



Universidad de Chile

Departamento de Ciencias de la Computación

Procesamiento de Lenguaje Natural

Apuntes de Clases

Felipe Bravo-Márquez

4 de julio de 2023

Índice general

Preface	1
1. Introducción	3
1.1. PLN y Lingüística Computacional	3
1.2. Niveles de descripción lingüística	5
1.2.1. Fonética	5
1.2.2. Fonología	5
1.2.3. Morfología	5
1.2.4. Sintaxis	6
1.2.5. Semántica	6
1.2.6. Pragmática	7
1.3. Procesamiento del Lenguaje Natural y Aprendizaje Automático	8
1.4. Desafíos del Lenguaje	9
1.5. Ejemplo de tareas NLP	9
1.5.1. Lingüística y Procesamiento del Lenguaje Natural (PNL)	11
1.6. Desafíos en el Procesamiento del Lenguaje Natural (PNL) . . .	12
1.7. Estudio de caso: Clasificación de sentimientos en tweets	13
1.8. Ingeniería de características y Aprendizaje Profundo	14
1.9. Historia	15
1.10. Conclusiones	16
2. Modelo de Espacio Vectorial y Recuperación de Información	17
2.1. Tokens y Tipos	17
2.1.1. Ley de Zipf	19
2.1.2. Listas de publicaciones y el índice invertido	20
2.1.3. Motores de búsqueda web	20
2.2. El modelo de espacio vectorial	21
2.2.1. Similitud entre vectores	23
2.3. Agrupamiento de Documentos	24
2.3.1. K-Means	25
2.4. Conclusiones y Conceptos Adicionales	25

3. Modelos de Lenguaje Probabilísticos	27
3.1. El Problema del Modelado del Lenguaje	27
3.1.1. ¿Por qué queríamos hacer esto?	28
3.1.2. Los Modelos de Lenguaje son Generativos	29
3.2. ¿Por qué los modelos de lenguaje son importantes?	29
3.2.1. Un Método Ingenuo	30
3.3. Procesos de Markov	31
3.3.1. Modelado de secuencias de longitud variable	31
3.4. Modelos de lenguaje trigram	32
3.4.1. El problema de estimación trigram	32
3.5. Evaluación de un modelo de lenguaje: Perplejidad	33
3.5.1. El trade-off entre sesgo y varianza	34
3.5.2. Estimación de máxima verosimilitud y overfitting	34
3.5.3. Técnicas de regularización	35
3.6. Interpolación Lineal	35
3.7. Estimación de los Valores λ	36
3.8. Métodos de Descuento	36
3.8.1. Modelos de Katz Back-Off (Bigramas)	37
3.9. Resumen	39
4. Text Classification and Naïve Bayes	41
4.1. Clasificación de texto: Definición	43
4.1.1. Métodos de clasificación: Reglas codificadas a mano	44
4.1.2. Métodos de clasificación: Aprendizaje automático supervisado	44
4.1.3. Problemas de aprendizaje supervisado	44
4.1.4. Modelos generativos	45
4.1.5. Clasificación con Modelos Generativos	45
4.2. Intuición del Bayes Ingenuo	45
4.2.1. Aplicación de la Regla de Bayes a Documentos y Clases	45
4.3. Clasificador Bayes Ingenuo	46
4.3.1. Suposiciones de Independencia del Bayes Ingenuo Multinomial	47
4.3.2. Clasificador Bayes Ingenuo Multinomial	47
4.3.3. Aplicación de los clasificadores Naive Bayes multinomiales a la clasificación de texto	48
4.3.4. Problemas al multiplicar muchas probabilidades	48
4.3.5. Aprendizaje del modelo Naive Bayes multinomial	48
4.3.6. Estimación de parámetros	49
4.3.7. Probabilidades cero y el problema de las palabras no vistas	49
4.3.8. Suavizado Laplaciano (Add-1) para Naïve Bayes	50
4.3.9. Naïve Bayes multinomial: aprendizaje	51
4.3.10. Palabras desconocidas	51
4.4. Ejemplo	51
4.5. Naïve Bayes como modelo de lenguaje	52
4.6. Evaluation	53

4.6.1.	The 2-by-2 Confusion Matrix	53
4.6.2.	Evaluation: Accuracy	54
4.6.3.	Evaluation: Precision and Recall	54
4.6.4.	Why Precision and Recall?	54
4.6.5.	A Combined Measure: F-measure	55
4.6.6.	Development Test Sets ("Devsets")	55
4.6.7.	Cross-validation: Multiple Splits	55
4.6.8.	Confusion Matrix for 3-class classification	57
4.6.9.	Macroaveraging and Microaveraging	57
5.	Modelos Lineales	59
5.1.	Supervised Learning	59
5.1.1.	Parameterized Functions	59
5.2.	Linear Models	60
5.2.1.	Example: Language Detection	60
5.3.	Log-linear Binary classification	62
5.4.	Multi-class Classification	64
5.5.	Representations	64
5.6.	One-Hot Vector Representation	66
5.7.	Log-linear Multi-class Classification	67
5.8.	Training	67
5.8.1.	Gradient-based Optimization	68
5.8.2.	Online Stochastic Gradient Descent	69
5.8.3.	Mini-batch Stochastic Gradient Descent	70
5.8.4.	Loss Functions	70
5.9.	Regularization	72
5.9.1.	L_2 Regularization	73
5.9.2.	L_1 Regularization	73
5.9.3.	Elastic-Net	73
5.10.	Beyond SGD	74
5.11.	Train, Test, and Validation Sets	74
5.12.	A limitation of linear models: the XOR problem	75
5.12.1.	Nonlinear input transformations	76
5.12.2.	Trainable mapping functions	77
6.	Redes Neuronales	79
6.1.	Feedforward Network Neural Networks	79
6.1.1.	Neural Networks as Mathematical Functions	81
6.2.	Representation Power	81
6.3.	Activation Functions	82
6.3.1.	Practical Issues	84
6.4.	Embedding Layers	84
6.4.1.	Dense Vectors vs. One-hot representations	86
6.5.	Neural Network Training	88
6.6.	Derivative Chain Rule Recap	88
6.7.	Backpropagation	90

6.8.	The Computation Graph Abstraction	93
6.8.1.	Forward Computation	95
6.8.2.	Backward Computation (Backprop)	96
6.8.3.	Summary of the Computation Graph Abstraction	97
6.8.4.	Derivatives of “non-mathematical” functions	97
6.9.	Regularization and Dropout	97
6.10.	Deep Learning Frameworks	98
7.	Vectores de Palabra	99
7.1.	Distributional Vectors	99
7.1.1.	Word-context Matrices	99
7.1.2.	PPMI	100
7.1.3.	Distributed Vectors or Word embeddings	101
7.2.	Word2Vec	102
7.2.1.	Skip-gram Model	102
7.2.2.	Parametrization of the Skip-gram model	104
7.2.3.	Skip-gram with Negative Sampling	105
7.2.4.	Continuous Bag of Words: CBOW	106
7.2.5.	GloVe	106
7.3.	Word Analogies	107
7.4.	Evaluation	107
7.5.	Correspondence between Distributed and Distributional Models	108
7.6.	FastText	108
7.7.	Sentiment-Specific Phrase Embeddings	109
7.8.	Gensim	110
8.	Etiquetado de Secuencias	111
8.1.	Overview	111
8.2.	Sequence Labeling or Tagging Tasks	111
8.3.	Part-of-Speech Tagging	112
8.4.	Named Entity Recognition (NER)	112
8.4.1.	Sequence Labeling as Supervised Learning	114
8.4.2.	Generative Approach for Sequence Labeling	114
8.5.	Hidden Markov Models	114
8.6.	Trigram Hidden Markov Models (Trigram HMMs)	115
8.6.1.	Parameters of the Model	115
8.7.	Independence Assumptions in Trigram HMMs	116
8.8.	Why the Name?	117
8.9.	Smoothed Estimation	117
8.10.	Dealing with Low-Frequency Words	118
8.11.	Decoding Problem	119
8.11.1.	Naive Brute Force Method	119
8.12.	Viterbi Decoding Dynamic Programming	120
8.13.	The Viterbi Algorithm	121
8.13.1.	A Recursive Definition	122
8.13.2.	The Viterbi Algorithm	123

8.13.3. The Viterbi Algorithm with Backpointers	124
8.13.4. The Viterbi Algorithm: Running Time	124
8.14. MEMMs	125
8.15. Example of Features used in Part-of-Speech Tagging	126
8.16. Feature Templates	127
8.17. MEMMs and Multi-class Softmax	127
8.18. Training MEMMs	128
8.19. Decoding with MEMMs	129
8.20. Comparison between MEMMs and HMMs	130
8.21. Conditional Random Fields (CRFs)	131
8.22. Decoding with CRFs	133
8.23. Parameter Estimation in CRFs (training)	134
8.24. CRFs and MEMMs	134
8.24.1. CRFs and MEMMs: the label bias problem	134
8.25. Links	135
9. Redes Neuronales Convolucionales	137
9.1. Basic Convolution + Pooling	137
9.2. 1D Convolutions over Text	138
9.3. Narrow vs. Wide Convolutions	139
9.4. Vector Pooling	140
9.5. Twitter Sentiment Classification with CNN	141
9.6. Very Deep Convolutional Networks for Text Classification	142
10. Redes Neuronales Recurrentes	143
10.1. The RNN Abstraction	143
10.2. Elman Network or Simple-RNN	146
10.3. RNN Training	146
10.4. RNN usage-patterns: Acceptor	147
10.5. RNN usage-patterns: Transducer	148
10.6. Bidirectional RNNS (BIRNN)	149
10.7. Multi-layer (stacked) RNNS	151
10.8. Gated Architectures	151
10.9. LSTM	154
10.10. GRU	156
10.11. Sentiment Classification with RNNs	157
10.12. Twitter Sentiment Classification with LSTMS Emojis	158
10.13. Bi-LSTM CRF	159
11. Modelos Secuencia a Secuencia y Atención Neuronal	163
11.1. Language Models and Language Generation	163
11.2. Sequence to Sequence Problems	164
11.2.1. Conditioned Generation	164
11.3. Decoding Approaches	166
11.3.1. Beam Search	166
11.4. Conditioned Generation with Attention	168

11.5. Attention and Word Alignments	170
11.6. Other types of Attention	170
12. Arquitectura de Transformer	173
12.1. What's Wrong with RNNs?	173
12.1.1. Dependencies in neural machine translations	173
12.2. The Transformer	174
12.3. Attention Mechanism in the Transformer	176
12.3.1. Queries	177
12.3.2. Keys	177
12.3.3. Values	178
12.3.4. Scaled Dot Product Attention	178
12.4. The Encoder	179
12.5. Self-Attention at a High Level	181
12.6. Self-Attention in Detail	181
12.7. Matrix Calculation of Self-Attention	184
12.8. Multi-Head Attention	185
12.9. Residual Connections	188
12.10. The Encoder: summary	188
12.11. The Decoder	189
12.12. The Final Linear and Training	191
12.13. Positional Encodings	193
12.14. Conclusions	194
13. Grandes Modelos de Lenguaje	195
13.1. Representations for a word	195
13.2. Neural Language Models can produce Contextualized Embeddings	195
13.3. ELMo: Embeddings from Language Models	196
13.3.1. ELMo: Use with a task	196
13.4. ULMfit	197
13.4.1. ULMfit emphasizes	197
13.4.2. ULMfit transfer learning	198
13.5. Let's scale it up!	198
13.6. BERT (Bidirectional Encoder Representations from Transformers)	198
13.7. Masked Language Modeling and Next Sentence Prediction	200
13.8. BERT sentence pair encoding	200
13.9. BERT Model Architecture and Training	200
13.10. BERT model fine tuning	201
13.10.1. BERT results on GLUE tasks	202
13.10.2. BERT Effect of pre-training task	203
13.11. Pre-training decoders GPT and GPT-2	203
13.12. What kinds of things does pretraining learn?	205
13.13. Phase Change: GPT-3 (2020)	206
13.14. Zero-shot, One-shot, and Few-shot Learning with GPT-3	206
13.15. Chain-of-thought Prompting	206

13.16	Language Models as User Assitants (or Chatbots)	207
13.16.1.	LaMDA: Language Models for Dialog Applications . . .	207
13.17	ChatGPT and RLHF	210
13.18	GPT-4 (2023)	211
13.19	Instruction Fine-tuning	211
13.20	Dangers of Large Language Models	211
13.21	Large Language Models Time-line	212
13.22	Prompt Engineering	212
13.23	Conclusions	213

Índice de cuadros

2.1. Matriz tf-idf	24
------------------------------	----

X

Índice de figuras

1.1.	Reconocimiento de Entidades Nombradas	3
2.1.	Ley de Zipf	20
2.2.	Índice invertido	20
2.3.	Los diversos componentes de un motor de búsqueda web [Manning et al., 2008].	21
2.4.	Similitud del coseno.	23
2.5.	Conjunto de documentos donde los grupos se pueden identificar claramente.	25
2.6.	Algoritmo K-means	26
4.1.	James Madison	42
4.2.	Alexander Hamilton	42
10.1.	Acceptor RNN training graph.	148
10.2.	Transducer RNN training graph.	149
11.1.	Source: [Cho et al., 2015]	171

Prefacio

Este apunte pretende ofrecer una introducción completa al Procesamiento del Lenguaje Natural (PLN), cubriendo los conceptos esenciales. Nos esforzamos 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 las modernas redes neuronales profundas, como los vectores de palabras, las redes neuronales recurrentes (RNN) y los transformers.

El material del curso procede de diversas fuentes. Los temas sobre redes neuronales se basan principalmente en el libro *Neural Network Methods for Natural Language Processing* de Goldberg. Los temas no relacionados con las redes neuronales, como los modelos probabilísticos del lenguaje, Naive Bayes y HMM, proceden del curso de Michael Collins y del libro de Dan Jurafsky. Además, algunas capítulos se han adaptado de tutoriales en línea y otros cursos, como el curso de Stanford de Manning.

Capítulo 1

Introducción

El volumen de datos textuales digitalizados que se genera cada día es enorme (por ejemplo, la web, redes sociales, registros médicos, libros digitalizados). Por lo tanto, también crece la necesidad de traducir, analizar y gestionar esta avalancha de palabras y texto.

El procesamiento del lenguaje natural (PLN) es el campo que se encarga de diseñar métodos y algoritmos que toman como entrada o producen como salida datos de **lenguaje natural** no estructurado [Goldberg, 2017]. El PLN se centra en el diseño y análisis de algoritmos computacionales y representaciones para procesar el lenguaje humano [Eisenstein, 2018].

Una tarea común de PLN es el Reconocimiento de Entidades Nombradas (NER, por sus siglas en inglés). Por ejemplo:

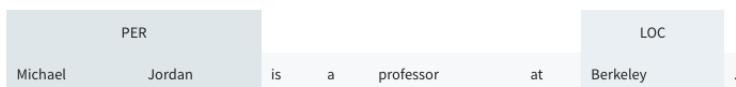


Figura 1.1: Reconocimiento de Entidades Nombradas

El lenguaje humano es altamente ambiguo, como en las frases: "Comí pizza con amigos", "Comí pizza con aceitunas.", "Comí pizza con un tenedor". Además, el lenguaje está en constante cambio y evolución, como ocurre con los hashtags en Twitter.

1.1. PLN y Lingüística Computacional

PLN suele confundirse con otra disciplina hermana llamada Lingüística Computacional (LC). Si bien ambas están estrechamente relacionadas, tienen un foco distinto. La LC busca responder preguntas fundamentales sobre el lenguaje mediante el uso de la computación, es decir, cómo entendemos el lenguaje.

je, cómo producimos lenguaje o cómo aprendemos lenguaje. Mientras que en PLN el foco está en resolver problemas específicos, tales 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 PLN, el éxito de una solución se mide en base métricas concretas (Ej: qué tan similar es la traducción automática a una hecha por un humano) independientemente si el modelo hace uso de alguna teoría lingüística.

El procesamiento del lenguaje natural (PLN) desarrolla métodos para resolver problemas prácticos relacionados con el lenguaje [Johnson, 2014].

Algunos ejemplos son:

- Reconocimiento automático del habla.
- Traducción automática.
- Extracción de información de documentos.

La lingüística computacional (LC) estudia los procesos computacionales subyacentes al lenguaje (humano).

- ¿Cómo comprendemos el lenguaje?
- ¿Cómo producimos el lenguaje?
- ¿Cómo aprendemos el lenguaje?

El PLN y la LC utilizan métodos y modelos similares.

Aunque existe una superposición sustancial, hay una diferencia importante en el enfoque. La LC se centra en la lingüística respaldada por métodos computacionales (similar a la biología computacional o la astronomía computacional). En lingüística, el lenguaje es el objeto de estudio. El PLN se centra en resolver tareas bien definidas relacionadas con el lenguaje humano (como la traducción, la respuesta a consultas, las conversaciones). Si bien los conocimientos lingüísticos fundamentales pueden ser cruciales para realizar estas tareas, el éxito se mide en función de si y cómo se logra el objetivo (según una métrica de evaluación) [Eisenstein, 2018].

El procesamiento del lenguaje natural y la lingüística computacional están estrechamente relacionados y se superponen en muchos aspectos. Ambos campos utilizan métodos y modelos similares para abordar problemas relacionados con el lenguaje humano. Sin embargo, la diferencia principal radica en el enfoque: la lingüística computacional se centra en la lingüística respaldada por métodos computacionales, mientras que el procesamiento del lenguaje natural se centra en resolver tareas prácticas relacionadas con el lenguaje. Ambos campos son fundamentales para comprender y aprovechar el poder del lenguaje humano en la era digital.

1.2. Niveles de descripción lingüística

El campo de la **descripción lingüística** abarca diferentes niveles:

- **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.

El PLN puede abordar tareas en cada uno de estos niveles, pero a menudo se enfoca en niveles más altos de representación y comprensión.

1.2.1. 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.

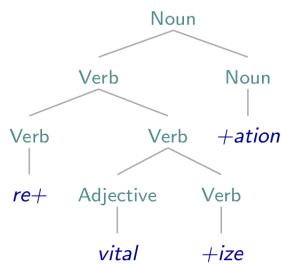
1.2.2. Fonología

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. Por ejemplo, en inglés, la "p" y la "b" son fonemas distintos porque cambian el significado de las palabras en las que se encuentran. La fonología también examina las variaciones en la pronunciación de los sonidos en diferentes contextos y dialectos [Fromkin et al., 2018].

1.2.3. Morfología

La morfología se ocupa del estudio de la estructura interna de las palabras. Los morfemas son las unidades mínimas de significado que componen las palabras. Por ejemplo, en la palabra "deshacer", los morfemas son "des-", "hacer" y "er". La morfología también se interesa por los procesos de formación de palabras, como la derivación, donde se agregan prefijos o sufijos a una palabra existente para formar una nueva palabra con un significado diferente [Johnson, 2014].

- La morfología estudia la estructura de las palabras (por ejemplo, re+estructur+ando, in+olvid+able) [Johnson, 2014]
- Morfema: el término lingüístico para la unidad más elemental de forma gramatical [Fromkin et al., 2018]. Por ejemplo, morfología = morf + ología (la ciencia de).
- Morfología derivativa: proceso de formar una nueva palabra a partir de una palabra existente, a menudo mediante la adición de un prefijo o sufijo.
- La morfología derivativa exhibe una estructura jerárquica. Ejemplo: re+vital+iz+ación



- El sufijo generalmente determina la categoría sintáctica (part-of-speech) de la palabra derivada.

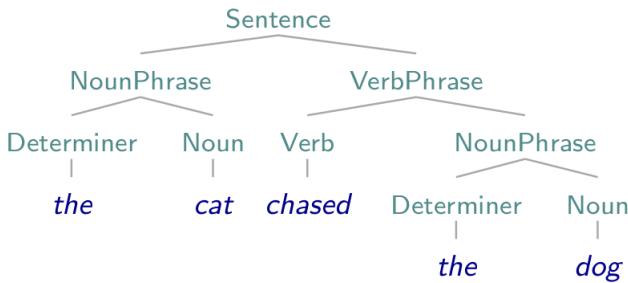
1.2.4. Sintaxis

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 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. Por ejemplo, en la oración "El perro persigue al gato", "el perro" es el sujeto, "persigue" es el verbo y "al gato" es el complemento directo [Johnson, 2014].

- La sintaxis estudia las formas en que las palabras se combinan para formar frases y oraciones [Johnson, 2014]
- El análisis sintáctico ayuda a identificar **quién hizo qué a quién**, un paso clave para comprender una oración.

1.2.5. 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. Por ejemplo, en la oración "El niño cortó la cuerda con una navaja", "el niño" es el agente, "la cuerda" es el tema y "una navaja" es el instrumento [Johnson, 2014].

La semántica se enfoca en el significado de las palabras, frases y oraciones. Estudia cómo se construye e interpreta este significado en el contexto del lenguaje. Además, dentro de la semántica, se analizan los roles semánticos, los cuales indican la función que desempeña cada entidad en una oración. Por ejemplo, en la oración "El niño cortó la cuerda con una navaja", se identifican distintos roles semánticos: "el niño" como el agente, "la cuerda" como el tema y "una navaja" como el instrumento utilizado [Johnson, 2014].

En resumen:

- La semántica estudia el significado de las palabras, frases y oraciones [Johnson, 2014].
- Dentro de la semántica, se analizan los roles semánticos, que indican el papel desempeñado por cada entidad en una oración.
- 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 una navaja", se puede identificar el agente como **el niño**, el tema como **la cuerda** y el instrumento como **una navaja**.
- Además de los roles semánticos, la semántica también abarca las relaciones léxicas, que son las relaciones entre diferentes palabras [Yule, 2016].
- Algunos ejemplos de relaciones léxicas incluyen la sinonimia (conceal/hide), la antonimia (shallow/deep) y la hipónimia (perro/animal).

1.2.6. 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 expre-

siones lingüísticas en situaciones reales y cómo los hablantes interpretan el significado implícito. Por ejemplo, la oración "Hace frío aquí" puede interpretarse como una sugerencia implícita de cerrar las ventanas [Fromkin et al., 2018].

1.3. Procesamiento del Lenguaje Natural y Aprendizaje Automático

Comprender y producir el lenguaje computacionalmente 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. Por ejemplo, si queremos tener un modelo que nos diga si un tweet tiene un sentimiento positivo o negativo respecto a un producto, primero necesito etiquetar manualmente un conjunto de tweets con su sentimiento asociado. Luego debo entrenar un algoritmo de aprendizaje sobre estos datos para poder predecir de manera automática el sentimiento asociado a tweets desconocidos. Como se 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.

Aunque los seres humanos somos grandes usuarios del lenguaje, también somos muy malos para comprender y describir formalmente las reglas que rigen el lenguaje.

Entender y producir lenguaje utilizando computadoras es altamente desafiante. Los métodos más conocidos para lidar con datos de lenguaje se basan en el aprendizaje automático supervisado.

El aprendizaje automático supervisado consiste en intentar inferir patrones y regularidades a partir de un conjunto de pares de entrada y salida preanotados (también conocido como conjunto de datos de entrenamiento).

Conjunto de Datos de Entrenamiento: Datos de NER CoNLL-2003 Cada línea contiene un token, una etiqueta de parte de la oración, 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

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

1.4. Desafíos del Lenguaje

Existen tres propiedades desafiantes del lenguaje: la discreción, la composicionalidad y la dispersión.

Discreción: no podemos inferir la relación entre dos palabras a partir de las letras que las componen (por ejemplo, hamburguesa y pizza).

Composicionalidad: el significado de una oración va más allá del significado individual de sus palabras.

Dispersión: la forma en que las palabras (símbolos discretos) pueden combinarse para formar significados es prácticamente infinita.

1.5. Ejemplo de tareas NLP

Clasificación de temas La clasificación de temas es una tarea de Procesamiento del Lenguaje Natural (PLN) en la cual se asigna a un documento una de varias categorías, como deportes, política, cotilleos o economía. Las palabras presentes en los documentos brindan pistas importantes sobre su tema. Sin embargo, redactar reglas para esta tarea es un desafío debido a la complejidad del lenguaje. La anotación de datos, en la cual los lectores clasifican los documentos por temas, puede ayudar a generar conjuntos de datos de entrenamiento para algoritmos de aprendizaje automático supervisado. Estos algoritmos aprenden patrones de uso de palabras que facilitan la categorización de los documentos.

- Clasificar un documento en una de las cuatro categorías: Deportes, Política, Cotilleos y Economía.
- Las palabras en los documentos proporcionan indicios muy sólidos.
- ¿Qué palabras brindan qué indicios?
- Elaborar reglas para esta tarea resulta bastante desafiante.
- No obstante, los lectores pueden categorizar fácilmente varios documentos según su tema (anotación de datos).
- Un algoritmo de aprendizaje automático supervisado puede identificar los patrones de uso de palabras que ayudan a categorizar los documentos.

Análisis de Sentimiento El análisis de sentimientos se refiere a la aplicación de técnicas de Procesamiento del Lenguaje Natural (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 frases se clasifican automáticamente en categorías

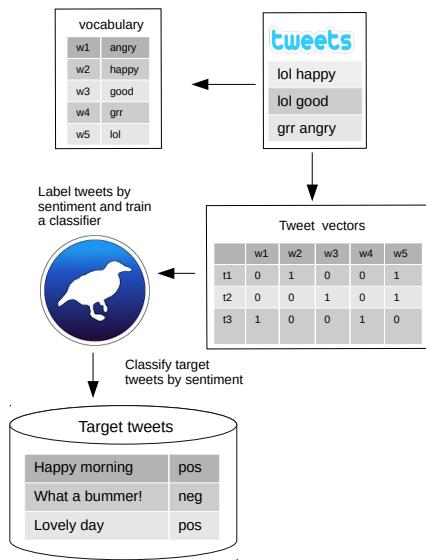
positivas, negativas o neutrales. Las soluciones más avanzadas utilizan modelos de aprendizaje automático supervisado entrenados con ejemplos anotados manualmente.

En este tipo de clasificación, es habitual emplear el aprendizaje supervisado, siendo las Máquinas de Vectores de Soporte (SVM) una opción popular. El objetivo de las SVM es encontrar un hiperplano que separe las clases con el margen máximo, logrando la mejor separación entre las clases positivas, negativas y neutrales [Eisenstein, 2018].

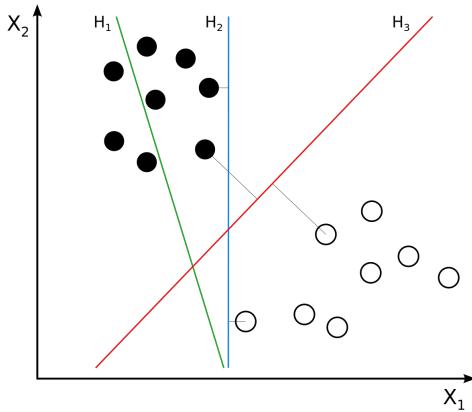
- Aplicación de técnicas de **PLN** para identificar y extraer información subjetiva de conjuntos de datos textuales.
- Clasificación automática de frases en las categorías **positiva**, **negativa** o **neutral**.



- Las soluciones más avanzadas emplean modelos de aprendizaje automático **supervisado**, entrenados con ejemplos **anotados manualmente** [Mohammad et al., 2013].



- Idea: Encontrar un hiperplano que separe las clases con el margen máximo (mayor separación).



- H_3 separa las clases con el margen máximo.

1.5.1. Lingüística y Procesamiento del Lenguaje Natural (PNL)

El conocimiento de las estructuras lingüísticas es fundamental para el diseño de características y el análisis de errores en el Procesamiento del Lenguaje Natural (PNL). Los enfoques de aprendizaje automático en PNL 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].

- El conocimiento de las estructuras lingüísticas es importante para el diseño de características y el análisis de errores en PNL [Bender, 2013].
- Los enfoques de aprendizaje automático en PNL requieren características que puedan describir y generalizar el uso del lenguaje.
- El objetivo es guiar al algoritmo de aprendizaje automático para encontrar correlaciones entre el uso del lenguaje y el conjunto de etiquetas objetivo.
- El conocimiento sobre las estructuras lingüísticas puede influir en el diseño de características para los enfoques de aprendizaje automático en PNL.

El PNL plantea diversos desafíos, como los costos de anotación, las variaciones de dominio y la necesidad de actualizaciones continuas. La anotación manual requiere mucho trabajo y tiempo. Las variaciones de dominio implican aprender patrones diferentes para diferentes corpus de texto. Los modelos entrenados en un dominio pueden no funcionar bien en otro. Además, los modelos de PNL pueden volverse obsoletos a medida que el uso del lenguaje evoluciona con el tiempo.

1.6. Desafíos en el Procesamiento del Lenguaje Natural (PNL)

- **Costos de Anotación:** la anotación manual es **laboriosa** y **consume mucho tiempo**.
- **Variaciones de Dominio:** el patrón que queremos aprender puede variar de un corpus a otro (por ejemplo, deportes, política).
- ¡Un modelo entrenado con datos anotados de un dominio no necesariamente funcionará en otro!
- Los modelos entrenados pueden quedar desactualizados con el tiempo (por ejemplo, nuevos hashtags).

Variación de Dominio en el Análisis de Sentimiento

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.

Superando los costos de anotación de datos Supervisión Distant:

- Etiquetar automáticamente datos no etiquetados (**API de Twitter**) utilizando un método heurístico.
- **Enfoque de Anotación de Emoticonos (EAA):** los tweets con emoticonos positivos :) o negativos :(se etiquetan según la polaridad indicada por el emotícono [Read, 2005].
- El emotícono se **elimina** del contenido.
- Este enfoque también se ha ampliado utilizando hashtags como #anger y emojis.
- No es trivial encontrar técnicas de supervisión distante para todo tipo de problemas de PNL.

Crowdsourcing

- Confiar en servicios como **Amazon Mechanical Turk** o **Crowdflower** para solicitar a la **multitud** que anote datos.
- Esto puede resultar costoso.
- Es difícil garantizar la calidad de las anotaciones.

1.7. Estudio de caso: Clasificación de sentimientos en tweets

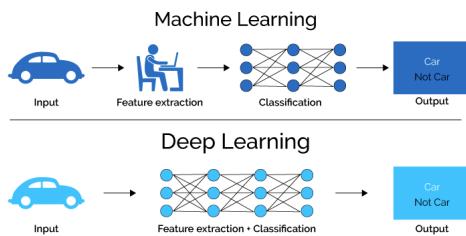
- En 2013, el taller de Evaluación Semántica (SemEval) organizó la tarea de "Análisis de sentimientos en Twitter" [Nakov et al., 2013].
- La tarea se dividió en dos sub-tareas: el nivel de expresión y el nivel del mensaje.
- Nivel de expresión: se centró en determinar la polaridad del sentimiento de un mensaje según una entidad marcada dentro de su contenido.
- Nivel del mensaje: se debía determinar la polaridad según el mensaje en general.
- Los organizadores lanzaron conjuntos de datos de entrenamiento y prueba para ambas tareas [Nakov et al., 2013].

El sistema NRC

- El equipo que logró el mejor rendimiento en ambas tareas, entre 44 equipos, fue el equipo *NRC-Canada* [Mohammad et al., 2013].
- El equipo propuso un enfoque supervisado utilizando un clasificador SVM lineal con las siguientes características hechas a mano para representar los tweets:
 1. N-gramas de palabras.
 2. N-gramas de caracteres.
 3. Etiquetas de partes del discurso.
 4. Agrupaciones de palabras entrenadas con el método de agrupamiento 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 emoticonos 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 lexicones de polaridad [Mohammad et al., 2013]. Dos de estos lexicones se generaron utilizando el método PMI a partir de tweets anotados con hashtags y emoticonos.

1.8. Ingeniería de características y Aprendizaje Profundo

- Hasta 2014, la mayoría de los sistemas de PNL de última generación se basaban en ingeniería de características + modelos de aprendizaje automático superficiales (por ejemplo, SVM, HMM).
- Diseñar las características de un sistema de PNL ganador requiere mucho conocimiento específico del dominio.
- El sistema NRC se construyó antes de que el aprendizaje profundo se hiciera popular en PNL.
- Por otro lado, los sistemas de Aprendizaje Profundo se basan en redes neuronales para aprender automáticamente buenas representaciones.



Ingeniería de características y Aprendizaje Profundo

- El Aprendizaje Profundo proporciona resultados de última generación en la mayoría de las tareas de PNL.
- Grandes cantidades de datos de entrenamiento y máquinas GPU multicore más rápidas son clave en el éxito del aprendizaje profundo.
- Las **redes neuronales** y las **incrustaciones de palabras** desempeñan un papel fundamental en los modelos modernos de PNL.

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.

1.9. Historia

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 tenido tres grandes períodos: 1) el racionalismo, 2) el empirismo, y 3) el aprendizaje profundo [Deng y Liu, 2018] que describimos a continuación.

El racionalismo abarca desde 1950 a 1990, donde las soluciones consistían en diseñar reglas manuales para incorporar mecanismos de conocimiento y razonamiento. Un ejemplo emblemático es el agente de conversación (o chatbot) ELIZA desarrollado por Joseph Weizenbaum que simulaba un psicoterapeuta rogeriano. Luego, a partir de la década de los 90s, el diseño de métodos estadísticos y de aprendizaje automático construidos sobre corpus llevan a PLN hacia un enfoque empírista. Las reglas ya no se construyen sino que se “aprenden” a partir de datos etiquetados. Algunos modelos representativos de esta época son los filtros de spam basados en modelos lineales, 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 por depender 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, comienzan a mostrar resultados muy superiores en varias tareas emblemáticas de PLN [Collobert et al., 2011]. La idea de estos modelos es representar la entrada (el texto) con una jerarquía de parámetros (o capas) que permiten encontrar representaciones idóneas para la tarea en cuestión, proceso al cual se refiere como “aprendizaje profundo”. Estos modelos se caracterizan por tener muchos más parámetros que los modelos anteriores (superando la barrera del millón en algunos casos) y requerir grandes volúmenes de datos para su entrenamiento. Una gracia de estos modelos es que pueden ser pre-entrenados con texto no etiquetado como libros, Wikipedia, texto de redes sociales y de la Web para encontrar representaciones iniciales de palabras y oraciones (a lo que conocemos como word embeddings), las cuales pueden ser posteriormente adaptadas para la tarea objetivo donde sí se tienen datos etiquetados (Proceso conocido como transfer learning). Aquí destacamos modelos como Word2Vec [Mikolov 2013], BERT [Devlin 2018] y

GPT-3 [Brown 2020].

Este tipo de modelos ha ido perfeccionándose en los últimos años, llegando a obtener resultados cada vez mejores para casi todos los problemas del área [NLPProgress]. Sin embargo, este progreso no ha sido libre de controversias. El aumento exponencial en la cantidad de parámetros de cada nuevo modelo respecto a su predecesor, hace que los recursos computacionales y energéticos necesarios para construirlos sólo estén al alcance de unos pocos. Además, varios estudios han mostrado que estos modelos aprenden y reproducen los sesgos y prejuicios (ej: género, religión, racial) presentes en los textos a partir de los cuales se entrena. Sin ir más lejos, la investigadora Timnit Gebru fue despedida de Google cuando se le negó el permiso para publicar un artículo que ponía de manifiesto estos problemas [Bender 2021].

El progreso de la PNL se puede dividir en tres oleadas principales: 1) racionalismo, 2) empirismo y 3) aprendizaje profundo [Deng and Liu, 2018].

- 1950 - 1990 Racionalismo: se enfocaba en diseñar reglas hechas a mano para incorporar conocimiento y mecanismos de razonamiento en sistemas de PNL inteligentes (por ejemplo, ELIZA para simular a un psicoterapeuta Rogeriano, MARGIE para estructurar información del mundo real en ontologías de conceptos).
- 1991 - 2009 Empirismo: se caracteriza por la explotación de corpora de datos y modelos de aprendizaje automático y estadísticos (superficiales) (por ejemplo, Naive Bayes, HMMs, modelos de traducción IBM).
- 2010 - Aprendizaje Profundo: la ingeniería de características (considerada como un cuello de botella) se reemplaza con el aprendizaje de representaciones y/o redes neuronales profundas (por ejemplo, <https://www.deepl.com/translator>). Un artículo muy influyente en esta revolución: [Collobert et al., 2011].

1.10. Conclusiones

En este capítulo, hemos explorado el desafío de entender y producir lenguaje utilizando computadoras. El aprendizaje automático supervisado es una de las principales técnicas utilizadas para abordar este desafío. Además, hemos discutido las propiedades desafiantes del lenguaje, como la discreción, la composicionalidad y la dispersión. Estos aspectos nos muestran la complejidad inherente al procesamiento del lenguaje natural y nos desafían a encontrar soluciones efectivas.

¹Las fechas son aproximadas.

Capítulo 2

Modelo de Espacio Vectorial y Recuperación de Información

- ¿Cómo recupera un motor de búsqueda, como Duckduckgo o Google, los documentos relevantes a partir de una consulta dada?
- ¿Cómo puede una empresa procesar las reclamaciones dejadas por sus usuarios en sus portales web?

Estos problemas se estudian en los siguientes campos:

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

¡Ambos están estrechamente relacionados con el Procesamiento del Lenguaje Natural (NLP, por sus siglas en inglés)! (las fronteras entre estos campos no están claras).

2.1. Tokens y Tipos

Tokenización: la tarea de dividir una oración o documento en fragmentos llamados *tokens*.

Se pueden emplear transformaciones adicionales, como la eliminación de caracteres especiales (por ejemplo, puntuación), minúsculas, etc. [Manning et al., 2008].

Ejemplo Entrada: Me gustan los lenguajes humanos y los lenguajes de programación.

Tokens: [Me] [gustan] [los] [lenguajes] [humanos] [y] [los] [lenguajes] [de] [programación]

Tipos

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

Tipos para la oración anterior: [Me] [gustan] [los] [lenguajes] [humanos] [y] [de] [programación] El token *lenguajes* se repitió en la oración.

Extracción de Vocabulario

- Un *término* es un *tipo* normalizado.
- La normalización es el proceso de crear clases de equivalencia de diferentes *tipos*. Esto quedará claro en las siguientes diapositivas.
- El vocabulario *V* es el conjunto de términos (tokens únicos normalizados) dentro de una colección de documentos o corpus *D*.

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.
- Estos términos se llaman *stopwords* e incluyen artículos, pronombres, preposiciones y conjunciones. Ejemplo: [un, una, y, cualquier, tiene, hacer, no, hizo, el, en].

¡La eliminación de stopwords puede ser inconveniente en muchas tareas de procesamiento del lenguaje natural!

Ejemplo: No me gusta la pizza => pizza (se eliminaron "no", "mez" "gusta")

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. Ejemplo: Algoritmo de Porter.

(F)	Rule	Example
SSES	→ SS	caresses → caress
IES	→ I	ponies → poni
SS	→ SS	caress → caress
S	→	cats → cat

Ejemplo: *d* = Me gustan los lenguajes humanos y los lenguajes de programación => Me gustan los lenguaj y los program lenguaj¹

El vocabulario del documento *d* después de eliminar stopwords y realizar stemming:

¹http://9ol.es/porter_js_demo.html

termId	value
t1	human
t2	languag
t3	program

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 *estudios*, una regla de stemming devolvería el término *estudi*, mientras que a través de la lematización obtendríamos el término *study*².

