be: w_j, 1 ≤ j ≤ C, C initial weight vectors;
//          (x̄_1, c_1), ..., (x̄_N, c_N), N training samples;
//          α ∈ ℝ > 0, "learning factor";, b ∈ ℝ, "margin".
do {
    m = 0                                // correctly classified samples
    for (n = 1; n ≤ N; n++) {             // for each training sample
        i = c_n; g = w_i^t x_n; error=false // correct class score
        for (j = 1; j ≤ C; j++) {
            if (j ≠ i) {
                if (w_j^t x_n + b > g) {      // incorrect classification
                    w_j = w_j - α x_n;           // decrease wrong weights
                    error=true
                }
            }
        }
        if (error) w_i = w_i + α x_n;       // increase correct weights
        else m = m + 1
    }
} while (m < N)                         // until all training samples well classified
  
```

Figure 8.3: Perceptron algorithm

The main limitation is that LDF provide linear frontiers only linearly separable classes can be properly separated. Using margin b allows for a non-optimal solution Combining several LDF in cascade does not solve the problem: result is still an LDF.

Solution: use of *Activation Functions* → **Logistic LDF** (neuron).

Some activation functions:

- ▶ *Linear*:  $f_L(z) = z, z \in \mathbb{R}$
- ▶ *Step*:  $f_E(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}, z \in \mathbb{R}$
- ▶ *ReLU* (rectified linear unit):  $f_R(z) = \max(0, z), z \in \mathbb{R}$
- ▶ *Sigmoid*:  $f_S(z) = \frac{1}{1 + \exp(-z)}, z \in \mathbb{R}$
- ▶ *Hyperbolic tangent*:  $f_T(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}, z \in \mathbb{R}$
- ▶ *Softmax*:  $f_{SM}(z_j) = \frac{\exp(z_j)}{\sum_{j'=1}^M \exp(z_{j'})}, z_1, \dots, z_M \in \mathbb{R}$

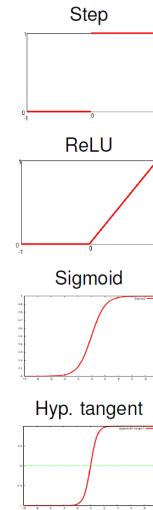


Figure 8.4: Activation Functions

### 8.2.1 Backpropagation

The **Backpropagation** algorithm is used to train neural networks by minimizing the error between the predicted output and the actual output. It works by calculating the gradient of the loss function with respect to each weight in the network, allowing for efficient weight updates.

#### The BackProp Algorithm

*Input*: Topology, initial weights  $w_{ij}^l, 1 \leq l \leq L, 1 \leq i \leq M_l, 0 \leq j \leq M_{l-1}$ , learning rate  $\rho$ , convergence conditions,  $N$  training samples  $\mathcal{X}$

*Output*: Weights of connections that minimize the MSE of  $\mathcal{X}$

While no convergence

For  $1 \leq l \leq L, 1 \leq i \leq M_l, 0 \leq j \leq M_{l-1}$ , initialize  $\Delta w_{ij}^l = 0$

For each training sample  $(\mathbf{x}, \mathbf{t}) \in \mathcal{X}$

From the input to the output layers ( $l = 0, \dots, L$ ):

For  $1 \leq i \leq M_l$  if  $l = 0$  then  $s_i^0 = x_i$  else compute  $z_i^l$  and  $s_i^l = f(z_i^l)$

From the output to the input layers ( $l = L, \dots, 1$ ),

For each node ( $1 \leq i \leq M_l$ )

$$\text{Compute } \delta_i^l = \begin{cases} f'(z_i^l) (t_{ni} - s_i^l) & \text{if } l == L \\ f'(z_i^l) (\sum_r \delta_r^{l+1} w_{ri}^{l+1}) & \text{otherwise} \end{cases}$$

For each weight  $w_{ij}^l (0 \leq j \leq M_{l-1})$  compute:  $\Delta w_{ij}^l = \Delta w_{ij}^l + \rho \delta_i^l s_j^{l-1}$

For  $1 \leq l \leq L, 1 \leq i \leq M_l, 0 \leq j \leq M_{l-1}$ , update weights:  $w_{ij}^l = w_{ij}^l + \frac{1}{N} \Delta w_{ij}^l$

Figure 8.5: Backpropagation Algorithm

**Algorithm 1** BackProp Algorithm

---

```

