

# Procesamiento de Lenguaje Natural

Apunte de Clases (Borrador)

Felipe Bravo Márquez

Felipe Bravo Márquez

Ilustración Portada por Paulette Filla

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN, UNIVERSIDAD DE CHILE

GITHUB.COM/DCCUCHILE/CC6205

Apuntes de clases del curso de Procesamiento de Lenguaje Natural de la Universidad de Chile.

El formato del apunte fue tomado del template de Jasmine Hao disponible en Overleaf.

*Borrador, 1 de abril de 2024*



## Índice general

<b>Prefacio .....</b>	<b>11</b>
<b>1    Introducción .....</b>	<b>13</b>
1.1    Desafíos del Procesamiento del Lenguaje Natural (PLN)	13
1.2    PLN y Lingüística Computacional	14
1.3    Tareas en Procesamiento del Lenguaje Natural (PLN)	15
1.4    Niveles de descripción lingüística	16
1.4.1    Fonética .....	16
1.4.2    Fonología .....	16
1.4.3    Morfología .....	16
1.4.4    Sintaxis .....	17
1.4.5    Semántica .....	17
1.4.6    Pragmática .....	18
1.5    Aprendizaje Automático en PLN	18
1.5.1    Ejemplo 1: Clasificación de Tópicos .....	19
1.5.2    Ejemplo 2: Análisis de Sentimiento .....	19
1.5.3    Lingüística y Procesamiento del Lenguaje Natural (PNL) .....	21
1.5.4    Limitaciones del Aprendizaje Supervisado .....	21
1.6    Etiquetado de Datos en PLN	21
1.6.1    Supervisión a Distancia .....	22
1.6.2    Crowdsourcing .....	23
1.7    Paradigmas de Aprendizaje Automático	23
1.8    Historia de PLN	24

1.9	Conclusiones y Estructura del Apunte	25
<b>2</b>	<b>Modelo de Espacio Vectorial .....</b>	<b>27</b>
2.1	Tokens y Tipos	27
2.1.1	Tipos .....	27
2.2	Eliminación de Stopwords	28
2.3	Stemming	28
2.4	Lematización	29
2.5	Ley de Zipf	29
2.6	Listas de posteo y el índice invertido	30
2.7	Motores de búsqueda web	30
2.8	El modelo de espacio vectorial	31
2.8.1	El modelo TF-IDF .....	32
2.8.2	Similitud entre vectores .....	33
2.9	Clustering de Documentos	34
2.10	Modelos de Tópicos	35
2.11	Conclusiones y Conceptos Adicionales	36
<b>3</b>	<b>Modelos de Lenguaje Probabilísticos .....</b>	<b>37</b>
3.1	El Problema del Modelado del Lenguaje	37
3.1.1	¿Por qué queríamos hacer esto? .....	38
3.2	Regla de la Cadena en Modelos de Lenguaje	39
3.3	Mirada Predictiva	39
3.4	Mirada Generativa	40
3.5	Un Método Ingenuo	41
3.6	Procesos de Markov	41
3.6.1	Modelado de secuencias de longitud variable .....	41
3.7	Modelos de lenguaje de Trigramas	42
3.8	Evaluación de un modelo de lenguaje: Perplejidad	43
3.8.1	Intuición sobre la perplejidad .....	43
3.8.2	El trade-off entre sesgo y varianza .....	44
3.9	Interpolación Lineal	45
3.9.1	Estimación de los Valores $\lambda$ .....	46
3.10	Modelos de Descuento (Katz Back-Off)	46
3.11	Historia	49
3.12	Conclusiones	50

<b>4</b>	<b>Clasificación de Texto y Naïve Bayes .....</b>	<b>51</b>
4.1	Ejemplos de Problemas de Clasificación	52
4.2	Modelos Generativos	53
4.3	Naïve Bayes Multinomial	54
4.3.1	Estimación de parámetros .....	55
4.3.2	Problemas al multiplicar muchas probabilidades .....	55
4.3.3	Suavizamiento de Laplace .....	56
4.3.4	Naïve Bayes como modelo de lenguaje .....	58
4.4	Evaluación	59
4.4.1	Matriz de Confusión .....	60
4.4.2	Métricas de Desempeño .....	60
4.4.3	Una Medida Combinada: Medida F .....	61
4.4.4	Conjuntos de Prueba de Desarrollo ("Devsets") .....	61
4.4.5	Validación Cruzada: Múltiples Divisiones .....	62
4.5	Conjuntos de entrenamiento, prueba y validación	62
4.5.1	Matriz de Confusión para clasificación de 3 clases .....	63
<b>5</b>	<b>Modelos Lineales .....</b>	<b>65</b>
5.1	Ejemplo: Detección de Idiomas	65
5.2	Clasificación binaria log-lineal	67
5.3	Clasificación multiclas	68
5.4	Representaciones	68
5.5	Representación de Vectores One-Hot	69
5.6	Entrenamiento	70
5.6.1	Optimización basada en Gradiente .....	71
5.6.2	Descenso de Gradiente Estocástico en Línea .....	71
5.6.3	Descenso de Gradiente Estocástico en Mini-batch .....	72
5.6.4	Funciones de Pérdida .....	73
5.7	Regularización	74
5.7.1	Regularización $L_2$ .....	74
5.7.2	Regularización $L_1$ .....	75
5.7.3	Elastic-Net .....	75
5.8	Más allá del SGD	75
5.9	Una limitación de los modelos lineales: el problema XOR	75
5.9.1	Transformaciones no lineales de las entradas .....	76
<b>6</b>	<b>Redes Neuronales .....</b>	<b>79</b>
6.1	Redes neuronales feedforward	79
6.1.1	Redes neuronales como funciones matemáticas .....	80

<b>6.2</b>	<b>Capacidad de representación</b>	<b>81</b>
<b>6.3</b>	<b>Funciones de activación</b>	<b>81</b>
6.3.1	Problemas Prácticos .....	83
<b>6.4</b>	<b>Capas de Embedding</b>	<b>84</b>
6.4.1	Vectores Densos vs Representaciones One-hot .....	84
<b>6.5</b>	<b>Entrenamiento de Redes Neuronales</b>	<b>86</b>
<b>6.6</b>	<b>Recordatorio de la Regla de la Cadena en Derivadas</b>	<b>86</b>
<b>6.7</b>	<b>Retropropagación</b>	<b>88</b>
<b>6.8</b>	<b>La Abstracción del Grafo de Cómputo</b>	<b>91</b>
6.8.1	Cómputo hacia Adelante .....	91
6.8.2	Cómputo hacia Atrás (Retropropagación) .....	92
6.8.3	Resumen de la Abstracción del Grafo de Cómputo .....	92
6.8.4	Derivadas de funciones no matemáticas .....	93
<b>6.9</b>	<b>Regularización y Dropout</b>	<b>93</b>
<b>6.10</b>	<b>Frameworks de Aprendizaje Profundo</b>	<b>94</b>
<b>7</b>	<b>Vectores de Palabra .....</b>	<b>95</b>
<b>7.1</b>	<b>Hipótesis Distribucional y Matrices Palabra Contexto</b>	<b>95</b>
<b>7.2</b>	<b>PPMI</b>	<b>96</b>
<b>7.3</b>	<b>Modelo de Lenguaje Neuronal</b>	<b>97</b>
<b>7.4</b>	<b>Word2Vec</b>	<b>98</b>
7.4.1	Parametrización del modelo Skip-gram .....	100
7.4.2	Skip-gram con Negative Sampling .....	101
7.4.3	Continuous Bag of Words: CBOW .....	101
7.4.4	GloVe .....	102
<b>7.5</b>	<b>Analogías de palabras</b>	<b>102</b>
<b>7.6</b>	<b>Evaluación</b>	<b>102</b>
<b>7.7</b>	<b>Correspondencia entre modelos distribuidos y distribucionales</b>	<b>103</b>
<b>7.8</b>	<b>FastText</b>	<b>104</b>
<b>7.9</b>	<b>Embeddings de frases específicas de sentimiento</b>	<b>104</b>
<b>7.10</b>	<b>Gensim</b>	<b>105</b>
<b>8</b>	<b>Etiquetado de Secuencias .....</b>	<b>107</b>
<b>8.1</b>	<b>Tareas de Etiquetado de Secuencias o Etiquetado</b>	<b>107</b>
<b>8.2</b>	<b>Etiquetado de Partes del Discurso</b>	<b>107</b>
<b>8.3</b>	<b>Reconocimiento de Entidades Nombradas (NER)</b>	<b>108</b>
8.3.1	Etiquetado de secuencias como aprendizaje supervisado .....	109

8.3.2	Enfoque generativo para el etiquetado de secuencias . . . . .	109
<b>8.4</b>	<b>Modelos Ocultos de Markov</b>	<b>109</b>
<b>8.5</b>	<b>Modelos Ocultos de Markov Trigramas (Trigram HMM)</b>	<b>109</b>
8.5.1	Parámetros del modelo . . . . .	110
<b>8.6</b>	<b>Supuestos de independencia en los HMM trigramas</b>	<b>110</b>
<b>8.7</b>	<b>¿Por qué el nombre?</b>	<b>111</b>
<b>8.8</b>	<b>Estimación Suavizada</b>	<b>111</b>
<b>8.9</b>	<b>Tratando con Palabras de Baja Frecuencia</b>	<b>112</b>
<b>8.10</b>	<b>Problema de Decodificación</b>	<b>112</b>
8.10.1	Método Bruto Ingenuo . . . . .	113
<b>8.11</b>	<b>Decodificación de Viterbi con Programación Dinámica</b>	<b>113</b>
<b>8.12</b>	<b>El Algoritmo de Viterbi</b>	<b>114</b>
8.12.1	Una Definición Recursiva . . . . .	115
8.12.2	El Algoritmo de Viterbi . . . . .	116
8.12.3	El Algoritmo de Viterbi con Punteros de Retroceso . . . . .	116
8.12.4	El Algoritmo de Viterbi: Tiempo de Ejecución . . . . .	117
<b>8.13</b>	<b>MEMMs</b>	<b>117</b>
<b>8.14</b>	<b>Ejemplo de características utilizadas en el etiquetado de partes del habla</b>	<b>119</b>
<b>8.15</b>	<b>Plantillas de características</b>	<b>119</b>
<b>8.16</b>	<b>MEMMs y Softmax Multiclas</b>	<b>119</b>
<b>8.17</b>	<b>Entrenamiento de los MEMMs</b>	<b>120</b>
<b>8.18</b>	<b>Decodificación con MEMMs</b>	<b>120</b>
<b>8.19</b>	<b>Comparación entre MEMMs y HMMs</b>	<b>122</b>
<b>8.20</b>	<b>Campos Aleatorios Condicionales (CRFs)</b>	<b>122</b>
<b>8.21</b>	<b>Decodificación con CRFs</b>	<b>124</b>
<b>8.22</b>	<b>Estimación de Parámetros en CRFs (Entrenamiento)</b>	<b>125</b>
<b>8.23</b>	<b>CRFs y MEMMs</b>	<b>125</b>
8.23.1	CRFs y MEMMs: el problema del sesgo de etiqueta . . . . .	125
<b>8.24</b>	<b>Enlaces</b>	<b>126</b>
<b>9</b>	<b>Redes Neuronales Convolucionales</b> . . . . .	<b>127</b>
<b>9.1</b>	<b>Redes Neuronales Convolucionales (CNN) en Procesamiento del Lenguaje Natural (PLN)</b>	<b>127</b>
<b>9.2</b>	<b>Convolución Básica + Agrupamiento</b>	<b>127</b>
<b>9.3</b>	<b>Convoluciones 1D sobre Texto</b>	<b>128</b>

9.4	Convoluciones Angostas vs. Amplias	128
9.5	Agrupamiento Vectorial	129
9.6	Clasificación de Sentimientos en Twitter con CNN	129
9.7	Redes Neuronales Convolucionales Muy Profundas para la Clasificación de Texto	129
<b>10</b>	<b>Redes Neuronales Recurrentes .....</b>	<b>131</b>
10.1	La Abstracción de las RNN	131
10.2	Red Elman o Simple-RNN	133
10.3	Entrenamiento de las RNN	133
10.4	Patrones de uso de las RNN: Aceptador	134
10.5	Patrones de uso de las RNN: Transductor	135
10.6	Redes neuronales recurrentes bidireccionales (BiRNN)	135
10.7	Redes neuronales recurrentes multi-capa (apiladas)	136
10.8	Arquitecturas con compuertas	137
10.9	GRU	140
10.10	Clasificación de sentimientos con RNN	141
10.11	Clasificación de sentimientos en Twitter con LSTMS y Emojis	141
10.12	Bi-LSTM CRF	142
<b>11</b>	<b>Modelos Secuencia a Secuencia .....</b>	<b>145</b>
11.1	Modelos de lenguaje y generación de lenguaje	145
11.2	Problemas de secuencia a secuencia	146
11.2.1	Generación condicionada .....	146
11.3	Enfoques de decodificación	147
11.3.1	Búsqueda Beam .....	147
11.4	Generación condicionada con atención	148
11.5	Atención y alineaciones de palabras	151
11.6	Otros tipos de atención	151
<b>12</b>	<b>Arquitectura de Transformer .....</b>	<b>153</b>
12.0.1	Dependencias en la traducción automática neuronal .....	153
12.1	El Transformer	153
12.2	Mecanismo de atención en el Transformer	154
12.2.1	Consultas .....	155
12.2.2	Claves .....	155

12.2.3	Valores . . . . .	156
12.2.4	Atención de producto puntual escalado . . . . .	156
<b>12.3</b>	<b>El Codificador</b>	<b>157</b>
<b>12.4</b>	<b>Autoatención a alto nivel</b>	<b>159</b>
<b>12.5</b>	<b>Autoatención en detalle</b>	<b>159</b>
<b>12.6</b>	<b>Cálculo matricial de la autoatención</b>	<b>162</b>
<b>12.7</b>	<b>Atención multi-head</b>	<b>163</b>
<b>12.8</b>	<b>Conexiones residuales</b>	<b>165</b>
<b>12.9</b>	<b>El codificador: resumen</b>	<b>165</b>
<b>12.10</b>	<b>El Decodificador</b>	<b>166</b>
<b>12.11</b>	<b>La Capa Lineal Final y el Entrenamiento</b>	<b>167</b>
<b>12.12</b>	<b>Codificaciones posicionales</b>	<b>168</b>
<b>12.13</b>	<b>Conclusiones</b>	<b>169</b>
<b>13</b>	<b>Grandes Modelos de Lenguaje . . . . .</b>	<b>171</b>
<b>13.1</b>	<b>Representaciones para una palabra</b>	<b>171</b>
<b>13.2</b>	<b>Los Modelos de Lenguaje Neurales pueden producir Incrustaciones Contextualizadas</b>	<b>171</b>
<b>13.3</b>	<b>ELMo: Incrustaciones de Modelos de Lenguaje</b>	<b>171</b>
13.3.1	ELMo: Uso con una tarea . . . . .	172
<b>13.4</b>	<b>ULMfit</b>	<b>172</b>
13.4.1	Énfasis de ULMfit . . . . .	173
13.4.2	Transferencia de aprendizaje con ULMfit . . . . .	173
<b>13.5</b>	<b>¡Aumentemos la escala!</b>	<b>173</b>
<b>13.6</b>	<b>BERT (Bidirectional Encoder Representations from Transformers)</b>	<b>173</b>
<b>13.7</b>	<b>Modelado de Lenguaje Mascarado y Predicción de la Siguiente Oración</b>	<b>174</b>
<b>13.8</b>	<b>Codificación de pares de oraciones en BERT</b>	<b>174</b>
<b>13.9</b>	<b>Arquitectura y entrenamiento del modelo BERT</b>	<b>175</b>
<b>13.10</b>	<b>Ajuste fino del modelo BERT</b>	<b>175</b>
13.10.1	Resultados de BERT en tareas GLUE . . . . .	175
13.10.2	Efecto de la tarea de preentrenamiento en BERT . . . . .	176
<b>13.11</b>	<b>Decodificadores de preentrenamiento GPT y GPT-2</b>	<b>176</b>
<b>13.12</b>	<b>¿Qué tipos de cosas aprende el preentrenamiento?</b>	<b>177</b>
<b>13.13</b>	<b>Cambio de fase: GPT-3 (2020)</b>	<b>178</b>
<b>13.14</b>	<b>Aprendizaje sin ejemplos, con un solo ejemplo y con pocos ejemplos con GPT-3</b>	<b>179</b>

<b>13.15 Chain-of-thought Prompting</b>	<b>180</b>
<b>13.16 Modelos de Lenguaje como Asistentes de Usuario (o Chatbots)</b>	<b>180</b>
13.16.1 LaMDA: Modelos de Lenguaje para Aplicaciones de Diálogo .....	181
13.16.2 ChatGPT y RLHF .....	182
13.16.3 GPT-4 (2023) .....	183
<b>13.17 Ajuste Fino de Instrucciones</b>	<b>184</b>
<b>13.18 Línea de tiempo de los Modelos de Lenguaje Grandes</b>	<b>185</b>
<b>13.19 Prompt Engineering</b>	<b>185</b>
<b>13.20 Peligros de los Grandes Modelos de Lenguaje</b>	<b>186</b>
<b>13.21 Conclusiones</b>	<b>187</b>



## Prefacio

Este apunte es el resultado de cinco años de experiencia en la enseñanza del curso de Procesamiento de Lenguaje Natural en la Universidad de Chile<sup>1</sup>. El objetivo es proporcionar una introducción a esta disciplina, centrándose en las técnicas y conceptos fundamentales. Se ha realizado un esfuerzo por lograr un equilibrio entre las técnicas tradicionales, como los modelos de lenguaje de N-gramas, Naive Bayes y los modelos ocultos de Markov (HMM), y los enfoques modernos basados en redes neuronales profundas, como los vectores de palabras, las redes neuronales recurrentes (RNN), los transformers y los grandes modelos de lenguaje. Sin embargo, es importante mencionar que existen temas relevantes que no se incluyen en este material, como el etiquetado de datos, las gramáticas, las técnicas de parsing, así como discusiones sobre tareas más específicas como el question answering.

El contenido se ha recopilado de diversas fuentes. Los temas relacionados con las redes neuronales se basan principalmente en el libro “Neural Network Methods for Natural Language Processing” de Goldberg [Goldberg, 2017]. Los temas no relacionados con las redes neuronales, como los modelos probabilísticos del lenguaje, Naive Bayes y HMM, se han tomado del curso de Michael Collins de Columbia [Collins, 2013] y del borrador de la tercera edición del libro de Dan Jurafsky y James H. Martin [Jurafsky and Martin, 2023]. Además, algunos capítulos se han adaptado de tutoriales en línea y otros cursos, como el curso de Stanford de Christopher Manning<sup>2</sup>. Las imágenes utilizadas se han extraído intencionalmente directamente de sus fuentes correspondientes. Algún día se diseñarán imágenes propias.

En la versión actual sólo los capítulos 1, 2 y 3 se encuentran pulidos. El resto son más bien notas desordenadas.

<sup>1</sup>El repositorio del curso se encuentra en: <https://github.com/dccuchile/CC6205>

<sup>2</sup><http://web.stanford.edu/class/cs224n/>





## 1. Introducción

El volumen de datos textuales digitalizados que se generan a diario a partir de fuentes como la web, las redes sociales, los registros médicos y los libros digitalizados es colosal. Como consecuencia, se ha hecho necesario traducir, analizar, resumir y extraer información de esta avalancha de palabras y texto.

El Procesamiento del Lenguaje Natural (PLN) es el campo que se encarga de diseñar métodos y algoritmos que procesan datos de lenguaje natural, ya sea como entrada o salida [Goldberg, 2017]. El objetivo principal del PLN es desarrollar y analizar algoritmos computacionales y representaciones para procesar el lenguaje humano de manera efectiva [Eisenstein, 2018].

Es importante destacar que el lenguaje natural puede provenir tanto de fuentes escritas como habladas. Aunque el PLN suele centrarse más en el procesamiento de texto, se pueden aplicar técnicas de reconocimiento de habla o transcripción para abordar de manera similar ambas fuentes de información.

El surgimiento de tecnologías de PLN disruptivas como ChatGPT y Google Bard que permiten reescribir oraciones completas, traducir y desarrollar ideas elaboradas a partir de un “prompt” hace aún más necesario el estudio de los fundamentos del área, el objetivo de este apunte es brindar los conocimientos necesarios para comprender el funcionamiento de dichos sistemas.

### 1.1 Desafíos del Procesamiento del Lenguaje Natural (PLN)

A continuación, se discuten varias propiedades del lenguaje humano que hacen extremadamente complejo cumplir con los objetivos del PLN.

#### Ambigüedad

El lenguaje humano es sumamente ambiguo.

■ **Ejemplo 1.1** Tomemos, por ejemplo, las siguientes oraciones:

1. “Yo comí pizza con amigos.”
2. “Yo comí pizza con aceitunas.”
3. “Yo comí pizza con un tenedor.”

Aunque las tres oraciones tienen una estructura gramatical muy similar, difieren en cómo las frases nominales siguientes a la preposición “con” se relacionan con las partes anteriores. En la primera oración, “amigos” modifica al pronombre “yo”; en la segunda, “aceitunas” modifica a la pizza; y en la tercera, “un tenedor” modifica al verbo “comer”. Otra fuente de ambigüedad es la polisemia; palabras con más de un significado.

■ **Ejemplo 1.2** Por ejemplo:

1. “Me senté en el banco”.
2. “Fui al banco a sacar plata”.

En estas dos oraciones “banco” tiene significado distintos. ■

Para poder comprender estas oraciones es necesaria realizar distinciones muy precisas que requieren tener en cuenta el contexto situacional, lo cual puede resultar natural para una persona, pero muy complejo para una máquina.

#### Dinamismo

El lenguaje está en constante cambio y evolución. Siempre surgen nuevas palabras o se les asignan nuevos significados, mientras que otras palabras caen en desuso. Las redes sociales, por ejemplo, pueden acelerar este proceso, como ocurre con los hashtags en Twitter.

#### Discretitud

El lenguaje escrito, como una oración o un documento, se puede entender como una secuencia de palabras discretas provenientes de un vocabulario finito. La naturaleza discreta de las palabras implica que no podemos inferir la relación entre dos palabras basándonos únicamente en las letras que las componen.

■ **Ejemplo 1.3** Por ejemplo, las palabras “hamburguesa”, “pizza” y “tiza”, aunque las dos primeras están más cerca entre sí semánticamente, las dos últimas tienen una mayor similitud en cuanto a las letras que las forman. ■

#### Composicionalidad

El significado de una oración va más allá del significado individual de sus palabras. Esto implica que, aunque tengamos formas de representar computacionalmente el significado de las palabras, esto no garantiza que podamos representar cómo se componen para formar significados a nivel de oración.

#### Dispersión (sparseness)

La forma en que las palabras (símbolos discretos) pueden combinarse para formar significados es prácticamente infinita. Esto implica que, en general, las oraciones que encontramos en un documento son únicas o rara vez han sido escritas antes. Por lo tanto, cualquier enfoque de fuerza bruta que intente memorizar oraciones a partir de una colección de documentos (o corpus) no garantiza una buena generalización a textos nuevos.

## 1.2 PLN y Lingüística Computacional

El Procesamiento del Lenguaje Natural (PLN) a menudo se confunde con otra disciplina relacionada llamada Lingüística Computacional (LC). Aunque están estrechamente vinculadas, tienen enfoques distintos. La LC busca abordar preguntas fundamentales sobre el lenguaje utilizando la

computación, investigando cómo entendemos, producimos y aprendemos lenguaje. En este sentido, la LC se acerca más a la lingüística, cuyo objeto de estudio es el lenguaje humano, apoyándose en métodos computacionales, de manera similar a la biología computacional o la astronomía computacional.

Por otro lado, en el PLN el enfoque está en resolver tareas específicas, como la transcripción automática del habla, la traducción automática, la extracción de información de documentos y el análisis de opiniones en redes sociales. Es importante señalar que en el PLN, el éxito de una solución se mide en función de métricas concretas, como la similitud de una traducción automática con una realizada por un humano, independientemente de si el modelo utiliza alguna teoría lingüística. Si bien los conocimientos lingüísticos fundamentales pueden ser cruciales para llevar a cabo estas tareas, el éxito se evalúa en función de si se logra o no el objetivo establecido, de acuerdo con una métrica de evaluación [Eisenstein, 2018].

### 1.3 Tareas en Procesamiento del Lenguaje Natural (PLN)

El procesamiento del lenguaje natural (PLN) desarrolla métodos para resolver problemas prácticos relacionados con el lenguaje [Johnson, 2014]. A estos problemas se les suele llamar “tareas” o “tasks” en inglés. Cada tarea define formalmente la entrada (input) y salida (output) esperada de un sistema de PLN.

A continuación, se presentan algunos ejemplos de estas tareas junto con sus nombres en inglés:

- **Reconocimiento automático del habla (Speech Recognition):** La entrada es una señal de audio con voz y la salida es texto escrito.
- **Traducción automática (Machine Translation):** La entrada es texto en el idioma fuente y la salida es texto en el idioma destino.
- **Extracción de información de documentos (Information Extraction):** La entrada es texto libre y la salida es una tabla estructurada que contiene la información extraída del texto.
- **Clasificación de texto (Text Classification):** La entrada es texto libre y la salida es la asignación a una categoría discreta dentro de un conjunto finito de categorías.
- **Extracción de Entidades Nombradas (Named Entity Recognition, NER):** La entrada es una oración y la salida es la marcación de las entidades identificadas en la oración, como personas, lugares u organizaciones, como se muestra en la Figura 1.1.
- **Respuestas a Preguntas (Question Answering):** La entrada es una pregunta y la salida es una respuesta.
- **Comprendión de Lectura (Reading Comprehension):** La entrada es un pasaje de texto y una pregunta, y la salida es la ubicación marcada donde se encuentra la respuesta correcta dentro del pasaje proporcionado.
- **Etiquetado Gramatical (Part-of-Speech Tagging):** La entrada es una oración y la salida son las categorías gramaticales (por ejemplo, verbo, sustantivo, adjetivo) de las palabras dentro de la oración.
- **Extracción de Resúmenes (Summarization):** La entrada es un documento y la salida es un párrafo que resume su contenido.
- **Desambiguación de Significado de Palabra (Word Sense Disambiguation):** La entrada es una oración y una palabra objetivo dentro de ella, la salida es una categoría que determina el significado de la palabra dentro de la oración según categorías de significados definidas por un diccionario externo.

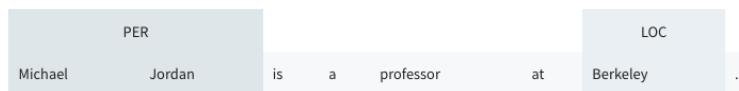


Figura 1.1: Reconocimiento de Entidades Nombradas

## 1.4 Niveles de descripción lingüística

El campo de la **lingüística** aborda el estudio del lenguaje en diferentes niveles de descripción:

- **Fonética y fonología:** estudio de los sonidos del habla.
- **Morfología:** estudio de la estructura de las palabras.
- **Sintaxis:** estudio de la estructura de las oraciones.
- **Semántica:** estudio del significado de las palabras y oraciones.
- **Pragmática:** estudio del uso del lenguaje en el contexto.

Estudiemos estos niveles con un poco más de detalle pues conocerlos tiene utilidad a la hora de diseñar sistemas de PLN como se discutirá más adelante.

### 1.4.1 Fonética

La fonética es la rama de la lingüística que se ocupa del estudio de los sonidos del lenguaje. Examina los órganos utilizados en la producción de sonidos, como la boca, la lengua, la garganta, la nariz, los labios y el paladar. Los sonidos del lenguaje se dividen en vocales y consonantes. Las vocales se producen con poca restricción del flujo de aire desde los pulmones, mientras que las consonantes implican alguna restricción o cierre en el tracto vocal [Johnson, 2014, Fromkin et al., 2018]. Además, el Alfabeto Fonético Internacional (AFI) proporciona una notación alfabética para representar los sonidos fonéticos de todos los idiomas.

### 1.4.2 Fonología

La fonología se centra en el estudio de cómo los sonidos del habla forman patrones y construyen significado. Los fonemas son las unidades básicas de sonido que diferencian el significado de las palabras.

- **Ejemplo 1.4** Por ejemplo, en inglés, la “p” y la “b” son fonemas distintos porque cambian el significado de las palabras en las que se encuentran (piensen las palabras “pat” y “bat”). ■

La fonología también examina las variaciones en la pronunciación de los sonidos en diferentes contextos y dialectos [Fromkin et al., 2018].

### 1.4.3 Morfología

La morfología es el campo de estudio encargado de analizar la estructura interna de las palabras. Los morfemas, que son las unidades mínimas de significado, son los componentes fundamentales de las palabras.

- **Ejemplo 1.5** Por ejemplo, en la palabra “deshacer”, los morfemas presentes son “des-”, “hacer” y “-er”. ■

Además, la morfología examina los procesos de formación de palabras, como la derivación, que implica agregar prefijos o sufijos a una palabra existente para crear una nueva palabra con un

significado diferente [Johnson, 2014].

■ **Ejemplo 1.6** Un ejemplo ilustrativo de la morfología derivativa es la palabra en inglés “revitalization”. En esta palabra, los morfemas “re+vital+ize+ation” se combinan para formar una estructura jerárquica en su morfología derivativa, como se muestra en la Figura 1.2. El sufijo “ation” determina la categoría gramatical (part-of-speech) de la palabra derivada, en este caso, un sustantivo (noun). ■

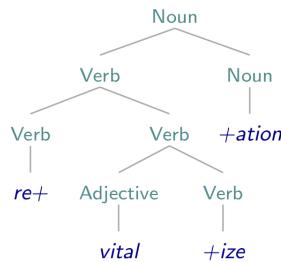


Figura 1.2: Árbol Jerárquico de la Morfología Derivativa

Otra propiedad morfológica importante es la inflexión, que se refiere a los cambios realizados en las palabras para expresar concordancia gramatical, como la conjugación de los verbos o la indicación de género y número en los sustantivos. Por ejemplo, “el perro”, “la perra”, “los perros”, son ejemplos de cómo se aplican las reglas de inflexión en el español para indicar el género y número (singular o plural) en los sustantivos.

#### 1.4.4 Sintaxis

La sintaxis es el estudio de cómo las palabras se combinan para formar frases y oraciones gramaticales. Examina las reglas y estructuras que determinan la organización de las palabras en una oración para un idioma particular y cómo influyen en el significado. La sintaxis también se ocupa de la relación entre las palabras y las funciones que desempeñan dentro de una oración.

■ **Ejemplo 1.7** Por ejemplo, en la oración “The cat chased the dog”, “The cat” es el sujeto, “chased” es el verbo y “the dog” es el complemento directo [Johnson, 2014], tal como muestra el árbol sintáctico en la Figura 1.3. ■

El análisis sintáctico ayuda a identificar **quién hizo qué a quién** en una oración, lo cual es muy útil para comprender su significado.

#### 1.4.5 Semántica

La semántica es el estudio del significado de las palabras, frases y oraciones, examinando cómo se construye e interpreta este significado en el contexto del lenguaje. Además, la semántica se interesa por los roles semánticos, que indican la función de cada entidad en una oración.

■ **Ejemplo 1.8** Algunos ejemplos de roles semánticos son: **agente** (la entidad que realiza la acción), **tema** (la entidad involucrada en la acción) y **instrumento** (otra entidad utilizada por el agente para llevar a cabo la acción). En la oración “El niño cortó la cuerda con un cuchillo”, se puede identificar el agente como **el niño**, el tema como **la cuerda** y el instrumento como **un cuchillo** [Johnson, 2014]. ■

Además de los roles semánticos, la semántica también se ocupa de las relaciones léxicas (o semántica léxica), que son las conexiones entre distintas palabras [Yule, 2016].

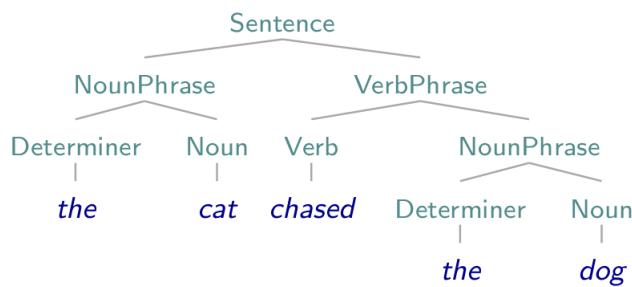


Figura 1.3: Árbol Sintáctico

- **Ejemplo 1.9** Algunos ejemplos de estas relaciones incluyen la sinonimia, que se refiere a palabras con significados similares como “esconder” y “ocultar”; la antonimia, que involucra palabras con significados opuestos como “alto” y “bajo”; y la hipónimia, que establece una relación entre una palabra más específica y otra más general, como “perro” y “animal”. Además, existe la hiperónimia, que representa la relación contraria. ■

#### 1.4.6 Pragmática

La pragmática se centra en cómo el contexto influye en la interpretación y el significado de las expresiones lingüísticas. Examina cómo se utilizan las expresiones lingüísticas en situaciones reales y cómo los hablantes interpretan el significado implícito.

- **Ejemplo 1.10** Por ejemplo, la oración “Hace frío aquí” puede interpretarse como una sugerencia implícita para cerrar las ventanas [Fromkin et al., 2018]. ■

### 1.5 Aprendizaje Automático en PLN

Dotar a los computadores de habilidades para comprender y producir el lenguaje humano es extremadamente complejo. La tecnología más exitosa actualmente para abordar PLN es el aprendizaje automático supervisado, que consiste en una familia de algoritmos que “aprenden” a construir la respuesta del problema en cuestión en base a encontrar patrones en datos de entrenamiento etiquetados. La esencia del aprendizaje automático supervisado es la creación de mecanismos que puedan examinar ejemplos y producir generalizaciones [Goldberg, 2017]. Diseñamos un algoritmo cuya entrada es un conjunto de ejemplos etiquetados y cuya salida es una función (o un programa) que recibe una instancia y produce la etiqueta deseada.

Por ejemplo, si la tarea es distinguir entre correos electrónicos de spam y no spam, los ejemplos etiquetados serían correos electrónicos etiquetados como spam y correos electrónicos etiquetados como no spam. Se espera que la función resultante produzca predicciones de etiquetas correctas también para instancias que no ha visto durante el entrenamiento.

- **Ejemplo 1.11** A continuación se muestra un ejemplo etiquetado tanto para la tarea de etiquetado gramatical como la de extracción de entidades nombradas del dataset NER CoNLL-2003<sup>1</sup>. Cada

<sup>1</sup>Fuente: <https://www.clips.uantwerpen.be/conll2003/ner/>

Línea contiene un token, una etiqueta de categoría gramatical, una etiqueta de sintagma y una etiqueta de entidad nombrada.

U.N.	NNP	I-NP	I-ORG
official	NN	I-NP	O
Ekeus	NNP	I-NP	I-PER
heads	VBZ	I-VP	O
for	IN	I-PP	O
Baghdad	NNP	I-NP	I-LOC
.	.	O	O

■

A continuación desarrollamos de forma más concreta el uso del aprendizaje automático para tareas de PLN mediante dos ejemplos.

### 1.5.1 Ejemplo 1: Clasificación de Tópicos

La clasificación de tópicos es una tarea específica de la clasificación de documentos, en la cual se asigna a cada documento una de varias categorías predefinidas, como deportes, política, farándula o economía. La idea fundamental es que las palabras presentes en los documentos pueden ser indicativas del tema que abordan. Sin embargo, crear reglas manuales para esta tarea resulta desafiante debido a la complejidad del lenguaje. La anotación de datos, en la cual personas etiquetan manualmente una colección de documentos según su tema, puede ser de gran ayuda para generar conjuntos de datos de entrenamiento utilizados por algoritmos de aprendizaje automático supervisado. Estos algoritmos aprenden patrones de uso de palabras que facilitan la categorización de los documentos y suelen ser más robustos que las reglas construidas de forma manual.

### 1.5.2 Ejemplo 2: Análisis de Sentimiento

El análisis de sentimientos se refiere a la aplicación de técnicas PLN para identificar y extraer información subjetiva de conjuntos de datos textuales. Un desafío común en el análisis de sentimientos es la clasificación de la polaridad a nivel de mensaje (MPC), donde las oraciones se clasifican automáticamente en categorías positivas, negativas o neutrales. Las soluciones más avanzadas utilizan modelos de aprendizaje automático supervisado entrenados con ejemplos anotados manualmente.

Una aplicación concreta del análisis de sentimiento es construir un modelo que nos diga si un tweet tiene un sentimiento positivo o negativo respecto a un producto. Para resolver el problema con aprendizaje supervisado primero necesitamos etiquetar manualmente un conjunto de tweets con su sentimiento asociado. Luego debemos entrenar un algoritmo de aprendizaje utilizando estos datos para poder predecir de manera automática el sentimiento asociado a tweets desconocidos. Como podrán imaginar, el etiquetado de datos es una parte fundamental de la solución y puede ser un proceso muy costoso, especialmente cuando se requiere conocimiento especializado para definir la etiqueta.

En este tipo de clasificación, es común transformar las oraciones en vectores de características (cada característica es una columna o dimensión del vector) y aplicar modelos lineales como las Máquinas de Vectores de Soporte (SVM), como se ilustra en la Figura 1.4. Una forma habitual de representar las oraciones o documentos en forma vectorial es mediante el enfoque de la bolsa-de-palabras, donde cada documento se representa como un vector con una columna por cada palabra

identificada en el conjunto de documentos (o corpus). Si una palabra está presente en un documento, se asigna un valor distinto de cero en la columna correspondiente, ya sea 1 o la frecuencia de la palabra en el documento. En caso de que la palabra no esté presente, se asigna un valor cero en esa columna.

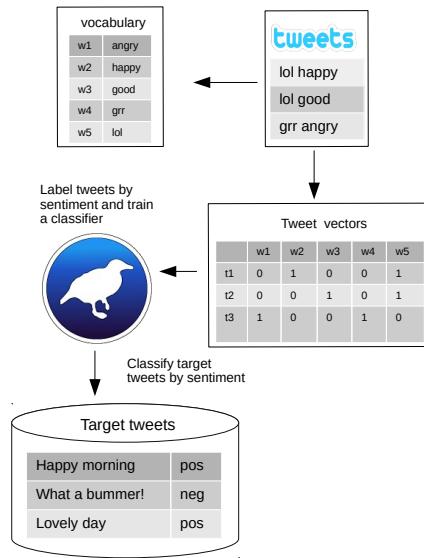


Figura 1.4: Proceso de Clasificación de Sentimiento en Tweets con Vectores Bolsa-de-Palabras.

El objetivo de las SVM es encontrar un hiperplano que separe las representaciones vectoriales de las clases con el margen máximo, logrando la mejor separación entre las clases positivas, negativas y neutrales [Eisenstein, 2018], tal como se ilustra en la Figura 1.5. Una vez encontrado el hiperplano, se puede utilizar para clasificar nuevos tweets en categorías de sentimiento proyectándolos primero en el espacio vectorial de la bolsa-de-palabras y luego determinando en qué lado del hiperplano se encuentra el vector resultante. De esta manera, se asigna una etiqueta de sentimiento (positivo, negativo o neutral) al tweet en base a su posición relativa al hiperplano.

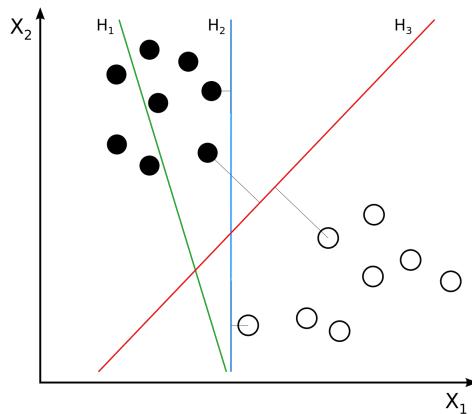


Figura 1.5: SVM, el hiperplano  $H_3$  separa las clases con el margen máximo.

### 1.5.3 Lingüística y Procesamiento del Lenguaje Natural (PLN)

El conocimiento de las estructuras lingüísticas es fundamental para el diseño de características y el análisis de errores en el Procesamiento del Lenguaje Natural (PLN). Los enfoques de aprendizaje automático en PLN se basan en características que describen y generalizan las instancias de uso del lenguaje. El conocimiento lingüístico orienta la selección y el diseño de estas características, ayudando al algoritmo de aprendizaje automático a encontrar correlaciones entre el uso del lenguaje y las etiquetas objetivo [Bender, 2013].

### 1.5.4 Limitaciones del Aprendizaje Supervisado

Si bien el aprendizaje automático puede resolver satisfactoriamente muchas tareas de Procesamiento del Lenguaje Natural (PLN), no está exento de limitaciones. En primer lugar, la anotación manual de datos requiere recursos humanos y tiempo de trabajo que no están al alcance de todas las organizaciones. Además, los modelos de aprendizaje supervisado pueden tener dificultades para generalizar correctamente a datos que difieren significativamente de los datos de entrenamiento, lo que se conoce como variación de dominio. Para comprender mejor este problema, consideremos las siguientes dos oraciones en el contexto de la clasificación de sentimientos:

1. Para mí, la cola era bastante **pequeña** y solo tuve que esperar unos 20 minutos, ¡pero valió la pena! :D @raynwise
2. Extraña espacialidad en Stuttgart. La habitación del hotel es tan **pequeña** que apenas puedo moverme, pero los alrededores son inhumanamente vastos y largos bajo construcción.

Podemos observar que la palabra “pequeña” tiene una connotación positiva en el contexto de una cola de un banco, pero una connotación negativa en referencia al tamaño de una habitación de hotel. Por lo tanto, un modelo entrenado únicamente con datos de reseñas de hoteles podría tener un mal desempeño al aplicarlo a contextos relacionados con colas de un banco.

Además, los modelos de PLN pueden volverse obsoletos a medida que el uso del lenguaje evoluciona con el tiempo. Por ejemplo, un modelo de clasificación de sentimientos entrenado antes de la aparición de los emojis no tendría en cuenta esta valiosa información. Como resultado, los modelos deben ser monitoreados constantemente y, en muchos casos, reentrenados con nuevos datos para funcionar correctamente en sus entornos de aplicación.

## 1.6 Etiquetado de Datos en PLN

La construcción de conjuntos de datos de entrenamiento y evaluación para modelos de aprendizaje automático en Procesamiento del Lenguaje Natural (PLN) requiere un enfoque cuidadoso para asegurar que los modelos aprendan a resolver la tarea objetivo de manera efectiva.

En primer lugar, es necesario obtener un corpus (o colección) de documentos objetivo que sean representativos de la tarea en cuestión. A continuación, se debe establecer una guía de anotación para los etiquetadores. Por ejemplo, si la tarea es la clasificación de sentimientos, se recopilan tweets (corpus) y se define una guía que solicita a los anotadores que determinen el sentimiento del autor del tweet en base a tres categorías (positivo, negativo, neutral).

Luego, se reclutan anotadores que deben ser independientes y tener suficientes conocimientos en el lenguaje y la tarea para realizar etiquetados consistentes. Esto suele llevarse a cabo a través de plataformas en línea especializadas.

Los anotadores etiquetan los documentos de manera sistemática y suelen ser evaluados de forma continua. Para evaluar su desempeño, se suelen pre-etiquetar algunos ejemplos con el fin de determinar si los anotadores comprenden correctamente el objetivo de la anotación.

A continuación, se comparan las anotaciones de los distintos etiquetadores utilizando criterios de concordancia inter-anotador (inter-annotator agreement). Para problemas de clasificación, dos métricas comunes son el coeficiente Kappa de Cohen y el coeficiente Kappa de Fleiss.

El coeficiente Kappa de Cohen se calcula utilizando la siguiente fórmula:

$$K = \frac{P_o - P_e}{1 - P_e}$$

donde  $P_o$  es la proporción observada de acuerdo entre los anotadores y  $P_e$  es la proporción esperada de acuerdo por azar. El coeficiente Kappa de Cohen mide la proporción de acuerdo entre los anotadores que es superior al acuerdo esperado al azar. Un valor de  $K$  cercano a 1 indica un buen acuerdo entre los anotadores, mientras que un valor cercano a 0 indica un acuerdo similar al que se podría esperar al azar.

El coeficiente Kappa de Fleiss también se utiliza para medir la concordancia inter-anotador en problemas de clasificación con múltiples anotadores. A diferencia del coeficiente Kappa de Cohen, el coeficiente Kappa de Fleiss tiene en cuenta el acuerdo más allá de dos anotadores y se calcula de la siguiente manera:

$$K = \frac{\overline{P}_o - \overline{P}_e}{1 - \overline{P}_e}$$

donde  $\overline{P}_o$  es la proporción media observada de acuerdo entre los anotadores y  $\overline{P}_e$  es la proporción media esperada de acuerdo por azar.

Finalmente, se lleva a cabo un proceso de consolidación de las anotaciones. Por ejemplo, si se cuenta con tres anotadores, se puede utilizar una regla de mayoría simple para asignar las etiquetas finales a los datos. En algunos casos, los ejemplos en los que hay desacuerdo pueden ser etiquetados nuevamente por anotadores más experimentados, mientras que en otros casos se eliminan del conjunto de datos. Todas estas decisiones pueden tener consecuencias en la calidad de los conjuntos de datos y, en consecuencia, en el rendimiento de los modelos entrenados con estos datos.

Al conjunto de datos resultante del proceso de consolidación se le llama “gold standard” o “ground truth” y se utiliza posteriormente para entrenar modelos de aprendizaje automático.

Referencias recomendadas en anotación de datos para PLN se encuentran en los libros [Fort, 2016] y [Pustejovsky and Stubbs, 2012].

### 1.6.1 Supervisión a Distancia

En algunos casos, es posible etiquetar datos de manera semi-automática con una técnica llamada supervisión a distancia, lo que permite ahorrar costos de anotación. Un ejemplo de esto es el enfoque de anotación de emoticones utilizado en la clasificación de sentimientos en tweets. En este enfoque, se utiliza la API de Twitter para recopilar tweets que contengan emoticones positivos :)) o negativos :(, y luego se etiquetan los tweets según la polaridad indicada por el emoticón [Read, 2005]. El emoticón se **elimina** del contenido del tweet. Este enfoque también se ha ampliado utilizando hashtags como #rabia y emojis. Sin embargo, no es trivial encontrar técnicas de supervisión a distancia que sean aplicables a todos los tipos de problemas en PLN.

### 1.6.2 Crowdsourcing

Las plataformas de crowdsourcing, como **Amazon Mechanical Turk (AMT)**<sup>2</sup>, ofrecen la posibilidad de etiquetar datos a gran escala mediante el pago a etiquetadores remotos. Si bien este enfoque permite la anotación masiva de datos, es difícil garantizar la calidad esperada por parte de los anotadores. Sin embargo, ha sido gracias a este enfoque que se ha logrado la proliferación de numerosos conjuntos de datos para entrenar modelos supervisados en PLN. En [Snow et al., 2008] se realiza una comparación exhaustiva sobre la calidad de las anotaciones obtenidas con AMT para diversas tareas de PLN. Los resultados muestran que las anotaciones obtenidas son comparables a las de los anotadores expertos.

## 1.7 Paradigmas de Aprendizaje Automático

Hasta el año 2010, el Procesamiento del Lenguaje Natural (PLN) se basaba en modelos de aprendizaje poco profundos y en características manuales. Por ejemplo, en 2013, el taller de Evaluación Semántica (SemEval) organizó la tarea de "Análisis de sentimientos en Twitter" [Nakov et al., 2013]. Esta tarea se dividió en dos sub-tareas: el nivel de expresión y el nivel del mensaje. El nivel de expresión se centró en determinar la polaridad del sentimiento de un mensaje según una entidad marcada dentro de su contenido, mientras que el nivel del mensaje buscaba determinar la polaridad según el mensaje en general. Los organizadores proporcionaron conjuntos de datos de entrenamiento y prueba para ambas tareas [Nakov et al., 2013].

El equipo que logró el mejor rendimiento en ambas tareas, entre 44 equipos participantes, fue el equipo llamado *NRC-Canada* [Mohammad et al., 2013]. Este equipo propuso un enfoque supervisado utilizando un clasificador SVM lineal y características hechas a mano para representar los tweets. Algunas de estas características fueron:

1. N-gramas de palabras (similares a los vectores bolsa-de-palabras, pero con secuencias contiguas de  $n$  palabras).
2. N-gramas de caracteres (similar a lo anterior, pero con secuencias contiguas de  $n$  caracteres).
3. Etiquetas de categorías gramaticales.
4. Clusters de palabras entrenadas con el método de Brown [Brown et al., 1992].
5. El número de palabras alargadas (palabras con un carácter repetido más de dos veces).
6. El número de palabras con todas las letras en mayúscula.
7. La presencia de emoticones positivos o negativos.
8. El número de negaciones individuales.
9. El número de secuencias contiguas de puntos, signos de interrogación y signos de exclamación.
10. Características derivadas de lexícones de polaridad [Mohammad et al., 2013] (listas de palabras con sentimiento asociado). Dos de estos lexícones se generaron utilizando el método PMI (point-wise mutual information) a partir de tweets anotados con hashtags y emoticones.

Cabe destacar que todas estas características se concatenan para crear un único vector por cada tweet, generando una representación de alta dimensión.

Hasta el año 2014, la mayoría de los sistemas del estado-del-arte en PLN se basaban en el paradigma de ingeniería de características (diseño manual de vectores de características) junto con modelos de aprendizaje automático superficiales, como SVM y CRF. Diseñar las características de un sistema de PLN ganador requería un amplio conocimiento específico del dominio. El sistema

<sup>2</sup><https://www.mturk.com/>

desarrollado por el equipo NRC-Canada se construyó antes de que el aprendizaje profundo (o deep learning) se volviera popular en el campo del PLN.

Por otro lado, los sistemas de aprendizaje profundo han revolucionado el campo del PLN. Estos sistemas se basan en redes neuronales profundas para aprender automáticamente buenas representaciones utilizando técnicas como los vectores de palabras (word embeddings) y arquitecturas especializadas como las redes neuronales recurrentes y los Transformers. Estos modelos requieren grandes volúmenes de datos y un alto poder computacional para su entrenamiento efectivo. Las grandes cantidades de datos textuales disponibles en los últimos años, junto con los procesadores de la tarjeta gráfica de múltiples núcleos (GPU y TPU) más rápidos, han sido fundamentales para el éxito del aprendizaje profundo en PLN.

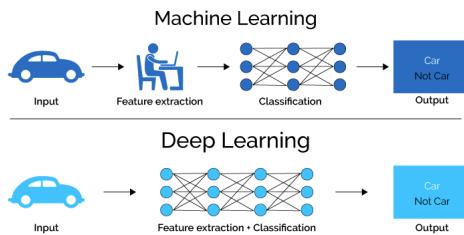


Figura 1.6: Ingeniería de Características vs Aprendizaje Profundo

### Aprendizaje Profundo y Conceptos Lingüísticos

Si los modelos de aprendizaje profundo pueden aprender representaciones automáticamente, ¿siguen siendo útiles los conceptos lingüísticos (por ejemplo, sintaxis, morfología)? Algunos defensores del aprendizaje profundo argumentan que estas propiedades lingüísticas inferidas y diseñadas manualmente no son necesarias, y que la red neuronal aprenderá estas representaciones intermedias (o equivalentes o mejores) por sí misma [Goldberg, 2016]. Aún no hay un consenso definitivo al respecto. Goldberg cree que muchos de estos conceptos lingüísticos pueden ser inferidos por la red por sí misma si se le proporciona suficiente cantidad de datos. Sin embargo, en muchos otros casos no disponemos de suficientes datos de entrenamiento para la tarea que nos interesa, y en estos casos proporcionar a la red los conceptos generales más explícitos puede ser muy valioso.

## 1.8 Historia de PLN

Los orígenes de PLN se remontan a los años 50 con el famoso test de Alan Turing: una máquina será considerada inteligente cuando sea capaz de conversar con una persona sin que esta pueda determinar si está hablando con una máquina o un ser humano. A lo largo de su historia, la disciplina ha experimentado tres grandes períodos: el racionalismo, el empirismo y el aprendizaje profundo [Deng and Liu, 2018].

El período del racionalismo abarcó desde 1950 hasta 1990, donde las soluciones consistían en diseñar reglas manuales para incorporar mecanismos de conocimiento y razonamiento. Un ejemplo emblemático de esta época es ELIZA, un agente de conversación (o chatbot) desarrollado por Joseph Weizenbaum, que simulaba ser un psicoterapeuta rogeriano. También se destaca en esa época el sistema MARGIE, utilizado para estructurar información del mundo real en ontologías de conceptos.

A partir de la década de los 90, el enfoque de PLN se inclinó hacia el empirismo, con el diseño de métodos estadísticos y de aprendizaje automático construidos sobre corpus de datos etiquetados.

En este período, las reglas ya no se construían manualmente, sino que se “aprendían” a partir de los datos. Algunos modelos representativos de esta época son los filtros de spam basados en modelos Bayesianos, las cadenas de Markov ocultas para la extracción de categorías sintácticas y los modelos probabilísticos de IBM para la traducción automática. Estos modelos se caracterizaban por ser poco profundos en su estructura de parámetros y dependían de características manualmente diseñadas para representar la entrada.

A partir del año 2010, las redes neuronales artificiales, que son una familia de modelos de aprendizaje automático, comenzaron a mostrar resultados sobresalientes en varias tareas emblemáticas de PLN [Collobert et al., 2011]. Estos modelos utilizan una jerarquía de parámetros (o capas) para representar la entrada (texto) y encontrar representaciones adecuadas para la tarea en cuestión, en lo que se conoce como “aprendizaje profundo”. Se caracterizan por tener muchos más parámetros que los modelos anteriores, superando la barrera del millón en algunos casos, y requieren grandes volúmenes de datos para su entrenamiento. Estos modelos pueden ser pre-entrenados con texto no etiquetado, como libros, Wikipedia, textos de redes sociales y de la web, para encontrar representaciones iniciales de palabras y oraciones (conocidas como word embeddings), que luego pueden ser adaptadas para la tarea específica utilizando datos etiquetados (proceso conocido como fine-tuning). Entre estos modelos se destacan Word2Vec [Mikolov et al., 2013], BERT [Kenton and Toutanova, 2019], GPT-3 [Brown et al., 2020] y otros grandes modelos de lenguaje que han surgido recientemente (ChatGPT, Google Bard).

Estos modelos han ido perfeccionándose en los últimos años, logrando resultados cada vez mejores en casi todos los problemas del área. Sin embargo, este progreso no ha estado exento de controversias. El aumento exponencial en la cantidad de parámetros de cada nuevo modelo en comparación con su predecesor ha hecho que la construcción de estos modelos requiera recursos computacionales y energéticos que solo están al alcance de unos pocos. Además, varios estudios han demostrado que estos modelos aprenden y reproducen sesgos y prejuicios presentes en los textos utilizados para su entrenamiento, como los relacionados con género, religión y raza. Un ejemplo destacado es el despido de la investigadora Timnit Gebru de Google, luego de que se le negara el permiso para publicar un artículo que revelaba estos problemas [Bender et al., 2021].

## 1.9 Conclusiones y Estructura del Apunte

En este capítulo, hemos explorado el desafío de comprender y generar lenguaje utilizando la computación. El aprendizaje automático supervisado se ha destacado como una de las principales técnicas utilizadas para abordar este desafío. También discutimos las limitaciones y desafíos que enfrenta el Procesamiento del Lenguaje Natural (PLN), como la anotación de datos, la generalización a nuevos dominios y la evolución del lenguaje, entre otros aspectos que requieren atención. Finalmente, hemos visto cómo el aprendizaje profundo ha demostrado mejoras significativas en el rendimiento de los modelos de PLN. Estos modelos basados en redes neuronales han logrado superar barreras anteriores al ser capaces de capturar patrones más complejos y aprender representaciones más ricas del lenguaje.

El resto de este apunte se estructura de la siguiente manera: en el Capítulo 2 se presenta el modelo de espacio vectorial junto a la disciplina hermana de PLN: la recuperación de información. En el Capítulo 3 se discuten los modelos de lenguaje probabilísticos basados en n-gramas. En el Capítulo 4, formalizamos la tarea de clasificación de texto e introducimos el modelo naïve bayes. Luego, en el Capítulo 5, se presentan los modelos lineales en PLN, y en el Capítulo 6, se introducen las redes neuronales. Los vectores de palabras o “word embeddings” se discuten en el Capítulo 7.

Las tareas de etiquetado de secuencia, como el etiquetado gramatical y la extracción de entidades nombradas, se formalizan en el Capítulo 8, junto con modelos clásicos para esta tarea, como las cadenas de Markov ocultas y los Conditional Random Fields.

A partir del Capítulo 9, se discuten arquitecturas especializadas de redes neuronales para PLN. Las redes convolucionales se tratan en el Capítulo 9, las recurrentes en el Capítulo 10, los modelos secuencia a secuencia, junto con los modelos de atención, en el Capítulo 11, y finalmente, la arquitectura de Transformer se desarrolla en el Capítulo 12.

Cerramos el apunte en el Capítulo 13, donde se presentan modelos modernos de vectores de palabras contextualizados y los grandes modelos de lenguaje.



## 2. Modelo de Espacio Vectorial

En este capítulo, exploraremos las primeras técnicas utilizadas para representar el lenguaje en forma vectorial. Para contextualizar esto, planteamos las siguientes preguntas:

- ¿Cómo recuperan los motores de búsqueda, como Duckduckgo o Google, los documentos relevantes a partir de una consulta dada?
- ¿Cómo pueden las empresas procesar las reclamaciones dejadas por sus usuarios en sus portales web?

Estas preguntas suelen abordarse desde dos disciplinas relacionadas:

- *Recuperación de Información*: ciencia que se ocupa de buscar información en colecciones de documentos.
- *Minería de Texto*: extracción automática de conocimiento a partir de texto.

Ambas disciplinas están estrechamente vinculadas al PLN, y las fronteras entre estos campos no siempre están claras.

### 2.1 Tokens y Tipos

El primer paso en el procesamiento de texto es, generalmente, la **tokenización**, que consiste en dividir una oración o documento en fragmentos llamados *tokens*. A menudo, se acompañan otras transformaciones adicionales, como la eliminación de caracteres especiales (por ejemplo, puntuación), la conversión de todo el texto a minúsculas y otras operaciones descritas en este capítulo.

- **Ejemplo 2.1** Entrada: “Me gustan los lenguajes humanos y los lenguajes de programación.”  
Tokens: [Me] [gustan] [los] [lenguajes] [humanos] [y] [los] [lenguajes] [de] [programación]

#### 2.1.1 Tipos

Un *tipo* es una clase de *token* que contiene una secuencia única de caracteres. Los tipos se obtienen identificando los tokens únicos dentro del documento.

- **Ejemplo 2.2** Por ejemplo, para la oración anterior, los tipos serían: [Me] [gustan] [los] [lenguajes] [humanos] [y] [de] [programación]. Observemos que el token *lenguajes* se incluye solo una vez en la lista de tipos. ■

### Extracción de Vocabulario

Un *término* es un *tipo* normalizado. La normalización implica la creación de clases de equivalencia para diferentes *tipos*, por ejemplo, cuando tratamos de manera indistinta dos tokens que difieren solo en el uso de mayúsculas (hombre y Hombre). El vocabulario *V* es el conjunto de términos (tokens únicos normalizados) dentro de una colección de documentos o corpus *D*.

## 2.2 Eliminación de Stopwords

Con el fin de reducir el tamaño del vocabulario y eliminar términos que no aportan mucha información, se eliminan los términos que ocurren con alta frecuencia en el corpus (o colección de documentos). Estos términos se llaman *stopwords* e incluyen artículos, pronombres, preposiciones y conjunciones.

- **Ejemplo 2.3** Ejemplo de stopwords: [un, una, y, cualquier, no, el, en]. ■

Es importante tener en cuenta que la eliminación de stopwords puede ser inconveniente en muchas tareas de procesamiento del lenguaje natural.

- **Ejemplo 2.4** Por ejemplo, en la oración “No me gusta la pizza”, al eliminar la negación, se pierde información valiosa sobre el sentimiento de la oración. ■

## 2.3 Stemming

Es un proceso de normalización de términos en el cual los términos se transforman a su raíz con el objetivo de reducir el tamaño del vocabulario. Se lleva a cabo aplicando reglas de reducción de palabras. La Tabla 2.1 muestra algunas reglas del algoritmo de Porter para el inglés con ejemplos de aplicación:

Regla	Ejemplo
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

Cuadro 2.1: Ejemplos del Algoritmo de Porter

- **Ejemplo 2.5** A continuación se muestra ejemplo usando el método de stemming Snowball para español para el lenguaje **python** usando la biblioteca **nltk**<sup>1</sup>.

```
from nltk import word_tokenize
from nltk.stem import SnowballStemmer
stemmer = SnowballStemmer('spanish')
texto = 'Me gustan los lenguajes humanos y los lenguajes de programación'
```

<sup>1</sup><https://www.nltk.org/>

```

def stem_sentence(text):
    return ' '.join([stemmer.stem(i) for i in word_tokenize(text)])
print(stem_sentence(texto))
>>
me gust los lenguaj human y los lenguaj de program

```

■

## 2.4 Lematización

Otra estrategia de normalización de términos. También transforma las palabras en sus raíces. Realiza un análisis morfológico utilizando diccionarios de referencia (tablas de búsqueda) para crear clases de equivalencia entre *tipos*. Por ejemplo, para el token *yendo*, una regla de stemming devolvería el término *yend*, mientras que a través de la lematización obtendríamos el término *ir*.