2.1.1. Ley de Zipf

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

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

- Donde cf es una constante dependiente de la colección y $\beta > 0$ es un factor de decaimiento.
- Si $\beta = 1$, entonces f sigue exactamente la Ley de Zipf; de lo contrario, sigue una distribución similar a la de Zipf.
- La ley se relaciona con el principio del mínimo esfuerzo. A menudo utilizamos pocas palabras para expresar ideas.
- La Ley de Zipf es un tipo de distribución de ley de potencia (distribuciones de cola larga).
- Si trazamos un gráfico de *log-log*, obtenemos una línea recta con una pendiente de $-\beta$.
- Enumerar las palabras más frecuentes de un corpus se puede utilizar para construir una lista de *stopwords*.

²<https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/>

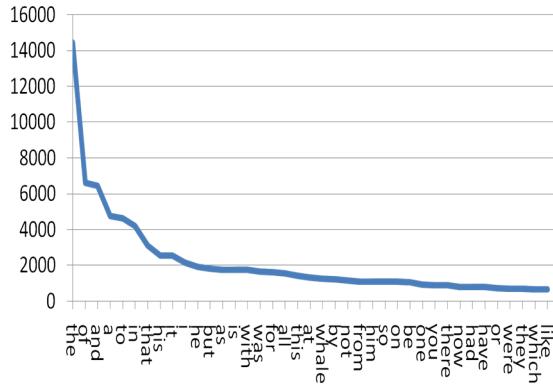


Figura 2.1: Ley de Zipf

2.1.2. Listas de publicaciones y el índice invertido

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

- La lista de publicaciones 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.
- Un índice invertido es una estructura de datos tipo diccionario que mapea los términos $t_i \in V$ con sus listas de publicaciones correspondientes.

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

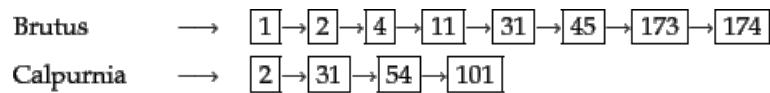


Figura 2.2: Índice invertido

2.1.3. 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 son:

- Rastreador: un robot que navega por la web según una estrategia definida. Por lo general, comienza navegando por un conjunto de sitios web iniciales 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 rastreador.
- 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 clasificación: la función utilizada por el procesador de consultas para clasificar los documentos indexados en la colección por relevancia según una consulta.
- Interfaz de usuario: recibe la consulta como entrada y devuelve los documentos clasificados por relevancia.

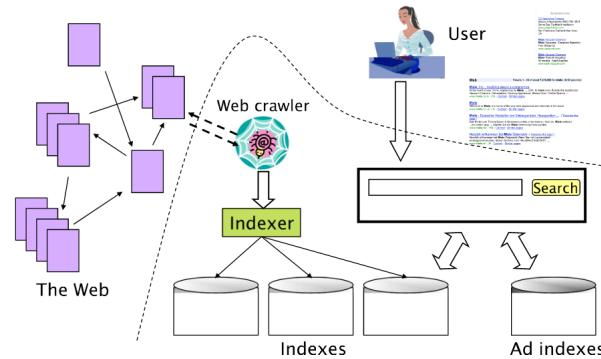


Figura 2.3: Los diversos componentes de un motor de búsqueda web [Manning et al., 2008].

2.2. El modelo de espacio vectorial

- Para clasificar consultas o medir la similitud entre dos documentos, necesitamos una métrica de similitud.
- Los documentos pueden ser *representados* como vectores de términos, donde cada término es una dimensión del vector [Salton et al., 1975].
- Documentos con diferentes palabras y longitudes residirán en el mismo espacio vectorial.
- Este tipo de representaciones se llaman *Bolsa de Palabras* (Bag of Words).
- En las representaciones de bolsa de palabras, se pierde el orden de las palabras y la estructura lingüística de una oración.

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

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

- ¿Cómo podemos modelar la información que aporta un término a un documento?

Frecuencia de Término - Frecuencia Inversa de Documento

- Sea $tf_{i,j}$ la frecuencia del término t_i en el documento d_j .
- Un término que ocurre 10 veces debería proporcionar más información que uno que ocurre solo una vez.
- ¿Qué ocurre cuando tenemos documentos que son mucho más largos que otros?
- Podemos normalizar dividiendo por la frecuencia máxima del término en el documento.

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

- ¿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.
- 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 $idf = 1$.
- El modelo de puntuación *tf-idf* combina las puntuaciones 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:

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

2.2.1. 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.
- 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.
- Observa que la similitud del coseno normaliza los vectores por su norma euclíadiana $\|\vec{d}\|_2$.

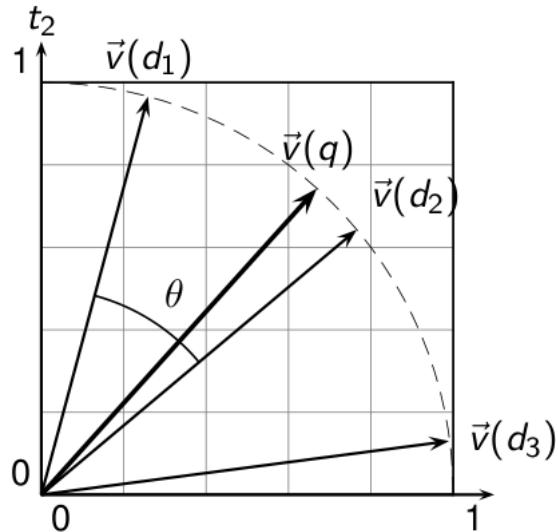


Figura 2.4: Similitud del coseno.

Ejercicio

- Supongamos que tenemos 3 documentos formados a partir de las siguientes secuencias de términos: $d_1 \rightarrow t_4t_3t_1t_4$ $d_2 \rightarrow t_5t_4t_2t_3t_5$ $d_3 \rightarrow t_2t_1t_4t_4$
- Construye una matriz término-documento de dimensiones 5×3 utilizando pesos simples de $tf\text{-}idf$ (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 $tf\text{-}idf$.
- ¿Cuál es el documento más cercano a d_1 ?

	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 2.1: Matriz tf-idf

2.3. Agrupamiento de Documentos

- ¿Cómo podemos agrupar documentos que son similares entre sí?
- El agrupamiento es el proceso de agrupar documentos que son similares entre sí.
- Cada grupo de documentos se llama *cluster* o grupo.
- En el agrupamiento, intentamos identificar grupos de documentos en los que la similitud entre documentos en el mismo grupo se maximiza y la similitud de documentos en diferentes grupos se minimiza.
- El agrupamiento de documentos permite identificar temas en un corpus y reducir el espacio de búsqueda en un motor de búsqueda, es decir, el índice invertido se organiza según los grupos.
- K-means es un algoritmo de agrupamiento simple que recibe el número de grupos k como parámetro.

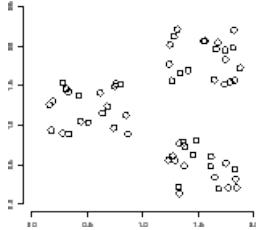


Figura 2.5: Conjunto de documentos donde los grupos se pueden identificar claramente.

- El algoritmo se basa en la idea de *centroide*, que es el vector promedio de los documentos que pertenecen al mismo grupo.
- Sea S un conjunto de vectores bidimensionales $3, 6, 1, 2, 5, 1$, el centroide de S es $(3 + 1 + 5)/3, (6 + 2 + 1)/3 = 3, 3$.

2.3.1. K-Means

1. Comenzamos con k centroides aleatorios.
2. Calculamos la similitud entre cada documento y cada centroide.
3. Asignamos cada documento a su centroide más cercano formando un grupo.
4. Se recalculan los centroides de acuerdo a los documentos asignados a ellos.
5. Este proceso se repite hasta la convergencia.

2.4. Conclusiones y Conceptos Adicionales

- Representar documentos como vectores es fundamental para calcular similitudes entre pares de documentos.
- Los vectores de "bag of words" carecen de estructura lingüística.
- Los vectores de "bag of words" son de alta dimensionalidad y dispersos.
- Los n-gramas de palabras pueden ayudar a capturar expresiones de múltiples palabras (por ejemplo, New York => new_york)
- Los sistemas modernos de recuperación de información van más allá de la similitud de vectores (PageRank, Retroalimentación de relevancia, Minería de registros de consultas, Grafo de conocimiento de Google, Aprendizaje automático).

```

K-MEANS( $\{\vec{x}_1, \dots, \vec{x}_N\}, K$ )
1  $(\vec{s}_1, \vec{s}_2, \dots, \vec{s}_K) \leftarrow \text{SELECTRANDOMSEEDS}(\{\vec{x}_1, \dots, \vec{x}_N\}, K)$ 
2 for  $k \leftarrow 1$  to  $K$ 
3 do  $\vec{\mu}_k \leftarrow \vec{s}_k$ 
4 while stopping criterion has not been met
5 do for  $k \leftarrow 1$  to  $K$ 
6   do  $\omega_k \leftarrow \{\}$ 
7   for  $n \leftarrow 1$  to  $N$ 
8   do  $j \leftarrow \arg \min_{j'} |\vec{\mu}_{j'} - \vec{x}_n|$ 
9      $\omega_j \leftarrow \omega_j \cup \{\vec{x}_n\}$  (reassignment of vectors)
10    for  $k \leftarrow 1$  to  $K$ 
11    do  $\vec{\mu}_k \leftarrow \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x}$  (recomputation of centroids)
12 return  $\{\vec{\mu}_1, \dots, \vec{\mu}_K\}$ 

```

Figura 2.6: Algoritmo K-means

- La recuperación de información y la minería de textos se preocupan menos por la estructura lingüística y más por producir algoritmos rápidos y escalables [Eisenstein, 2018].

Capítulo 3

Modelos de Lenguaje Probabilísticos

3.1. El Problema del Modelado del Lenguaje

- Tenemos un vocabulario (finito), digamos $\mathcal{V} = \{\text{el, un, hombre, telescopio, Beckham, dos, ...}\}$
- Tenemos un conjunto (infinito) de cadenas, \mathcal{V}^* .
- Por ejemplo:
 - el STOP
 - un STOP
 - el fan STOP
 - el fan vio a Beckham STOP
 - el fan vio vio STOP
 - el fan vio a Beckham jugar para el Real Madrid STOP
- Donde STOP es un símbolo especial que indica el final de una oración.
- Tenemos una muestra de entrenamiento de ejemplos de oraciones en inglés.
- Necesitamos “aprender” una distribución de probabilidad p .
- p es una función que satisface:

$$\sum_{x \in V^*} p(x) = 1$$
$$p(x) \geq 0 \quad \text{para todo } x \in V^*$$

- Ejemplos de probabilidades asignadas a las oraciones:

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

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

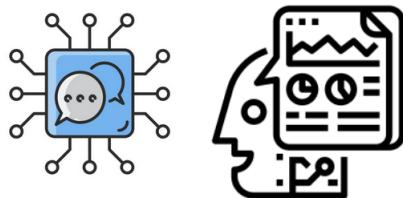
$$p(\text{el fan vio a Beckham STOP}) = 2 \times 10^{-8}$$

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

...

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

- Idea 1: El modelo asigna una probabilidad más alta a las oraciones fluidas (aquellas que tienen sentido y son gramaticalmente correctas).
- Idea 2: Estimar esta función de probabilidad a partir del texto (corpus).
- El modelo de lenguaje ayuda a los modelos de generación de texto a distinguir entre buenas y malas oraciones.



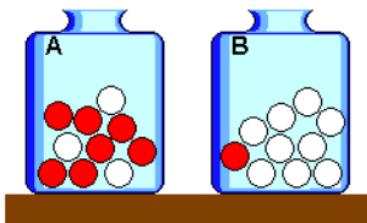
3.1.1. ¿Por qué queríamos hacer esto?

- El reconocimiento del habla fue la motivación original.
- Considera las oraciones: 1) reconocer el habla y 2) arruinar una playa bonita.
- Estas dos oraciones suenan muy similares al ser pronunciadas, 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 $p(\text{reconocer el habla})$ sobre $p(\text{arruinar una playa bonita})$.
- 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.
- Problemas relacionados son el reconocimiento óptico de caracteres y el reconocimiento de escritura a mano.
- 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, chatbots).
- Las técnicas de estimación desarrolladas para este problema serán MUY útiles para otros problemas en el procesamiento del lenguaje natural.

3.1.2. Los Modelos de Lenguaje son Generativos

- Los modelos de lenguaje pueden generar oraciones al muestrear secuencialmente a partir de las probabilidades.
- Esto es análogo a extraer bolas (palabras) de una urna donde sus tamaños son proporcionales a sus frecuencias relativas.
- Alternativamente, uno siempre podría extraer la palabra más probable, lo cual es equivalente a predecir la siguiente palabra.



3.2. ¿Por qué los modelos de lenguaje son importantes?

Los modelos de lenguaje son fundamentales en el procesamiento del lenguaje natural. Ayudan a abordar desafíos como el reconocimiento del habla, la transcripción automática y la generación de texto. Al comprender y estimar la probabilidad de ocurrencia de las secuencias de palabras, los modelos de

lenguaje mejoran la precisión y la calidad en diversas aplicaciones de procesamiento del lenguaje natural.

Los modelos de lenguaje permiten a los sistemas de reconocimiento del habla distinguir entre diferentes interpretaciones de palabras o frases que suenan similar pero tienen significados distintos. Esto es especialmente importante en situaciones en las que la ambigüedad podría llevar a una interpretación errónea. Al utilizar las probabilidades del modelo de lenguaje, los sistemas de reconocimiento del habla pueden seleccionar la interpretación más probable y coherente en función del contexto.

Además, los modelos de lenguaje son esenciales en tareas generativas, como la traducción automática, la generación de resúmenes y la creación de chatbots. Estos modelos ayudan a generar texto coherente y natural al muestrear secuencialmente palabras de acuerdo con sus probabilidades estimadas.

Las técnicas desarrolladas para el problema del modelado del lenguaje son también aplicables a otros desafíos en el procesamiento del lenguaje natural. Los avances en la estimación de probabilidades y en la generación de texto tienen un impacto significativo en campos como la inteligencia artificial, la lingüística computacional y la comunicación basada en texto.

En resumen, los modelos de lenguaje desempeñan un papel crucial en la comprensión y generación de texto, mejorando la precisión, la coherencia y la calidad en diversas aplicaciones de procesamiento del lenguaje natural. Son herramientas fundamentales para avanzar en la comprensión y la capacidad de interacción de las máquinas con el lenguaje humano.

3.2.1. Un Método Ingenuo

- Un método muy ingenuo para estimar la probabilidad de una oración es contar las apariciones de la oración en los datos de entrenamiento y dividirlo por el número total de oraciones de entrenamiento (N) para estimar la probabilidad.
- Tenemos N oraciones de entrenamiento.
- Para cualquier oración x_1, x_2, \dots, x_n , $c(x_1, x_2, \dots, x_n)$ es el número de veces que se ha visto la oración en nuestros datos de entrenamiento.
- Una estimación ingenua:

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

- Problema: A medida que el número de posibles oraciones crece de manera exponencial con la longitud de las oraciones y el tamaño del vocabulario, se vuelve cada vez más improbable que una oración específica aparezca en los datos de entrenamiento.
- En consecuencia, muchas oraciones tendrán una probabilidad cero según el modelo ingenuo, lo que lleva a una mala generalización.

3.3. Procesos de Markov

- Considera una secuencia de variables aleatorias X_1, X_2, \dots, X_n .
- Cada variable aleatoria puede tomar cualquier valor en un conjunto finito V .
- Por ahora, asumimos que la longitud n está fija (por ejemplo, $n = 100$).
- Nuestro objetivo: modelar $P(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n)$

Procesos de Markov de primer orden 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. La probabilidad conjunta de una secuencia de palabras se calcula multiplicando las probabilidades condicionales de las palabras sucesivas dado su predecesor inmediato. Esta es la suposición de Markov de primer orden:

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

Procesos de Markov de segundo orden 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 predecesores inmediatos. La suposición de Markov de segundo orden es la siguiente:

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

3.3.1. 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".

3.4. Modelos de lenguaje trigram

Un modelo de lenguaje trigram consiste en:

1. Un conjunto finito V de palabras.
2. Un parámetro $q(w|u, v)$ para cada trigram 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 trigram es:

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

Aquí, definimos $x_0 = x_{-1} = *$ por conveniencia.

Un ejemplo Para la oración "the dog barks STOP", tendríamos:

$$p(\text{the dog barks STOP}) = q(\text{the}|*, *) \times q(\text{dog}|*, \text{the}) \times q(\text{barks}|\text{the}, \text{dog}) \times q(\text{STOP}|\text{dog}, \text{barks})$$

3.4.1. El problema de estimación trigram

El problema de estimación restante es determinar los valores de los parámetros $q(w_i|w_{i-2}, w_{i-1})$. Por ejemplo:

$$q(\text{laughs}|\text{the}, \text{dog})$$

Una estimación natural (la "estimación de máxima verosimilitud") es la siguiente:

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

Por ejemplo:

$$q(\text{laughs}|\text{the}, \text{dog}) = \frac{\text{Count}(\text{the}, \text{dog}, \text{laughs})}{\text{Count}(\text{the}, \text{dog})}$$

Problemas de datos dispersos Una estimación natural (la "estimación de máxima verosimilitud") es la siguiente:

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

$$q(\text{laughs}|\text{the}, \text{dog}) = \frac{\text{Count}(\text{the}, \text{dog}, \text{laughs})}{\text{Count}(\text{the}, \text{dog})}$$

- Supongamos que el tamaño del vocabulario es $N = |V|$, entonces hay N^3 parámetros en el modelo.
- Por ejemplo, si $N = 20,000$, entonces $20,000^3 = 8 \times 10^{12}$ parámetros.

3.5. Evaluación de un modelo de lenguaje: Perplejidad

- Tenemos algunos datos de prueba, m oraciones: $s_1, s_2, s_3, \dots, s_m$
- Podemos analizar la probabilidad bajo nuestro modelo $\prod_{i=1}^m p(s_i)$. O más convenientemente, la probabilidad logarítmica:

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

- De hecho, la medida de evaluación habitual es la perplejidad:

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

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

Algo de intuición sobre la perplejidad

- Supongamos que tenemos un vocabulario V , y $N = |V| + 1$, y un modelo que predice:

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

- Es fácil calcular la perplejidad en este caso:

$$\text{Perplejidad} = 2^{-l} \quad \text{donde} \quad l = \log \frac{1}{N} \Rightarrow \text{Perplejidad} = N$$

- La perplejidad se puede ver como una medida del "factor de ramificación". efectivo.
- **Demostración:** Supongamos que tenemos m oraciones de longitud n en el corpus, y M es la cantidad de tokens en el corpus, $M = m \cdot n$.
- Consideraremos el logaritmo (base 2) de la probabilidad de una oración $s = w_1 w_2 \dots w_n$ 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 $q(w_i|w_{i-2}, w_{i-1})$ 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:

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

3.5.1. El trade-off entre sesgo y varianza

En el contexto de los modelos de lenguaje, el trade-off entre sesgo y varianza se refiere a la compensación entre la simplicidad del modelo y su capacidad para capturar la complejidad y la variabilidad de los datos de lenguaje.

- Modelos más simples, como los modelos de n-gramas de orden inferior, tienen un sesgo más alto pero una varianza más baja. Estos modelos asumen independencia condicional entre las palabras y simplifican la estructura del lenguaje.
- Modelos más complejos, como los modelos basados en redes neuronales, 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.

3.5.2. Estimación de máxima verosimilitud y overfitting

La estimación de máxima verosimilitud (MLE) es una técnica común para estimar los parámetros de un modelo de lenguaje. Sin embargo, los modelos de lenguaje basados en MLE pueden sufrir de overfitting (sobreajuste) a los datos de entrenamiento.

- Cuando se entrena un modelo de lenguaje con MLE, se maximiza la probabilidad de los datos de entrenamiento. Esto puede llevar a la asignación de 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.

3.5.3. Técnicas de regularización

Para abordar el problema del overfitting en modelos de lenguaje, se utilizan diversas técnicas de regularización. Estas técnicas ayudan a reducir la varianza del modelo y mejorar su capacidad de generalización.

Algunas técnicas comunes de regularización para modelos de lenguaje incluyen:

- **Suavizado de Laplace (Laplace smoothing):** Se agrega una cantidad pequeña a todas las cuentas de n-gramas para evitar la asignación de una probabilidad cero a n-gramas no observados en los datos de entrenamiento.
- **Suavizado de interpolación (Interpolation smoothing):** Se combina la distribución de probabilidad estimada por un modelo de n-gramas con las distribuciones estimadas por modelos de orden inferior. Esto permite incorporar información de n-gramas de orden inferior y reduce la varianza del modelo.
- **Modelos de interpolación de Kneser-Ney (Kneser-Ney interpolation models):** Estos modelos utilizan una técnica de suavizado específica (Kneser-Ney) que considera la frecuencia de unigramas y la frecuencia de n-gramas en contexto específicos para asignar probabilidades a n-gramas no observados.
- **Regularización de peso máximo (Weighted maximum likelihood regularization):** Se aplica una regularización que reduce los pesos de las cuentas de n-gramas más frecuentes. Esto ayuda a reducir la varianza y mejorar el rendimiento en datos de prueba.

Estas técnicas de regularización ayudan a controlar la varianza del modelo y a mitigar el overfitting, permitiendo un mejor equilibrio entre el sesgo y la varianza y mejorando la generalización a nuevas muestras.

3.6. Interpolación Lineal

- Tomamos nuestra estimación $q(w_i|w_{i-2}, w_{i-1})$ como:
$$q(w_i|w_{i-2}, w_{i-1}) = \lambda_1 \cdot q_{\text{ML}}(w_i|w_{i-2}, w_{i-1}) + \lambda_2 \cdot q_{\text{ML}}(w_i|w_{i-1}) + \lambda_3 \cdot q_{\text{ML}}(w_i)$$
donde $\lambda_1 + \lambda_2 + \lambda_3 = 1$, y $\lambda_i \geq 0$ para todo i .
- Nuestra estimación define correctamente una distribución (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 podemos demostrar que $q(w|u, v) \geq 0$ para todo $w \in V'$.

3.7. Estimación de los Valores λ

- Reservamos 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 trigram (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

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

3.8. Métodos de Descuento

- Consideraremos los siguientes recuentos y estimaciones de máxima verosimilitud:
- Las estimaciones de máxima verosimilitud son altas, especialmente para los elementos con recuentos bajos.
- Definimos los recuentos "descontados" de la siguiente manera:

$$\text{Recuento}^*(x) = \text{Recuento}(x) - 0,5$$

- Las nuevas estimaciones se basan en los recuentos descontados.

Frase	Recuento	$q_{ML}(w_i w_{i-1})$
the	48	
the, dog	15	15/48
the, woman	11	11/48
the, man	10	10/48
the, park	5	5/48
the, job	2	2/48
the, telescope	1	1/48
the, manual	1	1/48
the, afternoon	1	1/48
the, country	1	1/48
the, street	1	1/48

Frase	Recuento	Recuento*(x)	$q_{ML}(w_i w_{i-1})$
the	48		
the, dog	15	14.5	14,5/48
the, woman	11	10.5	10,5/48
the, man	10	9.5	9,5/48
the, park	5	4.5	4,5/48
the, job	2	1.5	1,5/48
the, telescope	1	0.5	0,5/48
the, manual	1	0.5	0,5/48
the, afternoon	1	0.5	0,5/48
the, country	1	0.5	0,5/48
the, street	1	0.5	0,5/48

- Ahora tenemos cierta “masa de probabilidad faltante”:

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

Por ejemplo, en nuestro caso:

$$\alpha(\text{the}) = \frac{10 \times 0,5}{48} = \frac{5}{48}$$

3.8.1. Modelos de Katz Back-Off (Bigramas)

- Para un modelo de bigrama, definimos dos conjuntos:

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

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

- Un modelo de bigrama:

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

- Donde:

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

- Para un modelo de trigramas, primero definimos dos conjuntos:

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

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

- Un modelo de trigramas se define en términos del modelo de bigramas:

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

- Donde:

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

Los modelos de Katz Back-Off son una técnica utilizada en modelos de lenguaje para abordar el desafío de la escasez de datos. Estos modelos permiten estimar las probabilidades condicionales de palabras en función de contextos más pequeños cuando no hay suficientes datos disponibles para estimar directamente las probabilidades completas.

En un modelo de bigrama, se define el conjunto $A(w_{i-1})$ como el conjunto de palabras w para las cuales la frecuencia de aparición de la secuencia (w_{i-1}, w) es mayor que cero, es decir, $\text{Count}(w_{i-1}, w) > 0$. Por otro lado, el conjunto $B(w_{i-1})$ se define como el conjunto de palabras w para las cuales la frecuencia de aparición de la secuencia (w_{i-1}, w) es igual a cero, es decir, $\text{Count}(w_{i-1}, w) = 0$.

En un modelo de bigrama, 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} . Por otro lado, si w_i está en el conjunto $B(w_{i-1})$, se utiliza una estimación suavizada que combina una constante de suavizado $\alpha(w_{i-1})$ y las probabilidades condicionales de máxima verosimilitud $q_{\text{ML}}(w_i)$ de las palabras en el conjunto $B(w_{i-1})$.

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

En el caso de un modelo de trigramas, se definen conjuntos similares $A(w_{i-2}, w_{i-1})$ y $B(w_{i-2}, w_{i-1})$ para las secuencias trigramas. 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 de suavizado basado en los conjuntos $A(w_{i-2}, w_{i-1})$ y $B(w_{i-2}, w_{i-1})$.

Estos modelos de Katz Back-Off permiten aproximar las probabilidades condicionales en situaciones donde la información disponible es limitada, al aprovechar información de contextos más pequeños cuando no se dispone de suficientes datos para estimaciones directas.

3.9. Resumen

- La derivación 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 las Asunciones de 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.
- Otros métodos para mejorar los modelos de lenguaje incluyen:
 - Introducir variables latentes para representar temas, conocidos como modelos de temas. [Blei et al., 2003]
 - 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.

Capítulo 4

Text Classification and Naïve Bayes

- La clasificación es fundamental tanto para la inteligencia humana como para la artificial.
- Decidir qué letra, palabra o imagen se ha presentado a nuestros sentidos, reconocer caras o voces, clasificar el correo, asignar calificaciones a las tareas.
- Estos son ejemplos de asignar una categoría a una entrada.
- El objetivo de la clasificación es tomar una única observación, extraer algunas características útiles y así clasificar la observación en una de las clases discretas establecidas.
- La mayoría de los casos de clasificación en el procesamiento del lenguaje se realizan mediante aprendizaje automático supervisado.
- Estas diapositivas se basan en el material del curso de Daniel Jurafsky: <https://web.stanford.edu/~jurafsky/slp3/4.pdf>

Ejemplo 1: Clasificación de spam

Ejemplo 2: ¿Quién escribió los documentos Federalist?

- 1787-8: Ensayos anónimos intentaron convencer a Nueva York de ratificar la Constitución de EE. UU.: Jay, Madison, Hamilton.
- La autoría de 12 de las cartas está en disputa.
- 1963: Resuelto por Mosteller y Wallace mediante métodos bayesianos.

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.

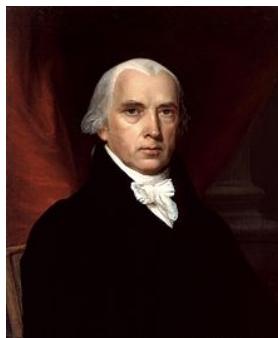


Figura 4.1: James Madison

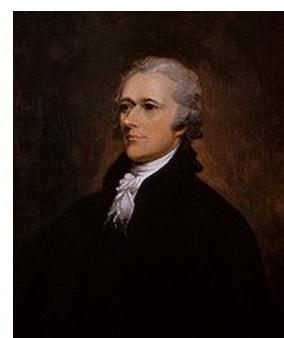


Figura 4.2: Alexander Hamilton

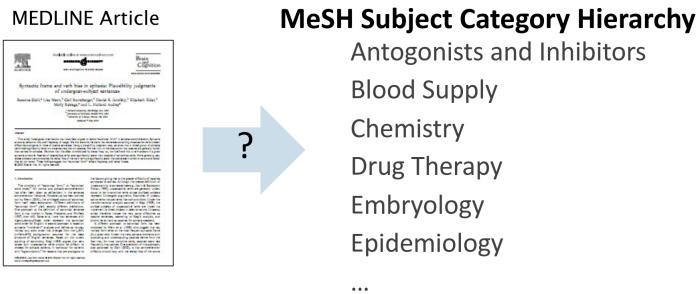
Ejemplo 3: ¿Cuál es el tema de este artículo médico?

Ejemplo 4: ¿Reseña de película positiva o negativa?

- + ...personajes extravagantes y sátira **rico** aplicada, y algunos **grandes** giros de la trama.
- - Fue **patético**. La peor parte fue las escenas de boxeo...
- + ...salsa de caramelo **increíble** y almendras dulces y tostadas. ¡Me **encanta** este lugar!
- - ...pizza **horrible** y **ridículamente** cara...

¿Por qué el análisis de sentimientos?

- Película: ¿Esta reseña es positiva o negativa?
- Productos: ¿Qué opinan las personas sobre el nuevo iPhone?
- Sentimiento público: ¿Cómo está la confianza del consumidor?



- Política: ¿Qué opinan las personas sobre este candidato o tema?
- Predicción: Predecir resultados electorales o tendencias del mercado a partir del sentimiento.

Clasificación básica de sentimientos El análisis de sentimientos es la detección de actitudes.

- Tarea simple en la que nos enfocamos en esta clase.
 - ¿Es la actitud de este texto positiva o negativa?

Resumen: Clasificación de texto 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
- ...

4.1. Clasificación de texto: Definición

Entrada:

- Un documento d
- Un conjunto fijo de clases $C = \{c_1, c_2, \dots, c_J\}$

Salida: Una clase predicha $c \in C$

4.1.1. Métodos de clasificación: Reglas codificadas a mano

Reglas basadas en combinaciones de palabras u otras características.

- Spam: dirección-en-lista-negra O (“dólares” Y “has sido seleccionado”)
- La precisión puede ser alta si las reglas se refinan cuidadosamente por expertos
- Pero construir y mantener estas reglas es costoso

4.1.2. Métodos de clasificación: Aprendizaje automático supervisado

Entrada:

- Un documento d
- Un conjunto fijo de clases $C = \{c_1, c_2, \dots, c_J\}$
- Un conjunto de entrenamiento de m documentos etiquetados manualmente: $(d_1, c_1), (d_2, c_2), \dots, (d_m, c_m)$

Salida:

- Un clasificador aprendido $\gamma : d \rightarrow c$

Cualquier tipo de clasificador se puede utilizar:

- Naïve Bayes
- Regresión logística
- Redes neuronales
- k-vecinos más cercanos

4.1.3. Problemas de aprendizaje supervisado

- Tenemos ejemplos de entrenamiento $x^{(i)}, y^{(i)}$ para $i = 1, \dots, m$. Cada $x^{(i)}$ es una entrada, cada $y^{(i)}$ es una etiqueta.
- La tarea es aprender una función f que asigna las entradas x a las etiquetas $f(x)$.
- Modelos condicionales:
 - Aprender una distribución $p(y|x)$ a partir de ejemplos de entrenamiento.
 - Para cualquier entrada de prueba x , definir $f(x) = \arg \max_y p(y|x)$.

4.1.4. Modelos generativos

- Dados ejemplos de entrenamiento $x^{(i)}, y^{(i)}$ para $i = 1, \dots, m$, la tarea es aprender una función f que asigna las entradas x a las etiquetas $f(x)$.
- Modelos generativos:
 - Aprender la distribución conjunta $p(x, y)$ a partir de los ejemplos de entrenamiento.
 - A menudo, tenemos $p(x, y) = p(y)p(x|y)$.
 - Nota: Luego tenemos

$$p(y|x) = \frac{p(y)p(x|y)}{p(x)} \quad \text{donde} \quad p(x) = \sum_y p(y)p(x|y).$$

4.1.5. Clasificación con Modelos Generativos

- Dados ejemplos de entrenamiento $x^{(i)}, y^{(i)}$ para $i = 1, \dots, m$. La tarea consiste en aprender una función f que mapee las entradas x a las etiquetas $f(x)$.
- Modelos generativos:
 - Aprenden la distribución conjunta $p(x, y)$ a partir de los ejemplos de entrenamiento.
 - A menudo, tenemos $p(x, y) = p(y)p(x|y)$.
- La salida del modelo es:

$$\begin{aligned} f(x) &= \arg \max_y p(y|x) = \arg \max_y \frac{p(y)p(x|y)}{p(x)} \\ &= \arg \max_y p(y)p(x|y) \end{aligned}$$

4.2. Intuición del Bayes Ingenuo

El Bayes Ingenuo es un método de clasificación simple ("ingenuo") basado en la regla de Bayes.

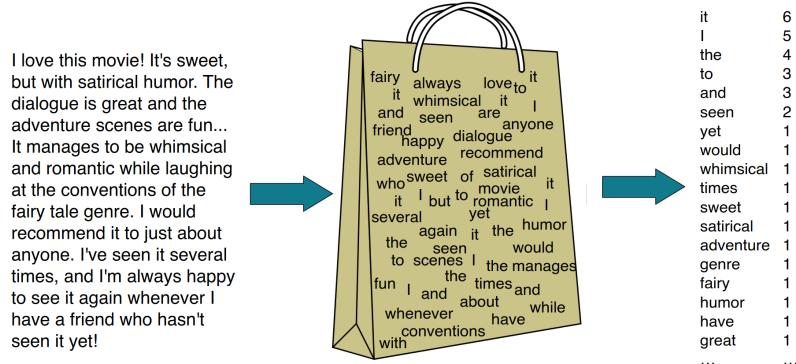
- Se basa en una representación muy simple de un documento: *Bolsa de palabras*.

La Representación de Bolsa de Palabras

4.2.1. Aplicación de la Regla de Bayes a Documentos y Clases

Para un documento d y una clase c :

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



4.3. Clasificador Bayes Ingenuo

- MAP significa "máximo a posteriori," que representa la clase más probable:

$$c_{\text{MAP}} = \arg \max_{c \in C} P(c|d)$$

- Para calcular la clase más probable, aplicamos la regla de Bayes:

$$= \arg \max_{c \in C} \frac{P(d|c)P(c)}{P(d)}$$

- Finalmente, podemos eliminar el denominador ya que permanece constante para todas las clases:

$$= \arg \max_{c \in C} P(d|c)P(c)$$

- Para clasificar el documento d , usamos la estimación MAP:

$$c_{\text{MAP}} = \arg \max_{c \in C} P(d|c)P(c)$$

- El documento d se representa como un conjunto de características x_1, x_2, \dots, x_n .

- El clasificador calcula la probabilidad condicional de las características dada una clase y la probabilidad a priori de la clase:

$$= \arg \max_{c \in C} P(x_1, x_2, \dots, x_n|c)P(c)$$

- El término $P(x_1, x_2, \dots, x_n|c)$ representa la "verosimilitud" de las características dada la clase.

- El término $P(c)$ representa la probabilidad .º priori" de la clase.

- El clasificador Bayes Ingenuo [McCallum et al., 1998] calcula la estimación MAP considerando las probabilidades de verosimilitud y a priori:

$$c_{\text{MAP}} = \arg \max_{c \in C} P(x_1, x_2, \dots, x_n | c) P(c)$$

- La probabilidad de las características dada la clase, $P(x_1, x_2, \dots, x_n | c)$, puede estimarse contando las frecuencias relativas en un corpus.
- La probabilidad a priori de la clase, $P(c)$, representa con qué frecuencia ocurre esta clase.
- Sin algunas suposiciones simplificadoras, estimar la probabilidad de cada posible combinación de características en $P(x_1, x_2, \dots, x_n | c)$ requeriría un gran número de parámetros y conjuntos de entrenamiento imposiblemente grandes.
- Por lo tanto, los clasificadores Bayes Ingenuo realizan dos suposiciones simplificadoras.

4.3.1. Suposiciones de Independencia del Bayes Ingenuo Multinomial

- Suposición de Bolsa de Palabras: asumimos que la posición de las palabras en el documento no importa.
- Suposición de Independencia Condicional: asumimos que las probabilidades de las características $P(x_i | c_j)$ son independientes dada la clase c_j .
- En el clasificador Bayes Ingenuo Multinomial, la probabilidad de un documento con características x_1, x_2, \dots, x_n dada la clase c se puede calcular como:

$$P(x_1, x_2, \dots, x_n | c) = P(x_1 | c) \cdot P(x_2 | c) \cdot P(x_3 | c) \cdot \dots \cdot P(x_n | c)$$

4.3.2. Clasificador Bayes Ingenuo Multinomial

- La estimación del Máximo A Posteriori (MAP) para la clase c en el clasificador Bayes Ingenuo Multinomial se calcula como:

$$c_{\text{MAP}} = \arg \max_{c \in C} P(x_1, x_2, \dots, x_n | c) P(c)$$

- Alternativamente, podemos escribirlo como:

$$c_{\text{NB}} = \arg \max_{c \in C} P(c_j) \prod_{x \in X} P(x | c)$$

- $P(c_j)$ representa la probabilidad a priori de la clase c_j .
- $\prod_{x \in X} P(x | c)$ representa la verosimilitud de las características x_1, x_2, \dots, x_n dadas la clase c .

4.3.3. Aplicación de los clasificadores Naive Bayes multinomiales a la clasificación de texto

El clasificador Naive Bayes multinomial para la clasificación de texto se puede aplicar de la siguiente manera:

$$c_{\text{NB}} = \arg \max_{c_j \in C} P(c_j) \prod_{i \in \text{positions}} P(x_i | c_j)$$

Donde:

- c_{NB} representa la clase predicha para el documento de prueba.
- C es el conjunto de todas las clases posibles.
- $P(c_j)$ es la probabilidad previa de la clase c_j .
- $\prod_{i \in \text{positions}} P(x_i | c_j)$ calcula la probabilidad de cada característica x_i en la posición i dada la clase c_j .
- El producto se toma sobre todas las posiciones de palabras en el documento de prueba.

4.3.4. Problemas al multiplicar muchas probabilidades

Multiplicar muchas probabilidades puede resultar en un desbordamiento de punto flotante, especialmente cuando se manejan probabilidades pequeñas. Por ejemplo, $0,0006 \times 0,0007 \times 0,0009 \times 0,01 \times 0,5 \times 0,000008 \dots$

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

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

Al tomar logaritmos, evitamos el problema del desbordamiento de punto flotante y realizamos cálculos en el espacio logarítmico. El clasificador se convierte en un modelo lineal, donde la predicción es el argmax de la suma de pesos (logaritmos de probabilidades) y las entradas (logaritmos de probabilidades condicionales). Por lo tanto, Naive Bayes es un clasificador lineal que opera en el espacio logarítmico.