1: Input: Topology, initial weights  $w_{ij}^l$ ,  $1 \leq l \leq L$ ,  $1 \leq i \leq M_l$ ,  $0 \leq j \leq M_{l-1}$ ; learning rate  $\rho$ ; convergence conditions;
    $N$  training samples  $\mathcal{X}$ 
2: Output: Weights that minimize the MSE of  $\mathcal{X}$ 
3: while no convergence do
4:   for  $1 \leq l \leq L$ ,  $1 \leq i \leq M_l$ ,  $0 \leq j \leq M_{l-1}$  do
5:     Initialize  $\Delta w_{ij}^l = 0$ 
6:     for each training sample  $(\mathbf{x}, \mathbf{t}) \in \mathcal{X}$  do                                 $\triangleright$  From input to output layers ( $l = 0, \dots, L$ )
7:       for  $1 \leq i \leq M_l$  do
8:         if  $l == 0$  then
9:            $s_i^0 = x_i$ 
10:        else
11:          Compute  $z_i^l$  and  $s_i^l = f(z_i^l)$                                           $\triangleright$  From output to input layers ( $l = L, \dots, 1$ )
12:        for each node  $1 \leq i \leq M_l$  do
13:          if  $l == L$  then
14:             $\delta_i^l = f'(z_i^l)(t_{ni} - s_i^L)$ 
15:          else
16:             $\delta_i^l = f'(z_i^l) (\sum_r \delta_r^{l+1} w_{ri}^{l+1})$ 
17:          for each weight  $w_{ij}^l$  ( $0 \leq j \leq M_{l-1}$ ) do
18:             $\Delta w_{ij}^l = \Delta w_{ij}^l + \rho \delta_i^l s_j^{l-1}$ 
19:        for  $1 \leq l \leq L$ ,  $1 \leq i \leq M_l$ ,  $0 \leq j \leq M_{l-1}$  do
20:          Update weights:  $w_{ij}^l = w_{ij}^l + \frac{1}{N} \Delta w_{ij}^l$ 

```

---

### 8.2.2 Resumen del algoritmo de Backpropagation

El algoritmo de Backpropagation (retropropagación) es fundamental para el entrenamiento de redes neuronales y funciona de la siguiente manera:

1. **Inicialización:** Se parte de una topología de red definida con pesos iniciales aleatorios, una tasa de aprendizaje y un conjunto de datos de entrenamiento.
2. **Ciclo principal:** Se itera hasta alcanzar la convergencia (cuando el error es suficientemente pequeño o se alcanza un número máximo de iteraciones).
3. **Para cada ciclo:**
  - ◊ Se inicializan los cambios de pesos a cero.
  - ◊ Para cada muestra de entrenamiento:
    - **Propagación hacia adelante (forward pass):** Se calcula la salida de la red, capa por capa, desde la entrada hasta la salida.
    - **Propagación hacia atrás (backward pass):** Se calcula el error en la capa de salida y se propaga hacia atrás, calculando los deltas (errores) para cada neurona en cada capa.
    - Se acumulan los cambios necesarios para cada peso basados en los deltas calculados y las activaciones previas.
  - ◊ Se actualizan todos los pesos usando el promedio de los cambios calculados para todas las muestras.

La clave del algoritmo está en la regla de la cadena del cálculo diferencial, que permite calcular eficientemente cómo contribuye cada peso al error total de la red, facilitando así su actualización en la dirección que reduce dicho error. Para la clasificación  $J$  se utiliza la Cross-Entropy Loss Function:

$$C_\chi = -\frac{1}{N} \sum_{n=1}^N \sum_{c=1}^C t_{nc} \log s_c^L(\mathbf{x}_n; \mathbf{w})$$

La convergencia depende de la tasa de aprendizaje  $\rho$  y del número de iteraciones  $N$ .

- ◊ Si  $\rho$  es muy grande, la convergencia puede ser inestable y oscilar.
- ◊ Si  $\rho$  es muy pequeña, la convergencia puede ser muy lenta
- ◊  $\rho = 2/(\delta_{max} + \delta_{min})$  es una buena aproximación, donde  $\delta_{max}$  y  $\delta_{min}$  son los valores máximos y mínimos de los deltas. Garantiza convergencia asymptoticamente.