■ **Ejemplo 2.6** Veamos a continuación el resultado de aplicar lematización a la misma oración anterior con la biblioteca **spacy**<sup>2</sup>:

```

# python -m spacy download es_core_news_sm
import spacy
nlp = spacy.load('es_core_news_sm')

def lemmatizer(text):
    doc = nlp(text)
    return ' '.join([word.lemma_ for word in doc])

print(lemmatizer(texto))
>>
yo gustar el lenguaje humano y el lenguaje de programación

```

■

## 2.5 Ley de Zipf

La Ley de Zipf, propuesta por George Kingsley Zipf en [Zipf, 1935], es una ley empírica que describe la frecuencia de los términos en una colección de documentos (corpus). Según esta ley, la frecuencia  $f$  de un término en un corpus es inversamente proporcional a su posición  $r$  en una tabla de frecuencias ordenada:

$$f = \frac{c}{r^\beta} \quad (2.1)$$

Aquí,  $c$  es una constante dependiente de la colección y  $\beta > 0$  es un factor de decaimiento. Cuando  $\beta = 1$ , la frecuencia sigue exactamente la Ley de Zipf, de lo contrario, sigue una distribución similar a la de Zipf. Esta ley nos indica que algunas palabras se utilizan con mucha más frecuencia que otras en un corpus. La Ley de Zipf se clasifica como una distribución de ley de potencia, que es un tipo de distribución de cola larga.

---

<sup>2</sup><https://spacy.io/>

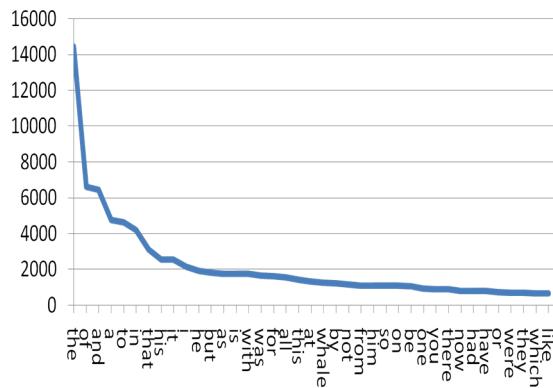


Figura 2.1: Ley de Zipf.

Cuando trazamos un gráfico en escala logarítmica (*log-log*), se obtiene una línea recta con una pendiente de  $-\beta$ . Esta relación logarítmica es útil para identificar las palabras más frecuentes en un corpus y construir una lista de stopwords.

## 2.6 Listas de posteo y el índice invertido

Sea  $D$  una colección de documentos y  $V$  el vocabulario de todos los términos extraídos de la colección:

**Definición 2.6.1** La lista de posteo de un término es la lista de todos los documentos donde el término aparece al menos una vez. Los documentos se identifican por sus identificadores.

**Definición 2.6.2** Un índice invertido es una estructura de datos tipo diccionario que mapea los términos  $t_i \in V$  con sus listas de posteo correspondientes.

$<\text{término}> \rightarrow <\text{idDocumento}>^*$

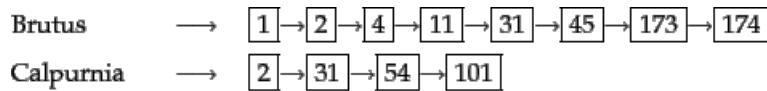


Figura 2.2: Índice invertido

## 2.7 Motores de búsqueda web

Un motor de búsqueda es un sistema de recuperación de información diseñado para buscar información en la web (satisfacer necesidades de información) [Manning et al., 2008]. Sus componentes básicos (ilustrados en la Figura 2.3) son:

- Crawler: un robot que navega por la web según una estrategia definida. Por lo general, comienza navegando por un conjunto de sitios web iniciales (semillas) y continúa navegando a través de sus enlaces.
- Indexador: se encarga de mantener un índice invertido con el contenido de las páginas recorridas por el crawler.

- Procesador de consultas: se encarga de procesar las consultas de los usuarios y buscar en el índice los documentos más relevantes para una consulta.
- Función de ranking: la función utilizada por el procesador de consultas para ordenar los documentos indexados en la colección por relevancia según una consulta.
- Interfaz de usuario: recibe la consulta como entrada y devuelve los documentos ordenados por relevancia.

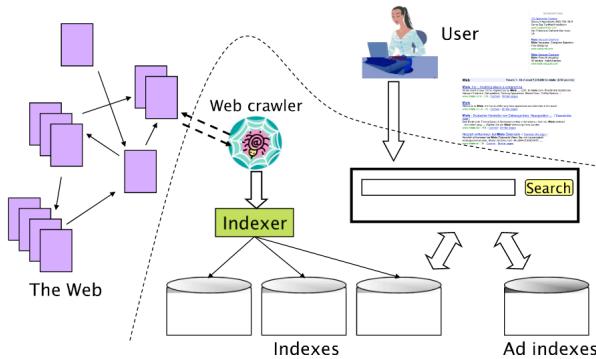


Figura 2.3: Los diversos componentes de un motor de búsqueda web [Manning et al., 2008].

## 2.8 El modelo de espacio vectorial

Para clasificar consultas o medir la similitud entre dos documentos, es esencial contar con una métrica que evalúe la similitud entre pares de documentos. Una estrategia para habilitar esta comparación implica la *representación* de los documentos como vectores de términos, en la que cada término se convierte en una dimensión en el espacio vectorial [Salton et al., 1975]. De esta manera, documentos con diferentes palabras y longitudes pueden coexistir en el mismo espacio vectorial, lo que facilita su comparación. Estas representaciones se conocen como modelos de *Bolsa-de-palabras* (Bag of Words), en los cuales se pierde información respecto al orden de las palabras y de la estructura lingüística de las oraciones, como se ilustra en la Figura 2.4.

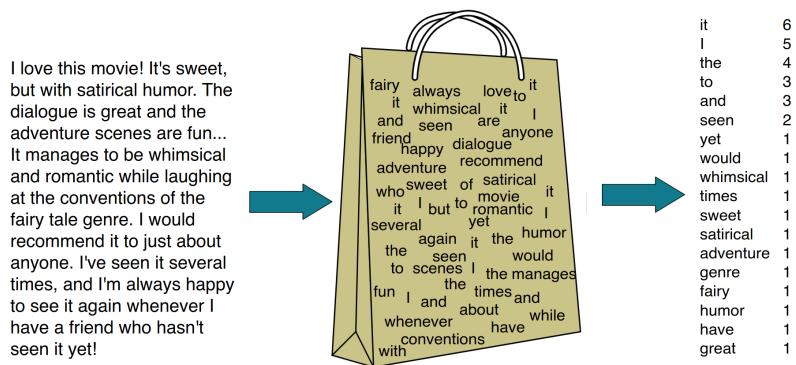


Figura 2.4: Representación de Bolsa-de-Palabras [Jurafsky and Martin, 2023].

El valor de cada dimensión es un peso que representa la relevancia del término  $t_i$  en el documento  $d$ .

$$d_j \rightarrow \vec{d}_j = (w(t_1, d_j), \dots, w(t_{|V|}, d_j)) \quad (2.2)$$

Para cuantificar la importancia que un término aporta a un documento, es necesario emplear métodos de ponderación de su relevancia, denotada como  $w(t_i, d_j)$ . Una técnica ampliamente utilizada para abordar esto es el modelo de Frecuencia de Término - Frecuencia Inversa de Documento, comúnmente conocido como “TF-IDF” (por sus siglas en inglés).

### 2.8.1 El modelo TF-IDF

Definimos  $tf_{i,j}$  como la frecuencia del término  $t_i$  en el documento  $d_j$ . La lógica subyacente en esta ponderación es que un término que aparece 10 veces en un documento debería aportar más información que uno que solo aparece una vez. Sin embargo, surge un problema cuando tenemos documentos de diferentes longitudes, ya que no deseamos favorecer a los documentos más extensos. Para abordar esta cuestión, normalizamos la frecuencia dividiendo por la frecuencia máxima del término en el documento, como se muestra a continuación:

$$ntf_{i,j} = \frac{tf_{i,j}}{\max_i(tf_{i,j})}$$

Ahora podemos preguntarnos ¿un término que ocurre en muy pocos documentos proporciona más o menos información que uno que ocurre varias veces? Por ejemplo, el documento *El respetado alcalde de Pelotillehue*. El término *Pelotillehue* ocurre en menos documentos que el término *alcalde*, por lo que debería ser más descriptivo. La frecuencia del término en el documento no es capaz de considerar esa relevancia.

**Definición 2.8.1** Sea  $N$  el número de documentos en la colección y  $n_i$  el número de documentos que contienen el término  $t_i$ , definimos la frecuencia inversa de documento ( $idf$ ) de  $t_i$  de la siguiente manera:

$$idf_{t_i} = \log_{10} \left( \frac{N}{n_i} \right)$$

Un término que aparece en todos los documentos tendría  $idf = 0$ , y uno que aparece en el 10 % de los documentos tendría  $idf = 1$ .

**Definición 2.8.2** El modelo  $tf-idf$  combina los valores de  $tf$  e  $idf$ , y resulta en los siguientes pesos  $w$  para un término en un documento:

$$w(t_i, d_j) = tf_{i,j} \times \log_{10} \left( \frac{N}{n_i} \right)$$

Las consultas de los motores de búsqueda también pueden ser modeladas como vectores. Sin embargo, en promedio, las consultas suelen tener entre 2 y 3 términos. Para evitar tener demasiadas dimensiones nulas, los vectores de consulta pueden suavizarse de la siguiente manera:

$$w(t_i, d_j) = (0,5 + 0,5 \times tf_{i,j}) \log_{10} \left( \frac{N}{n_i} \right)$$

### 2.8.2 Similitud entre vectores

Representar consultas y documentos como vectores permite calcular su similitud. Un enfoque podría ser utilizar la distancia euclíadiana. El enfoque común es calcular el coseno del ángulo entre los dos vectores. Si ambos documentos son iguales, el ángulo sería 0 y su coseno sería 1. Por otro lado, si son ortogonales, el coseno es 0.

**Definición 2.8.3** La similitud del coseno se calcula de la siguiente manera:

$$\text{similitud del coseno}(\vec{d}_1, \vec{d}_2) = \frac{\vec{d}_1 \cdot \vec{d}_2}{|\vec{d}_1| \times |\vec{d}_2|} = \frac{\sum_{i=1}^{|V|} (w(t_i, d_1) \times w(t_i, d_2))}{\sqrt{\sum_{i=1}^{|V|} w(t_i, d_1)^2} \times \sqrt{\sum_{i=1}^{|V|} w(t_i, d_2)^2}}$$

Esto se llama incorrectamente “distancia del coseno”. En realidad, es una métrica de similitud pues entre más cerca dos vectores mayor es el valor asociado. Nótese que la similitud del coseno normaliza los vectores por su norma euclíadiana  $\|\vec{d}\|_2$ .

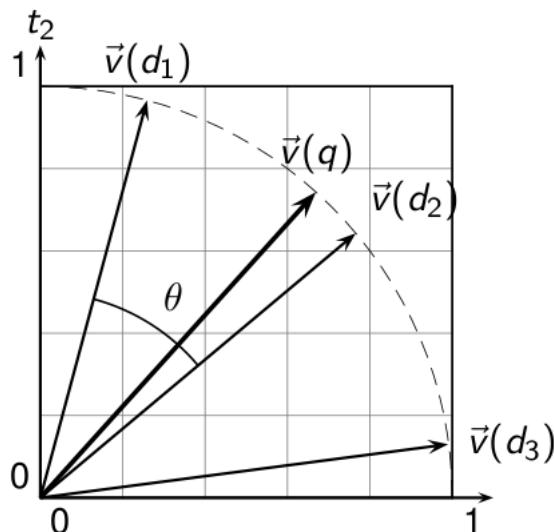


Figura 2.5: Similitud del coseno.

Un motor de búsqueda puede entonces usar el índice invertido para tener una representación vectorial de los documentos y así una vez llega una consulta usar la similitud coseno entre la consulta y los documentos indexados para producir el ranking de relevancia que se le muestra al usuario.

#### Ejercicio

- Supongamos que tenemos 3 documentos formados a partir de las siguientes secuencias de términos:  $d_1 \rightarrow t_4 t_3 t_1 t_4$   $d_2 \rightarrow t_5 t_4 t_2 t_3 t_5$   $d_3 \rightarrow t_2 t_1 t_4 t_4$
- Construya una matriz término-documento de dimensiones  $5 \times 3$  utilizando pesos simples de  $tf-idf$  (sin normalización).
- Recomendamos que primero construyas una lista con el número de documentos en los que aparece cada término (útil para calcular los valores de  $idf$ ).
- Luego, calcula los valores de  $idf$  para cada término.
- Rellena las celdas de la matriz con los valores de  $tf-idf$ .
- ¿Cuál es el documento más cercano a  $d_1$ ?

Resultado:

	d1	d2	d3
t1	0.176	0.000	0.176
t2	0.000	0.176	0.176
t3	0.176	0.176	0.000
t4	0.000	0.000	0.000
t5	0.000	0.954	0.000

Cuadro 2.2: Matriz tf-idf

## 2.9 Clustering de Documentos

¿Cómo podemos agrupar (o clusterizar) documentos que sean similares entre sí? El clustering es el proceso de agrupar documentos que comparten similitudes. Cada grupo de documentos se denomina *cluster*. En el clustering, buscamos identificar grupos de documentos donde la similitud entre los documentos del mismo grupo se maximice, mientras que la similitud entre los documentos de diferentes grupos se minimice.

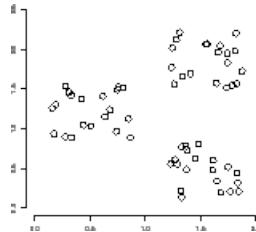


Figura 2.6: Conjunto de documentos donde los grupos se pueden identificar claramente.

El clustering de documentos permite identificar tópicos en un corpus y reducir el espacio de búsqueda en un motor de búsqueda. Por ejemplo, el índice invertido se organiza según los grupos. El algoritmo K-means es una técnica de clustering simple que toma como parámetro el número de grupos,  $k$ .

El algoritmo se basa en la noción de *centroídes*, que son los vectores promedio de los documentos pertenecientes al mismo grupo. Por ejemplo, dado un conjunto de vectores bidimensionales  $S = \{(3, 6), (1, 2), (5, 1)\}$ , el centroide de  $S$  sería  $(3, 3)$ , calculado como  $(3 + 1 + 5)/3$  y  $(6 + 2 + 1)/3$ .

El algoritmo K-means comienza seleccionando  $k$  centroídes de manera aleatoria. Luego, calcula la similitud entre cada documento y cada centroide, asignando cada documento al centroide más cercano, formando así un grupo. Los centroídes se recalculan de acuerdo con los documentos asignados a ellos. Este proceso se repite hasta que se cumple un criterio de convergencia, como se ilustra en el Algoritmo 1.

**Algorithm 1:** K-Means

---

**Input :** Datos vectoriales  $\vec{x}_1, \dots, \vec{x}_N$  y el número de clusters  $K$

**Output :** centroides de clusters  $\vec{\mu}_1, \dots, \vec{\mu}_K$

$$(\vec{s}_1, \vec{s}_2, \dots, \vec{s}_K) \leftarrow \text{SELECT\_RANDOM\_SEEDS}(\vec{x}_1, \dots, \vec{x}_N, K);$$

**for**  $k \leftarrow 1$  **to**  $K$  **do**

$$\quad \vec{\mu}_k \leftarrow \vec{s}_k;$$

**while** no se cumpla el criterio de parada **do**

**for**  $k \leftarrow 1$  **to**  $K$  **do**

$$\quad \omega_k \leftarrow \{\};$$

**for**  $n \leftarrow 1$  **to**  $N$  **do**

$$\quad \quad j \leftarrow \arg \max_{j'} \text{coseno}(\vec{\mu}_{j'}, \vec{x}_n);$$

$$\quad \quad \omega_j \leftarrow \omega_j \cup \{\vec{x}_n\};$$

**for**  $k \leftarrow 1$  **to**  $K$  **do**

$$\quad \quad \vec{\mu}_k \leftarrow \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x};$$

**return**  $\{\vec{\mu}_1, \dots, \vec{\mu}_K\};$

---

**2.10 Modelos de Tópicos**

Otra familia de modelos valiosos para explorar colecciones de documentos son los modelos de tópicos, como Latent Dirichlet Allocation (LDA) [Blei et al., 2003]. Estos modelos asumen que cada documento es una mezcla de variables latentes llamadas tópicos, donde cada tema es una bolsa de palabras con una probabilidad asociada.

LDA es uno de los modelos de tópicos más ampliamente utilizados y se basa en las siguientes suposiciones:

1. Cada documento está compuesto por combinación probabilística de  $K$  tópicos. La distribución de tópicos en un documento se puede representar mediante una distribución de probabilidad.
2. Cada tema es una distribución probabilística de palabras. Esto significa que cada palabra en un documento proviene de uno de los tópicos presentes en el documento.
3. La generación de palabras en un documento sigue un proceso de generación conjunta condicionado a la distribución de tópicos en el documento. Es decir, primero se selecciona un tema de la distribución de tópicos del documento y luego se selecciona una palabra del tema seleccionado.

La inferencia en LDA a partir de un corpus de documentos se realiza utilizando técnicas de inferencia Bayesiana, como el Muestreo de Gibbs y la Inferencia Variacional. Describir más formalmente LDA se escapa del alcance de este apunte, por ahora podemos decir que se relaciona estrechamente a los Modelos de Lenguaje Probabilísticos a estudiar en el Capítulo 3.

Si comparamos LDA con clusterizar documentos usando K-means, podemos decir que los tópicos que encuentra LDA puedan ser más fácilmente interpretados que los clusters obtenidos este último. Como cada tema es una distribución de probabilidad sobre todas las palabras del corpus de entrenamiento, uno puede analizar las primeras 10, 15 o 30 palabras más probables de cada tema para darle una interpretación. Análogamente uno puede analizar los tópicos más probables asignados a cada documento para entender los tópicos que trata.

La Tabla 2.3 muestra cuatro tópicos identificados por LDA en su trabajo original [Blei et al., 2003]

usando un corpus de resúmenes de artículos científicos. Se muestran las 15 palabras más probables para 4 tópicos identificados.

Arts	Budgets	Children	Education
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

Cuadro 2.3: Ejemplo de 4 tópicos y sus 15 palabras más probables.

## 2.11 Conclusiones y Conceptos Adicionales

En este capítulo, hemos explorado la representación de documentos como vectores para calcular similitudes entre ellos. Los vectores de “bolsa-de-palabras” son una forma común de representar documentos, pero tienen limitaciones ya que carecen de estructura lingüística y pueden ser de alta dimensionalidad y dispersos.

Una alternativa para capturar expresiones de múltiples palabras es utilizar n-gramas de palabras, que son secuencias contiguas de  $n$  palabras. Por ejemplo, representar “New York” como “new\_york” nos permite capturar la relación entre esas dos palabras en lugar de tratarlas como entidades separadas.

Es importante señalar que los sistemas modernos de recuperación de información van más allá de la similitud de vectores. Utilizan algoritmos como PageRank [Page et al., 1998] para determinar la relevancia de un sitio web basándose en los enlaces que lo apuntan. También emplean técnicas de aprendizaje automático que utilizan registros de consultas anteriores para mejorar el ranking de resultados. Además, el uso de grafos de conocimiento enriquece aún más los resultados al proporcionar información estructurada y relaciones entre entidades.

Es importante destacar que disciplinas como la recuperación de información y la minería de textos se centran menos en la estructura lingüística que PLN y más en producir algoritmos rápidos y escalables para procesar grandes volúmenes de texto.



### 3. Modelos de Lenguaje Probabilísticos

En esta sección introducimos el concepto de “modelo de lenguaje”. Un enfoque para asignar probabilidades a las oraciones que, como veremos a lo largo del apunte, desempeñará un papel fundamental en los enfoques modernos de PLN. De hecho, los grandes modelos de lenguaje como “ChatGPT” y “Google Bard” están basados en esta tecnología.

#### 3.1 El Problema del Modelado del Lenguaje

Supongamos que tenemos un corpus de documentos  $\mathcal{C}$ , del cual extraemos su vocabulario (finito) a partir de la tokenización de sus documentos,  $\mathcal{V} = \{\text{el, un, hombre, telescopio, dos, fan, vió, jugar, por, Real, Madrid, Zamorano, ...}\}$ .

Podemos formar a partir de este vocabulario un conjunto infinito de oraciones si dejamos que éstas sean de largo variable,  $\mathcal{V}^*$ .

■ **Ejemplo 3.1** Algunas oraciones que podríamos formar:

- el STOP
- un STOP
- el fan STOP
- el fan vió a Zamorano STOP
- el fan vió vió STOP
- el fan vió a Zamorano jugar por el Real Madrid STOP

Donde STOP es un símbolo especial que indica el final de una oración. ■

**Definición 3.1.1** Un modelo de lenguaje es una distribución de probabilidad  $p$  sobre todas las oraciones que se pueden formar a partir de un vocabulario finito. Entonces, la probabilidad de cada oración debe ser mayor o igual que cero, y la suma de las probabilidades de todas las

oraciones posibles debe ser uno:

$$\sum_{x \in V^*} p(x) = 1$$

$$p(x) \geq 0 \quad \text{para todo } x \in V^*$$

■ **Ejemplo 3.2** A continuación mostramos ejemplos de probabilidades asignadas por un modelo de lenguaje imaginario a algunas oraciones:

$$p(\text{el STOP}) = 10^{-12}$$

$$p(\text{el fan STOP}) = 10^{-8}$$

$$p(\text{el fan vio a Zamorano STOP}) = 2 \times 10^{-8}$$

$$p(\text{el fan vio vio STOP}) = 10^{-15}$$

$$\dots$$

$$p(\text{el fan vio a Zamorano jugar por el Real Madrid STOP}) = 2 \times 10^{-9}$$

Lo que esperamos del modelo es que le asigne una probabilidad más alta a las oraciones fluidas (aquellas que tienen sentido y son gramaticalmente correctas) de las que no. También esperamos poder estimar esta función de probabilidad a partir del texto (corpus).

A nivel general, el modelo de lenguaje ayuda a los modelos de generación de texto a distinguir entre buenas y malas oraciones.

### 3.1.1 ¿Por qué queríamos hacer esto?

La motivación original de estos modelos se origina en el problema de reconocimiento del habla o de transcripción automática para ayudar a estos sistemas a discernir entre distintas oraciones compatibles con la señal acústica. Consideremos las siguientes dos oraciones en inglés:

1. “recognize speech” (reconocer el habla)
2. “wreck a nice beach” (arruinar una playa bonita).

Estas dos oraciones suenan muy similares al ser pronunciadas (tienen casi la misma señal acústica), lo que dificulta que los sistemas automáticos de reconocimiento del habla las transcriban con precisión. Cuando el sistema de reconocimiento del habla analiza la entrada de audio e intenta transcribirlo, tiene en cuenta las probabilidades del modelo de lenguaje para determinar la interpretación más probable. El modelo de lenguaje favorecería  $p(\text{"recognize speech"})$  sobre  $p(\text{"wreck a nice beach"})$ . Esto se debe a que la primera oración es más común y debería ocurrir con más frecuencia en el corpus de entrenamiento.

Al incorporar modelos de lenguaje, los sistemas de reconocimiento del habla pueden mejorar la precisión al seleccionar la oración que se alinea mejor con los patrones lingüísticos y el contexto, incluso cuando se enfrentan a alternativas que suenan similar.

De hecho, los modelos de lenguaje son útiles en cualquier tarea de procesamiento del lenguaje natural que involucre la generación de lenguaje (por ejemplo, traducción automática, resumen, generación de resúmenes, chatbots, reconocimiento óptico de caracteres y el reconocimiento de escritura a mano).

Además la noción de modelamiento probabilístico del lenguaje se utiliza en muchos modelos y tareas de PLN, por lo que las técnicas de estimación desarrolladas en este capítulo serán muy útiles para entender muchos problemas en PLN.

### 3.2 Regla de la Cadena en Modelos de Lenguaje

En el contexto de los modelos de lenguaje, se parte de la premisa de que cualquier oración  $s$  puede ser vista como una secuencia de variables aleatorias  $X_1, X_2, \dots, X_n$ . Cada una de estas variables aleatorias puede tomar valores de un conjunto finito  $V$ . Para simplificar, asumimos una longitud fija  $n$  para la secuencia. Nuestro objetivo consiste en modelar la probabilidad conjunta  $p(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n)$ . Es importante notar que utilizamos letras mayúsculas para denotar las variables aleatorias ( $X_1, X_2, \dots, X_n$ ) y minúsculas para referirnos a instancias específicas de estas variables ( $x_1, x_2, \dots, x_n$ ), que representan elementos del vocabulario (por ejemplo, “perro”, “gato”, “comida”).

Una forma común de representar secuencias de variables aleatorias es mediante la aplicación de la regla de la cadena de probabilidades que definimos a continuación.

Para un conjunto de variables aleatorias  $X_1, \dots, X_n$ , la regla de la cadena de probabilidades nos permite expresar la función de probabilidad conjunta  $p(X_1, X_2, \dots, X_n)$  como un producto de  $n$  probabilidades condicionales:

$$p(X_1, X_2, \dots, X_n) = p(X_1)p(X_2|X_1)p(X_3|X_2, X_1) \dots p(X_n|X_{n-1}, \dots, X_2, X_1). \quad (3.1)$$

Por ejemplo, en el caso de  $n = 3$ , la expresión se simplifica a:

$$p(X_1, X_2, X_3) = p(X_1)p(X_2|X_1)p(X_3|X_2, X_1).$$

Al utilizar la definición de probabilidades condicionales, podemos expresar  $p(X_2|X_1)$  como  $p(X_1, X_2)/p(X_1)$  y  $p(X_3|X_2, X_1)$  como  $p(X_1, X_2, X_3)/p(X_1, X_2)$ . Al sustituir estas expresiones en la cadena de productos, vemos que todas las expresiones de cancelan y finalmente obtenemos  $p(X_1, X_2, X_3)$ .

La aplicación de la regla de la cadena en los modelos de lenguaje no impone ningún supuesto específico sobre el modelo en sí. Más bien, nos proporciona una forma alternativa de modelar la probabilidad de una oración. Sin embargo, la ventaja de utilizar la regla de la cadena es que nos permite aplicar supuestos y hacer simplificaciones en el modelo para facilitar su implementación y cálculo.

### 3.3 Mirada Predictiva

La regla de la cadena nos permite darle una interpretación predictiva a los modelos de lenguaje. Uno puede asumir que un modelo de lenguaje una vez entrenado (o construido) a partir de un corpus es capaz de entregar una distribución de probabilidad para todas las palabras a partir de un contexto específico. Supongamos que tenemos un contexto formado por una secuencia de palabras  $c = X_{n-1}, \dots, X_1$  y queremos predecir la palabra más probable a partir de ese contexto. Uno podría evaluar la probabilidad condicional  $p(X_n|X_{n-1}, \dots, X_1)$  o  $p(X_n|c)$  para todas las palabras del vocabulario que puede tomar  $X_n$  y escoger la más probable:

$$x_n = \operatorname{argmax}_{X_n \in \mathcal{V}} p(X_n|c)$$

■ **Ejemplo 3.3** Supongamos que el contexto es  $c = \text{"Cristóbal Colón descubrió"}$ . El modelo de lenguaje sería capaz de determinar:

$$p(X|\text{"descubrió", "Colón", "Cristóbal"})$$

$$\forall X \in V, \quad \text{donde } \sum_{X \in V} p(X|c) = 1$$

Por ejemplo:

$$p(\text{"América"}|\text{"descubrió", "Colón", "Cristóbal"}) = 0,02$$

$$p(\text{"Europa"}|\text{"descubrió", "Colón", "Cristóbal"}) = 0,01$$

$$p(\text{"Chile"}|\text{"descubrió", "Colón", "Cristóbal"}) = 0,001$$

$$p(\text{"camello"}|\text{"descubrió", "Colón", "Cristóbal"}) = 0,00001$$

...

Entonces, nuestro modelo de lenguaje predeciría  $X = \text{"América"}$  como la palabra siguiente más probable. ■

En el ejemplo anterior esperamos que un buen modelo de lenguaje asigne una probabilidad mayor a  $X = \text{"América"}$  que al resto de las opciones. Para realizar estas predicciones, el modelo de lenguaje necesita capturar las reglas sintácticas y semánticas del lenguaje, así como tener un amplio conocimiento del mundo.

Otra propiedad relevante, que se abordará en el Capítulo 13 sobre grandes modelos de lenguaje, es que se puede codificar prácticamente cualquier tarea de PLN como una instrucción dentro del contexto de un modelo de lenguaje. Por ejemplo, se podría utilizar la instrucción “traduce el siguiente texto de inglés a español: I like dogs”, para luego esperar que el modelo de lenguaje sea capaz de realizar la traducción correcta.

### 3.4 Mirada Generativa

Los modelos de lenguaje aprendidos a partir de un corpus de texto no solo tienen una capacidad predictiva, sino que también tienen una capacidad generativa. Estos modelos pueden utilizarse para generar oraciones nuevas muestreando secuencialmente a partir de las probabilidades condicionales estimadas.

El proceso generativo implica tomar una palabra inicial y, a medida que avanzamos en la generación, seleccionar sucesivas palabras basadas en las probabilidades condicionales proporcionadas por el modelo de lenguaje. Es similar a extraer bolas (palabras) de una urna, donde los tamaños de las bolas están en proporción a sus frecuencias relativas, que son determinadas por el modelo de lenguaje. Cada vez que se selecciona una palabra, se utiliza como contexto para determinar la siguiente palabra, y así sucesivamente hasta que se haya generado una oración completa.

En este proceso generativo, hay diferentes enfoques que se pueden tomar. Uno de ellos es el muestreo estocástico, donde se elige una palabra de acuerdo con su probabilidad condicional dada por el modelo. Esto permite cierta variabilidad en la generación de oraciones, ya que las palabras menos probables también tienen la posibilidad de ser seleccionadas. Por otro lado, también se puede optar por seleccionar la palabra más probable en cada paso, lo que equivale a predecir la siguiente palabra basándose únicamente en la información disponible hasta ese momento.

### 3.5 Un Método Ingenuo

Un método muy ingenuo para estimar la probabilidad de una oración es contar todas las apariciones de oraciones idénticas a esta en los datos de entrenamiento y dividirlo por el número total de oraciones de entrenamiento ( $N$ ).

Más formalmente, sea mi oración  $s = x_1, x_2, \dots, x_n$ , y  $c(x_1, x_2, \dots, x_n)$  el número de veces que ocurre la oración en el corpus de entrenamiento, el modelo de lenguaje ingenuo computa la probabilidad de la siguiente manera:

$$p(x_1, x_2, \dots, x_n) = \frac{c(x_1, x_2, \dots, x_n)}{N}$$

El problema de este enfoque, es que el número de posibles oraciones crece de manera exponencial con su cantidad de palabras y el tamaño del vocabulario. Entonces se vuelve cada vez más improbable que una oración específica haya ocurrido en los datos de entrenamiento. Esto es una consecuencia de la propiedad de **dispersión** del lenguaje planteada en el Capítulo 1.

En consecuencia, muchas oraciones tendrán una probabilidad cero según el modelo ingenuo, lo que lleva a una mala generalización.

### 3.6 Procesos de Markov

Un proceso de Markov de primer orden asume que la probabilidad de que una variable aleatoria tome un valor depende únicamente del valor inmediatamente anterior en la secuencia. En el contexto del modelado del lenguaje, esto significa que la probabilidad de una palabra en una oración depende solo de la palabra anterior. Si aplicamos este supuesto a la regla de cadena de probabilidades, la probabilidad conjunta de una secuencia de palabras se simplifica en una multiplicación de probabilidades condicionales de las palabras sucesivas dado su palabra predecesora inmediata:

$$p(X_i = x_i | X_1 = x_1, \dots, X_{i-1} = x_{i-1}) = p(X_i = x_i | X_{i-1} = x_{i-1}) \quad (3.2)$$

Un proceso de Markov de segundo orden amplía la suposición de Markov de primer orden y considera el valor de dos variables anteriores en la secuencia. En el modelado del lenguaje, esto significa que la probabilidad de una palabra en una oración depende de las dos palabras anteriores. La probabilidad conjunta de una secuencia de palabras se calcula multiplicando las probabilidades condicionales de las palabras sucesivas dado sus dos palabras predecesoras:

$$p(X_i = x_i | X_1 = x_1, \dots, X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) = p(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) \quad (3.3)$$

#### 3.6.1 Modelado de secuencias de longitud variable

Si queremos modelar secuencias de longitud variable, podemos considerar que la longitud de la secuencia,  $n$ , también es una variable aleatoria. Una forma simple de abordar esto es siempre definir  $X_n = \text{STOP}$ , donde “STOP” es un símbolo especial que marca el final de la secuencia. Luego, podemos usar un proceso de Markov como antes para modelar la probabilidad conjunta de las palabras en la secuencia:

$$p(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = \prod_{i=1}^n p(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

Aquí, asumimos que  $x_0 = x_{-1} = *$  por conveniencia, donde “\*” es un símbolo especial de “inicio” de oración.

### 3.7 Modelos de lenguaje de Trigramas

Un n-grama es una secuencia contigua de  $n$  palabras. Un modelo de lenguaje de trígrama (n-grama con  $n = 3$ ) acoge los supuestos de independencia del modelo de Markov de segundo orden que se discutió anteriormente. De esta forma permite expresar la probabilidad de una oración de la siguiente forma:

$$p(X_1, X_2, \dots, X_n) = p(X_1)p(X_2|X_1)p(X_3|X_2, X_1)\dots p(X_n|X_{n-1}, X_{n-2}). \quad (3.4)$$

Nótese que en vez de condicionar por todas las palabras anteriores como en la Ecuación 3.1 aquí solo se condiciona por las dos palabras anteriores.

Los componentes que definen al modelo de lenguaje de trígrama se definen a continuación:

1. Un vocabulario  $V$ , representado por un conjunto finito de palabras.
2. Un parámetro escalar  $q(w|u, v)$  para cada trígrama  $u, v, w$ , donde  $w \in V \cup \{\text{STOP}\}$  y  $u, v \in V \cup \{*\}$ .

Para cualquier oración  $x_1 \dots x_n$ , donde  $x_i \in V$  para  $i = 1 \dots (n - 1)$  y  $x_n = \text{STOP}$ , la probabilidad de la oración según el modelo de lenguaje trígrama es:

$$p(x_1 \dots x_n) = \prod_{i=1}^n q(x_i|x_{i-2}, x_{i-1})$$

Aquí, definimos  $x_0 = x_{-1} = *$  como dos tokens especiales antes del inicio del oración para poder condicionar tanto la primera como la segunda palabra de una oración por sus dos palabras. A este tipo de transformaciones se les conoce en PLN como “padding”.

■ **Ejemplo 3.4** Por ejemplo, para la oración “el perro ladra STOP”, estimaríamos su probabilidad de la siguiente forma:

$$p(\text{el perro ladra STOP}) = q(\text{el}|*, *) \times q(\text{perro}|*, \text{el}) \times q(\text{ladra}|\text{el}, \text{perro}) \times q(\text{STOP}|\text{perro}, \text{ladra})$$

■

Ahora abordemos el problema de la estimación de los parámetros  $q(w_i|w_{i-2}, w_{i-1})$  a partir de un corpus  $\mathcal{C}$ .

Una estimación natural es la “estimación de máxima verosimilitud” (MLE por sus siglas en inglés), donde se estiman los parámetros para maximizar la probabilidad de los datos de entrenamiento.

En el contexto de un modelo de trígrama, la estimación de cada parámetro proviene de distribuciones multinomiales, lo cual se traduce en el siguiente cálculo:

$$q(w_i|w_{i-2}, w_{i-1}) = \frac{\text{Count}(w_{i-2}, w_{i-1}, w_i)}{\text{Count}(w_{i-2}, w_{i-1})}$$

Esto requiere tokenizar el corpus de entrenamiento y guardar en una tabla los conteos de todos los trígramas (secuencias contiguas de 3 palabras) y bigramas (secuencias contiguas de dos palabras) encontrados.

■ **Ejemplo 3.5** Por ejemplo:

$$q(\text{ladra}|\text{el, perro}) = \frac{\text{Count(el, perro, ladra)}}{\text{Count(el, perro)}}$$

■

### 3.8 Evaluación de un modelo de lenguaje: Perplejidad

Un modelo de lenguaje efectivo se caracteriza por asignar probabilidades más altas a oraciones coherentes que a oraciones sin sentido. Para evaluar la calidad de un modelo de lenguaje, es esencial utilizar un corpus de prueba independiente diferente al corpus en el que fue entrenado. Esto nos permitirá verificar cómo generaliza el modelo a nuevas oraciones. La evaluación cuantitativa de modelos de lenguaje nos permite comparar distintos tipos de modelos de lenguaje y seleccionar el modelo más apropiado.

Para evaluar un modelo de lenguaje utilizamos un corpus de prueba independiente,  $\mathcal{C}_{\text{test}}$ , que contiene  $m$  oraciones:  $s_1, s_2, s_3, \dots, s_m$ , y cuantificamos la probabilidad que el modelo asigna a estas oraciones. Para esto, calculamos la probabilidad total asignada por el modelo a las oraciones del corpus de prueba, lo cual se puede expresar más convenientemente mediante la probabilidad logarítmica<sup>1</sup>:

$$\log \left( \prod_{i=1}^m p(s_i) \right) = \sum_{i=1}^m \log p(s_i)$$

**Definición 3.8.1** La medida de evaluación estándar para modelos de lenguaje es la perplejidad (o perplexity), que se calcula de la siguiente manera:

$$\text{Perplejidad} = 2^{-l} \quad \text{donde} \quad l = \frac{1}{M} \sum_{i=1}^m \log p(s_i)$$

Donde  $M$  es el número total de palabras en los datos de prueba.

La perplejidad es una métrica que favorece valores más bajos y tiene la propiedad de ser más fácil de interpretar que la probabilidad logarítmica, como se discutirá a continuación.

#### 3.8.1 Intuición sobre la perplejidad

Supongamos que tenemos un vocabulario  $V$ , y  $N = |V| + 1$ , y un modelo de lenguaje “uniforme” que le asigna la misma probabilidad a todos los trigramas:

$$q(w|u, v) = \frac{1}{N} \quad \text{para todo } w \in V \cup \{\text{STOP}\}, \text{ para todo } u, v \in V \cup \{*\}$$

La perplejidad de este modelo de lenguaje es  $N$  tal como se demuestra a continuación.

Supongamos que tenemos  $m$  oraciones de longitud  $n$  en el corpus, y  $M$  es la cantidad de tokens en el corpus,  $M = m \cdot n$ . Consideremos el logaritmo (base 2) de la probabilidad de una oración

---

<sup>1</sup>El logaritmo es una función monótona y permite llevar los productos a suma que evita problemas de underflow al trabajar con números pequeños.

$s = w_1 w_2 \dots w_n$  bajo el modelo:

$$\log p(s) = \log \prod_{i=1}^n q(w_i | w_{i-2}, w_{i-1}) = \sum_{i=1}^n \log q(w_i | w_{i-2}, w_{i-1})$$

Dado que cada parámetro  $q(w_i | w_{i-2}, w_{i-1})$  es igual a  $\frac{1}{N}$ , tenemos:

$$\log p(s) = \sum_{i=1}^n \log \frac{1}{N} = n \cdot \log \frac{1}{N} = -n \cdot \log N$$

$$l = \frac{1}{M} \sum_{i=1}^m \log p(s_i) = \frac{1}{M} \sum_{i=1}^m -n \cdot \log N = \frac{1}{M} \cdot -m \cdot n \cdot \log N = -\log N$$

Por lo tanto, la perplejidad está dada por:

$$\text{Perplejidad} = 2^{-l} = 2^{-(-\log N)} = N$$

Esperamos entonces, que cualquier modelo de lenguaje que sea capaz de aprovechar la distribución de las palabras (como un modelo de trigramas) tenga una perplejidad menor que el tamaño del vocabulario. En [Goodman, 2001], donde  $|V| = 50,000$ , se presentan los siguientes resultados:

- Modelo de trigramas:  $p(x_1, \dots, x_n) = \prod_{i=1}^n q(x_i | x_{i-2}, x_{i-1})$   
Perplejidad = 74
- Modelo de bigramas:  $p(x_1, \dots, x_n) = \prod_{i=1}^n q(x_i | x_{i-1})$   
Perplejidad = 137
- Modelo de unigramas:  $p(x_1, \dots, x_n) = \prod_{i=1}^n q(x_i)$   
Perplejidad = 955

Estos resultados muestran que la perplejidad disminuye a medida que aumentamos el nivel de contexto considerado en el modelo de lenguaje, lo que indica que los modelos con perplejidad más baja generalmente son más efectivos en la tarea de predicción del lenguaje. Sin embargo, como veremos a continuación, aumentar el tamaño del contexto no está libre de problemas.

### 3.8.2 El trade-off entre sesgo y varianza

Una limitación de los modelos de trigramas está relacionada con la dispersión del lenguaje, discutida en el Capítulo 1. A grandes rasgos, si el tamaño del vocabulario es  $N = |V|$ , entonces el modelo tiene  $N^3$  parámetros. Por ejemplo, si  $N = 20,000$ , habría  $20,000^3 = 8 \times 10^{12}$  parámetros. Esta cantidad de parámetros es enorme y dificulta la estimación adecuada de todos ellos sin requerir un corpus de entrenamiento de tamaño enorme. Como resultado, puede surgir un problema de sobreajuste (overfitting) a los datos de entrenamiento. Por otro lado, modelos más simples, como un modelo de lenguaje de unígrafo o de bigrama, no son capaces de modelar adecuadamente contextos más largos.

Este dilema es conocido en estadística y aprendizaje automático como el trade-off entre sesgo y varianza, que se refiere a la compensación entre la simplicidad del modelo y su capacidad para capturar la complejidad y variabilidad de los datos.

Los modelos más simples, como los modelos de n-gramas de orden inferior (unígramas, bigramas), tienen un sesgo más alto pero una varianza más baja. Estos modelos asumen independencia condicional entre las palabras y simplifican la estructura del lenguaje.

En contraste, los modelos más complejos, como modelos de n-gramas con  $n > 3$ , tienen una varianza más alta pero un sesgo más bajo. Estos modelos pueden capturar relaciones más complejas entre las palabras, pero también son más propensos a sobreajustar los datos de entrenamiento y tener dificultades para generalizar a nuevas muestras.

Cuando se entrena un modelo de lenguaje con MLE, se maximiza la probabilidad de los datos de entrenamiento. Esto puede llevar a asignar probabilidades altas a secuencias específicas que aparecen en los datos de entrenamiento, incluso si esas secuencias son poco probables en la distribución real del lenguaje.

Como resultado, el modelo puede tener un rendimiento deficiente en datos de prueba que contienen secuencias diferentes a las del conjunto de entrenamiento. Esto se debe a que el modelo se ha sobreajustado a los datos de entrenamiento y ha capturado sus características específicas en lugar de aprender patrones más generales del lenguaje.

Para abordar el problema del sobreajuste en modelos de lenguaje, se utilizan diversas técnicas de regularización. Estas técnicas ayudan a controlar la varianza del modelo y mitigar el sobreajuste, permitiendo un mejor equilibrio entre el sesgo y la varianza y mejorando la generalización a nuevas muestras. A continuación, veremos dos técnicas de regularización para modelos de lenguaje: la interpolación lineal y los modelos de descuento.

### 3.9 Interpolación Lineal

La principal idea detrás de la técnica de interpolación lineal es combinar las distribuciones de probabilidad estimadas por modelos de trigramas con las obtenidas a través de modelos de órdenes inferiores, como bigramas y unigramas. Esta estrategia ofrece la ventaja de incorporar información de n-gramas de órdenes más bajos, permitiendo abordar la indefinición de probabilidades en palabras no observadas durante el entrenamiento. Pues basta que una oración tenga un sólo trígrafo no observado en entrenamiento para recibir una probabilidad de cero por el efecto multiplicativo de la regla de la cadena.

En el modelo interpolado, se ponderan linealmente tres modelos distintos:

1. Modelo de trigramas:  $q_{ML}(w_i|w_{i-2}, w_{i-1}) = \frac{\text{Count}(w_{i-2}, w_{i-1}, w_i)}{\text{Count}(w_{i-2}, w_{i-1})}$
2. Modelo de bigramas:  $q_{ML}(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$
3. Modelo de unigramas:  $q_{ML}(w_i) = \frac{\text{Count}(w_i)}{\text{Count}(\#tokens \text{ en el corpus})}$ .

Cada parámetro  $q(w_i|w_{i-2}, w_{i-1})$  se interpola de la siguiente manera:

$$q(w_i|w_{i-2}, w_{i-1}) = \lambda_1 \cdot q_{ML}(w_i|w_{i-2}, w_{i-1}) + \lambda_2 \cdot q_{ML}(w_i|w_{i-1}) + \lambda_3 \cdot q_{ML}(w_i)$$

Donde  $\lambda_1$ ,  $\lambda_2$ , y  $\lambda_3$  son hiper-parámetros que deben definirse manualmente y cumplir con las condiciones  $\lambda_1 + \lambda_2 + \lambda_3 = 1$ , y  $\lambda_i \geq 0$  para todo  $i$ . Como podemos ver, con esta técnica abordamos el problema de indefinición de probabilidades en el modelo de trigramas, puesto que si se le quiere asignar probabilidad a un trígrafo no observado durante entrenamiento, este no necesariamente indefine las probabilidades pues puede recurrir a las probabilidades de los bigramas y unigramas correspondientes.

Además, se puede demostrar que el modelo interpolado define adecuadamente una distribución de probabilidad (donde definimos  $V' = V \cup \{\text{STOP}\}$ ):

$$\begin{aligned}
& \sum_{w \in V'} q(w|u, v) \\
&= \sum_{w \in V'} [\lambda_1 \cdot q_{\text{ML}}(w|u, v) + \lambda_2 \cdot q_{\text{ML}}(w|v) + \lambda_3 \cdot q_{\text{ML}}(w)] \\
&= \lambda_1 \sum_w q_{\text{ML}}(w|u, v) + \lambda_2 \sum_w q_{\text{ML}}(w|v) + \lambda_3 \sum_w q_{\text{ML}}(w) \\
&= \lambda_1 + \lambda_2 + \lambda_3 = 1
\end{aligned}$$

También es posible demostrar que  $q(w|u, v) \geq 0$  para todas las palabras en  $V'$ , ya que es una suma ponderada de tres cantidades no negativas.

### 3.9.1 Estimación de los Valores $\lambda$

Para encontrar un valor adecuado de los hiper-parámetros  $\lambda$  se suele reservar una parte del conjunto de entrenamiento como datos de *validación*. Definimos  $c'(w_1, w_2, w_3)$  como el número de veces que se observa el trígrama  $(w_1, w_2, w_3)$  en el conjunto de validación. Elegimos  $\lambda_1, \lambda_2, \lambda_3$  para maximizar:

$$L(\lambda_1, \lambda_2, \lambda_3) = \sum_{w_1, w_2, w_3} c'(w_1, w_2, w_3) \log q(w_3|w_1, w_2)$$

sujetos a  $\lambda_1 + \lambda_2 + \lambda_3 = 1$ , y  $\lambda_i \geq 0$  para todo  $i$ , donde

$$q(w_i|w_{i-2}, w_{i-1}) = \lambda_1 \cdot q_{\text{ML}}(w_i|w_{i-2}, w_{i-1}) + \lambda_2 \cdot q_{\text{ML}}(w_i|w_{i-1}) + \lambda_3 \cdot q_{\text{ML}}(w_i)$$

Esto generalmente se hace buscando en una grilla de posibles valores de todos los hiper-parámetros. Nótese que maximizar  $L(\lambda_1, \lambda_2, \lambda_3)$  es equivalente a minimizar la perplejidad en validación.

## 3.10 Modelos de Descuento (Katz Back-Off)

Los modelos de descuento representan una técnica alternativa a la interpolación con el propósito de mejorar la generalización del modelo de lenguaje a datos que difieren de los utilizados en el entrenamiento. La idea principal detrás de los modelos de descuento es redistribuir la probabilidad de los n-gramas observados hacia aquellos que no se han observado.

Para ilustrar estos modelos, consideremos un ejemplo utilizando los conteos de bigramas que inician con el unígrafo “la” en un corpus, junto con sus estimaciones de máxima verosimilitud, detalladas en la Tabla 3.2.

Si miramos las probabilidades de bigramas con baja frecuencia, como “la tuerca,” que se observa solo una vez y tiene una probabilidad de  $1/48$ , en contraste con un bigrama no observado, como “la puerta,” que tiene una probabilidad de  $0$ . ¿Podemos confiar plenamente en estas estimaciones o son simplemente el resultado de la aleatoriedad inherente al proceso de construir el corpus? Es posible que, si hubiéramos utilizado otro corpus, habríamos obtenido estimaciones diferentes. Esto indica que las estimaciones de MLE tienden a exagerar lo que se ha observado en el corpus y a subestimar lo que no se ha observado. Para abordar este problema, recurriremos al uso de conteos “descontados”.

Frase	Conteo	$q_{ML}(w_i w_{i-1})$
la	48	
la, gata	15	15/48
la, mujer	11	11/48
la, persona	10	10/48
la, plaza	5	5/48
la, actividad	2	2/48
la, tuerca	1	1/48
la, revista	1	1/48
la, tarde	1	1/48
la, ciudad	1	1/48
la, calle	1	1/48

Cuadro 3.1: Ejemplo de conteos de bigramas

**Definición 3.10.1** Definimos los conteos “descontados” de la siguiente manera:

$$\text{Conteo}^*(x) = \text{Conteo}(x) - \beta$$

Donde  $\beta$  es una constante que generalmente se encuentra entre 0 y 1, siendo un valor típico 0.5. Con  $\beta = 0.5$ , los conteos se descuentan, es decir, se reduce la probabilidad de los datos observados, como se muestra en la Tabla 3.2.

Frase	Conteo	Conteo <sup>*(x)</sup>	$q_{BO}(w_i w_{i-1})$
la	48		
la, gata	15	14.5	14,5/48
la, mujer	11	10.5	10,5/48
la, persona	10	9.5	9,5/48
la, plaza	5	4.5	4,5/48
la, actividad	2	1.5	1,5/48
la, tuerca	1	0.5	0,5/48
la, revista	1	0.5	0,5/48
la, tarde	1	0.5	0,5/48
la, ciudad	1	0.5	0,5/48
la, calle	1	0.5	0,5/48

Cuadro 3.2: Ejemplo de conteos de bigramas con conteos descontados

Las nuevas estimaciones se basan en los conteos descontados, y se introduce una “masa de probabilidad faltante” como se muestra a continuación:

**Definición 3.10.2**

$$\alpha(w_{i-1}) = 1 - \sum_w \frac{\text{Conteo}^*(w_{i-1}, w)}{\text{Conteo}(w_{i-1})}$$

Por ejemplo, en nuestro caso:

$$\alpha(\text{la}) = 1 - \frac{14,5 + 10,5 + 9,5 + 4,5 + 1,5 + 0,5 + 0,5 + 0,5 + 0,5}{48} = 1 - \frac{43}{48} = \frac{5}{48}$$

La idea es redistribuir esta masa faltante entre todos los bigramas que comienzan con la palabra “la” y que no fueron observados en el corpus, como por ejemplo “rana”, “bailarina” y “puerta”. No queremos distribuirlo uniformemente sino proporcionalmente con la probabilidad de cada unígrama considerado.

**Definición 3.10.3** En el modelo de Katz Back-Off de bigramas, se definen dos conjuntos:  $A(w_{i-1})$  y  $B(w_{i-1})$ , como se detalla a continuación:

$$A(w_{i-1}) = \{w : \text{Count}(w_{i-1}, w) > 0\}$$

En nuestro ejemplo,  $A(\text{la}) = \{\text{gata, mujer, persona, plaza, actividad, tuerca, revista, tarde, ciudad, calle}\}$ .

Por otro lado,  $B(w_{i-1})$  se define como:

$$B(w_{i-1}) = \{w : \text{Count}(w_{i-1}, w) = 0\}$$

En nuestro ejemplo,  $B(\text{la}) = \{\text{rana, bailarina, puerta, ...}\}$ .

Luego, la probabilidad condicional  $q_{\text{BO}}(w_i|w_{i-1})$  se calcula de la siguiente manera: si la palabra  $w_i$  está en el conjunto  $A(w_{i-1})$ , se utiliza una estimación basada en las frecuencias relativas de la secuencia  $(w_{i-1}, w_i)$  dividida por la frecuencia de  $w_{i-1}$ . Si  $w_i$  está en el conjunto  $B(w_{i-1})$ , se utiliza una estimación suavizada que combina la constante  $\alpha(w_{i-1})$  y las probabilidades condicionales de máxima verosimilitud de un modelo orden inferior (unígramas)  $q_{\text{ML}}(w_i)$  de las palabras en el conjunto  $B(w_{i-1})$ . La idea aquí es distribuir la masa de probabilidad faltante en proporción a la probabilidad de unígrama de  $w_i$  asegurando que las probabilidades resultantes queden bien definidas.

$$q_{\text{BO}}(w_i|w_{i-1}) = \begin{cases} \frac{\text{Count}^*(w_{i-1}, w_i)}{\text{Count}(w_{i-1})} & \text{si } w_i \in A(w_{i-1}) \\ \frac{\alpha(w_{i-1}) q_{\text{ML}}(w_i)}{\sum_{w \in B(w_{i-1})} q_{\text{ML}}(w)} & \text{si } w_i \in B(w_{i-1}) \end{cases}$$

La constante  $\alpha(w_{i-1})$  se calcula restando la suma de las frecuencias relativas de las palabras en  $A(w_{i-1})$  de uno tal como se definió anteriormente.

Para extender este enfoque a trigramas, se definen conjuntos similares  $A(w_{i-2}, w_{i-1})$  y  $B(w_{i-2}, w_{i-1})$  para las secuencias de trigramas:

**Definición 3.10.4**

$$A(w_{i-2}, w_{i-1}) = \{w : \text{Count}(w_{i-2}, w_{i-1}, w) > 0\}$$

$$B(w_{i-2}, w_{i-1}) = \{w : \text{Count}(w_{i-2}, w_{i-1}, w) = 0\}$$

La probabilidad condicional  $q_{\text{BO}}(w_i|w_{i-2}, w_{i-1})$  en un modelo de trigramas se calcula utilizando el modelo de bigrama correspondiente y aplicando la misma lógica anterior:

$$q_{\text{BO}}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} \frac{\text{Count}^*(w_{i-2}, w_{i-1}, w_i)}{\text{Count}(w_{i-2}, w_{i-1})} & \text{si } w_i \in A(w_{i-2}, w_{i-1}) \\ \frac{\alpha(w_{i-2}, w_{i-1}) q_{\text{BO}}(w_i|w_{i-1})}{\sum_{w \in B(w_{i-2}, w_{i-1})} q_{\text{BO}}(w|w_{i-1})} & \text{si } w_i \in B(w_{i-2}, w_{i-1}) \end{cases}$$

Donde:

$$\alpha(w_{i-2}, w_{i-1}) = 1 - \sum_{w \in A(w_{i-2}, w_{i-1})} \frac{\text{Count}^*(w_{i-2}, w_{i-1}, w)}{\text{Count}(w_{i-2}, w_{i-1})}$$

Estos modelos de Katz Back-Off y las técnicas de interpolación le agregan flexibilidad a los modelos de trigramas al aprovechar la información de contextos más pequeños ayudando a que no se indefinan probabilidades de n-gramas no observadas en entrenamiento y generalizar mejor a datos desconocidos.

### 3.11 Historia

En 1951, mientras trabajaba en los laboratorios Bell, Claude Shannon (en la Figura 3.1) realizó experimentos sobre la entropía del inglés, modelando el lenguaje escrito de manera estadística y predictiva [Shannon, 1951]. Utilizando modelos de lenguaje de n-gramas, Shannon exploró la dificultad de predecir palabras basándose en las palabras anteriores.

Prediction and Entropy of Printed English  
By C. E. SHANNON  
(Manuscript Received Sept. 15, 1950)

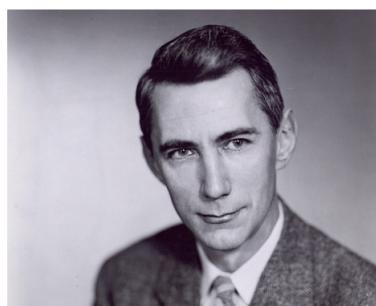


Figura 3.1: Imagen de Shannon.

Por otro lado, en su libro *Syntactic Structures* (1957), Noam Chomsky (en la Figura 4.1), un lingüista y científico cognitivo, cuestionó la capacidad de los modelos de lenguaje probabilísticos para capturar y comprender la gramática del lenguaje humano [Chomsky, 2009]. Según Chomsky, la noción de “gramaticalmente correcto” no puede ser equiparada a “significativo” en un sentido probabilista. Para ilustrar esto, presentó dos oraciones ficticias, ambas carentes de sentido:

1. Colorless green ideas sleep furiously.
2. Furiously sleep ideas green colorless.

Aunque ambas oraciones carecen de significado, Chomsky argumentó que solo la primera se considera gramaticalmente correcta por los hablantes de inglés. Además, enfatizó que la corección gramatical en inglés no puede determinarse únicamente mediante aproximaciones estadísticas.

Aunque es poco probable que ninguna de las dos oraciones (1) o (2) haya surgido en documentos escritos en inglés, un modelo estadístico como los modelos de lenguaje vistos en este capítulo las consideraría igualmente “remotas” en relación al inglés. Sin embargo, la oración (1) es gramaticalmente correcta, mientras que la oración (2) no lo es, lo que destaca las limitaciones de los enfoques estadísticos para capturar la gramática. Estos argumentos retrasaron el estudio de los modelos de lenguaje probabilísticos durante varios años [Jurafsky and Martin, 2023].

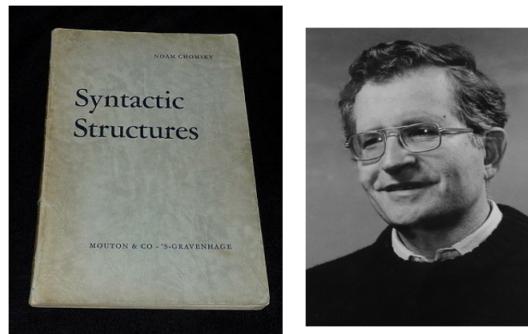


Figura 3.2: Imagen de Chomsky

### 3.12 Conclusiones

El cálculo de probabilidades en modelos de lenguaje probabilísticos implica tres pasos:

1. Expandir  $p(w_1, w_2, \dots, w_n)$  usando la regla de la Cadena.
2. Aplicar los supuestos Independencia de Markov  

$$p(w_i|w_1, w_2, \dots, w_{i-2}, w_{i-1}) = p(w_i|w_{i-2}, w_{i-1}).$$
3. Suavizar las estimaciones utilizando conteos de orden inferior.

No obstante, los modelos de lenguaje de bigramas o trigramas (que consideran dos palabras anteriores como contexto) tienen limitaciones en contextos largos y no pueden aprovechar contextos similares. Por ejemplo, consideremos los contextos:

- $c_1$ : Despues de comer cereales
- $c_2$ : Luego de desayunar avena

Aunque esperaríamos que las distribuciones de probabilidad  $p(w|c_1)$  y  $p(w|c_2)$  fueran similares, dado que  $c_1$  y  $c_2$  casi no comparten palabras, los modelos de n-gramas que se limitan a contar frecuencia de palabras no pueden capturar estas similitudes entre contextos.

Otros métodos para mejorar los modelos de lenguaje incluyen introducir variables latentes para representar tópicos, conocidos como modelos de tópicos [Blei et al., 2003] presentados en el Capítulo 2. O alternativamente, reemplazar  $p(w_i|w_1, w_2, \dots, w_{i-2}, w_{i-1})$  con una red neuronal predictiva y una “capa de embedding” para representar mejor contextos más grandes y aprovechar similitudes entre palabras en el contexto. [Bengio et al., 2000]

Los modelos de lenguaje modernos utilizan redes neuronales profundas en su estructura principal y tienen un vasto espacio de parámetros como se verá en el Capítulo 13.



## 4. Clasificación de Texto y Naïve Bayes

La clasificación, que implica la asignación de un objeto a una categoría específica, desempeña un papel fundamental tanto en la inteligencia humana como en la artificial. En este contexto, la clasificación abarca diversas tareas, que van desde determinar qué letra, palabra o imagen se ha presentado a nuestros sentidos, hasta reconocer caras, voces, clasificar correos electrónicos o calificar tareas.

El propósito subyacente de la clasificación es tomar una única observación, identificar y extraer características relevantes de la misma, y, en última instancia, ubicarla en una de las categorías discretas predefinidas.