4.3.5. Aprendizaje del modelo Naive Bayes multinomial

El primer intento: Estimaciones de máxima verosimilitud

- Las probabilidades se estiman utilizando las frecuencias observadas en los datos de entrenamiento.
- La probabilidad previa de una clase c_j se estima como:

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

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

- La estimación de la probabilidad de la palabra w_i dada la clase c_j se calcula como:

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

donde $w \in V$ representa una palabra en el vocabulario V .

- El denominador es la suma de las frecuencias de todas las palabras en el vocabulario dentro de la clase c_j .

4.3.6. Estimación de parámetros

Para estimar los parámetros del modelo Naive Bayes multinomial, seguimos estos pasos:

- Creamos un mega-documento para cada tema c_j concatenando todos los documentos de ese tema.
- Calculamos la frecuencia de la palabra w_i en el mega-documento, que representa la fracción de veces que la palabra w_i aparece entre todas las palabras en los documentos del tema 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 recuento de ocurrencias de w_i en el mega-documento del tema c_j por el recuento total de palabras en el mega-documento:

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

Aquí, $\text{count}(w_i, c_j)$ representa el número de veces que la palabra w_i aparece en el mega-documento del tema c_j , y $\text{count}(w, c_j)$ es el recuento total de palabras en el mega-documento.

4.3.7. Probabilidades cero y el problema de las palabras no vistas

Consideremos el escenario en el que no hemos encontrado la palabra "fantástico.^{en}" ningún documento de entrenamiento clasificado como positivo (pulgar hacia arriba). Utilizando la estimación de máxima verosimilitud, la probabilidad

$\hat{P}(\text{"fantástico"} \mid \text{positivo})$ se calcularía como:

$$\hat{P}(\text{"fantástico"} \mid \text{positivo}) = \frac{\text{count}(\text{"fantástico"}, \text{positivo})}{\sum_{w \in V} \text{count}(w, \text{positivo})}$$

En este caso, el recuento de la palabra “fantástico” en los documentos positivos es cero, lo que conduce a una probabilidad cero:

$$\hat{P}(\text{"fantástico"} \mid \text{positivo}) = \frac{0}{\sum_{w \in V} \text{count}(w, \text{positivo})} = 0$$

Sin embargo, las probabilidades cero no pueden eliminarse, independientemente de la evidencia adicional presente. Esto plantea un problema al calcular la estimación del máximo a posteriori (MAP), que se utiliza para la clasificación:

$$c_{\text{MAP}} = \arg \max_c \left(\hat{P}(c) \prod_i \hat{P}(x_i \mid c) \right)$$

Con una probabilidad cero para una palabra, toda la expresión se vuelve cero, independientemente de la otra evidencia.

4.3.8. Suavizado Laplaciano (Add-1) para Naïve Bayes

Manejo de probabilidades cero con el suavizado Laplaciano (Add-1):

- Para abordar el problema de las probabilidades cero, podemos utilizar la técnica de suavizado Laplaciano (Add-1).
- La estimación suavizada $\hat{P}(w_i \mid c)$ se calcula como:

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

- Aquí, se agrega un recuento adicional de 1 tanto al numerador como al denominador.
- El denominador se ajusta agregando el tamaño del vocabulario V para garantizar una normalización adecuada.
- Al hacerlo, evitamos las probabilidades cero y permitimos que cierta masa de probabilidad se distribuya a palabras no vistas.
- Esta técnica de suavizado ayuda a mitigar el problema de las palabras no vistas y evita la eliminación completa de ciertas clases durante la clasificación.

4.3.9. Naïve Bayes multinomial: aprendizaje

Aprendiendo el modelo Naïve Bayes multinomial:

- Para aprender los parámetros del modelo, necesitamos calcular los términos $P(c_j)$ y $P(w_k | c_j)$.
- Para cada clase c_j en el conjunto de clases C , realizamos los siguientes pasos:

- Recuperamos todos los documentos $docs_j$ que pertenecen a la clase c_j .
- Calculamos el término $P(w_k | c_j)$ para cada palabra w_k en el vocabulario V :

$$P(w_k | c_j) = \frac{n_k + \alpha}{n + \alpha \cdot |Vocabulary|}$$

donde n_k representa el número de ocurrencias de la palabra w_k en el documento concatenado $Text_j$.

- Calculamos la probabilidad a priori $P(c_j)$:

$$P(c_j) = \frac{|docs_j|}{\text{total number of documents}}$$

- Para calcular $P(w_k | c_j)$, necesitamos extraer el vocabulario V del corpus de entrenamiento.

4.3.10. Palabras desconocidas

Tratamiento de palabras desconocidas en los datos de prueba:

- Cuando encontramos palabras desconocidas en los datos de prueba que no aparecen en los datos de entrenamiento o en el vocabulario, las ignoramos.
- Eliminamos estas palabras desconocidas del documento de prueba como si no estuvieran presentes en absoluto.
- No asignamos ninguna probabilidad a estas palabras desconocidas en el proceso de clasificación.

Esto es una visión general del modelo Naive Bayes multinomial y su aplicación a la clasificación de texto. Cabe destacar que existen variantes y extensiones más sofisticadas de Naive Bayes que se adaptan a diferentes requisitos y características de los datos.

4.4. Ejemplo

Datos de Entrenamiento:

Test:

Categoría	Texto
Negative	Just plain boring, entirely predictable and lacks energy.
Negative	No surprises and very few laughs.
Positive	Very powerful.
Positive	The most fun film of the summer.

Categoría	Texto
?	Predictable with no fun.

4.5. Naive Bayes como modelo de lenguaje

Cuando utilizamos características de palabras individuales y consideramos todas las palabras en el texto, el naive Bayes tiene una similitud importante con la modelización del lenguaje.

Específicamente, un modelo naive Bayes se puede ver como un conjunto de modelos de lenguaje de unigramas específicos de cada clase, en el que el modelo para cada clase instancia un modelo de lenguaje de unígrafo.

Las características de verosimilitud del modelo naive Bayes asignan una probabilidad a cada palabra $P(\text{word}|c)$, y el modelo también asigna una probabilidad a cada oración:

$$P(s|c) = \prod_{i \in \text{positions}} P(w_i|c)$$

Consideremos un modelo naive Bayes con las clases positiva (+) y negativa (-) y los siguientes parámetros del modelo:

w	$P(w +)$	$P(w -)$
I	0.1	0.2
love	0.1	0.001
this	0.01	0.01
fun	0.05	0.005
film	0.1	0.1
...

Cada una de las dos columnas anteriores instancian un modelo de lenguaje que puede asignar una probabilidad a la oración "I love this fun film":

$$P("I \text{ love this fun film"} | +) = 0.1 \times 0.1 \times 0.01 \times 0.05 \times 0.1 = 0.0000005$$

$$P("I \text{ love this fun film"} | -) = 0.2 \times 0.001 \times 0.01 \times 0.005 \times 0.1 = 0.000000010$$

Como sucede, el modelo positivo asigna una probabilidad más alta a la oración:

$$P(s|\text{pos}) > P(s|\text{neg})$$

	Cat	Documents
Training	-	just plain boring entirely predictable and lacks energy no surprises and very few laughs very powerful the most fun film of the summer
Test	?	predictable with no fun

3. Likelihoods from training:

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

$$P(\text{"predictable"}|-) = \frac{1+1}{14+20} \quad P(\text{"predictable"}|+) = \frac{0+1}{9+20}$$

$$P(\text{"no"}|-) = \frac{1+1}{14+20} \quad P(\text{"no"}|+) = \frac{0+1}{9+20}$$

$$P(\text{"fun"}|-) = \frac{0+1}{14+20} \quad P(\text{"fun"}|+) = \frac{1+1}{9+20}$$

1. Prior from training:

$$\hat{P}(c_j) = \frac{N_{c_j}}{N_{total}} \quad P(-) = 3/5 \quad P(+) = 2/5$$

2. Drop "with"

4. Scoring the test set:

$$P(-)P(S|-) = \frac{3}{5} \times \frac{2 \times 2 \times 1}{34^3} = 6.1 \times 10^{-5}$$

$$P(+)P(S|+) = \frac{2}{5} \times \frac{1 \times 1 \times 2}{29^3} = 3.2 \times 10^{-5}$$

Cabe destacar que esto es solo la parte de verosimilitud del modelo naive Bayes; una vez que multiplicamos por la probabilidad a priori, un modelo naive Bayes completo podría tomar una decisión de clasificación diferente.

4.6. Evaluation

- Let's consider just binary text classification tasks.
- Imagine you're the CEO of Delicious Pie Company.
- You want to know what people are saying about your pies.
- So you build a "Delicious Pie" tweet detector with the following classes:
 - Positive class: tweets about Delicious Pie Co
 - Negative class: all other tweets

4.6.1. The 2-by-2 Confusion Matrix

	System Positive	System Negative
Gold Positive	True Positive (TP)	False Negative (FN)
Gold Negative	False Positive (FP)	True Negative (TN)

Recall (also known as **Sensitivity** or **True Positive Rate**):

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

Precision:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

4.6.2. Evaluation: Accuracy

Why don't we use accuracy as our metric?

Imagine we saw 1 million tweets:

- 100 of them talked about Delicious Pie Co.
- 999,900 talked about something else.

We could build a dumb classifier that just labels every tweet "not about pie":

- It would get 99.99 % accuracy!!! Wow!!!!
- But it would be useless! It doesn't return the comments we are looking for!

That's why we use precision and recall instead.

4.6.3. Evaluation: Precision and Recall

Precision measures the percentage of items the system detected (i.e., items the system labeled as positive) that are in fact positive (according to the human gold labels).

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall measures the percentage of items that were correctly identified by the system out of all the items that should have been identified.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

4.6.4. Why Precision and Recall?

Consider our dumb pie-classifier that just labels nothing as ".about pie."

- Accuracy = 99.99 % (it correctly labels most tweets as not about pie)
- Recall = 0 (it doesn't detect any of the 100 pie-related tweets)

Precision and recall, unlike accuracy, emphasize true positives:

- They focus on finding the things that we are supposed to be looking for.

4.6.5. A Combined Measure: F-measure

The F-measure is a single number that combines precision (P) and recall (R), defined as:

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

The F-measure, defined with the parameter β , differentially weights the importance of recall and precision.

- $\beta > 1$ favors recall
- $\beta < 1$ favors precision

When $\beta = 1$, precision and recall are equal, and we have the balanced F_1 measure:

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

4.6.6. Development Test Sets ("Devsets")

- To avoid overfitting and provide a more conservative estimate of performance, we commonly use a three-set approach: training set, devset, and testset.

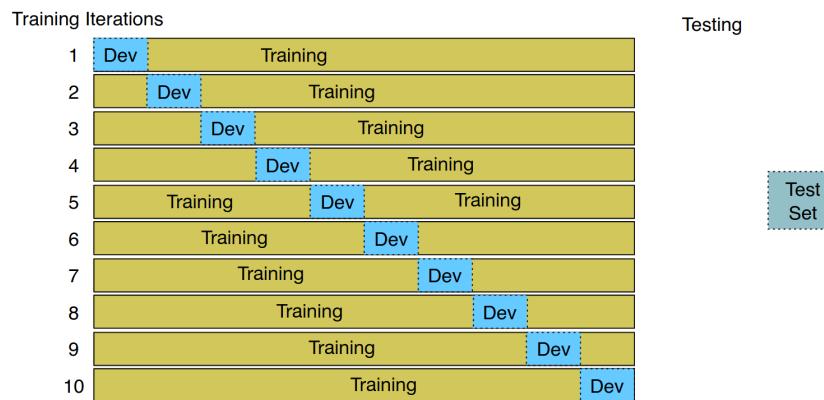


- **Training set:** Used to train the model.
- **Devset:** Used to tune the model and select the best hyperparameters.
- **Testset:** Used to report the final performance of the model.
- This approach ensures that the model is not tuned specifically to the test set, avoiding overfitting.
- However, it creates a paradox: we want as much data as possible for training, but also for the devset.
- How do we split the data?

4.6.7. Cross-validation: Multiple Splits

- Cross-validation allows us to use all our data for training and testing without having a fixed training set, devset, and test set.
- We choose a number k and partition our data into k disjoint subsets called folds.

- For each iteration, one fold is selected as the test set while the remaining $k - 1$ folds are used to train the classifier.
- We compute the error rate on the test set and repeat this process k times.
- Finally, we average the error rates from these k runs to obtain an average error rate.
- 10-fold cross-validation, for example, involves training 10 models on 90 % of the data and testing each model separately.
- The resulting error rates are averaged to obtain the final performance estimate.
- However, cross-validation requires the entire corpus to be blind, preventing examination of the data for feature suggestion or understanding system behavior.
- To address this, a fixed training set and test set are created, and 10-fold cross-validation is performed within the training set.
- The error rate is computed conventionally in the test set.



4.6.8. Confusion Matrix for 3-class classification

		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	$\text{precision}_s = \frac{200}{3+30+200}$
		$\text{recall}_u = \frac{8}{8+5+3}$	$\text{recall}_n = \frac{60}{10+60+30}$	$\text{recall}_s = \frac{200}{1+50+200}$	

How to combine binary metrics (Precision, Recall, F_1) from more than 2 classes to get one metric:

- Macroaveraging:

- Compute the performance metrics (Precision, Recall, F_1) for each class individually.
- Average the metrics over all classes.

- Microaveraging:

- Collect the decisions for all classes into one confusion matrix.
- Compute Precision and Recall from the confusion matrix.

Class 1: Urgent			Class 2: Normal			Class 3: Spam			Pooled		
system	true	true	system	true	true	system	true	true	system	true	true
urgent	urgent	not	normal	normal	not	spam	spam	not	yes	yes	no
system	8	11	system	60	55	system	200	33	system	268	99
urgent			normal			spam			yes		
system	8	340	system	40	212	system	51	83	system	99	635
not			not			not			no		

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

Capítulo 5

Modelos Lineales

5.1. Supervised Learning

- The essence of supervised machine learning is the creation of mechanisms that can look at examples and produce generalizations. [Goldberg, 2017]
- We design an algorithm whose input is a set of labeled examples, and its output is a function (or a program) that receives an instance and produces the desired label.
- Example: if the task is to distinguish from spam and not-spam email, the labeled examples are emails labeled as spam and emails labeled as not-spam.
- It is expected that the resulting function will produce correct label predictions also for instances it has not seen during training.
- This approach differs from designing an algorithm to perform the task (e.g., manually designed rule-based systems).

5.1.1. Parameterized Functions

- Searching over the set of all possible functions is a very hard (and rather ill-defined) problem. [Goldberg, 2017]
- We often restrict ourselves to search over specific families of functions.
- Example: the space of all linear functions with d_{in} inputs and d_{out} outputs,
- Such families of functions are called **hypothesis classes**.
- By restricting ourselves to a specific hypothesis class, we are injecting the learner with **inductive bias**.

- Inductive bias: a set of assumptions about the form of the desired solution.
- Some hypothesis classes facilitate efficient procedures for searching for the solution. [Goldberg, 2017]

5.2. Linear Models

- One common hypothesis class is that of high-dimensional linear function:

$$f(\vec{x}) = \vec{x} \cdot W + \vec{b} \quad (5.1)$$

$$\vec{x} \in \mathcal{R}^{d_{in}} \quad W \in \mathcal{R}^{d_{in} \times d_{out}} \quad \vec{b} \in \mathcal{R}^{d_{out}}$$

- The vector \vec{x} is the input to the function.
- The matrix W and the vector \vec{b} are the **parameters**.
- The goal of the learner is to set the values of the parameters W and \vec{b} such that the function behaves as intended on a collection of input values $\vec{x}_{1:k} = \vec{x}_1, \dots, \vec{x}_k$ and the corresponding desired outputs $\vec{y}_{1:k} = \vec{y}_1, \dots, \vec{y}_k$
- The task of searching over the space of functions is thus reduced to one of searching over the space of parameters. [Goldberg, 2017]

5.2.1. Example: Language Detection

- Consider the task of distinguishing documents written in English from documents written in German.
- This is a binary classification problem

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

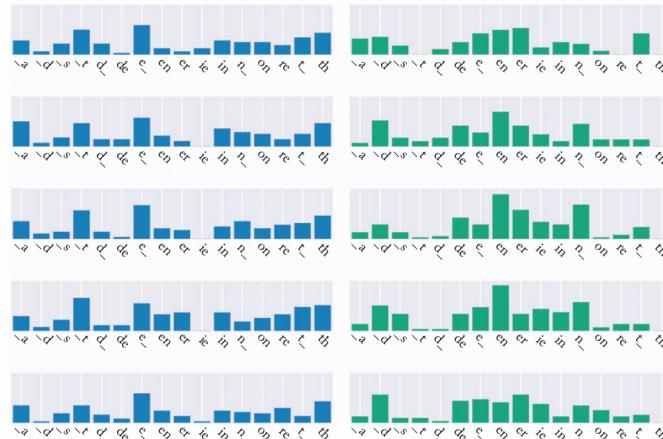
$d_{out} = 1$, \vec{w} is a vector, and b is a scalar.

- The range of the linear function is $[-\infty, \infty]$.
- In order to use it for binary classification, it is common to pass the output of $f(x)$ through the *sign* function, mapping negative values to -1 (the negative class) and non-negative values to +1 (the positive class).
- Letter frequencies make for quite good predictors (features) for this task.
- Even more informative are counts of letter bigrams, i.e., pairs of consecutive letters.
- One may think that words will also be good predictors i.e., using a bag of word representation of documents.

- Letters, or letter-bigrams are far more robust.
- We are likely to encounter a new document without any of the words we observed in the training set.
- While a document without any of the distinctive letter-bigrams is significantly less likely. [Goldberg, 2017]
- We assume we have an alphabet of 28 letters (a–z, space, and a special symbol for all other characters including digits, punctuations, etc.)
- Documents are represented as 28×28 dimensional vectors $\vec{x} \in \mathcal{R}^{784}$.
- Each entry $\vec{x}_{[i]}$ represents a count of a particular letter combination in the document, normalized by the document's length.
- For example, denoting by \vec{x}_{ab} the entry of \vec{x} corresponding to the letter bigram ab :

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

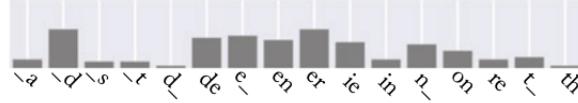
where $\#ab$ is the number of times the bigram ab appears in the document, and $|D|$ is the total number of bigrams in the document (the document's length).



Character-bigram histograms for documents in English (left, blue) and German (right, green). Underscores denote spaces. Only the top frequent character-bigrams are showed.

- Previous figure showed clear patterns in the data, and, given a new item, such as:

⁰Source:[Goldberg, 2017]



- We could probably tell that it is more similar to the German group than to the English one (observe the frequency of “th” and “ie”).
- We can’t use a single definite rule such as “if it has th its English” or “if it has ie its German”.
- While German texts have considerably less “th” than English, the “th” may and does occur in German texts, and similarly the “ie” combination does occur in English.
- The decision requires weighting different factors relative to each other.
- We can formalize the problem in a machine-learning setup using a linear model:

$$\begin{aligned}\hat{y} &= \text{sign}(f(\vec{x})) = \text{sign}(\vec{x} \cdot \vec{w} + b) \\ &= \text{sign}(\vec{x}_{aa} \times \vec{w}_{aa} + \vec{x}_{ab} \times \vec{w}_{ab} + \vec{x}_{ac} \times \vec{w}_{ac} \dots + b)\end{aligned}\tag{5.4}$$

- A document will be considered English if $f(\vec{x}) \geq 0$ and as German otherwise.

Intuition

1. Learning should assign large positive values to \vec{w} entries associated with letter pairs that are much more common in English than in German (i.e., “th”).
2. It should also assign negative values to letter pairs that are much more common in German than in English (“ie”, “en”).
3. Finally, it should assign values around zero to letter pairs that are either common or rare in both languages.

5.3. Log-linear Binary classification

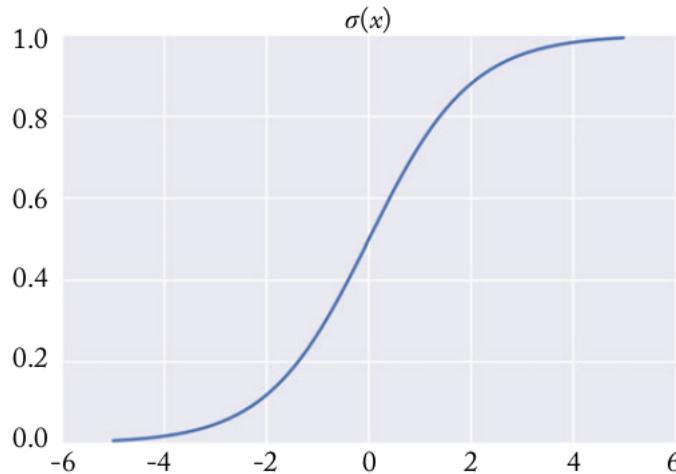
- The output $f(\vec{x})$ is in the range $[-\infty, \infty]$, and we map it to one of two classes $\{-1, +1\}$ using the *sign* function.
- This is a good fit if all we care about is the assigned class.

- We may be interested also in the confidence of the decision, or the probability that the classifier assigns to the class.
- An alternative that facilitates this is to map instead to the range $[0, 1]$, by pushing the output through a squashing function such as the sigmoid $\sigma(x)$:

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

resulting in:

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



- The sigmoid function is monotonically increasing, and maps values to the range $[0, 1]$, with 0 being mapped to $\frac{1}{2}$.
- When used with a suitable loss function (discussed later) the binary predictions made through the log-linear model can be interpreted as class membership probability estimates:

$$\sigma(f(\vec{x})) = P(\hat{y} = 1|\vec{x}) \quad \text{of } \vec{x} \text{ belonging to the positive class.} \quad (5.7)$$

- We also get $P(\hat{y} = 0|\vec{x}) = 1 - P(\hat{y} = 1|\vec{x}) = 1 - \sigma(f(\vec{x}))$
- The closer the value is to 0 or 1 the more certain the model is in its class membership prediction, with the value of 0.5 indicating model uncertainty.

5.4. Multi-class Classification

- Most classification problems are of a multi-class nature: examples are assigned to one of k different classes.
- Example: we are given a document and asked to classify it into one of six possible languages: English, French, German, Italian, Spanish, Other.
- Possible solution: consider six weight vectors $\vec{w}_{EN}, \vec{w}_{FR}, \dots$ and biases (one for each language).
- Predict the language resulting in the highest score:

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

- The six sets of parameters $\vec{w}_L \in \mathcal{R}^{784}$ and b_L can be arranged as a matrix $W \in \mathcal{R}^{784 \times 6}$ and vector $\vec{b} \in \mathcal{R}^6$, and the equation re-written as:

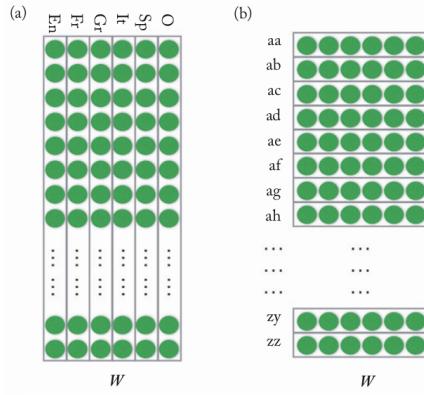
$$\begin{aligned} \vec{y} &= f(\vec{x}) = \vec{x} \cdot W + \vec{b} \\ \text{prediction} &= \hat{y} = \operatorname{argmax}_i \vec{y}_{[i]} \end{aligned} \quad (5.9)$$

- Here $\vec{y} \in \mathcal{R}^6$ is a vector of the scores assigned by the model to each language, and we again determine the predicted language by taking the argmax over the entries of \vec{y} (the columns with the highest value).

5.5. Representations

- Consider the vector \vec{y} resulting from applying a trained model to a document.
- The vector can be considered as a representation of the document.
- It captures the properties of the document that are important to us: the scores of the different languages.
- The representation \vec{y} contains strictly more information than the prediction $\operatorname{argmax}_i \vec{y}_{[i]}$.
- Example: \vec{y} can be used to distinguish documents in which the main language is German, but which also contain a sizeable amount of French words.
- Clustering documents based on \vec{y} could help to discover documents written in regional dialects, or by multilingual authors.
- The vectors \vec{x} containing the normalized letter-bigram counts for the documents are also representations of the documents.

- Arguably containing a similar kind of information to the vectors \vec{y} .
- However, the representations in \vec{y} is more compact (6 entries instead of 784) and more specialized for the language prediction objective.
- Clustering by the vectors \vec{x} would likely reveal document similarities that are not due to a particular mix of languages, but perhaps due to the document's topic or writing styles.
- The trained matrix $W \in \mathcal{R}^{784 \times 6}$ can also be considered as containing learned representations.
- We can consider two views of W , as rows or as columns. Two views of



the W matrix. (a) Each column corresponds to a language. (b) Each row corresponds to a letter bigram. Source: [Goldberg, 2017].

- A column of W can be taken to be a 784-dimensional vector representation of a language in terms of its characteristic letter-bigram patterns.
- We can then cluster the 6 language vectors according to their similarity.
- Each of the 784 rows of W provide a 6-dimensional vector representation of that bigram in terms of the languages it prompts.
- Representations are central to deep learning.
- One could argue that the main power of deep-learning is the ability to learn good representations.
- In the linear case, the representations are interpretable.
- We can assign a meaningful interpretation to each dimension in the representation vector.
- For example: each dimension corresponds to a particular language or letter-bigram.

- Deep learning models, on the other hand, often learn a cascade of representations of the input that build on top of each other.
- These representations are often not interpretable.
- We do not know which properties of the input they capture.
- However, they are still very useful for making predictions.

5.6. One-Hot Vector Representation

- The input vector \vec{x} in our language classification example contains the normalized bigram counts in the document D .
- This vector can be decomposed into an average of $|D|$ vectors, each corresponding to a particular document position i :

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

- Here, $D_{[i]}$ is the bigram at document position i .
- Each vector $\vec{x}^{D_{[i]}} \in \mathcal{R}^{784}$ is a one-hot vector.
- A one-hot vector: all entries are zero except the single entry corresponding to the letter bigram $D_{[i]}$, which is 1.
- The resulting vector \vec{x} is commonly referred to as an averaged bag of bigrams (more generally averaged bag of words , or just bag of words).
- Bag-of-words (BOW) representations contain information about the identities of all the “words” (here, bigrams) of the document, without considering their order.
- A one-hot representation can be considered as a bag-of-a-single-word.

Rome	Paris		word V
Rome	→	→	→
= [1, 0, 0, 0, 0, 0, ..., 0]			
Paris	= [0, 1, 0, 0, 0, 0, ..., 0]		
Italy	= [0, 0, 1, 0, 0, 0, ..., 0]		
France	= [0, 0, 0, 1, 0, 0, ..., 0]		

One-hot vectors of words. Source: <https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7>.

5.7. Log-linear Multi-class Classification

- In the binary case, we transformed the linear prediction into a probability estimate by passing it through the sigmoid function, resulting in a log-linear model.
- The analog for the multi-class case is passing the score vector through the **softmax** function:

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

Resulting in:

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

- The softmax transformation forces the values in \hat{y} to be positive and sum to 1, making them interpretable as a probability distribution.

5.8. Training

- When training a parameterized function (e.g., a linear model, a neural network) one defines a loss function $L(\hat{y}, y)$, stating the loss of predicting \hat{y} when the true output is y .

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

- We use the symbol Θ to denote all the parameters of the model (e.g., W, \vec{b})
- The training objective is then to minimize the loss across the different training examples.
- Formally, a loss function $L(\hat{y}, y)$ assigns a numerical score (a scalar) to a predicted output \hat{y} given the true expected output y .
- The loss function should attain its minimum value for cases where the prediction is correct.
- We can also define a corpus-wide loss with respect to the parameters Θ as the average loss over all training examples:

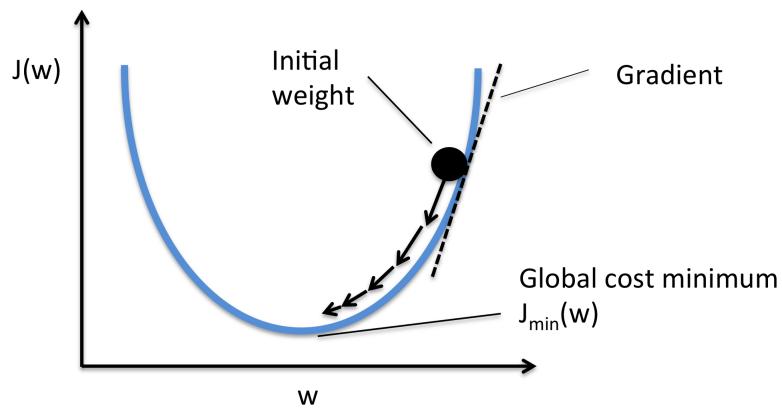
$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^n L(f(\vec{x}_i; \Theta), y_i)$$

- The goal of the training algorithm is then to set the values of the parameters Θ such that the value of \mathcal{L} is minimized.

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

5.8.1. Gradient-based Optimization

- Functions are trained using gradient-based methods.
- They work by repeatedly computing an estimate of the loss L over the training set.
- The training method computes gradients of the parameters (Θ) with respect to the loss estimate, and moving the parameters in the opposite directions of the gradient.
- Different optimization methods differ in how the error estimate is computed, and how moving in the opposite direction of the gradient is defined.
- If the function is convex, the optimum will be a global one.
- Otherwise, the process is only guaranteed to find a local optimum.



⁰Source: <https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png>
⁰[Goodfellow et al., 2016]

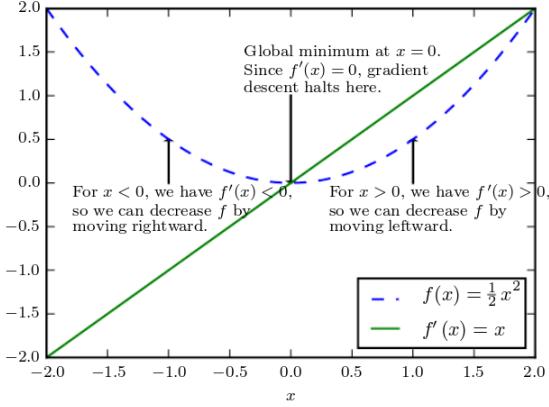


Figure 4.1: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

5.8.2. Online Stochastic Gradient Descent

- All the parameters are initialized with random values (Θ).
- For each training example (x, y) we calculate the loss L with current values of Θ .
- Then we update the parameters with the following rule until convergence:
- $\Theta_i \leftarrow \Theta_i - \eta \frac{\partial L}{\partial \Theta_i}(\hat{y}, y)$ (for all parameters Θ_i)

Algorithm 2.1 Online stochastic gradient descent training.

Input:

- Function $f(\mathbf{x}; \Theta)$ parameterized with parameters Θ .
- Training set of inputs $\mathbf{x}_1, \dots, \mathbf{x}_n$ and desired outputs y_1, \dots, y_n .
- Loss function L .

```

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

```

- The learning rate can either be fixed throughout the training process, or decay as a function of the time step t .

⁰Source:[Goldberg, 2017]

- The error calculated in line 3 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss L that we are aiming to minimize.
- The noise in the loss computation may result in inaccurate gradients (single examples may provide noisy information).

5.8.3. Mini-batch Stochastic Gradient Descent

- A common way of reducing this noise is to estimate the error and the gradients based on a sample of m examples.
- This gives rise to the minibatch SGD algorithm

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:

- Function $f(x; \Theta)$ parameterized with parameters Θ .
- 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:      $\hat{g} \leftarrow \hat{g} + \text{gradients of } \frac{1}{m} L(f(x_i; \Theta), y_i) \text{ w.r.t } \Theta$ 
7:    $\Theta \leftarrow \Theta - \eta \cdot \hat{g}$ 
8: return  $\Theta$ 

```

- Higher values of m provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence.
- For modest sizes of m , some computing architectures (i.e., GPUs) allow an efficient parallel implementation of the computation in lines 3-6.

5.8.4. Loss Functions

- Hinge (or SVM loss): for binary classification problems, the classifier's output is a single scalar \tilde{y} and the intended output y is in $\{+1, -1\}$. The classification rule is $\hat{y} = \text{sign}(\tilde{y})$, and a classification is considered correct if $y \cdot \tilde{y} > 0$.

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

- Binary cross entropy (or logistic loss): is used in binary classification with conditional probability outputs. The classifier's output \tilde{y} is transformed using the sigmoid function to the range $[0, 1]$, and is interpreted as the conditional probability $P(y = 1|x)$.

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

⁰Source:[Goldberg, 2017]

- The logistic loss has a probabilistic interpretation:
- We assume that $P(y = 1|\vec{x}; \Theta) = \sigma(f(\vec{x})) = \frac{1}{1+e^{-\vec{x} \cdot \vec{w} + b}}$ and $P(y = 0|\vec{x}; \Theta) = 1 - \sigma(f(\vec{x}))$
- We can write this in a more compact way:

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

- The above expression is the probability mass function of the Bernoulli distribution.
- Now we replace $\sigma(f(\vec{x}))$ by \hat{y} :

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

- If we perform maximum likelihood estimation to this expression we would apply a logarithm function and maximize the parameters Θ :

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

- Maximizing this expression is equivalent to minimizing the logistic loss!
- Many loss functions correspond to the negative log-likelihood of probabilistic models!
- Categorical cross-entropy loss: is used when a probabilistic interpretation of multi-class scores is desired. It measures the dissimilarity between the true label distribution \vec{y} and the predicted label distribution $\vec{\hat{y}}$.

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

- When using the cross-entropy loss, it is assumed that the classifier's output is transformed using the softmax transformation.
- The softmax function squashes the k -dimensional output to values in the range (0,1) with all entries adding up to 1. Hence, $\vec{\hat{y}}_{[i]} = P(y = i|x)$ represent the class membership conditional distribution.
- For hard-classification problems in which each training example has a single correct class assignment, \vec{y} is a one-hot vector representing the true class.
- In such cases, the cross entropy can be simplified to:

$$L_{\text{cross-entropy(hard classification)}}(\vec{\hat{y}}, \vec{y}) = - \log(\vec{\hat{y}}_{[t]})$$

where t is the correct class assignment.

5.9. Regularization

- Our optimization problem may admit multiple solutions, and, especially in higher dimensions, it can also over-fit.
- Scenario: In our language identification problem one of the documents in the training set (\vec{x}_O) is an outlier.
- The document is actually in German, but is labeled as French.
- In order to drive the loss down, the learner can identify features (letter bigrams) in \vec{x}_O that occur in only few other documents.
- The learner will give these features very strong weights toward the (incorrect) French class.
- This is a bad solution to the learning problem.
- The model is learning something incorrect.
- Test German documents which share many words with \vec{x}_O could be mistakenly classified as French.
- We would like to control for such cases by driving the learner away from such misguided solutions.
- Idea: it is OK to mis-classify a few examples if they don't fit well with the rest.
- Regularization: we add a regularization term R to the optimization objective.
- The goal of this term: to control the complexity (large weights) of the parameter value (Θ), and avoid cases of overfitting:

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

- The regularization term considers the parameter values, and scores their complexity.
- The value of hyperparameter λ has to be set manually, based on the classification performance on a development set.
- In practice, the regularizers R equate complexity with large weights.
- They work to keep the parameter values (Θ) low.
- They drive the learner toward solutions with low norms of the parameter matrices (W).
- Common choices for R are the L_2 norm, the L_1 norm, and the elastic-net.

5.9.1. L₂ Regularization

- In L₂ regularization, R takes the form of the squared L₂ norm of the parameters.
- Goal: try to keep the sum of the squares of the parameter values low.

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

- The L₂ regularizer is also called a Gaussian prior or weight decay.
- L₂ regularized models are severely punished for high parameter weights.
- Once the value is close enough to zero, their effect becomes negligible.
- The model will prefer to decrease the value of one parameter with high weight by 1 than to decrease the value of ten parameters that already have relatively low weights by 0.1 each.

5.9.2. L₁ Regularization

- In L₁ regularization, R takes the form of the L₁ norm of the parameters.
- Goal: try to keep the sum of the absolute values of the parameters low.

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

- In contrast to L₂, the L₁ regularizer is punished uniformly for low and high values.
- It has an incentive to decrease all the non-zero parameter values toward zero.
- It thus encourages a sparse solutions—models with many parameters with a zero value.
- The L₁ regularizer is also called a sparse prior or lasso [Tibshirani, 1996].

5.9.3. Elastic-Net

- The elastic-net regularization [Zou and Hastie, 2005] combines both L₁ and L₂ regularization:

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

5.10. Beyond SGD

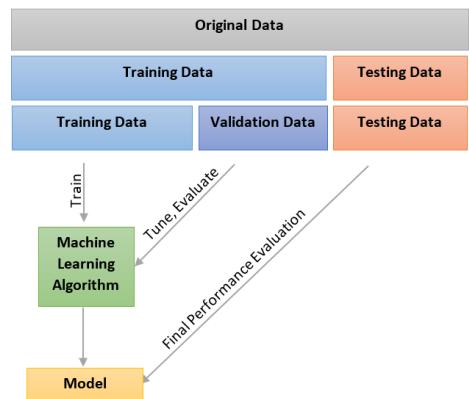
- While the SGD algorithm can and often does produce good results, more advanced algorithms are also available.
- The SGD+Momentum [Polyak, 1964] and Nesterov Momentum [Nesterov, 2018, Sutskever et al., 2013] algorithms are variants of SGD in which previous gradients are accumulated and affect the current update.
- Adaptive learning rate algorithms including AdaGrad [Duchi et al., 2011], AdaDelta [Zeiler, 2012], RMSProp [Tieleman and Hinton, 2012], and Adam [Kingma and Ba, 2014] are designed to select the learning rate for each minibatch.
- This sometimes done on a per-coordinate basis, potentially alleviating the need of fiddling with learning rate scheduling.
- For details of these algorithms, see the original papers or [Goodfellow et al., 2016](Sections 8.3, 8.4).

5.11. Train, Test, and Validation Sets

- When training a model our goal is to produce a function $f(\vec{x})$ that correctly maps inputs \vec{x} to outputs \hat{y} as evidenced by the training set.
- Performance on training data can be misleading: our goal is to train a function capable of generalizing to unseen examples.
- Held-out set: split training set into training and testing subsets (80 % and 20 % splits). Train on training and compute accuracy on testing.
- Problem: in practice you often train several models, compare their quality, and select the best one.
- Selecting the best model according to the held-out set's accuracy will result in an overly optimistic estimate of the model's quality.
- You don't know if the chosen settings of the final classifier are good in general, or are just good for the particular examples in the held-out sets.
- The accepted methodology is to use a three-way split of the data into train, validation (also called development), and test sets¹.
- This gives you two held-out sets: a validation set (also called development set), and a test set.

¹An alternative approach is cross-validation, but it doesn't scale well for training deep neural networks.

- All the experiments, tweaks, error analysis, and model selection should be performed based on the validation set.
- Then, a single run of the final model over the test set will give a good estimate of its expected quality on unseen examples.
- It is important to keep the test set as pristine as possible, running as few experiments as possible on it.
- Some even advocate that you should not even look at the examples in the test set, so as to not bias the way you design your model.



5.12. A limitation of linear models: the XOR problem

- The hypothesis class of linear (and log-linear) models is severely restricted.
- For example, it cannot represent the XOR function, defined as:

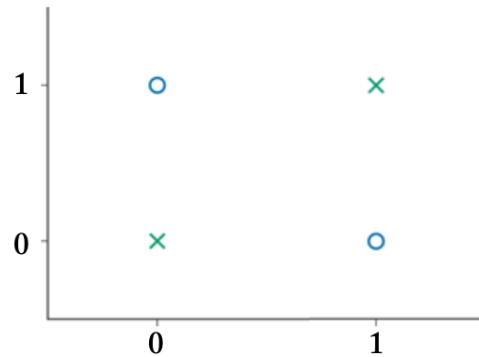
$$\begin{aligned}
 \text{xor}(0, 0) &= 0 \\
 \text{xor}(1, 0) &= 1 \\
 \text{xor}(0, 1) &= 1 \\
 \text{xor}(1, 1) &= 0
 \end{aligned} \tag{5.14}$$

¹source: <https://www.codeproject.com/KB/AI/1146582/validation.PNG>

- There is no parameterization $\vec{w} \in \mathcal{R}^2, b \in \mathcal{R}$ such that:

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

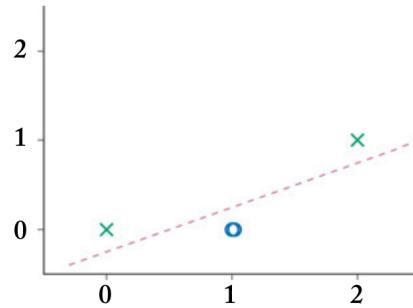
- To see why, consider the following plot of the XOR function, where blue Os denote the positive class and green Xs the negative class.



- It is clear that no straight line can separate the two classes.

5.12.1. Nonlinear input transformations

- If we transform the points by feeding each of them through the nonlinear function $\phi(x_1, x_2) = [x_1 \times x_2, x_1 + x_2]$, the XOR problem becomes linearly separable.



- The function ϕ mapped the data into a representation that is suitable for linear classification.

- We can now easily train a linear classifier to solve the XOR problem.

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

- Problem: we need to manually define the function ϕ .
- This process is dependent on the particular dataset, and requires a lot of human intuition.
- Solution: define a trainable nonlinear mapping function, and train it in conjunction with the linear classifier.
- Finding the suitable representation becomes the responsibility of the training algorithm.

5.12.2. Trainable mapping functions

- The mapping function can take the form of a parameterized linear model.
- Followed by a nonlinear activation function g that is applied to each of the output dimensions:

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

- By taking $g(x) = \max(0, x)$ and $W' = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\vec{b}' = (-1 \ 0)$.
- We get an equivalent mapping to $[x_1 \times x_2, x_1 + x_2]$ for the our points of interest $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.
- This successfully solves the XOR problem!
- Learning both the representation function and the linear classifier on top of it at the same time is the main idea behind deep learning and neural networks.
- In fact, previous equation describes a very common neural network architecture called a multi-layer perceptron (MLP).

Capítulo 6

Redes Neuronales

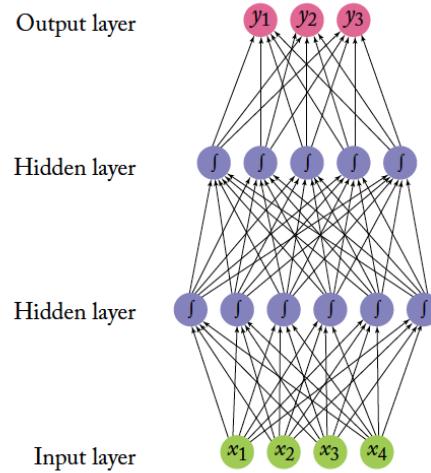
- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight w .
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also possible).
- It applies an activation function g (usually non-linear) to the result, and passes it to its output.
- Multiple layers can be stacked.
- The nonlinear activation function g has a crucial role in the network's ability to represent complex functions.
- Without the nonlinearity in g , the neural network can only represent linear transformations of the input.

Example: Feedforward Network with two Layers

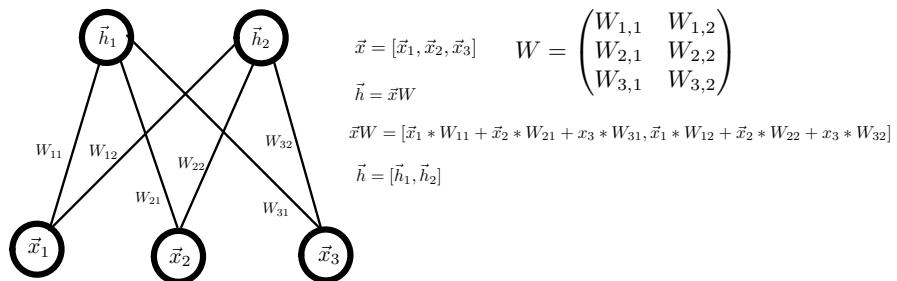
6.1. Feedforward Network Neural Networks

- The feedforward network from the picture is a stack of linear models separated by nonlinear functions.
- The values of each row of neurons in the network can be thought of as a vector.
- The input layer is a 4-dimensional vector (\vec{x}), and the layer above it is a 6-dimensional vector (\vec{h}^1).

⁰Source:[Goldberg, 2017]



- The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions.
- A fully connected layer implements a vector-matrix multiplication, $\vec{h} = \vec{x}W$.
- The weight of the connection from the i -th neuron in the input row to the j -th neuron in the output row is $W_{[i,j]}$.
- The values of \vec{h} are transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer.



Fully connected layers as vector-matrix multiplications

⁰Vectors are assumed to be row vectors and superscript indices correspond to network layers.

6.1.1. Neural Networks as Mathematical Functions

- The Multilayer Perceptron (MLP) from the figure is called MLP2 because it has two hidden layers.
- A simpler model would be MLP1, a multilayer perceptron of one hidden layer:

$$\begin{aligned}\vec{y} &= NN_{MLP1}(\vec{x}) = g(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2 \\ \vec{x} \in \mathcal{R}^{d_{in}}, W^1 \in \mathcal{R}^{d_{in} \times d_1}, \vec{b}^1 &\in \mathcal{R}^{d_1}, W^2 \in \mathcal{R}^{d_1 \times d_{out}}, \vec{b}^2 \in \mathcal{R}^{d_{out}}, \vec{y} \in \mathcal{R}^{d_{out}}\end{aligned}\tag{6.1}$$

- Here W^1 and \vec{b}^1 are a matrix and a bias term for the first linear transformation of the input.
- The function g is a nonlinear function that is applied element-wise (also called a nonlinearity or an activation function).
- W^2 and \vec{b}^2 are the matrix and bias term for a second linear transform.
- When describing a neural network, one should specify the dimensions of the layers (d_1), the input (d_{in}), and the output (d_{out}).
- MLP2 can be written as the following mathematical function:

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

- The matrices and the bias terms that define the linear transformations are the parameters of the network.
- Like in linear models, it is common to refer to the collection of all parameters as Θ .

6.2. Representation Power

- [Hornik et al., 1989] and [Cybenko, 1989] showed that a multilayer perceptron of one hidden later (MLP1) is a universal approximator.

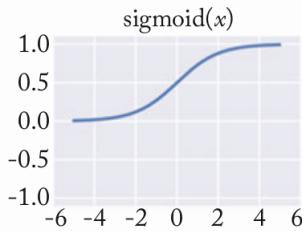
- MLP1 can approximate all continuous functions on a closed and bounded subset of \mathcal{R}^n .
- This may suggest there is no reason to go beyond MLP1 to more complex architectures.
- The result does not say how easy or hard it is to set the parameters based on training data and a specific learning algorithm.
- It also does not guarantee that a training algorithm will find the correct function generating our training data.
- Finally, it does not state how large the hidden layer should be.
- In practice, we train neural networks on relatively small amounts of data using local search methods.
- We also use hidden layers of relatively modest sizes (up to several thousands).
- The universal approximation theorem does not give any guarantees under these conditions.
- However, there is definitely benefit in trying out more complex architectures than MLP1.
- In many cases, however, MLP1 does indeed provide strong results.

6.3. Activation Functions

- The nonlinearity g can take many forms.
- There is currently no good theory as to which nonlinearity to apply in which conditions.
- Choosing the correct nonlinearity for a given task is for the most part an empirical question.

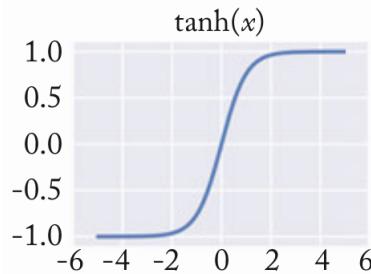
Sigmoid

- The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ is an S-shaped function, transforming each value x into the range $[0, 1]$.
- The sigmoid was the canonical nonlinearity for neural networks since their inception.
- It is currently considered to be deprecated for use in internal layers of neural networks, as the choices listed next prove to work much better empirically.



Hyperbolic tangent (tanh)

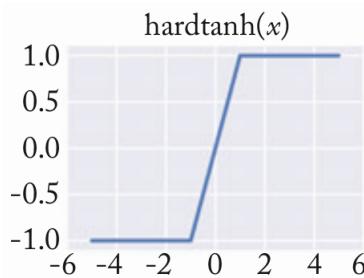
- The hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range $[-1, 1]$.



Hard tanh

- The hard-tanh activation function is an approximation of the tanh function which is faster to compute and to find derivatives thereof:

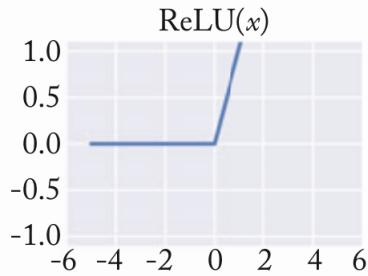
$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise.} \end{cases}$$



ReLU

- The rectifier activation function [Glorot et al., 2011], also known as the rectified linear unit is a very simple activation function.
- It is easy to work with and was shown many times to produce excellent results.
- The ReLU unit clips each value $x < 0$ at 0.

$$\text{ReLU}(x) = \max(0, x)$$



- It performs well for many tasks, especially when combined with the dropout regularization technique (to be explained later).

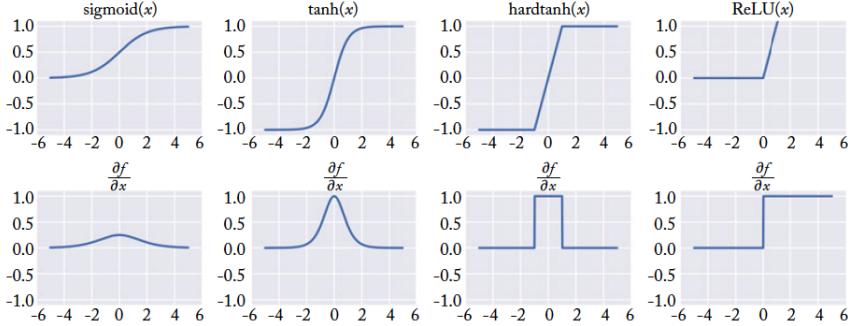
6.3.1. Practical Issues

- As a rule of thumb, both ReLU and tanh units work well, and significantly outperform the sigmoid.
- You may want to experiment with both tanh and ReLU activations, as each one may perform better in different settings.
- The figure from below shows the shapes of the different activations functions, together with the shapes of their derivatives.

6.4. Embedding Layers

- In NLP the input to the neural network contains symbolic categorical features (e.g., words from a closed vocabulary, character n-grams, POS tags).

⁰Source:[Goldberg, 2017]



- In linear models we usually represent the input with sparse vectors e.g., as the sum, average, or the concatenation of one-hot encoded vectors (the sum or the average can produce bag-of words representation).
- In neural networks, it is common to associate each possible feature value (i.e., each word in the vocabulary, each POS tag category) with a d -dimensional dense vector for some d .
- These vectors are then considered parameters of the model, and are trained jointly with the other parameters.
- The mapping from a symbolic feature values such as “word number 1249” to d -dimensional vectors is performed by an embedding layer (also called a lookup layer).
- The parameters in a word embedding layer are simply a matrix $E \in \mathcal{R}^{|vocab| \times d}$ where each row corresponds to a different word in the vocabulary.
- The lookup operation is then simply indexing: $v_{1249} = E_{[1249,:]}$.
- If the symbolic feature is encoded as a one-hot vector \vec{x} , the lookup operation can be implemented as a vector-matrix multiplication $\vec{x}E$.
- The embedding vectors are combined before being passed on to the next layer.
- Common combination operations are: concatenation, summation, average.
- A word embeddings matrix E can be initialized with pre-trained word vectors trained from unlabeled documents using specific methods based on the distributional hypothesis such as the ones implemented in Word2Vec (to be discussed later in the course).

The Embedding Matrix

One-hot-encoded word vector

$$\vec{x} = [0, 0, \dots, 1, \dots 0]^{1 \times |V|}$$

Embedding Matrix

$$E = \begin{bmatrix} -1.8 & 2.3 & \dots & 3.1 \\ \vdots & \vdots & \ddots & \vdots \\ 3.3 & -2.1 & \dots & 4.6 \\ \vdots & \vdots & \ddots & \vdots \\ 4.2 & 1.9 & \dots & -3.3 \end{bmatrix}^{|V| \times d}$$

\leftarrow abduct \leftarrow dog \leftarrow zumba

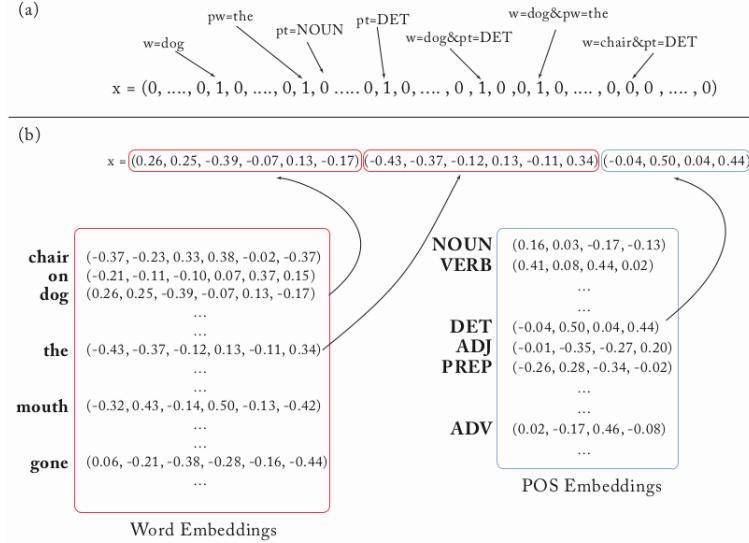
$$\vec{x}E = [3.3, -2.1, \dots, 4.6]$$

6.4.1. Dense Vectors vs. One-hot representations

- What are the benefits of representing our features as vectors instead of as unique IDs?
- Should we always represent features as dense vectors?
- Let's consider the two kinds of representations.
- 1) **One Hot:** each feature is its own dimension.
 - Dimensionality of one-hot vector is same as number of distinct features.
 - Features are completely independent from one another. The feature "word is 'dog'" is as dissimilar to "word is 'thinking'" than it is to "word is 'cat'".
- 2) **Dense:** each feature is a d -dimensional vector.
 - Dimensionality of vector is d .
 - Model training will cause similar features to have similar vectors: information is shared between similar features.

Example: Dense Vectors vs. One-hot representations

- Previous figure shows two encodings of the information: current word is "dog;" previous word is "the;" previous pos-tag is "DET."
- (a) Sparse feature vector.
 - Each dimension represents a feature.
 - Feature combinations receive their own dimensions.
 - Feature values are binary.
 - Dimensionality is very high.
- (b) Dense, embeddings-based feature vector.



- Each core feature is represented as a vector.
- Each feature corresponds to several input vector entries.
- No explicit encoding of feature combinations.
- Dimensionality is low.
- The feature-to-vector mappings come from an embedding table.

- One benefit of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors.
- However, this is just a technical obstacle, which can be resolved with some engineering effort.
- The main benefit of the dense representations is in generalization power.
- If we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities.
- Let's assume we have observed the word dog many times during training, but only observed the word cat a handful of times.
- If each of the words is associated with its own dimension (one-hot), occurrences of dog will not tell us anything about the occurrences of cat.
- However, in the dense vectors representation the learned vector for dog may be similar to the learned vector for cat.
- This will allow the model to share statistical strength between the two events.

- This argument assumes that we saw enough occurrences of the word cat such that its vector will be similar to that of dog.
- Pre-trained word embeddings (e.g., Word2Vec, Glove) to be discussed later in the course can be used to obtain dense vectors from unannotated text.

6.5. Neural Network Training

- Neural networks are trained in the same way as linear models.
- The network's output is used to compute a loss function $L(\hat{y}, y)$ that is minimized across the training examples using gradient descent.
- Backpropagation is an efficient technique for evaluating the gradient of a loss function L for a feed-forward neural network with respect to all its parameters [Bishop, 2006].¹
- Those parameters are: $W^1, \vec{b}^1, \dots, W^m, \vec{b}^m$, for a network of m layers.
- Recall that superscripts are used to denote layer indexes (not exponentiations).
- For simplicity, we will assume that L is calculated over a single example.
- Challenge: in neural networks the number of parameters can be huge and we need an efficient way to calculate the gradients.
- Idea: apply the derivative chain rule wisely.

6.6. Derivative Chain Rule Recap

- Simple chain rule: let $z = f(y), y = g(x)$,

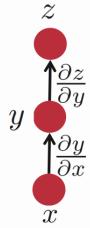
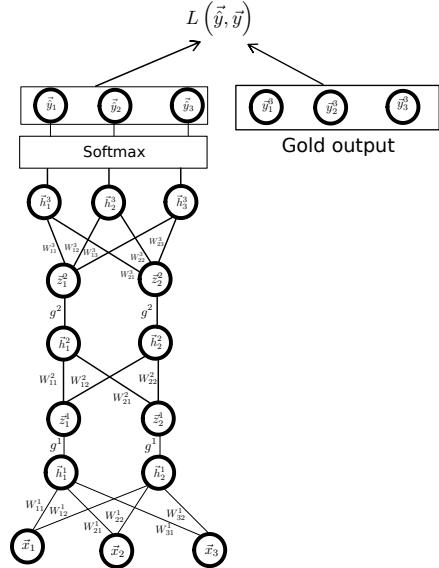
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$$

- Example: $z = e^y, y = 2x$

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

¹The following slides on backpropagation are based on [Bishop, 2006], we adapted the notation to be consistent with [Goldberg, 2017].

¹Figure taken from: <http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf>



- Multiple path chain rule: let $z = f(y_1, y_2)$, $y_1 = g_1(x)$, $y_2 = g_2(x)$

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

- Example: $z = e^{y_1 \times y_2}$, $y_1 = 2x$, $y_2 = x^2$

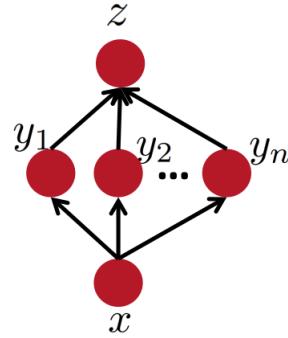
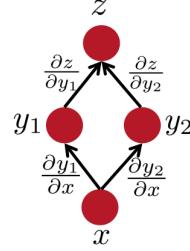
$$\frac{\partial z}{\partial x} = (e^{y_1 \times y_2} \times y_2) \times 2 + (e^{y_1 \times y_2} \times y_1) \times 2x = e^{2x^3} \times 6x^2$$

The general version of the multiple path chain rule would be:

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

¹Figure taken from: <http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf>

¹Figure taken from: <http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf>



6.7. Backpropagation

- In a general feed-forward network, each unit computes a weighted sum of its inputs in the form:

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

- The variable $\vec{z}_{[i]}^{(l-1)}$ is an input that sends a connection to unit $\vec{h}_{[j]}^l$, $W_{[i,j]}^l$ is the weight associated with that connection, and l is the layer index.
- The biases vectors $\vec{b}_{[j]}$ can be excluded from (eq.6.3) and included to the weight matrix $W_{[i,j]}^l$ by introducing an extra unit, or input, with activation fixed at +1.
- The inputs at layer l , $\vec{z}_{[i]}^{(l-1)}$ are the result of applying the activation function g to units from the previous layer:

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

- For the input layer ($l = 0$), \vec{z} corresponds to the input vector $\vec{z} = \vec{x}$

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

- For each instance in the training set, we supply the corresponding input vector \vec{x} to the network.
- Next we calculate the activations of all of the hidden and output units in the network by successive application of (eq.6.3) and (eq.6.4).
- This process is often called forward propagation because it can be regarded as a forward flow of information through the network.
- Now consider the evaluation of the derivative of L with respect to a weight $W_{[i,j]}^l$.
- Assuming that the loss L is calculated over a single example, we can note that L depends on the weight $W_{[i,j]}^l$ only via the summed input $\vec{h}_{[j]}^l$.
- We can therefore apply the chain rule for partial derivatives to give

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

- We now introduce a useful notation:

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

- Using (6.3), we can write

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

- Substituting (6.7) and (6.8) into (6.6), we then obtain

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

- Equation (6.9) tells us that the required derivative is obtained simply by multiplying the value of $\vec{\delta}_{[j]}^l$ by the value of $\vec{z}_{[i]}^{(l-1)}$.
- Thus, in order to evaluate the derivatives, we need only to calculate the value of $\vec{\delta}_{[j]}^l$ for each hidden and output unit in the network, and then apply (6.9).
- Calculating $\vec{\delta}_{[j]}^m$ for output units ($l = m$), is usually straightforward, since activation units $\vec{h}_{[j]}^m$ are directly observed in the loss expression.
- The same applies for shallow linear models.

- To evaluate the $\vec{\delta}_{[j]}^l$ for hidden units, we again make use of the chain rule for partial derivatives:

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

- The sum runs over all units $\vec{h}_{[k]}^{l+1}$ to which unit $\vec{h}_{[j]}^l$ sends connections.
- We assume that connections go only to consecutive layers in the network (from layer l to layer $(l+1)$).
- The units $\vec{h}_{[k]}^{l+1}$ could include other hidden units and/or output units.
- If we now substitute the definition of $\vec{\delta}_{[j]}^l$ given by (eq.6.7) into (eq.6.10), we get

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

- Now, for expression $\vec{h}_{[k]}^{l+1}$ we can go to its definition (eq.6.3):

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

- Now, we replace (eq.6.4) ($\vec{z}_{[i]}^l = g(\vec{h}_{[i]}^l)$) into previous equation and we obtain:

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

- Now when calculating $\frac{\partial \vec{h}_{[k]}^{l+1}}{\partial \vec{h}_{[j]}^l}$ all the terms in the summation where $i \neq j$ get canceled out.
- Hence:

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

- Now, if we substitute (eq.6.12) into (eq.6.11)

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

- Since $g'(\vec{h}_{[j]}^l)$ doesn't depend on k we can obtain the following backpropagation formula:

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

- Which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ 's backwards from units higher up in the network. [Bishop, 2006].

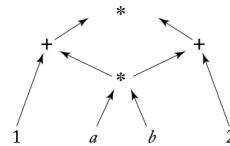
The backpropagation procedure can be summarized as follows.

1. Apply an input vector \vec{x} to the network and forward propagate through the network using (eq.6.3) and (eq.6.4) to find the activations of all the hidden and output units.
2. Evaluate the $\vec{\delta}_{[j]}^m$ for all the output units (recall that the derivatives involved here are easy to calculate).
3. Backpropagate the $\vec{\delta}_{[k]}^{(l+1)}$ using (eq.6.14) to obtain $\vec{\delta}_{[j]}^l$ for each hidden unit in the network. We go from higher to lower layers in the network.
4. Use (eq.6.9) ($\frac{\partial L}{\partial W_{[i,j]}^l} = \vec{\delta}_{[j]}^l \times \vec{z}_{[i]}^{(l-1)}$) to evaluate the required derivatives.

6.8. The Computation Graph Abstraction

- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of an arbitrary mathematical computation (e.g., a neural network) as a graph.
- This abstraction will allow us computing the gradients from any kind of neural network architecture using the backpropagation algorithm.
- Previous formulation was restricted to feedforward networks.
- A computation graph is a directed acyclic graph (DAG).
- Nodes correspond to mathematical operations or (bound) variables.
- Edges correspond to the flow of intermediary values between the nodes.

- The graph structure defines the order of the computation in terms of the dependencies between the different components.
 - The graph is a DAG and not a tree, as the result of one operation can be the input of several continuations.
 - Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



- The computation of $a * b$ is shared.
 - Since a neural network is essentially a mathematical expression, it can be represented as a computation graph.

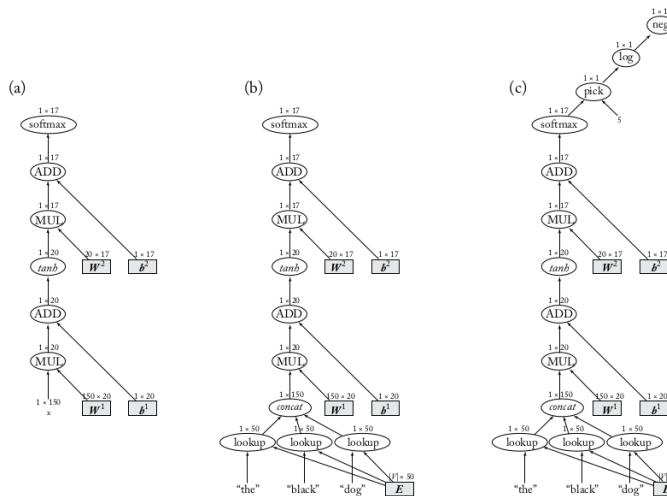


Figure 5.1: (a) Graph with unbound input. (b) Graph with concrete input. (c) Graph with concrete input, expected output, and a final loss node.

- The figure above shows the computation graph for an MLP with one hidden-layer and a softmax output transformation.
 - Oval nodes represent mathematical operations or functions, and shaded rectangle nodes represent parameters (bound variables).

¹Figure taken from: [Goldberg, 2017]

- Network inputs are treated as constants, and drawn without a surrounding node.
- Input and parameter nodes have no incoming arcs, and output nodes have no outgoing arcs.
- The output of each node is a matrix, the dimensionality of which is indicated above the node.
- This graph is incomplete: without specifying the inputs, we cannot compute an output.
- Figure 5.1b shows a complete graph for an MLP that takes three words as inputs, and predicts the distribution over part-of-speech tags for the third word.
- This graph can be used for prediction, but not for training, as the output is a vector (not a scalar) and the graph does not take into account the correct answer or the loss term.
- Finally, the graph in Figure 5.1c shows the computation graph for a specific training example, in which the inputs are the (embeddings of) the words “the,” “black,” “dog,” and the expected output is “NOUN” (whose index is 5).
- The pick node implements an indexing operation, receiving a vector and an index (in this case, 5) and returning the corresponding entry in the vector.

6.8.1. Forward Computation

- The forward pass computes the outputs of the nodes in the graph.
- Since each node’s output depends only on itself and on its incoming edges, it is trivial to compute the outputs of all nodes.
- This is done by traversing the nodes in a topological order and computing the output of each node given the already computed outputs of its predecessors.
- More formally, in a graph of N nodes, we associate each node with an index i according to their topological ordering.
- Let f_i be the function computed by node i (e.g., multiplication, addition , etc.).
- Let $\pi(i)$ be the parent nodes of node i , and $\pi^{-1}(i) = \{j|i \in \pi(j)\}$ the children nodes of node i (these are the arguments of f_i).
- Denote by $v(i)$ the output of node i , that is, the application of f_i to the output values of its arguments $\pi^{-1}(i)$.

- For variable and input nodes, f_i is a constant function and $\pi^{-1}(i)$ is empty.
- The computation-graph forward pass computes the values $v(i)$ for all $i \in [1, N]$.

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. Backward Computation (Backprop)

- The backward pass begins by designating a node N with scalar (1×1) output as a loss-node, and running forward computation up to that node.
- The backward computation computes the gradients of the parameters with respect to that node's value.
- Denote by $d(i)$ the quantity $\frac{\partial N}{\partial i}$.
- The backpropagation algorithm is used to compute the values $d(i)$ for all nodes i .
- The backward pass fills a table of values $d(1), \dots, d(N)$ as shown in the following algorithm.

Algorithm 5.4 Computation graph backward pass (backpropagation).

```

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

- The backpropagation algorithm is essentially following the chain-rule of differentiation.
- The quantity $\frac{\partial f_j}{\partial i}$ is the partial derivative of $f_j(\pi^{-1}(j))$ w.r.t the argument $i \in \pi^{-1}(j)$.
- This value depends on the function f_j and the values $v(a_1), \dots, v(a_m)$ (where $a_1, \dots, a_m = \pi^{-1}(j)$) of its arguments, which were computed in the forward pass.
- Thus, in order to define a new kind of node, one needs to define two methods: one for calculating the forward value $v(i)$ based on the node's inputs, and the another for calculating $\frac{\partial f_j}{\partial i}$ for each $x \in \pi^{-1}(i)$.

6.8.3. Summary of the Computation Graph Abstraction

- Notice that the above formulation of backpropagation is equivalent to one given earlier in the class.
- The computation graph abstraction allows us to:
 1. Easily construct arbitrary networks.
 2. Evaluate their predictions for given inputs (forward pass)
 3. Compute gradients for their parameters with respect to arbitrary scalar losses (backward pass or backpropagation).
- A nice property of the computation graph abstraction is that it allows computing the gradients for arbitrary networks (e.g., networks with skip-connections, shared weights, special loss functions, etc.)

6.8.4. Derivatives of “non-mathematical” functions

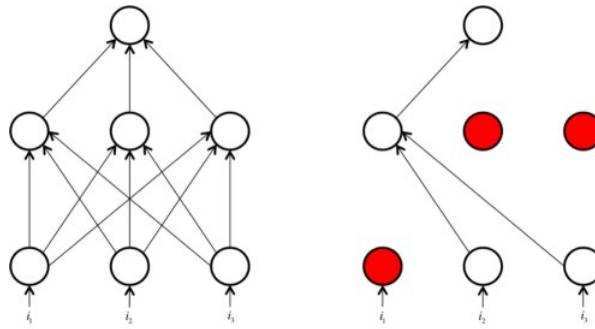
- Defining $\frac{\partial f_i}{\partial i}$ for mathematical functions such as \log or $+$ is straightforward.
- It may be challenging to think about the derivative of operations such as $\text{pick}(\vec{x}, 5)$ that selects the fifth element of a vector.
- The answer is to think in terms of the contribution to the computation.
- After picking the i -th element of a vector, only that element participates in the remainder of the computation.
- Thus, the gradient of $\text{pick}(\vec{x}, 5)$ is a vector \vec{v} with the dimensionality of \vec{x} where $\vec{v}_{[5]} = 1$ and $\vec{v}_{[i \neq 5]} = 0$.
- Similarly, for the function $\max(0, x)$ the value of the gradient is 1 for $x > 0$ and 0 otherwise.

6.9. Regularization and Dropout

- Multi-layer networks can be large and have many parameters, making them especially prone to overfitting.
- Model regularization is just as important in deep neural networks as it is in linear models, and perhaps even more so.
- The regularizers discussed for linear models, namely L_2 , L_1 , and the elastic-net, are also relevant for neural networks.

¹A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction can be found here: <https://colah.github.io/posts/2015-08-Backprop/>.

- Another effective technique for preventing neural networks from overfitting the training data is **dropout training** [Srivastava et al., 2014].
- The dropout method is designed to prevent the network from learning to rely on specific weights.
- It works by randomly dropping (setting to 0) half of the neurons in the network (or in a specific layer) in each training example in the stochastic-gradient training.



6.10. Deep Learning Frameworks

Several software packages implement the computation-graph model. All these packages support all the essential components (node types) for defining a wide range of neural network architectures.

- TensorFlow (<https://www.tensorflow.org/>): an open source software library for numerical computation using data-flow graphs originally developed by the Google Brain Team.
- Keras: High-level neural network API that runs on top of Tensorflow as well as other backends (<https://keras.io/>).
- PyTorch: open source machine learning library for Python, based on Torch, developed by Facebook's artificial-intelligence research group. It supports dynamic graph construction, a different computation graph is created from scratch for each training sample. (<https://pytorch.org/>)

¹Figure taken from: <https://www.kdnuggets.com/wp-content/uploads/drop-out-in-neural-networks.jpg>

Capítulo 7

Vectores de Palabra

- A major component in neural networks for language is the use of an embedding layer.
- A mapping of discrete symbols to continuous vectors.
- When embedding words, they transform from being isolated distinct symbols into mathematical objects that can be operated on.
- Distance between vectors can be equated to distance between words.
- This makes easier to generalize the behavior from one word to another.

7.1. Distributional Vectors

- **Distributional Hypothesis** [Harris, 1954]: words occurring in the same **contexts** tend to have similar meanings.
- Or equivalently: "a word is characterized by the **company** it keeps".
- **Distributional representations**: words are represented by **high-dimensional vectors** based on the context's where they occur.

7.1.1. Word-context Matrices

- Distributional vectors are built from word-context matrices M .
- Each cell (i, j) is a co-occurrence based association value between a **target word** w_i and a **context** c_j calculated from a corpus of documents.
- Contexts are commonly defined as windows of words surrounding w_i .
- The window length k is a parameter (between 1 and 8 words on both the left and the right sides of w_i).

- If the Vocabulary of the target words and context words is the same, M has dimensionality $|\mathcal{V}| \times |\mathcal{V}|$.
- Whereas shorter windows are likely to capture **syntactic information** (e.g, POS), longer windows are more likely to capture topical similarity [Goldberg, 2016, Jurafsky and Martin, 2008].

Distributional Vectors with context windows of size 1

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

The associations between words and contexts can be calculated using different approaches:

1. Co-occurrence counts.
2. Positive point-wise mutual information (PPMI).
3. The significance values of a paired t-test.

The most common of those according to [Jurafsky and Martin, 2008] is PPMI.

Distributional methods are also referred to as count-based methods.

7.1.2. PPMI

- PMI calculates the log of the probability of word-context pairs occurring together over the probability of them being independent.

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

⁰Example taken from: <http://cs224d.stanford.edu/lectures/CS224d-Lecture2.pdf>

- Negative PMI values suggest that the pair co-occurs less often than chance.
- These estimates are unreliable unless the counts are calculated from very large corpora [Jurafsky and Martin, 2008].
- PPMI corrects this problem by replacing negative values by zero:

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

7.1.3. Distributed Vectors or Word embeddings

- Count-based distributional vectors increase in size with vocabulary i.e., can have a very high dimensionality.
- Explicitly storing the co-occurrence matrix can be memory-intensive.
- Some classification models don't scale well to high-dimensional data.
- The neural network community prefers using **distributed representations**¹ or **word embeddings**.
- Word **embeddings** are low-dimensional continuous dense word vectors trained from document corpora using **neural networks**.
- The dimensions are not directly interpretable i.e., represent latent features of the word, "hopefully capturing useful syntactic and semantic properties" [Turian et al., 2010].
- They have become a crucial component of neural network architectures for NLP.
- There are two main approaches for obtaining word embeddings:
 1. Embedding layers: using an embedding layer in a task-specific neural network architecture trained from labeled examples (e.g., sentiment analysis).
 2. Pre-trained word embeddings: creating an auxiliary predictive task from unlabeled corpora (e.g., predict the following word) in which word embeddings will naturally arise from the neural-network architecture.
- These approaches can also be combined: one can initialize an embedding layer of a task-specific neural network with pre-trained word embeddings obtained with the second approach.
- Most popular models based on the second approach are skip-gram [Mikolov et al., 2013], continuous bag-of-words [Mikolov et al., 2013], and Glove [Pennington et al., 2014].

¹Idea: The meaning of the word is "distributed" over a combination of dimensions.

- Word embeddings have shown to be more powerful than distributional approaches in many NLP tasks [Baroni et al., 2014].
- In [Amir et al., 2015], they were used as **features** in a regression model for determining the association between Twitter words and **positive sentiment**.

7.2. Word2Vec

- Word2Vec is a software package that implements two neural network architectures for training word embeddings: Continuous Bag of Words (CBOW) and Skip-gram.
- It implements two optimization models: Negative Sampling and Hierarchical Softmax.
- These models are shallow neural networks that are trained to predict the contexts of words.
- A very comprehensive tutorial about the algorithms behind word2vec: <https://arxiv.org/pdf/1411.2738.pdf>.

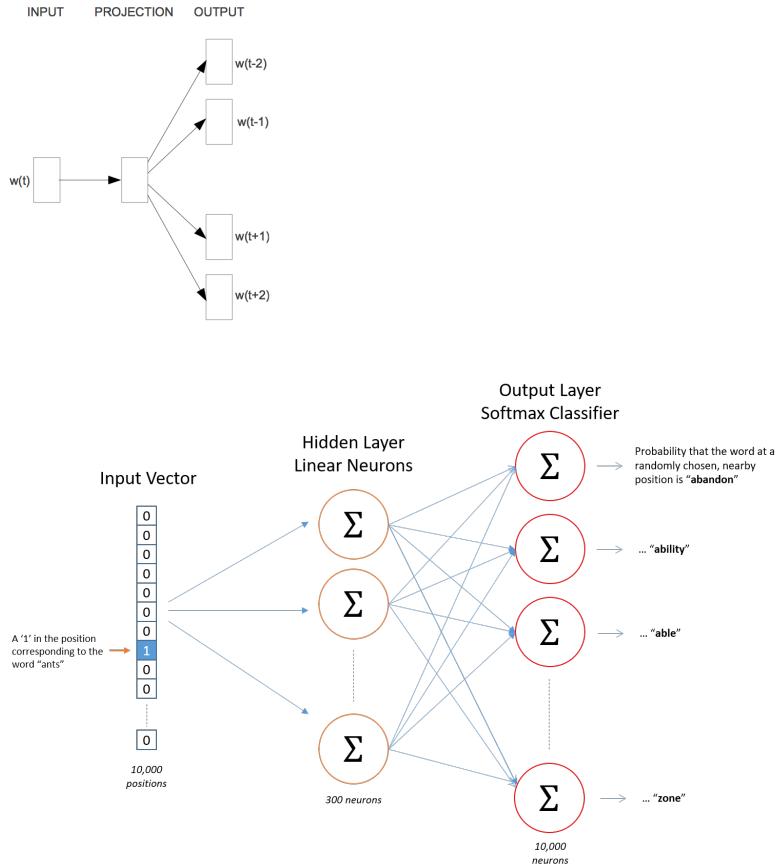
Good Word2Vec tutorial <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>

7.2.1. Skip-gram Model

- A neural network with one “projection” or “hidden” layer is trained for predicting the words surrounding a center word, within a window of size k that is shifted along the input corpus.
- The center and surrounding k words correspond to the input and output layers of the network.
- Words are initially represented by 1-hot vectors: vectors of the size of the vocabulary ($|V|$) with zero values in all entries except for the corresponding word index that receives a value of 1.
- The output layer combines the k 1-hot vectors of the surrounding words.
- The hidden layer has a dimensionality d , which determines the size of the embeddings (normally $d \ll |V|$).