# Chapter 9

# Deep Learning for Text Classification

FFN structure has problems when dealing with text representations:

- ◊ Texts are composed of discrete objects (words, characters), FFN expect numerical inputs
- ◊ Texts lengths are variable, input size of a FFN is fixed

Possible solutions:

- ◊ Word embeddings to convert text sequences into numerical data
- ◊ Recurrent networks or other alternatives to process sequences

These problems appear in other objects: speech, handwritten text, general temporal sequences, protein chains, video, large images,...

## 9.1 Word Embeddings

Discrete symbols can be encoded by assigning them one or more numbers E.g., “a” → 1, “b” → 2, ...

To fit the natural input of neural networks, many representations are possible:

- ◊ One-hot encoding (or local) - a binary vector, size of vocabulary, 1 in the position of the symbol, 0 in the rest of positions
- ◊ Distributed - binary encoding of an assigned natural number
- ◊ Embedding - real values vector of a predefined dimension D

Symbol	One-hot					Distributed		Embedding	
‘a’	0	0	0	0	1	0	0	0.15	0.66
‘b’	0	0	0	0	1	0	0	-0.66	0.02
‘c’	0	0	0	1	0	0	1	0.32	-0.49
‘d’	0	0	1	0	0	0	1	0.89	0.78
‘e’	0	1	0	0	0	0	1	-0.17	-0.12
‘f’	1	0	0	0	0	0	1	0	0.33
									0.55

Figure 9.2: Word Embeddings

## 9.2 Word Embeddings: Word2Vec

Word2Vec is a method to learn word embeddings from large text corpora. It uses a shallow neural network to predict the context of a word or the word itself given its context. The two main architectures are:

- ◊ Continuous Bag of Words (CBOW): Predicts a word given its context.
- ◊ Skip-Gram: Predicts the context given a word.

## 9.3 RNN: Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data, such as text or time series. They maintain a hidden state that captures information about previous inputs, allowing them to process sequences of variable length. Simple RNN have some problems:

- ◊ Output depends on past information
  - It depends on more recent or older information according the moment
  - If a balance is not kept, recent information dominates
- ◊ Errors in backward propagation tend to vanish (tend to 0) or oscillate (do not allow convergence)

Solution: units with dedicated parts (gates) to different inputs and outputs

- ◊ *Long-Short Term Memories (LSTM)*
- ◊ *Gated Recurrent Units (GRU)*

### 9.3.1 Encoders and Decoders

Encoders and decoders are architectures used in sequence-to-sequence tasks, such as machine translation or text summarization. The encoder processes the input sequence and compresses it into a fixed-size context vector, while

the decoder generates the output sequence from this context vector.

- ◊ **Encoder:** from input sequence  $x$  generates context representations sequence  $h^e$
- ◊ **Context:**  $c = f(h_n^e)$ , keeps input essence
- ◊ **Decoder:** from context  $c$  generates output sequence  $y$  Non-autoregressive in training, autoregressive in production

Hay el problema del *bottleneck* en el contexto, que puede ser resuelto con **attention mechanisms**. El problema del *bottleneck* es que la información de la secuencia de entrada se comprime en un vector de contexto, lo que puede llevar a una pérdida de información importante, especialmente en secuencias largas.

### Transformers

Transformers are a type of neural network architecture that uses self-attention mechanisms to process sequences. They are particularly effective for tasks like machine translation, text summarization, and language modeling. Transformers consist of an encoder and a decoder, both composed of multiple layers of self-attention and feed-forward networks.

- ◊ Query: current input where attention is computed
- ◊ Key: context compared to current input
- ◊ Value: final attention value

# Chapter 10

# Deep Learning for Image Classification

## 10.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are the most widespread architecture for image processing in DL. They are based on the idea of convolutional filters, which are small matrices that slide over the input image to detect local patterns. The main advantage of CNNs is their ability to learn hierarchical representations of the data, where lower layers detect simple features (e.g., edges, textures) and higher layers detect more complex features (e.g., shapes, objects). This hierarchical representation allows CNNs to achieve state-of-the-art performance in various image classification tasks, such as object detection, semantic segmentation, and image generation.