**Definición 4.0.1** Formalmente, definimos el problema de clasificación de texto, al escenario en que se tiene un documento  $d$  y un conjunto fijo de clases  $C = \{c_1, c_2, \dots, c_J\}$  y se requiere predecir una clase  $c \in C$  para  $d$ .

Como se discutió en el Capítulo 1, la construcción de clasificadores mediante reglas manuales no resulta ser un enfoque eficaz. Hoy en día la mayoría de las tareas de clasificación en PLN se abordan mediante enfoques de aprendizaje automático supervisado.

Formalmente, se parte de un conjunto fijo de clases  $C = \{c_1, c_2, \dots, c_J\}$  y un conjunto de entrenamiento compuesto por  $m$  documentos que han sido etiquetados manualmente como

$$(d_1, c_1), (d_2, c_2), \dots, (d_m, c_m).$$

A través de un proceso de entrenamiento, se desarrolla un clasificador que se define como una función  $\gamma: d \rightarrow c$ .

Diversos algoritmos de clasificación están disponibles para este propósito, como Naïve Bayes, Regresión logística, Redes neuronales, k-vecinos más cercanos, entre otros. En este capítulo, nos centraremos en el modelo Naïve Bayes. Este capítulo se basa en el material del curso de Daniel Jurafsky, al que se puede acceder a través del siguiente enlace<sup>1</sup>.

<sup>1</sup><https://web.stanford.edu/~jurafsky/slp3/4.pdf>

## 4.1 Ejemplos de Problemas de Clasificación

La clasificación de texto se puede aplicar a varias tareas, incluyendo:

- Análisis de sentimientos
  - Detección de spam
  - Identificación de autoría
  - Identificación de idioma
  - Asignación de categorías, temas o géneros
- Analicemos estos ejemplos en más detalle:

■ **Ejemplo 4.1** Clasificación de spam. En este problema el objeto a clasificar es un correo electrónico y las etiquetas posibles son SPAM y no SPAM. Las etiquetas se obtienen del mismo usuario que etiqueta los correos.

```
Subject: Important notice!
From: Stanford University <newsforum@stanford.edu>
Date: October 28, 2011 12:34:16 PM PDT
To: undisclosed-recipients:;

Greats News!

You can now access the latest news by using the link below to login to Stanford University News Forum.

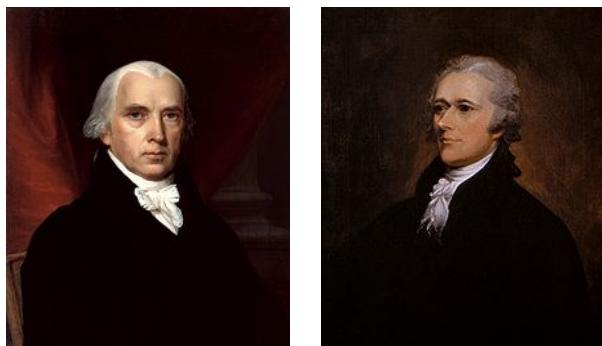
http://www.123contactform.com/contact-form-StanfordNew1-236335.html

Click on the above link to login for more information about this new exciting forum. You can also copy the
above link to your browser bar and login for more information
about the new services.

© Stanford University. All Rights Reserved.
```

Figura 4.1: Ejemplo de Spam

■ **Ejemplo 4.2** Detección de Autoría. Un ejemplo histórico de detección de autoría se refiere a la autoría de los Ensayos Federalistas de la Constitución de EE. UU. En 1787, se escribieron ensayos anónimos para persuadir a Nueva York de ratificar la Constitución de EE. UU. La autoría de 12 de estos ensayos estuvo en disputa entre James Madison y Alexander Hamilton (Tabla 4.1) hasta que en 1963, Mosteller y Wallace [Mosteller and Wallace, 1963] utilizaron métodos bayesianos para identificar que Hamilton era el autor de los mismos.



James Madison

Alexander Hamilton

Cuadro 4.1: Autores candidatos a autoría.

■ **Ejemplo 4.3** Clasificación por tópico. Aquí los ejemplos incluyen clasificar una noticia en una categoría temática (ej: deportes, política, economía) o etiquetar automáticamente un artículo en ciencias de la vida en una categoría del Medical Subject Headings (MeSH) como se ilustra en la Figura 4.2.

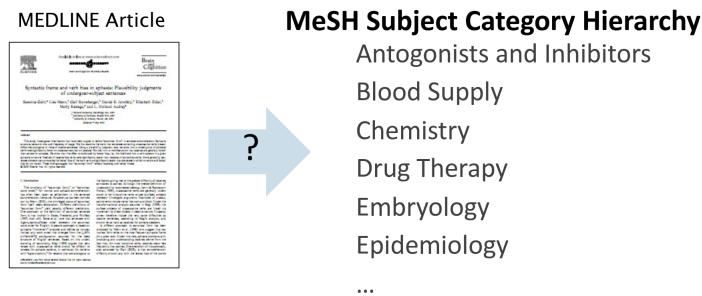


Figura 4.2: Ejemplo de clasificación por categorías clínicas.

■ **Ejemplo 4.4** Clasificación de sentimientos. La clasificación de sentimientos se utiliza para determinar si un documento exhibe un sentimiento positivo o negativo. Este enfoque se basa en el análisis del tono y las palabras utilizadas en el texto. A continuación, se presentan ejemplos de reseñas de películas y comentarios de restaurantes:

- + ...una película muy **buenas**, especialmente por los giros de la trama. ¡Fue **espectacular**!
- - La actuación fue **patética**, destacando las escenas de baile como lo **peor**.
- + ...la salsa de chocolate con almendras dulces es simplemente **increíble** en este lugar. ¡Me **encanta**!
- - ...la pizza estaba **horrible** y tenía un precio **ridículamente** alto.

Este enfoque se aplica en diversos casos, como medir la confianza del consumidor respecto a un producto o predecir resultados electorales y tendencias del mercado en función del sentimiento. Estas aplicaciones suelen ser el resultado de la implementación de modelos de análisis de sentimientos en datos procedentes de las redes sociales.

## 4.2 Modelos Generativos

Nuestra tarea principal consiste en aprender una función  $f$  que asigne etiquetas  $f(d)$  a las entradas  $d$ . El modelo Naïve Bayes sigue un enfoque probabilístico en el que buscamos estimar la probabilidad condicional  $p(c|d)$  utilizando ejemplos de entrenamiento para todas las clases o etiquetas (por ejemplo,  $c = c_1, c = c_2, \dots, c = c_k$ ), donde  $0 \leq p(c = c_j|d) \leq 1$  para cada  $j$  y  $\sum_j p(c = c_j|d) = 1$ . Luego, para cualquier entrada de prueba  $\tilde{d}$ , definimos  $f(\tilde{d}) = c_{\text{MAP}} = \arg \max_c p(c|\tilde{d})$  como el estimador MAP (Maximum a Posteriori), lo cual equivale a seleccionar la clase más probable dada la información disponible en los datos.

Es importante señalar que existen dos enfoques principales en el aprendizaje automático en términos de la modelización de esta probabilidad condicional  $p(c|d)$ . Los modelos que intentan estimar  $p(c|d)$  directamente de los datos se denominan modelos “discriminativos”, como la regresión logística que exploraremos en el Capítulo 5.

Por otro lado, Naïve Bayes se clasifica como un “modelo generativo”. Bajo este enfoque, se busca construir un modelo para cada clase y clasificar una entrada observando cuál clase tiene la mayor probabilidad de generar el ejemplo objetivo.

Este concepto puede formalizarse mediante el teorema de Bayes. Dado un documento  $d$  y una clase  $c$ , tenemos:

$$p(c|d) = \frac{p(d|c)p(c)}{p(d)}$$

Aquí,  $p(d|c)$  representa la “verosimilitud” que representa la probabilidad de que una clase particular “genere” el documento. Por otro lado,  $p(c)$  se refiere a la probabilidad “a priori” de la clase y  $p(d)$  corresponde a la probabilidad del documento, también conocida como “evidencia”.

Dado que solo estamos interesados en encontrar la clase más probable para la clasificación, podemos eliminar el denominador, ya que se mantiene constante para todas las clases. Esto se expresa como:

$$c_{\text{MAP}} = \arg \max_c p(d|c)p(c)$$

En resumen, el estimador MAP se utiliza para encontrar la clase más probable según la expresión anterior.

Alternativamente, podemos entender los modelos generativos como un intento de estimar la distribución conjunta  $p(d, c)$  a partir de los ejemplos de entrenamiento. Utilizando la definición de probabilidad condicional, tenemos  $p(d, c) = p(c)p(d|c)$ , lo cual es equivalente al numerador que maximizamos previamente utilizando el teorema de Bayes.

### 4.3 Naïve Bayes Multinomial

Naïve Bayes, o el modelo Bayesiano “ingenuo” [McCallum et al., 1998], se presenta como un modelo de clasificación generativo en el que un documento  $d$  se representa como una bolsa de palabras  $x_1, x_2, \dots, x_n$ . Similar al modelo vectorial explorado en el Capítulo 2, en este modelo se omiten las posiciones de las palabras dentro del documento.

La verosimilitud del modelo probabilístico se expresa de la siguiente manera:

$$p(d|c) = p(x_1, x_2, \dots, x_n|c)$$

Al estimar esta expresión sin hacer supuestos de independencia (es decir, contando las ocurrencias de  $x_1, x_2, \dots, x_n$  en cada clase), nos enfrentamos al mismo problema que se presenta en los modelos de lenguaje de n-gramas discutidos en el Capítulo 3. A menos que tengamos un corpus infinitamente grande, este enfoque asignaría probabilidades nulas a las nuevas oraciones en las que se aplica el modelo, ya que es muy poco probable que éstas aparezcan en el corpus de entrenamiento.

Por lo tanto, recurrimos al supuesto de independencia condicional, que postula que las palabras son independientes entre sí cuando se condicionan a la clase. Esto nos permite factorizar la verosimilitud como el producto de factores individuales para cada palabra en el vocabulario:

$$p(x_1, x_2, \dots, x_n|c) = p(x_1|c) \cdot p(x_2|c) \cdot p(x_3|c) \cdot \dots \cdot p(x_n|c) = \prod_i p(x_i|c)$$

Con esta factorización, la estimación MAP del modelo para clasificar un documento se formula de la siguiente manera:

$$c_{\text{MAP}} = \arg \max_{c \in C} \prod_i p(x_i | c) p(c)$$

#### 4.3.1 Estimación de parámetros

Para estimar los parámetros del modelo Naive Bayes multinomial, se utiliza el método de estimación por máxima verosimilitud, asumiendo una distribución multinomial para  $p(x_1, x_2, \dots, x_n | c)$ <sup>2</sup>. Esto implica la creación de un “mega-documento” para cada clase  $c_j$  mediante la concatenación de todos los documentos de esa clase. Luego, se calcula la frecuencia de todas las palabras  $w_i$  del vocabulario en el mega-documento para cada clase  $\text{count}(w_i, c_j)$ .

La probabilidad estimada  $\hat{p}(w_i | c_j)$  de la palabra  $w_i$  dada la clase  $c_j$  se obtiene dividiendo el conteo de ocurrencias de  $w_i$  en el mega-documento de la clase  $c_j$  por el total de palabras en el mega-documento:

$$\hat{P}(w_i | c_j) = \frac{\text{count}(w_i, c_j)}{\sum_{w \in V} \text{count}(w, c_j)}$$

La probabilidad previa de una clase  $c_j$ ,  $p(c_j)$ , se estima de la siguiente manera:

$$\hat{p}(c_j) = \frac{N_{c_j}}{N_{\text{total}}}$$

donde  $N_{c_j}$  es el número de documentos en la clase  $c_j$  y  $N_{\text{total}}$  es el número total de documentos.

Una vez que el modelo Naïve Bayes (NB) ha sido entrenado, se puede emplear para clasificar un nuevo documento  $\tilde{d}$  de la siguiente manera:

$$c_{\text{NB}} = \arg \max_{c_j} p(c_j) \prod_{i \in \text{posiciones}} p(x_i | c_j)$$

Básicamente, se debe iterar a través de las palabras en las posiciones del documento y calcular las probabilidades  $p(c_j)$  y  $p(x_i | c_j)$  para todas las clases y palabras del documento, para finalmente retornar la clase más probable. Estas probabilidades se derivan de los conteos almacenados durante el proceso de entrenamiento.

Cuando el documento de prueba  $\tilde{d}$  contiene palabras que son desconocidas, es decir, no se encuentran en los datos de entrenamiento ni en el vocabulario, optamos por omitirlas. En este contexto, no asignamos ninguna probabilidad a estas palabras desconocidas durante el proceso de clasificación.

#### 4.3.2 Problemas al multiplicar muchas probabilidades

La multiplicación de muchas probabilidades puede llevar a un underflow de punto flotante<sup>3</sup>, especialmente cuando se manejan probabilidades pequeñas. Por ejemplo,  $0,0006 \times 0,0007 \times 0,0009 \times 0,01 \times 0,5 \times 0,000008 \dots$  Esto resulta en productos con valores nulos y clasificaciones erróneas.

---

<sup>2</sup>Existe otra variante que asume una distribución Binomial pero no funciona tan bien para texto.

<sup>3</sup>[https://en.wikipedia.org/wiki/Arithmetic\\_underflow](https://en.wikipedia.org/wiki/Arithmetic_underflow)

Para abordar este problema, podemos recurrir a los logaritmos, ya que  $\log(ab) = \log(a) + \log(b)$ . En lugar de multiplicar las probabilidades, sumamos los logaritmos de las probabilidades. Así, el clasificador Naïve Bayes multinomial se puede expresar utilizando logaritmos de la siguiente manera:

$$c_{\text{NB}} = \arg \max_{c_j \in C} \left( \log(P(c_j)) + \sum_{i \in \text{posiciones}} \log(P(x_i|c_j)) \right)$$

La función logaritmo es monótona creciente, lo que significa que si una probabilidad es mayor que otra, el logaritmo de esa probabilidad también será mayor que el logaritmo de la otra probabilidad. Esto preserva el orden relativo de las probabilidades, por lo que el resultado óptimo de la clasificación no se ve afectado.

Al aplicar logaritmos, el clasificador Naïve Bayes se convierte en un modelo lineal (Capítulo 5), donde la predicción es el argmax de la suma de los logaritmos de las probabilidades y las entradas (logaritmos de las probabilidades condicionales). En otras palabras, Naïve Bayes se convierte en un clasificador lineal que opera en el espacio logarítmico.

#### 4.3.3 Suavizamiento de Laplace

De forma similar a los modelos de lenguaje de n-gramas mencionados en el Capítulo 3, Naïve Bayes también enfrenta problemas de probabilidades nulas.

Supongamos que estamos trabajando en un problema de clasificación de sentimientos con clases positivo y negativo. Por alguna razón, la palabra “fantástico” no aparece en ningún documento negativo. Luego, imaginemos que queremos clasificar el documento  $\tilde{d}$  = “Qué fantástico, acabo de aburrirme como nunca viendo la peor película de mi vida”. Claramente, este documento es negativo, pero si usamos la estimación de máxima verosimilitud, la probabilidad  $\hat{p}(\text{“fantástico”} | \text{negativo})$  sería:

$$\hat{p}(\text{“fantástico”} | \text{negativo}) = \frac{\text{count(“fantástico”, negativo)}}{\sum_{w \in V} \text{count}(w, \text{negativo})} = \frac{0}{\sum_{w \in V} \text{count}(w, \text{positivo})} = 0$$

El problema es que esta probabilidad cero anularía toda la evidencia proporcionada por otras palabras muy indicativas de la clase negativa, como “aburrirme” y “peor” por la forma multiplicativa de la función de verosimilitud del modelo, y el clasificador Naïve Bayes terminaría clasificando el ejemplo en la clase “positiva”.

Para abordar este problema, podemos utilizar la técnica de suavizamiento de Laplace, que consiste en agregar un conteo adicional a todas las combinaciones de palabras y clases y ajustar el denominador con el tamaño del vocabulario  $V$  para garantizar una normalización adecuada. La estimación suavizada  $\hat{p}(w_i | c)$  queda de la siguiente manera:

$$\hat{p}(w_i | c) = \frac{\text{count}(w_i, c) + 1}{\sum_{w \in V} (\text{count}(w, c) + 1)} = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|}$$

Esta técnica aborda el problema de las probabilidades nulas, ya que ahora cualquier palabra del vocabulario, incluso si nunca se ha visto con alguna clase, recibirá una probabilidad condicional mayor a cero. Como pueden ver, esta técnica permite que cierta masa de probabilidad se distribuya a palabras no vistas de manera similar a las técnicas de descuento vistas en el Capítulo de modelos de lenguaje (Capítulo 3).

■ **Ejemplo 4.5** Supongamos que tenemos el conjunto de entrenamiento mostrado en la Tabla 4.2 y queremos clasificar el ejemplo de la Tabla 4.3.

Categoría	Texto
Perú	Lima Lima Trujillo
Chile	Santiago Concepción Concepción
Perú	Perú Lima Trujillo América
Chile	Chile América Santiago

Cuadro 4.2: Datos de Entrenamiento.

Categoría	Texto
?	América Lima Perú Santiago

Cuadro 4.3: Ejemplo de Prueba.

En este ejemplo,  $|V| = 7$ , y tenemos dos clases:  $c_1 = \text{Perú}$  y  $c_2 = \text{Chile}$ .

La Tabla 4.4 muestra los conteos calculados a partir de los datos de entrenamiento.

id_palabra	palabra	count( $w_i$ , Perú)	count( $w_i$ , Chile)
$w_1$	América	1	1
$w_2$	Chile	0	1
$w_3$	Concepción	0	2
$w_4$	Lima	3	0
$w_5$	Perú	1	0
$w_6$	Santiago	0	2
$w_7$	Trujillo	2	0

Cuadro 4.4: Tabla de Conteos.

Ahora, debemos calcular las probabilidades condicionales suavizadas:

$$\hat{p}(w_i | c) = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|}$$

para todas las palabras y clases<sup>4</sup>.

El denominador de esta expresión para cada clase corresponde al largo del “mega-documento” que junta todos los documentos de la misma clase más el tamaño del vocabulario, que sería  $7 + 7 = 14$  para Perú y  $6 + 7 = 13$  para Chile.

Las probabilidades resultantes se muestran en la Tabla 4.5.

Ahora necesitamos estimar la probabilidad a priori de cada clase:

$$\hat{p}(c_j) = \frac{N_{c_j}}{N_{\text{total}}}$$

Ambas clases ocurren en 2 documentos sobre un total de 4. Entonces,  $\hat{p}(c = \text{Perú}) = 0,5$  y  $\hat{p}(c = \text{Chile}) = 0,5$ .

<sup>4</sup>Alternativamente podríamos sumar los logartimos de las probabilidades.

id_palabra	palabra	$\hat{p}(w_i   c = \text{Perú})$	$\hat{p}(w_i   c = \text{Chile})$
$w_1$	América	$\frac{1+1}{14} = 0,14$	$\frac{1+1}{13} = 0,15$
$w_2$	Chile	$\frac{0+1}{14} = 0,07$	$\frac{1+1}{13} = 0,15$
$w_3$	Concepción	$\frac{0+1}{14} = 0,07$	$\frac{2+1}{13} = 0,23$
$w_4$	Lima	$\frac{3+1}{14} = 0,29$	$\frac{0+1}{13} = 0,08$
$w_5$	Perú	$\frac{1+1}{14} = 0,14$	$\frac{0+1}{13} = 0,08$
$w_6$	Santiago	$\frac{0+1}{14} = 0,07$	$\frac{2+1}{13} = 0,23$
$w_7$	Trujillo	$\frac{2+1}{14} = 0,21$	$\frac{0+1}{13} = 0,08$

Cuadro 4.5: Probabilidades condicionales suavizadas.

Ahora debemos calcular:

$$c_{\text{NB}} = \arg \max_{c_j} p(c_j) \prod_{i \in \text{posiciones}} p(x_i | c_j)$$

Esto implica iterar posiciones para ambas clases del documento de prueba:

$$c_{\text{Perú}} = 0,14 \times 0,29 \times 0,14 \times 0,07 \times 0,5 = 0,00019894$$

$$c_{\text{Chile}} = 0,15 \times 0,08 \times 0,08 \times 0,23 \times 0,5 = 0,0001104$$

Entonces,  $c_{\text{NB}} = \text{"Perú"}$  y clasificamos el documento a la clase "Perú". ■

#### 4.3.4 Naïve Bayes como modelo de lenguaje

A continuación, exploraremos una interpretación del modelo Naïve Bayes utilizando conceptos de modelos de lenguaje. El supuesto de independencia condicional de palabras en Naïve Bayes lo asemeja mucho a un modelo de lenguaje de unigramas que se discutió en el Capítulo 3.

Específicamente, la función de verosimilitud en el modelo Naïve Bayes multinomial se puede entender como la construcción de un modelo de lenguaje de unigramas para cada clase.

■ **Ejemplo 4.6** Por ejemplo, consideremos un problema de clasificación de sentimientos con las clases "positivo" y "negativo" y los siguientes parámetros del modelo:

Palabra	$p(w c = \text{"positivo"})$	$p(w c = \text{"negativo"})$
Yo	0.1	0.2
amé	0.1	0.001
esta	0.01	0.01
gran	0.05	0.005
película	0.1	0.1
...	...	...

Cada una de las dos columnas anteriores se puede interpretar tanto como las probabilidades condicionales del modelo Naïve Bayes como las probabilidades de dos modelos de lenguaje de unigramas construidos con los datos de entrenamiento de cada clase. Para ilustrar esto, supongamos que deseamos clasificar la oración  $d = \text{"Yo amé esta gran película"}$ . Podemos calcular:

$$p(d|\text{"positivo"}) = 0,1 \times 0,1 \times 0,01 \times 0,05 \times 0,1 = 0,0000005$$

$$p(d|\text{"negativo"}) = 0,2 \times 0,001 \times 0,01 \times 0,005 \times 0,1 = 0,000000010$$

Observamos que el modelo positivo asigna una probabilidad más alta a la oración:

$$p(d|\text{"positivo"}) > p(d|\text{"negativo"})$$

Este cálculo puede interpretarse indistintamente como la probabilidad asignada por cada modelo de lenguaje a la oración objetivo o como el resultado de la función de verosimilitud del modelo Naïve Bayes. ■

Es importante destacar que en este análisis estamos utilizando solo la parte de verosimilitud del modelo Naïve Bayes. Una vez que multiplicamos por la probabilidad a priori de las clases en Naïve Bayes, podríamos obtener resultados diferentes a los obtenidos con el modelo de lenguaje.

#### 4.4 Evaluación

La evaluación juega un papel crucial en la construcción de sistemas de PLN basados en aprendizaje automático. Al construir estos sistemas, es común llevar a cabo comparaciones y experimentos con diversos modelos durante el proceso de entrenamiento, con el fin de elegir el modelo más idóneo para su posterior implementación en producción. Generalmente, adoptamos un enfoque cuantitativo para evaluar modelos, contrastando las predicciones realizadas con las predicciones esperadas, y resumiendo estos resultados mediante métricas de evaluación. Este enfoque cuantitativo se prefiere debido a que un enfoque cualitativo o de inspección manual no resulta escalable. Ilustremos este proceso de evaluación con un ejemplo.

■ **Ejemplo 4.7** Supongamos un problema de clasificación binaria, para la cual queremos comparar dos modelos de clasificación  $m_1$  y  $m_2$ . Probamos los modelos sobre un conjunto de datos de prueba y obtenemos las predicciones señaladas en la Tabla 4.6.

Texto	Clase	$m_1$	$m_2$
no estoy bien	negativo	negativo	positivo
al fin!	positivo	positivo	negativo
que pena	negativo	negativo	negativo
feliz	positivo	positivo	positivo
no quiero	negativo	positivo	negativo
jamás!	negativo	negativo	negativo
buenas ideas	positivo	positivo	negativo
chao contigo	negativo	positivo	negativo

Cuadro 4.6: Sentimiento de texto

Se puede observar que tanto el modelo  $m_1$  como el modelo  $m_2$  presentan errores en sus predicciones al compararse con las categorías reales de los ejemplos (columna clase), también conocidas como “gold labels” o “ground truth”. Sin embargo, ¿cómo determinamos cuál modelo es mejor? Para abordar esta pregunta, resulta conveniente calcular métricas de desempeño, las cuales se derivan al contrastar las predicciones de un modelo con sus salidas esperadas. ■

#### 4.4.1 Matriz de Confusión

La mayoría de las métricas de evaluación para problemas de clasificación se derivan de una matriz de confusión (véase Tabla 4.7), la cual representa una tabla de contingencia entre las predicciones de un modelo y las salidas esperadas en un conjunto de prueba. Para problemas de clasificación binaria, esta matriz consta de 4 componentes:

1. Verdadero Positivo (VP): cuando el modelo predice correctamente la clase positiva, coincidiendo con la salida esperada.
2. Verdadero Negativo (VN): cuando el modelo predice correctamente la clase negativa, coincidiendo con la salida esperada.
3. Falso Negativo (FN): cuando el modelo predice incorrectamente la clase negativa, siendo la salida esperada positiva.
4. Falso Positivo (FP): cuando el modelo predice incorrectamente la clase positiva, siendo la salida esperada negativa.

Es importante destacar que la definición de qué constituye la clase positiva es arbitraria en un problema de clasificación; generalmente se refiere a la clase que se desea detectar. En el caso de la clasificación de sentimientos, es una coincidencia que la clase positiva se asocie con el sentimiento “positivo”. Esta consideración es fundamental, ya que el valor de algunas métricas de desempeño (como precisión, recall y  $F_1$ ) depende de la elección de la clase positiva.

	Modelo Positivo	Modelo Negativo
Real Positivo	Verdadero Positivo (VP)	Falso Negativo (FN)
Real Negativo	Falso Positivo (FP)	Verdadero Negativo (VN)

Cuadro 4.7: Matriz de confusión

#### 4.4.2 Métricas de Desempeño

**Recall** (exhaustividad) (también conocido como **Sensibilidad** o **Tasa de Verdaderos Positivos**):

$$\text{Recall} = \frac{VP}{VP + FN}$$

**Precisión:**

$$\text{Precisión} = \frac{VP}{VP + FP}$$

**Accuracy** (exactitud):

$$\text{Accuracy} = \frac{VP + VN}{VP + FP + VN + FN}$$

¿Por qué no usamos la exactitud como nuestra métrica?

Imagina que vimos 1 millón de tweets:

- 100 de ellos hablaban sobre Delicious Pie Co.
- 999,900 hablaban de otra cosa.

Podríamos construir un clasificador tonto que simplemente etiquete todos los tweets como "no sobre pasteles":

- ¡¡¡Obtendría una exactitud del 99.99%!!! ¡¡¡Wow!!!

- ¡Pero sería inútil! ¡No devuelve los comentarios que estamos buscando!  
Por eso usamos precisión y recall en su lugar.

**Precisión** mide el porcentaje de elementos que el sistema detectó (es decir, los elementos que el sistema etiquetó como positivos) que son realmente positivos (según las etiquetas de oro humanas).

$$\text{Precisión} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Positivos}}$$

**Recall** mide el porcentaje de elementos que el sistema identificó correctamente de todos los elementos que deberían haber sido identificados.

$$\text{Recall} = \frac{\text{Verdaderos Positivos}}{\text{Verdaderos Positivos} + \text{Falsos Negativos}}$$

Considera nuestro clasificador tonto de pasteles que simplemente etiqueta nada como "sobre pasteles".

- Exactitud = 99.99 % (etiqueta correctamente la mayoría de los tweets como no relacionados con pasteles)
  - Recall = 0 (no detecta ninguno de los 100 tweets relacionados con pasteles)
- La precisión y el recall, a diferencia de la exactitud, enfatizan los verdaderos positivos:
- Se centran en encontrar las cosas que se supone que debemos buscar.

#### 4.4.3 Una Medida Combinada: Medida F

La medida F es un número único que combina la precisión (P) y el recall (R), definida como:

$$F_{\beta} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

La medida F, definida con el parámetro  $\beta$ , pondera diferencialmente la importancia del recall y la precisión.

- $\beta > 1$  favorece al recall
- $\beta < 1$  favorece a la precisión

Cuando  $\beta = 1$ , la precisión y el recall son iguales, y tenemos la medida  $F_1$  equilibrada:

$$F_1 = \frac{2PR}{P+R}$$

#### 4.4.4 Conjuntos de Prueba de Desarrollo ("Devsets")

- Para evitar el sobreajuste y proporcionar una estimación más conservadora del rendimiento, comúnmente utilizamos un enfoque de tres conjuntos: conjunto de entrenamiento, conjunto de desarrollo y conjunto de prueba.

Training set

Development Test Set

Test Set

- **Conjunto de entrenamiento:** Se utiliza para entrenar el modelo.

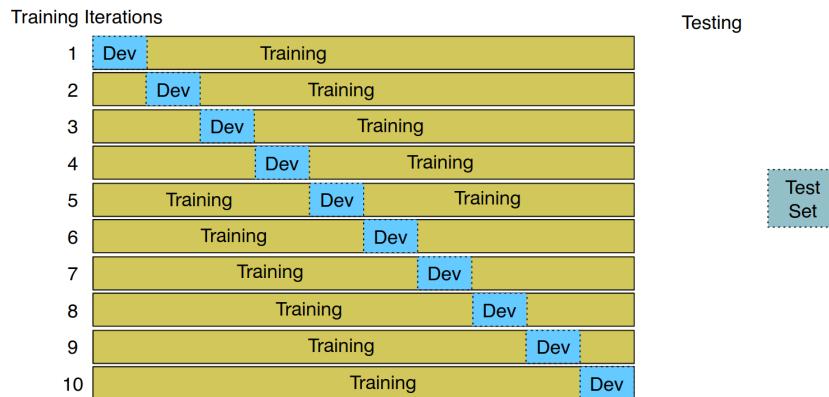
- **Conjunto de desarrollo:** Se utiliza para ajustar el modelo y seleccionar los mejores hiperparámetros.
- **Conjunto de prueba:** Se utiliza para informar el rendimiento final del modelo.
- Este enfoque garantiza que el modelo no esté ajustado específicamente al conjunto de prueba, evitando el sobreajuste.
- Sin embargo, crea una paradoja: queremos la mayor cantidad de datos posible para el entrenamiento, pero también para el conjunto de desarrollo.
- ¿Cómo dividimos los datos?

#### 4.4.5 Validación Cruzada: Múltiples Divisiones

La validación cruzada nos permite utilizar todos nuestros datos para el entrenamiento y la prueba sin tener un conjunto de entrenamiento, conjunto de desarrollo y conjunto de prueba fijos. Elegimos un número  $k$  y dividimos nuestros datos en  $k$  subconjuntos disjuntos llamados pliegues. En cada iteración, uno de los pliegues se selecciona como conjunto de prueba mientras que los  $k - 1$  pliegues restantes se utilizan para entrenar el clasificador.

Calculamos la tasa de error en el conjunto de prueba y repetimos este proceso  $k$  veces. Finalmente, promediamos las tasas de error de estas  $k$  ejecuciones para obtener una tasa de error promedio.

Por ejemplo, la validación cruzada de 10 pliegues implica entrenar 10 modelos con el 90 % de los datos y probar cada modelo por separado. Las tasas de error resultantes se promedian para obtener la estimación final del rendimiento. Sin embargo, la validación cruzada requiere que todo el corpus sea ciego, lo que impide examinar los datos para sugerir características o comprender el comportamiento del sistema. Para abordar esto, se crea un conjunto de entrenamiento y un conjunto de prueba fijos, y se realiza la validación cruzada de 10 pliegues dentro del conjunto de entrenamiento. La tasa de error se calcula convencionalmente en el conjunto de prueba.



## 4.5 Conjuntos de entrenamiento, prueba y validación

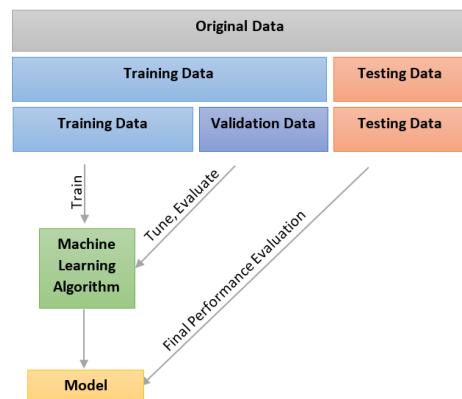
Cuando entrenamos un modelo, nuestro objetivo es producir una función  $f(\vec{x})$  que mapee correctamente las entradas  $\vec{x}$  a las salidas  $\hat{y}$  según lo evidenciado por el conjunto de entrenamiento. La evaluación del rendimiento en los datos de entrenamiento puede ser engañosa, ya que nuestro objetivo es entrenar una función capaz de generalizar a ejemplos no vistos. Una forma común de abordar esto es dividir el conjunto de entrenamiento en subconjuntos de entrenamiento y prueba

(80% y 20% respectivamente). Se entrena el modelo en el subconjunto de entrenamiento y se calcula la precisión en el subconjunto de prueba.

Sin embargo, este enfoque tiene una limitación. En la práctica, a menudo se entrenan varios modelos, se comparan sus calidades y se selecciona el mejor. Si se selecciona el mejor modelo en función de la precisión en el subconjunto de prueba, se obtendrá una estimación excesivamente optimista de la calidad del modelo. No se sabe si la configuración elegida del clasificador final es buena en general o simplemente es buena para los ejemplos particulares en los subconjuntos de prueba.

La metodología aceptada es utilizar una división de tres vías de los datos en conjuntos de entrenamiento, validación (también llamado desarrollo) y prueba<sup>5</sup>. Esto proporciona dos conjuntos apartados: un conjunto de validación (también llamado conjunto de desarrollo) y un conjunto de prueba. Todos los experimentos, ajustes, análisis de errores y selección de modelos deben realizarse

basados en el conjunto de validación. Luego, una única ejecución del modelo final sobre el conjunto de prueba proporcionará una buena estimación de su calidad esperada en ejemplos no vistos. Es importante mantener el conjunto de prueba lo más limpio posible, realizando la menor cantidad de experimentos posible en él. Incluso algunos defienden que no se deben mirar siquiera los ejemplos en el conjunto de prueba, para evitar sesgar el diseño del modelo.



#### 4.5.1 Matriz de Confusión para clasificación de 3 clases

		gold labels			
		urgent	normal	spam	
system output	urgent	8	10	1	$\text{precision}_u = \frac{8}{8+10+1}$
	normal	5	60	50	$\text{precision}_n = \frac{60}{5+60+50}$
	spam	3	30	200	$\text{precision}_s = \frac{200}{3+30+200}$
		$\text{recall}_u = \frac{8}{8+5+3}$	$\text{recall}_n = \frac{60}{10+60+30}$	$\text{recall}_s = \frac{200}{1+50+200}$	

<sup>5</sup>Un enfoque alternativo es la validación cruzada, pero no se escala bien para entrenar redes neuronales profundas.

Fuente: <https://www.codeproject.com/KB/AI/1146582/validation.PNG>

Cómo combinar métricas binarias (Precisión, Recall,  $F_1$ ) de más de 2 clases para obtener una métrica única:

- Macro-promedio:

- Calcular las métricas de rendimiento (Precisión, Recall,  $F_1$ ) para cada clase individualmente.
- Promediar las métricas en todas las clases.

- Micro-promedio:

- Recopilar las decisiones para todas las clases en una matriz de confusión.
- Calcular la Precisión y el Recall a partir de la matriz de confusión.

Class 1: Urgent		Class 2: Normal		Class 3: Spam		Pooled	
true	true	true	true	true	true	true	true
urgent	not	normal	not	spam	not	yes	no
system	8	11	system	60	55	system	268
urgent	8	340	normal	40	212	spam	99
system			system			not	635
not			not				

$\text{precision} = \frac{8}{8+11} = .42$        $\text{precision} = \frac{60}{60+55} = .52$        $\text{precision} = \frac{200}{200+33} = .86$        $\text{microaverage precision} = \frac{268}{268+99} = .73$   
 $\text{macroaverage precision} = \frac{.42+.52+.86}{3} = .60$



## 5. Modelos Lineales

Cuando hacemos aprendizaje supervisado, en esencia tenemos que buscar sobre el conjunto de todas las posibles funciones que pueden mapear desde las entradas hacia las salidas. Si no acotamos el espacio de búsqueda de alguna forma nos encontramos con un problema muy difícil (y bastante mal definido) [Goldberg, 2017]. A menudo nos limitamos a buscar dentro de familias específicas de funciones. Por ejemplo, el espacio de todas las funciones lineales con  $d_{in}$  entradas y  $d_{out}$  salidas. Estas familias de funciones se llaman clases de hipótesis. Al restringirnos a una clase de hipótesis específica, estamos inyectando al aprendiz con sesgo inductivo, es decir, un conjunto de suposiciones sobre la forma de la solución deseada. Algunas clases de hipótesis facilitan procedimientos eficientes para buscar la solución.

Una clase de hipótesis común es la de una función lineal de alta dimensión:

$$f(\vec{x}) = \vec{x} \cdot W + \vec{b} \quad (5.1)$$
$$\vec{x} \in \mathcal{R}^{d_{in}} \quad W \in \mathcal{R}^{d_{in} \times d_{out}} \quad \vec{b} \in \mathcal{R}^{d_{out}}$$

El vector  $\vec{x}$  es la entrada de la función. La matriz  $W$  y el vector  $\vec{b}$  son los parámetros. El objetivo del aprendizaje es establecer los valores de los parámetros  $W$  y  $\vec{b}$  de manera que la función se comporte como se pretende en una colección de valores de entrada  $\vec{x}_{1:k} = \vec{x}_1, \dots, \vec{x}_k$  y las correspondientes salidas deseadas  $\vec{y}_{1:k} = \vec{y}_1, \dots, \vec{y}_k$ . La tarea de buscar en el espacio de funciones se reduce así a buscar en el espacio de parámetros.

### 5.1 Ejemplo: Detección de Idiomas

Consideremos la tarea de distinguir entre documentos escritos en inglés y documentos escritos en alemán. Este es un problema de clasificación binaria:

$$f(\vec{x}) = \vec{x} \cdot \vec{w} + b \quad (5.2)$$

donde  $d_{out} = 1$ , donde  $\vec{w}$  es un vector y  $b$  es un escalar. El rango de la función lineal es  $[-\infty, \infty]$ . Para usarla en la clasificación binaria, es común pasar la salida de  $f(x)$  a través de la función *signo*, mapeando los valores negativos a -1 (clase negativa) y los valores no negativos a +1 (clase positiva).

Las frecuencias de letras son buenos predictores (características) para esta tarea, pero aún más informativos son los recuentos de bigramas de letras, es decir, pares de letras consecutivas. Es probable que nos encontremos con un documento nuevo sin ninguna de las palabras que observamos en el conjunto de entrenamiento, mientras que un documento sin ninguno de los distintivos bigramas de letras es significativamente menos probable [Goldberg, 2017]. Supongamos que tenemos un alfabeto de 28 letras (a-z, espacio y un símbolo especial para todos los demás caracteres, incluidos dígitos, puntuación, etc.). Los documentos se representan como vectores de dimensión  $28 \times 28$ , es decir,  $\vec{x} \in \mathcal{R}^{784}$ . Cada entrada  $\vec{x}_{[i]}$  representa un recuento de una combinación particular de letras en el documento, normalizado por la longitud del documento. Por ejemplo, si denotamos por  $\vec{x}_{ab}$  la entrada de  $\vec{x}$  correspondiente al bigrama de letras  $ab$ :

$$\vec{x}_{ab} = \frac{\#ab}{|D|} \quad (5.3)$$

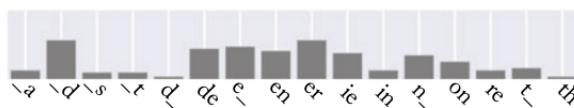
donde  $\#ab$  es el número de veces que aparece el bigrama  $ab$  en el documento y  $|D|$  es el número total de bigramas en el documento (la longitud del documento).

La figura muestra histogramas de bigramas de caracteres para documentos en inglés y alemán. Los guiones bajos representan espacios. Solo se muestran los bigramas de caracteres más frecuentes.



Fuente: [Goldberg, 2017]

La figura anterior muestra patrones claros en los datos. Dado un nuevo elemento, por ejemplo:



Probablemente podríamos decir que es más similar al grupo alemán que al grupo inglés (observar la frecuencia de "thz ie"). No podemos usar una regla única definitiva como "si tiene th es inglés".

"si tiene ie es alemán". Aunque los textos en alemán tienen considerablemente menos "th" que el inglés, la combinación "th" puede ocurrir en textos en alemán, al igual que la combinación "ie" puede ocurrir en inglés. La decisión requiere ponderar diferentes factores relativos entre sí.

Podemos formalizar el problema en un entorno de aprendizaje automático utilizando un modelo lineal:

$$\hat{y} = \text{sign}(\vec{x} \cdot \vec{w} + b) = \text{sign}(\vec{x}_{aa} \times \vec{w}_{aa} + \vec{x}_{ab} \times \vec{w}_{ab} + \vec{x}_{ac} \times \vec{w}_{ac} \cdots + b) \quad (5.4)$$

Un documento se considerará inglés si  $f(\vec{x}) \geq 0$  y alemán en caso contrario. La intuición detrás del aprendizaje es la siguiente:

1. El aprendizaje debe asignar valores positivos grandes a las entradas de  $\vec{w}$  asociadas con pares de letras que son mucho más comunes en inglés que en alemán (por ejemplo, "th").
2. También debe asignar valores negativos a los pares de letras que son mucho más comunes en alemán que en inglés (por ejemplo, "ie", ".en").
3. Finalmente, debe asignar valores alrededor de cero a los pares de letras que son comunes o raros en ambos idiomas.

## 5.2 Clasificación binaria log-lineal

La salida  $f(\vec{x})$  se encuentra en el rango  $[-\infty, \infty]$ , y la mapeamos a una de las dos clases  $\{-1, +1\}$  utilizando la función *signo*. Esto es adecuado si lo único que nos importa es la clase asignada. Sin embargo, puede que también estemos interesados en la confianza de la decisión o en la probabilidad que el clasificador asigna a la clase.

Una alternativa que facilita esto es mapear la salida al rango  $[0, 1]$  mediante una función de compresión como la función sigmoide  $\sigma(x)$ :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (5.5)$$

lo que resulta en:

$$\hat{y} = \sigma(f(\vec{x})) = \frac{1}{1 + e^{-\vec{x} \cdot \vec{w} + b}} \quad (5.6)$$

La función sigmoide es monótona creciente y mapea los valores al rango  $[0, 1]$ , con 0 mapeado a  $\frac{1}{2}$ . Cuando se utiliza con una función de pérdida adecuada (que se discutirá más adelante), las predicciones binarias realizadas mediante el modelo log-lineal se pueden interpretar como estimaciones de la probabilidad de pertenencia a la clase:

$$\sigma(f(\vec{x})) = P(\hat{y} = 1 | \vec{x}) \quad \text{de que } \vec{x} \text{ pertenezca a la clase positiva.} \quad (5.7)$$

También obtenemos  $P(\hat{y} = 0 | \vec{x}) = 1 - P(\hat{y} = 1 | \vec{x}) = 1 - \sigma(f(\vec{x}))$ . Cuanto más cercano sea el valor a 0 o 1, más seguro está el modelo en su predicción de pertenencia a la clase, y el valor 0,5 indica incertidumbre del modelo.

### 5.3 Clasificación multiclasa

La mayoría de los problemas de clasificación son de naturaleza multiclasa: se asignan ejemplos a una de las  $k$  clases diferentes. Por ejemplo, se nos puede dar un documento y se nos pide clasificarlo en uno de los seis posibles idiomas: inglés, francés, alemán, italiano, español y otros.

Una solución posible es considerar seis vectores de pesos  $\vec{w}_{EN}, \vec{w}_{FR}, \dots$  y sesgos (uno para cada idioma). Predecimos el idioma que resulta en el puntaje más alto:

$$\hat{y} = f(\vec{x}) = \operatorname{argmax}_{L \in \{EN, FR, GR, IT, SP, O\}} \quad \vec{x} \cdot \vec{w}_L + b_L \quad (5.8)$$

Los seis conjuntos de parámetros  $\vec{w}_L \in \mathcal{R}^{784}$  y  $b_L$  se pueden organizar en una matriz  $W \in \mathcal{R}^{784 \times 6}$  y un vector  $\vec{b} \in \mathcal{R}^6$ , y la ecuación se puede reescribir como:

$$\begin{aligned} \vec{y} &= f(\vec{x}) = \vec{x} \cdot W + \vec{b} \\ \text{predicción} &= \hat{y} = \operatorname{argmax}_i \hat{y}_{[i]} \end{aligned} \quad (5.9)$$

Aquí,  $\vec{y} \in \mathcal{R}^6$  es un vector de los puntajes asignados por el modelo a cada idioma, y determinamos el idioma predicho tomando el argmax sobre las entradas de  $\vec{y}$  (las columnas con el valor más alto).

### 5.4 Representaciones

Consideremos el vector  $\vec{y}$  resultante de aplicar un modelo entrenado a un documento. Podemos considerar que este vector es una representación del documento, ya que captura las propiedades del documento que son importantes para nosotros, es decir, los puntajes de los diferentes idiomas. La representación  $\vec{y}$  contiene estrictamente más información que la predicción  $\operatorname{argmax}_i \hat{y}_{[i]}$ .

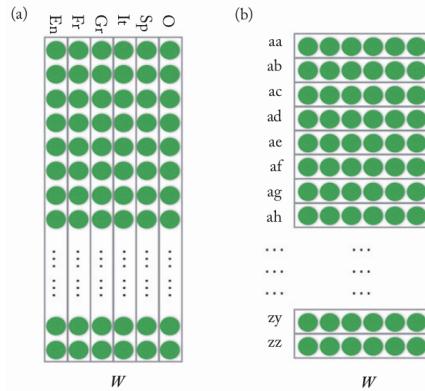
Por ejemplo,  $\vec{y}$  se puede utilizar para distinguir documentos en los que el idioma principal es el alemán, pero que también contienen una cantidad considerable de palabras en francés. El agrupamiento de documentos basado en  $\vec{y}$  podría ayudar a descubrir documentos escritos en dialectos regionales o por autores multilingües.

Los vectores  $\vec{x}$  que contienen los recuentos normalizados de los bigramas de letras para los documentos también son representaciones de los documentos. Se podría argumentar que contienen un tipo de información similar a los vectores  $\vec{y}$ . Sin embargo, las representaciones en  $\vec{y}$  son más compactas (6 entradas en lugar de 784) y más especializadas para el objetivo de predicción de idioma. El agrupamiento por los vectores  $\vec{x}$  probablemente revelaría similitudes en los documentos que no se deben a una mezcla particular de idiomas, sino tal vez al tema o estilo de escritura del documento.

La matriz entrenada  $W \in \mathcal{R}^{784 \times 6}$  también se puede considerar como una representación aprendida. Podemos considerar dos vistas de  $W$ , como filas o como columnas.

Dos vistas de la matriz  $W$ . (a) Cada columna corresponde a un idioma. (b) Cada fila corresponde a un bigrama de letras. Fuente: [Goldberg, 2017].

Una columna de  $W$  puede tomarse como una representación vectorial de 784 dimensiones de un idioma en términos de sus patrones característicos de bigramas de letras. Luego, podemos agrupar los 6 vectores de idioma según su similitud. Cada una de las 784 filas de  $W$  proporciona una representación vectorial de 6 dimensiones de ese bigrama en términos de los idiomas que promueve. Las representaciones son fundamentales para el aprendizaje profundo. Se podría argumentar que el principal poder del aprendizaje profundo es la capacidad de aprender buenas representaciones.



En el caso lineal, las representaciones son interpretables, ya que podemos asignar una interpretación significativa a cada dimensión en el vector de representación. Por ejemplo, cada dimensión puede corresponder a un idioma o a un determinado bigrama de letras.

Por otro lado, los modelos de aprendizaje profundo a menudo aprenden una cascada de representaciones de la entrada que se construyen una encima de la otra. Estas representaciones a menudo no son interpretables, es decir, no sabemos qué propiedades de la entrada capturan. Sin embargo, siguen siendo muy útiles para hacer predicciones.

## 5.5 Representación de Vectores One-Hot

El vector de entrada  $\vec{x}$  en nuestro ejemplo de clasificación de idioma contiene los recuentos normalizados de los bigramas en el documento  $D$ . Este vector se puede descomponer en un promedio de  $|D|$  vectores, cada uno correspondiente a una posición particular del documento  $i$ :

$$\vec{x} = \frac{1}{|D|} \sum_{i=1}^{|D|} \vec{x}^{D_{[i]}} \quad (5.10)$$

Aquí,  $D_{[i]}$  es el bigrama en la posición  $i$  del documento. Cada vector  $\vec{x}^{D_{[i]}} \in \mathcal{R}^{784}$  es un vector one-hot, lo que significa que todos los elementos son cero excepto la única entrada que corresponde al bigrama de letras  $D_{[i]}$ , que es 1. El vector resultante  $\vec{x}$  se conoce comúnmente como un promedio de bolsa de bigramas (más generalmente, una bolsa de palabras promediada, o simplemente una bolsa de palabras).

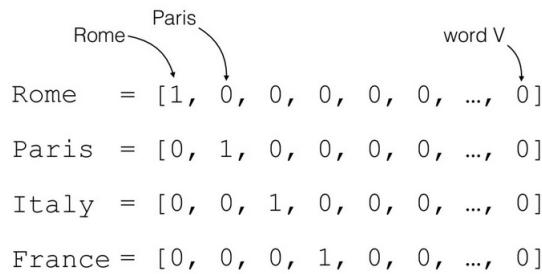
Las representaciones de bolsa de palabras contienen información sobre las identidades de todas las "palabras"(en este caso, bigramas) del documento, sin considerar su orden. Una representación one-hot se puede considerar como una bolsa de una sola "palabra".

Vectores one-hot de palabras.

Fuente: <https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7>.

En el caso binario, transformamos la predicción lineal en una estimación de probabilidad al pasarla por la función sigmoide, lo que resulta en un modelo log-lineal. En el caso de múltiples clases, el análogo es pasar el vector de puntajes a través de la función **softmax**:

$$\text{softmax}(\vec{x})_{[i]} = \frac{e^{\vec{x}_{[i]}}}{\sum_j e^{\vec{x}_{[j]}}} \quad (5.11)$$



Lo que resulta en:

$$\begin{aligned}
 \vec{\hat{y}} &= \text{softmax}(\vec{x} \cdot W + \vec{b}) \\
 \vec{\hat{y}}_{[i]} &= \frac{e^{(\vec{x} \cdot W + \vec{b})_{[i]}}}{\sum_j e^{(\vec{x} \cdot W + \vec{b})_{[j]}}}
 \end{aligned} \tag{5.12}$$

La transformación softmax fuerza a los valores en  $\hat{y}$  a ser positivos y sumar 1, lo que los hace interpretables como una distribución de probabilidad.

## 5.6 Entrenamiento

Cuando se entrena una función parametrizada (por ejemplo, un modelo lineal, una red neuronal), se define una función de pérdida  $L(\hat{y}, y)$ , que establece la pérdida al predecir  $\hat{y}$  cuando la salida verdadera es  $y$ .

$$L(f(\vec{x}; \Theta), y)$$

Utilizamos el símbolo  $\Theta$  para denotar todos los parámetros del modelo (por ejemplo,  $W, \vec{b}$ ).

El objetivo del entrenamiento es minimizar la pérdida en los diferentes ejemplos de entrenamiento. Formalmente, una función de pérdida  $L(\hat{y}, y)$  asigna una puntuación numérica (un escalar) a una salida predicha  $\hat{y}$  dada la salida esperada verdadera  $y$ . La función de pérdida debería alcanzar su valor mínimo para los casos en los que la predicción sea correcta.

También podemos definir una pérdida en todo el corpus con respecto a los parámetros  $\Theta$  como la pérdida promedio en todos los ejemplos de entrenamiento:

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^n L(f(\vec{x}_i; \Theta), y_i)$$

El objetivo del algoritmo de entrenamiento es establecer los valores de los parámetros  $\Theta$  de manera que el valor de  $\mathcal{L}$  se minimice.

$$\hat{\Theta} = \operatorname{argmin}_{\Theta} \mathcal{L}(\Theta) = \operatorname{argmin}_{\Theta} \frac{1}{n} \sum_{i=1}^n L(f(\vec{x}_i; \Theta), y_i)$$

### 5.6.1 Optimización basada en Gradiente

Las funciones se

entrenan utilizando métodos basados en gradientes. Estos métodos funcionan mediante el cálculo repetido de una estimación de la pérdida  $L$  sobre el conjunto de entrenamiento. El método de entrenamiento calcula los gradientes de los parámetros ( $\Theta$ ) con respecto a la estimación de pérdida y mueve los parámetros en la dirección opuesta al gradiente. Los diferentes métodos de optimización difieren en cómo se calcula la estimación de error y cómo se define el movimiento en la dirección opuesta al gradiente.

Si la función es convexa, el óptimo será global. De lo contrario, el proceso solo garantiza encontrar un óptimo local.

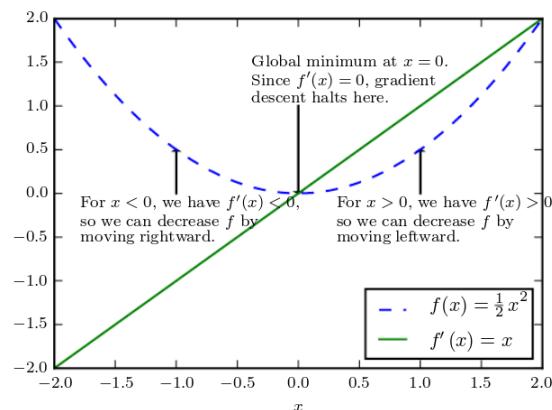
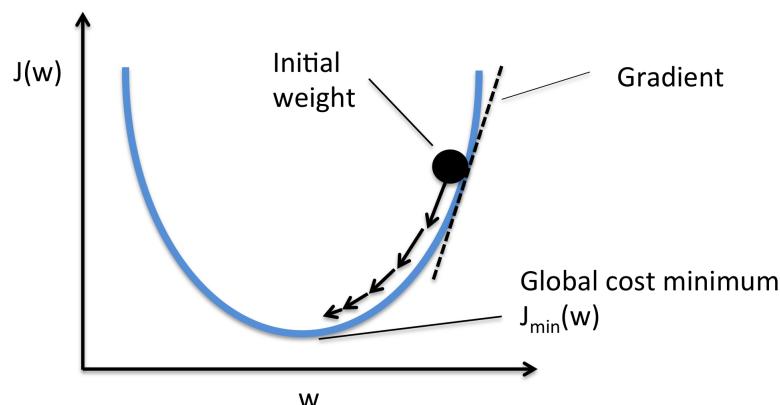


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

### 5.6.2 Descenso de Gradiente Estocástico en Línea

- Todos los parámetros se inicializan con valores aleatorios ( $\Theta$ ).

<sup>0</sup>Fuente: <https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png>

<sup>0</sup>[Goodfellow et al., 2016]

- Para cada ejemplo de entrenamiento  $(x, y)$ , calculamos la pérdida  $L$  con los valores actuales de  $\Theta$ .
- Luego actualizamos los parámetros con la siguiente regla hasta que se alcance la convergencia:
- $\Theta_i \leftarrow \Theta_i - \eta \frac{\partial L}{\partial \Theta_i}(\hat{y}, y)$  (para todos los parámetros  $\Theta_i$ )

---

**Algorithm 2.1** Online stochastic gradient descent training.

---

*Input:*

- Function  $f(x; \Theta)$  parameterized with parameters  $\Theta$ .
- Training set of inputs  $x_1, \dots, x_n$  and desired outputs  $y_1, \dots, y_n$ .
- Loss function  $L$ .

---

```

1: while stopping criteria not met do
2:   Sample a training example  $x_i, y_i$ 
3:   Compute the loss  $L(f(x_i; \Theta), y_i)$ 
4:    $\hat{g} \leftarrow$  gradients of  $L(f(x_i; \Theta), y_i)$  w.r.t  $\Theta$ 
5:    $\Theta \leftarrow \Theta - \eta_t \hat{g}$ 
6: return  $\Theta$ 

```

---

La tasa de aprendizaje puede ser fija durante todo el proceso de entrenamiento o puede decrecer como función del paso de tiempo  $t$ . El error calculado en la línea 3 se basa en un solo ejemplo de entrenamiento y, por lo tanto, es solo una estimación aproximada de la pérdida en todo el corpus  $L$  que queremos minimizar. El ruido en el cálculo de la pérdida puede resultar en gradientes inexactos (los ejemplos individuales pueden proporcionar información ruidosa).

### 5.6.3 Descenso de Gradiente Estocástico en Mini-batch

- Una forma común de reducir este ruido es estimar el error y los gradientes en función de una muestra de  $m$  ejemplos.
- Esto da lugar al algoritmo de SGD en mini-batch.

---

**Algorithm 2.2** Minibatch stochastic gradient descent training.

---

*Input:*

- Function  $f(x; \Theta)$  parameterized with parameters  $\Theta$ .
- Training set of inputs  $x_1, \dots, x_n$  and desired outputs  $y_1, \dots, y_n$ .
- Loss function  $L$ .

---

```

1: while stopping criteria not met do
2:   Sample a minibatch of  $m$  examples  $\{(x_1, y_1), \dots, (x_m, y_m)\}$ 
3:    $\hat{g} \leftarrow 0$ 
4:   for  $i = 1$  to  $m$  do
5:     Compute the loss  $L(f(x_i; \Theta), y_i)$ 
6:      $\hat{g} \leftarrow \hat{g} +$  gradients of  $\frac{1}{m}L(f(x_i; \Theta), y_i)$  w.r.t  $\Theta$ 
7:    $\Theta \leftarrow \Theta - \eta_t \hat{g}$ 
8: return  $\Theta$ 

```

---

Valores más altos de  $m$  proporcionan mejores estimaciones de los gradientes en todo el corpus, mientras que valores más pequeños permiten más actualizaciones y, a su vez, una convergencia más rápida.

Para tamaños moderados de

$m$ , algunas arquitecturas de cómputo (por ejemplo, GPUs) permiten una implementación paralela eficiente del cálculo en las líneas 3-6.

---

<sup>0</sup>Fuente:[Goldberg, 2017]

<sup>0</sup>Fuente:[Goldberg, 2017]

### 5.6.4 Funciones de Pérdida

Las funciones de pérdida son utilizadas en algoritmos de aprendizaje automático para medir la discrepancia entre las predicciones del modelo y los valores reales de los datos de entrenamiento. Algunas funciones de pérdida comunes son:

- Pérdida Hinge (o pérdida SVM): utilizada en problemas de clasificación binaria, donde la salida del clasificador es un escalar  $\tilde{y}$  y la salida deseada  $y$  está en el conjunto  $\{+1, -1\}$ . La regla de clasificación es  $\hat{y} = \text{signo}(\tilde{y})$ , y se considera una clasificación correcta si  $y \cdot \tilde{y} > 0$ . La función de pérdida se define como:

$$L_{\text{hinge(binaria)}}(\tilde{y}, y) = \max(0, 1 - y \cdot \tilde{y})$$

- Entropía cruzada binaria (o pérdida logística): utilizada en clasificación binaria con salidas de probabilidad condicional. La salida del clasificador  $\tilde{y}$  se transforma utilizando la función sigmoide para que esté en el rango  $[0, 1]$ , y se interpreta como la probabilidad condicional  $P(y = 1|x)$ . La función de pérdida se define como:

$$L_{\text{logística}}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

- La pérdida logística tiene una interpretación probabilística. Se asume que  $P(y = 1|\vec{x}; \Theta) = \sigma(f(\vec{x})) = \frac{1}{1 + e^{-\vec{x} \cdot \vec{w} + b}}$  y  $P(y = 0|\vec{x}; \Theta) = 1 - \sigma(f(\vec{x}))$ . Esto se puede escribir de manera más compacta como:

$$P(y|\vec{x}; \Theta) = \sigma(f(\vec{x}))^y \times (1 - \sigma(f(\vec{x})))^{1-y}$$

- La expresión anterior es la función de masa de probabilidad de la distribución de Bernoulli.
- Si reemplazamos  $\sigma(f(\vec{x}))$  por  $\hat{y}$ , obtenemos:

$$P(y|\vec{x}; \Theta) = \hat{y}^y \times (1 - \hat{y})^{1-y}$$

- Si aplicamos la estimación de máxima verosimilitud a esta expresión y tomamos el logaritmo, obtenemos:

$$y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

- ¡Maximizar esta expresión es equivalente a minimizar la pérdida logística!
- ¡Muchas funciones de pérdida corresponden al logaritmo negativo de la verosimilitud de modelos probabilísticos!
- Pérdida de entropía cruzada categórica: se utiliza cuando se desea una interpretación probabilística de las puntuaciones de múltiples clases. Mide la discrepancia entre la distribución de etiquetas reales  $\vec{y}$  y la distribución de etiquetas predichas  $\vec{\hat{y}}$ . La función de pérdida se define como:

$$L_{\text{entropía cruzada}}(\vec{\hat{y}}, \vec{y}) = - \sum_i \vec{y}_{[i]} \log(\vec{\hat{y}}_{[i]})$$

- Cuando se utiliza la pérdida de entropía cruzada, se asume que la salida del clasificador se transforma utilizando la función softmax.
- La función softmax comprime la salida de  $k$  dimensiones a valores en el rango  $(0,1)$  de modo que todas las entradas sumen 1. Por lo tanto,  $\vec{\hat{y}}_{[i]} = P(y = i|x)$  representa la distribución de pertenencia a la clase condicional.