We use hierarchical softmax where the vocabulary is represented as a Huffman binary tree. This follows previous observations that the frequency of words works well for obtaining classes in neural net language models [16]. Huffman

¹Picture taken from: <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>



trees assign short binary codes to frequent words, and this further reduces the number of output units that need to be evaluated

If two different words have very similar “contexts” (that is, what words are likely to appear around them), then our model needs to output very similar results for these two words. And one way for the network to output similar context predictions for these two words is if the word vectors are similar. So, if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words! Ta da!

And what does it mean for two words to have similar contexts? I think you could expect that synonyms like “intelligent” and “smart” would have very similar contexts. Or that words that are related, like “engine” and “transmission”, would probably have similar contexts as well.

This can also handle stemming for you – the network will likely learn similar word vectors for the words “ant” and “ants” because these should have similar contexts.

Source: <https://arxiv.org/pdf/1402.3722.pdf>

7.2.2. Parametrization of the Skip-gram model

- We are given an input corpus formed by a sequence of words $w_1, w_2, w_3, \dots, w_T$ and a window size k .
- We denote target or (center) words by letter w and surrounding context words by letter c .
- The context window $c_{1:k}$ of word w_t corresponds to words $w_{t-k/2}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k/2}$ (assuming that k is an even number).
- The objective of the Skip-gram model is to maximize the average log probability of the context words given the target words:

$$\frac{1}{T} \sum_{t=1}^T \sum_{c \in c_{1:k}} \log P(c|w_t)$$

- The conditional probability of a context word c given a center word w is modeled with a softmax (C is the set of all context words, which is usually the same as the vocabulary):

$$P(c|w) = \frac{e^{\vec{c} \cdot \vec{w}}}{\sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}}$$

- Model's parameters θ : \vec{c} and \vec{w} (vector representations of contexts and target words).
- Let D be the set of correct word-context pairs (i.e., word pairs that are observed in the Corpus).
- The optimization goal is to maximize the conditional log-likelihood of the contexts c (this is equivalent to minimizing the cross-entropy loss):

$$\arg \max_{\vec{c}, \vec{w}} \sum_{(w,c) \in D} \log P(c|w) = \sum_{(w,c) \in D} (\log e^{\vec{c} \cdot \vec{w}} - \log \sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}) \quad (7.3)$$

- Assumption: maximizing this function will result in good embeddings \vec{w} i.e., similar words will have similar vectors.
- The term $P(c|w)$ is computationally expensive because of the summation $\sum_{c' \in C} e^{\vec{c}' \cdot \vec{w}}$ over all the contexts c' .
- Fix: replace the softmax with a hierarchical softmax (the vocabulary is represented with a Huffman binary tree).
- Huffman trees assign short binary codes to frequent words, reducing the number of output units to be evaluated.

The distributional hypothesis states that words in similar contexts have similar meanings. The objective above clearly tries to increase the quantity for good word-context pairs, and decrease it for bad ones. Intuitively, this means that words that share many contexts will be similar to each other (note also that contexts sharing many words will also be similar to each other). This is, however, very hand-wavy.

Source: <https://arxiv.org/pdf/1402.3722.pdf>

Skip-gram and Negative Sampling are not the same

7.2.3. Skip-gram with Negative Sampling

- Negative-sampling (NS) is presented as a more efficient model for calculating skip-gram embeddings.
- However, it optimizes a different objective function [Goldberg and Levy, 2014].
- NS maximizes the probability that a word-context pair (w, c) came from the set of correct word-context pairs D using a sigmoid function:

$$P(D = 1|w, c_i) = \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}}$$

- Assumption: the contexts words c_i are independent from each other:

$$P(D = 1|w, c_{1:k}) = \prod_{i=1}^k P(D = 1|w, c_i) = \prod_{i=1}^k \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}}$$

- This leads to the following target function (log-likelihood):

$$\arg \max_{\vec{c}, \vec{w}} \log P(D = 1|w, c_{1:k}) = \sum_{i=1}^k \log \frac{1}{1 + e^{-\vec{w} \cdot \vec{c}_i}} \quad (7.4)$$

- This objective has a trivial solution if we set \vec{w}, \vec{c} such that $P(D = 1|w, c) = 1$ for every pair (w, c) from D .
- This is achieved by setting $\vec{w} = \vec{c}$ and $\vec{w} \cdot \vec{c} = K$ for all \vec{w}, \vec{c} , where K is a large number.
- We need a mechanism that prevents all the vectors from having the same value, by disallowing some (w, c) combinations.
- One way to do so, is to present the model with some (w, c) pairs for which $P(D = 1|w, c)$ must be low, i.e. pairs which are not in the data.
- This is achieved sampling negative samples from \tilde{D} .
- Sample m words for each word-context pair $(w, c) \in D$.

- Add each sampled word w_i together with the original context c as a negative example to \tilde{D} .
- \tilde{D} being m times larger than D .
- The number of negative samples m is a parameter of the algorithm.
- Final objective function:

$$\arg \max_{\vec{c}, \vec{w}} \sum_{(w,c) \in D} \log P(D = 1|w, c_{1:k}) + \sum_{(w,c) \in \tilde{D}} \log P(D = 0|w, c_{1:k}) \quad (7.5)$$

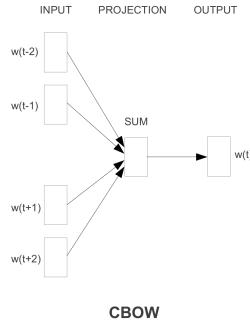
- The negative words are sampled from smoothed version of the corpus frequencies:

$$\frac{\#(w)^{0.75}}{\sum_{w'} \#(w')^{0.75}}$$

- This gives more relative weight to less frequent words.

7.2.4. Continuous Bag of Words: CBOW

Similar to the skip-gram model but now the center word is predicted from the surrounding context.



7.2.5. GloVe

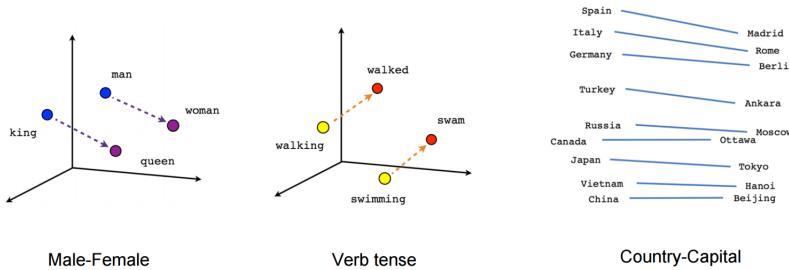
- GloVe (from global vectors) is another popular method for training word embeddings [Pennington et al., 2014].
- It constructs an explicit word-context matrix, and trains the word and context vectors \vec{w} and \vec{c} attempting to satisfy:

$$\vec{w} \cdot \vec{c} + b_{[w]} + b_{[c]} = \log \#(w, c) \quad \forall (w, c) \in D \quad (7.6)$$

- where $b_{[w]}$ and $b_{[c]}$ are word-specific and context-specific trained biases.
- In terms of matrix factorization, if we fix $b_{[w]} = \log \#(w)$ and $b_{[c]} = \log \#(c)$ we'll get an objective that is very similar to factorizing the word-context PMI matrix, shifted by $\log(|D|)$.
- In GloVe the bias parameters are learned and not fixed, giving it another degree of freedom.
- The optimization objective is weighted least-squares loss, assigning more weight to the correct reconstruction of frequent items.
- When using the same word and context vocabularies, the model suggests representing each word as the sum of its corresponding word and context embedding vectors.

7.3. Word Analogies

- Word embeddings can capture certain semantic relationships, e.g. male-female, verb tense and country-capital relationships between words.
- For example, the following relationship is found for word embeddings trained using Word2Vec: $\vec{w}_{king} - \vec{w}_{man} + \vec{w}_{woman} \approx \vec{w}_{queen}$.



²Source: <https://www.tensorflow.org/tutorials/word2vec>

7.4. Evaluation

- There are many datasets with human annotated associations of word pairs or gold analogies that can be used to evaluate word embeddings algorithms.
- Those approaches are called *Intrinsic Evaluation Approaches*.
- Most of them are implemented in: <https://github.com/kudkudak/word-embeddings-benchmarks>.

- Word embeddings can also be evaluated extrinsically by using them in an external NLP task (e.g., POS tagging, sentiment analysis).

7.5. Correspondence between Distributed and Distributional Models

- Both the distributional “count-based” methods and the distributed “neural” ones are based on the distributional hypothesis.
- The both attempt to capture the similarity between words based on the similarity between the contexts in which they occur.
- Levy and Goldebrg showed in [Levy and Goldberg, 2014] that Skip-gram negative sampling (SGNS) is implicitly factorizing a word-context matrix, whose cells are the pointwise mutual information (PMI) of the respective word and context pairs, shifted by a global constant.
<https://levyomer.files.wordpress.com/2014/09/neural-word-embeddings-as-implicit-matrix-factorization.pdf>
- This ties the neural methods and the traditional “count-based” suggesting that in a deep sense the two algorithmic families are equivalent.

7.6. FastText

- FastText embeddings extend the skipgram model to take into account the internal structure of words while learning word representations [Bojanowski et al., 2016].
- A vector representation is associated with each character n -gram.
- Words are represented as the sum of these representations.
- Taking the word *where* and $n = 3$, it will be represented by the character n -grams: <wh, whe, her, ere, re>, and the special sequence <where>.
- Note that the sequence <her>, corresponding to the word “her” is different from the tri-gram “her” form the word “here”.
- FastText is useful for morphologically rich languages. For example, the words “amazing” and “amazingly” share information in FastText through their shared n -grams, whereas in Word2Vec these two words are completely unrelated.
- Let \mathcal{G}_w be the set of n -grams appearing in w .
- FastText associates a vector \vec{g} with each n -gram in \mathcal{G}_w .

- In FastText the probability that a word-context pair (w, c) came from the input corpus D is calculated as follows:

$$P(D|w, c) = \frac{1}{1 + e^{-s(w, c)}}$$

where,

$$s(w, c) = \sum_{g \in G_w} \vec{g} \cdot \vec{c}$$

- The negative sampling algorithm can be calculated in the same form as in the skip-gram model with this formulation.

7.7. Sentiment-Specific Phrase Embeddings

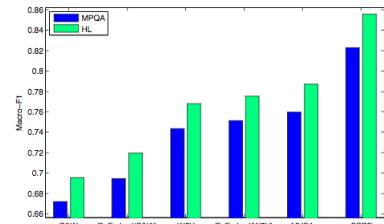
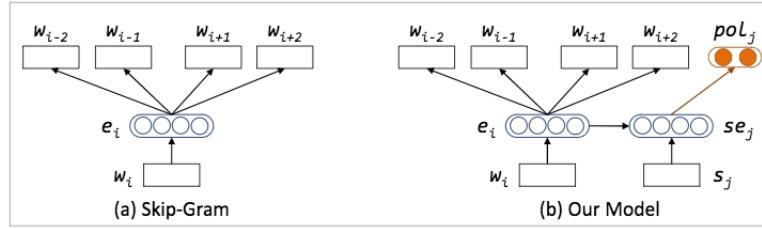
<https://pdfs.semanticscholar.org/107f/b80ff801894b6191d0613af41aba91c134a4.pdf>

- Problem of word embeddings: antonyms can be used in similar contexts e.g., my car is nice vs my car is ugly.
- In [Tang et al., 2014] **sentiment-specific** word embeddings are proposed by combining the skip-gram model with emoticon-annotated tweets :-(
- These embeddings are used for **training** a word-level polarity classifier.
- The model integrates sentiment information into the continuous representation of phrases by developing a tailored neural architecture.
- Input: $\{w_i, s_j, pol_j\}$, where w_i is a phrase (or word), s_j the sentence, and pol_j the sentence's polarity.
- The training objective uses the embedding of w_i to predict its context words (in the same way as the skip-gram model), and uses the sentence representation se_j to predict pol_j .
- Sentences (se_j) are represented by averaging the word vectors of their words.
- The objective of the sentiment part is to maximize the average of log sentiment probability:

$$f_{sentiment} = \frac{1}{S} \sum_{j=1}^S \log p(pol_j | se_j)$$

- The final training objective is to maximize the linear combination of the skip-gram and sentiment objectives:

$$f = \alpha f_{skipgram} + (1 - \alpha) f_{sentiment}$$



(b) Sentiment classification of lexicons with different embedding learning algorithms.

7.8. Gensim

Gensim is an open source Python library for natural language processing that implements many algorithms for training word embeddings.

- <https://radimrehurek.com/gensim/>
- <https://machinelearningmastery.com/develop-word-embeddings-python-gensim/>



Capítulo 8

Etiquetado de Secuencias

8.1. Overview

- The Sequence Labeling (or Tagging) Problem
- Generative models, and the noisy-channel model, for supervised learning
- Hidden Markov Model (HMM) taggers
 - Basic definitions
 - Parameter estimation
 - The Viterbi algorithm

This slides are based on the course material by Michael Collins: <http://www.cs.columbia.edu/~mcollins/cs4705-spring2019/slides/tagging.pdf>

8.2. Sequence Labeling or Tagging Tasks

- Sequence Labeling or Tagging is a task in NLP different from document classification.
- Here the goal is to map a sentence represented as a sequence of tokens x_1, x_2, \dots, x_n into a sequence of tags or labels y_1, y_2, \dots, y_n .
- More specifically, The goal of sequence labeling is to assign tags to words, or more generally, to assign discrete labels to discrete elements in a sequence [Eisenstein, 2018].
- Well known examples of this task are Part-of-Speech (POS) tagging and Named Entity Recognition (NER) to be presented next.

8.3. Part-of-Speech Tagging

INPUT: Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT: Profits/**N** soared/**V** at/**P** Boeing/**N** Co./**N** /, easily/**ADV** toping/**V** forecasts/**N** on/**P** Wall/**N** Street/**N** /, as/**P** their/**POSS** CEO/**N** Alan/**N** Mulally/**N** announced/**V** first/**ADJ** quarter/**N** results/**N** /.

- **N** = Noun
- **V** = Verb
- **P** = Preposition
- **Adv** = Adverb
- **Adj** = Adjective
- ...

Tag	Description	Example
Open Class	ADJ Adjective: noun modifiers describing properties	<i>red, young, awesome</i>
	ADV Adverb: verb modifiers of time, place, manner	<i>very, slowly, home, yesterday</i>
	NOUN words for persons, places, things, etc.	<i>algorithm, cat, mango, beauty</i>
	VERB words for actions and processes	<i>draw, provide, go</i>
	PROPN Proper noun: name of a person, organization, place, etc..	<i>Regina, IBM, Colorado</i>
Closed Class Words	INTJ Interjection: exclamation, greeting, yes/no response, etc.	<i>oh, um, yes, hello</i>
	ADP Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	<i>in, on, by, under</i>
	AUX Auxiliary: helping verb marking tense, aspect, mood, etc.,	<i>can, may, should, are</i>
	CCONJ Coordinating Conjunction: joins two phrases/clauses	<i>and, or, but</i>
	DET Determiner: marks noun phrase properties	<i>a, an, the, this</i>
	NUM Numeral	<i>one, two, first, second</i>
	PART Particle: a function word that must be associated with another word	<i>'s, not, (infinitive) to</i>
	PRON Pronoun: a shorthand for referring to an entity or event	<i>she, who, I, others</i>
	SCONJ Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	<i>that, which</i>
Other	PUNCT Punctuation	<i>: , ()</i>
	SYM Symbols like \$ or emoji	<i>\$, %</i>
	X Other	<i>asdf, qwfg</i>

Source: [Jurafsky and Martin, 2008]

8.4. Named Entity Recognition (NER)

A **named entity** is, roughly speaking, anything that can be referred to with a named entity proper name: a person, a location, an organization.

INPUT: Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT: Profits soared at [Company Boeing Co.], easily topping forecasts on [Location Wall Street], as their CEO [Person Alan Mulally] announced first quarter results.

- Since entities can span multiple words (i.e., a span recognition problem), we can use BIO tagging [Ramshaw and Marcus, 1999] to turn the problem into a sequence labeling problem.
- BIO tagging: use tags that capture both the boundary and the named entity type.

BIO tagging: NER as Sequence Labeling **INPUT:** Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT: Profits/O soared/O at/O Boeing/B-C Co./I-C,/O easily/O top-
ping/O forecasts/O on/O Wall/B-L Street/I-L,/O as/O their/O CEO/O Alan/B-
P Mulally/I-P announced/O first/O quarter/O results/O ./O

- O = Outside (no entity)
- B-C = Begin Company
- I-C = Inside Company
- B-L = Begin Location
- I-L = Inside Location
- B-P = Begin Person
- I-P = Inside Person

Our Goal Training set:

1. Pierre/NNP Vinken/NNP ,/, 61/CD years/NNS old/JJ ,/, will/MD join/VB
the/DT board/NN as/IN a/DT nonexecutive/JJ director/NN Nov./NNP
29/CD ./.
2. Mr./NNP Vinken/NNP is/VBZ chairman/NN of/IN Elsevier/NNP N.V./NNP
, the/DT Dutch/NNP publishing/VBG group/NN ./.
3. Rudolph/NNP Agnew/NNP ,/, 55/CD years/NNS old/JJ and/CC chair-
man/NN of/IN Consolidated/NNP Gold/NNP Fields/NNP PLC/NNP
, was/VBD named/VBN a/DT nonexecutive/JJ director/NN of/IN
this/DT British/JJ industrial/JJ conglomerate/NN ./.
4. ...

Our Goal: From the training set, induce a function/algorithm that maps new sentences to their tag sequences.

Two Types of Constraints Influential/JJ members/NNS of/IN the/DT House/NNP Ways/NNP and/CC Means/NNP Committee/NNP introduced/VBD legislation/NN that/WDT would/MD restrict/VB how/WRB the/DT new/JJ savings-and-loan/NN bailout/NN agency/NN can/MD raise/VB capital/NN ./.

"Local":

- e.g., "can" is more likely to be a modal verb MD rather than a noun NN

"Contextual":

- e.g., a noun is much more likely than a verb to follow a determiner

Sometimes these preferences are in conflict:

- The trash can is in the garage

8.4.1. Sequence Labeling as Supervised Learning

- We have a sequence of inputs $x = (x_1, x_2, \dots, x_n)$ and corresponding labels $y = (y_1, y_2, \dots, y_n)$.
- Task is to learn a function f that maps input sequences to label sequences: $f(x_1, x_2, \dots, x_n) = y_1, y_2, \dots, y_n$.
- We have a training set of labeled sequences: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$.

8.4.2. Generative Approach for Sequence Labeling

- Generative models such as Naive Bayes was used for classification can also be used for sequence labeling tasks in NLP.
- Approach:
 - Training: Learn the joint distribution $p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$ of input sequences.
 - Decoding: Use the learned distribution to predict label sequences for new input sequences.
- Decoding in sequence labeling involves finding the label sequence with the highest joint probability: $\arg \max_{y_1, y_2, \dots, y_n} p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$.

8.5. Hidden Markov Models

- Hidden Markov Models (HMMs) provide a principled way to handle sequence labeling problems using generative modeling and efficient decoding algorithms.

- We have an input sentence $x = x_1, x_2, \dots, x_n$ (x_i is the i -th word in the sentence).
- We have a tag sequence $y = y_1, y_2, \dots, y_n$ (y_i is the i -th tag in the sentence).
- We'll use an HMM to define $p(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$ for any sentence x_1, \dots, x_n and tag sequence y_1, \dots, y_n of the same length. [Kupiec, 1992]
- Then, the most likely tag sequence for x is:

$$\arg \max_{y_1, \dots, y_n} p(x_1, \dots, x_n, y_1, \dots, y_n)$$

8.6. Trigram Hidden Markov Models (Trigram HMMs)

For any sentence x_1, \dots, x_n where $x_i \in V$ for $i = 1, \dots, n$, and any tag sequence y_1, \dots, y_{n+1} where $y_i \in S$ for $i = 1, \dots, n$, and $y_{n+1} = \text{STOP}$, the joint probability of the sentence and tag sequence is:

$$p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^n e(x_i | y_i)$$

where we have assumed that $x_0 = x_{-1} = *$.

8.6.1. Parameters of the Model

- $q(s|u, v)$ for any $s \in S \cup \{\text{STOP}\}$, $u, v \in S \cup \{*\}$
 - The value for $q(s|u, v)$ can be interpreted as the probability of seeing the tag s immediately after the bigram of tags (u, v) .
- $e(x|s)$ for any $s \in S$, $x \in V$
 - The value for $e(x|s)$ can be interpreted as the probability of seeing observation x paired with state s .

An Example If we have $n = 3$, x_1, x_2, x_3 equal to the sentence "the dog laughs", and y_1, y_2, y_3, y_4 equal to the tag sequence "D N V STOP", then:

$$\begin{aligned} p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) &= q(D|*, *) \times q(N|*, D) \\ &\quad \times q(V|D, N) \times q(\text{STOP}|N, V) \\ &\quad \times e(\text{the}|D) \times e(\text{dog}|N) \times e(\text{laughs}|V) \end{aligned}$$

- STOP is a special tag that terminates the sequence.
- We take $y_0 = y_{-1} = *$, where * is a special "padding" symbol.

8.7. Independence Assumptions in Trigram HMMs

- Trigram Hidden Markov Models (HMMs) are derived by making specific independence assumptions in the model.
- Consider two sequences of random variables: X_1, \dots, X_n and Y_1, \dots, Y_n , where n is the length of the sequences.
- Each X_i can take any value in a finite set V of words, and each Y_i can take any value in a finite set K of possible tags (e.g., $K = \{D, N, V, \dots\}$).
- Our goal is to model the joint probability:

$$\begin{aligned}
 P(x_1, x_2, \dots, x_n, y_1, \dots, y_n) \\
 &= p(y_1) \times p(y_2|y_1) \\
 &\quad \times \dots \\
 &\quad \times p(y_n|y_{n-1}, y_{n-2}, \dots, y_1) \\
 &\quad \times p(x_1|y_n, y_{n-1}, \dots, y_1) \\
 &\quad \times p(x_2|x_1, y_n, y_{n-1}, \dots, y_1) \\
 &\quad \times p(x_n|x_{n-1}, \dots, x_1, y_n, y_{n-1}, \dots, y_1)
 \end{aligned}$$

- We define an additional random variable Y_{n+1} that always takes the value "STOP."
- The key idea in HMMs is the factorization of the joint probability:

$$\begin{aligned}
 &P(X_1 = x_1, \dots, X_n = x_n, Y_1 = y_1, \dots, Y_{n+1} = y_{n+1}) \\
 &= \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \times \prod_{i=1}^n P(X_i = x_i | Y_i = y_i)
 \end{aligned}$$

- We first assume that:

$$P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) = q(y_i | y_{i-2}, y_{i-1})$$

- This assumes that the sequence Y_1, \dots, Y_{n+1} is a second-order Markov sequence, where each state depends only on the previous two states.
- And we also assume that:

$$P(X_i = x_i | Y_i = y_i) = e(x_i | y_i)$$

- This assumes that the value of the random variable X_i depends only on the value of Y_i .
- These independence assumptions allow for the derivation of the joint probability equation.

8.8. Why the Name?

$$\begin{aligned}
p(x_1, \dots, x_n, y_1, \dots, y_n) &= q(\text{STOP}|y_{n-1}, y_n) \\
&\times \prod_{j=1}^n q(y_j|y_{j-2}, y_{j-1}) \\
&\times \prod_{j=1}^n e(x_j|y_j)
\end{aligned}$$

- Markov Chain component:

$$q(\text{STOP}|y_{n-1}, y_n) \times \prod_{j=1}^n q(y_j|y_{j-2}, y_{j-1})$$

These transitions are not directly observed for a given sequence of words (x_1, \dots, x_n) , hence the name “hidden”.

- Observed component:

$$\prod_{j=1}^n e(x_j|y_j)$$

The observed component of HMMs models the emission probabilities of observed symbols (x 's) conditioned on the corresponding hidden states (y 's).

8.9. Smoothed Estimation

$$\begin{aligned}
q(Vt|DT, JJ) &= \lambda_1 \times \frac{\text{Count}(Dt, JJ, Vt)}{\text{Count}(Dt, JJ)} \\
&+ \lambda_2 \times \frac{\text{Count}(JJ, Vt)}{\text{Count}(JJ)} \\
&+ \lambda_3 \times \frac{\text{Count}(Vt)}{\text{Count}()}
\end{aligned}$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$, and for all i , $\lambda_i \geq 0$.

$$e(\text{base}|Vt) = \frac{\text{Count}(Vt, \text{base})}{\text{Count}(Vt)}$$

8.10. Dealing with Low-Frequency Words

A common method is as follows:

- Step 1: Split vocabulary into two sets
 - Frequent words = words occurring ≥ 5 times in training
 - Low frequency words = all other words
- Step 2: Map low frequency words into a small, finite set, depending on prefixes, suffixes, etc.

Below is an example of word classes for named entity recognition [Bikel et al., 1999]:

Word class	Example	Intuition
twoDigitNum	90	Two-digit year
fourDigitNum	1990	Four-digit year
containsDigitAndAlpha	A8956 – 67	Product code
containsDigitAndDash	09 – 96	Date
containsDigitAndSlash	11/9/89	Date
containsDigitAndComma	23,000,00	Monetary amount
containsDigitAndPeriod	1,00	Monetary amount, percentage
othernum	456789	Other number
allCaps	BBN	Organization
capPeriod	M.	Person name initial
firstWord	First word of sentence	No useful capitalization information
initCap	Sally	Capitalized word
lowercase	can	Uncapitalized word
other	,	Punctuation marks, all other words

Original Sentence: Profits/O soared/O at/O Boeing/B-C Co./I-C ,/O easily/O topping/O forecasts/O on/O Wall/B-L Street/I-L ,/O as/O their/O CEO/O Alan/B-P Mulally/I-P announced/O first/O quarter/O results/O ./O

Transformed Sentence:

firstword/O soared/O at/O initCap/B-C Co./I-C ,/O easily/O lowercase/O forecasts/O on/O initCap/B-L Street/I-L ,/O as/O their/O CEO/O Alan/B-P initCap/I-P announced/O first/O quarter/O results/O ./O

- O = Outside (no entity)
- B-C = Begin Company
- I-C = Inside Company
- B-L = Begin Location

- I-L = Inside Location
- B-P = Begin Person
- I-P = Inside Person

8.11. Decoding Problem

Decoding Problem: For an input $x_1 \dots x_n$, find

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

where the arg máx is taken over all sequences $y_1 \dots y_{n+1}$ such that $y_i \in S$ for $i = 1 \dots n$, and $y_{n+1} = \text{STOP}$.

We assume that p takes the form:

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^n e(x_i | y_i)$$

Recall that we have assumed in this definition that $y_0 = y_{-1} = *$, and $y_{n+1} = \text{STOP}$.

8.11.1. Naive Brute Force Method

The naive, brute force method for finding the highest scoring tag sequence is to enumerate all possible tag sequences y_1, \dots, y_{n+1} , score them under the function p , and select the sequence with the highest score.

- Example:
 - Input sentence: *the dog barks*
 - Set of possible tags: $K = \{D, N, V\}$
- Enumerate all possible tag sequences:
 - $D D D \text{STOP}$
 - $D D N \text{STOP}$
 - $D D V \text{STOP}$
 - $D N D \text{STOP}$
 - $D N N \text{STOP}$
 - $D N V \text{STOP}$
 - ...
- In this case, there are $3^3 = 27$ possible sequences.

- However, for longer sentences, this method becomes inefficient.
- For an input sentence of length n , there are $|K|^n$ possible tag sequences.
- The exponential growth makes brute-force search infeasible for reasonable length sentences.

8.12. Viterbi Decoding Dynamic Programming

- The algorithm used by HMMs to perform efficient decoding is called Viterbi decoding.
- Viterbi decoding uses dynamic programming.
- Dynamic programming is a technique for solving optimization problems by breaking them down into overlapping subproblems.
- It stores the solutions to these subproblems in a table so that they do not have to be recalculated.
- Dynamic programming can greatly improve the efficiency of algorithms.
- Next, we show how dynamic programming works with two examples: Factorial and Fibonacci

Factorial

- Recursive implementation:

```
def recur_factorial(n):
    # Base case
    if n == 1:
        return n
    else:
        return n * recur_factorial(n-1)
```

- Dynamic programming implementation:

```
def dynamic_factorial(n):
    table = [0 for i in range(0, n+1)]

    # Base case
    table[0] = 1

    for i in range(1, len(table)):
        table[i] = i * table[i-1]

    return table[n]
```

Fibonacci

- Recursive implementation:

```
def recur_fibonacci(n):
    if n == 1 or n == 0:
        return 1
    else:
        return recur_fibonacci(n-1) + recur_fibonacci(n-2)
```

- Dynamic programming implementation:

```
def dynamic_fibonacci(n):
    table = [0 for i in range(0, n+1)]

    # Base case
    table[0] = 1
    table[1] = 1

    for i in range(2, len(table)):
        table[i] = table[i-1] + table[i-2]

    return table[n]
```

Complexity

- In recursive implementations, the complexity can be quite high due to repeated calculations of the same subproblems.
- However, dynamic programming can significantly reduce the complexity by storing the solutions to subproblems in a table or array and reusing them when needed.
- This approach eliminates the redundant calculations and allows for a more efficient computation.
- For the case of Fibonacci the complexity is reduced from exponential to linear.

8.13. The Viterbi Algorithm

The Viterbi algorithm efficiently computes the maximum probability of a tag sequence by using dynamic programming.

Definitions:

- Define n as the length of the sentence.

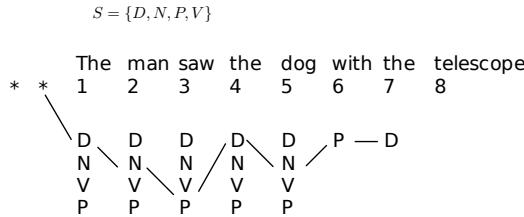
- Define S_k for $k = -1 \dots n$ as the set of possible tags at position k : $S_{-1} = S_0 = \{*\}$, $S_k = S$ for $k \in \{1 \dots n\}$.
- Define a truncated version of the probability encoded by the HMM until position k , $r(y_{-1}, y_0, y_1, \dots, y_k)$ as:

$$r(y_{-1}, y_0, y_1, \dots, y_k) = \prod_{i=1}^k q(y_i | y_{i-2}, y_{i-1})$$

- Define a dynamic programming table $\pi(k, u, v)$ as the maximum probability of a tag sequence ending in tags u, v at position k :

$$\pi(k, u, v) = \max_{y_{-1}, y_0, y_1, \dots, y_{k-1}: y_{k-1}=u, y_k=v} r(y_{-1}, y_0, y_1, \dots, y_k)$$

An Example Recall that $\pi(k, u, v)$ is maximum probability of a tag sequence ending in tags u, v at position k



- There are many possible sequences of tags.
- Each of them has a probability calculated from the parameters q and e .
- $\pi(7, P, D)$ is the maximum probability that one of these tag sequences ends in $P D$ at position 7.
- The path represents the sequence with the maximum probability.

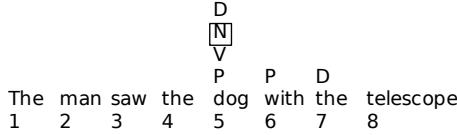
8.13.1. A Recursive Definition

Base case:

$$\pi(0, *, *) = 1$$

Recursive definition: For any $k \in \{1 \dots n\}$, for any $u \in S_{k-1}$ and $v \in S_k$:

$$\pi(k, u, v) = \max_{w \in S_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$



$$\mathcal{S}_5 = \mathcal{S} = \{D, N, V, P\}$$

$$\Pi(7, P, D) = \max_{w \in \mathcal{S}_5} (\Pi(6, w, P) \times q(D|w, P) \times e(\text{the}|D))$$

Justification for the Recursive Definition For any $k \in \{1 \dots n\}$, for any $u \in S_{k-1}$ and $v \in S_k$:

$$\pi(k, u, v) = \max_{w \in \mathcal{S}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

- Let's consider an arbitrary tag sequence that ends with tags P and D at position 7.
- It must contain some tag at position 5.
- We are basically searching for the tag that maximizes the probability at position 5.

8.13.2. The Viterbi Algorithm

Algorithm 1: Viterbi Algorithm

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u, v)$ and $e(x|s)$

Initialization: Set $\pi(0, *, *) = 1$; $S_{-1} = S_0 = \{*\}$, $S_k = S$ for $k \in \{1 \dots n\}$.

```

for  $k = 1$  to  $n$  do
    for  $u \in S_{k-1}, v \in S_k$  do
         $\pi(k, u, v) = \max_{w \in \mathcal{S}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$ 
    end
end
return ( $\max_{u \in S_{n-1}, v \in S_n} (\pi(n, u, v) \times q(STOP|u, v))$ )

```

8.13.3. The Viterbi Algorithm with Backpointers

Algorithm 2: Viterbi Algorithm with Backpointers

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u, v)$ and $e(x|s)$
Initialization: Set $\pi(0, *, *) = 1$; $S_{-1} = S_0 = \{*\}$, $S_k = S$ for $k \in \{1 \dots n\}$.

```

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

 $(y_{n-1}, y_n) = \arg \max_{(u,v)} (\pi(n, u, v) \times q(\text{STOP}|u, v))$ ; // 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

return (the tag sequence  $y_1 \dots y_n$ ) ; // Return the final tag
sequence

```

8.13.4. The Viterbi Algorithm: Running Time

- $O(n|S|^3)$ time to calculate $q(s|u, v) \times e(x_k|s)$ for all k, s, u, v .
- $n|S|^2$ entries in π to be filled in.
- $O(|S|)$ time to fill in one entry.

$\Rightarrow O(n|S|^3)$ time in total.

Pros and Cons

- Hidden Markov Model (HMM) taggers are simple to train (compile counts from training corpus).
- They perform relatively well (over 90 % performance on named entity recognition).
- Main difficulty is modeling $e(\text{word}|tag)$, which can be very complex if "words" are complex.

8.14. MEMMs

- Maximum-entropy Markov models (MEMMs) make use of log-linear multi-class models for sequence labeling tasks [McCallum et al., 2000].
- In the early NLP literature, logistic regression was often called maximum entropy classification [Eisenstein, 2018].
- Hence, MEMMs will look very similar to the multi-class softmax models seen in the lecture about linear models.
- In contrast to HMMs, here we rely on parameterized functions.
- The goal of MEMMs is model the following conditional distribution:

$$P(s_1, s_2 \dots, s_m | x_1, \dots, x_m)$$

- Where each x_j for $j = 1 \dots m$ is the j -th input symbol (for example the j -th word in a sentence), and each s_j for $j = 1 \dots m$ is the j -th tag.¹
- We would expect $P(\text{DET}, \text{NOUN}, \text{VERB} | \text{the}, \text{dog}, \text{barks})$ to be higher than $P(\text{VERB}, \text{VERB}, \text{VERB} | \text{the}, \text{dog}, \text{barks})$ in a model trained from a POS-tagging training dataset.
- We use S to denote the set of possible tags.
- We assume that S is a finite set.
- For example, in part-of-speech tagging of English, S would be the set of all possible parts of speech in English (noun, verb, determiner, preposition, etc.).
- Given a sequence of words x_1, \dots, x_m , there are k^m possible part-of-speech sequences s_1, \dots, s_m , where $k = |S|$ is the number of possible parts of speech.
- We want to estimate a distribution over these k^m possible sequences.
- In a first step, MEMMs use the following decomposition (s_0 has always a special tag *):

$$\begin{aligned} P(s_1, s_2 \dots, s_m | x_1, \dots, x_m) &= \prod_{i=1}^m P(s_i | s_1 \dots, s_{i-1}, x_1, \dots, x_m) \\ &= \prod_{i=1}^m P(s_i | s_{i-1}, x_1, \dots, x_m) \end{aligned} \tag{8.1}$$

¹These slides are based on lecture notes of Michael Collins <http://www.cs.columbia.edu/~mcollins/crf.pdf>. The notation and terminology has been adapted to be consistent with the rest of the course.

- The first equality is exact (it follows by the chain rule of conditional probabilities).
- The second equality follows from an independence assumption, namely that for all i ,

$$P(s_i | s_1, \dots, s_{i-1}, x_1, \dots, x_m) = P(s_i | s_{i-1}, x_1, \dots, x_m)$$

- Hence we are making a first order Markov assumption similar to the Markov assumption made in HMMs².
- The tag in the i -th position depends only on the tag in the $(i-1)$ -th position.
- Having made these independence assumptions, we then model each term using a multiclass log-linear (softmax) model:

$$P(s_i | s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))} \quad (8.2)$$

Here $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)$ is a feature vector where:

- x_1, \dots, x_m is the entire sentence being tagged.
- i is the position to be tagged (can take any value from 1 to m).
- s_{i-1} is the previous tag value (can take any value in S).
- s_i is the new tag value (can take any value in S).

The scope of the feature vector is **restricted** to the whole input sequence x_1, x_m , and only the previous and current tag values. This restriction allows efficient training of both MEMMs and CRFs.

8.15. Example of Features used in Part-of-Speech Tagging

1. $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[1]} = 1$ if $s_i = \text{ADVERB}$ and word x_i ends in “-ly”;
0 otherwise.

If the weight $w_{[1]}$ associated with this feature is large and positive, then this feature is essentially saying that we prefer labelings where words ending in -ly get labeled as ADVERB.

²We actually made a second order Markov assumption in HMMs. MEMMs can also be extended to second order assumptions.

2. $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[2]} = 1$ if $i = 1$, $s_i = \text{VERB}$, and $x_m = ?$; 0 otherwise.
 If the weight $\vec{w}_{[2]}$ associated with this feature is large and positive, then labelings that assign VERB to the first word in a question (e.g., “Is this a sentence beginning with a verb?”) are preferred.

3. $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[3]} = 1$ if $s_{i-1} = \text{ADJECTIVE}$ and $s_i = \text{NOUN}$; 0 otherwise.

Again, a positive weight for this feature means that adjectives tend to be followed by nouns.

4. $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[4]} = 1$ if $s_{i-1} = \text{PREPOSITION}$ and $s_i = \text{PREPOSITION}$.

A negative weight $\vec{w}_{[4]}$ for this function would mean that prepositions don't tend to follow prepositions.

³Source: <https://blog.echen.me/2012/01/03/introduction-to-conditional-random-fields/>

8.16. Feature Templates

It is possible to define more general feature templates covering unigrams, bigrams, n-grams of words as well as tag values of s_{i-1} and s_i .