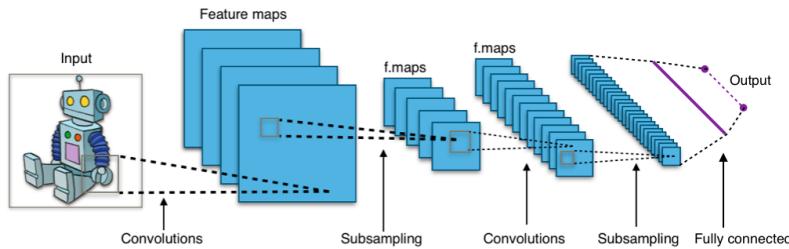


Figure 10.1: Typical CNN architecture

Main components:

- ◊ Convolutional blocks: convolutions, activation functions, subsampling
- ◊ Classification/regression block: fully connected layers

Trainable components:

- ◊ Convolution values
- ◊ Fully connected layers weights

The process is based on 2D convolution. In each convolution block we have the following features:

- ◊ **Kernel size  $K$** : usually small kernels ( $3 \times 3, 5 \times 5, \dots$ )
- ◊ **Number of filters  $F$** : how many different kernels are used
- ◊ **Padding size  $P$** : border padding, usually with zero values
- ◊ **Stride  $S$** : pixel sampling,  $S = 1$  means all pixels

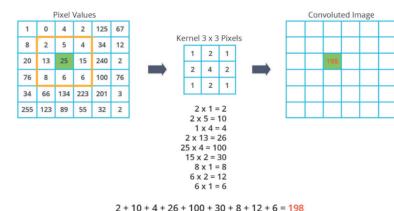


Figure 10.2: Example of 2D convolution

Each convolutional block has as output a set of  $F$  maps; each map is built by the application of the corresponding filter to each possible pixel/component of the input.

Size for each map is the following (for  $H \times W$  input):

$$\frac{1}{S^2} (H + 2P - (K - 1)) \times (W + 2P - (K - 1))$$

Spatial cost mainly dependent on number of filters.

The total number of products (for  $H \times W$  input) instead:

$$K^2 \frac{F}{S^2} (H + 2P - (K - 1))(W + 2P - (K - 1))$$

Temporal cost mainly dependent on kernel size

### 10.1.1 Pooling

Usual activation function: *ReLU*

Subsampling (pooling):

- ◊ Allows generalization and improves robustness
- ◊ Reduces size and computational cost
- ◊ Fixed operators, small square size ( $2 \times 2$ )

Subsampling is usually done after each convolutional block, and consists of reducing the spatial dimensions of feature maps while preserving their important information. The most common types of pooling are:

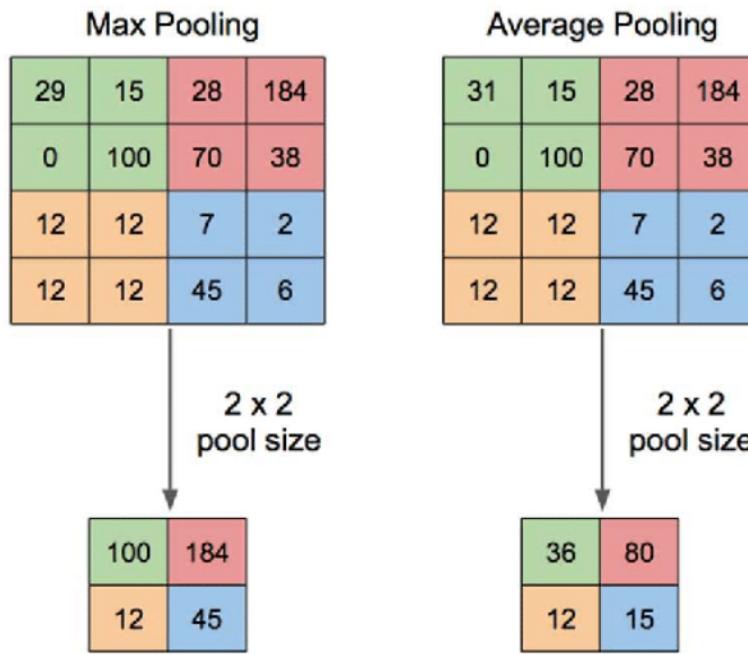


Figure 10.2: Different types of pooling operations

- ◊ **Max pooling:** Takes the maximum value in each window
- ◊ **Average pooling:** Takes the average of all values in the window
- ◊ **Global pooling:** Applies pooling across the entire feature map

Max pooling is most commonly used because it helps in:

- ◊ Retaining the strongest features
- ◊ Making the network more invariant to small translations
- ◊ Reducing spatial dimensions, which decreases computation
- ◊ Controlling overfitting by reducing the number of parameters

The output size after applying pooling with window size  $P$  and stride  $S$  is:

$$\lfloor \frac{W - P}{S} + 1 \rfloor \times \lfloor \frac{H - P}{S} + 1 \rfloor$$

#### 10.1.1.1 Convolutional Blocks

- ◊ Input (verde):  
Sono rappresentate  $M$  mappe di input (ad esempio, i canali di un'immagine RGB hanno  $M = 3$ ). Ogni mappa ha dimensioni  $H \times W$ .