- Para problemas de clasificación dura en los que cada ejemplo de entrenamiento tiene una única asignación de clase correcta,  $\vec{y}$  es un vector one-hot que representa la clase verdadera. En tales casos, la entropía cruzada se simplifica a:

$$L_{\text{entropía cruzada (clasificación dura)}}(\vec{\hat{y}}, \vec{y}) = -\log(\vec{\hat{y}}_{[t]})$$

donde  $t$  es la asignación de clase correcta.

## 5.7 Regularización

Nuestro problema de optimización puede tener múltiples soluciones y, especialmente en dimensiones más altas, puede sufrir de sobreajuste. Consideremos el siguiente escenario en nuestro problema de identificación de idioma: uno de los documentos en el conjunto de entrenamiento ( $\vec{x}_O$ ) es un valor atípico. En realidad, el documento está en alemán, pero está etiquetado como francés.

Para reducir la pérdida, el modelo puede identificar características (bigramas de letras) en  $\vec{x}_O$  que ocurren en pocos otros documentos. El modelo asignará a estas características pesos muy altos hacia la clase francesa (incorrecta). Esto es una solución incorrecta para el problema de aprendizaje, ya que el modelo está aprendiendo algo incorrecto. Los documentos alemanes que comparten muchas palabras con  $\vec{x}_O$  podrían clasificarse erróneamente como franceses. Nos gustaría controlar estos casos y alejar al modelo de soluciones equivocadas.

La idea de la regularización es agregar un término de regularización  $R$  al objetivo de optimización. El objetivo de este término es controlar la complejidad (pesos grandes) del valor de los parámetros ( $\Theta$ ) y evitar el sobreajuste:

$$\begin{aligned} \hat{\Theta} &= \operatorname{argmin}_{\Theta} \mathcal{L}(\Theta) + \lambda R(\Theta) \\ &= \operatorname{argmin}_{\Theta} \frac{1}{n} \sum_{i=1}^n L(f(\vec{x}_i; \Theta), y_i) + \lambda R(\Theta) \end{aligned} \quad (5.13)$$

El término de regularización considera los valores de los parámetros y evalúa su complejidad. El valor del hiperparámetro  $\lambda$  debe establecerse manualmente en función del rendimiento de clasificación en un conjunto de desarrollo.

En la práctica, los regularizadores  $R$  consideran la complejidad como pesos grandes. Trabajan para mantener los valores de los parámetros ( $\Theta$ ) bajos. Las opciones comunes para  $R$  son la norma  $L_2$ , la norma  $L_1$  y la elastic-net.

### 5.7.1 Regularización $L_2$

En la regularización  $L_2$ ,  $R$  toma la forma de la norma al cuadrado  $L_2$  de los parámetros. El objetivo es mantener baja la suma de los cuadrados de los valores de los parámetros:

$$R_{L_2}(W) = \|W\|_2^2 = \sum_{i,j} (W_{[i,l]})^2$$

El regularizador  $L_2$  también se llama una priori gaussiana o decaimiento de peso. Los modelos regularizados con  $L_2$  se ven severamente penalizados por pesos de parámetros altos. Una vez que el valor está lo suficientemente cerca de cero, su efecto se vuelve insignificante. El modelo preferirá disminuir el valor de un parámetro con peso alto en 1 en lugar de disminuir el valor de diez parámetros que ya tienen pesos relativamente bajos en 0.1 cada uno.

### 5.7.2 Regularización $L_1$

En la regularización  $L_1$ ,  $R$  toma la forma de la norma  $L_1$  de los parámetros. El objetivo es mantener baja la suma de los valores absolutos de los parámetros:

$$R_{L_1}(W) = ||W||_1 = \sum_{i,j} |W_{i,j}|$$

A diferencia de  $L_2$ , el regularizador  $L_1$  se penaliza de manera uniforme para valores bajos y altos. Tiene incentivos para disminuir todos los valores de parámetros no nulos hacia cero. Por lo tanto, fomenta soluciones dispersas, es decir, modelos con muchos parámetros con valor cero. El regularizador  $L_1$  también se llama una priori dispersa o lasso [Tibshirani, 1996].

### 5.7.3 Elastic-Net

El método de regularización elastic-net [Zou and Hastie, 2005] combina tanto la regularización  $L_1$  como la regularización  $L_2$  de la siguiente manera:

$$R_{\text{elastic-net}}(W) = \lambda_1 R_{L_1}(W) + \lambda_2 R_{L_2}(W)$$

## 5.8 Más allá del SGD

Aunque el algoritmo SGD puede producir buenos resultados, también existen algoritmos más avanzados disponibles. Los algoritmos SGD+Momentum [Polyak, 1964] y Nesterov Momentum [Nesterov, 2018, Sutskever et al., 2013] son variantes de SGD en las que los gradientes anteriores se acumulan y afectan la actualización actual. Los algoritmos de tasa de aprendizaje adaptativa, como AdaGrad [Duchi et al., 2011], AdaDelta [Zeiler, 2012], RMSProp [Tieleman and Hinton, 2012] y Adam [Kingma and Ba, 2014], están diseñados para seleccionar la tasa de aprendizaje para cada minibatch. A veces, esto se hace de manera individual por coordenada, lo que puede aliviar la necesidad de ajustar la programación de la tasa de aprendizaje. Para obtener más detalles sobre estos algoritmos, consulte los documentos originales o [Goodfellow et al., 2016] (Secciones 8.3, 8.4).

## 5.9 Una limitación de los modelos lineales: el problema XOR

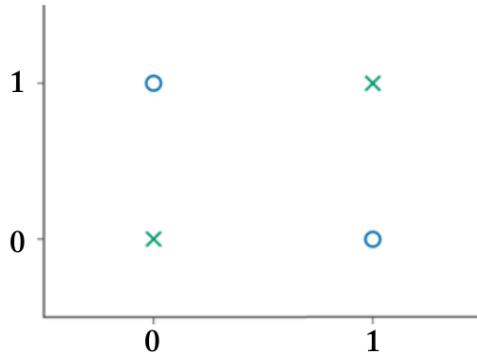
La clase de hipótesis de modelos lineales (y log-lineales) está severamente restringida. Por ejemplo, no puede representar la función XOR, definida como:

$$\begin{aligned} \text{xor}(0,0) &= 0 \\ \text{xor}(1,0) &= 1 \\ \text{xor}(0,1) &= 1 \\ \text{xor}(1,1) &= 0 \end{aligned} \tag{5.14}$$

No existe una parametrización  $\vec{w} \in \mathbb{R}^2, b \in \mathbb{R}$  tal que:

$$\begin{aligned}
 (0,0) \cdot \vec{w} + b &< 0 \\
 (0,1) \cdot \vec{w} + b &\geq 0 \\
 (1,0) \cdot \vec{w} + b &\geq 0 \\
 (1,1) \cdot \vec{w} + b &< 0
 \end{aligned} \tag{5.15}$$

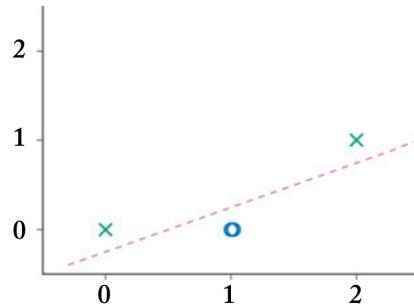
Para ver por qué, consideremos el siguiente gráfico de la función XOR, donde los Os azules denotan la clase positiva y las X verdes la clase negativa.



Es evidente que ninguna línea recta puede separar las dos clases.

### 5.9.1 Transformaciones no lineales de las entradas

Si transformamos los puntos alimentándolos a través de la función no lineal  $\phi(x_1, x_2) = [x_1 \times x_2, x_1 + x_2]$ , el problema XOR se vuelve linealmente separable.



La función  $\phi$  mapea los datos a una representación adecuada para la clasificación lineal. Ahora podemos entrenar fácilmente un clasificador lineal para resolver el problema XOR.

$$\hat{y} = f(\vec{x}) = \phi(\vec{x}) \cdot \vec{w} + b \tag{5.16}$$

El problema es que necesitamos definir manualmente la función  $\phi$ . Este proceso depende del conjunto de datos particular y requiere mucha intuición humana. La sol

ción es definir una función de mapeo no lineal entrenable y entrenarla junto con el clasificador lineal. Encontrar la representación adecuada se convierte en responsabilidad del algoritmo de entrenamiento.

Las funciones de mapeo pueden tomar la forma de un modelo lineal parametrizado, seguido de una función de activación no lineal  $g$  que se aplica a cada una de las dimensiones de salida:

$$\begin{aligned}\hat{y} &= f(\vec{x}) = \phi(\vec{x}) \cdot \vec{w} + b \\ \phi(\vec{x}) &= g(\vec{x}W' + \vec{b}')\end{aligned}\tag{5.17}$$

Si tomamos  $g(x) = \max(0, x)$  y  $W' = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ ,  $\vec{b}' = (-1 \quad 0)$ , obtenemos un mapeo equivalente a  $[x_1 \times x_2, x_1 + x_2]$  para nuestros puntos de interés  $(0,0)$ ,  $(0,1)$ ,  $(1,0)$  y  $(1,1)$ . ¡Esto resuelve con éxito el problema XOR!

Aprender tanto la función de representación como el clasificador lineal en la parte superior de ella al mismo tiempo es la idea principal detrás del aprendizaje profundo y las redes neuronales. De hecho, la ecuación anterior describe una arquitectura de red neuronal muy común llamada perceptrón multicapa (MLP, por sus siglas en inglés).

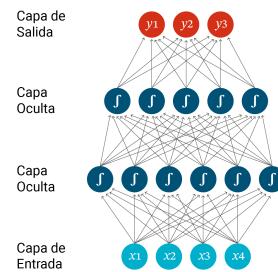




## 6. Redes Neuronales

Las redes neuronales son una familia muy popular de modelos de aprendizaje automático formados por unidades llamadas **neuronas**. Una neurona es una unidad computacional que tiene entradas y salidas escalares. Cada entrada tiene asociado un peso  $w$ . La neurona multiplica cada entrada por su peso y luego los suma (también son posibles otras funciones como **max**). Aplica una función de activación  $g$  (generalmente no lineal) al resultado y lo pasa a su salida. Se pueden apilar múltiples capas. La función de activación no lineal  $g$  juega un papel crucial en la capacidad de la red para representar funciones complejas. Sin la no linealidad en  $g$ , la red neuronal solo puede representar transformaciones lineales de la entrada.

Ejemplo: Red feedforward con dos capas



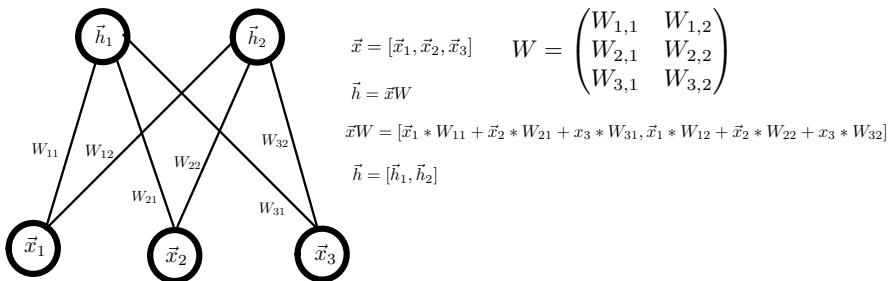
### 6.1 Redes neuronales feedforward

- La red feedforward de la imagen es una concatenación de modelos lineales separados por funciones no lineales.
- Los valores de cada fila de neuronas en la red se pueden pensar como un vector.

<sup>0</sup>Fuente:[Goldberg, 2017]

- La capa de entrada es un vector de 4 dimensiones ( $\vec{x}$ ), y la capa superior es un vector de 6 dimensiones ( $\vec{h}^1$ ).
- La capa completamente conectada se puede pensar como una transformación lineal de 4 dimensiones a 6 dimensiones.
- Una capa completamente conectada implementa una multiplicación de vector-matriz,  $\vec{h} = \vec{x}W$ .
- El peso de la conexión desde la neurona  $i$ -ésima en la fila de entrada hasta la neurona  $j$ -ésima en la fila de salida es  $W_{[i,j]}$ .
- Los valores de  $\vec{h}$  se transforman mediante una función no lineal  $g$  que se aplica a cada valor antes de pasar al siguiente nivel.

### Capas completamente conectadas como multiplicaciones de vectores y matrices



#### 6.1.1 Redes neuronales como funciones matemáticas

- El Perceptrón Multicapa (MLP, por sus siglas en inglés) de la figura se llama MLP2 porque tiene dos capas ocultas.
- Un modelo más simple sería MLP1, un perceptrón multicapa de una capa oculta:

$$\vec{y} = NN_{MLP1}(\vec{x}) = g(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2 \quad (6.1)$$

$$\vec{x} \in \mathcal{R}^{d_{in}}, W^1 \in \mathcal{R}^{d_{in} \times d_1}, \vec{b}^1 \in \mathcal{R}^{d_1}, W^2 \in \mathcal{R}^{d_1 \times d_{out}}, \vec{b}^2 \in \mathcal{R}^{d_{out}}, \vec{y} \in \mathcal{R}^{d_{out}}$$

- Aquí,  $W^1$  y  $\vec{b}^1$  son una matriz y un término de sesgo para la primera transformación lineal de la entrada.
- La función  $g$  es una función no lineal que se aplica elemento a elemento (también se llama no linealidad o función de activación).
- $W^2$  y  $\vec{b}^2$  son la matriz y el término de sesgo para una segunda transformación lineal.
- Al describir una red neuronal, se deben especificar las dimensiones de las capas ( $d_1$ ), la entrada ( $d_{in}$ ) y la salida ( $d_{out}$ ).
- MLP2 se puede escribir como la siguiente función matemática:

<sup>0</sup>Se asume que los vectores son vectores fila y los índices en superíndices corresponden a las capas de la red.

$$\begin{aligned}
 NN_{MLP2}(\vec{x}) &= \vec{y} \\
 \vec{h}^1 &= \vec{x}W^1 + \vec{b}^1 \\
 \vec{h}^2 &= g^1(\vec{h}^1)W^2 + \vec{b}^2 \\
 \vec{y} &= g^2(\vec{h}^2)W^3 \\
 \vec{y} &= (g^2(g^1(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2))W^3.
 \end{aligned} \tag{6.2}$$

- Las matrices y los términos de sesgo que definen las transformaciones lineales son los parámetros de la red.
- Al igual que en los modelos lineales, es común referirse a la colección de todos los parámetros como  $\Theta$ .

## 6.2 Capacidad de representación

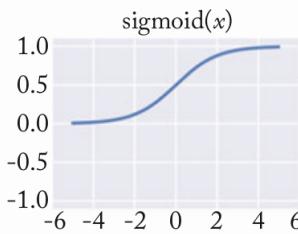
- [Hornik et al., 1989] y [Cybenko, 1989] mostraron que un perceptrón multicapa de una capa oculta (MLP1) es un aproximador universal.
- MLP1 puede aproximar todas las funciones continuas en un subconjunto cerrado y acotado de  $\mathcal{R}^n$ .
- Esto puede sugerir que no hay razón para ir más allá de MLP1 en arquitecturas más complejas.
- El resultado no dice qué tan fácil o difícil es establecer los parámetros basándose en los datos de entrenamiento y un algoritmo de aprendizaje específico.
- Tampoco garantiza que un algoritmo de entrenamiento encontrará la función correcta que genera nuestros datos de entrenamiento.
- Finalmente, no establece qué tan grande debería ser la capa oculta.
- En la práctica, entrenamos redes neuronales con cantidades relativamente pequeñas de datos utilizando métodos de búsqueda local.
- También utilizamos capas ocultas de tamaños relativamente modestos (hasta varios miles).
- El teorema de aproximación universal no ofrece ninguna garantía bajo estas condiciones.
- Sin embargo, definitivamente hay beneficios en probar arquitecturas más complejas que MLP1.
- En muchos casos, sin embargo, MLP1 brinda resultados sólidos.

## 6.3 Funciones de activación

- La no linealidad  $g$  puede tomar muchas formas.
- Actualmente no existe una buena teoría sobre qué no linealidad aplicar en qué condiciones.
- Elegir la no linealidad correcta para una tarea determinada es en su mayor parte una cuestión empírica.

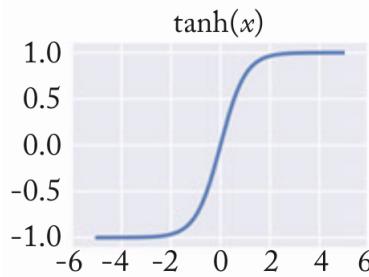
### Sigmoide

- La función de activación sigmoide  $\sigma(x) = \frac{1}{1+e^{-x}}$  es una función en forma de S, que transforma cada valor  $x$  en el rango  $[0, 1]$ .
- El sigmoide fue la no linealidad canónica para las redes neuronales desde su inicio.
- Actualmente se considera obsoleta para su uso en capas internas de redes neuronales, ya que las opciones que se enumeran a continuación funcionan mucho mejor empíricamente.



### Tangente hiperbólica ( $\tanh$ )

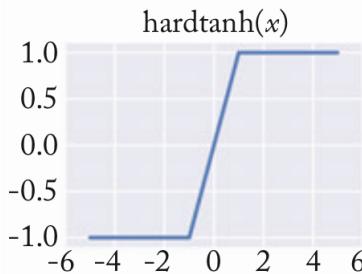
- La función de activación tangente hiperbólica  $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$  es una función en forma de S que transforma los valores x en el rango  $[-1, 1]$ .



### Hard tanh

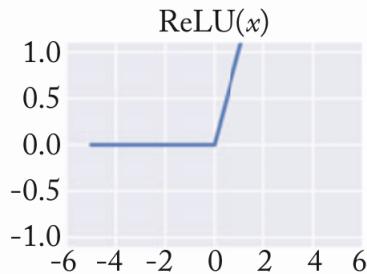
- La función de activación hard-tanh es una aproximación de la función tangente hiperbólica que es más rápida de calcular y encontrar sus derivadas:

$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{en otros casos.} \end{cases}$$



### ReLU

- La función de activación rectificador [Glorot et al., 2011], también conocida como unidad lineal rectificada, es una función de activación muy simple.
- Es fácil de trabajar y se ha demostrado muchas veces que produce excelentes resultados.
- La función ReLU se define como  $\text{ReLU}(x) = \max(0, x)$ .



### Leaky ReLU

- La función Leaky ReLU es similar a ReLU, pero permite una pequeña pendiente para valores negativos.
- Se define como  $\text{LeakyReLU}(x) = \max(\alpha x, x)$ , donde  $\alpha$  es un hiperparámetro pequeño (por lo general, en el rango de 0,01 a 0,2).

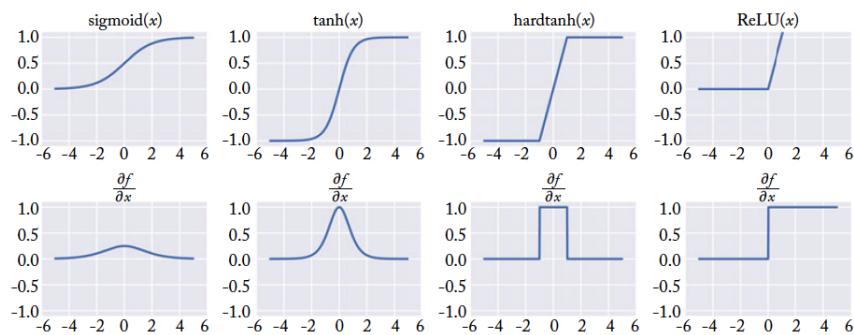
### ELU

- La función de activación ELU (Exponential Linear Unit) [?] es una versión mejorada de ReLU que también tiene una pendiente para valores negativos.
- Se define como  $\text{ELU}(x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & \text{en otros casos.} \end{cases}$
- Aquí,  $\alpha$  es un hiperparámetro que controla la pendiente negativa.

Estas son solo algunas de las muchas opciones disponibles para las funciones de activación en redes neuronales. La elección de la función de activación puede depender del problema específico y puede requerir pruebas empíricas para determinar cuál funciona mejor.

#### 6.3.1 Problemas Prácticos

En términos generales, tanto las unidades ReLU como las unidades tangente hiperbólica ( $\tanh$ ) funcionan bien y superan significativamente a la función sigmoide. Sin embargo, puede ser beneficioso experimentar con ambas activaciones, ya que cada una puede funcionar mejor en diferentes configuraciones. La Figura 1 muestra las formas de las diferentes funciones de activación, junto con las formas de sus derivadas.



<sup>0</sup>Fuente:[Goldberg, 2017]

## 6.4 Capas de Embedding

En el procesamiento del lenguaje natural (PLN), la entrada a la red neuronal contiene características categóricas simbólicas (por ejemplo, palabras de un vocabulario cerrado, n-gramas de caracteres, etiquetas POS). En los modelos lineales, generalmente representamos la entrada con vectores dispersos, como la suma, el promedio o la concatenación de vectores codificados one-hot (la suma o el promedio pueden producir una representación de bolsa de palabras). En las redes neuronales, es común asociar cada valor de característica posible (es decir, cada palabra en el vocabulario, cada categoría de etiqueta POS) con un vector denso de  $d$  dimensiones.

Estos vectores luego se consideran parámetros del modelo y se entrenan conjuntamente con los demás parámetros. El mapeo desde valores de características simbólicas, como "número de palabra 1249", a vectores de  $d$  dimensiones se realiza mediante una capa de embedding (también llamada capa de búsqueda). Los parámetros en una capa de embedding de palabras son simplemente una matriz  $E \in \mathbb{R}^{|vocab| \times d}$  donde cada fila corresponde a una palabra diferente en el vocabulario. La operación de búsqueda es simplemente una indexación:  $v_{1249} = E_{[1249,:]}$ . Si la característica simbólica se codifica como un vector one-hot  $\vec{x}$ , la operación de búsqueda se puede implementar como una multiplicación de matriz-vector  $\vec{x}E$ . Los vectores de embedding se combinan antes de pasar a la siguiente capa. Las operaciones comunes de combinación son concatenación, suma y promedio. Una matriz de embeddings de palabras  $E$  se puede inicializar con vectores de palabras preentrenados a partir de documentos no etiquetados utilizando métodos específicos basados en la hipótesis distribucional, como los implementados en Word2Vec (que se discutirán más adelante en el curso).

One-hot-encoded word vector		Embedding Matrix
$\vec{x} = [0, 0, \dots, 1, \dots, 0]$	$^{1 \times  V }$	$^{ \mathcal{V}  \times d}$
abduct	dog	zumba
		$E = \begin{bmatrix} -1.8 & 2.3 & \dots & 3.1 \\ \vdots & \vdots & \ddots & \vdots \\ 3.3 & -2.1 & \dots & 4.6 \\ \vdots & \vdots & \ddots & \vdots \\ 4.2 & 1.9 & \dots & -3.3 \end{bmatrix}$
		← abduct ← dog ← zumba

$$\vec{x}E = [3.3, -2.1, \dots, 4.6]$$

### 6.4.1 Vectores Densos vs Representaciones One-hot

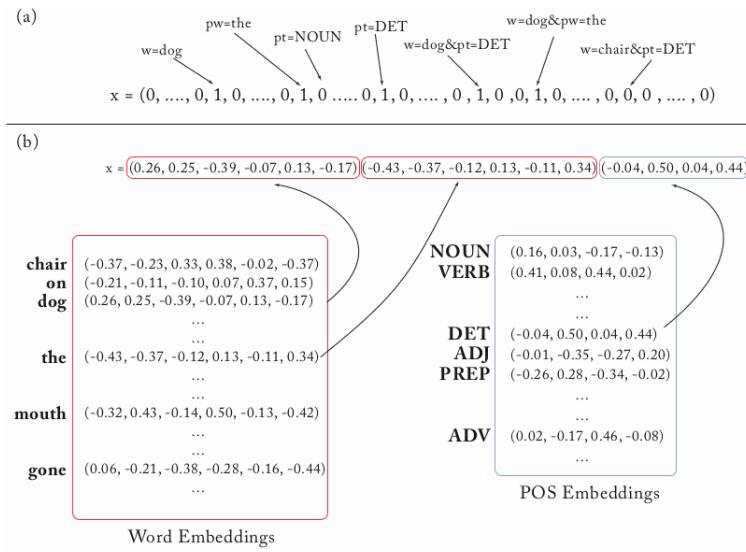
¿Cuáles son los beneficios de representar nuestras características como vectores en lugar de como identificadores únicos? ¿Deberíamos siempre representar las características como vectores densos? Consideremos los dos tipos de representaciones.

1. **One Hot:** cada característica es su propia dimensión.
  - La dimensionalidad del vector one-hot es igual al número de características distintas.
  - Las características son completamente independientes entre sí. La característica "la palabra es 'perro'" es tan diferente de "la palabra es 'pensando'" como lo es de "la palabra es 'gato'".
2. **Dense:** cada característica es un vector de  $d$  dimensiones.
  - La dimensionalidad del vector es  $d$ .
  - El entrenamiento del modelo hará que características similares tengan vectores similares: la información se comparte entre características similares.

Matriz de embeddings

$$E = \begin{bmatrix} |V| \times d \end{bmatrix}$$

$$\begin{bmatrix} -6.6 & 4.3 & \dots & -2.4 \\ \vdots & \vdots & \ddots & \vdots \\ 3.6 & -1.9 & \dots & 4.1 \\ \vdots & \vdots & \ddots & \vdots \\ -1.8 & 2.3 & \dots & 3.1 \\ \vdots & \vdots & \ddots & \vdots \\ 2.5 & 3.4 & \dots & -2.7 \\ \vdots & \vdots & \ddots & \vdots \\ 5.0 & -3.4 & \dots & 1.4 \\ \vdots & \vdots & \ddots & \vdots \\ 4.2 & 1.9 & \dots & 4.2 \\ \vdots & \vdots & \ddots & \vdots \\ 3.3 & -2.1 & \dots & 4.6 \\ \vdots & \vdots & \ddots & \vdots \\ -2.6 & 1.5 & \dots & 1.6 \end{bmatrix} \leftarrow \begin{array}{l} \text{abrazar} \\ \text{canino} \\ \text{el} \\ \text{fuerte} \\ \text{grueso} \\ \text{ladra} \\ \text{perro} \\ \text{zancudo} \end{array}$$

**Ejemplo: Vectores Densos vs Representaciones One-hot**

La figura anterior muestra dos codificaciones de la información: la palabra actual es "perro"; la palabra anterior es ".el"; la etiqueta POS anterior es "DET".

(a) Vector de características dispersas:

- Cada dimensión representa una característica.
- Las combinaciones de características tienen sus propias dimensiones.
- Los valores de las características son binarios.
- La dimensionalidad es muy alta.

(b) Vector de características densas basado en embeddings.

- Cada característica principal se representa como un vector.
- Cada característica se corresponde con varias entradas del vector de entrada.
- No hay una codificación explícita de combinaciones de características.

- La dimensionalidad es baja.
- Los mapeos de características a vectores provienen de una tabla de embeddings.

Un beneficio de usar vectores densos y de baja dimensionalidad es computacional: la mayoría de las bibliotecas de redes neuronales no funcionan bien con vectores dispersos de alta dimensionalidad. Sin embargo, este es solo un obstáculo técnico que se puede resolver con cierto esfuerzo de ingeniería.

El principal beneficio de las representaciones densas radica en el poder de generalización. Si creemos que algunas características pueden proporcionar pistas similares, vale la pena proporcionar una representación que pueda capturar estas similitudes. Supongamos que hemos observado la palabra "perro" muchas veces durante el entrenamiento, pero solo hemos observado la palabra "gato" pocas veces. Si cada una de las palabras se asocia con su propia dimensión (one-hot), las ocurrencias de "perro" no nos dirán nada sobre las ocurrencias de "gato". Sin embargo, en la representación de vectores densos, el vector aprendido para "perro" puede ser similar al vector aprendido para "gato". Esto permitirá que el modelo comparta fuerza estadística entre los dos eventos. Este argumento asume que hemos visto suficientes ocurrencias de la palabra "gato" como para que su vector sea similar al de "perro". Los vectores de palabras preentrenados (por ejemplo, Word2Vec, GloVe), que se discutirán más adelante en el curso, se pueden utilizar para obtener vectores densos a partir de texto no anotado.

## 6.5 Entrenamiento de Redes Neuronales

Las redes neuronales se entranan de la misma manera que los modelos lineales. La salida de la red se utiliza para calcular una función de pérdida  $L(\hat{y}, y)$  que se minimiza en todos los ejemplos de entrenamiento utilizando descenso de gradiente. El algoritmo de retropropagación es una técnica eficiente para evaluar el gradiente de una función de pérdida  $L$  en una red neuronal de alimentación directa con respecto a todos sus parámetros (Bishop, 2006). Los parámetros de la red incluyen  $W^1, \vec{b}^1, \dots, W^m, \vec{b}^m$  para una red de  $m$  capas. Cabe destacar que los superíndices se utilizan para denotar los índices de las capas (no exponentiación). Para simplificar, asumiremos que  $L$  se calcula sobre un solo ejemplo. El desafío radica en que en las redes neuronales el número de parámetros puede ser enorme y necesitamos una forma eficiente de calcular los gradientes. La idea es aplicar la regla de la cadena de derivadas de manera inteligente.

## 6.6 Recordatorio de la Regla de la Cadena en Derivadas

La regla de la cadena simple establece que si  $z = f(y)$  y  $y = g(x)$ , entonces

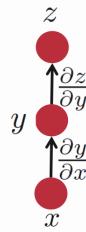
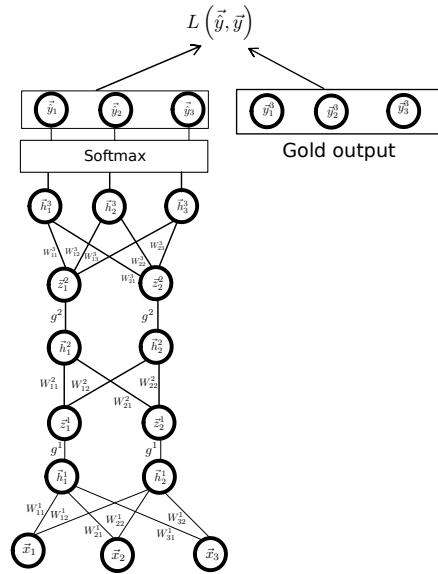
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$$

Por ejemplo, si  $z = e^y$  y  $y = 2x$ , entonces

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x} = e^y \times 2 = 2e^{2x}$$

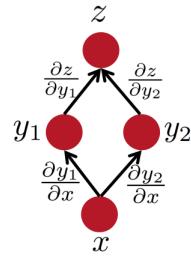
La regla de la cadena múltiple establece que si  $z = f(y_1, y_2)$ ,  $y_1 = g_1(x)$  y  $y_2 = g_2(x)$ , entonces

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \times \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \times \frac{\partial y_2}{\partial x}$$



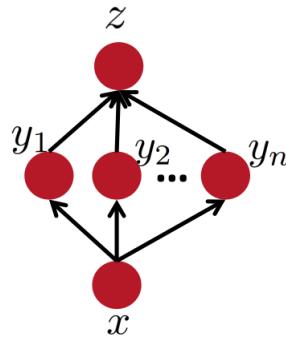
Por ejemplo, si  $z = e^{y_1 \times y_2}$ ,  $y_1 = 2x$  y  $y_2 = x^2$ , entonces

$$\frac{\partial z}{\partial x} = (e^{y_1 \times y_2} \times y_2) \times 2 + (e^{y_1 \times y_2} \times y_1) \times 2x = e^{2x^3} \times 6x^2$$



La versión general de la regla de la cadena múltiple es:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \times \frac{\partial y_i}{\partial x}$$



## 6.7 Retroproyección

En una red de alimentación directa general, cada unidad calcula una suma ponderada de sus entradas de la siguiente forma:

$$\vec{h}_{[j]}^l = \left( \sum_i W_{[i,j]}^l \times \vec{z}_{[i]}^{(l-1)} \right) + \vec{b}_{[j]}^l \quad (6.3)$$

La variable  $\vec{z}_{[i]}^{(l-1)}$  es una entrada que envía una conexión a la unidad  $\vec{h}_{[j]}^l$ ,  $W_{[i,j]}^l$  es el peso asociado con esa conexión, y  $l$  es el índice de la capa.

Los vectores de sesgo  $\vec{b}_{[j]}$  pueden excluirse de (eq. 6.3) e incluirse en la matriz de pesos  $W_{[i,j]}^l$  al introducir una unidad adicional, o entrada, con una activación fija de  $+1$ .

Las entradas en la capa  $l$ ,  $\vec{z}_{[i]}^{(l-1)}$ , son el resultado de aplicar la función de activación  $g$  a las unidades de la capa anterior:

$$\vec{z}_{[j]}^l = g(\vec{h}_{[j]}^l) \quad (6.4)$$

Para la capa de entrada ( $l = 0$ ),  $\vec{z}$  corresponde al vector de entrada  $\vec{z} = \vec{x}$ :

$$\vec{z}_{[j]}^0 = \vec{x}_{[j]} \quad (6.5)$$

Para cada instancia en el conjunto de entrenamiento, proporcionamos el vector de entrada correspondiente  $\vec{x}$  a la red. Luego calculamos las activaciones de todas las unidades ocultas y de salida en la red mediante la aplicación sucesiva de (eq. 6.3) y (eq. 6.4).

Este proceso a menudo se denomina propagación hacia adelante porque se puede considerar como un flujo de información hacia adelante a través de la red.

Ahora consideraremos la evaluación de la derivada de  $L$  con respecto a un peso  $W_{[i,j]}^l$ .

Suponiendo que la pérdida  $L$  se calcula sobre un solo ejemplo, podemos observar que  $L$  depende del peso  $W_{[i,j]}^l$  únicamente a través de la suma de las entradas  $\vec{h}_{[j]}^l$ .

Por lo tanto, podemos aplicar la

regla de la cadena para derivadas parciales para obtener:

$$\frac{\partial L}{\partial W_{[i,j]}^l} = \frac{\partial L}{\partial \vec{h}_{[j]}^l} \times \frac{\partial \vec{h}_{[j]}^l}{\partial W_{[i,j]}^l} \quad (6.6)$$

Ahora introducimos una notación útil:

$$\vec{\delta}_{[j]}^l \equiv \frac{\partial L}{\partial \vec{h}_{[j]}^l} \quad (6.7)$$

Usando (6.3), podemos escribir

$$\frac{\partial \vec{h}_{[j]}^l}{\partial W_{[i,j]}^l} = \vec{z}_{[i]}^{(l-1)} \quad (6.8)$$

Sustituyendo (6.7) y (6.8) en (6.6), obtenemos

$$\frac{\partial L}{\partial W_{[i,j]}^l} = \vec{\delta}_{[j]}^l \times \vec{z}_{[i]}^{(l-1)} \quad (6.9)$$

La ecuación (6.9) nos dice que la derivada requerida se obtiene simplemente multiplicando el valor de  $\vec{\delta}_{[j]}^l$  por el valor de  $\vec{z}_{[i]}^{(l-1)}$ .

Por lo tanto, para evaluar las derivadas, solo necesitamos calcular el valor de  $\vec{\delta}_{[j]}^l$  para cada unidad oculta y de salida en la red, y luego aplicar (6.9) para actualizar los pesos de la red. Este proceso se conoce como retropropagación, ya que el cálculo del gradiente se propaga hacia atrás a través de la red.

Calcular  $\vec{\delta}_{[j]}^m$  para las unidades de salida ( $l = m$ ) suele ser directo, ya que las unidades de activación  $\vec{h}_{[j]}^m$  se observan directamente en la expresión de pérdida.

Lo mismo se aplica a los modelos lineales poco profundos.

Para evaluar  $\vec{\delta}_{[j]}^l$  para las unidades ocultas, nuevamente hacemos uso de la regla de la cadena para derivadas parciales:

$$\vec{\delta}_{[j]}^l \equiv \frac{\partial L}{\partial \vec{h}_{[j]}^l} = \sum_k \left( \frac{\partial L}{\partial \vec{h}_{[k]}^{l+1}} \times \frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^l} \right) \quad (6.10)$$

La suma se realiza sobre todas las unidades  $\vec{h}_{[k]}^{l+1}$  a las que la unidad  $\vec{h}_{[j]}^l$  envía conexiones.

Suponemos que las conexiones se realizan solo entre capas consecutivas en la red (desde la capa  $l$  hasta la capa  $(l+1)$ ).

Las unidades  $\vec{h}_{[k]}^{l+1}$  podrían incluir otras unidades ocultas y/o unidades de salida.

Si ahora sustituimos la definición de  $\vec{\delta}_{[j]}^l$  dada por la ecuación (6.7) en la ecuación (6.10), obtenemos:

$$\vec{\delta}_{[j]}^l \equiv \frac{\partial L}{\partial \vec{h}_{[j]}^l} = \sum_k \left( \vec{\delta}_{[k]}^{(l+1)} \times \frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^l} \right) \quad (6.11)$$

Ahora, para la expresión  $\vec{h}_{[k]}^{l+1}$  podemos ir a su definición (ecuación 6.3):

$$\vec{h}_{[k]}^{(l+1)} = \left( \sum_i W_{[i,k]}^{l+1} \times \vec{z}_{[i]}^l \right) + \vec{b}_{[k]}^{(l+1)}$$

Reemplazando ahora la ecuación (6.4) ( $\vec{z}_{[i]}^l = g(\vec{h}_{[i]}^l)$ ) en la ecuación anterior, obtenemos:

$$\vec{h}_{[k]}^{(l+1)} = \left( \sum_i W_{[i,k]}^{l+1} \times g(\vec{h}_{[i]}^l) \right) + \vec{b}_{[k]}^{(l+1)}$$

Al calcular  $\frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^l}$ , todos los términos en la suma donde  $i \neq j$  se cancelan.

Por lo tanto, tenemos:

$$\frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^l} = W_{[j,k]}^{l+1} \times g'(\vec{h}_{[j]}^l) \quad (6.12)$$

Sustituyendo la ecuación (6.12) en la ecuación (6.11), obtenemos:

$$\vec{\delta}_{[j]}^l \equiv \frac{\partial L}{\partial \vec{h}_{[j]}^l} = \sum_k \left( \vec{\delta}_{[k]}^{(l+1)} \times W_{[j,k]}^{l+1} \times g'(\vec{h}_{[j]}^l) \right) \quad (6.13)$$

Dado que  $g'(\vec{h}_{[j]}^l)$  no depende de  $k$ , podemos obtener la siguiente fórmula de retropropagación:

$$\vec{\delta}_{[j]}^l = g'(\vec{h}_{[j]}^l) \times \sum_k \left( \vec{\delta}_{[k]}^{(l+1)} \times W_{[j,k]}^{l+1} \right) \quad (6.14)$$

Esto nos dice que el valor de  $\vec{\delta}$  para una unidad oculta en particular se puede obtener propagando los  $\vec{\delta}$  hacia atrás desde las unidades superiores en la red [Bishop, 2006].

El procedimiento de retropropagación se puede resumir de la siguiente manera:

1. Aplicar un vector de entrada  $\vec{x}$  a la red y propagarlo hacia adelante a través de la red utilizando las ecuaciones (6.3) y (6.4) para encontrar las activaciones de todas las unidades ocultas y de salida.
2. Calcular  $\vec{\delta}_{[j]}^m$  para todas las unidades de salida (recordar que las derivadas involucradas aquí son fáciles de calcular).
3. Retropropagar los  $\vec{\delta}_{[k]}^{(l+1)}$  utilizando la ecuación (6.14) para obtener  $\vec{\delta}_{[j]}^l$  para cada unidad oculta en la red. Se realiza de capas superiores a capas inferiores en la red.
4. Utilizar la ecuación (6.9) ( $\frac{\partial L}{\partial W_{[i,j]}} = \vec{\delta}_{[j]}^l \times \vec{z}_{[i]}^{(l-1)}$ ) para evaluar las derivadas requeridas.

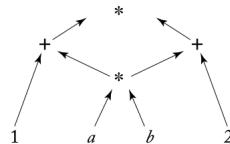
## 6.8 La Abstracción del Grafo de Cómputo

Uno puede calcular los gradientes de los varios parámetros de una red a mano e implementarlos en código. Sin embargo, este procedimiento es engorroso y propenso a errores. Por lo tanto, para la mayoría de los propósitos, es preferible utilizar herramientas automáticas para el cálculo de gradientes [Bengio, 2012].

Una representación de una computación matemática arbitraria (por ejemplo, una red neuronal) como un grafo es llamada un grafo de cómputo. Esta abstracción nos permite calcular los gradientes para cualquier tipo de arquitectura de red neuronal utilizando el algoritmo de retropropagación. La formulación anterior estaba restringida a redes feedforward.

Un grafo de cómputo es un grafo dirigido acíclico (DAG, por sus siglas en inglés). Los nodos corresponden a operaciones matemáticas o variables (ligadas), y las aristas corresponden al flujo de valores intermedios entre los nodos. La estructura del grafo define el orden de la computación en términos de las dependencias entre los diferentes componentes. Dado que el resultado de una operación puede ser la entrada de varias continuaciones, el grafo es un DAG y no un árbol.

Consideremos, por ejemplo, un grafo para el cálculo de  $(a * b + 1) * (a * b + 2)$ :



La computación de  $a * b$  es compartida. Dado que una red neuronal es esencialmente una expresión matemática, se puede representar como un grafo de cómputo.

La figura anterior muestra el grafo de cómputo para una MLP con una capa oculta y una transformación de salida softmax [Goldberg, 2017]. Los nodos ovalados representan operaciones matemáticas o funciones, y los nodos rectangulares sombreados representan parámetros (variables ligadas). Las entradas de la red se tratan como constantes y se dibujan sin un nodo circundante. Los nodos de entrada y parámetros no tienen aristas de entrada, y los nodos de salida no tienen aristas de salida. La salida de cada nodo es una matriz, cuya dimensionalidad se indica sobre el nodo.

Este grafo es incompleto: sin especificar las entradas, no podemos calcular una salida. La figura 5.1b muestra un grafo completo para una MLP que toma tres palabras como entradas y predice la distribución de etiquetas gramaticales para la tercera palabra. Este grafo se puede utilizar para la predicción, pero no para el entrenamiento, ya que la salida es un vector (no un escalar) y el grafo no tiene en cuenta la respuesta correcta ni el término de pérdida. Finalmente, el grafo en la figura 5.1c muestra el grafo de cómputo para un ejemplo de entrenamiento específico, en el cual las entradas son las (incrustaciones de) las palabras "the", "black", "dog", y la salida esperada es "NOUN"(cuyo índice es 5). El nodo de selección implementa una operación de indexación, recibiendo un vector y un índice (en este caso, 5) y devolviendo la entrada correspondiente en el vector.

### 6.8.1 Cómputo hacia Adelante

El paso hacia adelante (forward pass) calcula las salidas de los nodos en el grafo. Dado que la salida de cada nodo depende únicamente de sí mismo y de las aristas entrantes, es trivial calcular las salidas de todos los nodos.

Esto se hace recorriendo los nodos en un orden topológico y calculando la salida de cada nodo dado que las salidas de sus predecesores ya han sido calculadas.

Más formalmente, en un grafo de  $N$  nodos, asociamos a cada nodo un índice  $i$  de acuerdo con su orden topológico. Sea  $f_i$  la función calculada por el nodo  $i$  (por ejemplo, multiplicación, suma, etc.). Sea  $\pi(i)$  los nodos padres del nodo  $i$ , y  $\pi^{-1}(i) = \{j|i \in \pi(j)\}$  los nodos hijos del nodo  $i$  (estos son los argumentos de  $f_i$ ). Denotemos por  $v(i)$  la salida del nodo  $i$ , es decir, la aplicación de  $f_i$  a los valores de salida de sus argumentos  $\pi^{-1}(i)$ . Para los nodos de variables e entrada,  $f_i$  es una función constante y  $\pi^{-1}(i)$  está vacío. El paso hacia adelante en el grafo de cómputo calcula los valores  $v(i)$  para todos los  $i \in [1, N]$ .

---

**Algorithm 5.3** Computation graph forward pass.

---

```

1: for  $i = 1$  to  $N$  do
2:   Let  $a_1, \dots, a_m = \pi^{-1}(i)$ 
3:    $v(i) \leftarrow f_i(v(a_1), \dots, v(a_m))$ 
```

---

### 6.8.2 Cómputo hacia Atrás (Retropropagación)

El paso hacia atrás (backward pass) comienza designando un nodo  $N$  con una salida escalar ( $1 \times 1$ ) como nodo de pérdida y ejecutando el cómputo hacia adelante hasta ese nodo.

El cómputo hacia atrás calcula los gradientes de los parámetros con respecto al valor de ese nodo.

Denotemos por  $d(i)$  la cantidad  $\frac{\partial N}{\partial i}$ . El algoritmo de retropropagación se utiliza para calcular los valores  $d(i)$  para todos los nodos  $i$ .

El paso hacia atrás llena una tabla de valores  $d(1), \dots, d(N)$  como se muestra en el siguiente algoritmo.

---

**Algorithm 5.4** Computation graph backward pass (backpropagation).

---

1: $d(N) \leftarrow 1$	$\triangleright \frac{\partial N}{\partial N} = 1$
2: <b>for</b> $i = N-1$ to $1$ <b>do</b>	$\triangleright \frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \frac{\partial j}{\partial i}$
3: $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$	

---

El algoritmo de retropropagación sigue esencialmente la regla de la cadena de la diferenciación. La cantidad  $\frac{\partial f_j}{\partial i}$  es la derivada parcial de  $f_j(\pi^{-1}(j))$  con respecto al argumento  $i \in \pi^{-1}(j)$ . Este valor depende de la función  $f_j$  y los valores  $v(a_1), \dots, v(a_m)$  (donde  $a_1, \dots, a_m = \pi^{-1}(j)$ ) de sus argumentos, los cuales fueron calculados en el paso hacia adelante.

Por lo tanto, para definir un nuevo tipo de nodo, es necesario definir dos métodos: uno para calcular el valor hacia adelante  $v(i)$  basado en las entradas del nodo, y otro para calcular  $\frac{\partial f_j}{\partial i}$  para cada  $x \in \pi^{-1}(i)$ .

### 6.8.3 Resumen de la Abstracción del Grafo de Cómputo

Observa que la formulación anterior de la retropropagación es equivalente a la dada anteriormente en clase.

La abstracción del grafo de cómputo nos permite:

1. Construir fácilmente redes arbitrarias.
2. Evaluar sus predicciones para entradas dadas (paso hacia adelante).

3. Calcular gradientes para sus parámetros con respecto a pérdidas escalares arbitrarias (paso hacia atrás o retropropagación).

Una propiedad interesante de la abstracción del grafo de cómputo es que nos permite calcular los gradientes para redes arbitrarias (por ejemplo, redes con conexiones saltadas, pesos compartidos, funciones de pérdida especiales, etc.).

#### 6.8.4 Derivadas de funciones no matemáticas

Definir  $\frac{\partial f_j}{\partial i}$  para funciones matemáticas como  $\log$  o  $+$  es sencillo.

Puede resultar desafiante pensar en la derivada de operaciones como  $\text{pick}(\vec{x}, 5)$ , que selecciona el quinto elemento de un vector.

La respuesta es pensar en términos de la contribución al cálculo. Después de seleccionar el elemento  $i$ -ésimo de un vector, solo ese elemento participa en el resto del cálculo.

Por lo tanto, el gradiente de  $\text{pick}(\vec{x}, 5)$  es un vector  $\vec{v}$  con la dimensionalidad de  $\vec{x}$  donde  $\vec{v}_{[5]} = 1$  y  $\vec{v}_{[i \neq 5]} = 0$ .

De manera similar, para la función  $\max(0, x)$ , el valor del gradiente es 1 para  $x > 0$  y 0 en caso contrario.

## 6.9 Regularización y Dropout

Las redes de múltiples capas pueden ser grandes y tener muchos parámetros, lo que las hace especialmente propensas al sobreajuste.

La regularización del modelo es tan importante en las redes neuronales profundas como lo es en los modelos lineales, tal vez incluso más.

Las regularizaciones discutidas para modelos lineales, es decir,  $L_2$ ,  $L_1$  y la elastic-net, también son relevantes para las redes neuronales.

Otra técnica efectiva para evitar que las redes neuronales sobreajusten los datos de entrenamiento es el **dropout training** [Srivastava et al., 2014].

El método de dropout está diseñado para evitar que la red aprenda a depender de unidades o conexiones específicas en el proceso de entrenamiento, lo que ayuda a reducir el sobreajuste.

La idea básica detrás del dropout es apagar aleatoriamente unidades (neuronas) en cada paso de entrenamiento, lo que hace que la red aprenda a ser más robusta y generalice mejor.

El dropout se puede aplicar a las unidades ocultas (neuronas) y/o a las conexiones entre ellas.

Durante el entrenamiento, en cada paso, se aplica una máscara binaria aleatoria a las unidades o conexiones seleccionadas para el dropout. Las unidades o conexiones que están "pagadas" tienen un valor de cero y no contribuyen al cálculo hacia adelante ni hacia atrás. Solo las unidades o conexiones "encendidas" se utilizan en el cálculo de la predicción y en la retropropagación del error.

Durante la inferencia o evaluación, no se aplica el dropout y todas las unidades o conexiones se utilizan para realizar la predicción.

Es importante destacar que el dropout no es una técnica exclusiva de las redes neuronales, pero ha demostrado ser especialmente efectiva en este contexto debido a la gran cantidad de parámetros y conexiones que suelen tener las redes neuronales profundas.

El valor típico para la tasa de dropout, es decir, la fracción de unidades o conexiones que se apagan en cada paso de entrenamiento, suele ser del orden del 0.2 al 0.5. Sin embargo, el valor

---

<sup>0</sup>Un tutorial completo sobre el algoritmo de retropropagación sobre la abstracción del grafo de cómputo se puede encontrar aquí: <https://colah.github.io/posts/2015-08-Backprop/>.

óptimo puede variar según el problema y la arquitectura de la red. Por lo tanto, es recomendable experimentar con diferentes tasas de dropout para encontrar la mejor configuración para cada caso.

En resumen, la regularización y el dropout son técnicas efectivas para evitar el sobreajuste en las redes neuronales. Al utilizar regularización, como  $L_2$  o  $L_1$ , se penalizan los grandes valores de los parámetros, lo que ayuda a controlar la complejidad del modelo. El dropout, por otro lado, apaga aleatoriamente unidades o conexiones durante el entrenamiento, lo que promueve la robustez y generalización del modelo. Ambas técnicas pueden utilizarse en conjunto para obtener mejores resultados en la generalización y evitar el sobreajuste.

## 6.10 Frameworks de Aprendizaje Profundo

Existen varios paquetes de software que implementan el modelo de grafo de cómputo. Todos estos paquetes admiten todos los componentes esenciales (tipos de nodos) para definir una amplia gama de arquitecturas de redes neuronales.

Uno de estos paquetes es **TensorFlow** (<https://www.tensorflow.org/>), una biblioteca de software de código abierto para cálculos numéricos utilizando gráficos de flujo de datos, desarrollada originalmente por el equipo de Google Brain.

Otro paquete popular es **Keras**, que es una API de alto nivel para redes neuronales que se ejecuta sobre TensorFlow y otros backends (<https://keras.io/>).

También tenemos **PyTorch**, una biblioteca de aprendizaje automático de código abierto para Python basada en Torch, desarrollada por el grupo de investigación de inteligencia artificial de Facebook. PyTorch admite la construcción de gráficos dinámicos, lo que significa que se crea un grafo de cómputo diferente desde cero para cada muestra de entrenamiento (<https://pytorch.org/>).



## 7. Vectores de Palabra

Un componente importante en las redes neuronales para el lenguaje es el uso de una capa de embedding. Se trata de una asignación de símbolos discretos a vectores continuos. Cuando se realiza el embedding de palabras, estas pasan de ser símbolos distintos y aislados a objetos matemáticos en los que se puede operar. La distancia entre vectores puede equiparse a la distancia entre palabras. Esto facilita generalizar el comportamiento de una palabra a otra.

### 7.1 Hipótesis Distribucional y Matrices Palabra Contexto

La **hipótesis distribucional** [Harris, 1954] establece que las palabras que ocurren en los mismos **contextos** tienden a tener significados similares. O, en otras palabras, una palabra se caracteriza por las compañías que mantiene". Las **representaciones distribucionales** representan las palabras mediante **vectores de alta dimensionalidad** basados en los contextos en los que ocurren.

Los vectores distribucionales se construyen a partir de matrices de palabras-contexto  $M$ . Cada celda  $(i, j)$  es un valor de asociación basado en la co-ocurrencia entre una **palabra objetivo**  $w_i$  y un **contexto**  $c_j$ , calculado a partir de un corpus de documentos. Los contextos se definen comúnmente como ventanas de palabras que rodean a  $w_i$ . La longitud de la ventana  $k$  es un parámetro que va desde 1 hasta 8 palabras en ambos lados de  $w_i$ . Si el vocabulario de las palabras objetivo y las palabras de contexto es el mismo,  $M$  tiene una dimensionalidad de  $|\mathcal{V}| \times |\mathcal{V}|$ . Mientras que las ventanas más cortas capturan información sintáctica (por ejemplo, POS), las ventanas más largas capturan más probabilidades de similitud temática [Goldberg, 2016, Jurafsky and Martin, 2023].

Ejemplo: Vectores de Distribución con ventanas contextuales de tamaño 1

Las asociaciones entre palabras y contextos se pueden calcular utilizando diferentes enfoques:

1. Recuento de co-ocurrencias.
2. Información mutua puntual positiva (PPMI, por sus siglas en inglés).
3. Valores de significancia de una prueba t emparejada.

Según [Jurafsky and Martin, 2023], el más común de estos enfoques es PPMI. Los métodos de distribución también se conocen como métodos basados en conteo.

Example corpus:

- I like deep learning.
- I like NLP.
- I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Figura 7.1: Ejemplo tomado de: <http://cs224d.stanford.edu/lectures/CS224d-Lecture2.pdf>

## 7.2 PPMI

PPMI calcula el logaritmo de la probabilidad de que los pares palabra-contexto ocurran juntos en comparación con la probabilidad de que sean independientes.

$$\text{PMI}(w, c) = \log_2 \left( \frac{P(w, c)}{P(w)P(c)} \right) = \log_2 \left( \frac{\text{count}(w, c) \times |D|}{\text{count}(w) \times \text{count}(c)} \right) \quad (7.1)$$

Los valores de PMI negativos sugieren que la pareja ocurre menos a menudo de lo esperado por azar. Estas estimaciones no son confiables a menos que los conteos se calculen a partir de corpus muy grandes [Jurafsky and Martin, 2023]. PPMI corrige este problema reemplazando los valores negativos por cero.

$$\text{PPMI}(w, c) = \max(0, \text{PMI}(w, c)) \quad (7.2)$$

Los vectores distribuidos o word embeddings son vectores continuos de palabras de baja dimensionalidad que se entrena a partir de corpus de documentos utilizando redes neuronales. El tamaño de los vectores de distribución basados en recuento aumenta con el vocabulario, es decir, pueden tener una dimensionalidad muy alta. Almacenar explícitamente la matriz de co-ocurrencia puede requerir mucha memoria. Algunos modelos de clasificación no escalan bien con datos de alta dimensionalidad. La comunidad de redes neuronales prefiere utilizar representaciones **distribuidas**<sup>1</sup> o **word embeddings**. Los word embeddings son vectores densos de palabras de baja dimensionalidad que se entrena a partir de corpus de documentos utilizando redes neuronales. Las dimensiones no son directamente interpretables, es decir, representan características latentes de la palabra, “capturando esperanzadamente propiedades sintácticas y semánticas útiles” [Turian et al., 2010]. Se han convertido en un componente crucial de las arquitecturas de redes neuronales para el procesamiento del lenguaje natural.

Existen dos enfoques principales para obtener word embeddings:

<sup>1</sup>Idea: El significado de la palabra está "distribuido" en una combinación de dimensiones.

1. Capas de embedding: utilizando una capa de embedding en una arquitectura de red neuronal específica para una tarea, entrenada a partir de ejemplos etiquetados (por ejemplo, análisis de sentimientos).
2. Word embeddings pre-entrenados: creando una tarea predictiva auxiliar a partir de corpus no etiquetados (por ejemplo, predecir la siguiente palabra) en la que los word embeddings surgirán naturalmente a partir de la arquitectura de la red neuronal.

Estos enfoques también se pueden combinar: se puede inicializar una capa de embedding de una red neuronal específica para una tarea con word embeddings pre-entrenados obtenidos mediante el segundo enfoque. Los modelos más populares basados en el segundo enfoque son skip-gram [Mikolov et al., 2013], continuous bag-of-words [Mikolov et al., 2013] y GloVe [Pennington et al., 2014]. Los word embeddings han demostrado ser más poderosos que los enfoques distribucionales en muchas tareas de procesamiento del lenguaje natural [Baroni et al., 2014]. En [Amir et al., 2015], se utilizaron como características en un modelo de regresión para determinar la asociación entre palabras de Twitter y sentimientos positivos.

### 7.3 Modelo de Lenguaje Neuronal

En el año 2000, Bengio y colaboradores propusieron utilizar redes neuronales feed-forward para construir modelos de lenguaje [Bengio et al., 2000]. Utilizando la regla de la cadena (ver Capítulo 3) y restringiendo el contexto a un tamaño fijo de palabras (por ejemplo, 5), se puede modelar la probabilidad  $p(w|c)$  mediante una red neuronal. En este enfoque, las palabras anteriores que conforman el contexto se convierten en las entradas de la red neuronal, y la palabra siguiente es la salida deseada. Luego se aplica una función denominada softmax, que transforma las salidas de la red (que es un vector de puntajes para cada palabra del vocabulario) en una distribución de probabilidad. De esta forma es posible tomar un corpus de texto y recorrerlo, extrayendo ventanas de palabras junto con sus palabras siguientes para entrenar la red neuronal y desarrollar un modelo de lenguaje. Este proceso se ilustra en la Figura 7.2.

Las palabras en el contexto se transforman en vectores de dimensionalidad igual a la del vocabulario. Cada palabra se codifica utilizando la técnica “one-hot”, asignando un valor de 1 a la posición correspondiente a la palabra en el vocabulario y cero a todas las demás posiciones. Estos vectores luego son proyectados en una capa de embeddings, donde se convierten en vectores densos. Estos vectores resultantes se combinan en una capa intermedia y posteriormente se proyectan hacia la salida, cuya dimensionalidad coincide con la del vocabulario. Finalmente, se aplica la función Softmax para generar una distribución de probabilidad que abarca todas las palabras posibles en la salida.

Sin duda, la propiedad más interesante de los modelos de lenguaje neuronales es la noción de “word embedding” o representación vectorial de palabras. Esto implica proyectar las palabras discretas presentes en el contexto hacia vectores densos de cientos de dimensiones. Lo que resulta particularmente fascinante es que las palabras con significado relacionado (por ejemplo sinónimos) tienden a adquirir vectores o “embeddings” cercanos en el espacio vectorial. Este fenómeno encuentra su raíz en una teoría lingüística denominada hipótesis distribucional [Harris, 1954, Firth, 1957], que sostiene que las palabras presentes en contextos parecidos suelen compartir significados. Como resultado, los modelos de lenguaje neuronales se distinguen de sus contrapartes basadas en n-gramas al ser capaces de aprovechar contextos similares aunque estos difieran en las palabras utilizadas.