1. A word unigram and tag unigram feature template: $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[\text{index}(j, z)]} = 1$ if $s_i = \text{TAG}_{[j]}$ and $x_i = \text{WORD}_{[z]}$; 0 otherwise $\forall j, z$.

Notice that j is an index spanning all possible tags in S and z is another index spanning the words in the vocabulary V .

2. A word bigram and tag bigram feature template: $\vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i)_{[\text{index}(j, z, u, v)]} = 1$ if $s_{i-1} = \text{TAG}_{[j]}$ and $s_i = \text{TAG}_{[z]}$ and $x_{i-1} = \text{WORD}_{[u]}$ and $x_i = \text{WORD}_{[v]}$; 0 otherwise $\forall j, z, u, v$.

The function $\text{index}(j, k, \dots)$ will map each different feature to a unique index in the feature vector.

Notice that the resulting vector will be very high-dimensional and sparse.

Example

8.17. MEMMs and Multi-class Softmax

- Notice that the log-linear model from above is very similar to the multi-class softmax model presented in the lecture about linear models.
- A general log-linear model has the following form:

$$P(y|x; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x, y))}{\sum_{y' \in Y} \exp(\vec{w} \cdot \vec{\phi}(x, y'))}$$

$$P(s_i | s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))}$$

Example:

1	2	3	4
The	dog	barks	loudly
DT	NN	VB	ADV

Let's check that $P(s_4 = \text{ADV} | s_3 = \text{VB, the, dog, barks, loudly}) > P(s_4 = \text{VB} | s_3 = \text{VB, the, dog, barks, loudly})$

$$P(s_4 = \text{ADV} | s_3 = \text{VB, the, dog, barks, loudly}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(\text{the, dog, barks, loudly, 4, VB, ADV}))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(\text{the, dog, barks, loudly, 4, VB, } s'))}$$

$$P(s_4 = \text{VB} | s_3 = \text{VB, the, dog, barks, loudly}) = \frac{\exp(\vec{w} \cdot \vec{\phi}(\text{the, dog, barks, loudly, 4, VB, VB}))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(\text{the, dog, barks, loudly, 4, VB, } s'))}$$

- A multi-class softmax model has the following form:

$$\begin{aligned}\hat{y} &= \text{softmax}(\vec{x} \cdot W + \vec{b}) \\ \hat{y}_{[i]} &= \frac{e^{(\vec{x} \cdot W + \vec{b})_{[i]}}}{\sum_j e^{(\vec{x} \cdot W + \vec{b})_{[j]}}}\end{aligned}\tag{8.3}$$

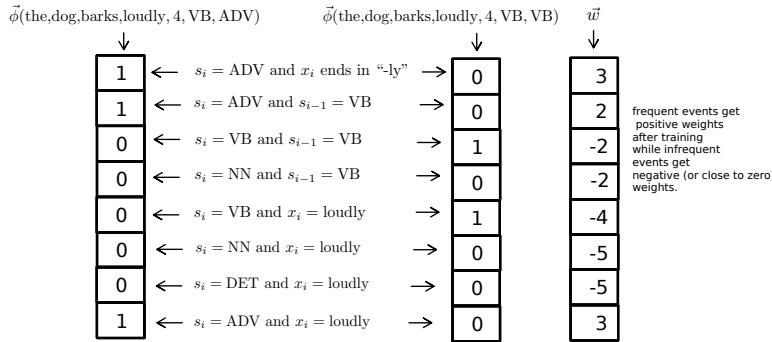
- Difference 1: in the log-linear model we have a fixed parameter vector \vec{w} instead of having multiple vectors (one column of W for each class value).
- Difference 2: the feature vector of the log-linear model $\vec{\phi}(x, y)$ includes information of the label y , whereas the input vector \vec{x} of the softmax model is independent of y .
- Log-linear models allow using features that consider the interaction between x and y (e.g., x ends in "ly" and y is an ADVERB).

8.18. Training MEMMs

- Once we've defined the feature vectors $\vec{\phi}$, we can train the parameters \vec{w} of the model in the usual way linear models are trained.
- We set the negative log-likelihood as the loss function and optimize parameters using gradient descent from the training examples.
- This is equivalent as using the cross-entropy loss.
- "Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model" [Goodfellow et al., 2016].

This is the same as checking if:

$$\vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{ADV}) > \vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{VB})$$



$$\vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{ADV}) = 1 * 3 + 1 * 2 + 1 * 3 = 6$$

$$> \vec{w} \cdot \vec{\phi}(\text{the}, \text{dog}, \text{barks}, \text{loudly}, 4, \text{VB}, \text{VB}) = 1 * -2 + 1 * -4 = -6$$

8.19. Decoding with MEMMs

- The decoding problem is as follows.
 - We are given a new test sequence x_1, \dots, x_m .
 - Our goal is to compute the most likely state sequence for this test sequence,

$$\arg \max_{s_1, \dots, s_m} P(s_1, \dots, s_m | x_1, \dots, x_m). \quad (8.4)$$

- There are k^m possible state sequences, so for any reasonably large sentence length m brute-force search through all the possibilities will not be possible.
 - We can use the Viterbi algorithm in a similar way as used for HMMs.
 - The basic data structure in the algorithm will be a dynamic programming table π with entries $\pi[j, s]$ for $j = 1, \dots, m$, and $s \in S$.
 - $\pi[j, s]$ will store the maximum probability for any state sequence ending in state s at position j .
 - More formally, our algorithm will compute

$$\pi[j, s] = \max_{s_1, \dots, s_{j-1}} \left(P(s|s_{j-1}, x_1, \dots, x_m) \prod_{k=1}^{j-1} P(s_k|s_{k-1}, x_1, \dots, x_m) \right)$$

for all $j = 1, \dots, m$, and for all $s \in S$.

The algorithm is as follows:

- Initialization: for $s \in S$

$$\pi[1, s] = P(s|s_0, x_1, \dots, x_m)$$

where s_0 is a special “initial” state.

- For $j \in \{2, \dots, m\}$, $s \in \{1, \dots, k\}$

$$\pi[j, s] = \max_{s' \in S} [\pi[j - 1, s'] \times P(s|s', x_1, \dots, x_m)]$$

- Finally, having filled in the $\pi[j, s]$ values for all j, s , we can calculate

$$\max_{s_1, \dots, s_m} = \max_s \pi[m, s].$$

- The algorithm runs in $O(mk^2)$ time (i.e., linear in the sequence length m , and quadratic in the number of tags k).
- As in the Viterbi algorithm for HMMs, we can compute the highest-scoring sequence using backpointers in the dynamic programming algorithm.

8.20. Comparison between MEMMs and HMMs

- So what is the motivation for using MEMMs instead of HMMs?
- Note that the Viterbi decoding algorithms for the two models are very similar.
- In MEMMs, the probability associated with each state transition s_{i-1} to s_i is

$$P(s_i|s_{i-1}, x_1, \dots, x_m) = \frac{\exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s_i))}{\sum_{s' \in S} \exp(\vec{w} \cdot \vec{\phi}(x_1, \dots, x_m, i, s_{i-1}, s'))}$$

- In HMMs, the probability associated with each transition is:

$$P(s_i|s_{i-1}, x_1, \dots, x_m) = P(s_1|s_{i-1})P(x_i|s_i)$$

- The key advantage of MEMMs is that the use of feature vectors $\vec{\phi}$ allows much richer representations than those used in HMMs.
- For example, the transition probability can be sensitive to any word in the input sequence x_1, \dots, x_m .

- In addition, it is very easy to introduce features that are sensitive to spelling features (e.g., prefixes or suffixes) of the current word x_i , or of the surrounding words.
- These features are useful in many NLP applications, and are difficult to incorporate within HMMs in a clean way.

8.21. Conditional Random Fields (CRFs)

- We now turn to Conditional Random Fields (CRFs) [Lafferty et al., 2001].
- Notation: for convenience, we'll use $x_{1:m}$ to refer to an input sequence x_1, \dots, x_m , and $s_{1:m}$ to refer to a sequence of tags s_1, \dots, s_m .
- The set of all possible tags is again S .
- The set of all possible tag sequences is S^m .
- In conditional random fields we'll again build a model of

$$P(s_1, \dots, s_m | x_1, \dots, x_m) = P(s_{1:m} | x_{1:m})$$

- A first key idea in CRFs will be to define a feature vector that maps an entire input sequence $x_{1:m}$ paired with an entire tag sequence $s_{1:m}$ to some d -dimensional feature vector:

$$\vec{\Phi}(x_{1:m}, s_{1:m}) \in \mathcal{R}^d$$

- We'll soon give a concrete definition for $\vec{\Phi}$.
- For now just assume that some definition exists.
- We will often refer to $\vec{\Phi}$ as being a “global” feature vector.
- It is global in the sense that it takes the entire state sequence into account.
- In CRFs we build a giant log-linear model:

$$P(s_{1:m} | x_{1:m}; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))}$$

- This is “just” another log-linear model, but it is “giant”.
- The space of possible values for $s_{1:m}$ is huge S^m .
- The normalization constant (denominator in the above expression) involves a sum over all possible tag sequences S^m .
- These issues might seem to cause severe computational problems.

- Under appropriate assumptions we can train and decode efficiently with this type of model.
- We define the global feature vector $\vec{\Phi}(x_{1:m}, s_{1:m})$ as follows:

$$\vec{\Phi}(x_{1:m}, s_{1:m}) = \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$$

where $\vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$ are the same as the feature vectors used in MEMMs.

- Example: $\vec{\Phi}([\text{the}, \text{dog}, \text{barks}], \text{DET}, \text{NOUN}, \text{VERB}) = \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 1, *, \text{DET}) + \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 2, \text{DET}, \text{NOUN}) + \vec{\phi}([\text{the}, \text{dog}, \text{barks}], 3, \text{NOUN}, \text{VERB})$
- Essentially, we are adding up many sparse vectors.

	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
	0	1	0	1
	0	1	0	1
	0	0	0	0
	0	0	1	1
	0	0	1	1

Example ⁴source: http://people.ischool.berkeley.edu/~dbamman/nlpF18/slides/12_neural_sequence_labeling.pdf

- We are assuming that for any dimension of $\vec{\Phi}_{[k]}$, $k = 1, \dots, d$, the k 'th global feature is:

$$\vec{\Phi}(x_{1:m}, s_{1:m})_{[k]} = \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)_{[k]}$$

- Thus $\vec{\Phi}(x_{1:m}, s_{1:m})_{[k]}$ is calculated by summing the “local” feature vector $\vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)_{[k]}$ over the m different tag transitions in s_1, \dots, s_m .
- We would expect each local vector to encode relevant information about the tag transition by turning on some vector dimensions (setting the value to one).

- We now turn to two critical practical issues in CRFs: first, decoding, and second, parameter estimation (training).

8.22. Decoding with CRFs

- The decoding problem in CRFs is as follows.
- For a given input sequence $x_{1:m} = x_1, x_2, \dots, x_m$, we would like to find the most likely underlying state sequence under the model, that is,

$$\begin{aligned}
\arg \max_{s_{1:m} \in S^m} P(s_{1:m} | x_{1:m}; \vec{w}) &= \arg \max_{s_{1:m} \in S^m} \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))} \\
&= \arg \max_{s_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m})) \\
&= \arg \max_{s_{1:m} \in S^m} \vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}) \\
&= \arg \max_{s_{1:m} \in S^m} \vec{w} \cdot \sum_{j=1}^m \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j) \\
&= \arg \max_{s_{1:m} \in S^m} \sum_{j=1}^m \vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)
\end{aligned} \tag{8.5}$$

- We have shown that finding the most likely sequence under the model is equivalent to finding the sequence that maximizes:

$$\arg \max_{s_{1:m} \in S^m} \sum_{j=1}^m \vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$$

- This problem has a clear intuition. Each transition from tag s_{j-1} to tag s_j has an associated score: $\vec{w} \cdot \vec{\phi}(x_{1:m}, j, s_{j-1}, s_j)$
- This score could be positive or negative.
- Intuitively, this score will be relatively high if the state transition is plausible, relatively low if this transition is implausible.
- The decoding problem is to find an entire sequence of states such that the sum of transition scores is maximized.
- We can again solve this problem using a variant of the Viterbi algorithm, in a very similar way to the decoding algorithm for HMMs or MEMMs.

8.23. Parameter Estimation in CRFs (training)

- For parameter estimation, we assume we have a set of n labeled examples, $\{(x_{1:m}^i, s_{1:m}^i)\}_{i=1}^n$. Each $x_{1:m}^i$ is an input sequence x_1^i, \dots, x_m^i each $s_{1:m}^i$ is a tag sequence s_1^i, \dots, s_m^i .
- We again set the negative log-likelihood (or cross-entropy) as the loss function L as optimize parameters using gradient descent.
- The main challenge here is that gradient calculations $\frac{\partial L}{\partial \vec{w}_{[k]}}$ involve summing over S^m (a very large set containing all possible tag sequences).
- This sum can be computed efficiently using the Forward-backward algorithm⁵.
- This is another dynamic programming algorithm that is closely related to the Viterbi algorithm.

8.24. CRFs and MEMMs

- CRFs and MEMMS are discriminative sequence labeling models: they model the conditional probability directly via a parameterized log-linear multi-class function (softmax).
- HMMs, on the other hand, are generative models.
- In MEMM the normalization (denominator of the softmax) is local: it happens at each tag step (the sum runs over all possible tag values S).
- In CRFs the normalization is global: the sum runs over all possible tag sequences S^m .
- Training a MEMM is quite easy: just train a multi-class log-linear model for a given word to the label. This classifier is used at each word step to predict the whole sequence.
- Training CRF is more complex. The objective is to maximize the log probability of the most likely sequence.

8.24.1. CRFs and MEMMs: the label bias problem

- MEMMs end up making up decision at each time step independently.
- This leads to a problem called label bias: in some tag space configurations, MEMMs essentially completely ignore important aspects of the context.

⁵<http://www.cs.columbia.edu/~mcollins/fb.pdf>

- Example: The right POS labeling of sentence “will to fight” (la voluntad de pelear) is “NN TO VB”.⁶
- Here NN stands for “noun”, TO stands for “infinitive to”, and VB stands for “verb base form”.
- Modals (MD) show up much more frequently at the start of the sentence than nouns do (e.g., questions).
- Hence, tag “MD” will receive a higher score than tag “NN” when $x_0 = \text{“will”}$: $P(s_1 = MD | s_0 = *, x_1 = \text{“will”}, \dots) > P(s_1 = NN | s_{i-1} = *, x_1 = \text{“will”})$.
- But we know that MD + TO is very rare: “... can to eat”, “... would to eat”.
- The word “to” is relatively deterministic (almost always has tag TO) so it doesn’t matter what tag precedes it.
- Because of the local normalization of MEMMs, $P(s_i = TO | s_{i-1}, x_1, \dots, x_i = \text{“to”}, \dots, x_n)$ will always be 1 when $x_i = \text{“to”}$ regardless of the value of s_{i-1} (MD or NN).
- That means our prediction for “to” can’t help us disambiguate “will”.
- We lose the information that MD + TO sequences rarely happen.
- As a consequence: a MEMMS would likely label the first word to “MD”.
- CRF overcomes this issue by doing a global normalization: it considers the score of the whole sequence before normalizing to make it a probability distribution.

Label Bias In some state space configurations, MEMMs essentially completely ignore the inputs. “label bias problem,” where states with low-entropy transition distributions “effectively ignore” their observations. These are names for situations when one source of information is ignored because it is explained away by another source

8.25. Links

- 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
- <https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/>

⁶Here we are using the PENN Treebank tagset: <https://www.eecis.udel.edu/~vijay/cis889/ie/pos-set.pdf>

- <https://www.quora.com/What-are-the-pros-and-cons-of-these-three-sequence-models>
- <https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf>
- http://www.davidsbatista.net/blog/2017/11/13/Conditional_Random_Fields

Capítulo 9

Redes Neuronales Convolucionales

- Convolutional neural networks (CNNs) became very popular in the computer vision community due to its success for detecting objects (“cat”, “bicycles”) regardless of its position in the image.
- They identify indicative local predictors in a structure (e.g., images, sentences).
- These predictors are combined to produce a fixed size vector representation for the structure.
- When used in NLP, the network captures the n-grams that are most informative for the target predictive task.
- For sentiment classification, these local aspects correspond to n-grams conveying sentiment (e.g., not bad, very good).
- The fundamental idea of CNNs [LeCun et al., 1998] is to consider feature extraction and classification as one jointly trained task.

9.1. Basic Convolution + Pooling

- Sentences are usually modeled as sequences of word embeddings.
- These embeddings can be obtained either from pre-trained word embeddings or from an embedding layer.
- The CNN applies nonlinear (learned) functions or “filters” mapping windows of k words into scalar values.
- Several filters can be applied, resulting in an l -dimensional vector (one dimension per filter).

- The filters capture relevant properties of the words in the window.
- These filters correspond to the “convolution layer” of the network.
- The “pooling” layer is used to combine the vectors resulting from the different windows into a single l -dimensional vector.
- This is done by taking the max or the average value observed in each of the dimensions over the different windows.
- The goal is to capture the most important “features” in the sentence, regardless of the position.
- The resulting l -dimensional vector is then fed further into a network that is used for prediction (e.g., softmax).
- The gradients are propagated back from the network’s loss tuning the parameters of the filter.
- The filters learn to highlight the aspects of the data (n-grams) that are important for the target task.

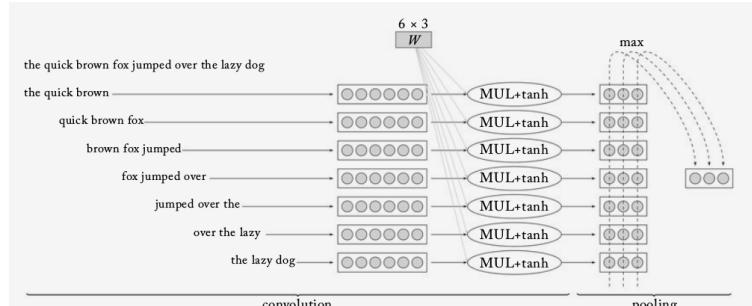


Figure 13.2: 1D convolution+pooling over the sentence “the quick brown fox jumped over the lazy dog.” This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transferred through a 6×3 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.

¹Source: [Goldberg, 2017]

9.2. 1D Convolutions over Text

- We focus on the one-dimensional convolution operation².

²1D here refers to a convolution operating over 1-dimensional inputs such as sequences, as opposed to 2D convolutions which are applied to images.

- Consider a sequence of words $w_{1:n} = w_1, \dots, w_n$ each with their corresponding d_{emb} dimensional word embedding $E_{[w_i]} = \vec{w}_i$.
- A 1D convolution of width k works by moving a sliding-window of size k over the sentence, and applying the same filter to each window in the sequence.
- A filter is a dot-product with a weight vector \vec{u} , which is often followed by a nonlinear activation function.
- Define the operator $\oplus(w_{i:i+k-1})$ to be the concatenation of the vectors $\vec{w}_i, \dots, \vec{w}_{i+k-1}$.
- The concatenated vector of the i -th window is $\vec{x}_i = \oplus(w_{i:i+k-1}) = [\vec{w}_i; \vec{w}_{i+1}; \dots; \vec{w}_{i+k-1}]$, $x_i \in \mathcal{R}^{k \cdot d_{emb}}$.
- We then apply the filter to each window vector resulting in scalar values $p_i = g(\vec{x}_i \cdot \vec{u})$. ($p_i \in \mathcal{R}$)
- It is customary to use l different filters, $\vec{u}_1, \dots, \vec{u}_l$, which can be arranged into a matrix U , and a bias vector \vec{b} is often added: $\vec{p}_i = g(\vec{x}_i \cdot U + \vec{b})$.
- Each vector \vec{p}_i is a collection of l values that represent (or summarise) the i -th window ($\vec{p}_i \in \mathcal{R}^l$).
- Ideally, each dimension captures a different kind of indicative information.
- The main idea behind the convolution layer is to apply the same parameterized function over all k -grams in the sequence.
- This creates a sequence of m vectors, each representing a particular k -gram in the sequence.
- The representation is sensitive to the identity and order of the words within a k -gram.
- However, the same representation will be extracted for a k -gram regardless of its position within the sequence.

9.3. Narrow vs. Wide Convolutions

- How many vectors \vec{p}_i do we have?
- For a sentence of length n with a window of size k , there are $n - k + 1$ positions in which to start the sequence.
- We get $n - k + 1$ vectors $\vec{p}_{1:n-k+1}$.
- This approach is called **narrow convolution**.

- An alternative is to pad the sentence with $k - 1$ padding-words to each side, resulting in $n + k + 1$ vectors $\vec{p}_{1:n+k+1}$.
- This is called a **wide convolution**.

9.4. Vector Pooling

- Applying the convolution over the text results in m vectors $\vec{p}_{1:m}$, each $\vec{p}_i \in \mathcal{R}^l$.
- These vectors are then combined (pooled) into a single vector $c \in \mathcal{R}^l$ representing the entire sequence.
- Max pooling: this operator takes the maximum value across each dimension (most common pooling operation).

$$\vec{c}_{[j]} = \max_{1 < i \leq m} \vec{p}_{i[j]} \quad \forall j \in [1, l]$$

where $\vec{p}_{i[j]}$ denotes the j -th component of \vec{p}_i .

- Average Pooling (second most common): takes the average value of each index:

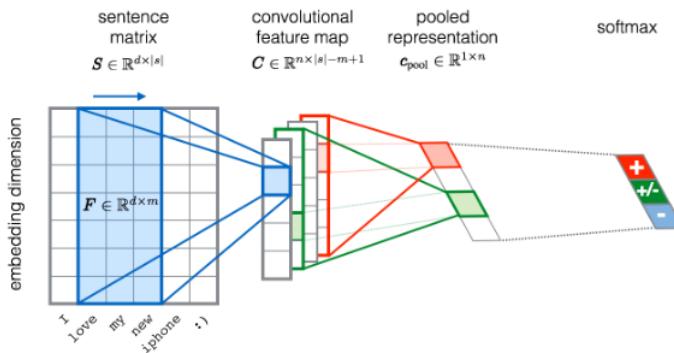
$$\vec{c} = \frac{1}{m} \sum_{i=1}^m \vec{p}_i$$

- Ideally, the vector \vec{c} will capture the essence of the important information in the sequence.
- The nature of the important information that needs to be encoded in the vector \vec{c} is task dependent.
- If we are performing sentiment classification, the essence are informative ngrams that indicate sentiment.
- If we are performing topic-classification, the essence are informative n -grams that indicate a particular topic.
- During training, the vector \vec{c} is fed into downstream network layers (i.e., an MLP), culminating in an output layer which is used for prediction.
- The training procedure of the network calculates the loss with respect to the prediction task, and the error gradients are propagated all the way back through the pooling and convolution layers, as well as the embedding layers.

- The training process tunes the convolution matrix U , the bias vector \vec{b} , the downstream network, and potentially also the embeddings matrix E^3 such that the vector \vec{c} resulting from the convolution and pooling process indeed encodes information relevant to the task at hand.

9.5. Twitter Sentiment Classification with CNN

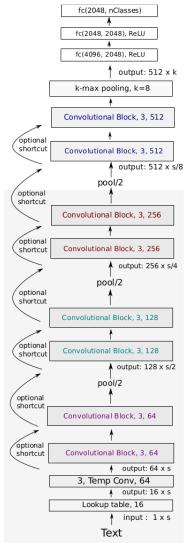
- A convolutional neural network architecture for Twitter sentiment classification is developed in [Severyn and Moschitti, 2015].
- Each tweet is represented as a matrix whose columns correspond to the words in the tweet, preserving the order in which they occur.
- The words are represented by dense vectors or embeddings trained from a large corpus of unlabeled tweets using word2vec.
- The network is formed by the following layers: an input layer with the given tweet matrix, a single convolutional layer, a rectified linear activation function, a max pooling layer, and a softmax classification layer.
- The weights of the neural network are pre-trained using emoticon-annotated data, and then trained with the hand-annotated tweets from the SemEval competition.
- Experimental results show that the pre-training phase allows for a proper initialization of the network's weights, and hence, has a positive impact on classification accuracy.



³While some people leave the embedding layer fixed during training, others allow the parameters to change.

9.6. Very Deep Convolutional Networks for Text Classification

- CNNs architectures for NLP are rather shallow in comparison to the deep convolutional networks which have pushed the state-of-the-art in computer vision.
- A text processing neural architecture (VDCNN) that operates directly at the character level and uses only small convolutions and pooling operations is proposed in [Conneau et al., 2017].
- Character-level embeddings are used instead of word-embeddings.
- Characters are the lowest atomic representation of text.
- The performance of this model increases with depth: using up to 29 convolutional layers, authors report improvements over the state-of-the-art on several public text classification tasks.
- Most notorious improvements are achieved on large datasets.
- One of the first words showing the the benefits of depth neural architectures for NLP.



Capítulo 10

Redes Neuronales Recurrentes

- While representations derived from convolutional networks offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.
- Recurrent neural networks (RNNs) allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs [Goldberg, 2016].
- RNNs, particularly ones with gated architectures such as the LSTM and the GRU, are very powerful at capturing statistical regularities in sequential inputs.

10.1. The RNN Abstraction

- We use $\vec{x}_{i:j}$ to denote the sequence of vectors $\vec{x}_i, \dots, \vec{x}_j$.
- On a high-level, the RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in} -dimensional vectors $\vec{x}_{1:n} = \vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$ ($\vec{x}_i \in \mathcal{R}^{d_{in}}$) and returns as output a single d_{out} dimensional vector $\vec{y}_n \in \mathcal{R}^{d_{out}}$:

$$\begin{aligned} \vec{y}_n &= RNN(\vec{x}_{1:n}) \\ \vec{x}_i &\in \mathcal{R}^{d_{in}} \quad \vec{y}_n \in \mathcal{R}^{d_{out}} \end{aligned} \tag{10.1}$$

- This implicitly defines an output vector \vec{y}_i for each prefix $\vec{x}_{1:i}$ of the sequence $\vec{x}_{i:n}$.

- We denote by RNN^* the function returning this sequence:

$$\begin{aligned}\vec{y}_{1:n} &= RNN^*(\vec{x}_{1:n}) \\ \vec{y}_i &= RNN(\vec{x}_{1:i}) \\ \vec{x}_i &\in \mathcal{R}^{d_{in}} \quad \vec{y}_n \in \mathcal{R}^{d_{out}}\end{aligned}\tag{10.2}$$

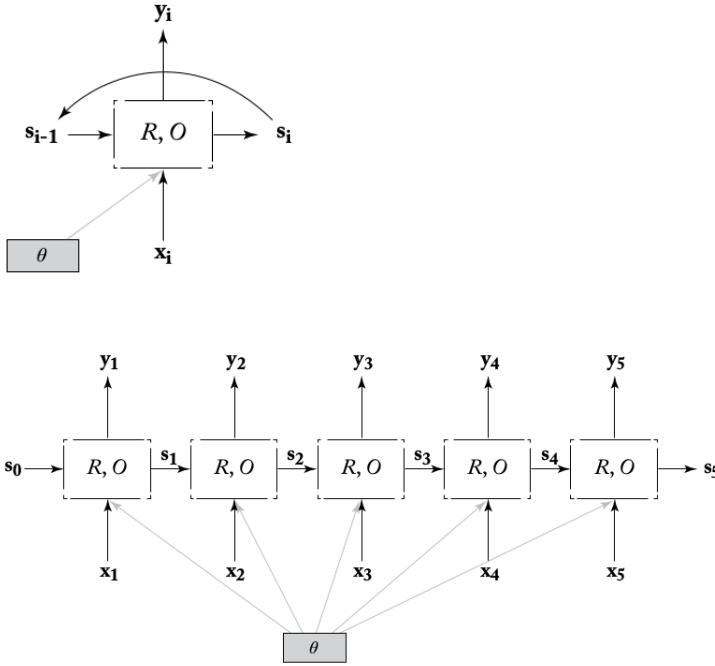
- The output vector \vec{y}_n is then used for further prediction.
- For example, a model for predicting the conditional probability of an event e given the sequence $\vec{x}_{1:n}$ can be defined as the j -th element in the output vector resulting from the softmax operation over a linear transformation of the RNN encoding:

$$p(e = j | \vec{x}_{1:n}) = \text{softmax}(RNN(\vec{x}_{1:n}) \cdot W + \vec{b})_{[j]}$$

- The RNN function provides a framework for conditioning on the entire history without resorting to the Markov assumption which is traditionally used for modeling sequences.
- The RNN is defined recursively, by means of a function R taking as input a state vector \vec{s}_{i-1} and an input vector \vec{x}_i and returning a new state vector \vec{s}_i .
- The state vector \vec{s}_i is then mapped to an output vector \vec{y}_i using a simple deterministic function $O(\cdot)$.
- The base of the recursion is an initial state vector, \vec{s}_0 , which is also an input to the RNN.
- For brevity, we often omit the initial vector s_0 , or assume it is the zero vector.
- When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs \vec{x}_i as well as the dimensions of the outputs \vec{y}_i .

$$\begin{aligned}RNN^*(\vec{x}_{1:n}; \vec{s}_0) &= \vec{y}_{1:n} \\ \vec{y}_i &= O(\vec{s}_i) \\ \vec{s}_i &= R(\vec{s}_{i-1}, \vec{x}_i) \\ \vec{x}_i &\in \mathcal{R}^{d_{in}}, \quad \vec{y}_i \in \mathcal{R}^{d_{out}}, \quad \vec{s}_i \in \mathcal{R}^{f(d_{out})}\end{aligned}\tag{10.3}$$

- The functions R and O are the same across the sequence positions.
- The RNN keeps track of the states of computation through the state vector \vec{s}_i that is kept and being passed across invocations of R .
- This presentation follows the recursive definition, and is correct for arbitrarily long sequences.



- For a finite sized input sequence (and all input sequences we deal with are finite) one can unroll the recursion.
- The parameters θ highlight the fact that the same parameters are shared across all time steps.
- Different instantiations of R and O will result in different network structures.
- We note that the value of \vec{s}_i (and hence \vec{y}_i) is based on the entire input $\vec{x}_1, \dots, \vec{x}_i$.
- For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned}
s_4 &= R(s_3, x_4) \\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}$$

- Thus, \vec{s}_n and \vec{y}_n can be thought of as encoding the entire input sequence.
- The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are trying to solve.

10.2. Elman Network or Simple-RNN

- After describing the RNN abstraction, we are now in place to discuss the simplest instantiations of it.
- Recall that we are interested in a recursive function $\vec{s}_i = R(\vec{x}_i, \vec{s}_{i-1})$ such that \vec{s}_i encodes the sequence $\vec{x}_{1:n}$.
- The simplest RNN formulation is known as an Elman Network or Simple-RNN (S-RNN).

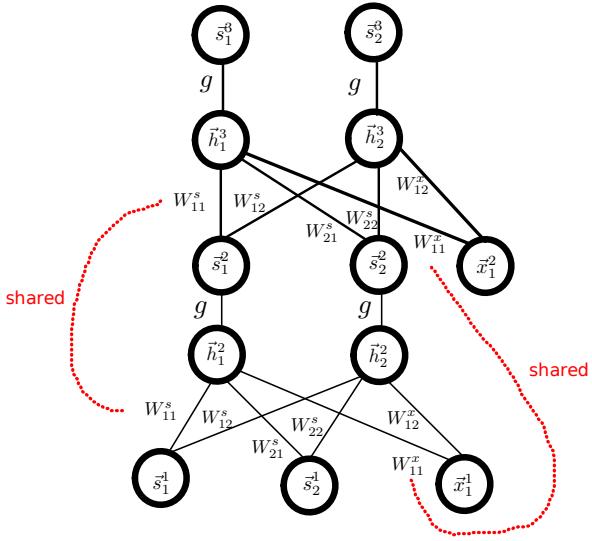
$$\begin{aligned}\vec{s}_i &= R_{SRNN}(\vec{x}_i, \vec{s}_{i-1}) = g(\vec{s}_{i-1}W^s + \vec{x}_iW^x + \vec{b}) \\ \vec{y}_i &= O_{SRNN}(\vec{s}_i) = \vec{s}_i\end{aligned}\tag{10.4}$$

$\vec{s}_i, \vec{y}_i \in \mathcal{R}^{d_s}, \quad \vec{x}_i \in \mathcal{R}^{d_x}, \quad W^x \in \mathcal{R}^{d_x \times d_s}, \quad W^s \in \mathcal{R}^{d_s \times d_s}, \quad \vec{b} \in \mathcal{R}^{d_s}$

- The state \vec{s}_i and the input \vec{x}_i are each linearly transformed.
- The results are added (together with a bias term) and then passed through a nonlinear activation function g (commonly tanh or ReLU).
- The Simple RNN provides strong results for sequence tagging as well as language modeling.

10.3. RNN Training

- An unrolled RNN is just a very deep neural network.
- The same parameters are shared across many parts of the computation.
- Additional input is added at various layers.
- To train an RNN network, need to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph.
- Then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss.
- This procedure is referred to in the RNN literature as backpropagation through time (BPTT).



- The RNN does not do much on its own, but serves as a trainable component in a larger network.
- The final prediction and loss computation are performed by that larger network, and the error is back-propagated through the RNN.
- This way, the RNN learns to encode properties of the input sequences that are useful for the further prediction task.
- The supervision signal is not applied to the RNN directly, but through the larger network.

10.4. RNN usage-patterns: Acceptor

- A common RNN usage-pattern is the acceptor.
- This pattern is used for text classification.
- The supervision signal is only based on the final output vector \vec{y}_n .
- The RNN's output vector \vec{y}_n is fed into a fully connected layer or an MLP, which produce a prediction.
- The error gradients are then backpropagated through the rest of the sequence.
- The loss can take any familiar form: cross entropy, hinge, etc.
- Example 1: an RNN that reads the characters of a word one by one and then use the final state to predict the part-of-speech of that word.

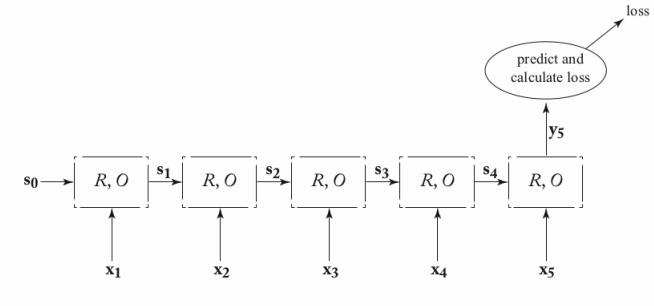


Figura 10.1: Acceptor RNN training graph.

- Example 2: an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment.

10.5. RNN usage-patterns: Transducer

- Another option is to treat the RNN as a transducer, producing an output \hat{t}_i for each input it reads in.
- This pattern is very handy for sequence labeling tasks (e.g., POS tagging, NER).
- We compute a local loss signal (e.g., cross-entropy) $L_{\text{local}}(\hat{t}_i, t_i)$ for each of the outputs \hat{t}_i based on a true label t_i .
- The loss for unrolled sequence will then be: $L(\hat{t}_{i:n}, t_{i:n}) = \sum_i L_{\text{local}}(\hat{t}_i, t_i)$, or using another combination rather than a sum such as an average or a weighted average
- Example 1: a sequence tagger (e.g., NER, POS), in which we take $\vec{x}_{i:n}$ to be feature representations for the n words of a sentence, and t_i as an input for predicting the tag assignment of word i based on words $1 : i$.
- Example 2: language modeling, in which the sequence of words $x_{1:n}$ is used to predict a distribution over the $(i + 1)^{\text{th}}$ word.
- RNN-based language models are shown to provide vastly better perplexities than traditional language models.
- Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history.

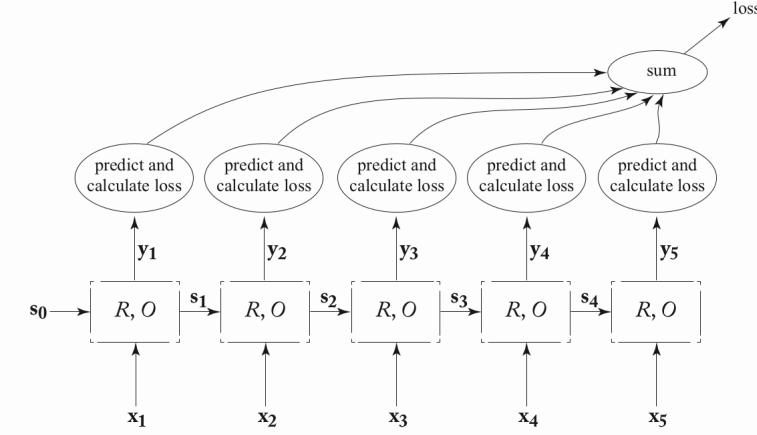


Figura 10.2: Transducer RNN training graph.

10.6. Bidirectional RNNS (BiRNN)

- A useful elaboration of an RNN is a bidirectional-RNN (also commonly referred to as biRNN)
- Consider the task of sequence tagging over a sentence.
- An RNN allows us to compute a function of the i -th word x_i based on the past words $x_{1:i}$ up to and including it.
- However, the following words $x_{i+1:n}$ may also be useful for prediction,
- The biRNN allows us to look arbitrarily far at both the past and the future within the sequence.
- Consider an input sequence $\vec{x}_{1:n}$.
- The biRNN works by maintaining two separate states, s_i^f and s_i^b for each input position i .
- The forward state s_i^f is based on $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_i$, while the backward state s_i^b is based on $\vec{x}_n, \vec{x}_{n-1}, \dots, \vec{x}_i$.
- The forward and backward states are generated by two different RNNs.
- The first RNN(R^f, O^f) is fed the input sequence $\vec{x}_{1:n}$ as is, while the second RNN(R^b, O^b) is fed the input sequence in reverse.
- The state representation \vec{s}_i is then composed of both the forward and backward states.

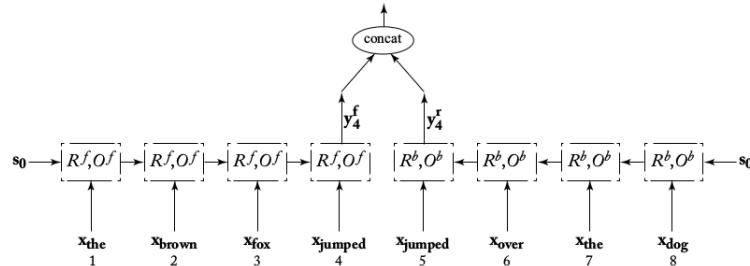
- The output at position i is based on the concatenation of the two output vectors:

$$\vec{y}_i = [\vec{y}_i^f; \vec{y}_i^b] = [O^f(s_i^f); O^b(s_i^b)]$$

- The output takes into account both the past and the future.
- The biRNN encoding of the i th word in a sequence is the concatenation of two RNNs, one reading the sequence from the beginning, and the other reading it from the end.
- We define $biRNN(\vec{x}_{1:n}, i)$ to be the output vector corresponding to the i th sequence position:

$$biRNN(\vec{x}_{1:n}, i) = \vec{y}_i = [RNN^f(\vec{x}_{1:i}); RNN^b(\vec{x}_{n:i})]$$

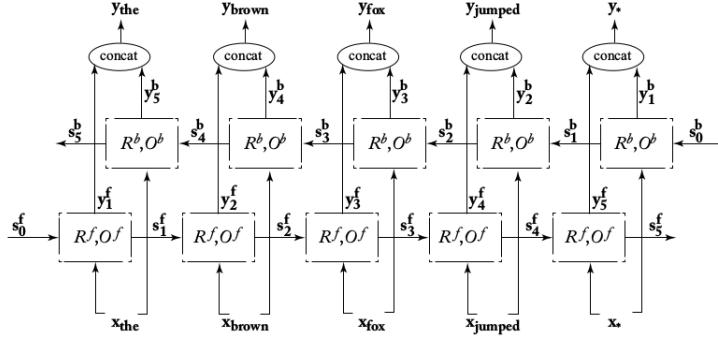
- The vector \vec{y}_i can then be used directly for prediction, or fed as part of the input to a more complex network.
- While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs.
- Feeding the vector \vec{y}_i through an MLP prior to prediction will further mix the forward and backward signals.