- ◊ Kernels (azzurro):  
Ogni filtro (kernel) convoluzionale ha dimensioni  $K \times K$  e viene applicato a tutte le  $M$  mappe di input. Ci sono

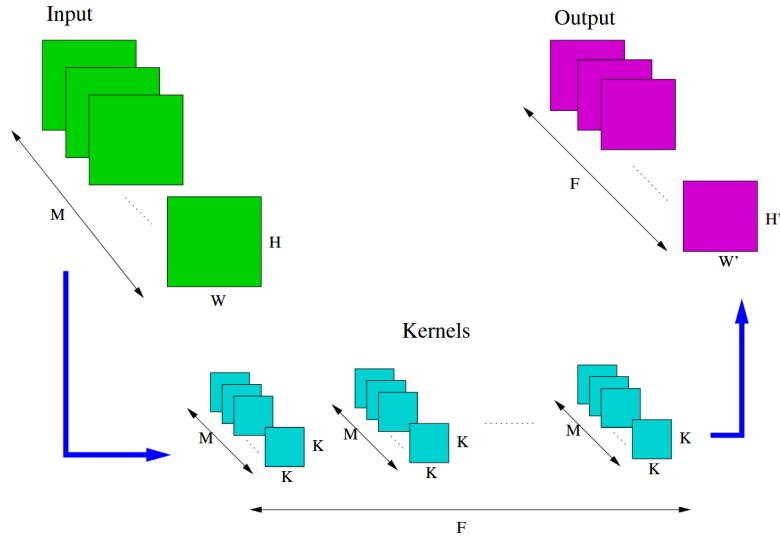


Figure 10.3: Connection between convolutional blocks: each input map has its own set of filters, but output maps are derived from the related filters

$F$  filtri diversi, ognuno dei quali genera una mappa di output. Ogni filtro è quindi un tensore di dimensioni  $M \times K \times K$ .

◊ Output (viola):

L'applicazione dei  $F$  filtri produce  $F$  mappe di output, ciascuna di dimensioni  $H' \times W'$  (dove  $H'$  e  $W'$  dipendono da padding e stride).

Ogni filtro convoluzionale “scorre” su tutte le mappe di input, combinando le informazioni tramite prodotti scalari.  
Ogni filtro produce una mappa di output. Tutte le mappe di output vengono raccolte e formano l'output del blocco convoluzionale.

In sintesi:

L'immagine mostra come, partendo da  $M$  mappe di input, tramite  $F$  filtri convoluzionali (ognuno con profondità  $M$ ), si ottengono  $F$  mappe di output. Questo è il cuore del funzionamento di un blocco convoluzionale nelle CNN.

Last convolutional block output is a set of 2D maps (matrices) Classification block expects 1D input (vector) Flattening operation is needed to pass from 2D to 1D Flattening size is

$$\frac{F}{S^2 p^2} (H + 2P - (K - 1))(W + 2P - (K - 1))$$

$(H, W)$  size of input maps for last conv block, with  $F$  filters of size  $K$ , padding  $P$ , stride  $S$ , pooling size  $p$

Classification block:

- ◊ Input layer: size of flattening
- ◊ Output layer: size of class number, softmax activation
- ◊ Hidden layers: size usually reducing

Much more parameters in this block than in convolutional blocks

Additional operations: dropout, batch normalisation

## 10.2 CNN Architectures

Specific tasks require specific structures of convolutional and non-convolutional blocks. Many different CNN architectures have been developed over the years, each with specific strengths and applications. The following are some of the most influential CNN architectures in the field:

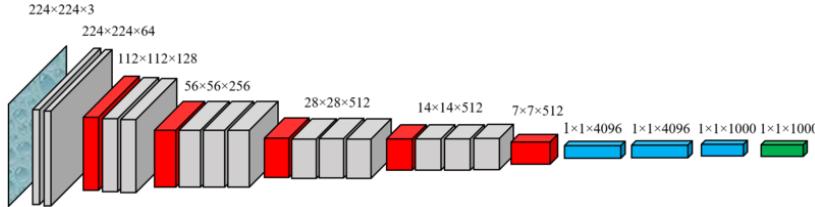


Figure 10.4: VGGNet

### 10.2.1 VGGNet

Introduced by the Visual Geometry Group at Oxford, VGG networks (particularly VGG-16 and VGG-19) feature a simple, uniform architecture with small ( $3 \times 3$ ) convolutional filters stacked deeply. They demonstrated that depth is crucial for performance.