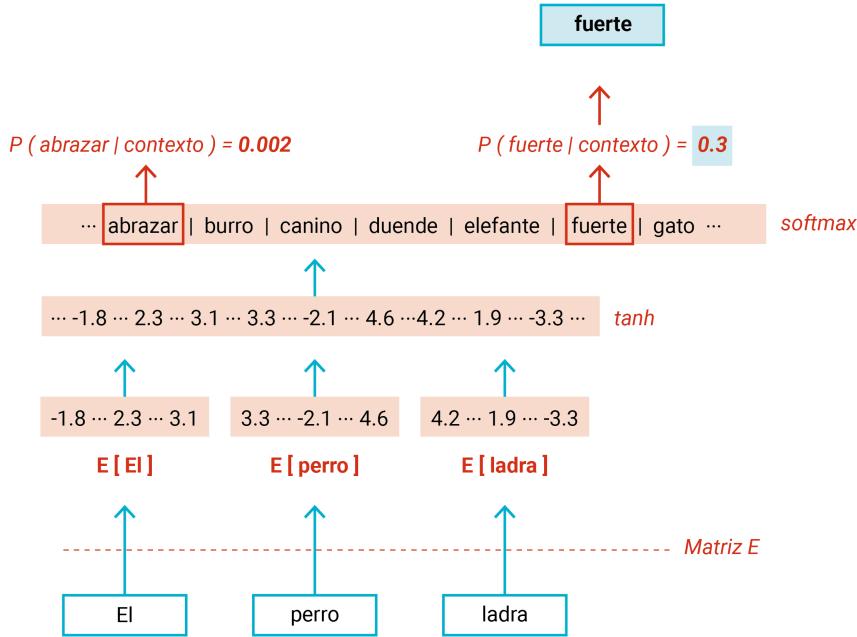


Figura 7.2: Modelo de Lenguaje Neuronal

## 7.4 Word2Vec

Tuvieron que pasar 13 años para que los modelos de lenguaje neuronales fueran adoptados de forma masiva. Esto fue posible gracias al lanzamiento del software de código abierto llamado Word2Vec [Mikolov et al., 2013], el cual permitía entrenar vectores de palabras empleando modelos similares a los propuestos por [Bengio et al., 2000], pero con una eficiencia superior. Este software permitió a miles de usuarios entrenar sus propios vectores de palabra utilizando sus colecciones de documentos particulares, lo que les permitió explorar las características intrínsecas de sus palabras.

Word2Vec es un paquete de software que implementa dos arquitecturas de redes neuronales para entrenar word embeddings: Continuous Bag of Words (CBOW) y Skip-gram. Implementa dos modelos de optimización: Muestreo Negativo (Negative Sampling) y Softmax Jerárquico (Hierarchical Softmax). Estos modelos son redes neuronales poco profundas que se entranan para predecir los contextos de las palabras. Se puede encontrar un tutorial muy completo sobre los algoritmos detrás de Word2Vec en <https://arxiv.org/pdf/1411.2738.pdf>.

El modelo Skip-gram entrena una red neuronal con una capa oculta para predecir las palabras que rodean a una palabra central, dentro de una ventana de tamaño  $k$  que se desplaza a lo largo del corpus de entrada. La palabra central y las  $k$  palabras circundantes corresponden a las capas de entrada y salida de la red. Inicialmente, las palabras se representan mediante vectores one-hot: vectores del tamaño del vocabulario ( $|V|$ ) con valores cero en todas las entradas excepto en el índice correspondiente a la palabra, que recibe un valor de 1.

La capa de salida combina los  $k$  vectores one-hot de las palabras circundantes. La capa oculta tiene una dimensionalidad  $d$ , que determina el tamaño de los embeddings (normalmente  $d \ll |V|$ ).

El modelo Skip-gram utiliza un softmax jerárquico donde el vocabulario se representa como

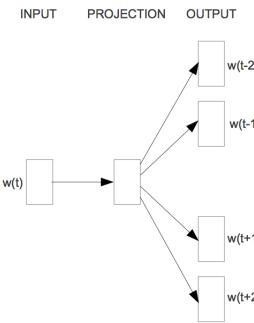
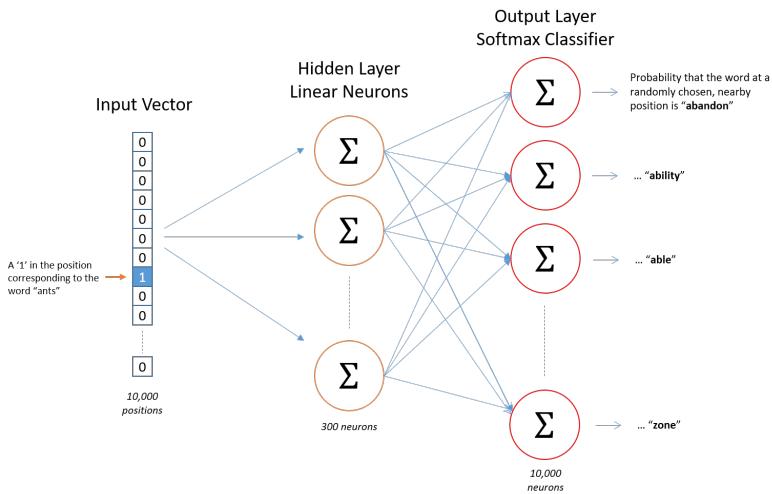


Figura 7.3: Imagen tomada del artículo original

Figura 7.4: Imagen tomada de: <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>

un árbol binario de Huffman. Esto se basa en observaciones anteriores de que la frecuencia de las palabras funciona bien para obtener clases en modelos de lenguaje basados en redes neuronales. Los árboles de Huffman asignan códigos binarios cortos a palabras frecuentes, lo que reduce aún más el número de unidades de salida que deben evaluarse.

Si dos palabras diferentes tienen contextos muy similares (es decir, qué palabras es probable que aparezcan a su alrededor), entonces nuestro modelo necesita generar resultados muy similares para estas dos palabras. Y una forma de lograr que la red genere predicciones de contexto similares para estas dos palabras es si los vectores de palabras son similares. Por lo tanto, si dos palabras tienen contextos similares, ¡nuestra red estará motivada a aprender vectores de palabras similares para estas dos palabras! ¡Ta-da!

¿Y qué significa que dos palabras tengan contextos similares? Creo que se podría esperar que sinónimos como "inteligente" y "stuto" tengan contextos muy similares. O que palabras relacionadas, como "motor" y "transmisión", probablemente también tengan contextos similares.

### 7.4.1 Parametrización del modelo Skip-gram

Se nos proporciona un corpus de entrada formado por una secuencia de palabras  $w_1, w_2, w_3, \dots, w_T$  y un tamaño de ventana  $k$ .

Denotamos las palabras objetivo o centrales con la letra  $w$  y las palabras de contexto circundantes con la letra  $c$ .

La ventana de contexto  $c_{1:k}$  de la palabra  $w_t$  corresponde a las palabras  $w_{t-k/2}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k/2}$  (asumiendo que  $k$  es un número par).

El objetivo del modelo Skip-gram es maximizar la log-probabilidad promedio de las palabras de contexto dadas las palabras objetivo:

$$\frac{1}{T} \sum_{t=1}^T \sum_{c \in c_{1:k}} \log P(c|w_t)$$

La probabilidad condicional de una palabra de contexto  $c$  dada una palabra central  $w$  se modela con una softmax ( $C$  es el conjunto de todas las palabras de contexto, que generalmente es igual al vocabulario):

$$P(c|w) = \frac{e^{\vec{c} \cdot \vec{w}}}{\sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}}$$

Los parámetros del modelo  $\theta$  son  $\vec{c}$  y  $\vec{w}$  (representaciones vectoriales de los contextos y las palabras objetivo).

Sea  $D$  el conjunto de pares palabra-contexto correctos (es decir, pares de palabras que se observan en el corpus).

El objetivo de optimización es maximizar la log-verosimilitud condicional de los contextos  $c$  (lo cual es equivalente a minimizar la pérdida de entropía cruzada):

$$\arg \max_{\vec{c}, \vec{w}} \sum_{(w,c) \in D} \log P(c|w) = \sum_{(w,c) \in D} (\log e^{\vec{c} \cdot \vec{w}} - \log \sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}) \quad (7.3)$$

Suposición: maximizar esta función resultará en buenos embeddings  $\vec{w}$ , es decir, palabras similares tendrán vectores similares.

El término  $P(c|w)$  es computacionalmente costoso debido a la suma  $\sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}$  sobre todos los contextos  $c'$ .

Solución: reemplazar la softmax por una softmax jerárquica (el vocabulario se representa con un árbol binario de Huffman).

Los árboles de Huffman asignan códigos binarios cortos a palabras frecuentes, lo que reduce el número de unidades de salida que deben evaluarse.

La hipótesis de distribución establece que las palabras en contextos similares tienen significados similares. El objetivo anterior claramente trata de aumentar la cantidad de buenos pares palabra-contexto y disminuirla para los malos. Intuitivamente, esto significa que las palabras que comparten muchos contextos serán similares entre sí (también se observa que los contextos que comparten muchas palabras también serán similares entre sí). Sin embargo, esto es muy simplista.

Fuente: <https://arxiv.org/pdf/1402.3722.pdf>

El modelo Skip-gram y el Negative Sampling no son lo mismo.

### 7.4.2 Skip-gram con Negative Sampling

El Negative Sampling (NS) se presenta como un modelo más eficiente para calcular los embeddings del Skip-gram. Sin embargo, optimiza una función objetivo diferente [Goldberg and Levy, 2014].

El NS maximiza la probabilidad de que un par palabra-contexto  $(w, c)$  provenga del conjunto de pares palabra-contexto correctos  $D$  utilizando una función sigmoide:

$$P(D = 1|w, c_i) = \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}}$$

Suposición: las palabras de contexto  $c_i$  son independientes entre sí:

$$P(D = 1|w, c_{1:k}) = \prod_{i=1}^k P(D = 1|w, c_i) = \prod_{i=1}^k \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}}$$

Esto conduce a la siguiente función objetivo (log-verosimilitud):

$$\arg \max_{\vec{c}, \vec{w}} \log P(D = 1|w, c_{1:k}) = \sum_{i=1}^k \log \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}} \quad (7.4)$$

Esta función objetivo tiene una solución trivial si establecemos  $\vec{w}$ ,  $\vec{c}$  de manera que  $P(D = 1|w, c) = 1$  para cada par  $(w, c)$  de  $D$ . Esto se logra estableciendo  $\vec{w} = \vec{c}$  y  $\vec{w} \cdot \vec{c} = K$  para todos los  $\vec{w}$ ,  $\vec{c}$ , donde  $K$  es un número grande. Necesitamos un mecanismo que evite que todos los vectores tengan el mismo valor, al no permitir algunas combinaciones  $(w, c)$ . Una forma de hacerlo es presentarle al modelo algunos pares  $(w, c)$  para los cuales  $P(D = 1|w, c)$  debe ser bajo, es decir, pares que no están en los datos. Esto se logra muestreando ejemplos negativos de  $\tilde{D}$ . Se muestrean  $m$  palabras para cada par palabra-contexto  $(w, c) \in D$ . Se agrega cada palabra muestreada  $w_i$  junto con el contexto original  $c$  como un ejemplo negativo en  $\tilde{D}$ .  $\tilde{D}$  es  $m$  veces más grande que  $D$ . El número de ejemplos negativos  $m$  es un parámetro del algoritmo.

Función objetivo final:

$$\arg \max_{\vec{c}, \vec{w}} \sum_{(w, c) \in D} \log P(D = 1|w, c_{1:k}) + \sum_{(w, c) \in \tilde{D}} \log P(D = 0|w, c_{1:k}) \quad (7.5)$$

Las palabras negativas se muestrean a partir de una versión suavizada de las frecuencias del corpus:

$$\frac{\#(w)^{0,75}}{\sum_{w'} \#(w')^{0,75}}$$

Esto otorga más peso relativo a las palabras menos frecuentes.

### 7.4.3 Continuous Bag of Words: CBOW

Similar al modelo Skip-gram, pero ahora se predice la palabra central a partir del contexto circundante.

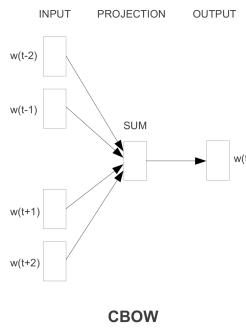


Figura 7.5: Imagen tomada de: <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>

#### 7.4.4 GloVe

GloVe (de global vectors) es otro método popular para entrenar word embeddings [Pennington et al., 2014]. Construye una matriz explícita palabra-contexto y entrena los vectores de palabra y contexto  $\vec{w}$  y  $\vec{c}$  intentando satisfacer la siguiente ecuación:

$$\vec{w} \cdot \vec{c} + b_{[w]} + b_{[c]} = \log \#(w, c) \quad \forall (w, c) \in D \quad (7.6)$$

donde  $b_{[w]}$  y  $b_{[c]}$  son sesgos entrenados específicos de la palabra y el contexto.

En términos de factorización de matrices, si fijamos  $b_{[w]} = \log \#(w)$  y  $b_{[c]} = \log \#(c)$ , obtendremos un objetivo muy similar a la factorización de la matriz PMI palabra-contexto, desplazada por  $\log(|D|)$ . En GloVe, los parámetros de sesgo se aprenden y no se fijan, lo que le da otro grado de libertad.

El objetivo de optimización es la pérdida de mínimos cuadrados ponderada, asignando más peso a la reconstrucción correcta de elementos frecuentes.

Cuando se utiliza el mismo vocabulario de palabras y contextos, el modelo sugiere representar cada palabra como la suma de sus vectores de embedding de palabra y contexto correspondientes.

### 7.5 Analogías de palabras

Una propiedad descubierta fue la capacidad de realizar analogías semánticas a través de operaciones aritméticas en el espacio vectorial de las palabras tales como “hombre es a mujer como rey es a reina”, relaciones verbales como “nadar es a nadando como comer es a comiendo”, además de establecer relaciones entre países y sus capitales, como “Santiago es a Chile como Madrid es a España”, entre otras. Ver Figura 7.6.

Los word embeddings pueden capturar ciertas relaciones semánticas, como relaciones de género, tiempo verbal y país-capital entre palabras.

Por ejemplo, la siguiente relación se encuentra en los word embeddings entrenados con Word2Vec:  $\vec{w}_{king} - \vec{w}_{man} + \vec{w}_{woman} \approx \vec{w}_{queen}$ .

### 7.6 Evaluación

Existen muchos conjuntos de datos con asociaciones de palabras anotadas por humanos o analogías de oro que se pueden utilizar para evaluar algoritmos de word embeddings.

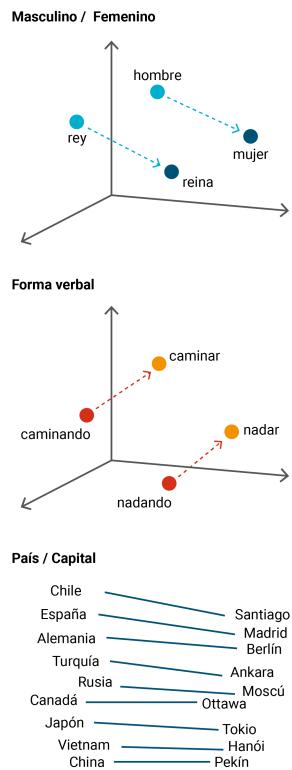


Figura 7.6: Relaciones Encontradas

Estos enfoques se llaman “Enfoques de Evaluación Intrínseca”.

La mayoría de ellos están implementados en: <https://github.com/kudkudak/word-embeddings-benchmarks>.

Los word embeddings también se pueden evaluar extrínsecamente utilizando en tareas externas de procesamiento del lenguaje natural (por ejemplo, etiquetado de partes del discurso, análisis de sentimientos).

## 7.7 Correspondencia entre modelos distribuidos y distribucionales

Tanto los métodos distribucionales "basados en recuento" como los distribuidos "neurales" se basan en la hipótesis de distribución.

Ambos intentan capturar la similitud entre palabras basándose en la similitud entre los contextos en los que ocurren.

Levy y Goldeberg mostraron en [Levy and Goldberg, 2014] que el modelo Skip-gram con negative sampling (SGNS) está factorizando implícitamente una matriz palabra-contexto, cuyas celdas son la información

mutua puntual (PMI) de los pares respectivos de palabra y contexto, desplazada por una constante global.

Esto vincula los métodos neuronales y los tradicionales "basados en recuento", sugiriendo que en un sentido profundo, las dos familias algorítmicas son equivalentes.

## 7.8 FastText

Los embeddings de FastText amplían el modelo skipgram teniendo en cuenta la estructura interna de las palabras mientras aprenden las representaciones de las palabras [Bojanowski et al., 2016].

Se asocia una representación vectorial con cada  $n$ -gramo de caracteres.

Las palabras se representan como la suma de estas representaciones.

Tomando la palabra *where* y  $n = 3$ , se representará por los  $n$ -gramos de caracteres: <wh, whe, her, ere, re>, y la secuencia especial <where>.

Es importante tener en cuenta que la secuencia <her>, correspondiente a la palabra "her", es diferente del tri-grama "her" de la palabra "here".

FastText es útil para los idiomas con una rica morfología. Por ejemplo, las palabras *amazing* y *mazingly* comparten información en FastText a través de sus  $n$ -gramos compartidos, mientras que en Word2Vec estas dos palabras no tienen ninguna relación.

Sea  $\mathcal{G}_w$  el conjunto de  $n$ -gramos que aparecen en  $w$ .

FastText asocia un vector  $\vec{g}$  con cada  $n$ -gramo en  $\mathcal{G}_w$ .

En FastText, la probabilidad de que un par palabra-contexto  $(w, c)$  provenga del corpus de entrada  $D$  se calcula de la siguiente manera:

$$P(D|w, c) = \frac{1}{1 + e^{-s(w, c)}}$$

donde,

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \vec{g} \cdot \vec{c}.$$

El algoritmo de muestreo negativo se puede calcular de la misma forma que en el modelo skipgram con esta formulación.

## 7.9 Embeddings de frases específicas de sentimiento

El problema de los embeddings de palabras es que los antónimos pueden usarse en contextos similares, por ejemplo, "mi auto es lindo" vs "mi auto es feo".

En [Tang et al., 2014], se proponen embeddings de palabras específicos de sentimiento combinando el modelo skipgram con tweets con emoticones anotados :) :(.

Estos embeddings se utilizan para entrenar un clasificador de polaridad a nivel de palabra.

El modelo integra la información de sentimiento en la representación continua de frases mediante el desarrollo de una arquitectura neural adaptada.

Entrada:  $\{w_i, s_j, pol_j\}$ , donde  $w_i$  es una frase (o palabra),  $s_j$  es la oración y  $pol_j$  es la polaridad de la oración.

El objetivo de entrenamiento utiliza el embedding de  $w_i$  para predecir las palabras de contexto (de la misma manera que el modelo skipgram) y utiliza la representación de la oración  $se_j$  para predecir  $pol_j$ .

Las oraciones ( $se_j$ ) se representan promediando los vectores de palabras que las componen.

El objetivo de la parte de sentimiento es maximizar el promedio de la probabilidad logarítmica del sentimiento:

$$f_{sentimiento} = \frac{1}{S} \sum_{j=1}^S \log p(\text{pol}_j | se_j)$$

El objetivo de entrenamiento final es maximizar la combinación lineal de los objetivos skipgram y sentimiento:

$$f = \alpha f_{skipgram} + (1 - \alpha) f_{sentimiento}$$

## 7.10 Gensim

Gensim es una biblioteca de Python de código abierto para el procesamiento del lenguaje natural que implementa muchos algoritmos para entrenar word embeddings.

- <https://radimrehurek.com/gensim/>
- <https://machinelearningmastery.com/develop-word-embeddings-python-gensim/>







## 8. Etiqetado de Secuencias

### 8.1 Tareas de Etiqetado de Secuencias o Etiqetado

- El etiqetado de secuencias o etiqetado es una tarea en PNL que difiere de la clasificación de documentos.
- El objetivo es asignar etiqetas o etiqetas a una oración representada como una secuencia de tokens  $x_1, x_2, \dots, x_n$ .
- En particular, el objetivo del etiqetado de secuencias es asignar etiqetas a palabras, o más generalmente, asignar etiqetas discretas a elementos discretos en una secuencia [Eisenstein, 2018].
- Ejemplos conocidos de esta tarea son el etiqetado de partes del discurso (POS) y el reconocimiento de entidades nombradas (NER).

### 8.2 Etiqetado de Partes del Discurso

**ENTRADA:** Los beneficios aumentaron en Boeing Co., superando fácilmente las previsiones en Wall Street, cuando su CEO Alan Mulally anunció los resultados del primer trimestre.

**SALIDA:** Los beneficios/N aumentaron/V en/P Boeing/N Co./N ./, superando/V fácilmente/ADV las/DET previsiones/N en/P Wall/N Street/N ./, cuando/P su/DET CEO/N Alan/N Mulally/N anunció/V los/DET resultados/N del/PREP primer/ADJ trimestre/N ./.

- N = Sustantivo
- V = Verbo
- P = Preposición
- Adv = Adverbio
- Adj = Adjetivo
- ...

Fuente: [Jurafsky and Martin, 2023]

Tag	Description	Example
Open Class	<b>ADJ</b> Adjective: noun modifiers describing properties	<i>red, young, awesome</i>
	<b>ADV</b> Adverb: verb modifiers of time, place, manner	<i>very, slowly, home, yesterday</i>
	<b>NOUN</b> words for persons, places, things, etc.	<i>algorithm, cat, mango, beauty</i>
	<b>VERB</b> words for actions and processes	<i>draw, provide, go</i>
	<b>PROPN</b> Proper noun: name of a person, organization, place, etc..	<i>Regina, IBM, Colorado</i>
Closed Class Words	<b>INTJ</b> Interjection: exclamation, greeting, yes/no response, etc.	<i>oh, um, yes, hello</i>
	<b>ADP</b> Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	<i>in, on, by, under</i>
	<b>AUX</b> Auxiliary: helping verb marking tense, aspect, mood, etc.,	<i>can, may, should, are</i>
	<b>CCONJ</b> Coordinating Conjunction: joins two phrases/clauses	<i>and, or, but</i>
	<b>DET</b> Determiner: marks noun phrase properties	<i>a, an, the, this</i>
	<b>NUM</b> Numeral	<i>one, two, first, second</i>
	<b>PART</b> Particle: a function word that must be associated with another word	<i>'s, not, (infinitive) to</i>
	<b>PRON</b> Pronoun: a shorthand for referring to an entity or event	<i>she, who, I, others</i>
	<b>SCONJ</b> Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	<i>that, which</i>
Other	<b>PUNCT</b> Punctuation	<i>: , )</i>
	<b>SYM</b> Symbols like \$ or emoji	<i>\$, %</i>
	<b>X</b> Other	<i>asdf, qwfg</i>

### 8.3 Reconocimiento de Entidades Nombradas (NER)

Una **entidad nombrada** es, hablando en términos generales, cualquier cosa a la que se puede hacer referencia con un nombre propio: una persona, un lugar, una organización.

**ENTRADA:** Los beneficios aumentaron en Boeing Co., superando fácilmente las previsiones en Wall Street, cuando su CEO Alan Mulally anunció los resultados del primer trimestre.

**SALIDA:** Los beneficios aumentaron en [Compañía Boeing Co.], superando fácilmente las previsiones en [Lugar Wall Street], cuando su CEO [Persona Alan Mulally] anunció los resultados del primer trimestre.

- Dado que las entidades pueden abarcar varias palabras (es decir, un problema de reconocimiento de fragmentos), podemos utilizar el etiquetado BIO [Ramshaw and Marcus, 1999] para convertir el problema en un problema de etiquetado de secuencias.
- Etiquetado BIO: utilizar etiquetas que capturen tanto el límite como el tipo de entidad nombrada.

#### Etiquetado BIO: NER como Etiquetado de Secuencias

**ENTRADA:** Los beneficios aumentaron en Boeing Co., superando fácilmente las previsiones en Wall Street, cuando su CEO Alan Mulally anunció los resultados del primer trimestre.

**SALIDA:** Los/O beneficios/O aumentaron/O en/O Boeing/B-C Co./I-C ,/O superando/O fácilmente/O las/O previsiones/O en/O Wall/B-L Street/I-L ,/O cuando/O su/O CEO/O Alan/B-P Mulally/I-P anunció/O los/O resultados/O del/O primer/O trimestre/O ./O

- O = Fuera (sin entidad)
- B-C = Comienzo de Compañía
- I-C = Dentro de Compañía
- B-L = Comienzo de Lugar
- I-L = Dentro de Lugar
- B-P = Comienzo de Persona

- I-P = Dentro de Persona

Nuestro objetivo: **Conjunto de entrenamiento:**

1. Pierre/NNP Vinken/NNP/, 61/CD años/NNS de edad/JJ/, se unirá a la junta directiva como director no ejecutivo el 29 de noviembre./NNP
2. El Sr./NNP Vinken/NNP es presidente/NN de Elsevier/NNP N.V./NNP, el grupo editorial holandés./NNP
3. Rudolph/NNP Agnew/NNP/, de 55/CD años/NNS y presidente/NN de...

### 8.3.1 Etiquetado de secuencias como aprendizaje supervisado

- Tenemos una secuencia de entradas  $x = (x_1, x_2, \dots, x_n)$  y las etiquetas correspondientes  $y = (y_1, y_2, \dots, y_n)$ .
- La tarea consiste en aprender una función  $f$  que mapea secuencias de entrada a secuencias de etiquetas:  $f(x_1, x_2, \dots, x_n) = y_1, y_2, \dots, y_n$ .
- Tenemos un conjunto de entrenamiento de secuencias etiquetadas:  $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$ .

### 8.3.2 Enfoque generativo para el etiquetado de secuencias

- Los modelos generativos, como el clasificador de Naive Bayes utilizado para clasificación, también se pueden utilizar para tareas de etiquetado de secuencias en PLN.
- Enfoque:
  - Entrenamiento: Aprender la distribución conjunta  $p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$  de las secuencias de entrada.
  - Decodificación: Utilizar la distribución aprendida para predecir secuencias de etiquetas para nuevas secuencias de entrada.
- La decodificación en el etiquetado de secuencias implica encontrar la secuencia de etiquetas con la mayor probabilidad conjunta:  $\arg \max_{y_1, y_2, \dots, y_n} p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$ .

## 8.4 Modelos Ocultos de Markov

- Los Modelos Ocultos de Markov (HMM, por sus siglas en inglés) proporcionan una forma sistemática de manejar problemas de etiquetado de secuencias mediante modelos generativos y algoritmos de decodificación eficientes.
- Tenemos una oración de entrada  $x = x_1, x_2, \dots, x_n$  ( $x_i$  es la  $i$ -ésima palabra en la oración).
- Tenemos una secuencia de etiquetas  $y = y_1, y_2, \dots, y_n$  ( $y_i$  es la  $i$ -ésima etiqueta en la oración).
- Usaremos un HMM para definir  $p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$  para cualquier oración  $x_1, \dots, x_n$  y secuencia de etiquetas  $y_1, \dots, y_n$  de la misma longitud [Kupiec, 1992].
- Luego, la secuencia de etiquetas más probable para  $x$  es:

$$\arg \max_{y_1, \dots, y_n} p(x_1, \dots, x_n, y_1, \dots, y_n)$$

## 8.5 Modelos Ocultos de Markov Trigramas (Trigram HMM)

Para cualquier oración  $x_1, \dots, x_n$  donde  $x_i \in V$  para  $i = 1, \dots, n$ , y cualquier secuencia de etiquetas  $y_1, \dots, y_{n+1}$  donde  $y_i \in S$  para  $i = 1, \dots, n$ , y  $y_{n+1} = \text{STOP}$ , la probabilidad conjunta de la oración y la secuencia de etiquetas es:

$$p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^n e(x_i | y_i)$$

donde hemos asumido que  $x_0 = x_{-1} = *$ .

### 8.5.1 Parámetros del modelo

- $q(s|u, v)$  para cualquier  $s \in S \cup \{\text{STOP}\}$ ,  $u, v \in S \cup \{*\}$ 
  - El valor de  $q(s|u, v)$  se puede interpretar como la probabilidad de ver la etiqueta  $s$  inmediatamente después del bigrama de etiquetas  $(u, v)$ .
- $e(x|s)$  para cualquier  $s \in S$ ,  $x \in V$ 
  - El valor de  $e(x|s)$  se puede interpretar como la probabilidad de ver la observación  $x$  emparejada con el estado  $s$ .

#### Un ejemplo

Si tenemos  $n = 3$ ,  $x_1, x_2, x_3$  igual a la oración “the dog laughs”, y  $y_1, y_2, y_3, y_4$  igual a la secuencia de etiquetas “D N V STOP”, entonces:

$$\begin{aligned} p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) &= q(D|*, *) \times q(N|*, D) \\ &\quad \times q(V|D, N) \times q(\text{STOP}|N, V) \\ &\quad \times e(\text{el}|D) \times e(\text{perro}|N) \times e(\text{se}|V) \times e(\text{ríe}|\text{STOP}) \end{aligned}$$

- STOP es una etiqueta especial que termina la secuencia.
- Tomamos  $y_0 = y_{-1} = *$ , donde \* es un símbolo especial de "padding".

## 8.6 Supuestos de independencia en los HMM trigramas

- Los Modelos Ocultos de Markov trigramas (HMM trigramas) se derivan mediante la realización de supuestos específicos de independencia en el modelo.
- Consideremos dos secuencias de variables aleatorias:  $X_1, \dots, X_n$  y  $Y_1, \dots, Y_n$ , donde  $n$  es la longitud de las secuencias.
- Cada  $X_i$  puede tomar cualquier valor en un conjunto finito  $V$  de palabras, y cada  $Y_i$  puede tomar cualquier valor en un conjunto finito  $K$  de etiquetas posibles (por ejemplo,  $K = \{D, N, V, \dots\}$ ).
- Nuestro objetivo es modelar la probabilidad conjunta:

$$\begin{aligned} P(x_1, x_2, \dots, x_n, y_1, \dots, y_n) &= p(y_1) \times p(y_2|y_1) \\ &\quad \times \dots \\ &\quad \times p(y_n|y_{n-1}, y_{n-2}, \dots, y_1) \\ &\quad \times p(x_1|y_n, y_{n-1}, \dots, y_1) \\ &\quad \times p(x_2|x_1, y_n, y_{n-1}, \dots, y_1) \\ &\quad \times p(x_n|x_{n-1}, \dots, x_1, y_n, y_{n-1}, \dots, y_1) \end{aligned}$$

- Definimos una variable aleatoria adicional  $Y_{n+1}$  que siempre toma el valor "STOP".
- La idea clave en los HMM es la factorización de la probabilidad conjunta:

$$\begin{aligned} P(X_1 = x_1, \dots, X_n = x_n, Y_1 = y_1, \dots, Y_{n+1} = y_{n+1}) &= \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \times \prod_{i=1}^n P(X_i = x_i | Y_i = y_i) \end{aligned}$$

- Primero asumimos que:

$$P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) = q(y_i | y_{i-2}, y_{i-1})$$

- Esto asume que la secuencia  $Y_1, \dots, Y_{n+1}$  es una secuencia de Markov de segundo orden, donde cada estado depende solo de los dos estados anteriores.
- Y también asumimos que:

$$P(X_i = x_i | Y_i = y_i) = e(x_i | y_i)$$

- Esto asume que las observaciones y los estados están condicionalmente independientes dadas las etiquetas.

## 8.7 ¿Por qué el nombre?

$$\begin{aligned} p(x_1, \dots, x_n, y_1, \dots, y_n) &= q(\text{STOP} | y_{n-1}, y_n) \\ &\quad \times \prod_{j=1}^n q(y_j | y_{j-2}, y_{j-1}) \\ &\quad \times \prod_{j=1}^n e(x_j | y_j) \end{aligned}$$

Los componentes de la cadena de Markov:

$$q(\text{STOP} | y_{n-1}, y_n) \times \prod_{j=1}^n q(y_j | y_{j-2}, y_{j-1})$$

Estas transiciones no se observan directamente para una secuencia dada de palabras  $(x_1, \dots, x_n)$ , de ahí el nombre de ".ocultas".

El componente observable:

$$\prod_{j=1}^n e(x_j | y_j)$$

El componente observable de los HMM modela las probabilidades de emisión de los símbolos observados ( $x$ ) condicionados a los estados ocultos correspondientes ( $y$ ).

## 8.8 Estimación Suavizada

$$\begin{aligned} q(Vt | DT, JJ) &= \lambda_1 \times \frac{\text{Count}(Dt, JJ, Vt)}{\text{Count}(Dt, JJ)} \\ &\quad + \lambda_2 \times \frac{\text{Count}(JJ, Vt)}{\text{Count}(JJ)} \\ &\quad + \lambda_3 \times \frac{\text{Count}(Vt)}{\text{Count}()} \end{aligned}$$

donde  $\lambda_1 + \lambda_2 + \lambda_3 = 1$ , y para todo  $i$ ,  $\lambda_i \geq 0$ .

$$e(\text{base}|Vt) = \frac{\text{Count}(Vt, \text{base})}{\text{Count}(Vt)}$$

## 8.9 Tratando con Palabras de Baja Frecuencia

Un método común es el siguiente:

- Paso 1: Dividir el vocabulario en dos conjuntos
  - Palabras frecuentes = palabras que ocurren  $\geq 5$  veces en el entrenamiento
  - Palabras de baja frecuencia = todas las demás palabras
- Paso 2: Mapear las palabras de baja frecuencia a un conjunto pequeño y finito, dependiendo de prefijos, sufijos, etc.

A continuación se muestra un ejemplo de clases de palabras para el reconocimiento de entidades nombradas [Bikel et al., 1999]:

Clase de palabra	Ejemplo	Intuición
twoDigitNum	90	Año de dos dígitos
fourDigitNum	1990	Año de cuatro dígitos
containsdígito y letra	A8956 – 67	Código de producto
containsDigitAndDash	09 – 96	Fecha
containsDigitAndSlash	11/9/89	Fecha
containsDigitAndComma	23,000,00	Cantidad monetaria
containsDigitAndPeriod	1,00	Cantidad monetaria, porcentaje
othernum	456789	Otro número
allCaps	BBN	Organización
capPeriod	M.	Inicial de nombre de persona
firstWord	Primera palabra de la oración	Sin información útil de capitalización
initCap	Sally	Palabra capitalizada
lowercase	can	Palabra sin capitalizar
other	,	Signos de puntuación, todas las demás palabras

## 8.10 Problema de Decodificación

Problema de decodificación: Para una entrada  $x_1 \dots x_n$ , encontrar

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

donde  $\arg \max$  se toma sobre todas las secuencias  $y_1 \dots y_{n+1}$  tales que  $y_i \in S$  para  $i = 1 \dots n$ , y  $y_{n+1} = \text{STOP}$ .

Suponemos que  $p$  toma la forma:

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^n e(x_i | y_i)$$

Recordemos que hemos asumido en esta definición que  $y_0 = y_{-1} = *$  y  $y_{n+1} = \text{STOP}$ .

### 8.10.1 Método Bruto Ingenuo

El método bruto ingenuo para encontrar la secuencia de etiquetas con la puntuación más alta es enumerar todas las posibles secuencias de etiquetas  $y_1, \dots, y_{n+1}$ , calcular su puntuación utilizando la función  $p$  y seleccionar la secuencia con la puntuación más alta.

- Ejemplo:
  - Oración de entrada: *the dog barks*
  - Conjunto de etiquetas posibles:  $K = \{D, N, V\}$
- Enumerar todas las posibles secuencias de etiquetas:
  - *D D D STOP*
  - *D D N STOP*
  - *D D V STOP*
  - *D N D STOP*
  - *D N N STOP*
  - *D N V STOP*
  - ...

En este caso, hay  $3^3 = 27$  secuencias posibles. Sin embargo, para oraciones más largas, este método se vuelve ineficiente. Para una oración de entrada de longitud  $n$ , hay  $|K|^n$  secuencias posibles de etiquetas. El crecimiento exponencial hace que la búsqueda exhaustiva sea impracticable para oraciones de longitud razonable.

## 8.11 Decodificación de Viterbi con Programación Dinámica

El algoritmo utilizado por los HMM para realizar una decodificación eficiente se denomina decodificación de Viterbi. La decodificación de Viterbi utiliza programación dinámica, que es una técnica para resolver problemas de optimización dividiéndolos en subproblemas superpuestos. Al almacenar las soluciones a estos subproblemas en una tabla, no es necesario recalcularlos, lo que mejora considerablemente la eficiencia de los algoritmos. A continuación, mostramos cómo funciona la programación dinámica con dos ejemplos: el factorial y Fibonacci.

### Factorial

- Implementación recursiva:

```
def recur_factorial(n):
    # Caso base
    if n == 1:
        return n
    else:
        return n * recur_factorial(n-1)
```

- Implementación con programación dinámica:

```
def dynamic_factorial(n):
    tabla = [0 for i in range(0, n+1)]

    # Caso base
    tabla[0] = 1

    for i in range(1, len(tabla)):
```

```

tabla[i] = i * tabla[i-1]

return tabla[n]

```

### Fibonacci

- Implementación recursiva:

```

def recur_fibonacci(n):
    if n == 1 or n == 0:
        return 1
    else:
        return recur_fibonacci(n-1) + recur_fibonacci(n-2)

```

- Implementación con programación dinámica:

```

def dynamic_fibonacci(n):
    tabla = [0 for i in range(0, n+1)]

    # Caso base
    tabla[0] = 1
    tabla[1] = 1

    for i in range(2, len(tabla)):
        tabla[i] = tabla[i-1] + tabla[i-2]

    return tabla[n]

```

### Complejidad

En las implementaciones recursivas, la complejidad puede ser bastante alta debido a los cálculos repetidos de los mismos subproblemas. Sin embargo, la programación dinámica puede reducir significativamente la complejidad al almacenar las soluciones a los subproblemas en una tabla o matriz y reutilizarlas cuando sea necesario. Este enfoque elimina los cálculos redundantes y permite una computación más eficiente. En el caso de Fibonacci, la complejidad se reduce de exponencial a lineal.

## 8.12 El Algoritmo de Viterbi

El algoritmo de Viterbi calcula eficientemente la máxima probabilidad de una secuencia de etiquetas utilizando programación dinámica.

#### Definiciones:

- Definimos  $n$  como la longitud de la oración.
- Definimos  $S_k$  para  $k = -1 \dots n$  como el conjunto de etiquetas posibles en la posición  $k$ :  $S_{-1} = S_0 = \{*\}$ ,  $S_k = S$  para  $k \in \{1 \dots n\}$ .
- Definimos una versión truncada de la probabilidad codificada por el HMM hasta la posición  $k$ ,  $r(y_{-1}, y_0, y_1, \dots, y_k)$ , como:

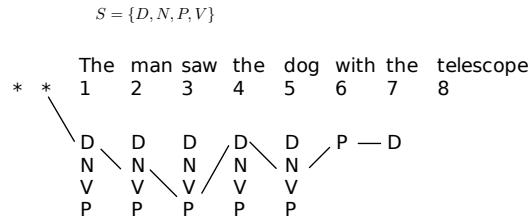
$$r(y_{-1}, y_0, y_1, \dots, y_k) = \prod_{i=1}^k q(y_i | y_{i-2}, y_{i-1})$$

- Definimos una tabla de programación dinámica  $\pi(k, u, v)$  como la máxima probabilidad de una secuencia de etiquetas que termina en las etiquetas  $u, v$  en la posición  $k$ :

$$\pi(k, u, v) = \max_{y_{-1}, y_0, y_1, \dots, y_k : y_{k-1} = u, y_k = v} r(y_{-1}, y_0, y_1, \dots, y_k)$$

### Un Ejemplo

Recuerde que  $\pi(k, u, v)$  es la máxima probabilidad de una secuencia de etiquetas que termina en las etiquetas  $u, v$  en la posición  $k$ .



- Hay muchas secuencias posibles de etiquetas.
- Cada una de ellas tiene una probabilidad calculada a partir de los parámetros  $q$  y  $e$ .
- $\pi(7, P, D)$  es la máxima probabilidad de que una de estas secuencias de etiquetas termine en  $P$  y  $D$  en la posición 7.
- La ruta representa la secuencia con la máxima probabilidad.

#### 8.12.1 Una Definición Recursiva

**Caso base:**

$$\pi(0, *, *) = 1$$

**Definición recursiva:**

Para cualquier  $k \in \{1 \dots n\}$ , para cualquier  $u \in S_{k-1}$  y  $v \in S_k$ :

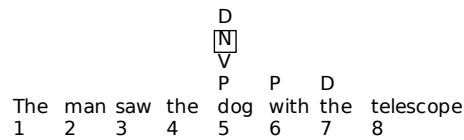
$$\pi(k, u, v) = \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

### Justificación de la Definición Recursiva

Para cualquier  $k \in \{1 \dots n\}$ , para cualquier  $u \in S_{k-1}$  y  $v \in S_k$ :

$$\pi(k, u, v) = \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

- Consideraremos una secuencia de etiquetas arbitraria que termina con las etiquetas  $P$  y  $D$  en la posición 7.
- Debe contener alguna etiqueta en la posición 5.
- Básicamente, estamos buscando la etiqueta que maximiza la probabilidad en la posición 5.



$$\mathcal{S}_5 = \mathcal{S} = \{D, N, V, P\}$$

$$\Pi(7, P, D) = \max_{w \in \mathcal{S}_5} (\Pi(6, w, P) \times q(D|w, P) \times e(\text{the}|D))$$

### 8.12.2 El Algoritmo de Viterbi

---

#### Algorithm 2: Algoritmo de Viterbi

---

**Entrada:** una oración  $x_1 \dots x_n$ , parámetros  $q(s|u, v)$  y  $e(x|s)$   
**Inicialización:** Establecer  $\pi(0, *, *) = 1$ ;  $S_{-1} = S_0 = \{*\}$ ,  $S_k = S$  para  $k \in \{1 \dots n\}$ .

```

for  $k = 1$  to  $n$  do
    for  $u \in S_{k-1}, v \in S_k$  do
        |  $\pi(k, u, v) = \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$ 
    end
end
retornar ( $\max_{u \in S_{n-1}, v \in S_n} (\pi(n, u, v) \times q(STOP|u, v))$ )

```

---

### 8.12.3 El Algoritmo de Viterbi con Punteros de Retroceso

---

#### Algorithm 3: Algoritmo de Viterbi con Punteros de Retroceso

---

**Entrada:** una oración  $x_1 \dots x_n$ , parámetros  $q(s|u, v)$  y  $e(x|s)$   
**Inicialización:** Establecer  $\pi(0, *, *) = 1$ ;  $S_{-1} = S_0 = \{*\}$ ,  $S_k = S$  para  $k \in \{1 \dots n\}$ .

```

for  $k = 1$  to  $n$  do
    for  $u \in S_{k-1}, v \in S_k$  do
        |
        |  $\pi(k, u, v) = \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$ 
        |
        |  $bp(k, u, v) = \arg \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$ 
    end
end
 $(y_{n-1}, y_n) = \arg \max_{(u, v)} (\pi(n, u, v) \times q(STOP|u, v))$ ; // Find maximum probability
// and corresponding tags
for  $k = (n-2)$  to 1 do
    |  $y_k = bp(k+2, y_{k+1}, y_{k+2})$ ; // Retrieve tag sequence using backpointers
end
retornar (the tag sequence  $y_1 \dots y_n$ ) ; // Return the final tag sequence

```

---

En el Algoritmo 3, se utiliza el puntero de retroceso para reconstruir la secuencia de etiquetas con la máxima probabilidad. El algoritmo recorre las tablas  $\pi$  en orden inverso, comenzando desde la posición  $n$  hasta la posición 1. En cada iteración, se elige la etiqueta  $u$  y  $v$  que maximizan la expresión

$\pi(k, u, v) \times q(\text{STOP}|u, v)$ . Estas etiquetas se agregan al comienzo de la secuencia de etiquetas *path*, y al final del algoritmo, se devuelve la secuencia *path*[1 : –1], eliminando las etiquetas de inicio y detención.

Con el algoritmo de Viterbi, podemos encontrar eficientemente la secuencia de etiquetas más probable para una oración dada, utilizando programación dinámica y aprovechando la estructura de dependencia entre las etiquetas en el HMM. Esto es especialmente útil en problemas de etiquetado de secuencias, como el etiquetado de partes del habla en NLP.

#### 8.12.4 El Algoritmo de Viterbi: Tiempo de Ejecución

El tiempo de ejecución del algoritmo de Viterbi se puede analizar de la siguiente manera:

- Se requiere un tiempo de  $O(n|S|^3)$  para calcular  $q(s|u, v) \times e(x_k|s)$  para todos los valores de  $k, s, u, v$ .
  - La tabla  $\pi$  tiene  $n|S|^2$  entradas que deben ser llenadas.
  - Se necesita un tiempo de  $O(|S|)$  para llenar una entrada.
- Por lo tanto, el tiempo de ejecución total es  $O(n|S|^3)$ .

#### Ventajas y Desventajas

El uso de etiquetadores basados en Modelos de Markov Ocultos (HMM) presenta ciertas ventajas y desventajas:

- Los etiquetadores HMM son fáciles de entrenar (se recopilan conteos a partir de un corpus de entrenamiento).
- Tienen un rendimiento relativamente bueno (más del 90 % de precisión en el reconocimiento de entidades nombradas).
- La principal dificultad radica en modelar  $e(\text{palabra}|etiqueta)$ , lo cual puede ser muy complejo si las "palabras" son complejas.

Aunque los etiquetadores HMM tienen ventajas en términos de simplicidad y rendimiento, la modelización de la probabilidad condicional  $e(\text{palabra}|etiqueta)$  puede resultar desafiante en situaciones donde las palabras son complejas o ambiguas. En tales casos, se pueden requerir técnicas más avanzadas para mejorar la precisión del etiquetado de secuencias.

### 8.13 MEMMs

- Los modelos de Markov de entropía máxima (MEMMs) utilizan modelos log-lineales multiclase para tareas de etiquetado de secuencias [McCallum et al., 2000].
- En la literatura temprana de PLN, la regresión logística a menudo se llamaba clasificación de entropía máxima [Eisenstein, 2018].
- Por lo tanto, los MEMMs se parecerán mucho a los modelos de softmax multiclase vistos en la conferencia sobre modelos lineales.
- A diferencia de los HMM, aquí confiamos en funciones parametrizadas.
- El objetivo de los MEMMs es modelar la siguiente distribución condicional:

$$P(s_1, s_2, \dots, s_m | x_1, \dots, x_m)$$

- Donde cada  $x_j$  para  $j = 1 \dots m$  es el símbolo de entrada  $j$  (por ejemplo, la  $j$ -ésima palabra en una oración), y cada  $s_j$  para  $j = 1 \dots m$  es la  $j$ -ésima etiqueta<sup>1</sup>.

---

<sup>1</sup>Estas diapositivas se basan en notas de conferencia de Michael Collins <http://www.cs.columbia.edu/>

- Esperaríamos que  $P(\text{DET}, \text{NOUN}, \text{VERB}|\text{the}, \text{dog}, \text{barks})$  sea mayor que  $P(\text{VERB}, \text{VERB}, \text{VERB}|\text{the}, \text{dog}, \text{barks})$  en un modelo entrenado a partir de un conjunto de datos de entrenamiento de etiquetado POS.
- Usamos  $S$  para denotar el conjunto de etiquetas posibles.
- Suponemos que  $S$  es un conjunto finito.
- Por ejemplo, en el etiquetado de partes del habla en inglés,  $S$  sería el conjunto de todas las posibles partes del habla en inglés (sustantivo, verbo, determinante, preposición, etc.).
- Dada una secuencia de palabras  $x_1, \dots, x_m$ , hay  $k^m$  secuencias posibles de partes del habla  $s_1, \dots, s_m$ , donde  $k = |S|$  es el número de partes del habla posibles.
- Queremos estimar una distribución sobre estas  $k^m$  secuencias posibles.
- En un primer paso, los MEMMs usan la siguiente descomposición ( $s_0$  siempre tiene una etiqueta especial \*):

$$\begin{aligned} P(s_1, s_2, \dots, s_m | x_1, \dots, x_m) &= \prod_{i=1}^m P(s_i | s_1, \dots, s_{i-1}, x_1, \dots, x_m) \\ &= \prod_{i=1}^m P(s_i | s_{i-1}, x_1, \dots, x_m) \end{aligned} \tag{8.1}$$

- La primera igualdad es exacta (se sigue de la regla de la cadena de probabilidades condicionales).
- La segunda igualdad se deriva de un supuesto de independencia, a saber, que para todo  $i$ ,

$$P(s_i | s_1, \dots, s_{i-1}, x_1, \dots, x_m) = P(s_i | s_{i-1}, x_1, \dots, x_m)$$

- Por lo tanto, estamos haciendo una suposición de Markov de primer orden similar a la suposición de Markov realizada en los HMM<sup>2</sup>.
- La etiqueta en la posición  $i$  depende solo de la etiqueta en la posición  $(i-1)$ .
- Habiendo hecho estas suposiciones de independencia, luego modelamos cada término usando un modelo log-lineal multiclasa (softmax):

$$P(s_i | s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))} \tag{8.2}$$

Aquí,  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)$  es un vector de características donde:

- $x_1, \dots, x_m$  es la oración completa que se está etiquetando.
- $i$  es la posición a etiquetar (puede tomar cualquier valor de 1 a  $m$ ).
- $s_{i-1}$  es el valor de la etiqueta anterior (puede tomar cualquier valor en  $S$ ).
- $s_i$  es el nuevo valor de la etiqueta (puede tomar cualquier valor en  $S$ ).

El alcance del vector de características está **restringido** a toda la secuencia de entrada  $x_1, x_m$ , y solo las etiquetas anteriores y actuales. Esta restricción permite el entrenamiento eficiente tanto de los MEMMs como de los CRFs.

<sup>1</sup>mcollins/crf.pdf. La notación y la terminología se han adaptado para ser coherentes con el resto del curso.

<sup>2</sup>De hecho, hicimos una suposición de Markov de segundo orden en los HMM. Los MEMMs también se pueden extender a suposiciones de segundo orden.

### 8.14 Ejemplo de características utilizadas en el etiquetado de partes del habla

1.  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[1]} = 1$  si  $s_i = \text{ADVERB}$  y la palabra  $x_i$  termina en “-ly”; 0 en caso contrario.

Si el peso  $\vec{w}_{[1]}$  asociado con esta característica es grande y positivo, entonces esta característica está diciendo básicamente que preferimos etiquetas donde las palabras que terminan en -ly se etiquetan como ADVERB.

2.  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[2]} = 1$  si  $i = 1$ ,  $s_i = \text{VERB}$  y  $x_m = ?$ ; 0 en caso contrario.

Si el peso  $\vec{w}_{[2]}$  asociado con esta característica es grande y positivo, entonces se prefieren las etiquetas que asignan VERB a la primera palabra de una pregunta (por ejemplo, “¿Es esta una oración que comienza con un verbo?”).

3.  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[3]} = 1$  si  $s_{i-1} = \text{ADJECTIVE}$  y  $s_i = \text{NOUN}$ ; 0 en caso contrario.

Nuevamente, un peso positivo para esta característica significa que los adjetivos tienden a ir seguidos de sustantivos.

4.  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[4]} = 1$  si  $s_{i-1} = \text{PREPOSITION}$  y  $s_i = \text{PREPOSITION}$ .

Un peso negativo  $\vec{w}_{[4]}$  para esta función significaría que las preposiciones no tienden a seguir a las preposiciones.

<sup>3</sup>Fuente: <https://blog.echen.me/2012/01/03/introduction-to-conditional-random-fields/>

### 8.15 Plantillas de características

Es posible definir plantillas de características más generales que cubran unigramas, bigramas, n-gramas de palabras, así como valores de etiqueta de  $s_{i-1}$  y  $s_i$ .

1. Una plantilla de características de unigramas de palabras y unigramas de etiquetas:  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[\text{index}(j,z)]} = 1$  si  $s_i = \text{TAG}_{[j]}$  y  $x_i = \text{WORD}_{[z]}$ ; 0 en caso contrario  $\forall j, z$ .

Nótese que  $j$  es un índice que abarca todas las etiquetas posibles en  $S$  y  $z$  es otro índice que abarca las palabras en el vocabulario  $V$ .

2. Una plantilla de características de bigramas de palabras y bigramas de etiquetas:  $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[\text{index}(j,z,u,v)]} = 1$  si  $s_{i-1} = \text{TAG}_{[j]}$ ,  $s_i = \text{TAG}_{[z]}$ ,  $x_{i-1} = \text{WORD}_{[u]}$  y  $x_i = \text{WORD}_{[v]}$ ; 0 en caso contrario  $\forall j, z, u, v$ .

La función  $\text{index}(j, k, \dots)$  asignará a cada característica diferente un índice único en el vector de características.

Nótese que el vector resultante tendrá una dimensión muy alta y será disperso.

#### Ejemplo

### 8.16 MEMMs y Softmax Multiclasificación

- Observemos que el modelo log-lineal mencionado anteriormente es muy similar al modelo softmax multiclasificación presentado en la conferencia sobre modelos lineales.
- Un modelo log-lineal general tiene la siguiente forma:

$$P(y|x; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x, y))}{\sum_{y' \in Y} \exp(\vec{w} \cdot \vec{\phi}(x, y'))}$$

- Un modelo softmax multiclasificación tiene la siguiente forma:

$$\begin{aligned} \hat{y} &= \text{softmax}(\vec{x} \cdot W + \vec{b}) \\ \hat{y}_{[i]} &= \frac{e^{(\vec{x} \cdot W + \vec{b})_{[i]}}}{\sum_j e^{(\vec{x} \cdot W + \vec{b})_{[j]}}} \end{aligned} \tag{8.3}$$

Figura 8.1: Ejemplo de características en el etiquetado de partes del habla

$$P(s_i|s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))}$$

Example:

1	2	3	4
The dog barks loudly			
DT	NN	VB	ADV

Let's check that  $P(s_4 = \text{ADV}|s_3 = \text{VB, the,dog,barks,loudly}) > P(s_4 = \text{VB}|s_3 = \text{VB, the,dog,barks,loudly})$

$$P(s_4 = \text{ADV}|s_3 = \text{VB, the,dog,barks,loudly}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(\text{the,dog,barks,loudly, 4, VB, ADV}))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(\text{the,dog,barks,loudly, 4, VB, } s'))}$$

$$P(s_4 = \text{VB}|s_3 = \text{VB, the,dog,barks,loudly}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(\text{the,dog,barks,loudly, 4, VB, VB}))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(\text{the,dog,barks,loudly, 4, VB, } s'))}$$

- Diferencia 1: en el modelo log-lineal tenemos un vector de parámetros fijo  $\vec{w}$  en lugar de tener múltiples vectores (una columna de  $W$  por cada valor de clase).
- Diferencia 2: el vector de características del modelo log-lineal  $\vec{\phi}(x, y)$  incluye información de la etiqueta  $y$ , mientras que el vector de entrada  $\vec{x}$  del modelo softmax es independiente de  $y$ .
- Los modelos log-lineales permiten usar características que consideran la interacción entre  $x$  e  $y$  (por ejemplo,  $x$  termina en "ly" e  $y$  es un ADVERBIO).

## 8.17 Entrenamiento de los MEMMs

- Una vez que hemos definido los vectores de características  $\vec{\phi}$ , podemos entrenar los parámetros  $\vec{w}$  del modelo de la misma manera que se entrena los modelos lineales.
- Establecemos la log-verosimilitud negativa como función de pérdida y optimizamos los parámetros utilizando descenso de gradiente a partir de los ejemplos de entrenamiento.
- Esto es equivalente a usar la pérdida de entropía cruzada.
- Cualquier pérdida que consista en una log-verosimilitud negativa es una entropía cruzada entre la distribución empírica definida por el conjunto de entrenamiento y la distribución de probabilidad definida por el modelo" [Goodfellow et al., 2016].

## 8.18 Decodificación con MEMMs

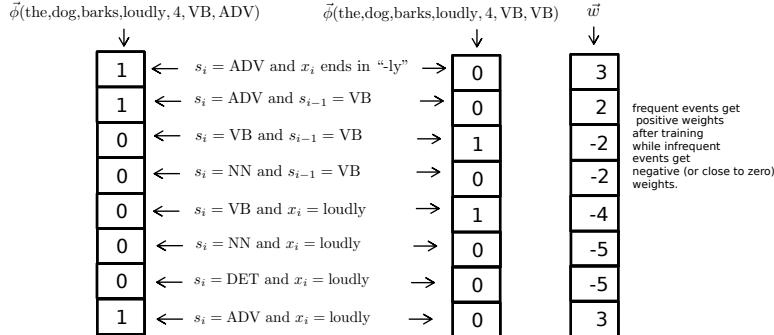
- El problema de decodificación es el siguiente.
- Se nos proporciona una nueva secuencia de prueba  $x_1, \dots, x_m$ .
- Nuestro objetivo es calcular la secuencia de estados más probable para esta secuencia de prueba,

$$\arg \max_{s_1, \dots, s_m} P(s_1, \dots, s_m | x_1, \dots, x_m). \quad (8.4)$$

- Hay  $k^m$  posibles secuencias de estados, por lo que para cualquier longitud de oración razonablemente grande  $m$ , la búsqueda exhaustiva de todas las posibilidades no será posible.
- Podemos usar el algoritmo de Viterbi de manera similar a como se usa para los HMM.

Figura 8.2: Ejemplo de características en el etiquetado de partes del habla (continuación)  
This is the same as checking if:

$$\vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{ADV}) > \vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{VB})$$



$$\vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{ADV}) = 1 * 3 + 1 * 2 + 1 * 3 = 6$$

$$> \vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{VB}) = 1 * -2 + 1 * -4 = -6$$

- La estructura de datos básica en el algoritmo será una tabla de programación dinámica  $\pi$  con entradas  $\pi[j, s]$  para  $j = 1, \dots, m$  y  $s \in S$ .
- $\pi[j, s]$  almacenará la probabilidad máxima para cualquier secuencia de estados que termine en el estado  $s$  en la posición  $j$ .
- Formalmente, nuestro algoritmo calculará

$$\pi[j, s] = \max_{s_1, \dots, s_{j-1}} \left( P(s|s_{j-1}, x_1, \dots, x_m) \prod_{k=1}^{j-1} P(s_k|s_{k-1}, x_1, \dots, x_m) \right)$$

para todo  $j = 1, \dots, m$  y para todo  $s \in S$ .

El algoritmo es el siguiente:

- Inicialización: para  $s \in S$

$$\pi[1, s] = P(s|s_0, x_1, \dots, x_m)$$

donde  $s_0$  es un estado especial "inicial".

- Para  $j \in \{2, \dots, m\}$ ,  $s \in \{1, \dots, k\}$

$$\pi[j, s] = \max_{s' \in S} [\pi[j-1, s'] \times P(s|s', x_1, \dots, x_m)]$$

- Finalmente, después de haber completado los valores de  $\pi[j, s]$  para todos los  $j, s$ , podemos calcular

$$\max_{s_1, \dots, s_m} = \max_s \pi[m, s].$$

- El algoritmo se ejecuta en tiempo  $O(mk^2)$  (es decir, lineal en la longitud de la secuencia  $m$ , y cuadrático en el número de etiquetas  $k$ ).
- Al igual que en el algoritmo de Viterbi para HMM, podemos calcular la secuencia con el puntaje más alto utilizando punteros inversos en el algoritmo de programación dinámica.

### 8.19 Comparación entre MEMMs y HMMs

- Entonces, ¿cuál es la motivación para usar MEMMs en lugar de HMMs?
- Observa que los algoritmos de decodificación Viterbi para los dos modelos son muy similares.
- En MEMMs, la probabilidad asociada con cada transición de estado  $s_{i-1}$  a  $s_i$  es

$$P(s_i|s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))}$$

- En los HMM, la probabilidad asociada a cada transición es:

$$P(s_i|s_{i-1}, x_1, \dots, x_m) = P(s_i|s_{i-1})P(x_i|s_i)$$

- La ventaja clave de MEMMs es que el uso de vectores de características  $\vec{\phi}$  permite obtener representaciones mucho más ricas que las utilizadas en los HMM.
- Por ejemplo, la probabilidad de transición puede ser sensible a cualquier palabra en la secuencia de entrada  $x_1, \dots, x_m$ .
- Además, es muy fácil introducir características que sean sensibles a las características ortográficas (por ejemplo, prefijos o sufijos) de la palabra actual  $x_i$ , o de las palabras circundantes.
- Estas características son útiles en muchas aplicaciones de PNL, y son difíciles de incorporar dentro de HMMs de una manera limpia.

### 8.20 Campos Aleatorios Condicionales (CRFs)

- Ahora pasamos a hablar de Campos Aleatorios Condicionales (CRFs) [Lafferty et al., 2001].
- Notación: para mayor comodidad, usaremos  $x_{1:m}$  para referirnos a una secuencia de entrada  $x_1, \dots, x_m$ , y  $s_{1:m}$  para referirnos a una secuencia de etiquetas  $s_1, \dots, s_m$ .
- El conjunto de todas las etiquetas posibles es nuevamente  $S$ .
- El conjunto de todas las posibles secuencias de etiquetas es  $S^m$ .
- En los campos aleatorios condicionales construiremos un modelo de

$$P(s_1, \dots, s_m | x_1, \dots, x_m) = P(s_{1:m} | x_{1:m})$$

- Una primera idea clave en los CRFs será definir un vector de características que mapea una secuencia de entrada completa  $x_{1:m}$  emparejada con una secuencia de etiquetas completa  $s_{1:m}$  a un vector de características de  $d$  dimensiones:

$$\vec{\Phi}(x_{1:m}, s_{1:m}) \in \mathbb{R}^d$$

- Pronto daremos una definición concreta para  $\vec{\Phi}$ .