- Note how the vector \vec{y}_4 , corresponding to the word **jumped**, encodes an infinite window around (and including) the focus vector \vec{x}_{jumped} .
- Similarly to the RNN case, we also define $biRNN^*(\vec{x}_{1:n})$ as the sequence of vectors $\vec{y}_{1:n}$:

$$biRNN^*(\vec{x}_{1:n}) = \vec{y}_{1:n} = biRNN(\vec{x}_{1:n}, 1), \dots, biRNN(\vec{x}_{1:n}, n)$$

- The n output vectors $\vec{y}_{1:n}$ can be efficiently computed in linear time by first running the forward and backward RNNs, and then concatenating the relevant outputs.



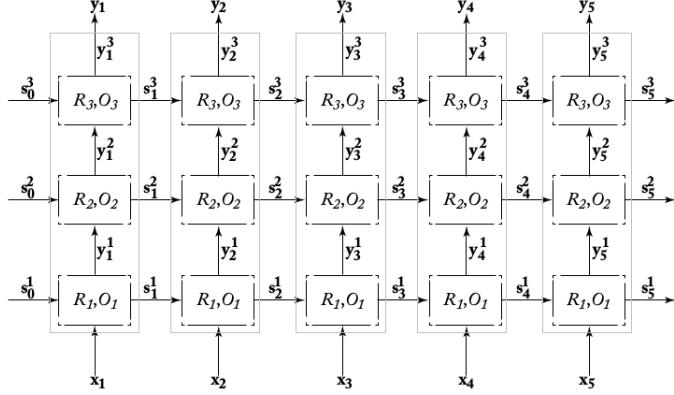
- The biRNN is very effective for tagging tasks, in which each input vector corresponds to one output vector.
- It is also useful as a general-purpose trainable feature-extracting component, that can be used whenever a window around a given word is required.

10.7. Multi-layer (stacked) RNNS

- RNNs can be stacked in layers, forming a grid.
- Consider k RNNs, RNN_1, \dots, RNN_k , where the j th RNN has states $\vec{s}_{1:n}^j$ and outputs $\vec{y}_{1:n}^j$.
- The input for the first RNN are $\vec{x}_{1:n}$.
- The input of the j th RNN ($j \geq 2$) are the outputs of the RNN below it, $\vec{y}_{1:n}^{j-1}$.
- The output of the entire formation is the output of the last RNN, $\vec{y}_{1:n}^k$.
- Such layered architectures are often called deep RNNs.
- It is not theoretically clear what is the additional power gained by the deeper architecture.
- It was observed empirically that deep RNNs work better than shallower ones on some tasks (e.g., machine translation).

10.8. Gated Architectures

- So far we have seen only one instantiation of the RNN: the simple RNN (S-RNN) or Elman network.



- This model is hard to train effectively because of the **vanishing gradients** problem.
- Error signals (gradients) in later steps in the sequence diminish quickly in the backpropagation process.
- Thus, they do not reach earlier input signals, making it hard for the S-RNN to capture long-range dependencies.
- Gating-based architectures, such as the LSTM [Hochreiter and Schmidhuber, 1997] and the GRU [Cho et al., 2014] are designed to solve this deficiency.
- Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers.
- For the Simple-RNN, the gradients then include repeated multiplication of the matrix W , making it very likely for the values to vanish or explode.
- The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.
- Consider the RNN as a general purpose computing device, where the state \vec{s}_i represents a finite memory.
- Each application of the function R reads in an input \vec{x}_{i+1} , reads in the current memory \vec{s}_i , operates on them in some way, and writes the result into memory.
- This results in a new memory state \vec{s}_{i+1} .
- An apparent problem with the S-RNN architecture is that the memory access is not controlled.
- At each step of the computation, the entire memory state is read, and the entire memory state is written.

- How does one provide more controlled memory access?
- Consider a binary vector $\vec{g} \in [0, 1]^n$.
- Such a vector can act as a **gate** for controlling access to n -dimensional vectors, using the hadamard-product operation $\vec{x} \odot \vec{g}$.
- The hadamard operation is the same as the element-wise multiplication of two vectors:

$$\vec{x} = \vec{u} \odot \vec{v} \Leftrightarrow \vec{x}_{[i]} = \vec{u}_{[i]} \cdot \vec{v}_{[i]} \quad \forall i \in [1, n]$$

- Consider a memory $\vec{s} \in \mathcal{R}^d$, an input $\vec{x} \in \mathcal{R}^d$ and a gate $\vec{g} \in [0, 1]^d$.
- The following computation:

$$\vec{s}' \leftarrow \vec{g} \odot \vec{x} + (\vec{1} - \vec{g}) \odot (\vec{s})$$

- Reads the entries in \vec{x} that correspond to the 1 values in \vec{g} , and writes them to the new memory \vec{s}' .
- Locations that weren't read to are copied from the memory \vec{s} to the new memory \vec{s}' through the use of the gate $(\vec{1} - \vec{g})$.

$$\begin{array}{c|c|c|c|c} \begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix} & \leftarrow & \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix} & + & \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \odot \begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix} \\ s' & g & x & (1-g) & s \end{array}$$

- This gating mechanism can serve as a building block in our RNN.
- Gate vectors can be used to control access to the memory state \vec{s}_i .
- We are still missing two important (and related) components:
 1. The gates should not be static, but be controlled by the current memory state and the input.
 2. Their behavior should be learned.
- This introduced an obstacle, as learning in our framework entails being differentiable (because of the backpropagation algorithm).
- The binary 0-1 values used in the gates are not differentiable.
- A solution to this problem is to approximate the hard gating mechanism with a soft—but differentiable—gating mechanism.

- To achieve these differentiable gates , we replace the requirement that $\vec{g} \in [0, 1]^n$ and allow arbitrary real numbers, $\vec{g}' \in \mathcal{R}^n$.
- These real numbers are then pass through a sigmoid function $\sigma(\vec{g}')$.
- This bounds the value in the range $(0, 1)$, with most values near the borders.
- When using the gate $\sigma(\vec{g}') \odot \vec{x}$, indices in \vec{x} corresponding to near-one values in $\sigma(\vec{g}')$ are allowed to pass.
- While those corresponding to near-zero values are blocked.
- The gate values can then be conditioned on the input and the current memory.
- And can be trained using a gradient-based method to perform a desired behavior.
- This controllable gating mechanism is the basis of the LSTM and the GRU architectures.
- At each time step, differentiable gating mechanisms decide which parts of the inputs will be written to memory and which parts of memory will be overwritten (forgotten).

10.9. LSTM

- The Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] was designed to solve the vanishing gradients problem.
- It was the first architecture to introduce the gating mechanism.
- The LSTM architecture explicitly splits the state vector \vec{s}_i into two halves: 1) memory cells and 2) working memory.
- The memory cells are designed to preserve the memory, and also the error gradients, across time, and are controlled through differentiable gating components¹.
- At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten.
- Mathematically, the LSTM architecture is defined as:
- The state at time j is composed of two vectors, \vec{c}_j and \vec{h}_j , where \vec{c}_j is the memory component and \vec{h}_j is the hidden state component.

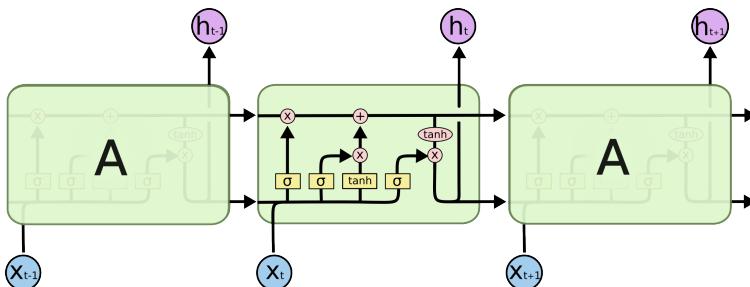
¹Smooth mathematical functions that simulate logical gates.

$$\begin{aligned}
s_j &= R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j; h_j] \\
c_j &= f \odot c_{j-1} + i \odot z \\
h_j &= o \odot \tanh(c_j) \\
i &= \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \\
f &= \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \\
o &= \sigma(x_j W^{xo} + h_{j-1} W^{ho}) \\
z &= \tanh(x_j W^{xz} + h_{j-1} W^{hz})
\end{aligned}$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \quad x_i \in \mathbb{R}^{d_x}, \quad c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \quad W^{xo} \in \mathbb{R}^{d_x \times d_h}, \quad W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

- There are three gates, \vec{i} , \vec{f} , and \vec{o} , controlling for input, forget, and output.
- The gate values are computed based on linear combinations of the current input \vec{x}_j and the previous state \vec{h}_{j-1} , passed through a sigmoid activation function.
- An update candidate \vec{z} is computed as a linear combination of \vec{x}_j and \vec{h}_{j-1} , passed through a tanh activation function (to push the values to be between -1 and 1).
- The memory \vec{c}_j is then updated: the forget gate controls how much of the previous memory to keep ($\vec{f} \odot \vec{c}_{j-1}$), and the input gate controls how much of the proposed update to keep ($\vec{i} \odot \vec{z}$).
- Finally, the value of \vec{h}_j (which is also the output \vec{y}_j) is determined based on the content of the memory \vec{c}_j , passed through a tanh nonlinearity and controlled by the output gate.
- The gating mechanisms allow for gradients related to the memory part \vec{c}_j to stay high across very long time ranges.² source: <http://colah.github.io/posts/2015-08-09-LSTM/>.



github.io/posts/2015-08-Understanding-LSTMs/

- Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers.
- For the Simple-RNN, the gradients then include repeated multiplication of the matrix W .
- This makes the gradient values to vanish or explode.
- The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.
- LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results.
- The main competitor of the LSTM RNN is the GRU, to be discussed next.

10.10. GRU

- The LSTM architecture is very effective, but also quite complicated.
- The complexity of the system makes it hard to analyze, and also computationally expensive to work with.
- The gated recurrent unit (GRU) was introduced in [Cho et al., 2014] as an alternative to the LSTM.
- It was subsequently shown in [Chung et al., 2014] to perform comparably to the LSTM on several (non textual) datasets.
- Like the LSTM, the GRU is also based on a gating mechanism, but with substantially fewer gates and without a separate memory component.

$$\begin{aligned}
 s_j &= R_{\text{GRU}}(s_{j-1}, x_j) = (1 - z) \odot s_{j-1} + z \odot \tilde{s}_j \\
 z &= \sigma(x_j W^{xz} + s_{j-1} W^{sz}) \\
 r &= \sigma(x_j W^{xr} + s_{j-1} W^{sr}) \\
 \tilde{s}_j &= \tanh(x_j W^{xs} + (r \odot s_{j-1}) W^{sg})
 \end{aligned}$$

$$y_j = O_{\text{GRU}}(s_j) = s_j$$

$$s_j, \tilde{s}_j \in \mathbb{R}^{d_s}, \quad x_i \in \mathbb{R}^{d_x}, \quad z, r \in \mathbb{R}^{d_s}, \quad W^{xo} \in \mathbb{R}^{d_x \times d_s}, \quad W^{so} \in \mathbb{R}^{d_s \times d_s}.$$

- One gate \vec{r} is used to control access to the previous state \vec{s}_{j-1} and compute a proposed update $\vec{\tilde{s}}_j$.
- The updated state \vec{s}_j (which also serves as the output \vec{y}_j) is then determined based on an interpolation of the previous state \vec{s}_{j-1} and the proposal $\vec{\tilde{s}}_j$.
- The proportions of the interpolation are controlled using the gate \vec{z} .
- The GRU was shown to be effective in language modeling and machine translation.
- However, the jury is still out between the GRU, the LSTM and possible alternative RNN architectures, and the subject is actively researched.
- For an empirical exploration of the GRU and the LSTM architectures, see [Jozefowicz et al., 2015].

10.11. Sentiment Classification with RNNs

- The simplest use of RNNs is as acceptors: read in an input sequence, and produce a binary or multi-class answer at the end.
- RNNs are very strong sequence learners, and can pick-up on very intricate patterns in the data.
- An example of naturally occurring positive and negative sentences in the movie-reviews domain would be the following:
 - Positive: It's not life-affirming—it's vulgar and mean, but I liked it.
 - Negative: It's a disappointing that it only manages to be decent instead of dead brilliant.
 - Note that the positive example contains some negative phrases (not life affirming, vulgar, and mean).
 - While the negative example contains some positive ones (dead brilliant).
- Correctly predicting the sentiment requires understanding not only the individual phrases but also the context in which they occur, linguistic constructs such as negation, and the overall structure of the sentence.
- The sentence-level sentiment classification task is modelled using an RNN-acceptor.
- After tokenization, the RNN reads in the words of the sentence one at a time.

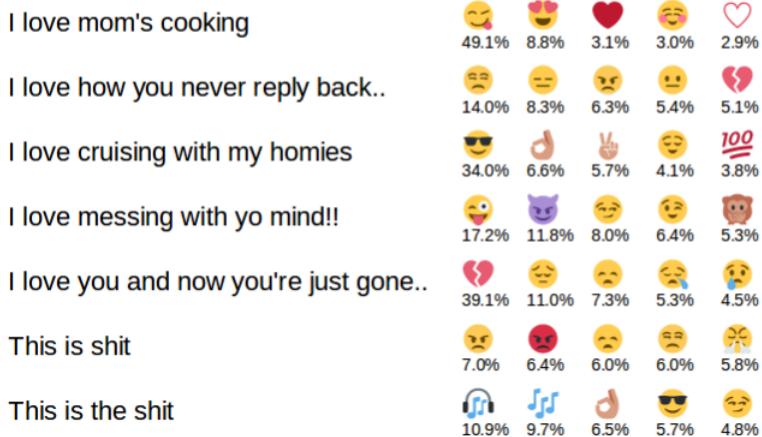
- The final RNN state is then fed into an MLP followed by a softmax-layer with two outputs (positive and negative).
- The network is trained with cross-entropy loss based on the gold sentiment labels.

$$\begin{aligned}
 p(\text{label} = k | \vec{w}_{1:n}) &= \hat{\vec{y}}_{[k]} \\
 \hat{\vec{y}} &= \text{softmax}(\text{MLP}(\text{RNN}(\vec{x}_{1:n}))) \\
 \vec{x}_{1:n} &= E_{[w_1]}, \dots, E_{[w_n]}
 \end{aligned} \tag{10.5}$$

- The word embeddings matrix E is initialized using pre-trained embeddings learned over a large external corpus using an algorithm such as Word2vec or Glove with a relatively wide window.
- It is often helpful to extend the model by considering bidirectional RNNs.
- For longer sentences, [Li et al., 2015] found it useful to use a hierarchical architecture, in which the sentence is split into smaller spans based on punctuation.
- Then, each span is fed into a bidirectional RNN.
- Sequence of resulting vectors (one for each span) are then fed into an RNN acceptor.
- A similar hierarchical architecture was used for document-level sentiment classification in [Tang et al., 2015].

10.12. Twitter Sentiment Classification with LSTMS Emojis

- An emoji-based distant supervision model for detecting sentiment and other affective states from short social media messages was proposed in [Felbo et al., 2017].
- Emojis are used as a distant supervision approach for various affect detection tasks (e.g., emotion, sentiment, sarcasm) using a large corpus of 634M tweets with 64 emojis.
- A neural network architecture is pretrained with this corpus.
- The network is an LSTM variant formed by an embedding layer, 2 bidirectional LSTM layers with normal skip connections and temporal average pooling-skip connections.

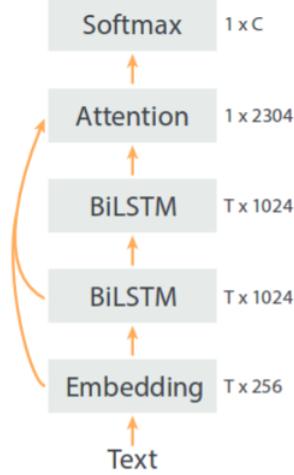


- Authors propose the chain-thaw transfer-learning approach in which the pretrained network is fine-tuned for the target task.
- Here, each layer is individually fine-tuned in each step with the target gold data, and then they are all fine-tuned together.
- The model achieves state-of-the-art results the detection of emotion, sentiment, and sarcasm.
- The pretrained network is released to the public.
- A demo of the model: <https://deepmoji.mit.edu/>.

10.13. Bi-LSTM CRF

- An alternative approach to the transducer for sequence labeling is to combine a BI-LSTM with a Conditional Random Field (CRF).
- This produces a powerful tagger [Huang et al., 2015] called BI-LSTM-CRF, in which the LSTM acts as feature extractor, and the CRF as a special layer that models the transitions from one tag to another.
- The CRF layers performs a global normalization over all possible sequences which helps to find the optimum sequence.
- In a CRF we model the conditional probability of the output tag sequence given the input sequence:

$$P(s_1, \dots, s_m | x_1, \dots, x_m) = P(s_{1:m} | x_{1:m})$$



- We do this by defining a feature map

$$\vec{\Phi}(x_{1:m}, s_{1:m}) \in \mathcal{R}^d$$

that maps an entire input sequence $x_{1:m}$ paired with an entire tag sequence $s_{1:m}$ to some d -dimensional feature vector.

- Then we can model the probability as a log-linear model with the parameter vector $\vec{w} \in \mathcal{R}^d$:

$$P(s_{1:m}|x_{1:m}; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))}$$

where $s'_{1:m}$ ranges over all possible output sequences.

- We can view the expression $\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}) = \text{score}_{crf}(x_{1:m}, s_{1:m})$ as a score of how well the tag sequence fits the given input sequence.
- Recall from the CRF lecture that $\vec{\Phi}(x_{1:m}, s_{1:m})$ was created with manually designed features.
- The idea of the LSTM CRF is to replace $\vec{\Phi}(x_{1:m}, s_{1:m})$ with the output of an LSTM transducer (automatically learned features):

$$\text{score}_{lstm-crif}(x_{1:m}, s_{1:m}) = \sum_{i=0}^m W_{[s_{i-1}, s_i]} \cdot \text{LSTM}^*(x_{1:m})_{[i]} + \vec{b}[s_{i-1}, s_i]$$

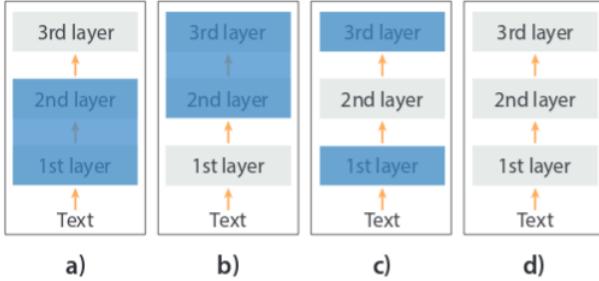


Figure 2: Illustration of the chain-thaw transfer learning approach, where each layer is fine-tuned separately. Layers covered with a blue rectangle are frozen. Step a) tunes any new layers, b) then tunes the 1st layer and c) the next layer until all layers have been fine-tuned individually. Lastly, in step d) all layers are fine-tuned together.

where $W_{[s_{i-1}, s_i]}$ corresponds to a transition score from tag s_{i-1} to s_i , $\vec{b}[s_{i-1}, s_i]$ is a bias term for the transition and $\text{LSTM}^*(x_{1:m})_{[i]}$ comes from the hidden state of the Bi-LSTM at timestep i.

- All these parameters are learned jointly.
- Note that the transition matrix W is position independent.
- The forward-backward algorithm is used during training and the Viterbi algorithm during decoding.
- More info in³ and⁴.

³https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html

⁴<https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/>

Capítulo 11

Modelos Secuencia a Secuencia y Atención Neuronal

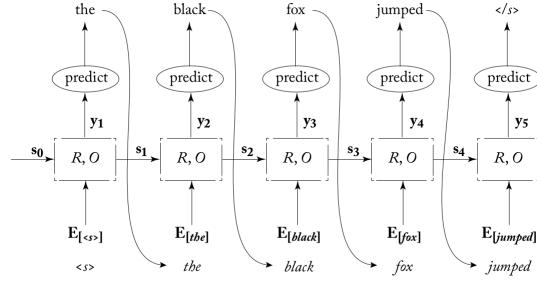
11.1. Language Models and Language Generation

- Language modeling is the task of assigning a probability to sentences in a language.
- Example: what is the probability of seeing the sentence “the lazy dog barked loudly”?
- The task can be formulated as the task of predicting the probability of seeing a word conditioned on previous words:

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

- RNNs can be used to train language models by tying the output at time i with its input at time $i + 1$.
- This network can be used to generate sequences of words or random sentences.
- Generation process: predict a probability distribution over the first word conditioned on the start symbol, and draw a random word according to the predicted distribution.
- Then predict a probability distribution over the second word conditioned on the first, and so on, until predicting the end-of-sequence $</s>$ symbol.

- After predicting a distribution over the next output symbols $P(t_i = k|t_{1:i-1})$, a token t_i is chosen and its corresponding embedding vector is fed as the input to the next step.



- Teacher-forcing: during **training** the generator is fed with the ground-truth previous word even if its own prediction put a small probability mass on it.
- It is likely that the generator would have generated a different word at this state in **test time**.

11.2. Sequence to Sequence Problems

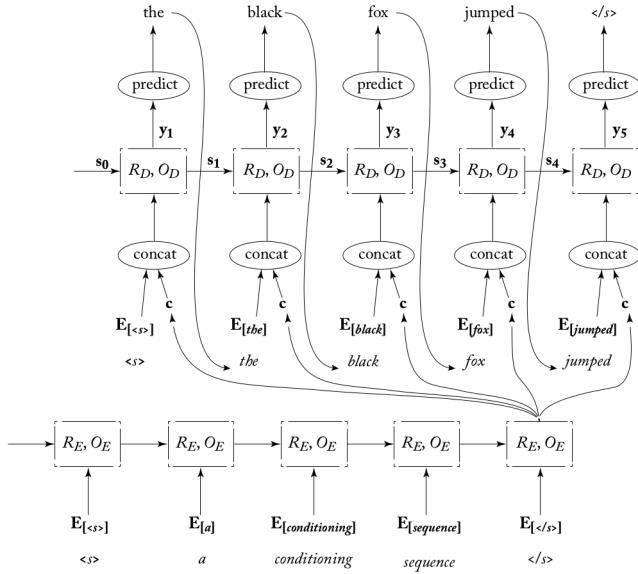
Nearly any task in NLP can be formulated as a sequence to sequence (or conditioned generation) task i.e., generate output sequences from input ones. Input and output sequences can have different lengths.

- Machine Translation: source language to target language.
- Summarization: long text to short text.
- Dialogue (chatbots): previous utterances to next utterance.

11.2.1. Conditioned Generation

- While using the RNN as a generator is a cute exercise for demonstrating its strength, the power of RNN generator is really revealed when moving to a conditioned generation or encoder-decoder framework.
- Core idea: using two RNNs.
- Encoder: One RNN is used to encode the source input into a vector \vec{c} .
- Decoder: Another RNN is used to decode the encoder's output and generate the target output.

- At each stage of the generation process the context vector \vec{c} is concatenated to the input \hat{t}_j and the concatenation is fed into the RNN. Encoder Decoder Framework



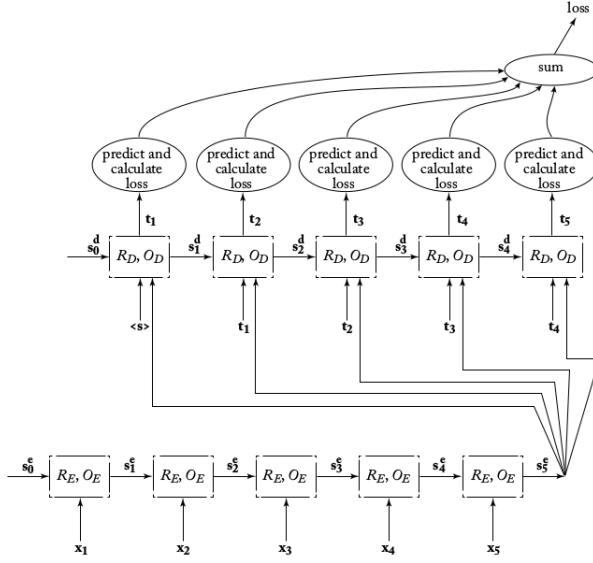
- This setup is useful for mapping sequences of length n to sequences of length m .
- The encoder summarizes the source sentence as a vector \vec{c} .
- The decoder RNN is then used to predict (using a language modeling objective) the target sequence words conditioned on the previously predicted words as well as the encoded sentence \vec{c} .
- The encoder and decoder RNNs are trained jointly.
- The supervision happens only for the decoder RNN, but the gradients are propagated all the way back to the encoder RNN.

Sequence to Sequence Training Graph

Neural Machine Translation

Machine Translation BLEU progress over time [Edinburgh En-De WMT]

⁰source: http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf



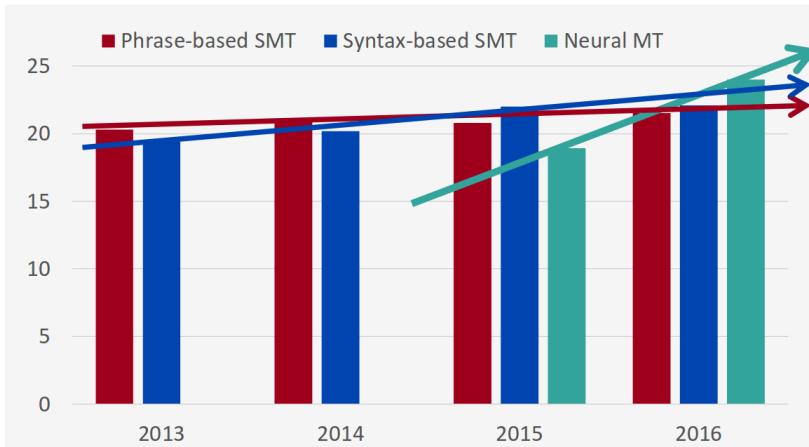
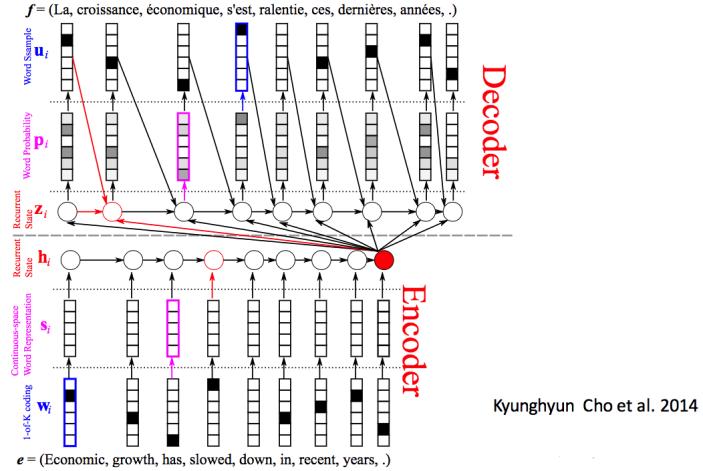
11.3. Decoding Approaches

- The decoder aims to generate the output sequence with maximal score (or maximal probability), i.e., such that $\sum_{i=1}^n P(\hat{t}_i | \hat{t}_{1:i-1})$ is maximized.
- The non-markovian nature of the RNN means that the probability function cannot be decomposed into factors that allow for exact search using standard dynamic programming.
- Exact search: finding the optimum sequence requires evaluating every possible sequence (computationally prohibitive).
- Thus, it only makes sense to solving the optimization problem above approximately.
- Greedy search: choose the highest scoring prediction (word) at each step.
- This may result in sub-optimal overall probability leading to prefixes that are followed by low-probability events.

11.3.1. Beam Search

- Beam search interpolates between the exact search and the greedy search by changing the size K of hypotheses maintained throughout the search procedure [Cho, 2015].

⁰Source: [Cho, 2015]



- The Beam search algorithm works in stages.
- We first pick the K starting words with the highest probability
- At each step, each candidate sequence is expanded with all possible next steps.
- Each candidate step is scored.
- The K sequences with the most likely probabilities are retained and all other candidates are pruned.
- The search process can halt for each candidate separately either by reaching a maximum length, by reaching an end-of-sequence token, or by reaching a threshold likelihood.
- The sentence with the highest overall probability is selected.

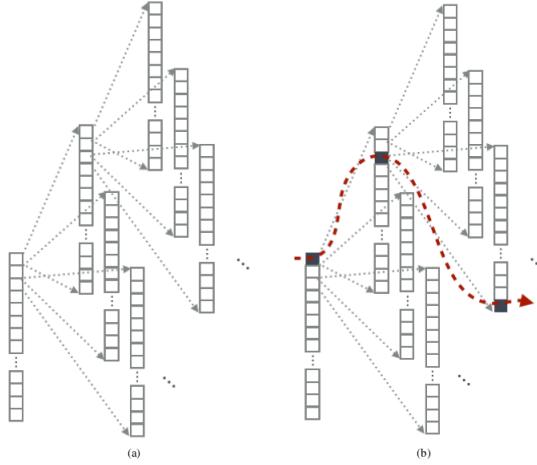


Figure 6.4: (a) Search space depicted as a tree. (b) Greedy search.

11.4. Conditioned Generation with Attention

- In the encoder-decoder networks the input sentence is encoded into a single vector, which is then used as a conditioning context for an RNN-generator.
- This architecture forces the encoded vector \vec{c} to contain all the information required for generation.
- It doesn't work well for long sentences!
- It also requires the generator to be able to extract this information from the fixed-length vector.
- "You can't cram the meaning of a whole sentence into a single vector!" -Raymond Mooney
- This architecture can be substantially improved (in many cases) by the addition of an attention mechanism.
- The attention mechanism attempts to solve this problem by allowing the decoder to "look back" at the encoder's hidden states based on its current state.
- The input sentence (a length n input sequence $\vec{x}_{1:n}$) is encoded using a biRNN as a sequence of vectors $\vec{c}_{1:n}$.
- The decoder uses a soft attention mechanism in order to decide on which parts of the encoding input it should focus.

⁰More info at: <https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/>

- At each stage j the decoder sees a weighted average of the vectors $\vec{c}_{1:n}$, where the attention weights ($\vec{\alpha}^j$) are chosen by the attention mechanism.

$$\vec{c}^j = \sum_{i=1}^n \vec{\alpha}_{[i]}^j \cdot \vec{c}_i$$

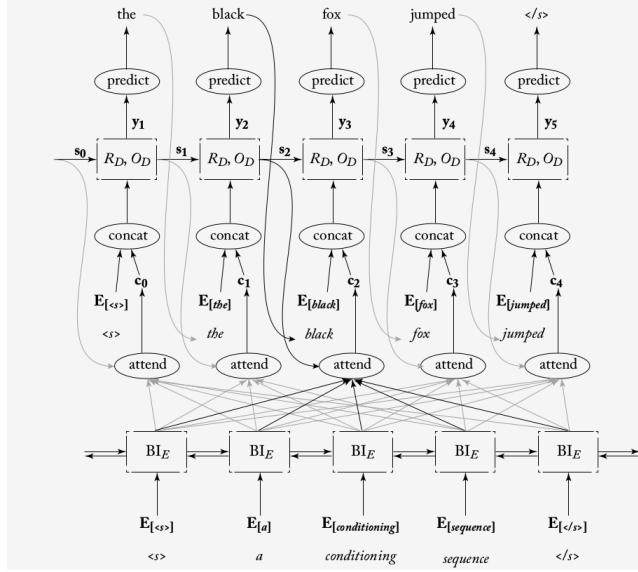
- The elements of $\vec{\alpha}^j$ are all positive and sum to one.
- Unnormalized attention weights ($\vec{\alpha}_{[i]}^j$) are produced taking into account the decoder state at time j (\vec{s}_j) and each of the vectors \vec{c}_i .
- They can be obtained in various ways, basically any differentiable function returning a scalar out of two vectors \vec{s}_j and \vec{c}_i could be employed.
- The simplest approach is a dot product: $\vec{\alpha}_{[i]}^j = \vec{s}_j \cdot \vec{c}_i$.
- The one we will use in these slides is Additive attention, which uses a Multilayer Perceptron: $\vec{\alpha}_{[i]}^j = \text{MLP}^{\text{att}}([\vec{s}_j; \vec{c}_i]) = \vec{v} \cdot \tanh([\vec{s}_j; \vec{c}_i]U + \vec{b})$
- These unnormalized weights are then normalized into a probability distribution using the softmax function.

$$\begin{aligned} \text{attend}(c_{1:n}, \hat{c}_{1:j}) &= c^j \\ c^j &= \sum_{i=1}^n \alpha_{[i]}^j \cdot c_i \\ \alpha^j &= \text{softmax}(\tilde{\alpha}_{[1]}^j, \dots, \tilde{\alpha}_{[n]}^j) \\ \tilde{\alpha}_{[i]}^j &= \text{MLP}^{\text{att}}([s_j; c_i]), \end{aligned}$$

- The encoder, decoder, and attention mechanism are all trained jointly in order to play well with each other.

The entire sequence-to-sequence generation with attention is given by:

- Why use the biRNN encoder to translate the conditioning sequence $\vec{x}_{1:n}$ into the context vectors $\vec{c}_{1:n}$?
- Why we just don't attend directly on the inputs (word embeddings) $\text{MLP}^{\text{att}}([\vec{s}_j; \vec{x}_i])$?
- We could, but we get important benefits from the encoding process.
- First, the biRNN vectors \vec{c}_i represent the items \vec{x}_i in their sentential context.
- Sentential context: a window focused around the input item \vec{x}_i and not the item itself.



- Second, by having a trainable encoding component that is trained jointly with the decoder, the encoder and decoder evolve together.
- Hence, the network can learn to encode relevant properties of the input that are useful for decoding, and that may not be present at the source sequence $\vec{x}_{1:n}$ directly.

11.5. Attention and Word Alignments

In the context of machine translation, one can think of MLP^{att} as computing a soft alignment between the current decoder state \vec{s}_j (capturing the recently produced foreign words) and each of the source sentence components \vec{c}_i .

11.6. Other types of Attention

Source: <https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>

$$\begin{aligned}
p(t_{j+1} = k \mid \hat{t}_{1:j}, \mathbf{x}_{1:n}) &= f(O_{\text{dec}}(s_{j+1})) \\
s_{j+1} &= R_{\text{dec}}(s_j, [\hat{t}_j; c^j]) \\
c^j &= \sum_{i=1}^n \alpha_{[i]}^j \cdot c_i \\
c_{1:n} &= \text{biRNN}_{\text{enc}}^*(x_{1:n}) \\
\alpha^j &= \text{softmax}(\tilde{\alpha}_{[1]}^j, \dots, \tilde{\alpha}_{[n]}^j) \\
\tilde{\alpha}_{[i]}^j &= \text{MLP}^{\text{att}}([s_j; c_i]) \\
\hat{t}_j &\sim p(t_j \mid \hat{t}_{1:j-1}, \mathbf{x}_{1:n}) \\
f(z) &= \text{softmax}(\text{MLP}^{\text{out}}(z)) \\
\text{MLP}^{\text{att}}([s_j; c_i]) &= v \tanh([s_j; c_i]U + b).
\end{aligned}$$

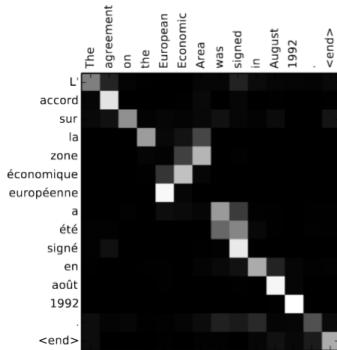


Fig. 2. Visualization of the attention weights α_j^t of the attention-based neural machine translation model [32]. Each row corresponds to the output symbol, and each column the input symbol. Brighter the higher α_j^t .

Figura 11.1: Source: [Cho et al., 2015]

Summary

Below is a summary table of several popular attention mechanisms (or broader categories of attention mechanisms).

Name	Alignment score function	Citation
Additive(*)	$\text{score}(\mathbf{s}_i, \mathbf{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\mathbf{s}_i; \mathbf{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{i,j} = \text{softmax}(\mathbf{W}_a \mathbf{s}_i)$ Note: This simplifies the softmax alignment max to only depend on the target position.	Luong2015
General	$\text{score}(\mathbf{s}_i, \mathbf{h}_i) = \mathbf{s}_i^\top \mathbf{W}_a \mathbf{h}_i$ where \mathbf{W}_a is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\mathbf{s}_i, \mathbf{h}_i) = \mathbf{s}_i^\top \mathbf{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\mathbf{s}_i, \mathbf{h}_i) = \frac{\mathbf{s}_i^\top \mathbf{h}_i}{\sqrt{n}}$ 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.

Capítulo 12

Arquitectura de Transformer

12.1. What's Wrong with RNNs?

- Given what we just learned on previous lecture, it would seem like attention solves all the problems with RNNs and encoder-decoder architectures¹.
- There are a few shortcomings of RNNs that another architecture called the **Transformer** tries to address.
- The Transformer discards the recursive component of the Encoder-Decoder architecture and purely relies on attention mechanisms [Vaswani et al., 2017].
- When we process a sequence using RNNs, each hidden state depends on the previous hidden state.
- This becomes a major pain point on GPUs: GPUs have a lot of computational capability and they hate having to wait for data to become available.
- Even with technologies like CuDNN, RNNs are painfully inefficient and slow on the GPU.

12.1.1. Dependencies in neural machine translations

In essence, there are three kinds of dependencies in neural machine translations:

1. Dependencies between the input and output tokens.
2. Dependencies between the input tokens themselves.

¹The following material is based on: <http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/> and <http://jalammar.github.io/illustrated-transformer/>

3. Dependencies between the output tokens themselves.

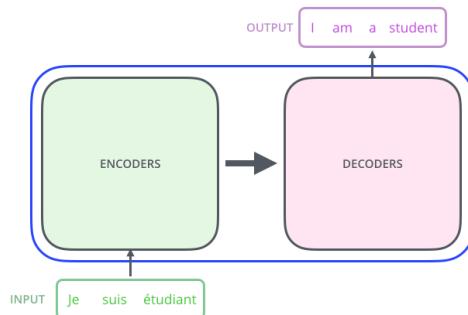
The traditional attention mechanism largely solved the first dependency by giving the decoder access to the entire input sequence. The second and third dependencies were addressed by the RNNs.