### 10.2.2 ResNet

ResNet: based on residual blocks to avoid vanishing gradients (He et al., 2016)

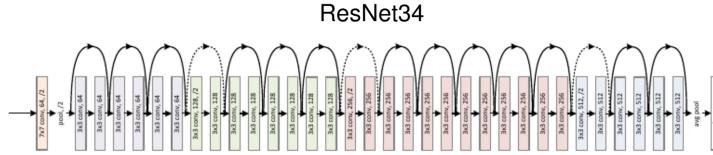
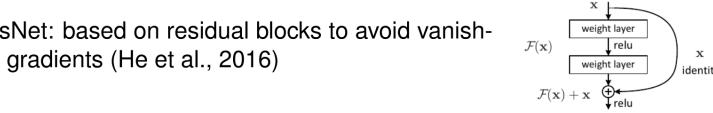


Figure 10.5: ResNet

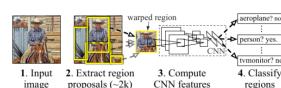
Residual Networks address the degradation problem in very deep networks through skip connections or "residual blocks" that allow gradients to flow through the network more easily. ResNet made it possible to train networks with hundreds or even thousands of layers.

### 10.2.3 Detection Architectures

Detection architectures: two-steps (R-CNN family), one-step (YOLO family)

Region CNN (R-CNN) (Girshick et al., 2014)

- ▶ Region extraction, CNN processing, classification (SVM)
- ▶ Fast R-CNN: global CNN processing
- ▶ Faster R-CNN: Region Proposal Network (RPN)



You Only Look Once (YOLO) (Redmon et al., 2016)

- ▶ Grid division ( $S \times S$ )
- ▶ Bounding box and class prediction by cell
- ▶ Tensor encoding
- ▶ Several versions: YOLOv2, YOLOv3, YOLOv5, YOLOv7, ..., YOLO11

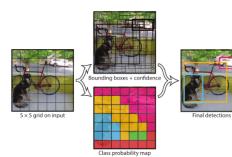


Figure 10.6: R-CNN vs YOLO

#### 10.2.3.1 R-CNN

Region-based Convolutional Neural Networks (R-CNN) is a family of models designed for object detection. The original R-CNN proposed a two-stage approach: first, it generates region proposals using selective search, and then it classifies these regions using a CNN. This method significantly improved object detection performance compared to traditional methods. R-CNN has evolved into several variants, including Fast R-CNN and Faster R-CNN, which improve speed and accuracy by integrating region proposal networks (RPNs) directly into the architecture.

### 10.2.3.2 YOLO

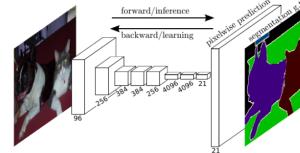
You Only Look Once (YOLO) is a real-time object detection system that treats object detection as a single regression problem. Instead of generating region proposals, YOLO divides the image into a grid and predicts bounding boxes and class probabilities directly from the entire image in one pass. This approach allows for faster inference times compared to R-CNN-based methods, making YOLO suitable for real-time applications. YOLO has undergone several iterations, with improvements in accuracy and speed. The latest versions, such as YOLOv4 and YOLOv5, have introduced various enhancements, including better backbone networks, data augmentation techniques, and advanced loss functions.