- Por ahora, asumamos que existe alguna definición.
- A menudo nos referiremos a  $\vec{\Phi}$  como un vector de características "global".
- Es global en el sentido de que tiene en cuenta toda la secuencia de estados.
- En los CRFs construimos un modelo log-lineal gigante:

$$P(s_{1:m}|x_{1:m}; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))}$$

- Este es "solo" otro modelo log-lineal, pero es "gigante".
- El espacio de posibles valores para  $s_{1:m}$  es enorme  $S^m$ .
- La constante de normalización (denominador en la expresión anterior) implica una suma sobre todas las posibles secuencias de etiquetas  $S^m$ .
- Estos problemas podrían parecer causar graves problemas computacionales.
- Bajo suposiciones apropiadas, podemos entrenar y decodificar de manera eficiente con este tipo de modelo.
- Definimos el vector de características global  $\vec{\Phi}(x_{1:m}, s_{1:m})$  de la siguiente manera:

$$\vec{\Phi}(x_{1:m}, s_{1:m}) = \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$$

donde  $\vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$  es igual a los vectores de características utilizados en los MEMMs.

- Ejemplo:  $\vec{\Phi}([\text{the}, \text{dog}, \text{barks}], \text{DET}, \text{NOUN}, \text{VERB}) = \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 1, *, \text{DET}) + \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 2, \text{DET}, \text{NO}) + \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 3, \text{NOUN}, \text{VERB})$
- Esencialmente, estamos sumando muchos vectores dispersos.

### Ejemplo

	will	to	fight	$\Phi(x, \text{NN TO VB})$
$\Phi(x, 1, y_1, y_0)$	1	0	0	1
$\Phi(x, 2, y_2, y_1)$	1	0	0	1
$\Phi(x, 3, y_3, y_2)$	0	0	0	0
	0	0	0	0
$\Phi(x, 1, y_1, y_0)$	0	1	0	1
$\Phi(x, 2, y_2, y_1)$	0	1	0	1
$\Phi(x, 3, y_3, y_2)$	0	0	0	0
$\Phi(x, 1, y_1, y_0)$	0	0	1	1
$\Phi(x, 2, y_2, y_1)$	0	0	1	1

Figura 8.3: CRF Example

<sup>4</sup>Fuente: [http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/12\\_neural\\_sequence\\_labeling.pdf](http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/12_neural_sequence_labeling.pdf)

- Suponemos que, para cualquier dimensión de  $\vec{\Phi}_{[k]}, k = 1, \dots, d$ , el  $k$ -ésimo atributo global es:

$$\vec{\Phi}(x_{1:m}, s_{1:m})_{[k]} = \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)_{[k]}$$

- Por lo tanto,  $\vec{\Phi}(x_{1:m}, s_{1:m})_{[k]}$  se calcula sumando el vector de características "local"  $\vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)_{[k]}$  sobre las  $m$  transiciones de etiquetas diferentes en  $s_1, \dots, s_m$ .
- Esperaríamos que cada vector local codifique información relevante sobre la transición de etiquetas activando algunas dimensiones del vector (estableciendo el valor en uno).
- Ahora pasamos a dos problemas prácticos críticos en los CRFs: primero, la decodificación, y segundo, la estimación de parámetros (entrenamiento).

## 8.21 Decodificación con CRFs

- El problema de decodificación en los CRFs es el siguiente.
- Dada una secuencia de entrada  $x_{1:m} = x_1, x_2, \dots, x_m$ , nos gustaría encontrar la secuencia de estados subyacente más probable bajo el modelo, es decir,

$$\begin{aligned} \arg \max_{s_{1:m} \in S^m} P(s_{1:m} | x_{1:m}; \vec{w}) &= \arg \max_{s_{1:m} \in S^m} \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))} \\ &= \arg \max_{s_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m})) \\ &= \arg \max_{s_{1:m} \in S^m} \vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}) \\ &= \arg \max_{s_{1:m} \in S^m} \vec{w} \cdot \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j) \\ &= \arg \max_{s_{1:m} \in S^m} \sum_{j=1}^m \vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j) \end{aligned} \tag{8.5}$$

- Hemos demostrado que encontrar la secuencia más probable bajo el modelo es equivalente a encontrar la secuencia que maximiza:

$$\arg \max_{s_{1:m} \in S^m} \sum_{j=1}^m \vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$$

- Este problema tiene una intuición clara. Cada transición de la etiqueta  $s_{j-1}$  a la etiqueta  $s_j$  tiene un puntaje asociado:  $\vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$ .
- Este puntaje podría ser positivo o negativo.
- Intuitivamente, este puntaje será relativamente alto si la transición de estados es plausible, relativamente bajo si esta transición es implausible.
- El problema de decodificación consiste en encontrar una secuencia completa de estados tal que la suma de los puntajes de transición sea maximizada.
- Podemos resolver este problema utilizando una variante del algoritmo de Viterbi, de manera muy similar al algoritmo de decodificación para HMM o MEMM.

## 8.22 Estimación de Parámetros en CRFs (Entrenamiento)

- Para la estimación de parámetros, asumimos que tenemos un conjunto de  $n$  ejemplos etiquetados,  $\{(x_{1:m}^i, s_{1:m}^i)\}_{i=1}^n$ . Cada  $x_{1:m}^i$  es una secuencia de entrada  $x_1^i, \dots, x_m^i$  y cada  $s_{1:m}^i$  es una secuencia de etiquetas  $s_1^i, \dots, s_m^i$ .
- Nuevamente, establecemos la log-verosimilitud negativa (o entropía cruzada) como la función de pérdida  $L$  y optimizamos los parámetros utilizando el descenso de gradiente.
- El principal desafío aquí es que los cálculos del gradiente  $\frac{\partial L}{\partial w^{[k]}}$  implican una suma sobre  $S^m$  (un conjunto muy grande que contiene todas las posibles secuencias de etiquetas).
- Esta suma se puede calcular de manera eficiente utilizando el algoritmo de avance-atrás (Forward-backward algorithm)<sup>5</sup>.
- Este es otro algoritmo de programación dinámica que está estrechamente relacionado con el algoritmo de Viterbi.

## 8.23 CRFs y MEMMs

- Los CRFs y los MEMMs son modelos de etiquetado de secuencias discriminatorios: modelan la probabilidad condicional directamente a través de una función lineal-logarítmica multiclas parametrizada (softmax).
- Los HMM, por otro lado, son modelos generativos.
- En los MEMM, la normalización (denominador del softmax) es local: ocurre en cada paso de la etiqueta (la suma se realiza sobre todos los valores de etiqueta posibles  $S$ ).
- En los CRFs, la normalización es global: la suma se realiza sobre todas las posibles secuencias de etiquetas  $S^m$ .
- Entrenar un MEMM es bastante fácil: simplemente se entrena un modelo log-lineal multiclas para una palabra dada a la etiqueta. Este clasificador se utiliza en cada paso de la palabra para predecir toda la secuencia.
- Entrenar un CRF es más complejo. El objetivo es maximizar la log-probabilidad de la secuencia más probable.

### 8.23.1 CRFs y MEMMs: el problema del sesgo de etiqueta

- Los MEMMs terminan tomando decisiones en cada paso de manera independiente.
- Esto lleva a un problema llamado "sesgo de etiqueta": en algunas configuraciones del espacio de etiquetas, los MEMMs esencialmente ignoran aspectos importantes del contexto.
- Ejemplo: La etiqueta correcta del POS para la oración "will to fight"(voluntad de luchar) es "NN TO VB"<sup>6</sup>.
- Aquí, NN significa "sustantivo", TO significa infinitivo az VB significa "forma base verbal".
- Los modales (MD) aparecen con mucha más frecuencia al comienzo de la oración que los sustantivos (por ejemplo, en preguntas).
- Por lo tanto, la etiqueta "MDrecibirá un puntaje más alto que la etiqueta "NNcuando  $x_0 = \text{"will"}$ :  $P(s_1 = MD | s_0 = *, x_1 = \text{"will"}, \dots) > P(s_1 = NN | s_{i-1} = *, x_1 = \text{"will"})$ .
- Pero sabemos que MD + TO es muy raro: "puede comer", "quisiera comer".

<sup>5</sup><http://www.cs.columbia.edu/~mcollins/fb.pdf>

<sup>6</sup>Aquí estamos utilizando el conjunto de etiquetas PENN Treebank: <https://www.eecis.udel.edu/~vijay/cis889/ie/pos-set.pdf>

- La palabra "to" es relativamente determinística (casi siempre tiene la etiqueta TO), por lo que no importa qué etiqueta la preceda.
- Debido a la normalización local de los MEMMs,  $P(s_i = TO | s_{i-1}, x_1, \dots, x_i = "to", \dots, x_n)$  siempre será 1 cuando  $x_i = "to"$ , independientemente del valor de  $s_{i-1}$  (MD o NN).
- Eso significa que nuestra predicción para "to" no puede ayudarnos a desambiguar "will".
- Perdemos la información de que las secuencias MD + TO rara vez ocurren.
- Como consecuencia, es probable que un MEMM etiquete la primera palabra como "MD".
- Los CRFs superan este problema al realizar una normalización global: consideran el puntaje de toda la secuencia antes de normalizarlo y convertirlo en una distribución de probabilidades.

## 8.24 Enlaces

- [http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/11\\_memm\\_crf.pdf](http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/11_memm_crf.pdf)
- [http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/12\\_neural\\_sequence\\_labeling.pdf](http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/12_neural_sequence_labeling.pdf)
- <https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/>
- <https://www.quora.com/What-are-the-pros-and-cons-of-these-three-sequence-models-MaxEnt->
- <https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf>
- [http://www.davidsbatista.net/blog/2017/11/13/Conditional\\_Random\\_Fields](http://www.davidsbatista.net/blog/2017/11/13/Conditional_Random_Fields)



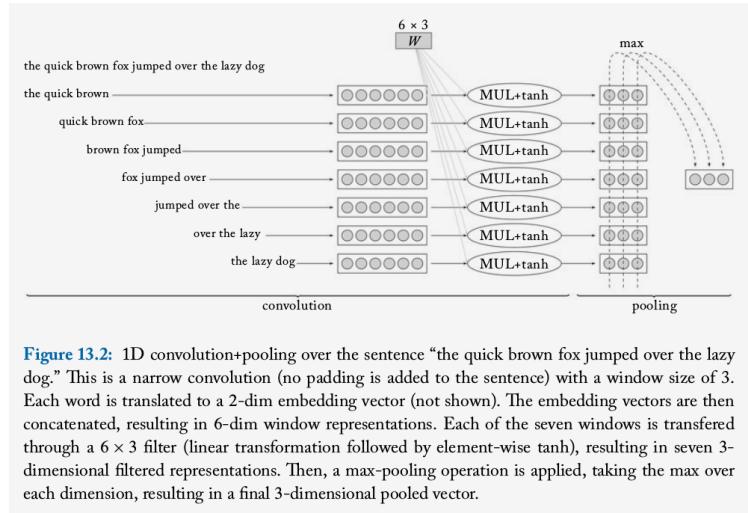
## 9. Redes Neuronales Convolucionales

### 9.1 Redes Neuronales Convolucionales (CNN) en Procesamiento del Lenguaje Natural (PLN)

Las redes neuronales convolucionales (CNN) se volvieron muy populares en la comunidad de visión por computadora debido a su éxito en la detección de objetos ("gato", "bicicletas") independientemente de su posición en la imagen. Estas redes identifican predictores locales indicativos en una estructura (por ejemplo, imágenes, oraciones) y los combinan para producir una representación vectorial de tamaño fijo para la estructura. En el procesamiento del lenguaje natural, la CNN captura los n-gramos que son más informativos para la tarea predictiva objetivo. Por ejemplo, en la clasificación de sentimientos, estos aspectos locales corresponden a n-gramos que transmiten sentimiento, como "no está mal." "muy bueno". La idea fundamental de las CNN [LeCun et al., 1998] es considerar la extracción de características y la clasificación como una tarea conjuntamente entrenada.

### 9.2 Convolución Básica + Agrupamiento

En el procesamiento del lenguaje natural, las oraciones suelen ser modeladas como secuencias de vectores de palabras. Estos vectores se pueden obtener a partir de incrustaciones de palabras pre-entrenadas o de una capa de incrustación. La CNN aplica funciones no lineales (aprendidas) o "filtros" que mapean ventanas de tamaño  $k$  de palabras a valores escalares. Se pueden aplicar varios filtros, lo que resulta en un vector de dimensión  $l$  (una dimensión por filtro). Los filtros capturan propiedades relevantes de las palabras en la ventana y corresponden a la "capa de convolución" de la red. La capa de "agrupamiento" se utiliza para combinar los vectores resultantes de las diferentes ventanas en un solo vector de dimensión  $l$ . Esto se logra tomando el valor máximo o el valor promedio observado en cada dimensión en las diferentes ventanas. El objetivo es capturar las características más importantes de la oración, independientemente de la posición. El vector resultante de dimensión  $l$  se alimenta luego a una red que se utiliza para la predicción (por ejemplo, softmax). Los gradientes se propagan desde la pérdida de la red ajustando los parámetros del filtro. Los filtros aprenden a resaltar los aspectos de los datos (n-gramos) que son importantes para la tarea objetivo.



**Figure 13.2:** 1D convolution+pooling over the sentence “the quick brown fox jumped over the lazy dog.” This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transferred through a  $6 \times 3$  filter (linear transformation followed by element-wise tanh), resulting in seven 3-dimensional filtered representations. Then, a max-pooling operation is applied, taking the max over each dimension, resulting in a final 3-dimensional pooled vector.

Figura 9.1: Arquitectura básica de una CNN para procesamiento del lenguaje natural.

<sup>1</sup>Fuente: [Goldberg, 2017]

### 9.3 Convoluciones 1D sobre Texto

En el contexto del procesamiento del lenguaje natural, nos centramos en la operación de convolución unidimensional<sup>2</sup>. Consideremos una secuencia de palabras  $w_{1:n} = w_1, \dots, w_n$ , cada una con su vector de palabras de  $d_{emb}$  dimensiones correspondiente  $E_{[w_i]} = \vec{w}_i$ . Una convolución 1D de ancho  $k$  funciona deslizando una ventana de tamaño  $k$  sobre la oración y aplicando el mismo filtro a cada ventana en la secuencia. Un filtro es un producto escalar con un vector de pesos  $\vec{u}$ , seguido a menudo de una función de activación no lineal. Definimos el operador  $\oplus(w_{i:i+k-1})$  como la concatenación de los vectores  $\vec{w}_i, \dots, \vec{w}_{i+k-1}$ . El vector concatenado de la ventana  $i$ -ésima es  $\vec{x}_i = \oplus(w_{i:i+k-1}) = [\vec{w}_i; \vec{w}_{i+1}; \dots; \vec{w}_{i+k-1}]$ , donde  $x_i \in \mathbb{R}^{k \cdot d_{emb}}$ . Luego, aplicamos el filtro a cada vector de ventana, lo que resulta en valores escalares  $p_i = g(\vec{x}_i \cdot \vec{u})$ , donde  $p_i \in \mathbb{R}$ . Es común usar  $l$  filtros diferentes  $\vec{u}_1, \dots, \vec{u}_l$ , que se pueden organizar en una matriz  $U$ , y a menudo se agrega un vector de sesgo  $\vec{b}$ :  $\vec{p}_i = g(\vec{x}_i \cdot U + \vec{b})$ . Cada vector  $\vec{p}_i$  es una colección de  $l$  valores que representan (o resumen) la  $i$ -ésima ventana ( $\vec{p}_i \in \mathbb{R}^l$ ). Idealmente, cada dimensión captura un tipo diferente de información indicativa. La idea principal detrás de la capa de convolución es aplicar la misma función parametrizada a todos los n-gramos en la secuencia, lo que crea una secuencia de  $m$  vectores, cada uno representando un n-gramo particular en la secuencia. La representación es sensible a la identidad y al orden de las palabras dentro del n-gramo, pero se extraerá la misma representación para un n-gramo independientemente de su posición en la secuencia.

### 9.4 Convoluciones Angostas vs. Amplias

¿Cuántos vectores  $\vec{p}_i$  tenemos? Para una oración de longitud  $n$  con una ventana de tamaño  $k$ , hay  $n - k + 1$  posiciones en las que se puede comenzar la secuencia. Obtenemos  $n - k + 1$  vectores  $\vec{p}_{1:n-k+1}$ . Este enfoque se conoce como **convolución angosta**. Una alternativa es llenar la oración

<sup>2</sup>1D se refiere a una convolución que se aplica a entradas unidimensionales como secuencias, a diferencia de las convoluciones 2D que se aplican a imágenes.

con  $k - 1$  palabras de relleno a cada lado, lo que resulta en  $n + k + 1$  vectores  $\vec{p}_{1:n+k+1}$ . Esto se llama **convolución amplia**.

## 9.5 Agrupamiento Vectorial

Aplicar la convolución sobre el texto da como resultado  $m$  vectores  $\vec{p}_{1:m}$ , cada uno de los cuales es un vector  $\vec{p}_i \in \mathbb{R}^l$ . Estos vectores se combinan (se agrupan) en un solo vector  $\vec{c} \in \mathbb{R}^l$  que representa toda la secuencia. Existen dos operaciones de agrupamiento comunes:

- Max pooling: este operador toma el valor máximo en cada dimensión (es la operación de agrupamiento más común). Para cada dimensión  $j$ , se calcula  $\vec{c}_{[j]} = \max_{1 < i \leq m} \vec{p}_{i[j]}$ , donde  $\vec{p}_{i[j]}$  denota la  $j$ -ésima componente de  $\vec{p}_i$ .
- Average pooling: este operador toma el valor promedio en cada índice. Se calcula  $\vec{c} = \frac{1}{m} \sum_{i=1}^m \vec{p}_i$ .

Idealmente, el vector  $\vec{c}$  capturará la esencia de la información importante en la secuencia. La naturaleza de la información importante que debe ser codificada en el vector  $\vec{c}$  depende de la tarea. Por ejemplo, si estamos realizando clasificación de sentimientos, la esencia son los n-gramos informativos que indican sentimiento. Durante el entrenamiento, el vector  $\vec{c}$  se alimenta a capas de red adicionales (por ejemplo, una capa de perceptrón multicapa), culminando en una capa de salida que se utiliza para la predicción. El procedimiento de entrenamiento de la red calcula la pérdida con respecto a la tarea de predicción, y los gradientes de error se propagan hacia atrás a través de las capas de agrupamiento y convolución, así como a través de las capas de incrustación. El proceso de entrenamiento ajusta la matriz de convolución  $U$ , el vector de sesgo  $\vec{b}$ , la red posterior y potencialmente también la matriz de incrustación  $E$ <sup>3</sup> de manera que el vector  $\vec{c}$  resultante del proceso de convolución y agrupamiento realmente codifique información relevante para la tarea en cuestión.

## 9.6 Clasificación de Sentimientos en Twitter con CNN

En [Severyn and Moschitti, 2015] se desarrolla una arquitectura de red neuronal convolucional para la clasificación de sentimientos en Twitter. Cada tweet se representa como una matriz cuyas columnas corresponden a las palabras en el tweet, preservando el orden en el que aparecen. Las palabras se representan mediante vectores densos o incrustaciones entrenadas a partir de un gran corpus de tweets no etiquetados utilizando word2vec. La red está formada por las siguientes capas: una capa de entrada con la matriz de tweets dada, una única capa de convolución, una función de activación lineal rectificada (ReLU), una capa de agrupamiento máximo (max pooling) y una capa de clasificación softmax. Los pesos de la red neuronal se pre-entrenan utilizando datos con anotaciones de emoticonos y luego se entrena con los tweets anotados a mano del concurso SemEval. Los resultados experimentales muestran que la fase de pre-entrenamiento permite una inicialización adecuada de los pesos de la red y, por lo tanto, tiene un impacto positivo en la precisión de la clasificación.

## 9.7 Redes Neuronales Convolucionales Muy Profundas para la Clasificación de Texto

Las arquitecturas de CNN para PLN son bastante superficiales en comparación con las redes neuronales convolucionales profundas que han impulsado el estado del arte en visión por compu-

---

<sup>3</sup>Algunas personas dejan fija la capa de incrustación durante el entrenamiento, mientras que otros permiten que los parámetros cambien.

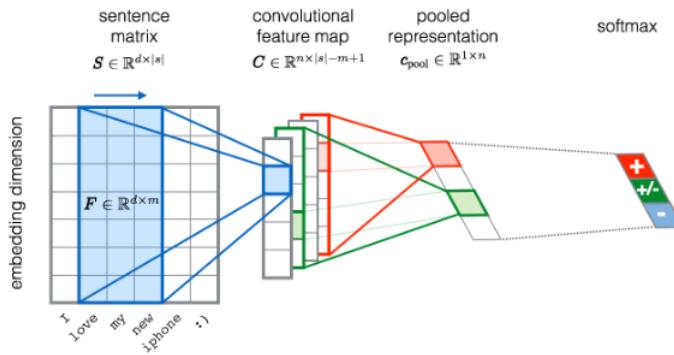


Figura 9.2: Arquitectura CNN para la clasificación de sentimientos en Twitter.

tadora. En [Conneau et al., 2017] se propone una arquitectura de red neuronal para el procesamiento de texto (VDCNN) que opera directamente a nivel de caracteres y utiliza solo convoluciones pequeñas y operaciones de agrupamiento. En lugar de utilizar incrustaciones de palabras, se utilizan incrustaciones de nivel de caracteres. Los caracteres son la representación atómica más baja del texto. El rendimiento de este modelo aumenta con la profundidad: utilizando hasta 29 capas de convolución, los autores informan mejoras sobre el estado del arte en varias tareas públicas de clasificación de texto. Las mejoras más notables se logran en conjuntos de datos grandes. Este fue uno de los primeros trabajos que mostró los beneficios de las arquitecturas neuronales profundas para PLN.

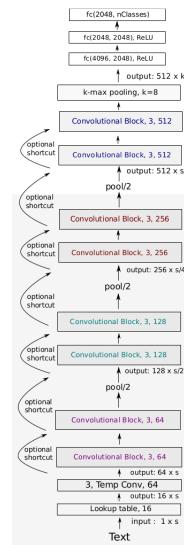


Figura 9.3: Arquitectura VDCNN para la clasificación de texto.



## 10. Redes Neuronales Recurrentes

### 10.1 La Abstracción de las RNN

Si bien las representaciones derivadas de las redes convolucionales ofrecen cierta sensibilidad al orden de las palabras, su sensibilidad al orden se limita principalmente a patrones locales y no tiene en cuenta el orden de patrones que están lejos en la secuencia. Las redes neuronales recurrentes (RNN) permiten representar entradas secuenciales de longitud arbitraria en vectores de tamaño fijo, prestando atención a las propiedades estructuradas de las entradas [Goldberg, 2016]. Las RNN, especialmente aquellas con arquitecturas con compuertas como LSTM y GRU, son muy eficaces para capturar regularidades estadísticas en entradas secuenciales.

Utilizamos  $\vec{x}_{i:j}$  para denotar la secuencia de vectores  $\vec{x}_i, \dots, \vec{x}_j$ . A alto nivel, la RNN es una función que toma como entrada una secuencia ordenada de longitud arbitraria de  $n$  vectores de dimensión  $d_{in}$ ,  $\vec{x}_{1:n} = \vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$  ( $\vec{x}_i \in \mathbb{R}^{d_{in}}$ ), y devuelve como salida un solo vector de dimensión  $d_{out}$ ,  $\vec{y}_n \in \mathbb{R}^{d_{out}}$ :

$$\begin{aligned} \vec{y}_n &= \text{RNN}(\vec{x}_{1:n}) \\ \vec{x}_i &\in \mathbb{R}^{d_{in}}, \quad \vec{y}_n \in \mathbb{R}^{d_{out}} \end{aligned} \tag{10.1}$$

Esto define implícitamente un vector de salida  $\vec{y}_i$  para cada prefijo  $\vec{x}_{1:i}$  de la secuencia  $\vec{x}_{i:n}$ . Denotamos por  $RNN^*$  a la función que devuelve esta secuencia:

$$\begin{aligned} \vec{y}_{1:n} &= RNN^*(\vec{x}_{1:n}) \\ \vec{y}_i &= \text{RNN}(\vec{x}_{1:i}) \\ \vec{x}_i &\in \mathbb{R}^{d_{in}}, \quad \vec{y}_n \in \mathbb{R}^{d_{out}} \end{aligned} \tag{10.2}$$

Luego, se utiliza el vector de salida  $\vec{y}_n$  para realizar predicciones adicionales. Por ejemplo, un modelo para predecir la probabilidad condicional de un evento  $e$  dado la secuencia  $\vec{x}_{1:n}$  se puede definir como el  $j$ -ésimo elemento del vector de salida resultante de la operación softmax sobre una transformación lineal de la codificación RNN:

$$p(e = j | \vec{x}_{1:n}) = \text{softmax}(\text{RNN}(\vec{x}_{1:n}) \cdot W + \vec{b})_{[j]}$$

La función RNN proporciona un marco para condicionar toda la historia sin recurrir a la suposición de Markov que se utiliza tradicionalmente para modelar secuencias. La RNN se define de forma recursiva mediante una función  $R$  que toma como entrada un vector de estado  $\vec{s}_{i-1}$  y un vector de entrada  $\vec{x}_i$ , y devuelve un nuevo vector de estado  $\vec{s}_i$ . Luego, el vector de estado  $\vec{s}_i$  se asigna a un vector de salida  $\vec{y}_i$  mediante una función determinista simple  $O(\cdot)$ . La base de la recursión es un vector de estado inicial  $\vec{s}_0$ , que también es una entrada de la RNN. Por brevedad, a menudo omitimos el vector inicial  $\vec{s}_0$  o asumimos que es el vector cero. Al construir una RNN, al igual que al construir una red de alimentación directa, es necesario especificar la dimensión de las entradas  $\vec{x}_i$ , así como las dimensiones de las salidas  $\vec{y}_i$ :

$$\begin{aligned} \text{RNN}^*(\vec{x}_{1:n}; \vec{s}_0) &= \vec{y}_{1:n} \\ \vec{y}_i &= O(\vec{s}_i) \\ \vec{s}_i &= R(\vec{s}_{i-1}, \vec{x}_i) \\ \vec{x}_i &\in \mathbb{R}^{d_{in}}, \quad \vec{y}_i \in \mathbb{R}^{d_{out}}, \quad \vec{s}_i \in \mathbb{R}^{f(d_{out})} \end{aligned} \tag{10.3}$$

Las funciones  $R$  y  $O$  son las mismas en todas las posiciones de la secuencia. La RNN realiza un seguimiento de los estados de la computación a través del vector de estado  $\vec{s}_i$  que se guarda y se pasa en las invocaciones de  $R$ .

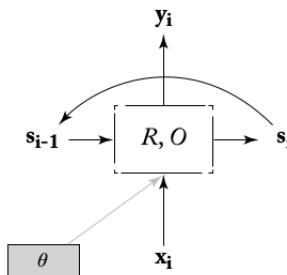


Figura 10.1: Representación gráfica de una RNN.

Esta presentación sigue la definición recursiva y es válida para secuencias de longitud arbitraria. Sin embargo, para una secuencia de entrada de tamaño finito (y todas las secuencias de entrada con las que trabajamos son finitas), se puede desenrollar la recursión.

Los parámetros  $\theta$  resaltan el hecho de que los mismos parámetros se comparten en todos los pasos de tiempo. Diferentes instancias de  $R$  y  $O$  darán como resultado estructuras de red diferentes. Observamos que el valor de  $\vec{s}_i$  (y, por lo tanto,  $\vec{y}_i$ ) se basa en toda la entrada  $\vec{x}_1, \dots, \vec{x}_i$ . Por ejemplo, al expandir la recursión para  $i = 4$  obtenemos:

Así,  $\vec{s}_n$  y  $\vec{y}_n$  pueden considerarse como la codificación de toda la secuencia de entrada. El objetivo del entrenamiento de la red es establecer los parámetros de  $R$  y  $O$  de manera que el estado transmita información útil para la tarea que estamos tratando de resolver.

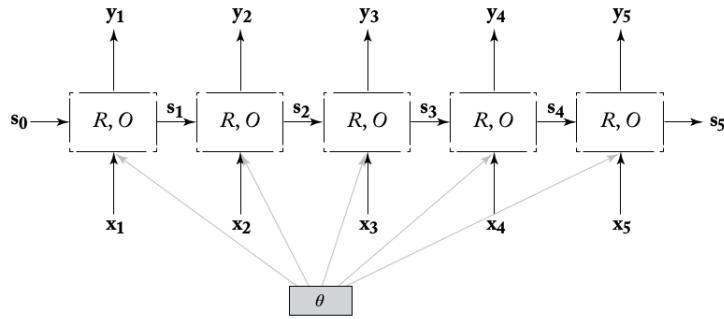


Figura 10.2: Representación gráfica de una RNN desenrollada para una secuencia de longitud finita.

$$\begin{aligned}
 s_4 &= R(s_3, x_4) \\
 &= R(\overbrace{s_3}^{s_3}, x_4) \\
 &= R(R(\overbrace{s_2}^{s_2}, x_3), x_4) \\
 &= R(R(R(\overbrace{s_1}^{s_1}, x_2), x_3), x_4).
 \end{aligned}$$

Figura 10.3: Representación gráfica de la RNN después de expandir la recursión.

## 10.2 Red Elman o Simple-RNN

Después de describir la abstracción de las RNN, ahora podemos discutir las instancias más simples de estas. Recordemos que estamos interesados en una función recursiva  $\vec{s}_i = R(\vec{x}_i, \vec{s}_{i-1})$  tal que  $\vec{s}_i$  codifique la secuencia  $\vec{x}_{1:n}$ . La formulación más sencilla de una RNN se conoce como Red Elman o Simple-RNN (S-RNN).

$$\begin{aligned}
 \vec{s}_i &= R_{SRNN}(\vec{x}_i, \vec{s}_{i-1}) = g(\vec{s}_{i-1}W^s + \vec{x}_iW^x + \vec{b}) \\
 \vec{y}_i &= O_{SRNN}(\vec{s}_i) = \vec{s}_i
 \end{aligned} \tag{10.4}$$

$\vec{s}_i, \vec{y}_i \in \mathbb{R}^{d_s}, \quad \vec{x}_i \in \mathbb{R}^{d_x}, \quad W^x \in \mathbb{R}^{d_x \times d_s}, \quad W^s \in \mathbb{R}^{d_s \times d_s}, \quad \vec{b} \in \mathbb{R}^{d_s}$

El estado  $\vec{s}_i$  y la entrada  $\vec{x}_i$  se transforman linealmente. Los resultados se suman (junto con un término de sesgo) y luego se pasan a través de una función de activación no lineal  $g$  (comúnmente tangente hiperbólica o ReLU). El Simple-RNN ofrece buenos resultados para etiquetado de secuencias y modelado del lenguaje.

## 10.3 Entrenamiento de las RNN

Una RNN desenrollada es simplemente una red neuronal muy profunda. Los mismos parámetros se comparten en muchas partes de la computación y se agrega una entrada adicional en varias capas. Para entrenar una red RNN, se necesita crear el grafo computacional desenrollado para una secuencia de entrada dada, agregar un nodo de pérdida al grafo desenrollado y luego utilizar el algoritmo de retropropagación para calcular los gradientes con respecto a esa pérdida. Este procedimiento se

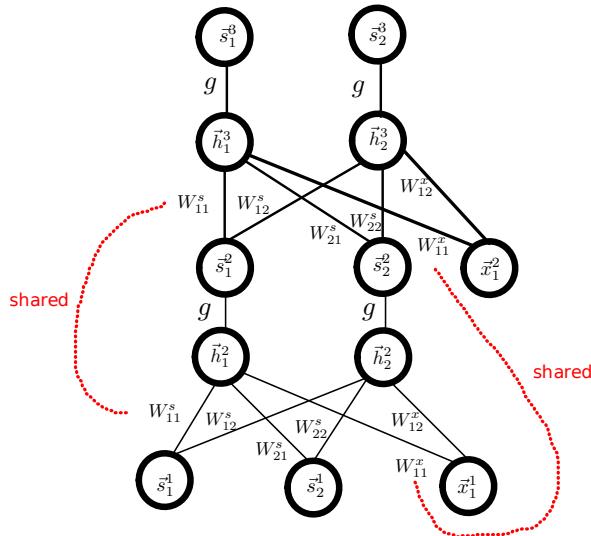


Figura 10.4: Red Elman o Simple-RNN.

conoce en la literatura de las RNN como retropropagación a través del tiempo (BPTT). La RNN por sí sola no hace mucho, sino que sirve como un componente entrenable en una red más grande. La predicción final y el cálculo de pérdida se realizan en esa red más grande y el error se retropropaga a través de la RNN. De esta manera, la RNN aprende a codificar propiedades de las secuencias de entrada que son útiles para la tarea de predicción. La señal de supervisión no se aplica directamente a la RNN, sino a través de la red más grande.

#### 10.4 Patrones de uso de las RNN: Aceptador

Un patrón común de uso de las RNN es el patrón del aceptador. Este patrón se utiliza para la clasificación de texto, donde la señal de supervisión se basa únicamente en el vector de salida final  $\vec{y}_n$ . El vector de salida de la RNN  $\vec{y}_n$  se alimenta a una capa completamente conectada o un MLP, que produce una predicción. Los gradientes de error luego se retropropagan a través del resto de la secuencia. La pérdida puede tener cualquier forma conocida, como entropía cruzada o hinge, etc.

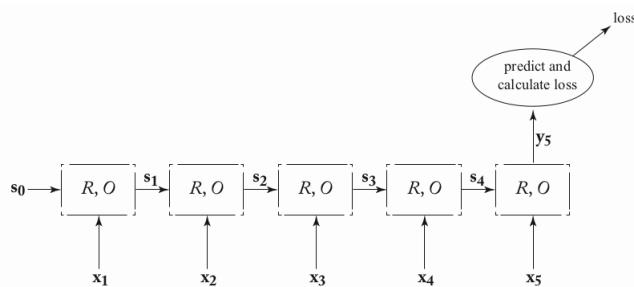


Figura 10.5: Gráfico de entrenamiento de una RNN del tipo aceptador.

Por ejemplo, se puede tener una RNN que lee los caracteres de una palabra uno por uno y luego utiliza el estado final para predecir la categoría gramatical de esa palabra. Otro ejemplo es una RNN

que lee una oración  $y$ , en función del estado final, decide si transmite un sentimiento positivo o negativo.

## 10.5 Patrones de uso de las RNN: Transductor

Otra opción es tratar la RNN como un transductor, produciendo una salida  $\hat{t}_i$  para cada entrada que lee. Este patrón es muy útil para tareas de etiquetado de secuencias (por ejemplo, etiquetado de partes del discurso, reconocimiento de entidades nombradas, etc.). Se calcula una señal de pérdida local (por ejemplo, entropía cruzada)  $L_{\text{local}}(\hat{t}_i, t_i)$  para cada una de las salidas  $\hat{t}_i$  basada en una etiqueta verdadera  $t_i$ . La pérdida para la secuencia desenrollada será entonces:  $L(\hat{t}_{1:n}, t_{1:n}) = \sum_i L_{\text{local}}(\hat{t}_i, t_i)$ , o utilizando otra combinación en lugar de la suma, como el promedio o un promedio ponderado.

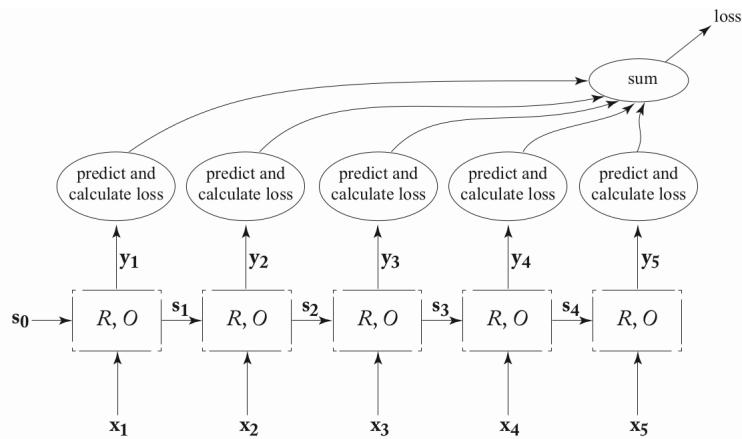


Figura 10.6: Gráfico de entrenamiento de una RNN del tipo transductor.

Por ejemplo, un etiquetador de secuencias (por ejemplo, NER, POS), en el que  $\vec{x}_{i:n}$  representa las representaciones de características de las  $n$  palabras de una oración, y  $t_i$  se utiliza para predecir la asignación de etiquetas de la palabra  $i$  en función de las palabras  $1 : i$ . Otro ejemplo es el modelado de lenguaje, en el que la secuencia de palabras  $x_{1:n}$  se utiliza para predecir una distribución sobre la palabra  $(i + 1)$ -ésima. Los modelos de lenguaje basados en RNN han demostrado tener una perplejidad mucho mejor que los modelos de lenguaje tradicionales.

El uso de las RNN como transductores nos permite relajar la suposición de Markov que se hace tradicionalmente en los modelos de lenguaje y los etiquetadores HMM, y condicionar en todo el historial de predicciones.

## 10.6 Redes neuronales recurrentes bidireccionales (BiRNN)

Una elaboración útil de una RNN es una red neuronal recurrente bidireccional (también conocida como BiRNN). Consideremos la tarea de etiquetado de secuencias en una oración. Una RNN nos permite calcular una función de la  $i$ -ésima palabra  $x_i$  en función de las palabras anteriores  $x_{1:i}$ , incluyendo la palabra actual. Sin embargo, las palabras siguientes  $x_{i+1:n}$  también pueden ser útiles para la predicción. La BiRNN nos permite mirar arbitrariamente lejos tanto al pasado como al futuro dentro de la secuencia. Consideremos una secuencia de entrada  $\vec{x}_{1:n}$ . La BiRNN funciona manteniendo dos estados separados,  $s_i^f$  y  $s_i^b$  para cada posición de entrada  $i$ . El estado hacia adelante

$s_i^f$  se basa en  $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_i$ , mientras que el estado hacia atrás  $s_i^b$  se basa en  $\vec{x}_n, \vec{x}_{n-1}, \dots, \vec{x}_i$ . Los estados hacia adelante y hacia atrás se generan mediante dos RNN diferentes. La primera RNN ( $R^f, O^f$ ) recibe la secuencia de entrada  $\vec{x}_{1:n}$  tal como está, mientras que la segunda RNN ( $R^b, O^b$ ) recibe la secuencia de entrada en orden inverso. La representación del estado  $\vec{s}_i$  está compuesta tanto por los estados hacia adelante como por los estados hacia atrás. La salida en la posición  $i$  se basa en la concatenación de los dos vectores de salida:

$$\vec{y}_i = [\vec{y}_i^f; \vec{y}_i^b] = [O^f(s_i^f); O^b(s_i^b)]$$

La salida tiene en cuenta tanto el pasado como el futuro. La codificación biRNN de la palabra  $i$  en una secuencia es la concatenación de dos RNN, una que lee la secuencia desde el principio y otra que la lee desde el final. Definimos  $biRNN(\vec{x}_{1:n}, i)$  como el vector de salida correspondiente a la  $i$ -ésima posición de la secuencia:

$$biRNN(\vec{x}_{1:n}, i) = \vec{y}_i = [RNN^f(\vec{x}_{1:i}); RNN^b(\vec{x}_{n:i})]$$

El vector  $\vec{y}_i$  se puede utilizar directamente para la predicción o se puede alimentar como parte de la entrada a una red más compleja. Mientras que las dos RNN se ejecutan de forma independiente, los gradientes de error en la posición  $i$  fluirán tanto hacia adelante como hacia atrás a través de las dos RNN. Al alimentar el vector  $\vec{y}_i$  a través de un MLP antes de la predicción, se mezclarán aún más las señales hacia adelante y hacia atrás.

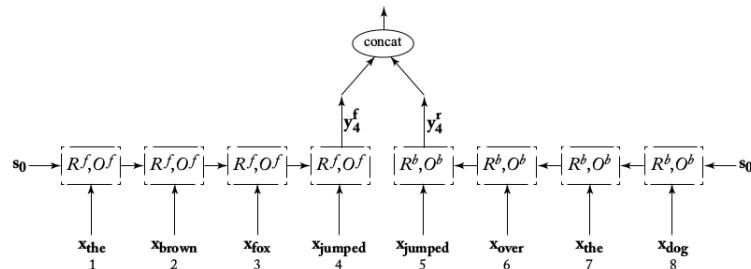


Figura 10.7: Red neuronal recurrente bidireccional (biRNN).

Observa cómo el vector  $\vec{y}_4$ , correspondiente a la palabra **saltó**, codifica una ventana infinita alrededor (e incluyendo) el vector de enfoque  $\vec{x}_{\text{saltó}}$ . La biRNN es muy efectiva para tareas de etiquetado, en las que cada vector de entrada corresponde a un vector de salida. También es útil como componente de extracción de características entrenable de propósito general, que se puede utilizar siempre que se requiera una ventana alrededor de una palabra determinada.

## 10.7 Redes neuronales recurrentes multi-capas (apiladas)

Las RNN se pueden apilar en capas, formando una estructura en forma de cuadrícula. Consideremos  $k$  RNN,  $RNN_1, \dots, RNN_k$ , donde la  $j$ -ésima RNN tiene estados  $\vec{s}_{1:n}^j$  y salidas  $\vec{y}_{1:n}^j$ . La entrada para la primera RNN son  $\vec{x}_{1:n}$ . La entrada de la  $j$ -ésima RNN ( $j \geq 2$ ) son las salidas de la RNN debajo de ella,  $\vec{y}_{1:n}^{j-1}$ . La salida de toda la formación es la salida de la última RNN,  $\vec{y}_{1:n}^k$ . Estas arquitecturas en capas se conocen comúnmente como RNN profundas. No

existe una restricción en la cantidad de capas que se pueden apilar, pero es importante tener en cuenta que agregar más capas puede aumentar la complejidad del modelo y requerir más datos y tiempo de entrenamiento.

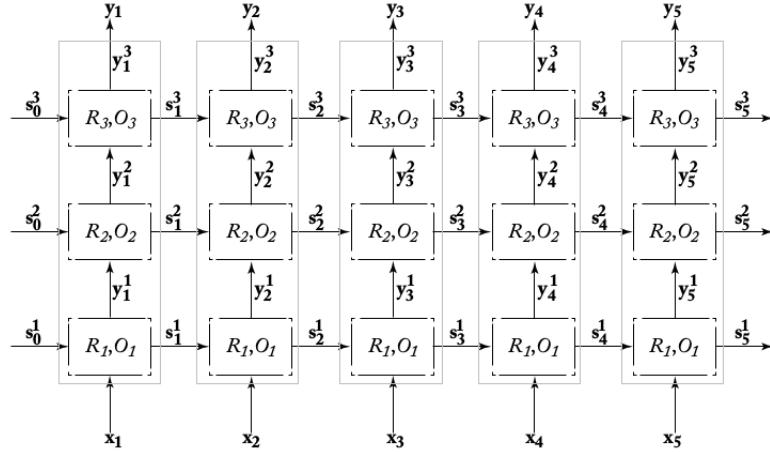


Figura 10.8: Redes neuronales recurrentes multi-capas (apiladas).

La ventaja de las RNN apiladas es que cada capa puede aprender representaciones de características de nivel superior en la secuencia de entrada. Las capas superiores pueden capturar patrones de más largo alcance y dependencias más complejas. Cada capa procesa la secuencia de entrada de manera incremental, y las capas superiores reciben información de las capas inferiores para ayudar a hacer predicciones más precisas. Las RNN apiladas son particularmente efectivas para tareas de modelado del lenguaje y traducción automática.

## 10.8 Arquitecturas con compuertas

Hasta ahora hemos visto solo una instancia de la RNN: la RNN simple (S-RNN) o red Elman. Este modelo es difícil de entrenar efectivamente debido al problema de los **gradientes que desaparecen**. Las señales de error (gradientes) en pasos posteriores de la secuencia disminuyen rápidamente en el proceso de retropropagación, por lo que no llegan a las señales de entrada anteriores, lo que dificulta que la S-RNN capture dependencias a largo plazo. Las arquitecturas basadas en compuertas, como LSTM [Hochreiter and Schmidhuber, 1997] y GRU [Cho et al., 2014], están diseñadas para solucionar esta deficiencia.

Intuitivamente, las redes neuronales recurrentes pueden considerarse como redes de alimentación profunda con parámetros compartidos entre diferentes capas. En el caso de la RNN simple, los gradientes incluyen multiplicaciones repetidas de la matriz  $W$ , lo que hace muy probable que los valores desaparezcan o exploten. El mecanismo de las compuertas mitiga este problema en gran medida al eliminar esta multiplicación repetida de una sola matriz.

Podemos considerar la RNN como un dispositivo de computación de propósito general, donde el estado  $\vec{s}_i$  representa una memoria finita. Cada aplicación de la función  $R$  lee una entrada  $\vec{x}_{i+1}$ , lee la memoria actual  $\vec{s}_i$ , opera en ella de alguna manera y escribe el resultado en la memoria. Esto resulta en un nuevo estado de memoria  $\vec{s}_{i+1}$ . Sin embargo, un problema aparente con la arquitectura S-RNN es que el acceso a la memoria no está controlado. En cada paso de la computación, se lee todo el

estado de memoria y se escribe todo el estado de memoria. ¿Cómo se puede proporcionar un acceso a la memoria más controlado?

Podemos utilizar un vector binario  $\vec{g} \in [0, 1]^n$  que actúa como una **compuerta** para controlar el acceso a vectores de dimensión  $n$  utilizando la operación producto de Hadamard  $\vec{x} \odot \vec{g}$ . La operación Hadamard es la multiplicación elemento a elemento de dos vectores:

$$\vec{x} = \vec{u} \odot \vec{v} \Leftrightarrow \vec{x}_{[i]} = \vec{u}_{[i]} \cdot \vec{v}_{[i]} \quad \forall i \in [1, n]$$

Si consideramos una memoria  $\vec{s} \in \mathbb{R}^d$ , una entrada  $\vec{x} \in \mathbb{R}^d$  y una compuerta  $\vec{g} \in [0, 1]^d$ , la siguiente computación:

$$\vec{s}' \leftarrow \vec{g} \odot \vec{x} + (\vec{1} - \vec{g}) \odot (\vec{s})$$

lee las entradas en  $\vec{x}$  que corresponden a los valores  $\vec{1}$  en  $\vec{g}$  y las escribe en la nueva memoria  $\vec{s}'$ . Las ubicaciones que no se leyeron se copian de la memoria  $\vec{s}$  a la nueva memoria  $\vec{s}'$  mediante el uso de la compuerta  $(\vec{1} - \vec{g})$ .

Este mecanismo de compuerta puede servir como un bloque de construcción en nuestra RNN. Los vectores de compuerta se pueden utilizar para controlar el acceso al estado de memoria  $\vec{s}_i$ . Sin embargo, todavía nos faltan dos componentes importantes (y relacionados): 1) las compuertas no deben ser estáticas, sino que deben estar controladas por el estado de memoria actual y la entrada, y 2) su comportamiento debe ser aprendido.

Esto presenta un obstáculo, ya que el aprendizaje en nuestro marco implica ser diferenciable (debido al algoritmo de retropropagación). Los valores binarios de 0-1 utilizados en las compuertas no son diferenciables. Una solución a este problema es aproximar el mecanismo de compuerta dura con un mecanismo de compuerta suave, pero diferenciable.

Para lograr compuertas diferenciables, reemplazamos el requisito de que  $\vec{g} \in [0, 1]^n$  y permitimos números reales arbitrarios  $\vec{g}' \in \mathbb{R}^n$ . Estos números reales luego pasan a través de una función sigmoide  $\sigma(\vec{g}')$ , lo que limita los valores en el rango  $(0, 1)$ , con la mayoría de los valores cerca de los bordes. Al utilizar la compuerta  $\sigma(\vec{g}') \odot \vec{x}$ , los índices en  $\vec{x}$  que corresponden a valores cercanos a uno en  $\sigma(\vec{g}')$  pueden pasar, mientras que los que corresponden a valores cercanos a cero son bloqueados. Los valores de las compuertas pueden condicionarse a la entrada y al estado de memoria actual, y pueden entrenarse utilizando un método basado en gradientes para realizar un comportamiento deseado.

Este mecanismo de compuerta controlable es la base de las arquitecturas LSTM y GRU. En cada paso de tiempo, los mecanismos de compuerta diferenciables deciden qué partes de las entradas se escribirán en la memoria y qué partes de la memoria se sobreescriturán (olvidarán).

$$\vec{s}_j = R_{\text{LSTM}}(\vec{s}_{j-1}, \vec{x}_j) = [\vec{c}_j; \vec{h}_j]$$

$$\vec{c}_j = \vec{f} \odot \vec{c}_{j-1} + \vec{i} \odot \vec{z}$$

$$\vec{h}_j = \vec{o} \odot \tanh(\vec{c}_j)$$

$$\vec{i} = \sigma(\vec{x}_j \mathbf{W}^{xi} + \vec{h}_{j-1} \mathbf{W}^{hi})$$

$$\vec{f} = \sigma(\vec{x}_j \mathbf{W}^{xf} + \vec{h}_{j-1} \mathbf{W}^{hf})$$

$$\vec{o} = \sigma(\vec{x}_j \mathbf{W}^{xo} + \vec{h}_{j-1} \mathbf{W}^{ho})$$

$$\vec{z} = \tanh(\vec{x}_j \mathbf{W}^{xz} + \vec{h}_{j-1} \mathbf{W}^{hz})$$

$$\vec{y}_j = O_{\text{LSTM}}(\vec{s}_j) = \vec{h}_j$$

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j; h_j]$$

$$c_j = f \odot c_{j-1} + i \odot z$$

$$h_j = o \odot \tanh(c_j)$$

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \quad x_i \in \mathbb{R}^{d_x}, \quad c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \quad W^{xo} \in \mathbb{R}^{d_x \times d_h}, \quad W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

Figura 10.9: Formulación matemática de la arquitectura LSTM.

El estado en el tiempo  $j$  se compone de dos vectores,  $\vec{c}_j$  y  $\vec{h}_j$ , donde  $\vec{c}_j$  es el componente de memoria y  $\vec{h}_j$  es el componente de estado oculto. Hay tres compuertas,  $\vec{i}$ ,  $\vec{f}$  y  $\vec{o}$ , que controlan la **entrada** (input), **olvido** (forget) y **salida** (output), respectivamente. Los valores de las compuertas se calculan en función de las combinaciones lineales de la entrada actual  $\vec{x}_j$  y el estado anterior  $\vec{h}_{j-1}$ , pasados a través de una función de activación sigmoide. Se calcula un candidato de actualización  $\vec{z}$  como una combinación lineal de  $\vec{x}_j$  y  $\vec{h}_{j-1}$ , pasados a través de una función de activación tangente hiperbólica ( $\tanh$ ) (para llevar los valores al rango de -1 a 1). Luego, se actualiza la memoria  $\vec{c}_j$ : la compuerta de olvido controla cuánta de la memoria anterior se debe mantener ( $\vec{f} \odot \vec{c}_{j-1}$ ), y la compuerta de entrada controla cuánta de la actualización propuesta se debe mantener ( $\vec{i} \odot \vec{z}$ ). Finalmente, el valor de  $\vec{h}_j$  (que también es la salida  $\vec{y}_j$ ) se determina en función del contenido de la memoria  $\vec{c}_j$ , pasado a través de una no linealidad tangente hiperbólica y controlada por la compuerta de salida. Los mecanismos de compuertas permiten que los gradientes relacionados con la parte de memoria  $\vec{c}_j$  se mantengan altos en rangos de tiempo muy largos.

<sup>0</sup>Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

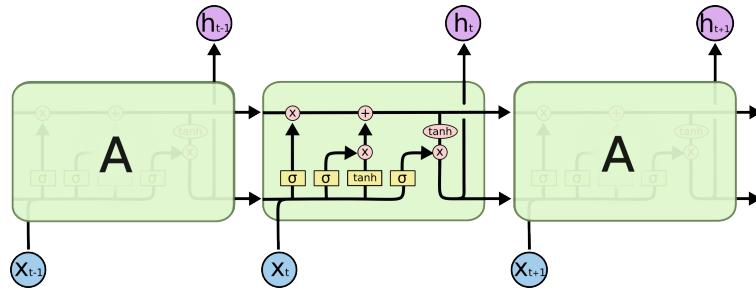


Figura 10.10: Cadena desenrollada de una arquitectura LSTM.

Intuitivamente, las redes neuronales recurrentes pueden considerarse como redes de alimentación profunda con parámetros compartidos entre diferentes capas. En el caso de la RNN simple, los gradientes incluyen multiplicaciones repetidas de la matriz  $W$ , lo que hace que los valores de gradiente desaparezcan o exploten. El mecanismo de las compuertas mitiga este problema en gran medida al eliminar esta multiplicación repetida de una sola matriz. Las LSTMs son actualmente el tipo de arquitectura RNN más exitoso y son responsables de muchos resultados de modelado de secuencias de vanguardia. El principal competidor de la RNN LSTM es la GRU, que se discutirá a continuación.

## 10.9 GRU

La arquitectura LSTM es muy efectiva, pero también bastante complicada. La complejidad del sistema dificulta el análisis y también es computacionalmente costosa de trabajar. El Gated Recurrent Unit (GRU) se introdujo en [Cho et al., 2014] como una alternativa a LSTM. Posteriormente, se demostró en [Chung et al., 2014] que tiene un rendimiento comparable a LSTM en varios conjuntos de datos (no textuales). Al igual que LSTM, GRU también se basa en un mecanismo de compuerta, pero con sustancialmente menos compuertas y sin un componente de memoria separado.

$$\begin{aligned}s_j &= R_{\text{GRU}}(s_{j-1}, x_j) = (1 - z) \odot s_{j-1} + z \odot \tilde{s}_j \\z &= \sigma(x_j W^{xz} + s_{j-1} W^{sz}) \\r &= \sigma(x_j W^{xr} + s_{j-1} W^{sr}) \\ \tilde{s}_j &= \tanh(x_j W^{xs} + (r \odot s_{j-1}) W^{sg})\end{aligned}$$

$$y_j = O_{\text{GRU}}(s_j) = s_j$$

$$s_j, \tilde{s}_j \in \mathbb{R}^{d_s}, \quad x_i \in \mathbb{R}^{d_x}, \quad z, r \in \mathbb{R}^{d_s}, \quad W^{xo} \in \mathbb{R}^{d_x \times d_s}, \quad W^{so} \in \mathbb{R}^{d_s \times d_s}.$$

Figura 10.11: Arquitectura GRU.

Se utiliza una sola compuerta  $\tilde{r}$  para controlar el acceso al estado anterior  $\tilde{s}_{j-1}$  y calcular una actualización propuesta  $\tilde{s}_j$ . El estado actualizado  $s_j$  (que también sirve como salida  $y_j$ ) se determina luego en función de una interpolación entre el estado anterior  $\tilde{s}_{j-1}$  y la propuesta  $\tilde{s}_j$ . Las proporciones de la interpolación se controlan mediante la compuerta  $z$ . Se ha demostrado que GRU es efectivo

en el modelado del lenguaje y la traducción automática. Sin embargo, aún se está investigando y comparando la GRU, la LSTM y posibles arquitecturas de RNN alternativas. Para una exploración empírica de las arquitecturas GRU y LSTM, consulte [Jozefowicz et al., 2015].

## 10.10 Clasificación de sentimientos con RNN

El uso más simple de las RNN es como aceptores: leer una secuencia de entrada y producir una respuesta binaria o multi-clase al final. Las RNN son aprendices de secuencia muy poderosos y pueden captar patrones muy intrincados en los datos. Un ejemplo de oraciones naturalmente positivas y negativas en el dominio de las críticas de películas sería el siguiente: Positiva: "No es afirmativa de la vida, es vulgar y cruel, pero me gustó". Negativa: "Es decepcionante que solo logre ser decente en lugar de brillante".

Observa que el ejemplo positivo contiene algunas frases negativas ("no afirmativa de la vida", "vulgar y cruel"). Mientras que el ejemplo negativo contiene algunas frases positivas ("brillante"). Predecir correctamente el sentimiento requiere entender no solo las frases individuales sino también el contexto en el que ocurren, construcciones lingüísticas como la negación y la estructura general de la oración. La tarea de clasificación de sentimientos a nivel de oración se modela utilizando un RNN-aceptor. Después de la tokenización, la RNN lee las palabras de la oración una a la vez. Luego, el estado final de la RNN se alimenta a una MLP seguida de una capa softmax con dos salidas (positiva y negativa). La red se entrena con pérdida de entropía cruzada basada en las etiquetas de sentimiento correctas.

$$\begin{aligned} p(\text{etiqueta} = k | \vec{w}_{1:n}) &= \hat{\vec{y}}_{[k]} \\ \hat{\vec{y}} &= \text{softmax}(\text{MLP}(\text{RNN}(\vec{x}_{1:n}))) \\ \vec{x}_{1:n} &= E_{[w_1]}, \dots, E_{[w_n]} \end{aligned} \tag{10.5}$$

La matriz de incrustación de palabras  $E$  se inicializa utilizando incrustaciones pre-entrenadas aprendidas sobre un corpus externo grande utilizando un algoritmo como Word2vec o GloVe con una ventana relativamente amplia. A menudo es útil extender el modelo considerando RNN bidireccionales. Para oraciones más largas, [Li et al., 2015] encontró útil utilizar una arquitectura jerárquica, en la cual la oración se divide en segmentos más pequeños basados en la puntuación. Luego, cada segmento se alimenta a una RNN bidireccional. La secuencia de vectores resultantes (uno para cada segmento) se alimenta luego a un RNN aceptor. Se utilizó una arquitectura jerárquica similar para la clasificación de sentimientos a nivel de documento en [Tang et al., 2015].

## 10.11 Clasificación de sentimientos en Twitter con LSTMS y Emojis

Se propuso un modelo de supervisión remota basado en emojis para detectar sentimientos y otros estados afectivos en mensajes cortos de redes sociales en [Felbo et al., 2017]. Se utilizan emojis como enfoque de supervisión remota para diversas tareas de detección afectiva (por ejemplo, emoción, sentimiento, sarcasmo) utilizando un corpus grande de 634 millones de tweets con 64 emojis. Se pre-entrena una arquitectura de red neuronal con este corpus. La red es una variante de LSTM formada por una capa de incrustación, 2 capas LSTM bidireccionales con conexiones de omisión normales y conexiones de agrupamiento promedio temporal.

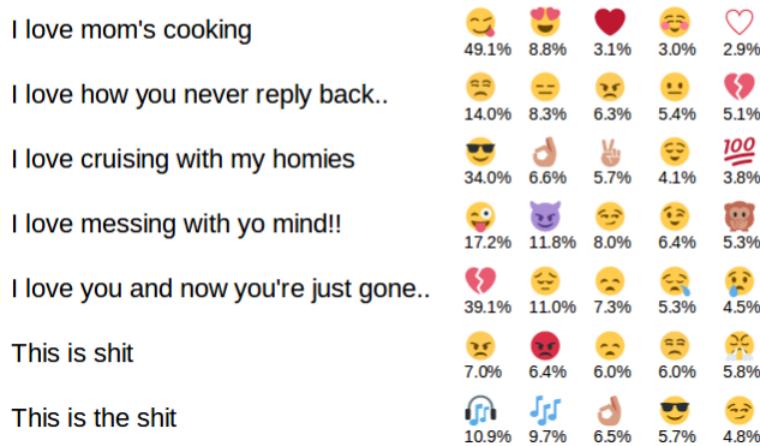


Figura 10.12: Arquitectura de red neuronal para la clasificación de sentimientos en Twitter con LSTMS y emojis (parte 1).

Los autores proponen el enfoque de transferencia de deshielo encadenado, en el cual la red pre-entrenada se ajusta finamente para la tarea objetivo. Aquí, cada capa se ajusta finamente individualmente en cada paso con los datos objetivos correctos, y luego todas se ajustan finamente juntas. El modelo logra resultados de vanguardia en la detección de emociones, sentimientos y sarcasmo. La red pre-entrenada se ha publicado para el público en

## 10.12 Bi-LSTM CRF

Un enfoque alternativo al transductor para el etiquetado de secuencias es combinar una BI-LSTM con un Campo Aleatorio Condicional (CRF). Esto produce un etiquetador poderoso [Huang et al., 2015] llamado BI-LSTM-CRF, en el cual la LSTM actúa como extractor de características y el CRF como una capa especial que modela las transiciones de una etiqueta a otra.

La capa CRF realiza una normalización global sobre todas las posibles secuencias, lo que ayuda a encontrar la secuencia óptima. En un CRF, modelamos la probabilidad condicional de la secuencia de etiquetas de salida dada la secuencia de entrada:

$$P(s_1, \dots, s_m | x_1, \dots, x_m) = P(s_{1:m} | x_{1:m})$$

Lo hacemos definiendo un mapa de características  $\vec{\Phi}(x_{1:m}, s_{1:m}) \in \mathcal{R}^d$  que mapea una secuencia de entrada completa  $x_{1:m}$  junto con una secuencia de etiquetas completa  $s_{1:m}$  a un vector de características de  $d$  dimensiones.