### 10.2.4 Segmentation Architectures

Segmentation architectures: FCN, U-Net

Fully Connected Network (FCN) (Long et al., 2015)

- ▶ Only convolutional blocks
- ▶ Image-to-image network
- ▶ Pixel-based segmentation



U-Net (Ronneberger et al., 2015)

- ▶ Only convolutional blocks
- ▶ Downsampling + upsampling
- ▶ Low data requirements
- ▶ Overlapped window segmentation

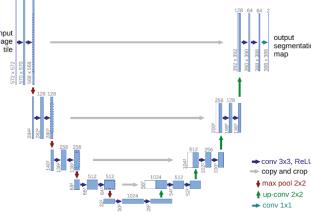


Figure 10.7: Segmentation architectures

#### 10.2.4.1 FCN

Fully Convolutional Networks (FCNs) are a type of CNN designed for semantic segmentation tasks. Unlike traditional CNNs that output a single label for the entire image, FCNs produce a pixel-wise classification map. This is achieved by replacing fully connected layers with convolutional layers, allowing the network to maintain spatial information throughout the process. FCNs are trained end-to-end, meaning that the entire network is optimized simultaneously for the segmentation task. They can be used for various applications, including image segmentation, object detection, and scene understanding. FCNs are built upon standard CNN architectures, but they differ in the following ways:

- ◊ **No fully connected layers:** FCNs replace the fully connected layers with convolutional layers, allowing the network to produce output maps of the same spatial dimensions as the input image.
- ◊ **Upsampling layers:** FCNs use upsampling (or deconvolution) layers to increase the spatial resolution of the output maps, enabling pixel-wise classification.
- ◊ **Skip connections:** FCNs often incorporate skip connections from earlier layers to later layers, allowing the network to combine low-level features with high-level features for better segmentation results.

#### 10.2.4.2 U-Net

U-Net is a specialized architecture for semantic segmentation, particularly in biomedical image analysis. It consists of an encoder-decoder structure with skip connections that allow the network to combine low-level features from the encoder with high-level features from the decoder. This helps in preserving spatial information and improving segmentation accuracy. U-Net is particularly effective for tasks where the amount of training data is limited, as it can learn to produce accurate segmentations even with small datasets. It has been widely adopted in medical imaging, satellite imagery analysis, and other applications requiring precise pixel-wise classification.

### 10.2.5 Transformer Architectures

#### 10.2.5.1 Vision Transformer (ViT)

Vision Transformer (ViT) is a novel architecture that applies the transformer model, originally designed for natural language processing, to image classification tasks. Instead of using convolutional layers, ViT treats an image as a sequence of patches and processes them using self-attention mechanisms. This allows the model to capture long-range dependencies and relationships between different parts of the image. ViT has shown competitive performance

compared to traditional CNNs, especially when trained on large datasets. It has inspired further research into transformer-based architectures for various computer vision tasks, including object detection and segmentation.

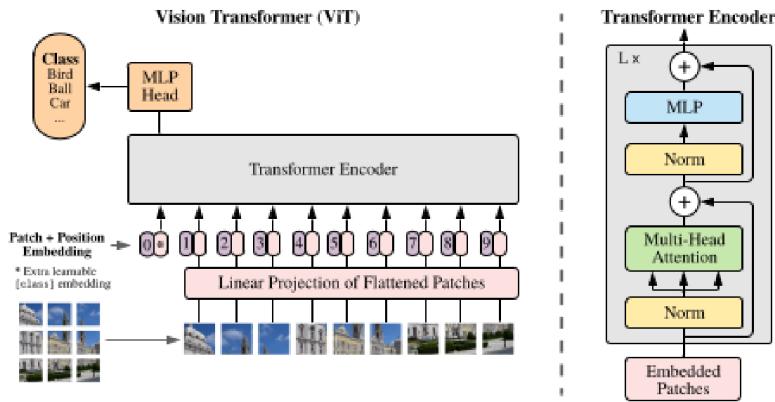


Figure 10.8: Vision Transformer

#### 10.2.5.2 DeTr (Detection Transformer)

Detection Transformer (DeTr) is a transformer-based architecture designed for object detection tasks. It combines the strengths of transformers and CNNs to achieve state-of-the-art performance in object detection. DeTr uses a transformer encoder-decoder architecture to process image features and predict bounding boxes and class labels directly. The key innovation of DeTr is its ability to model relationships between objects in the image using self-attention mechanisms, allowing for better handling of complex scenes and occlusions. This approach eliminates the need for region proposal networks (RPNs) and simplifies the object detection pipeline.