Luego, podemos modelar la probabilidad como un modelo log-lineal con el vector de parámetros  $\vec{w} \in \mathcal{R}^d$ :

$$P(s_{1:m} | x_{1:m}; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))}$$

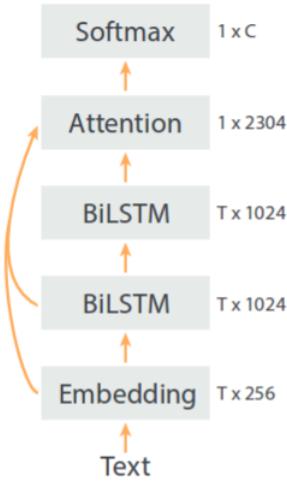


Figura 10.13: Arquitectura de red neuronal para la clasificación de sentimientos en Twitter con LSTMS y emojis (parte 2).

donde  $s'_{1:m}$  varía sobre todas las posibles secuencias de salida.

Podemos ver la expresión  $\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}) = \text{score}_{crf}(x_{1:m}, s_{1:m})$  como una puntuación de qué tan bien se ajusta la secuencia de etiquetas a la secuencia de entrada dada.

Recordemos de la clase de CRF que  $\vec{\Phi}(x_{1:m}, s_{1:m})$  se creó con características diseñadas manualmente.

La idea del LSTM CRF es reemplazar  $\vec{\Phi}(x_{1:m}, s_{1:m})$  con la salida de un transductor LSTM (características aprendidas automáticamente):

$$\text{score}_{lstm-crif}(x_{1:m}, s_{1:m}) = \sum_{i=0}^m W_{[s_{i-1}, s_i]} \cdot \text{LSTM}^*(x_{1:m})_{[i]} + \vec{b}[s_{i-1}, s_i]$$

donde  $W_{[s_{i-1}, s_i]}$  corresponde a una puntuación de transición de la etiqueta  $s_{i-1}$  a  $s_i$ ,  $\vec{b}[s_{i-1}, s_i]$  es un término de sesgo para la transición y  $\text{LSTM}^*(x_{1:m})_{[i]}$  proviene del estado oculto de la Bi-LSTM en el paso de tiempo  $i$ .

Todos estos parámetros se aprenden conjuntamente. Observa que la matriz de transición  $W$  no depende de la posición. Se utiliza el algoritmo de avance-retroceso durante el entrenamiento y el algoritmo de Viterbi durante la decodificación. Para obtener más información, consulta <sup>1</sup> y <sup>2</sup>.

<sup>1</sup>[https://pytorch.org/tutorials/beginner/nlp/advanced\\_tutorial.html](https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html)

<sup>2</sup><https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/>

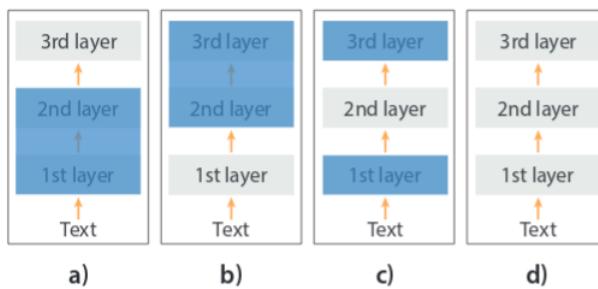


Figure 2: Illustration of the chain-thaw transfer learning approach, where each layer is fine-tuned separately. Layers covered with a blue rectangle are frozen. Step a) tunes any new layers, b) then tunes the 1st layer and c) the next layer until all layers have been fine-tuned individually. Lastly, in step d) all layers are fine-tuned together.

Figura 10.14: Demostración del modelo: <https://deepmoji.mit.edu/>.



## 11. Modelos Secuencia a Secuencia

### 11.1 Modelos de lenguaje y generación de lenguaje

- La modelización del lenguaje es la tarea de asignar una probabilidad a las oraciones en un lenguaje.
- Por ejemplo, ¿cuál es la probabilidad de ver la oración *.el perro perezoso ladró ruidosamente*?
- La tarea se puede formular como la tarea de predecir la probabilidad de ver una palabra condicionada a las palabras anteriores:

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

- Las RNN se pueden utilizar para entrenar modelos de lenguaje vinculando la salida en el tiempo  $i$  con su entrada en el tiempo  $i + 1$ .
- Esta red se puede utilizar para generar secuencias de palabras o frases aleatorias.
- Proceso de generación: predecir una distribución de probabilidad sobre la primera palabra condicionada al símbolo de inicio y seleccionar una palabra aleatoria de acuerdo con la distribución predicha.
- Luego, predecir una distribución de probabilidad sobre la segunda palabra condicionada a la primera, y así sucesivamente, hasta predecir el símbolo de fin de secuencia  $</s>$ .
- Después de predecir una distribución sobre los siguientes símbolos de salida  $P(t_i = k|t_{1:i-1})$ , se elige un token  $t_i$  y su vector de incrustación correspondiente se alimenta como entrada al siguiente paso.
- Teacher-forcing: durante el **entrenamiento**, se alimenta al generador con la palabra anterior verdadera, incluso si su propia predicción le asignó una pequeña probabilidad.
- Es probable que el generador haya generado una palabra diferente en este estado durante la **prueba**.

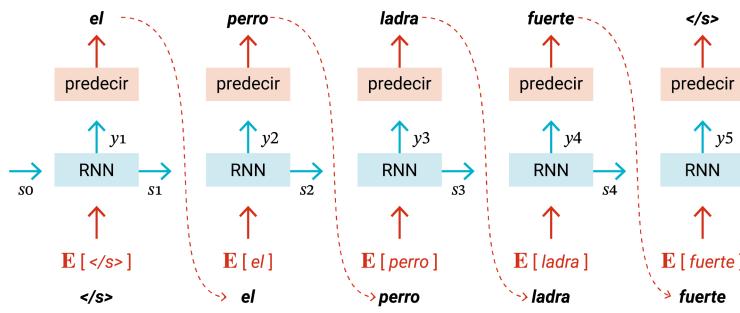


Figura 11.1: Arquitectura de generación de lenguaje con una RNN.

## 11.2 Problemas de secuencia a secuencia

Casi cualquier tarea en NLP se puede formular como un problema de secuencia a secuencia (o generación condicionada), es decir, generar secuencias de salida a partir de secuencias de entrada. Las secuencias de entrada y salida pueden tener longitudes diferentes.

- Traducción automática: de un idioma fuente a un idioma objetivo.
- Resumen: de texto largo a texto corto.
- Diálogo (chatbots): de las intervenciones anteriores a la siguiente intervención.

### 11.2.1 Generación condicionada

- Si bien utilizar la RNN como generador es un ejercicio interesante para demostrar su fortaleza, el verdadero poder del generador de RNN se revela cuando se pasa a una generación condicionada o un marco de codificador-decodificador.
- Idea central: usar dos RNN.
- Codificador: se utiliza una RNN para codificar la entrada fuente en un vector  $\vec{c}$ .
- Decodificador: se utiliza otra RNN para decodificar la salida del codificador y generar la salida objetivo.
- En cada etapa del proceso de generación, el vector de contexto  $\vec{c}$  se concatena a la entrada  $\hat{t}_j$  y la concatenación se alimenta a la RNN. Marco codificador-decodificador
- Esta configuración es útil para mapear secuencias de longitud  $n$  a secuencias de longitud  $m$ .
- El codificador resume la oración fuente en un vector  $\vec{c}$ .
- Luego, la RNN del decodificador se utiliza para predecir (utilizando un objetivo de modelado del lenguaje) las palabras de la secuencia objetivo condicionadas tanto a las palabras predichas anteriormente como a la oración codificada  $\vec{c}$ .
- Las RNN del codificador y del decodificador se entran conjuntamente.
- La supervisión solo ocurre para la RNN del decodificador, pero los gradientes se propagan hasta la RNN del codificador.

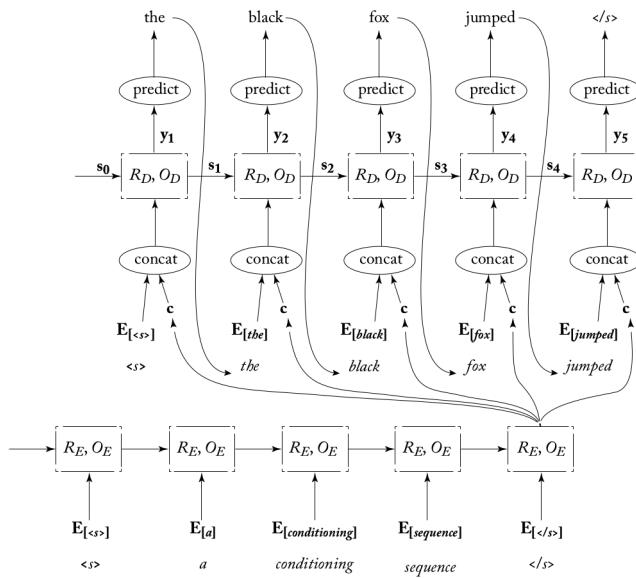
#### Gráfico de entrenamiento de secuencia a secuencia

#### Traducción automática neuronal

#### Progreso de BLEU en traducción automática

[Edinburgh En-De WMT]

<sup>0</sup>fuent: [http://www.meta-net.eu/events/meta-forum-2016/slides/09\\_sennrich.pdf](http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf)



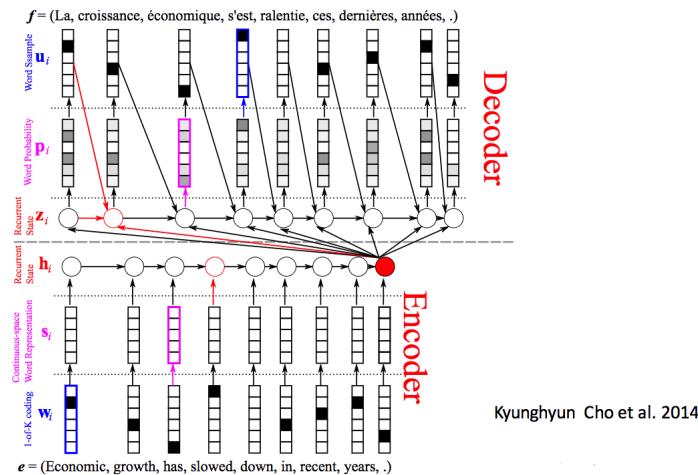
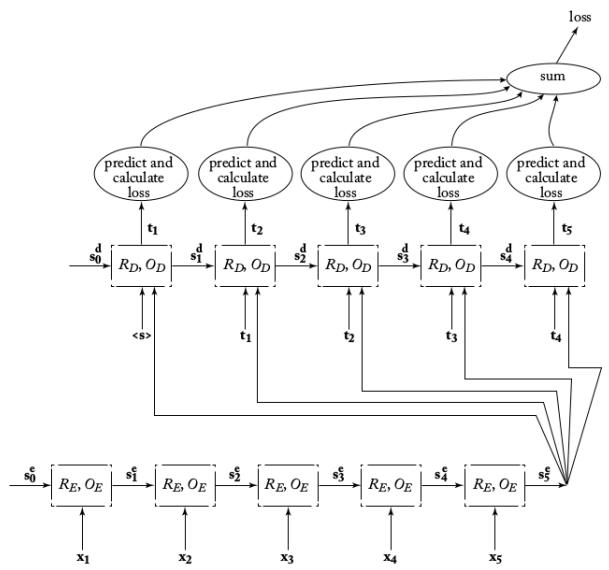
## 11.3 Enfoques de decodificación

- El decodificador tiene como objetivo generar la secuencia de salida con la puntuación máxima (o probabilidad máxima), es decir, de modo que se maximice  $\sum_{i=1}^n P(\hat{f}_i | \hat{f}_{1:i-1})$ .
- La naturaleza no markoviana de la RNN significa que la función de probabilidad no se puede descomponer en factores que permitan una búsqueda exacta utilizando la programación dinámica estándar.
- Búsqueda exacta: encontrar la secuencia óptima requiere evaluar todas las secuencias posibles (computacionalmente prohibitivo).
- Por lo tanto, solo tiene sentido resolver aproximadamente el problema de optimización anterior.
- Búsqueda por greedy: elegir la predicción (palabra) de mayor puntuación en cada paso.
- Esto puede dar como resultado una probabilidad global subóptima y llevar a prefijos seguidos de eventos de baja probabilidad.

### 11.3.1 Búsqueda Beam

- La búsqueda Beam intercala entre la búsqueda exacta y la búsqueda greedy al cambiar el tamaño  $K$  de las hipótesis mantenidas durante todo el procedimiento de búsqueda [Cho, 2015].
- El algoritmo de búsqueda Beam funciona en etapas.
- Primero se eligen las  $K$  palabras de inicio con la probabilidad más alta.
- En cada paso, cada secuencia candidata se expande con todas las posibles siguientes etapas.
- Cada paso candidato se puntúa.
- Se conservan las  $K$  secuencias con las probabilidades más probables y se eliminan todas las demás candidatas.
- El proceso de búsqueda puede detenerse para cada candidato por separado ya sea alcanzando una longitud máxima, alcanzando un token de fin de secuencia o alcanzando una probabilidad umbral.
- Se selecciona la oración con la probabilidad global más alta.

<sup>0</sup>Más información en: <https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/>



## 11.4 Generación condicionada con atención

- En las redes codificador-decodificador, la oración de entrada se codifica en un solo vector, que luego se utiliza como contexto de condicionamiento para un generador RNN.
- Esta arquitectura obliga al vector codificado  $\vec{c}$  a contener toda la información requerida para la generación.
- ¡No funciona bien para oraciones largas!
- También requiere que el generador pueda extraer esta información del vector de longitud fija.
- "¡No puedes meter el significado de una maldita oración en un solo maldito vector!Raymond Mooney
- Esta arquitectura se puede mejorar sustancialmente (en muchos casos) mediante la adición de un mecanismo de atención.
- El mecanismo de atención intenta resolver este problema permitiendo que el decodificador "mire hacia atrás" los estados ocultos del codificador según su estado actual.
- La oración de entrada (una secuencia de entrada de longitud  $n$ ,  $\vec{x}_{1:n}$ ) se codifica utilizando una

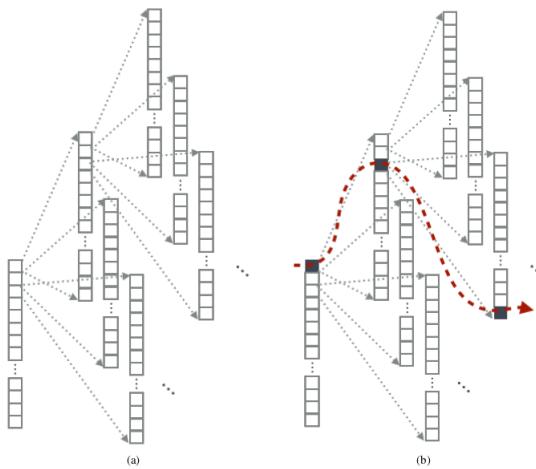
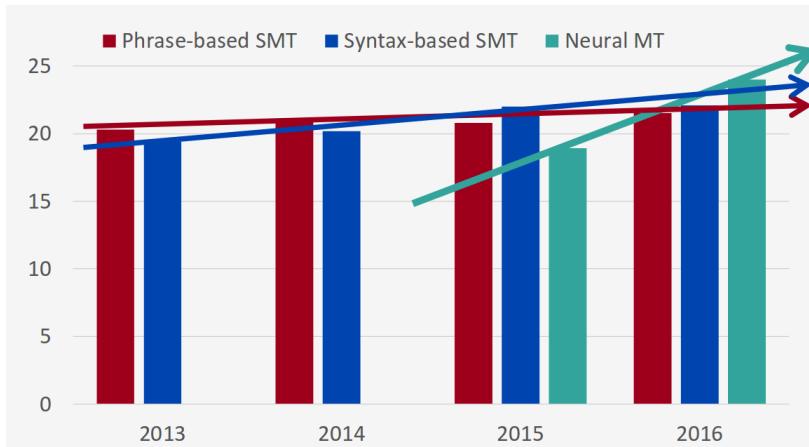


Figure 6.4: (a) Search space depicted as a tree. (b) Greedy search.

Figura 11.2: Búsqueda greedy en la decodificación de secuencia a secuencia.

biRNN como una secuencia de vectores  $\vec{c}_{1:n}$ .

- El decodificador utiliza un mecanismo de atención suave para decidir en qué partes de la entrada codificada debe enfocarse.
- En cada etapa  $j$ , el decodificador ve un promedio ponderado de los vectores  $\vec{c}_{1:n}$ , donde los pesos de atención ( $\vec{\alpha}^j$ ) son elegidos por el mecanismo de atención.

$$\vec{c}^j = \sum_{i=1}^n \vec{\alpha}_{[i]}^j \cdot \vec{c}_i$$

- Los elementos de  $\vec{\alpha}^j$  son todos positivos y suman uno.
- Se producen pesos de atención no normalizados ( $\bar{\alpha}_{[i]}^j$ ) teniendo en cuenta el estado del decodificador en el tiempo  $j$  ( $\vec{s}_j$ ) y cada uno de los vectores  $\vec{c}_i$ .
- Se pueden obtener de varias maneras, básicamente cualquier función diferenciable que devuelva un escalar de dos vectores  $\vec{s}_j$  y  $\vec{c}_i$  se puede utilizar.
- El enfoque más simple es un producto escalar:  $\bar{\alpha}_{[i]}^j = \vec{s}_j \cdot \vec{c}_i$ .

- El que utilizaremos en estas diapositivas es la atención aditiva, que utiliza un perceptrón multicapa:  $\tilde{\alpha}_{[i]}^j = \text{MLP}^{\text{att}}([\vec{s}_j; \vec{c}_i]) = \vec{v} \cdot \tanh([\vec{s}_j; \vec{c}_i]U + \vec{b})$
- Luego, estos pesos no normalizados se normalizan en una distribución de probabilidad utilizando la función softmax.

$$\begin{aligned}\text{attend}(c_{1:n}, \hat{t}_{1:j}) &= c^j \\ c^j &= \sum_{i=1}^n \alpha_{[i]}^j \cdot c_i \\ \alpha^j &= \text{softmax}(\tilde{\alpha}_{[1]}^j, \dots, \tilde{\alpha}_{[n]}^j) \\ \tilde{\alpha}_{[i]}^j &= \text{MLP}^{\text{att}}([\vec{s}_j; c_i]).\end{aligned}$$

- El codificador, el decodificador y el mecanismo de atención se entrenan conjuntamente para interactuar bien entre sí.

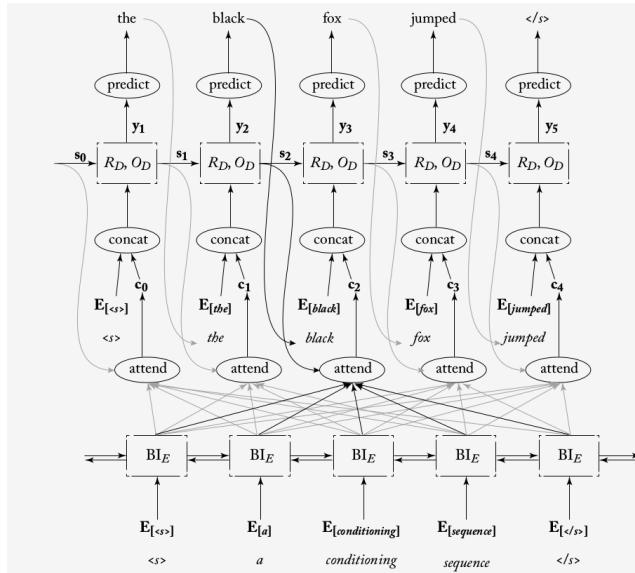


Figura 11.3: Arquitectura de un modelo codificador-decodificador con atención.

La generación de toda la secuencia a secuencia con atención se define como:

- ¿Por qué usar el codificador biRNN para traducir la secuencia de condicionamiento  $\vec{x}_{1:n}$  en los vectores de contexto  $\vec{c}_{1:n}$ ?
- ¿Por qué no simplemente centrarse directamente en las entradas (incrustaciones de palabras)  $\text{MLP}^{\text{att}}([\vec{s}_j; \vec{x}_i])$ ?
- Podríamos hacerlo, pero obtenemos beneficios importantes del proceso de codificación.
- En primer lugar, los vectores biRNN  $\vec{c}_i$  representan los elementos  $\vec{x}_i$  en su contexto oracional.
- Contexto oracional: una ventana enfocada alrededor del elemento de entrada  $\vec{x}_i$  y no el elemento en sí.
- En segundo lugar, al tener un componente de codificación entrenable que se entrena conjuntamente con el decodificador, el codificador y el decodificador evolucionan juntos.
- Por lo tanto, la red puede aprender a codificar propiedades relevantes de la entrada que son útiles para la decodificación y que pueden no estar presentes en la secuencia fuente  $\vec{x}_{1:n}$ .

$$\begin{aligned}
p(t_{j+1} = k \mid \hat{t}_{1:j}, \mathbf{x}_{1:n}) &= f(O_{\text{dec}}(s_{j+1})) \\
s_{j+1} &= R_{\text{dec}}(s_j, [\hat{t}_j; c^j]) \\
c^j &= \sum_{i=1}^n \alpha_{[i]}^j \cdot c_i \\
c_{1:n} &= \text{biRNN}_{\text{enc}}^*(\mathbf{x}_{1:n}) \\
\alpha^j &= \text{softmax}(\bar{\alpha}_{[1]}^j, \dots, \bar{\alpha}_{[n]}^j) \\
\bar{\alpha}_{[i]}^j &= \text{MLP}^{\text{att}}([s_j; c_i]) \\
\hat{t}_j &\sim p(t_j \mid \hat{t}_{1:j-1}, \mathbf{x}_{1:n}) \\
f(z) &= \text{softmax}(\text{MLP}^{\text{out}}(z)) \\
\text{MLP}^{\text{att}}([s_j; c_i]) &= v \tanh([s_j; c_i]U + b).
\end{aligned}$$

directamente.

## 11.5 Atención y alineaciones de palabras

En el contexto de la traducción automática, se puede pensar que  $\text{MLP}^{\text{att}}$  calcula una alineación suave entre el estado actual del decodificador  $\vec{s}_j$  (capturando las palabras extranjeras recién producidas) y cada uno de los componentes de la oración fuente  $\vec{c}_i$ .

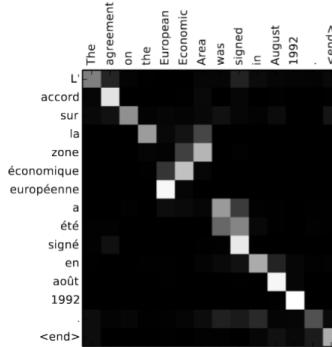


Fig. 2. Visualization of the attention weights  $\alpha_j^t$  of the attention-based neural machine translation model [32]. Each row corresponds to the output symbol, and each column the input symbol. Brighter the higher  $\alpha_j^t$ .

Figura 11.4: Fuente: [Cho et al., 2015]

## 11.6 Otros tipos de atención

### Summary

Below is a summary table of several popular attention mechanisms (or broader categories of attention mechanisms).

Name	Alignment score function	Citation
Additive(*)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_t; \mathbf{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \mathbf{s}_t)$ Note: This simplifies the softmax alignment max to only depend on the target position.	Luong2015
General	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{s}_t^\top \mathbf{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\mathbf{s}_t, \mathbf{h}_i) = \frac{\mathbf{s}_t^\top \mathbf{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017
Self-Attention(&)	Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.	Cheng2016
Global/Soft	Attending to the entire input state space.	Xu2015
Local/Hard	Attending to the part of input state space; i.e. a patch of the input image.	Xu2015; Luong2015

(\*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.

(^) It adds a scaling factor  $1/\sqrt{n}$ , motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

(&) Also, referred to as "intra-attention" in Cheng et al., 2016 and some other papers.

Figura 11.5: Fuente: <https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>



## 12. Arquitectura de Transformer

Dado lo que acabamos de aprender en la conferencia anterior, parecería que la atención resuelve todos los problemas con las RNN y las arquitecturas codificador-decodificador<sup>1</sup>. Sin embargo, hay algunas deficiencias de las RNN que otra arquitectura llamada **Transformer** intenta abordar. El Transformer descarta el componente recursivo de la arquitectura codificador-decodificador y se basa únicamente en los mecanismos de atención [Vaswani et al., 2017]. Cuando procesamos una secuencia con RNN, cada estado oculto depende del estado oculto anterior. Esto se convierte en un gran problema en las GPU: las GPU tienen una gran capacidad de cálculo y no les gusta tener que esperar a que los datos estén disponibles. Incluso con tecnologías como CuDNN, las RNN son dolorosamente inefficientes y lentas en la GPU.

### 12.0.1 Dependencias en la traducción automática neuronal

En esencia, hay tres tipos de dependencias en la traducción automática neuronal:

1. Dependencias entre los tokens de entrada y los tokens de salida.
2. Dependencias entre los tokens de entrada en sí.
3. Dependencias entre los tokens de salida en sí.

El mecanismo de atención tradicional resolvió en gran medida la primera dependencia al darle al decodificador acceso a toda la secuencia de entrada. Las dependencias segunda y tercera fueron abordadas por las RNN.

### 12.1 El Transformer

La idea novedosa del Transformer es extender este mecanismo para procesar las oraciones de entrada y salida también. La RNN procesa secuencias de entrada de manera secuencial. El Transformer, por otro lado, permite que el codificador y el decodificador vean toda la secuencia de entrada de una vez. Esto se logra mediante la atención.

<sup>1</sup>El material siguiente se basa en: <http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/> y <http://jalammar.github.io/illustrated-transformer/>

Comencemos por ver el Transformer como una caja negra única. En una aplicación de traducción automática, tomaría una oración en un idioma y produciría su traducción en otro.



Figura 12.1: El Transformer como una caja negra única.

Esta caja negra se puede descomponer en un componente de codificación, un componente de decodificación y conexiones entre ellos.

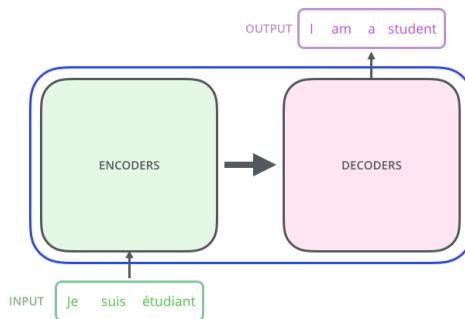


Figura 12.2: El Transformer descompuesto en componentes de codificación y decodificación.

El componente de codificación es una pila de codificadores. El Transformer original apila seis de ellos uno encima del otro. Pero no hay nada mágico en el número seis, definitivamente se pueden experimentar con otros arreglos. El componente de decodificación es una pila de decodificadores del mismo número.

El Transformer todavía utiliza el diseño codificador-decodificador básico de los sistemas de traducción automática neuronal RNN. El lado izquierdo es el codificador y el lado derecho es el decodificador. Las entradas iniciales al codificador son las incrustaciones de la secuencia de entrada. Las entradas iniciales al decodificador son las incrustaciones de las salidas hasta ese momento. El codificador y el decodificador están compuestos por  $N$  bloques (donde  $N = 6$  para ambas redes). Estos bloques también están compuestos por bloques más pequeños. Antes de examinar cada bloque con más detalle, intentemos comprender el mecanismo de atención implementado por el Transformer.

## 12.2 Mecanismo de atención en el Transformer

El mecanismo de atención en el Transformer se interpreta como una forma de calcular la relevancia de un conjunto de **valores** (información) en función de algunas **claves** y **consultas**. El mecanismo de atención se utiliza como una forma para que el modelo **se centre en la información relevante** según lo que está procesando actualmente. En la arquitectura codificador-decodificador RNN con atención:

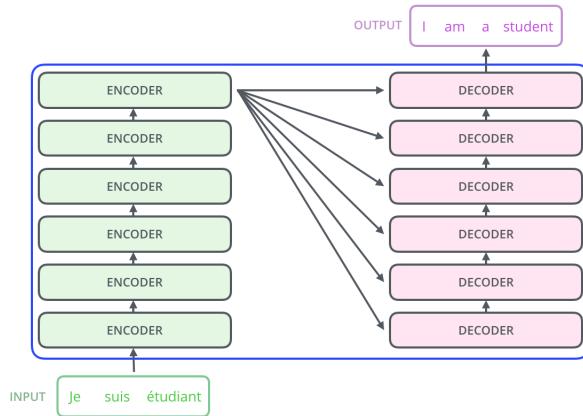


Figura 12.3: Pila de codificadores y decodificadores del Transformer.

1. Los pesos de atención eran la relevancia de los estados ocultos del codificador (valores) en el procesamiento del estado del decodificador (consulta).
2. Estos valores se calcularon en función de los estados ocultos del codificador (claves) y el estado oculto del decodificador (consulta).

En este ejemplo, la consulta es la palabra que se está decodificando (lo cual significa “perro”) y tanto las claves como los valores son la oración fuente. El puntaje de atención representa la relevancia y, en este caso, es alto para la palabra “perro” y bajo para las demás. Cuando pensamos en la atención de esta manera, podemos ver que las claves, valores y consultas podrían ser cualquier cosa. Incluso podrían ser iguales. Por ejemplo, tanto los valores como las consultas podrían ser las incrustaciones de entrada (autoatención).

### 12.2.1 Consultas

Las consultas son representaciones de la secuencia de destino o salida que el modelo Transformer utiliza para determinar cuánta atención se debe prestar a cada palabra en la secuencia de entrada. Cada palabra o token en la secuencia de salida se asocia con una consulta. Las consultas ayudan a recuperar la información relevante de la secuencia de entrada. Ejemplo: Si estamos generando la traducción para la oración “Je adore les chats” (que significa “Me encantan los gatos” en francés), cada palabra en la secuencia de salida tendría su representación de consulta correspondiente. Por ejemplo, “Je” tendría un vector de consulta, “adore” tendría un vector de consulta, y así sucesivamente.

### 12.2.2 Claves

Las claves son representaciones de la secuencia de entrada que el modelo Transformer utiliza para calcular los puntajes de atención. Cada palabra o token en la secuencia de entrada se asocia con una clave. Estas claves capturan la información necesaria para comprender el contexto y las relaciones entre las palabras de la secuencia. Ejemplo: Consideremos la secuencia de entrada: “I love cats” (Amo a los gatos). Cada palabra en la secuencia tendría su representación de clave correspondiente, como “I” teniendo un vector de clave, “love” teniendo un vector de clave, y así sucesivamente.

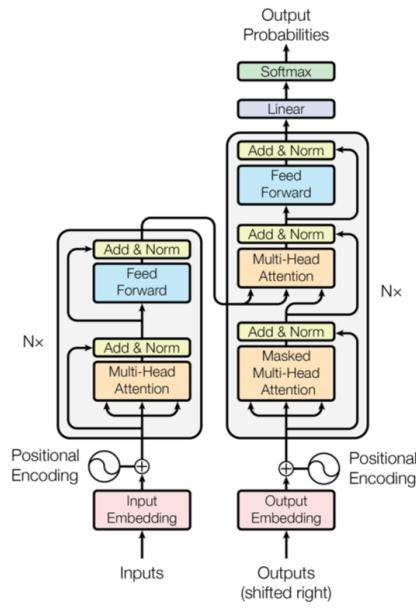


Figure 1: The Transformer - model architecture.

Figura 12.4: Estructura general del Transformer.

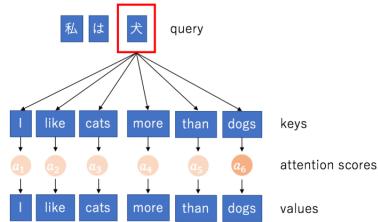


Figura 12.5: Concepto de atención en el Transformer.

### 12.2.3 Valores

Los valores son la información o características reales asociadas con cada palabra de la secuencia de entrada. Estos valores se utilizan para calcular la suma ponderada durante el cálculo de la atención, lo que ayuda a determinar la importancia o relevancia de cada palabra. Ejemplo: Considerando la misma secuencia de entrada “I love cats”, cada palabra tendría su representación de valor asociada. Estos valores contienen la información contextual para cada palabra. Por ejemplo, “I” tendría un vector de valor, “love” tendría un vector de valor, y así sucesivamente. Las claves y los valores son muy difíciles de distinguir a primera vista porque en la codificación y decodificación RNN con atención clásica son iguales. Veremos con el Transformer que aunque provienen de la misma secuencia, pueden corresponder a vectores diferentes.

### 12.2.4 Atención de producto puntual escalado

El Transformer utiliza una forma particular de atención llamada “Atención de Producto Puntual Escalado”. Para un vector de consulta dado  $\vec{q}$ , una secuencia de vectores clave  $\vec{k}_{1:m}$  y una secuencia

de vectores valor  $\vec{v}_{1:m}$ , los pesos de atención  $\alpha_1, \dots, \alpha_m$  se calculan de la siguiente manera:

$$\alpha_1, \dots, \alpha_m = \text{softmax} \left( \frac{\vec{q} \cdot \vec{k}_1}{\sqrt{d}}, \dots, \frac{\vec{q} \cdot \vec{k}_1}{\sqrt{d}} \right)$$

$d$  representa la dimensionalidad de las consultas y las claves.

La normalización sobre  $\sqrt{d}$  se utiliza para reescalar los productos punto entre consultas y claves (los productos punto tienden a crecer con la dimensionalidad). Los pesos de atención se multiplican entonces por sus valores correspondientes para calcular una suma ponderada, que luego se pasa a las capas subsiguientes de la red:

$$\alpha_1 * \vec{v}_1 + \dots + \alpha_m * \vec{v}_m$$

Ahora estamos listos para analizar más de cerca cada parte del Transformer.

## 12.3 El Codificador

El codificador contiene capas de autoatención.

En una capa de autoatención, todas las claves, valores y consultas provienen del mismo lugar, en este caso, la salida de la capa anterior en el codificador.

Cada posición en el codificador puede atender a todas las posiciones en la capa anterior del codificador.

El codificador está compuesto por dos bloques (que llamaremos subcapas para distinguirlos de los  $N$  bloques que componen el codificador y el decodificador).

Uno es la subcapa de atención multihead sobre las entradas, mencionada anteriormente.

El otro es una red neuronal de alimentación directa simple.

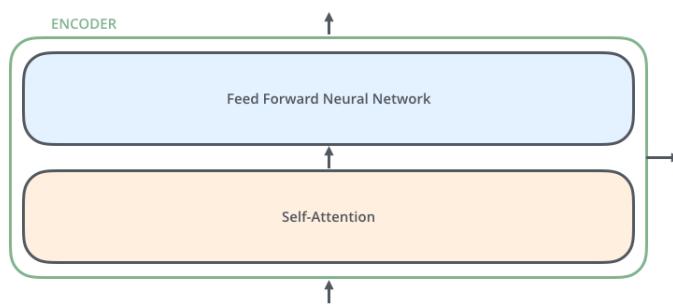


Figura 12.6: Estructura del codificador del Transformer.

### Introduciendo los tensores en la imagen

Comenzamos convirtiendo cada palabra de entrada en un vector utilizando una capa de incrustación de tamaño 512 en el codificador más inferior.

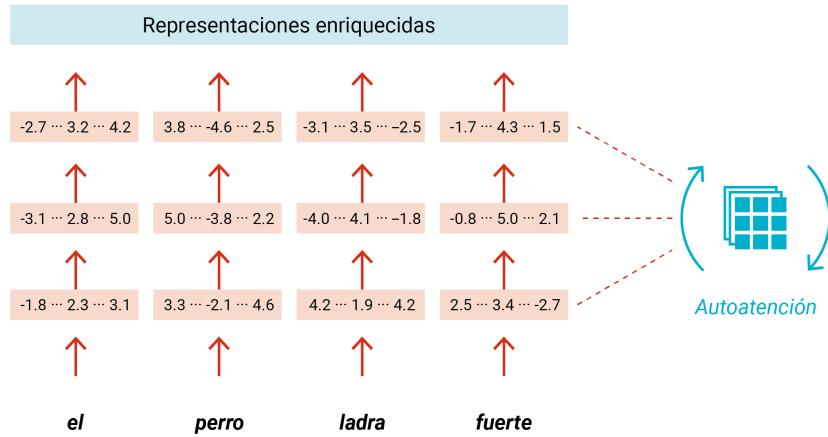


Figura 12.7: Idea general del proceso de auto-atención.

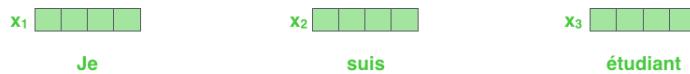


Figura 12.8: Capa de incrustación en el codificador.

En el codificador inferior, esto serían las incrustaciones de palabras, pero en otros codificadores, sería la salida del codificador que está directamente debajo. El tamaño de esta lista es un hiperparámetro que podemos establecer: básicamente sería la longitud de la oración más larga en nuestro conjunto de datos de entrenamiento.

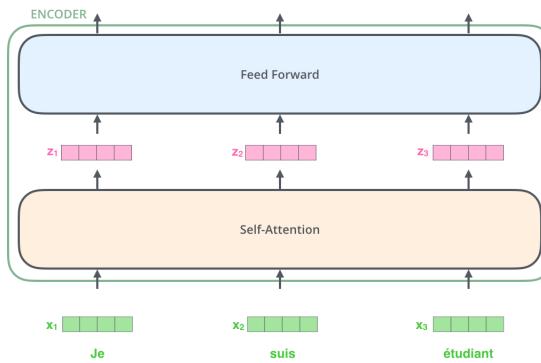


Figura 12.9: Representación de las incrustaciones y los vectores de entrada en el codificador.

La palabra en cada posición fluye a través de su propio camino en el codificador. Hay dependencias entre estos caminos en la capa de autoatención. La capa de alimentación directa no tiene esas dependencias. Por lo tanto, los diversos caminos pueden ejecutarse en paralelo mientras fluyen a través de la capa de alimentación directa. ¡Ahora estamos codificando! Como hemos mencionado anteriormente, un codificador recibe una lista de vectores como entrada. Procesa esta lista pasando

estos vectores a una capa de “autoatención”, luego a una red neuronal de alimentación directa, y luego envía la salida hacia arriba al siguiente codificador.

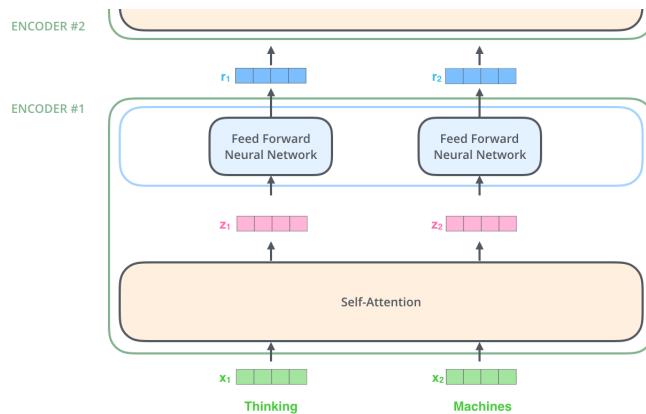


Figura 12.10: Flujo de datos en el codificador del Transformer.

## 12.4 Autoatención a alto nivel

Supongamos que la siguiente oración es una oración de entrada que queremos traducir:  
**“The animal didn’t cross the street because it was too tired”**  
 ¿A qué se refiere “it” en esta oración?  
 ¿Se refiere a la calle o al animal? No es una pregunta tan simple para un algoritmo como lo es para un humano.

Cuando el modelo está procesando la palabra “it”, la autoatención le permite asociarla con “animal”.

A medida que el modelo procesa cada palabra (cada posición en la secuencia de entrada), la autoatención le permite examinar otras posiciones en la secuencia de entrada en busca de pistas que puedan ayudar a obtener una mejor codificación para esta palabra.

Piense en cómo mantener un estado oculto permite que una RNN incorpore su representación de palabras/vectores anteriores que ha procesado con la palabra actual que está procesando. La autoatención es el método que el Transformer utiliza para integrar la “comprensión” de otras palabras relevantes en la que estamos procesando actualmente.

## 12.5 Autoatención en detalle

### Paso 1

El **primer paso** para calcular la autoatención con producto puntual escalado consiste en crear tres vectores a partir de cada uno de los vectores de entrada del codificador (en este caso, la incrustación de cada palabra).

Para cada palabra, creamos un vector de consulta (**Query**), un vector clave (**Key**) y un vector de valor (**Value**).

Estos vectores se crean multiplicando la incrustación por tres matrices que hemos entrenado durante el proceso de entrenamiento.

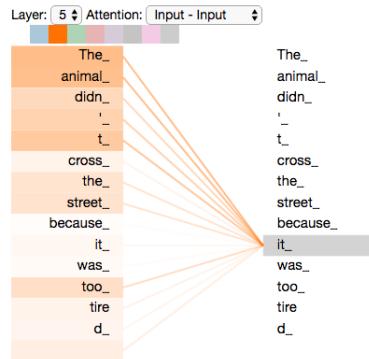
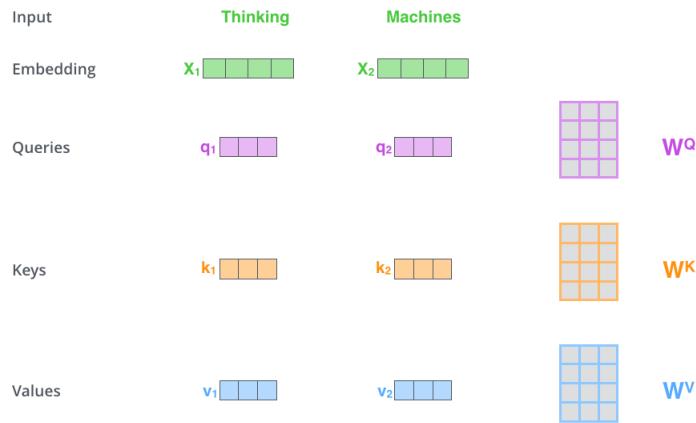


Figura 12.11: Visualización de la autoatención en el Transformer.

Observa que estos nuevos vectores son de menor dimensión que el vector de incrustación. Su dimensionalidad es de 64, mientras que los vectores de entrada/salida del codificador tienen una dimensionalidad de 512. No es necesario que sean más pequeños, esta es una elección arquitectónica para hacer que el cálculo de la atención multihead sea (en su mayoría) constante.



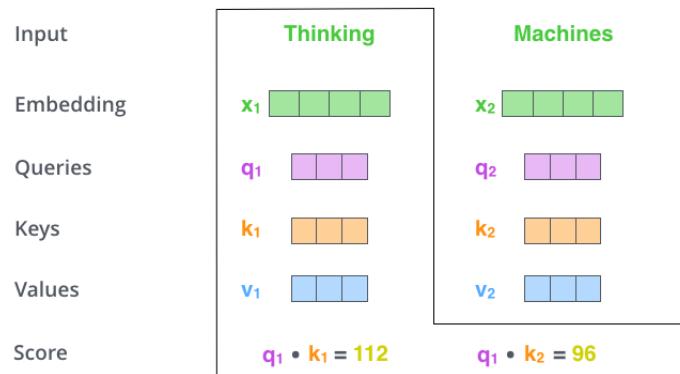
Por ejemplo, en la oración "Thinking Machines", al multiplicar  $x_1$  por la matriz de pesos  $WQ$  se obtiene el vector  $q_1$ , que es el vector de consulta asociado a esa palabra. De esta manera, creamos una proyección de consulta, clavez "valor"para cada palabra de la oración de entrada.

## Paso 2

El **segundo paso** para calcular la autoatención es calcular una puntuación.

Supongamos que estamos calculando la autoatención para la primera palabra en este ejemplo, "Thinking". Necesitamos puntuar cada palabra de la oración de entrada en relación con esta palabra. La puntuación determina cuánto enfoque debemos poner en otras partes de la oración de entrada a medida que codificamos una palabra en una posición determinada.

La puntuación se calcula tomando el producto punto del vector de consulta"(Query) con el vector de clave"(Key) de la palabra correspondiente que estamos puntuando. Por lo tanto, si estamos procesando la autoatención para la palabra en la posición #1, la primera puntuación sería el producto punto de  $q_1$  y  $k_1$ . La segunda puntuación sería el producto punto de  $q_1$  y  $k_2$ .

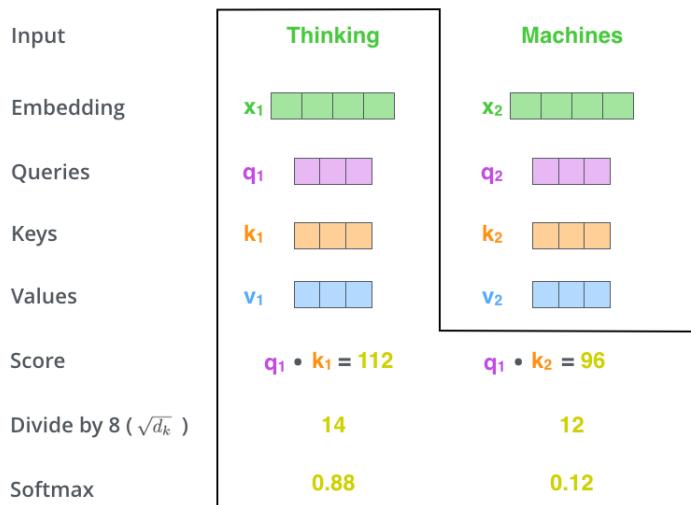


### Pasos 3 y 4

Los **tercer** y **cuarto** pasos consisten en dividir las puntuaciones por 8 (la raíz cuadrada de la dimensión de los vectores clave utilizados en el artículo, es decir, 64).

Esto ayuda a obtener gradientes más estables. Podría haber otros posibles valores aquí, pero este es el valor por defecto. Luego, se pasa el resultado por una operación de softmax. La función softmax normaliza las puntuaciones para que sean todas positivas y sumen 1.

Estas puntuaciones de softmax determinan cuánto se expresará cada palabra en la posición actual. Claramente, la palabra en la posición actual tendrá la puntuación softmax más alta, pero a veces es útil prestar atención a otra palabra relevante para la palabra actual.



### Autoatención en detalle: pasos 5 y 6

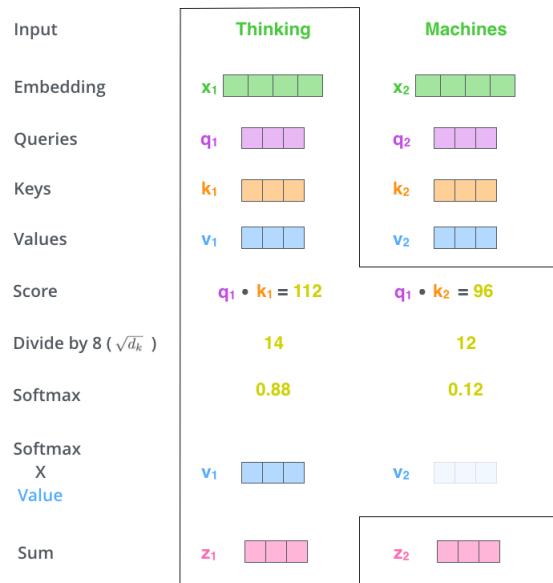
El **quinto paso** es multiplicar cada vector de valor por la puntuación softmax (en preparación para sumarlos).

La intuición aquí es mantener intactos los valores de las palabras en las que queremos enfocarnos y atenuar las palabras irrelevantes (multiplicándolas por números pequeños como 0.001, por ejemplo).

El **sexto paso** es sumar los vectores de valor ponderados. Esto produce la salida de la capa de autoatención en la posición actual (para la primera palabra en este ejemplo).

Con esto concluye el cálculo de la autoatención con producto puntual escalado. El vector

resultante es el que podemos enviar a la red neuronal de avance (*feed-forward neural network*).



## 12.6 Cálculo matricial de la autoatención

En la implementación real, la autoatención con producto puntual escalado se calcula en forma matricial para un procesamiento más rápido. Veamos eso ahora que hemos comprendido la intuición del cálculo a nivel de palabras. El primer paso es calcular las matrices de consulta ( $Q$ ), clave ( $K$ ) y valor ( $V$ ):

$$\begin{array}{ccc}
 X & W^Q & Q \\
 \begin{matrix} \text{green, green, green} \\ \times \end{matrix} & \begin{matrix} \text{purple, purple, purple} \\ \times \end{matrix} & = \begin{matrix} \text{purple, purple, purple} \end{matrix} \\
 \\ 
 X & W^K & K \\
 \begin{matrix} \text{green, green, green} \\ \times \end{matrix} & \begin{matrix} \text{orange, orange, orange} \\ \times \end{matrix} & = \begin{matrix} \text{orange, orange, orange} \end{matrix} \\
 \\ 
 X & W^V & V \\
 \begin{matrix} \text{green, green, green} \\ \times \end{matrix} & \begin{matrix} \text{blue, blue, blue} \\ \times \end{matrix} & = \begin{matrix} \text{blue, blue, blue} \end{matrix}
 \end{array}$$

Lo hacemos empaquetando nuestras incrustaciones en una matriz  $X$  y multiplicándola por las matrices de pesos que hemos entrenado ( $WQ$ ,  $WK$ ,  $WV$ ).

Finalmente, como estamos tratando con matrices, podemos condensar los pasos dos a seis en una fórmula para calcular las salidas de la capa de autoatención.

$$\begin{aligned}
 & \text{softmax} \left( \frac{\text{Q} \times \text{K}^T}{\sqrt{d_k}} \right) \text{V} \\
 = & \text{Z}
 \end{aligned}$$

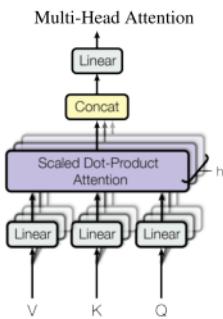
Q      K<sup>T</sup>      V  
Z

## 12.7 Atención multi-head

Si solo calculáramos una única suma ponderada de atención de los valores, sería difícil capturar diferentes aspectos del input. En el ejemplo anterior,  $z_1$  contiene un poco de cada codificación, pero podría estar dominado por la palabra en sí misma. Si estamos traduciendo una oración como "The animal didn't cross the street because it was too tired"(El animal no cruzó la calle porque estaba demasiado cansado), sería útil saber a qué se refiere la palabra *it*"(él o ella). Para expandir el modelo y aprender representaciones diversas centradas en diferentes posiciones, el Transformer utiliza el bloque de atención multi-head.

La atención multi-head calcula múltiples sumas ponderadas de atención en lugar de una sola pasada de atención sobre los valores. En esencia, aplica transformaciones lineales diferentes a los valores, claves y consultas para cada "head" de atención.

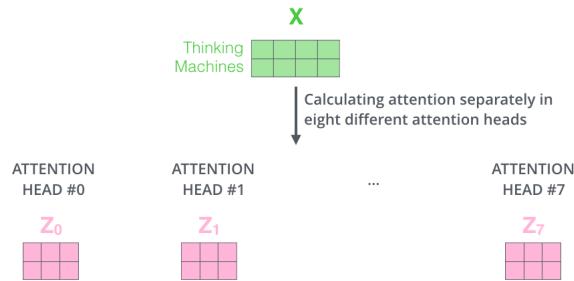
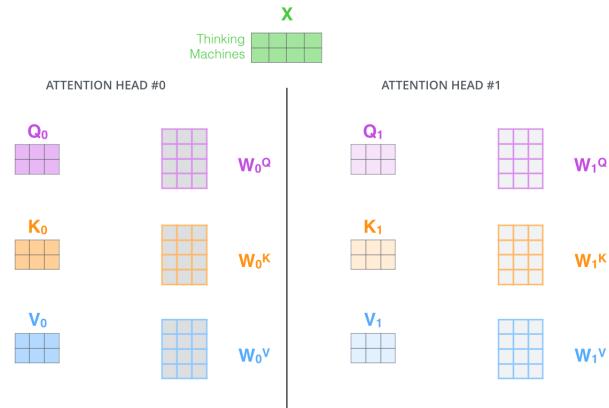
El bloque de atención multi-head aplica múltiples bloques de atención con producto puntual escalado en paralelo, concatena sus salidas y luego aplica una única transformación lineal.



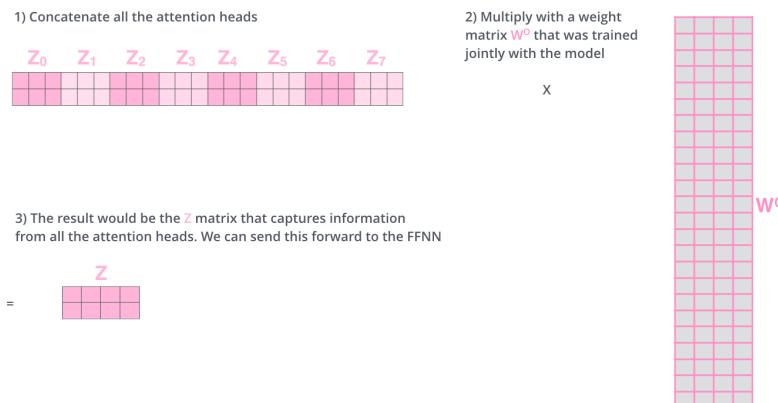
Con la atención multi-head, no tenemos uno, sino varios conjuntos de matrices de pesos de consulta/clave/valor. El Transformer utiliza ocho cabezas de atención, por lo que obtenemos ocho conjuntos para cada codificador/decodificador. Cada uno de estos conjuntos se inicializa de forma aleatoria. Luego, después del entrenamiento, cada conjunto se utiliza para proyectar las incrustaciones de entrada (o vectores de codificadores/decodificadores inferiores) en un subespacio de representación diferente. El componente de atención multi-head proporciona a la capa de atención múltiples "subespacios de representación"

Con la atención multi-head, mantenemos matrices de pesos de consulta/clave/valor separadas para cada cabeza, lo que resulta en diferentes matrices de consulta/clave/valor. Como hicimos antes, multiplicamos  $X$  por las matrices  $WQ/WK/WV$  para obtener las matrices de consulta/clave/valor.

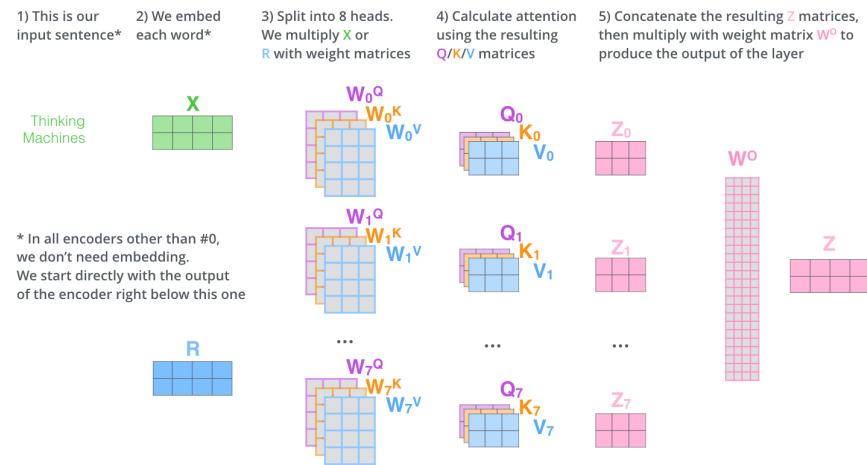
Si realizamos el mismo cálculo de autoatención que hemos descrito anteriormente, pero ocho



veces diferentes con diferentes matrices de pesos, obtendremos ocho matrices  $Z$  diferentes. Sin embargo, la red de alimentación espera una única matriz (un vector para cada palabra), no ocho matrices. Entonces, necesitamos una forma de condensar estas ocho matrices en una sola matriz. Esto se logra concatenando las matrices y luego multiplicándolas por una matriz de pesos adicional  $W^O$ .

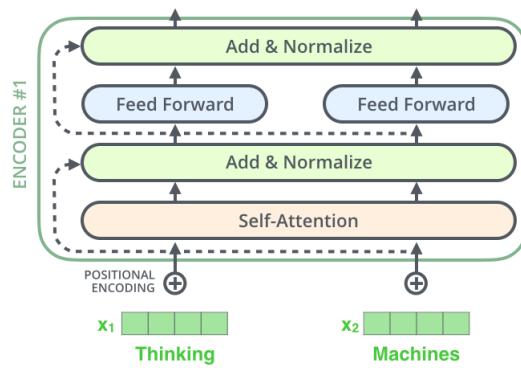


Veamos todas estas matrices juntas en una visualización para poder verlas en un solo lugar.



## 12.8 Conexiones residuales

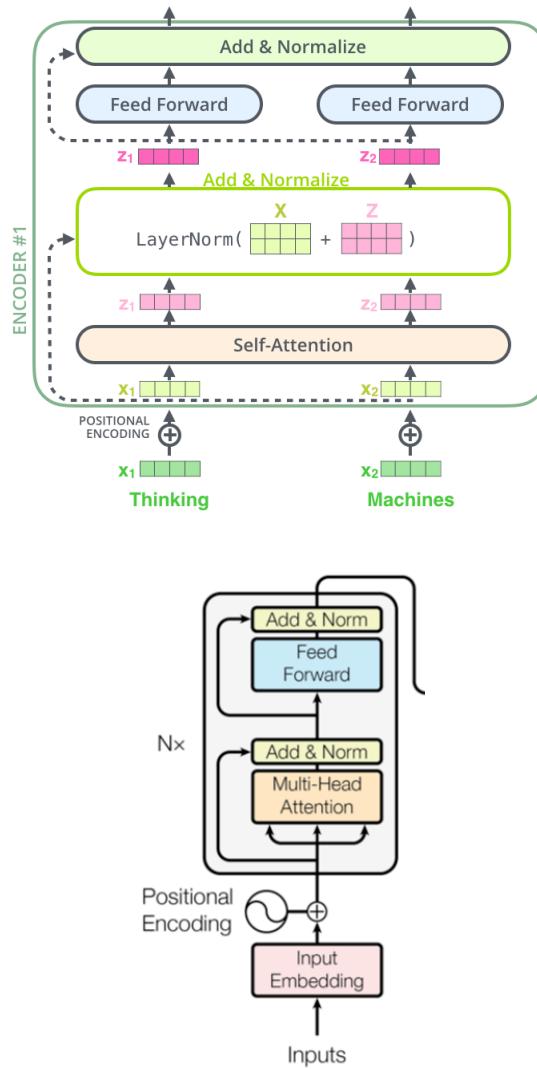
Entre cada subcapa, hay una conexión residual seguida de una normalización de capa. Una conexión residual consiste básicamente en tomar la entrada y sumarla a la salida de la subred, y es una forma de facilitar el entrenamiento de redes profundas. La normalización de capa es un método de normalización en el aprendizaje profundo que es similar a la normalización de lote (*batch normalization*).



Si visualizamos los vectores y la operación de normalización de capa asociada con la autoatención, se vería así:

## 12.9 El codificador: resumen

Lo que hace cada bloque codificador es, en realidad, una serie de multiplicaciones de matrices seguidas de un par de transformaciones elemento a elemento. Es por eso que el Transformer es tan rápido: todo se reduce a multiplicaciones de matrices paralelizables. El punto es que, apilando estas transformaciones una encima de la otra, podemos crear una red muy potente. El núcleo de esto es el mecanismo de atención, que modifica y atiende una amplia gama de información.

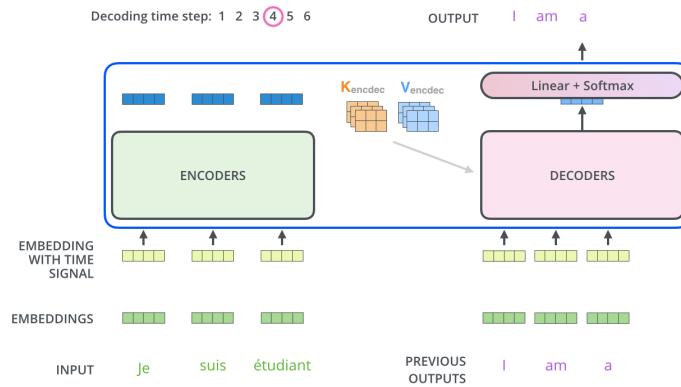


## 12.10 El Decodificador

El decodificador del Transformer consta de dos tipos de capas de atención: atención propia y atención codificador-decodificador. Las capas de atención propia en el decodificador permiten que cada posición en el decodificador preste atención a todas las posiciones dentro del propio decodificador, de manera similar a cómo funciona el estado oculto en las arquitecturas de traducción de máquinas RNN. Por otro lado, la capa de “atención codificador-decodificador” permite que el decodificador se enfoque en partes relevantes de la secuencia de entrada.

La capa de atención codificador-decodificador funciona de manera similar a la atención propia de múltiples cabezas, pero genera su matriz de consultas a partir de la capa inferior y utiliza las matrices de claves y valores de la salida de la pila del codificador. Esto permite que cada posición en el decodificador atienda a todas las posiciones en la secuencia de entrada, imitando los mecanismos de atención codificador-decodificador típicos vistos en los modelos de secuencia a secuencia RNN.

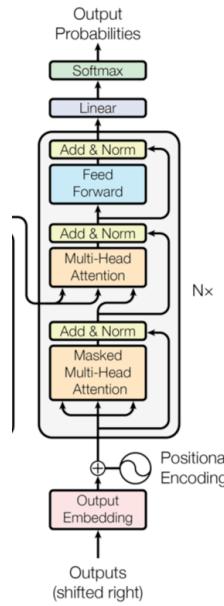
Cada paso en la fase de decodificación produce un elemento de la secuencia de salida (la oración de traducción al inglés en este caso). Este proceso se repite hasta que se alcanza un símbolo especial



que indica que el decodificador del Transformer ha completado su salida. Cuando entrenamos el Transformer, queremos procesar todas las oraciones al mismo tiempo.