#### DeTr: Detection Transformer (Carion et al., 2020)

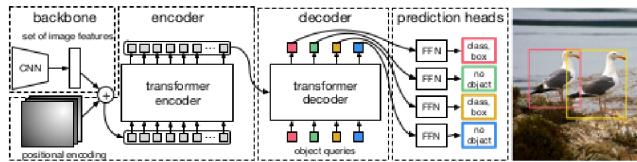


Figure 10.9: Detection Transformer

#### 10.2.5.3 SeTr (Segmentation Transformer)

Segmentation Transformer (SeTr) is a transformer-based architecture designed for semantic segmentation tasks. Similar to ViT, SeTr treats images as sequences of patches and applies self-attention mechanisms to capture relationships between different parts of the image. SeTr has shown competitive performance in various segmentation benchmarks, demonstrating the potential of transformers in computer vision tasks. SeTr uses a transformer encoder-decoder architecture to process image features and predict pixel-wise class labels. The key advantage of SeTr is its ability to model long-range dependencies and relationships between pixels, allowing for better segmentation results in complex scenes.

## 10.3 Transfer Learning

Transfer Learning is a technique in deep learning that allows a model trained on one task to be adapted for another related task. This is particularly useful in scenarios where the target task has limited labeled data or when training a model from scratch is computationally expensive. Transfer learning leverages the knowledge gained from a pre-trained model, which has already learned to extract relevant features from a large dataset. By fine-tuning this model on the target task, we can achieve better performance with less data and training time.

Transfer learning is a powerful technique that allows us to leverage pre-trained models for various tasks, especially when we have limited data or computational resources. It has become a standard practice in deep learning, particularly in computer vision and natural language processing.

### SETR: Segmentation Transformer (Zheng et al., 2021)

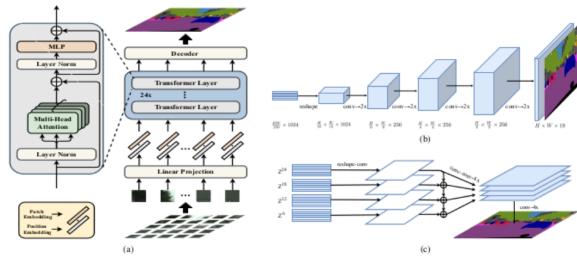


Figure 10.10: Segmentation Transformer

Training from scratch for an image task in DL is difficult

- ◊ High data requirements
- ◊ High computational cost (even with specialised hardware)

Transfer learning tries to alleviate this problem:

- ◊ A generic-trained model is taken as init point
- ◊ The model is changed to fit the task requirements
- ◊ The model is re-trained with data related task
  - Less data
  - Less computational cost

In CNN, basically:

- ◊ Convolutional blocks are kept and classification block is changed
- ◊ Convolutional weights are frozen in the re-training process
- ◊ Optional step (fine tuning): unfroze convolutional weights and re-train

Transfer learning strategy depends on:

- ◊ Data availability
  - ◊ Data similarity
  - ◊ Computational resources
  - ◊ High data available and low similarity: train from scratch
  - ◊ High data available and high similarity: freeze a high number of layers
  - ◊ Low data available and low similarity: freeze a low number of layers
  - ◊ Low data available and high similarity: freeze all convolutional layers
- In case of low computational resources, freeze as much as possible

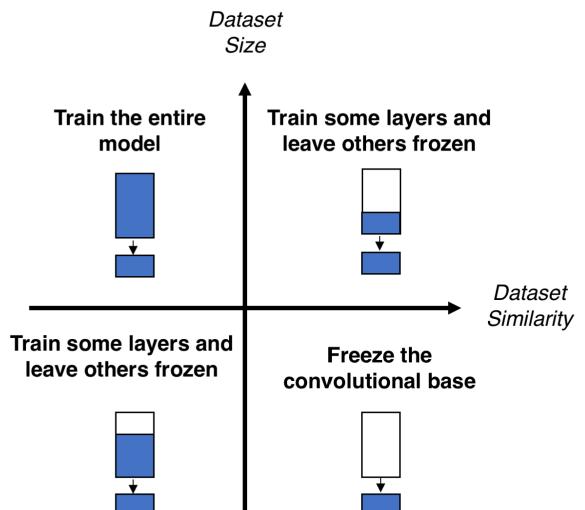


Figure 10.11: Transfer Learning strategy concerning data availability and similarity