Sin embargo, si le damos al decodificador acceso a toda la oración de destino, el modelo puede simplemente repetir la oración de destino (en otras palabras, no necesita aprender nada). La capa de atención propia solo debe poder atender a las posiciones anteriores en la secuencia de salida. Esto se logra enmascarando los tokens “futuros” al decodificar una palabra determinada.

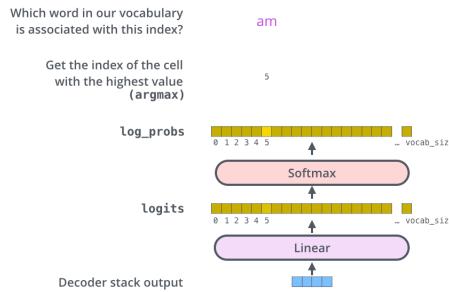
El enmascaramiento se realiza estableciendo en  $-\infty$  todos los valores en la entrada de la función softmax que corresponden a conexiones ilegales. Es por eso que los “bloques de atención propia” en el decodificador se llaman “enmascarados”: las entradas al decodificador de pasos de tiempo futuros están enmascaradas.



## 12.11 La Capa Lineal Final y el Entrenamiento

La pila del decodificador genera un vector de números reales como su salida. Para convertir este vector en una palabra, se utiliza la capa lineal final, seguida de una capa softmax. La capa lineal

es una red neuronal completamente conectada que proyecta el vector de salida del decodificador en un vector más grande llamado vector de logits. El vector de logits tiene una dimensión igual al tamaño del vocabulario de salida del modelo, que contiene 10,000 palabras en inglés únicas. Cada celda del vector de logits representa la puntuación de una palabra específica. La capa softmax luego transforma estas puntuaciones en probabilidades, asegurándose de que todas sean positivas y sumen 1.0.

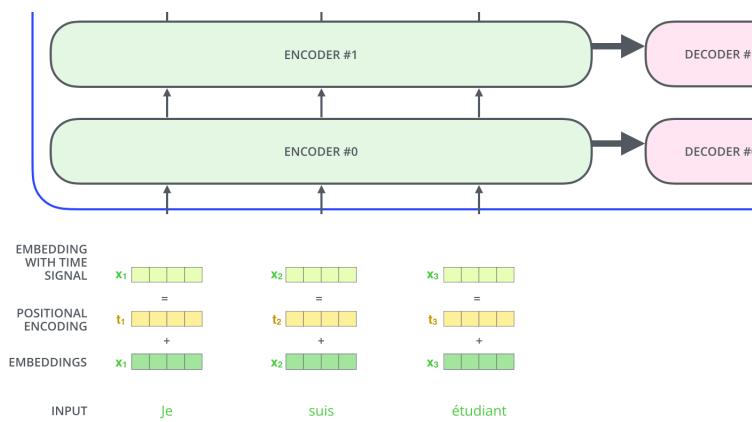


La palabra con la probabilidad más alta se elige como la salida para el paso de tiempo actual. Durante el entrenamiento del Transformer, se utiliza una pérdida de entropía cruzada. Esta pérdida mide la diferencia entre la distribución de palabras predicha y la palabra objetivo, que se representa como un vector one-hot.

## 12.12 Codificaciones posicionales

A diferencia de las redes recurrentes, la red de atención múltiple no puede utilizar de manera natural la posición de las palabras en la secuencia de entrada. Sin codificaciones posicionales, la salida de la red de atención múltiple sería la misma para las oraciones “I like cats more than dogs” y “I like dogs more than cats”.

Las codificaciones posicionales codifican explícitamente las posiciones relativas/absolutas de las entradas como vectores y luego se suman a las incrustaciones de entrada.



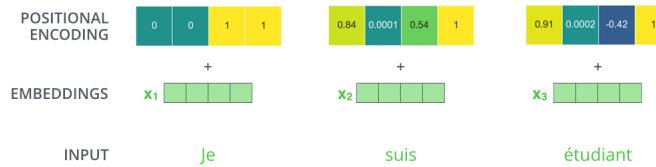
Para darle al modelo una idea del orden de las palabras, agregamos vectores de codificación posicional, cuyos valores siguen un patrón específico.

El documento utiliza la siguiente ecuación para calcular las codificaciones posicionales:  

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$$

$$PE(pos, 2i + 1) = \cos(pos/10000^{2i/d_{model}})$$

Donde  $pos$  representa la posición, e  $i$  es la dimensión. Básicamente, cada dimensión de la codificación posicional es una onda con una frecuencia diferente.



Un ejemplo real de codificación posicional con un tamaño de incrustación de juguete de 4.

## 12.13 Conclusiones

El Transformer logra mejores puntuaciones BLEU que los modelos anteriores del estado del arte para la traducción del inglés al alemán y del inglés al francés a una fracción del costo de entrenamiento.

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	<b><math>3.3 \cdot 10^{18}</math></b>	
Transformer (big)	<b>28.4</b>	<b>41.0</b>	$2.3 \cdot 10^{19}$	

El Transformer es una alternativa potente y eficiente a las redes neuronales recurrentes para modelar dependencias utilizando solo mecanismos de atención.

Se ha establecido como la arquitectura de facto en el procesamiento del lenguaje natural y sirve como base para los modelos de lenguaje grandes y modernos.





## 13. Grandes Modelos de Lenguaje

### 13.1 Representaciones para una palabra

Hasta ahora, básicamente hemos tenido una representación de las palabras, las incrustaciones de palabras que ya hemos aprendido: Word2vec, GloVe, fastText. Estas incrustaciones tienen una calidad semi-supervisada útil, ya que se pueden aprender a partir de corpora no etiquetados y se utilizan en nuestras arquitecturas orientadas a tareas posteriores (LSTM, CNN, Transformer).

Sin embargo, presentan dos problemas. El problema 1 es que siempre producen la misma representación para un tipo de palabra, independientemente del contexto en el que se encuentre un token de palabra. Podríamos querer una desambiguación muy precisa del sentido de las palabras. El problema 2 es que solo tenemos una representación para una palabra, pero las palabras tienen diferentes aspectos, incluyendo la semántica, el comportamiento sintáctico y el registro/connotaciones.

### 13.2 Los Modelos de Lenguaje Neurales pueden producir Incrustaciones Contextualizadas

En un Modelo de Lenguaje Neural (MLN), introducimos inmediatamente vectores de palabras (tal vez solo entrenados en el corpus) a través de capas LSTM. Estas capas LSTM se entran para predecir la siguiente palabra. Sin embargo, estos modelos de lenguaje producen representaciones de palabras específicas del contexto en los estados ocultos de cada posición.

### 13.3 ELMo: Incrustaciones de Modelos de Lenguaje

La idea detrás de ELMo es entrenar un modelo de lenguaje grande (LM) con una red neuronal recurrente y utilizar sus estados ocultos como "incrustaciones de palabras contextualizadas" [Peters et al., 2018]. ELMo es un modelo de lenguaje bidireccional con 2 capas de biLSTM y alrededor de 100 millones de parámetros. Utiliza una CNN de caracteres para construir la representación inicial de las palabras. Utiliza 2048 filtros de n-gramos de caracteres y 2 capas de paso alto, con una proyección de 512 dimensiones. Utiliza estados LSTM ocultos/celdas de 4096 dimensiones con proyecciones de 512

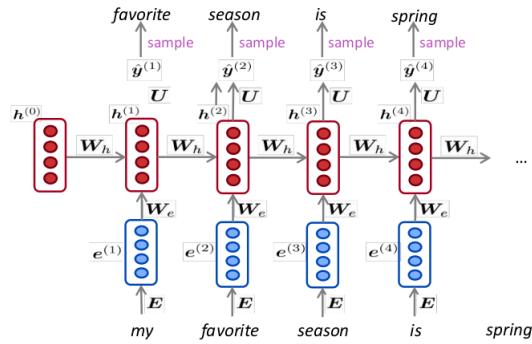


Figura 13.1: Modelo de Lenguaje Neural con capas LSTM

dimensiones para la siguiente entrada. Utiliza una conexión residual y los parámetros de la entrada de token y la salida (softmax) están ligados.

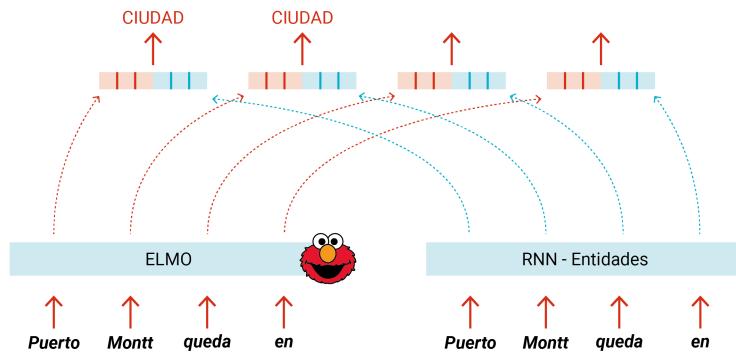


Figura 13.2: Arquitectura de ELMo

### 13.3.1 ELMo: Uso con una tarea

El primer paso es ejecutar el modelo de lenguaje bidireccional para obtener representaciones para cada palabra. Luego, el modelo de tarea específica puede utilizar estas representaciones. Se congelan los pesos de ELMo con fines de modelo supervisado y se concatenan los pesos de ELMo en el modelo específico de la tarea.

#### ELMo: Resultados

### 13.4 ULMfit

Howard y Ruder (2018) propusieron Universal Language Model Fine-tuning (ULMfit) para la clasificación de texto [Howard and Ruder, 2018]. La idea general es transferir el conocimiento de un modelo de lenguaje a la tarea específica. En ULMfit, se entrena un modelo de lenguaje en un corpus grande y general y luego se ajusta en los datos de la tarea objetivo. Finalmente, se utiliza el modelo ajustado como clasificador en la tarea objetivo.

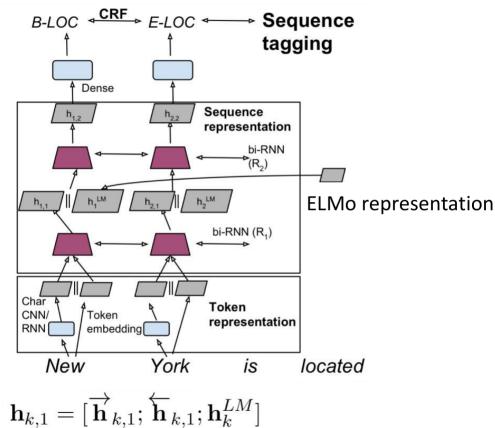


Figura 13.3: Uso de ELMo en una tarea específica

Name	Description	Year	F1
ELMo	ELMo in BiLSTM	2018	92.22
TagLM Peters	LSTM BiLM in BiLSTM tagger	2017	91.93
Ma + Hovy	BiLSTM + char CNN + CRF layer	2016	91.21
Tagger Peters	BiLSTM + char CNN + CRF layer	2017	90.87
Ratinov + Roth	Categorical CRF+Wikipedia+word cls	2009	90.80
Finkel et al.	Categorical feature CRF	2005	86.86
IBM Florian	Linear/softmax/TBL/HMM ensemble, gazettes++	2003	88.76
Stanford	MEMM softmax markov model	2003	86.07

Figura 13.4: Resultados de ELMo en diferentes tareas

### 13.4.1 Énfasis de ULMfit

ULMfit utiliza un modelo de lenguaje de "1 GPU" de tamaño razonable, en lugar de uno enorme. Se presta mucha atención al ajuste fino del modelo de lenguaje, con tasas de aprendizaje diferentes por capa, un programa de aprendizaje con tasas de aprendizaje triangular y un descongelamiento gradual de capas y programación de tasas de aprendizaje triangular al aprender el clasificador. Para la clasificación, se utiliza la concatenación de los estados  $h_T$ ,  $\text{maxpool}(h)$  y  $\text{meanpool}(h)$ .

### 13.4.2 Transferencia de aprendizaje con ULMfit

## 13.5 ¡Aumentemos la escala!

### 13.6 BERT (Bidirectional Encoder Representations from Transformers)

La idea detrás de BERT es combinar ideas de ELMO, ULMFit y el Transformer [Kenton and Toutanova, 2019]. BERT es un modelo grande (335 millones de parámetros) entrenado a partir de un corpus no etiquetado utilizando un codificador Transformer y luego se ajusta en otras tareas posteriores.

Las propiedades paralelizables del Transformer permiten que el modelo se escala a más parámetros, a diferencia de las RNN, que deben procesarse secuencialmente. BERT no predice la siguiente palabra en una oración como un modelo de lenguaje tradicional, sino que utiliza un objetivo de

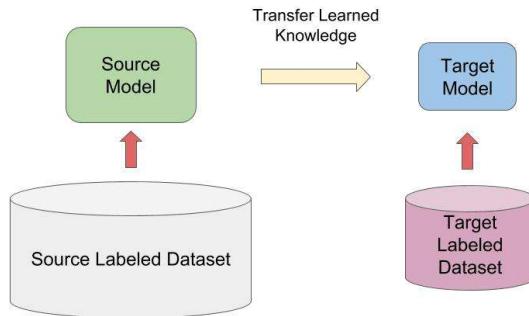


Figura 13.5: ULMfit: Ajuste fino de un modelo de lenguaje

Model	Test	Model	Test
CoVe (McCann et al., 2017)	8.2	CoVe (McCann et al., 2017)	4.2
IMDb oh-LSTM (Johnson and Zhang, 2016)	5.9	TBCNN (Mou et al., 2015)	4.0
Virtual (Miyato et al., 2016)	5.9	LSTM-CNN (Zhou et al., 2016)	3.9
ULMFIT (ours)	<b>4.6</b>	ULMFIT (ours)	<b>3.6</b>

Figura 13.6: Tasas de error de clasificación de un clasificador de texto

"modelado de lenguaje enmascarado"(MLM) durante el preentrenamiento.

En MLM, se enmascaran palabras aleatorias en una oración y el modelo se entrena para predecir esas palabras enmascaradas en función del contexto circundante. BERT también incorpora una tarea de "predicción de la siguiente oración", donde se alimentan pares de oraciones al modelo y aprende a predecir si la segunda oración sigue a la primera en el texto original.

Para el ajuste fino de BERT, se agrega una capa específica de la tarea sobre el modelo pre-entrenado y se entrena en un conjunto de datos etiquetados para la tarea objetivo. BERT logró resultados de vanguardia en el momento de su lanzamiento en tareas de procesamiento del lenguaje natural, incluyendo clasificación de oraciones, reconocimiento de entidades nombradas, respuestas a preguntas y más.

### 13.7 Modelado de Lenguaje Mascarado y Predicción de la Siguiente Oración

MLM implica enmascarar  $k\%$  de las palabras de entrada y luego predecir las palabras enmascaradas. Usualmente, se utiliza  $k = 15\%$ . Si se enmascaran muy pocas palabras, el entrenamiento se vuelve muy costoso. Si se enmascaran demasiadas palabras, se pierde contexto suficiente.

La predicción de la siguiente oración se utiliza para aprender las relaciones entre las oraciones. El modelo intenta predecir si la oración B es la siguiente en el texto original después de la oración A, o si es una oración aleatoria.

### 13.8 Codificación de pares de oraciones en BERT

BERT utiliza incrustaciones de tokens para representar palabras. Las palabras se dividen en unidades más pequeñas llamadas "word pieces"; cada "word piece" se asigna a una incrustación de

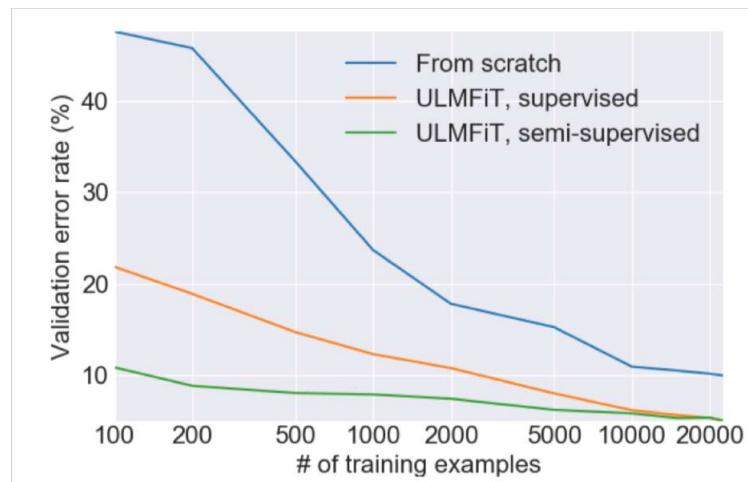


Figura 13.7: Transferencia de aprendizaje con ULMfit

token. BERT aprende una incrustación segmentada [SEP] para diferenciar entre las dos oraciones en un par. También utiliza incrustaciones posicionales para capturar la posición de cada palabra dentro de la oración.

## 13.9 Arquitectura y entrenamiento del modelo BERT

BERT se basa en el codificador Transformer. El bloque de atención propia de múltiples cabezas del Transformer permite a BERT considerar el contexto a larga distancia de manera efectiva. El uso de la atención propia también permite cálculos eficientes en GPU/TPU, con solo una multiplicación por capa. BERT se entrenó en una gran cantidad de datos de texto no etiquetado de Wikipedia y BookCorpus. Se entrenaron dos tamaños de modelo diferentes:

1. BERT-Base: 12 capas, 768 unidades ocultas y 12 cabezas de atención.
2. BERT-Large: 24 capas, 1024 unidades ocultas y 16 cabezas de atención.

El proceso de entrenamiento involucró el uso de configuraciones de TPU (Tensor Processing Unit) 4x4 o 8x8 para una computación más rápida. El entrenamiento de los modelos BERT tomó aproximadamente 4 días para completarse.

## 13.10 Ajuste fino del modelo BERT

El ajuste fino implica personalizar el modelo preentrenado de BERT para tareas específicas. Para el ajuste fino de BERT, se agrega una capa específica de la tarea sobre el modelo preentrenado de BERT. La capa específica de la tarea puede variar según la tarea en cuestión, como el etiquetado de secuencias o la clasificación de oraciones. Se entrena el modelo completo, incluido el modelo preentrenado de BERT y la capa específica de la tarea, para la tarea específica.

### 13.10.1 Resultados de BERT en tareas GLUE

BERT fue extremadamente popular y versátil, y el ajuste fino de BERT condujo a nuevos resultados de vanguardia en una amplia gama de tareas. El rendimiento de BERT se evaluó utilizando el benchmark GLUE, una colección de diversas tareas de procesamiento del lenguaje natural. El

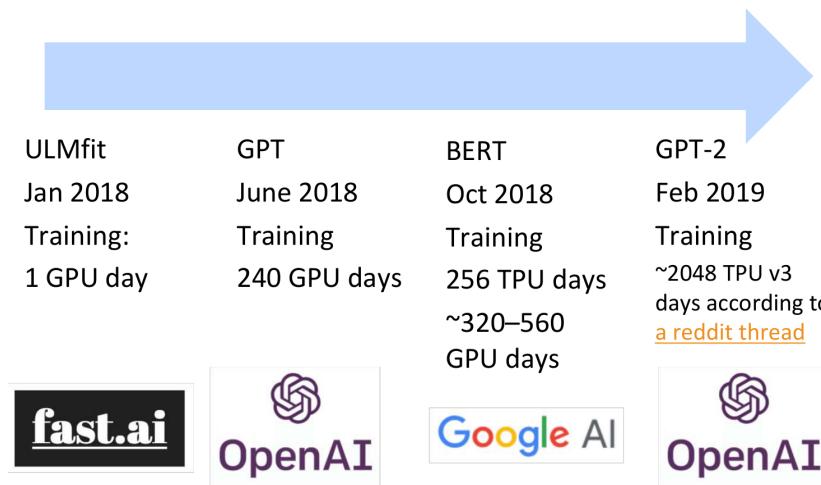


Figura 13.8: Aumentando la escala de los modelos de lenguaje grandes

benchmark GLUE consiste principalmente en tareas de inferencia de lenguaje natural, pero también incluye tareas de similitud de oraciones y análisis de sentimientos.

#### Tareas de GLUE

- QQP: Quora Question Pairs (detectar preguntas parafraseadas)
- QNLI: inferencia de lenguaje natural sobre datos de respuesta a preguntas
- SST-2: análisis de sentimientos
- CoLA: corpus de aceptabilidad lingüística (detecta si las frases son gramaticales.)
- STS-B: similitud semántica textual
- MRPC: corpus de paráfrasis de Microsoft
- RTE: pequeño corpus de inferencia en lenguaje natural

Ejemplo de tarea: MultiNLI (Inferencia de lenguaje natural)

- Premisa: "Las colinas y montañas son especialmente sagradas en el jainismo."
- Hipótesis: <sup>El</sup> jainismo odia la naturaleza."
- Etiqueta: Contradicción

Ejemplo de tarea: CoLa

- Oración: "La carreta resonó por el camino."
- Etiqueta: Aceptable
- Oración: <sup>El</sup> automóvil tocó la bocina por el camino."
- Etiqueta: Inaceptable

#### 13.10.2 Efecto de la tarea de preentrenamiento en BERT

#### 13.11 Decodificadores de preentrenamiento GPT y GPT-2

De manera contemporánea a BERT, OpenAI presentó un enfoque alternativo llamado Generative Pretrained Transformer (GPT) [Radford et al., 2018]. La idea detrás de GPT es entrenar un gran modelo de lenguaje estándar utilizando la parte generativa del Transformer, específicamente el decodificador. GPT es un decodificador Transformer con 12 capas y 117 millones de parámetros. Tiene estados ocultos de 768 dimensiones y capas ocultas de avance feed-forward de 3072 dimensiones.

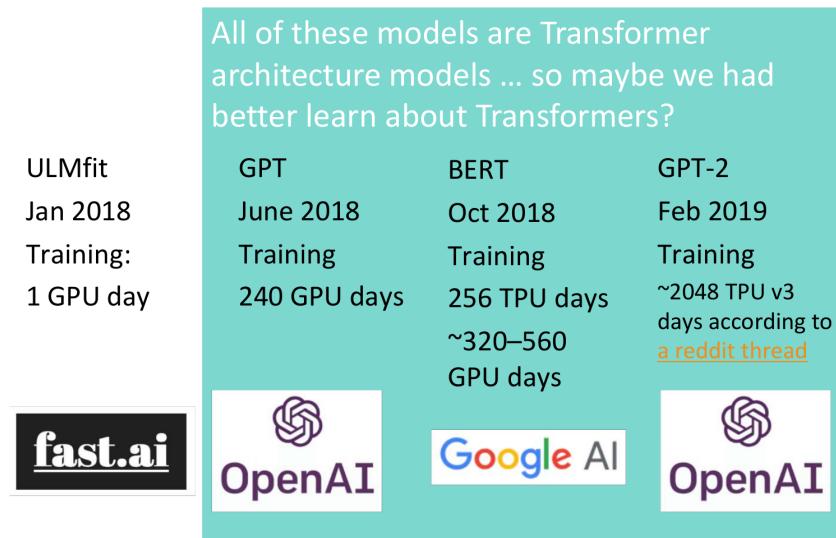


Figura 13.9: Evolución de los modelos de lenguaje grandes

GPT utiliza codificación de pares de bytes con 40,000 fusiones para manejar unidades de subpalabras. GPT fue entrenado en BooksCorpus, que consta de más de 7,000 libros únicos.

Posteriormente, OpenAI presentó GPT-2, una versión más grande con 1.5 mil millones de parámetros, entrenada con aún más datos. Se ha demostrado que GPT-2 genera muestras relativamente convincentes de lenguaje natural.

### Salida del modelo GPT-2 (seleccionada)

#### Inicio proporcionado por un humano:

En un descubrimiento impactante, los científicos descubrieron una manada de unicornios viviendo en un valle remoto previamente inexplicado, en las montañas de los Andes. Aún más sorprendente para los investigadores fue el hecho de que los unicornios hablaban un inglés perfecto.

#### Completado del modelo:

Los científicos nombraron a la población, debido a su distintivo cuerno, Unicornio de Ovidio. Estos unicornios plateado-blancos de cuatro cuernos eran desconocidos previamente para la ciencia.

Ahora, después de casi dos siglos, el misterio de lo que desencadenó este fenómeno peculiar finalmente se resuelve.

El Dr. Jorge Pérez, un biólogo evolutivo de la Universidad de La Paz, y varios compañeros, estaban explorando las montañas de los Andes cuando encontraron un pequeño valle, sin otros animales ni humanos. Pérez notó que el valle tenía lo que parecía ser una fuente natural, rodeada por dos picos de roca y nieve plateada.

## 13.12 ¿Qué tipos de cosas aprende el preentrenamiento?

- La Universidad de Stanford está ubicada en \_\_\_\_\_, California. [Datos curiosos]
- Puse el \_\_\_\_\_ tenedor en la mesa. [sintaxis]
- La mujer cruzó la calle, verificando el tráfico sobre \_\_\_\_\_ hombro. [co-referencia]
- Fui al océano para ver los peces, tortugas, focas y \_\_\_\_\_. [semántica léxica/tema]

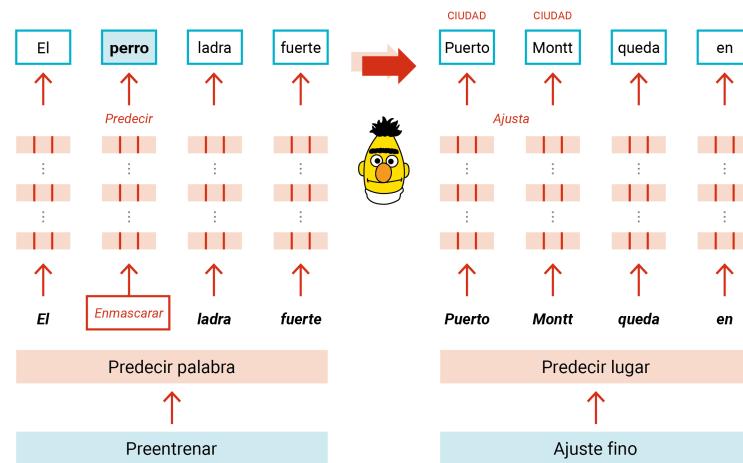


Figura 13.10: Ajuste fino del modelo BERT

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.9	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	88.1	91.3	45.4	80.0	82.3	56.0	75.2
BERT <sub>BASE</sub>	84.6/83.4	71.2	90.1	93.5	52.1	85.8	88.9	66.4	79.6
BERT <sub>LARGE</sub>	<b>86.7/85.9</b>	<b>72.1</b>	<b>91.1</b>	<b>94.9</b>	<b>60.5</b>	<b>86.5</b>	<b>89.3</b>	<b>70.1</b>	<b>81.9</b>

Figura 13.11: Resultados de BERT en tareas GLUE

- En general, el valor que obtuve de las dos horas viéndolo fue la suma total de las palomitas de maíz y la bebida. La película fue \_\_\_\_\_. [sentimiento]
- Iroh entró en la cocina para preparar un poco de té. De pie junto a Iroh, Zuko reflexionó sobre su destino. Zuko dejó el \_\_\_\_\_. [algún razonamiento - esto es más difícil]
- Estaba pensando en la secuencia que va 1, 1, 2, 3, 5, 8, 13, 21, \_\_\_\_\_. [aritmética básica; no aprenden la secuencia de Fibonacci]

### 13.13 Cambio de fase: GPT-3 (2020)

GPT-3 es otro modelo de lenguaje basado en Transformer (LM) que empujó los límites con casi 200 mil millones de parámetros, convirtiéndolo en el modelo más grande en ese momento [Brown et al., 2020]. Fue entrenado en un corpus masivo que consiste en casi 500 mil millones de palabras.

**Aprendizaje en contexto:** GPT-3 demostró la capacidad de resolver varias tareas de procesamiento del lenguaje natural (NLP) utilizando **aprendizaje sin ejemplos**, **aprendizaje con un solo ejemplo** y **aprendizaje con pocos ejemplos**. La clave de esta capacidad reside en la instrucción o contexto proporcionado a GPT-3. GPT-3 demostró la capacidad de resolver diversas tareas sin realizar actualizaciones de gradiente en el modelo base.

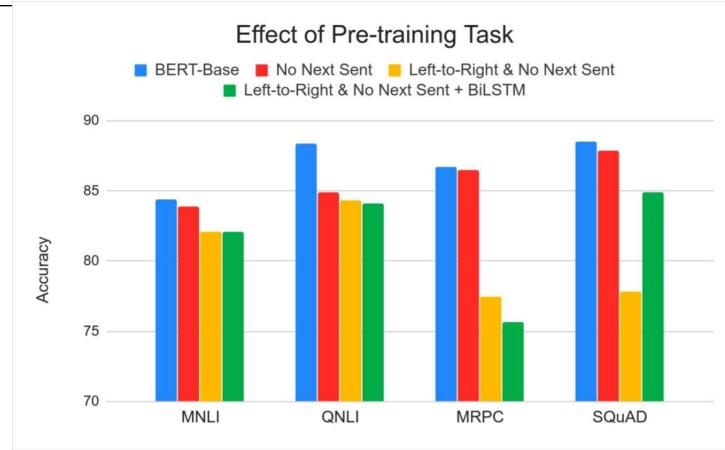


Figura 13.12: Efecto de la tarea de preentrenamiento en BERT



### 13.14 Aprendizaje sin ejemplos, con un solo ejemplo y con pocos ejemplos con GPT-3

GPT-3, uno de los modelos más destacados en el campo de los Modelos de Lenguaje Grandes (LLMs), ha demostrado la capacidad de realizar tareas de procesamiento de lenguaje natural (NLP) mediante aprendizaje sin ejemplos (zero-shot learning), aprendizaje con un solo ejemplo (one-shot learning) y aprendizaje con unos pocos ejemplos (few-shot learning).

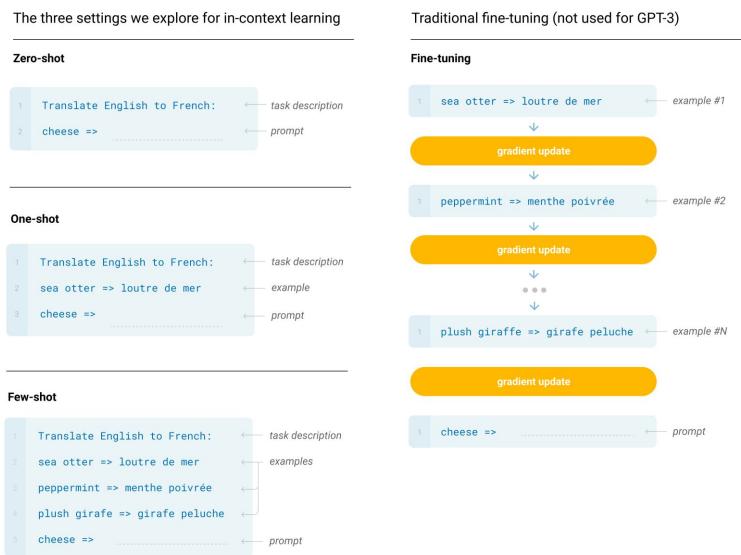
En el aprendizaje sin ejemplos, GPT-3 es capaz de abordar tareas sin ningún entrenamiento específico. Esto se logra proporcionando al modelo un prompt o una instrucción que guíe su proceso de generación. Por ejemplo, al proporcionarle una instrucción como "Traduce esta oración en inglés al francés", GPT-3 puede generar la oración traducida sin necesidad de un entrenamiento explícito para tareas de traducción.

En el aprendizaje con un solo ejemplo, GPT-3 puede realizar una tarea al agregar un solo par de entrada-salida a la instrucción. Por ejemplo, si se le proporciona un solo ejemplo de una pregunta y su respuesta, GPT-3 puede generar respuestas coherentes a preguntas similares.

En el aprendizaje con unos pocos ejemplos, se sigue una idea similar a la del aprendizaje con un solo ejemplo, pero proporcionando un número limitado de pares de entrada-salida después de la instrucción en el prompt. Estos ejemplos adicionales ayudan a GPT-3 a generalizar y mejorar su capacidad para abordar la tarea específica.

Estos enfoques de aprendizaje permiten que GPT-3 realice tareas de procesamiento de lenguaje natural sin necesidad de entrenamiento intensivo en cada tarea específica. Esto ha abierto nuevas posibilidades en el desarrollo de aplicaciones de NLP y ha demostrado el poder de los Modelos de Lenguaje Grandes en la resolución de diversas tareas de lenguaje.

GPT-3 Resultados en Few-shot learning



### 13.15 Chain-of-thought Prompting

Chain-of-thought prompting es un mecanismo simple para provocar un comportamiento de razonamiento de múltiples pasos en modelos de lenguaje grandes.

La idea detrás de chain-of-thought prompting es ampliar cada ejemplo en el prompt de few-shot learning con una cadena de pensamiento asociada a la respuesta. En lugar de proporcionar solo un par de entrada-salida, se agrega un contexto adicional que guía al modelo a través de una serie de pasos de razonamiento [Wei et al., 2022].

Por ejemplo, en lugar de simplemente proporcionar un ejemplo de pregunta y respuesta, se puede agregar una cadena de pensamiento que muestra cómo se llega a la respuesta paso a paso. Esto ayuda a los modelos de lenguaje a comprender mejor el proceso de razonamiento requerido para abordar la tarea.

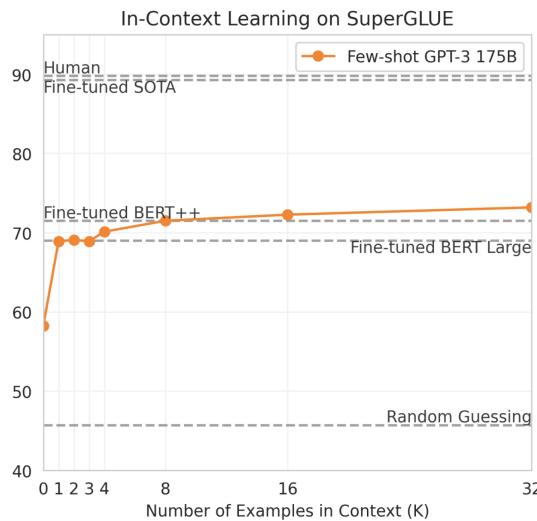
La idea detrás de chain-of-thought prompting es mejorar la capacidad de los modelos de lenguaje para realizar razonamientos más sofisticados y responder a preguntas que requieren múltiples pasos de inferencia. Al proporcionar un contexto más detallado y explicativo en el prompt, se espera que los modelos puedan generar respuestas más completas y coherentes.

Chain-of-thought prompting es una técnica prometedora que puede mejorar la capacidad de los modelos de lenguaje para realizar razonamientos complejos y abordar tareas de procesamiento de lenguaje natural que requieren múltiples pasos de inferencia. Su aplicación puede ampliar las capacidades de los modelos y permitir su uso en una variedad más amplia de aplicaciones de NLP.

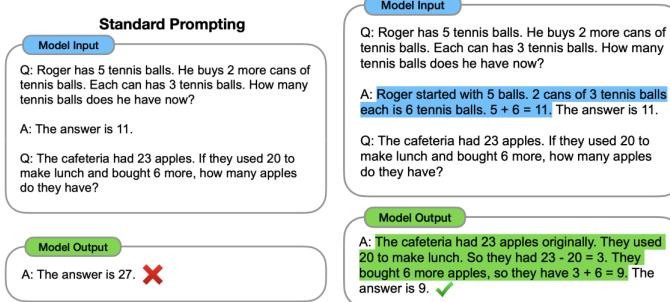
### 13.16 Modelos de Lenguaje como Asistentes de Usuario (o Chatbots)

Los Modelos de Lenguaje Grandes (LLMs) han demostrado ser herramientas poderosas para interactuar con los usuarios y actuar como asistentes de usuario o chatbots. Sin embargo, los LLMs autoregresivos no están alineados directamente con la intención del usuario y pueden generar respuestas que no cumplen con las expectativas.

Un problema común con los LLMs autoregresivos es que pueden producir respuestas vagas,



#### Chain-of-thought prompting



repetitivas o poco relevantes para las consultas de los usuarios. Esto se debe a la naturaleza generativa del modelo, que puede generar texto sin una comprensión profunda de la intención del usuario o el contexto de la conversación.

Una solución a este problema es el ajuste fino de los modelos de lenguaje para alinearlos con la intención del usuario. El ajuste fino implica entrenar el modelo en un conjunto de datos específico para mejorar su capacidad de generar respuestas relevantes y coherentes.

PROMPT	Explain the moon landing to a 6 year old in a few sentences.
COMPLETION	GPT-3 Explain the theory of gravity to a 6 year old. Explain the theory of relativity to a 6 year old in a few sentences. Explain the big bang theory to a 6 year old. Explain evolution to a 6 year old.

#### 13.16.1 LaMDA: Modelos de Lenguaje para Aplicaciones de Diálogo

LaMDA es un modelo de lenguaje desarrollado por Google basado en el Transformer, optimizado para diálogos en dominio abierto [Thoppilan et al., 2022]. Este modelo cuenta con 137 mil millones

de parámetros y se entrena con 1.56 mil millones de palabras.

El proceso de entrenamiento de LaMDA es similar al de los modelos de lenguaje tradicionales, donde se realiza un pre-entrenamiento para predecir palabras. Sin embargo, LaMDA se enfoca especialmente en datos de diálogos durante el pre-entrenamiento. Posteriormente, el modelo se ajusta finamente para generar respuestas teniendo en cuenta varios criterios.

Para lograr que LaMDA cumpla con todos estos criterios, se trabajó con un gran número de trabajadores de la multitud. Estas personas etiquetaron manualmente las conversaciones a partir del modelo pre-entrenado.

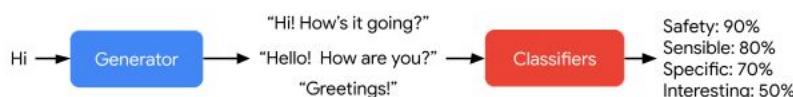


Figura 13.13: Arquitectura del modelo de lenguaje LaMDA.

El desarrollo de LaMDA se centró en cumplir con los siguientes criterios de optimización:

#### Criterios de Optimización de LaMDA

##### Calidad

- Sensatez: proporcionar respuestas significativas.
- Especificidad: evitar respuestas vagas.
- Interesante: proporcionar respuestas perspicaces, inesperadas o ingeniosas.

##### Seguridad

- Evitar lenguaje violento.
- Evitar discursos de odio.
- Evitar discursos estereotipados.

##### Basamento e Informatividad

- Evitar proporcionar respuestas no validadas por fuentes externas.
- Optimizar la fracción de respuestas que pueden ser validadas en fuentes autorizadas mediante el uso de motores de búsqueda.

LaMDA ha sido evaluado comparándolo con el modelo pre-entrenado original y con juicios humanos. Para llevar a cabo la evaluación, se empleó a un grupo independiente de personas que respondieron cuestionarios específicos.

Los resultados de la evaluación demuestran la mejora lograda con LaMDA en comparación con el modelo pre-entrenado original. El enfoque en los criterios de calidad, seguridad, basamento e informatividad ha permitido que LaMDA genere respuestas más precisas y relevantes en diálogos de dominio abierto.

#### 13.16.2 ChatGPT y RLHF

ChatGPT es un modelo desarrollado por OpenAI, similar a LaMDA, que también se centra en aplicaciones de diálogo. Fue lanzado a finales de 2022 y utiliza un enfoque de colaboración masiva para mejorar sus respuestas. Sin embargo, a diferencia de LaMDA, ChatGPT utiliza el Aprendizaje por Refuerzo a partir de Retroalimentación Humana (RLHF) en su proceso de ajuste fino [Ouyang et al., 2022].

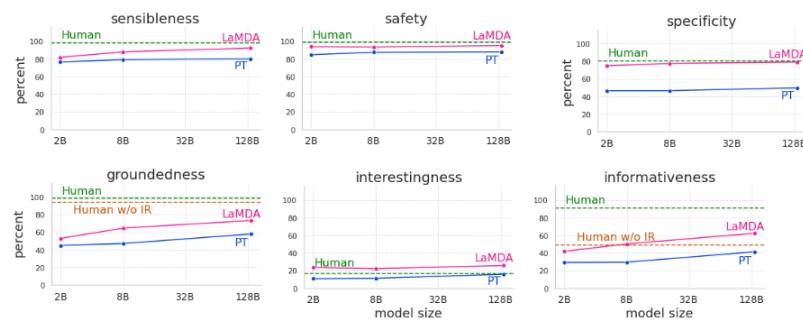
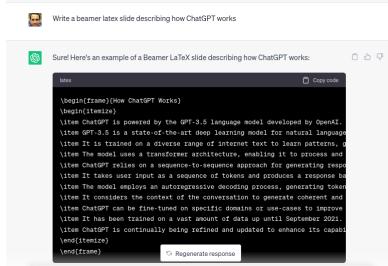


Figura 13.14: Resultados de la evaluación de LaMDA.

El Aprendizaje por Refuerzo (RL) es un paradigma de aprendizaje automático en el que un agente aprende a tomar decisiones en un entorno para maximizar una recompensa acumulativa. En el caso de ChatGPT, el modelo se ajusta finamente mediante RLHF utilizando una función de preferencia que asigna una puntuación a las respuestas generadas y ajusta el modelo de lenguaje en consecuencia.

El proceso de RLHF implica la recopilación de muestras de interacciones entre el modelo y los usuarios reales. Estas muestras son utilizadas para entrenar un modelo de clasificación que asigna una puntuación a las respuestas generadas. El modelo de lenguaje se ajusta entonces utilizando algoritmos de optimización por refuerzo para maximizar la recompensa esperada.

El enfoque RLHF tiene como objetivo mejorar la calidad y la relevancia de las respuestas generadas por ChatGPT al guiar el ajuste fino del modelo con la ayuda de la retroalimentación humana. Este proceso iterativo de ajuste y retroalimentación ayuda a refinar las respuestas del modelo y alinearlas mejor con las expectativas de los usuarios.

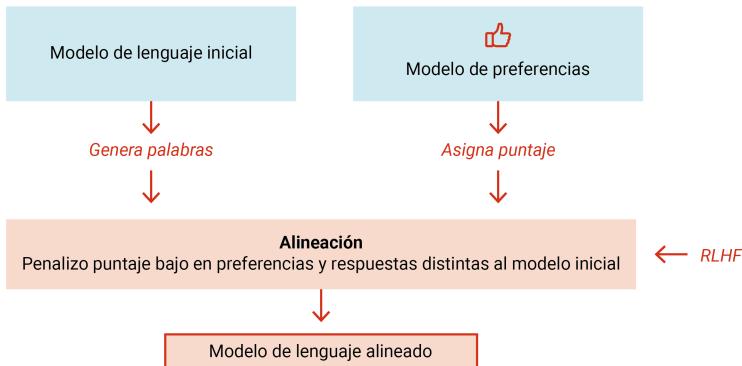


Source: <https://huggingface.co/blog/r1hf>

### 13.16.3 GPT-4 (2023)

GPT-4 [OpenAI, 2023] es el último modelo de lenguaje desarrollado por OpenAI. Fue lanzado en 2023 y representa una evolución significativa en comparación con sus predecesores. Una de las principales características de GPT-4 es su capacidad para incluir imágenes como parte de las instrucciones o el contexto de entrada.

GPT-4 sigue siendo un modelo de lenguaje basado en la arquitectura Transformer, pero con mejoras en la capacidad de procesamiento y generación de texto en combinación con imágenes. Esta mejora permite a GPT-4 comprender y responder a consultas que involucran información visual, como preguntas sobre el contenido de una imagen o la descripción de elementos visuales específicos.



Un aspecto destacado de GPT-4 es su capacidad para aprobar exámenes en varias disciplinas. Al poder procesar las imágenes asociadas con las preguntas, GPT-4 puede generar respuestas precisas y detalladas basadas tanto en información textual como visual.

Es importante destacar que, a partir de ChatGPT y GPT-4, las empresas han dejado de hacer públicos todos los detalles de la construcción de sus modelos.

Example of GPT-4 visual input	
User	What is funny about this image? Describe it panel by panel.
	
Source:	<a href="https://www.reddit.com/r/humor/comments/ob86vz/funny/">https://www.reddit.com/r/humor/comments/ob86vz/funny/</a>
GPT-4	The image shows a package for a "Lightning Cable" adapter with three panels.
	Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.
	Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.
	Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.
	The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.
Table 3.	Example prompt demonstrating GPT-4's visual input capability. The prompt consists of a question about an image with multiple panels which GPT-4 is able to answer.

### 13.17 Ajuste Fino de Instrucciones

Una forma más eficiente de ajustar los Modelos de Lenguaje Grandes (LLMs) es mediante el ajuste fino de instrucciones [Chung et al., 2022]. Este enfoque consiste en recopilar ejemplos de pares (instrucción, salida) para diversas tareas y ajustar el LLM en base a estos ejemplos.

La idea detrás del ajuste fino de instrucciones es utilizar ejemplos de instrucciones y las salidas esperadas correspondientes para enseñar al LLM a realizar tareas específicas de manera más precisa. Al recopilar una amplia variedad de ejemplos de diferentes tareas, se puede mejorar la capacidad del modelo para comprender y generar respuestas relevantes y coherentes.

Una vez que se ha realizado el ajuste fino de las instrucciones, se evalúa el rendimiento del modelo en tareas no vistas previamente. Esto permite medir la generalización del modelo y su

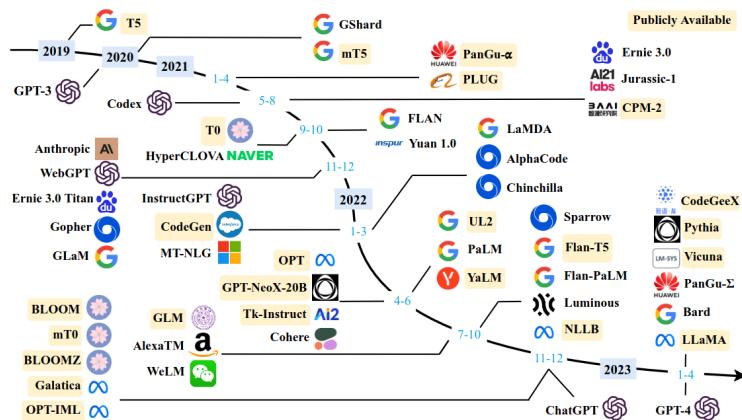
capacidad para abordar nuevas tareas más allá de las utilizadas durante el ajuste fino.

El ajuste fino de instrucciones es un enfoque prometedor para mejorar la eficiencia del ajuste fino en los LLMs. Al recopilar ejemplos de instrucciones y salidas esperadas en lugar de realizar ajustes finos en cada tarea por separado, se ahorra tiempo y recursos computacionales.

### 13.18 Línea de tiempo de los Modelos de Lenguaje Grandes

A día de hoy (2023), el desarrollo de nuevos Modelos de Lenguaje Grandes continúa sin interrupciones. Estos modelos han experimentado un crecimiento y avance acelerado en los últimos años, superando constantemente los límites anteriores en términos de tamaño y capacidad.

La siguiente es una línea de tiempo que muestra algunos de los modelos de lenguaje grandes existentes en los últimos años, que tienen un tamaño superior a los 10 mil millones de parámetros [Zhao et al., 2023]. Esta línea de tiempo destaca los avances significativos que se han logrado en este campo y cómo los modelos han evolucionado a lo largo del tiempo.



### 13.19 Prompt Engineering

La ingeniería de prompts es una nueva disciplina para desarrollar y optimizar prompts que aprovechen eficientemente los modelos de lenguaje (LMs). Consiste en diseñar instrucciones o contextos específicos para guiar el comportamiento de los modelos y obtener resultados deseados.

En el caso de los modelos de lenguaje generativos, como GPT-3, el prompt es crucial para influir en la generación de texto. Un prompt bien diseñado puede ayudar a obtener respuestas más precisas y coherentes, alineadas con la intención del usuario.

Existen diferentes técnicas de ingeniería de prompts. Algunas estrategias comunes incluyen:

- **Prompt priming:** proporcionar un contexto inicial que establezca el tema o la dirección de la conversación. Por ejemplo, si se desea obtener información sobre el clima en una ubicación específica, el prompt podría comenzar con "Dime cómo estará el clima en".
- **Prompt expansion:** ampliar el prompt con información adicional o detalles relevantes para obtener respuestas más precisas. Por ejemplo, si se busca información sobre un libro, se podría incluir el título, el autor y el género en el prompt.
- **Prompt rewriting:** reformular el prompt para hacerlo más claro y específico. Esto puede implicar simplificar la pregunta, eliminar ambigüedades o especificar los detalles requeridos.

- **Prompt combination:** combinar múltiples prompts o preguntas en una sola para obtener respuestas más completas o enriquecedoras. Esto puede ser útil cuando se busca información detallada sobre un tema específico.

La ingeniería de prompts es un proceso iterativo que requiere experimentación y ajustes para lograr los resultados deseados. Es importante comprender el comportamiento del modelo y cómo responde a diferentes tipos de prompts para obtener los mejores resultados.



## 13.20 Peligros de los Grandes Modelos de Lenguaje

A medida que los Modelos de Lenguaje Grandes (LLMs) continúan evolucionando y volviéndose más poderosos, la comunidad de investigación ha planteado preocupaciones sobre varios peligros asociados a estos modelos [Bender et al., 2021].

- **Alucinación:** Los modelos de lenguaje probabilísticos pueden generar información fabricada sin una base factual. Esto significa que pueden generar respuestas que no están respaldadas por datos verificables o información precisa.
- **Equidad:** Los LLMs pueden perpetuar los sesgos presentes en los datos de entrenamiento, lo que incluye lenguaje tóxico, racismo y discriminación de género. Estos sesgos pueden llevar a respuestas sesgadas y perjudiciales para ciertos grupos de personas.
- **Infracción de derechos de autor:** Los LLMs grandes pueden violar las leyes de derechos de autor al reproducir contenido sin la autorización adecuada. Esto plantea preocupaciones legales y éticas en relación con la propiedad intelectual.
- **Falta de transparencia:** La naturaleza compleja de los LLMs dificulta la interpretación de sus predicciones y el entendimiento del razonamiento detrás de respuestas específicas. Esto puede generar desconfianza en los resultados y dificultar la rendición de cuentas.
- **Monopolización:** El costo alto de entrenar estos modelos crea barreras para que las empresas que no son de tecnología puedan competir en el mismo nivel. Esto puede llevar a una concentración de poder en manos de unas pocas grandes empresas de tecnología.
- **Huella de carbono elevada:** El proceso de entrenamiento intensivo en energía de los LLMs contribuye a una huella de carbono significativa. Esto plantea preocupaciones sobre el impacto ambiental y la sostenibilidad de estos modelos a gran escala.

Estas preocupaciones resaltan la importancia de abordar los desafíos éticos, legales y sociales asociados con los LLMs. Se necesita una regulación adecuada, transparencia en la investigación y desarrollo, y enfoques responsables para garantizar que los LLMs se utilicen de manera ética y beneficiosa para la sociedad en general.

## 13.21 Conclusiones

El crecimiento en tamaño y potencia de los Modelos de Lenguaje Grandes ha acelerado de manera significativa en los últimos años. Estos modelos han demostrado capacidades impresionantes en diversas tareas de procesamiento de lenguaje natural y han abierto nuevas posibilidades en áreas como la generación de texto, la traducción automática, los chatbots y mucho más.

A medida que avanzamos, es difícil predecir con certeza qué nos deparará el futuro en términos de desarrollo de modelos de lenguaje. Sin embargo, hay algunas tendencias que se pueden anticipar con confianza.

En primer lugar, es probable que veamos una proliferación de modelos generativos para diversos formatos, como texto, código, imágenes, video y realidades virtuales. Los Modelos de Lenguaje Grandes seguirán siendo el centro de atención en la generación de contenido en estos formatos.

Además, habrá una gran cantidad de agentes y programas que interactuarán y tomarán decisiones basadas en la interacción con estos modelos. Esto podría incluir asistentes médicos, programas de inversión, agentes de viajes y muchos otros sistemas que utilizan modelos de lenguaje para comprender y responder a las necesidades de los usuarios.

Si bien los avances en los Modelos de Lenguaje Grandes son emocionantes, también es importante abordar los desafíos y peligros asociados. Es fundamental tener en cuenta la ética, la transparencia, la equidad y la sostenibilidad al desarrollar y utilizar estos modelos para garantizar que se beneficien a todos los usuarios y la sociedad en general.





## Bibliografía

- [Amir et al., 2015] Amir, S., Ling, W., Astudillo, R., Martins, B., Silva, M. J., and Trancoso, I. (2015). Inesc-id: A regression model for large scale twitter sentiment lexicon induction. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 613–618, Denver, Colorado. Association for Computational Linguistics.
- [Baroni et al., 2014] Baroni, M., Dinu, G., and Kruszewski, G. (2014). Don’t count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 238–247. Association for Computational Linguistics.
- [Bender, 2013] Bender, E. M. (2013). Linguistic fundamentals for natural language processing: 100 essentials from morphology and syntax. *Synthesis lectures on human language technologies*, 6(3):1–184.
- [Bender et al., 2021] Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623.
- [Bengio, 2012] Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer.
- [Bengio et al., 2000] Bengio, Y., Ducharme, R., and Vincent, P. (2000). A neural probabilistic language model. *Advances in neural information processing systems*, 13.
- [Bikel et al., 1999] Bikel, D. M., Schwartz, R. M., and Weischedel, R. M. (1999). An algorithm that learns what’s in a name. *Mach. Learn.*, 34(1-3):211–231.
- [Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.
- [Blei et al., 2003] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022.

- [Bojanowski et al., 2016] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016). Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.
- [Brown et al., 1992] Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479.
- [Brown et al., 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- [Cho, 2015] Cho, K. (2015). Natural language understanding with distributed representation. *arXiv preprint arXiv:1511.07916*.
- [Cho et al., 2015] Cho, K., Courville, A., and Bengio, Y. (2015). Describing multimedia content using attention-based encoder-decoder networks. *IEEE Transactions on Multimedia*, 17(11):1875–1886.
- [Cho et al., 2014] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- [Chomsky, 2009] Chomsky, N. (2009). Syntactic structures. In *Syntactic Structures*. De Gruyter Mouton.
- [Chung et al., 2022] Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., Valter, D., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E. H., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q. V., and Wei, J. (2022). Scaling instruction-finetuned language models.
- [Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.
- [Collins, 2013] Collins, M. (2013). Statistical nlp lecture notes. *Lecture Notes*.
- [Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537.
- [Conneau et al., 2017] Conneau, A., Schwenk, H., Barrault, L., and Lecun, Y. (2017). Very deep convolutional networks for text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 1107–1116.
- [Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.
- [Deng and Liu, 2018] Deng, L. and Liu, Y. (2018). *Deep Learning in Natural Language Processing*. Springer.

- [Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.
- [Eisenstein, 2018] Eisenstein, J. (2018). Natural language processing. Technical report, Georgia Tech.
- [Felbo et al., 2017] Felbo, B., Mislove, A., Søgaard, A., Rahwan, I., and Lehmann, S. (2017). Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1615–1625.
- [Firth, 1957] Firth, J. (1957). A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, pages 10–32.
- [Fort, 2016] Fort, K. (2016). *Collaborative annotation for reliable natural language processing: Technical and sociological aspects*. John Wiley & Sons.
- [Fromkin et al., 2018] Fromkin, V., Rodman, R., and Hyams, N. (2018). *An introduction to language*. Cengage Learning.
- [Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.
- [Goldberg, 2016] Goldberg, Y. (2016). A primer on neural network models for natural language processing. *J. Artif. Intell. Res.(JAIR)*, 57:345–420.
- [Goldberg, 2017] Goldberg, Y. (2017). Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309.
- [Goldberg and Levy, 2014] Goldberg, Y. and Levy, O. (2014). word2vec explained: deriving mikolov et al.’s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.
- [Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.
- [Goodman, 2001] Goodman, J. T. (2001). A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434.
- [Harris, 1954] Harris, Z. (1954). Distributional structure. *Word*, 10(23):146–162.
- [Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- [Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

- [Howard and Ruder, 2018] Howard, J. and Ruder, S. (2018). Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia. Association for Computational Linguistics.
- [Huang et al., 2015] Huang, Z., Xu, W., and Yu, K. (2015). Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*.
- [Johnson, 2014] Johnson, M. (2014). Introduction to computational linguistics and natural language processing (slides). 2014 Machine Learning Summer School.
- [Jozefowicz et al., 2015] Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350.
- [Jurafsky and Martin, 2023] Jurafsky, D. and Martin, J. H. (2023). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Upper Saddle River, NJ, USA, 3rd (draft) edition.
- [Kenton and Toutanova, 2019] Kenton, J. D. M.-W. C. and Toutanova, L. K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186.
- [Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- [Kupiec, 1992] Kupiec, J. (1992). Robust part-of-speech tagging using a hidden markov model. *Computer speech & language*, 6(3):225–242.
- [Lafferty et al., 2001] Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Brodley, C. E. and Danyluk, A. P., editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289. Morgan Kaufmann.
- [LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- [Levy and Goldberg, 2014] Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185.
- [Li et al., 2015] Li, J., Luong, M.-T., Jurafsky, D., and Hovy, E. (2015). When are tree structures necessary for deep learning of representations? *arXiv preprint arXiv:1503.00185*.
- [Manning et al., 2008] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.
- [McCallum et al., 2000] McCallum, A., Freitag, D., and Pereira, F. C. (2000). Maximum entropy markov models for information extraction and segmentation. In *Icml*, volume 17, pages 591–598.

- [McCallum et al., 1998] McCallum, A., Nigam, K., et al. (1998). A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI.
- [Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc.
- [Mohammad et al., 2013] Mohammad, S. M., Kiritchenko, S., and Zhu, X. (2013). Nrc-canada: Building the state-of-the-art in sentiment analysis of tweets. *Proceedings of the seventh international workshop on Semantic Evaluation Exercises (SemEval-2013)*.
- [Mosteller and Wallace, 1963] Mosteller, F. and Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, 58(302):275–309.
- [Nakov et al., 2013] Nakov, P., Rosenthal, S., Kozareva, Z., Stoyanov, V., Ritter, A., and Wilson, T. (2013). Semeval-2013 task 2: Sentiment analysis in twitter. In *Proceedings of the seventh international workshop on Semantic Evaluation Exercises*, pages 312–320, Atlanta, Georgia, USA. Association for Computational Linguistics.
- [Nesterov, 2018] Nesterov, Y. (2018). *Lectures on convex optimization*, volume 137. Springer.
- [OpenAI, 2023] OpenAI (2023). Gpt-4 technical report.
- [Ouyang et al., 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.
- [Page et al., 1998] Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The pagerank citation ranking: Bring order to the web. Technical report, Technical report, stanford University.
- [Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543.
- [Peters et al., 2018] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana. Association for Computational Linguistics.
- [Polyak, 1964] Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.
- [Pustejovsky and Stubbs, 2012] Pustejovsky, J. and Stubbs, A. (2012). *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications*. "O'Reilly Media, Inc.".

- [Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.
- [Ramshaw and Marcus, 1999] Ramshaw, L. A. and Marcus, M. P. (1999). *Text Chunking Using Transformation-Based Learning*, pages 157–176. Springer Netherlands, Dordrecht.
- [Read, 2005] Read, J. (2005). Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL Student Research Workshop*, ACLstudent '05, pages 43–48, Stroudsburg, PA, USA. Association for Computational Linguistics.
- [Salton et al., 1975] Salton, G., Wong, A., and Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620.
- [Severyn and Moschitti, 2015] Severyn, A. and Moschitti, A. (2015). Twitter sentiment analysis with deep convolutional neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 959–962, New York, NY, USA. ACM.
- [Shannon, 1951] Shannon, C. E. (1951). Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64.
- [Snow et al., 2008] Snow, R., O’connor, B., Jurafsky, D., and Ng, A. Y. (2008). Cheap and fast—but is it good? evaluating non-expert annotations for natural language tasks. In *Proceedings of the 2008 conference on empirical methods in natural language processing*, pages 254–263.
- [Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.
- [Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147.
- [Tang et al., 2015] Tang, D., Qin, B., and Liu, T. (2015). Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432.
- [Tang et al., 2014] Tang, D., Wei, F., Qin, B., Zhou, M., and Liu, T. (2014). Building large-scale twitter-specific sentiment lexicon : A representation learning approach. In *COLING 2014, 25th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, August 23-29, 2014, Dublin, Ireland*, pages 172–182.
- [Thoppilan et al., 2022] Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. (2022). Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.
- [Tibshirani, 1996] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

- [Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.
- [Turian et al., 2010] Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics.
- [Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.
- [Wei et al., 2022] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. (2022). Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.
- [Yule, 2016] Yule, G. (2016). *The study of language*. Cambridge university press.
- [Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.
- [Zhao et al., 2023] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. (2023). A survey of large language models.
- [Zipf, 1935] Zipf, G. K. (1935). *The Psychobiology of Language*. Houghton-Mifflin, New York, NY, USA.
- [Zou and Hastie, 2005] Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320.