12.2. The Transformer

- The novel idea of the Transformer is to extend this mechanism to the processing input and output sentences as well.
- The RNN processes input sequences sequentially.
- The Transformer, on the other hand, allows the encoder and decoder to see the entire input sequence all at once.
- This is done using attention.
- Let's begin by looking at the Transformer as a single black box.
- In a machine translation application, it would take a sentence in one language, and output its translation in another.



This blackbox can be decomposed by an encoding component, a decoding component, and connections between them.



- The encoding component is a stack of encoders.
- The original Transformer stacks six of them on top of each other.
- But there's nothing magical about the number six, one can definitely experiment with other arrangements.
- The decoding component is a stack of decoders of the same number.

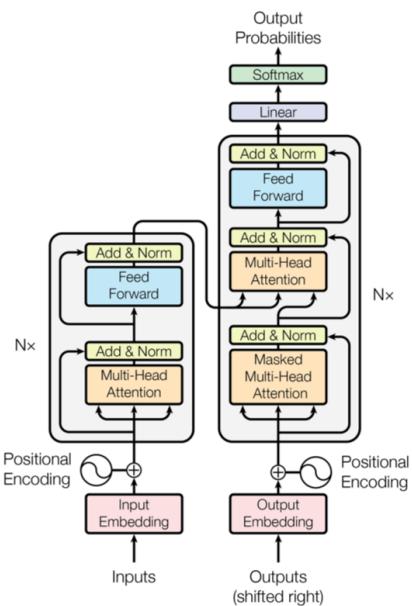
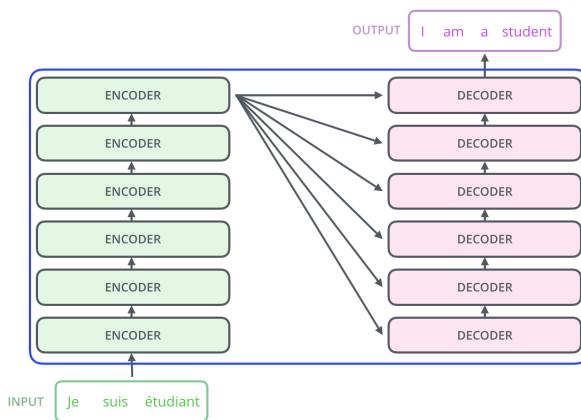
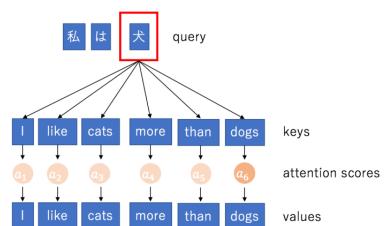


Figure 1: The Transformer - model architecture.

- The Transformer still uses the basic encoder-decoder design of RNN neural machine translation systems.
- The left-hand side is the encoder, and the right-hand side is the decoder.
- The initial inputs to the encoder are the embeddings of the input sequence.
- The initial inputs to the decoder are the embeddings of the outputs up to that point.
- The encoder and decoder are composed of N blocks (where $N = 6$ for both networks).
- These blocks are composed of smaller blocks as well.
- Before looking at each block in further detail, let's try to understand the Attention mechanism implemented by the Transformer.

12.3. Attention Mechanism in the Transformer

- The attention mechanism in the Transformer is interpreted as a way of computing the relevance of a set of **values** (information) based on some **keys** and **queries**.
- The attention mechanism is used as a way for the model to **focus on relevant information** based on what it is currently processing.
- In the RNN encoder-decoder architecture with attention:
 1. Attention weights were the relevance of the encoder hidden states (values) in processing the decoder state (query).
 2. These values were calculated based on the encoder hidden states (keys) and the decoder hidden state (query).



- In this example, the query is the word being decoded (which means dog) and both the keys and values are the source sentence.

- The attention score represents the relevance, and in this case is large for the word “dog” and small for others.
- When we think of attention this way, we can see that the keys, values, and queries could be anything.
- They could even be the same.
- For instance, both values and queries could be input embeddings (self attention).

12.3.1. Queries

- Queries are representations of the target or output sequence that the Transformer model uses to determine how much attention should be given to each word in the input sequence.
- Each word or token in the output sequence is associated with a query.
- The queries help in retrieving relevant information from the input sequence.
- Example: If we are generating the translation for the sentence “Je adore les chats” (meaning I love cats in French), each word in the output sequence would have its corresponding query representation.
- For example, “Je” would have a query vector, “adore” would have a query vector, and so on.

12.3.2. Keys

- Keys are representations of the input sequence that the Transformer model uses to compute the attention scores.
- Each word or token in the input sequence is associated with a key.
- These keys capture the information needed to understand the context and relationships between words in the sequence.
- Example: Consider the input sequence: “I love cats.”
- Each word in the sequence would have its corresponding key representation, such as “I” having a key vector, “love” having a key vector, and so on.

12.3.3. Values

- Values are the actual information or features associated with each word in the input sequence.
- These values are used to calculate the weighted sum during the attention computation, which helps determine the importance or relevance of each word.
- Example: Considering the same input sequence “I love cats,” each word would have its associated value representation.
- These values contain the contextual information for each word.
- For instance, “I” would have a value vector, “love” would have a value vector, and so forth.
- Keys and values are very difficult to distinguish at first sight because in RNN encoding and decoding with classical attention they are the same.
- We will see with the Transformer that although they come from the same sequence, they may correspond to different vectors.

12.3.4. Scaled Dot Product Attention

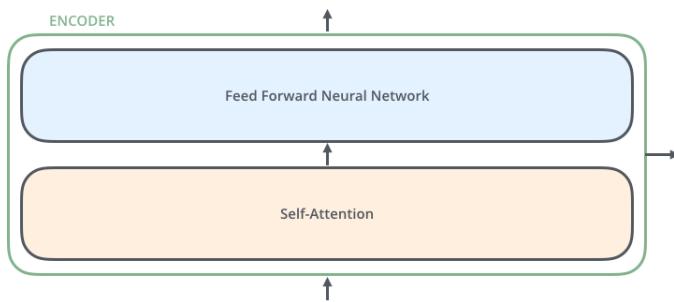
- Transformer uses a particular form of attention called the “Scaled Dot-Product Attention”:
- For a given query vector \vec{q} , a sequence of key vectors $\vec{k}_{1:m}$, and a sequence of value vectors $\vec{v}_{1:m}$, the attention weights $\alpha_1, \dots, \alpha_m$ are computed as follows:
$$\alpha_1, \dots, \alpha_m = \text{softmax} \left(\frac{\vec{q} \cdot \vec{k}_1}{\sqrt{d}}, \dots, \frac{\vec{q} \cdot \vec{k}_m}{\sqrt{d}} \right)$$
- d represents the dimensionality of the queries and keys.
- The normalization over \sqrt{d} is used to rescale the dot products between queries and keys (dot products tend to grow with the dimensionality).
- The attention weights are then multiplied by their corresponding values to compute a weighted sum, which is then passed to the subsequent layers of the network:

$$\alpha_1 * \vec{v}_1 + \dots + \alpha_m * \vec{v}_m$$

- Now we are ready to take a closer look at each part of the Transformer.

12.4. The Encoder

- The encoder contains self-attention layers.
- In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder.
- Each position in the encoder can attend to all positions in the previous layer of the encoder.
- The encoder is composed of two blocks (which we will call sub-layers to distinguish from the N blocks composing the encoder and decoder).
- One is the Multi-Head Attention sub-layer over the inputs, mentioned above.
- The other is a simple feed-forward network.

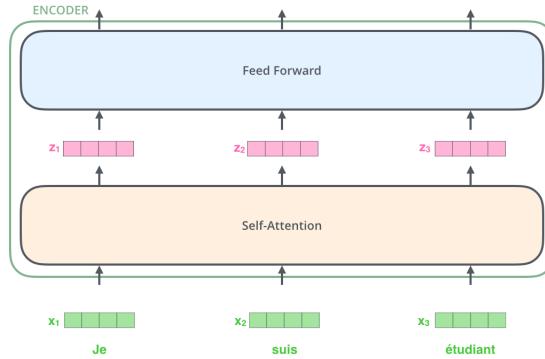


Bringing The Tensors Into The Picture

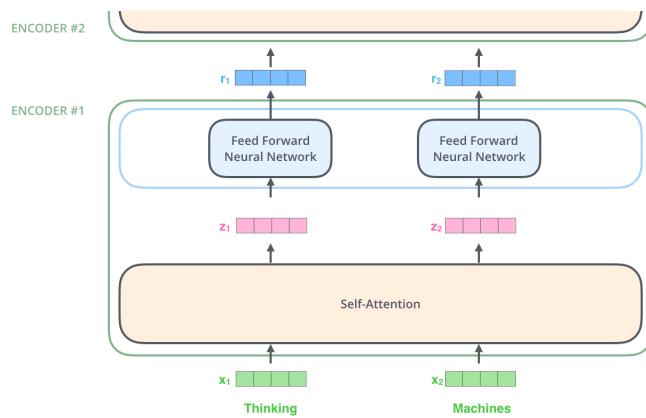
- We begin by turning each input word into a vector using an embedding layer of the size 512 in the bottom-most encoder.



- In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below.
- The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

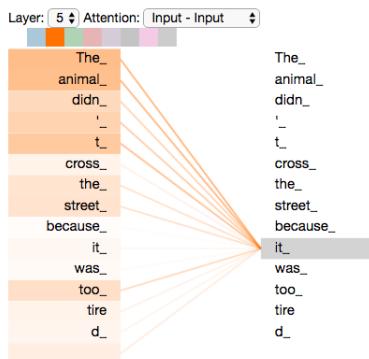


- The word in each position flows through its own path in the encoder.
- There are dependencies between these paths in the self-attention layer.
- The feed-forward layer does not have those dependencies.
- However, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.
- Now We're Encoding!: As we've mentioned already, an encoder receives a list of vectors as input.
- It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



12.5. Self-Attention at a High Level

- Say the following sentence is an input sentence we want to translate:
“The animal didn’t cross the street because it was too tired”
- What does “it” in this sentence refer to?
- Is it referring to the street or to the animal?
- It’s a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.
- As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.
- Think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it’s processing.
- Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.

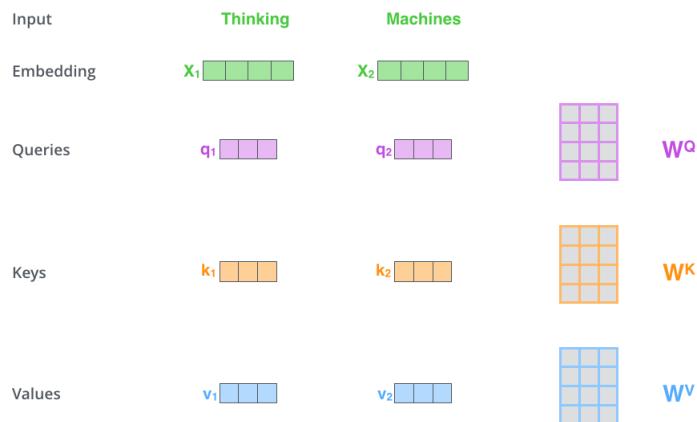


12.6. Self-Attention in Detail

step 1

- The **first step** in calculating scaled dot product self-attention is to create three vectors from each of the encoder’s input vectors (in this case, the embedding of each word).

- So for each word, we create a Query vector, a Key vector, and a Value vector.
- These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
- Notice that these new vectors are smaller in dimension than the embedding vector.
- Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.
- They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

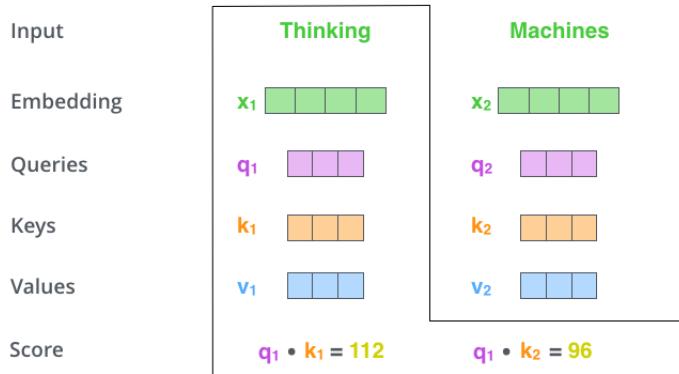


- Example sentence: “Thinking Machines”.
- Multiplying x_1 by the W^Q weight matrix produces q_1 , the “query” vector associated with that word.
- We end up creating a “query”, a “key”, and a “value” projection of each word in the input sentence.

Step 2

- The **second step** in calculating self-attention is to calculate a score.
- Say we're calculating the self-attention for the first word in this example, “Thinking”.
- We need to score each word of the input sentence against this word.
- The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

- The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.
- So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q_1 and k_1 .
- The second score would be the dot product of q_1 and k_2 .



Steps 3 and 4

- The **third** and **fourth** steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64).
- This leads to having more stable gradients.
- There could be other possible values here, but this is the default), then pass the result through a softmax operation.
- Softmax normalizes the scores so they're all positive and add up to 1.
- This softmax score determines how much each word will be expressed at the current position.
- Clearly the word at the current position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

Self-Attention in Detail: steps 5 and 6

- The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up).
- The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

Input	Thinking	Machines
Embedding	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)	14	12
Softmax	0.88	0.12

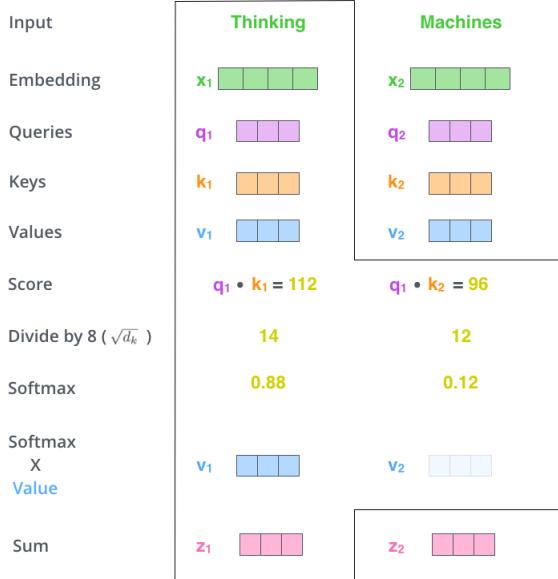
- The **sixth step** is to sum up the weighted value vectors.
- This produces the output of the self-attention layer at the current position (for the first word in this example).
- That concludes the scaled dot product self-attention calculation.
- The resulting vector is one we can send along to the feed-forward neural network.

12.7. Matrix Calculation of Self-Attention

- In the actual implementation, scaled dot product self-attention is computed in matrix form for faster processing.
- So let's look at that now that we've seen the intuition of the calculation on the word level.
- The first step is to calculate the Query, Key, and Value matrices: Q , K , V .
- We do that by packing our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (WQ , WK , WV).

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) * V$$

- Every row in the X matrix corresponds to a word in the input sentence.



- We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure).
- Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

12.8. Multi-Head Attention

- If we only compute a single attention weighted sum of the values, it would be difficult to capture various different aspects of the input.
- In the example above, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself.
- If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.
- To expand the model's to learn diverse representations focusing on different positions, the Transformer uses the Multi-Head Attention block.
- Multi-head attention computes multiple attention weighted sums instead of a single attention pass over the values.

$$X \times W^Q = Q$$

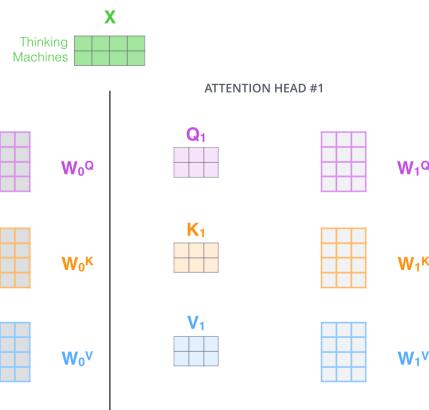
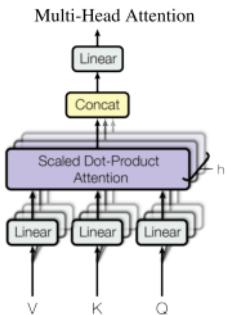
$$X \times W^K = K$$

$$X \times W^V = V$$

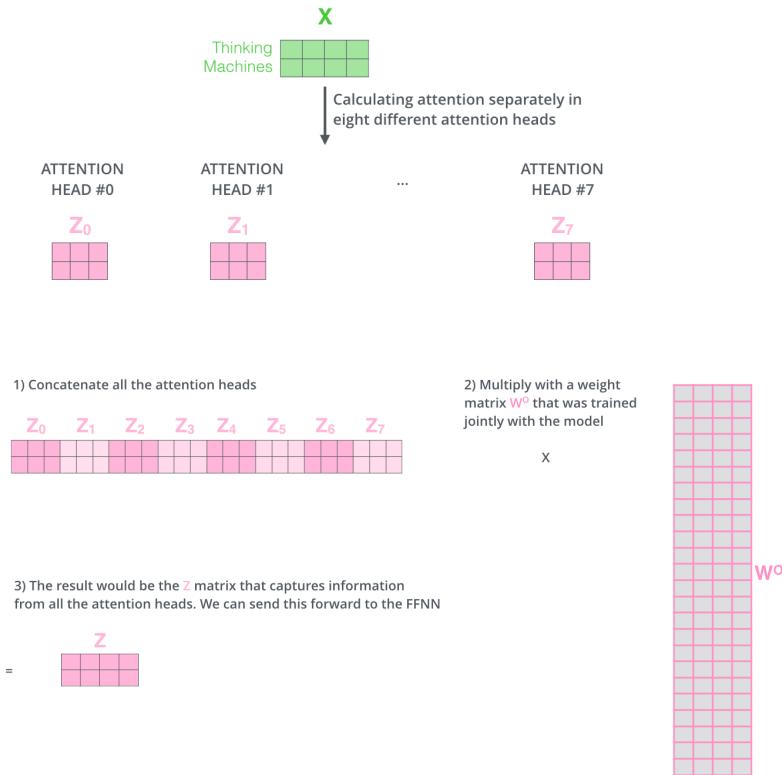
$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V$$

$$= Z$$

- In essence, it applies different linear transformations to the values, keys, and queries for each “head” of attention.
- The Multi-Head Attention block applies multiple scaled dot product attention blocks in parallel, concatenates their outputs, then applies one single linear transformation.
- With multi-headed attention, we have not only one, but multiple sets of Query/Key/Value weight matrices.
- The Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder.
- Each of these sets is randomly initialized.
- Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.
- The multi-headed component gives the attention layer multiple “representation subspaces”.



- With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices.
- As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.
- If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices.
- However, the feed-forward it's expecting a single matrix (a vector for each word) not eight matrices.
- So we need a way to condense these eight down into a single matrix.
- This is done by concatenating the matrices and then multiplying them by an additional weight matrix WO.
- Let's put all these matrices in one visual so we can look at them in one place.

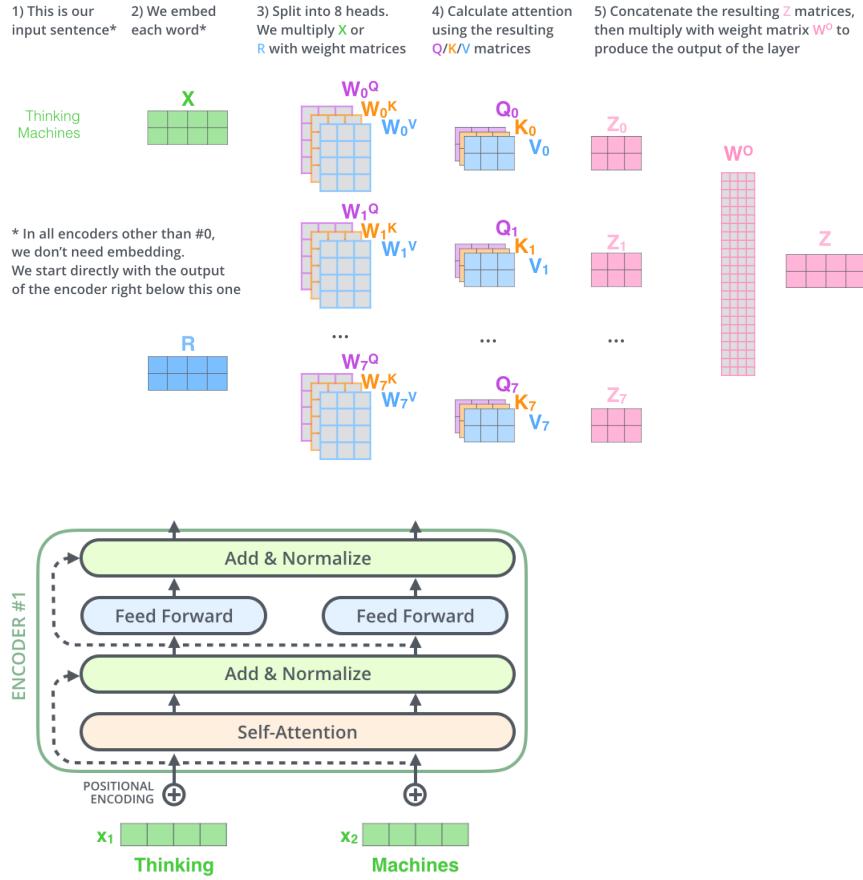


12.9. Residual Connections

- Between each sub-layer, there is a residual connection followed by a layer normalization.
- A residual connection is basically just taking the input and adding it to the output of the sub-network, and is a way of making training deep networks easier.
- Layer normalization is a normalization method in deep learning that is similar to batch normalization.
- If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

12.10. The Encoder: summary

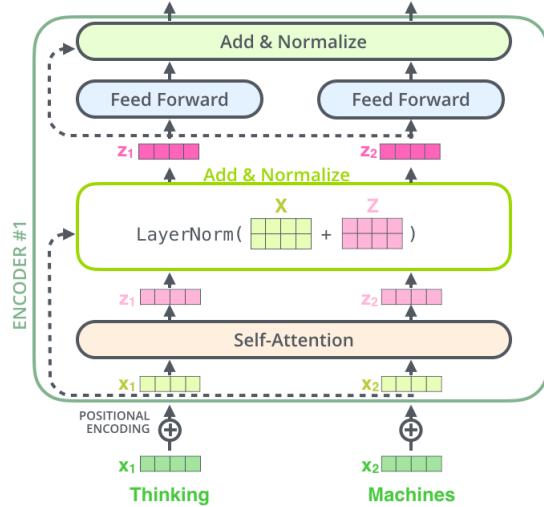
- What each encoder block is doing is actually just a bunch of matrix multiplications followed by a couple of element-wise transformations.



- This is why the Transformer is so fast: everything is just parallelizable matrix multiplications.
- The point is that by stacking these transformations on top of each other, we can create a very powerful network.
- The core of this is the attention mechanism which modifies and attends over a wide range of information.

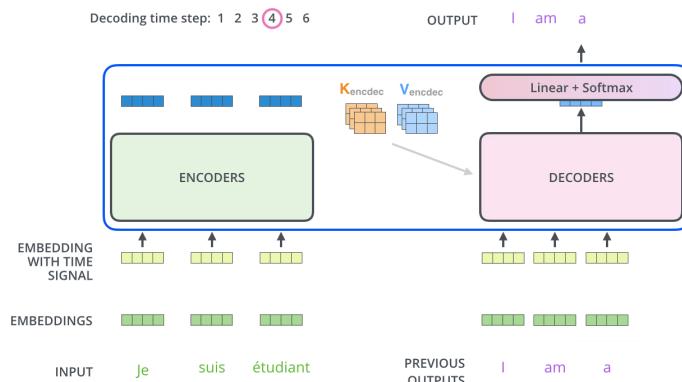
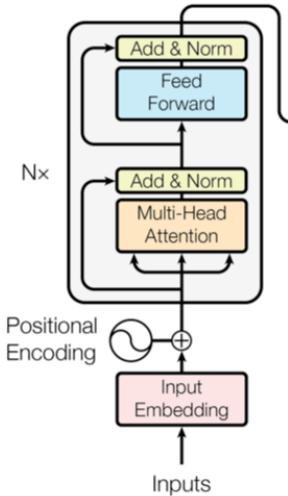
12.11. The Decoder

- The Transformer decoder consists of two types of attention layers: self-attention and encoder-decoder attention.
- Self-attention layers in the decoder allow each position in the decoder to pay attention to all positions within the decoder itself, similar to how the



hidden state in RNN machine translation architectures works.

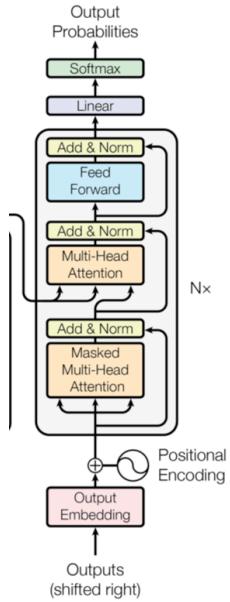
- On the other hand, the “encoder-decoder attention” layer enables the decoder to focus on relevant parts of the input sequence.
- The encoder-decoder attention layer functions similarly to multiheaded self-attention, but it generates its queries matrix from the layer below it, and uses the keys and values matrix from the output of the encoder stack.
- This allows each position in the decoder to attend to all positions in the input sequence, mimicking the typical encoder-decoder attention mechanisms seen in RNN sequence-to-sequence models.
- Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).
- This process is repeated until a special symbol is reached indicating the transformer decoder has completed its output.
- When we train the Transformer, we want to process all the sentences at the same time.
- However, if we give the decoder access to the entire target sentence, the model can just repeat the target sentence (in other words, it doesn’t need to learn anything).
- The self-attention layer should only be allowed to attend to earlier positions in the output sequence.
- This is done by masking the “future” tokens when decoding a certain word.



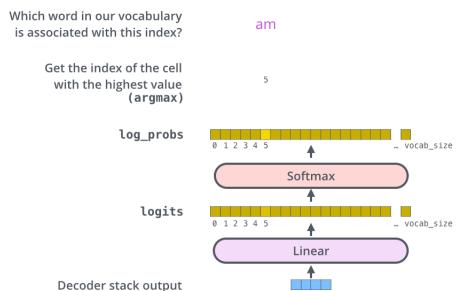
- The masking is done by setting to $-\infty$ all values in the input of the softmax which correspond to illegal connections.
- This is why “self attention blocks” in the decoder are referred to as “masked”: the inputs to the decoder from future time-steps are masked.

12.12. The Final Linear and Training

- The decoder stack generates a vector of floats as its output.
- To convert this vector into a word, the final Linear layer is used, followed by a Softmax layer.



- The **Linear** layer is a fully connected neural network that projects the decoder's output vector into a larger vector called a logits vector.
- The logits vector has a width equal to the size of the model's output vocabulary, which contains 10,000 unique English words.
- Each cell in the logits vector represents the score of a specific word.
- The softmax layer then transforms these scores into probabilities, ensuring they are all positive and sum up to 1.0.

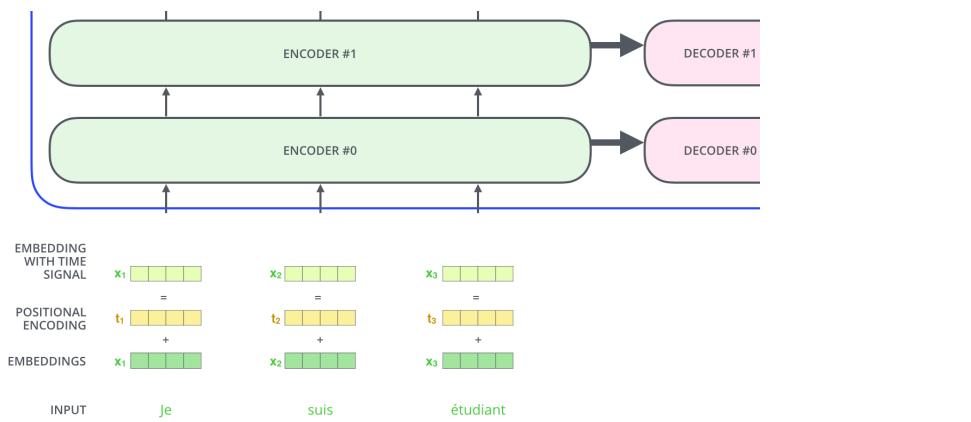


- The word with the highest probability is chosen as the output for the current time step.
- During the training of the Transformer, a cross-entropy loss is employed.

- This loss measures the difference between the predicted distribution of words and the target word, which is represented as a one-hot vector.

12.13. Positional Encodings

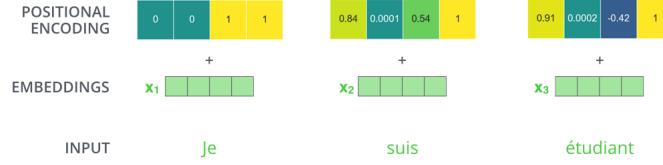
- Unlike recurrent networks, the multi-head attention network cannot naturally make use of the position of the words in the input sequence.
- Without positional encodings, the output of the multi-head attention network would be the same for the sentences “I like cats more than dogs” and “I like dogs more than cats”.
- Positional encodings explicitly encode the relative/absolute positions of the inputs as vectors and are then added to the input embeddings.



- To give the model a sense of the order of the words, we add positional encoding vectors – the values of which follow a specific pattern.
- The paper uses the following equation to compute the positional encodings:

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$$

$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}})$$
- Where pos represents the position, and i is the dimension.
- Basically, each dimension of the positional encoding is a wave with a different frequency.
- A real example of positional encoding with a toy embedding size of 4.



12.14. Conclusions

- The Transformer achieves better BLEU scores than previous state-of-the-art models for English-to-German translation and English-to-French translation at a fraction of the training cost.

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	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.0	$2.3 \cdot 10^{19}$	

- The Transformer is a powerful and efficient alternative to recurrent neural networks to model dependencies using only attention mechanisms.
- It has established itself as the de facto architecture in NLP and serves as the foundation for modern large language models.

Capítulo 13

Grandes Modelos de Lenguaje

13.1. Representations for a word

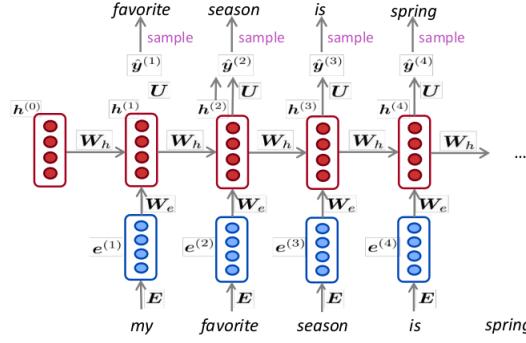
- So far, we've basically had one representation of words, the word embeddings we've already learned: Word2vec, GloVe, fastText.¹.
- These embeddings have a useful semi-supervised quality, as they can be learned from unlabeled corpora and used in our downstream task-oriented architectures (LSTM, CNN, Transformer).
- However, they exhibit two problems.
- Problem 1: They always produce the same representation for a word type regardless of the context in which a word token occurs
- We might want very fine-grained word sense disambiguation
- Problem 2: We just have one representation for a word, but words have different aspects, including semantics, syntactic behavior, and register/- connotations

13.2. Neural Language Models can produce Contextualized Embeddings

- In a Neural Language Model (NLM), we immediately stuck word vectors (perhaps only trained on the corpus) through LSTM layers
- Those LSTM layers are trained to predict the next word.

¹These slides are partially based on the Stanford CS224N: Natural Language Processing with Deep Learning course: <http://web.stanford.edu/class/cs224n/>

- But these language models produce context-specific word representations in the hidden states of each position.



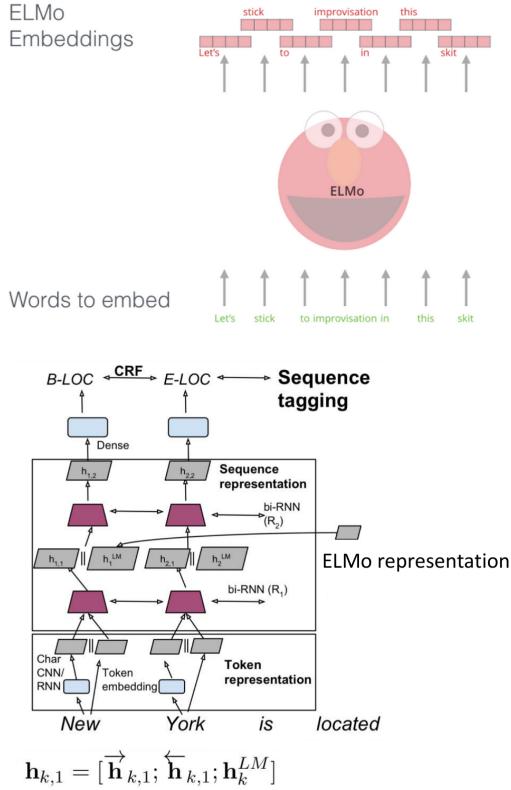
13.3. ELMo: Embeddings from Language Models

- Idea: train a large language model (LM) with a recurrent neural network and use its hidden states as “contextualized word embeddings” [Peters et al., 2018].
- ELMo is bidirectional LM with 2 biLSTM layers and around 100 million parameters.
- Uses character CNN to build initial word representation (only)
- 2048 char n-gram filters and 2 highway layers, 512 dim projection
- User 4096 dim hidden/cell LSTM states with 512 dim projections to next input
- Uses a residual connection
- Parameters of token input and output (softmax) are tied.

13.3.1. ELMo: Use with a task

- First run biLM to get representations for each word.
- Then let (whatever) end-task model use them.
- Freeze weights of ELMo for purposes of supervised model.
- Concatenate ELMo weights into task-specific model.

ELMo: Results



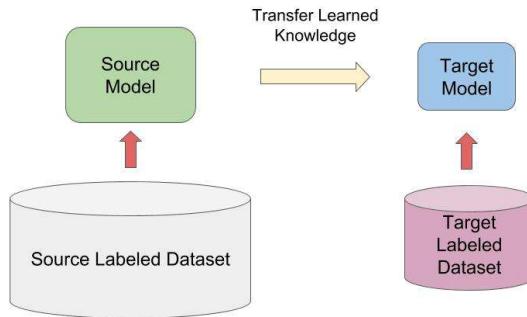
13.4. ULMfit

- Howard and Ruder (2018) Universal Language Model Fine-tuning for Text Classification [Howard and Ruder, 2018].
- Same general idea of transferring NLM knowledge
- Here applied to text classification
- Train LM on big general domain corpus (use biLM)
- Tune LM on target task data
- Fine-tune as classifier on target task

13.4.1. ULMfit emphases

- Use reasonable-size “1 GPU” language model not really huge one
- A lot of care in LM fine-tuning
- Different per-layer learning rates

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



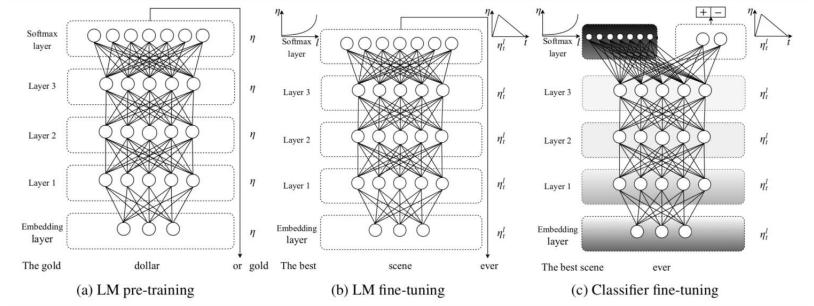
- Slanted triangular learning rate (STLR) schedule
- Gradual layer unfreezing and STLR when learning classifier
- Classify using concatenation $[h_T, \text{maxpool}(h), \text{meanpool}(h)]$

13.4.2. ULMfit transfer learning

13.5. Let's scale it up!

13.6. BERT (Bidirectional Encoder Representations from Transformers)

- Idea: combine ideas from ELMO, ULMFit and the Transformer [Kenton and Toutanova, 2019].
- How: Train a large model (335 million parameters) from a large unlabeled corpus using a Transformer encoder and then fine-tune it for other downstream tasks.

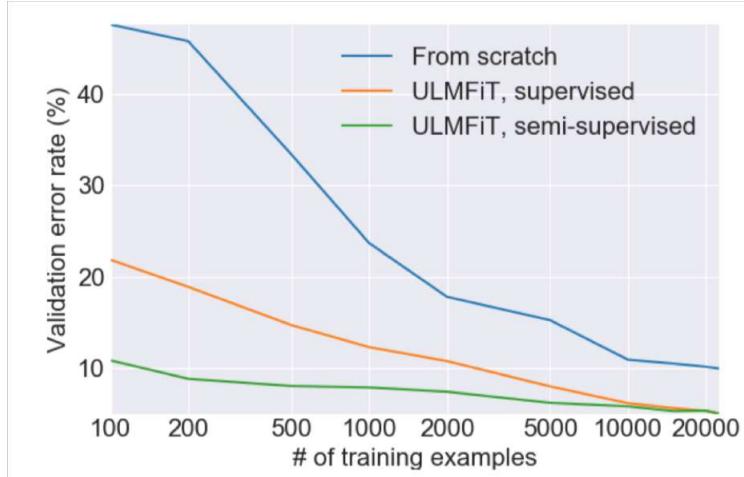


Model	Test	Model	Test
CoVe (McCann et al., 2017)	8.2	CoVe (McCann et al., 2017)	4.2
oh-LSTM (Johnson and Zhang, 2016)	5.9	TBCNN (Mou et al., 2015)	4.0
Virtual (Miyato et al., 2016)	5.9	LSTM-CNN (Zhou et al., 2016)	3.9
ULMFiT (ours)	4.6	ULMFiT (ours)	3.6

Text clas-

sifier error rates

- The parallelizable properties of the Transformer (unlike RNNs, which must be processed sequentially) allow the model to scale to more parameters.
- This model is related but a little bit different from a standard Language Model.
- BERT doesn't predict the next word in a sentence like a traditional language model, but rather learns utilizes a "**masked language modeling**" (MLM) objective during pre-training.
- In MLM, random words in a sentence are masked and the model is trained to predict those masked words based on the surrounding context.
- BERT also incorporates a "**next sentence prediction**" task, where pairs of sentences are fed to the model, and it learns to predict whether the second sentence follows the first in the original text.
- Fine-tuning** BERT involves adding a task-specific layer on top of the pre-trained model and training it on a labeled dataset for the target task.
- BERT achieved state-of-the-art results at the time of its release on NLP tasks, including sentence classification, named entity recognition, question answering, and more.



13.7. Masked Language Modeling and Next Sentence Prediction

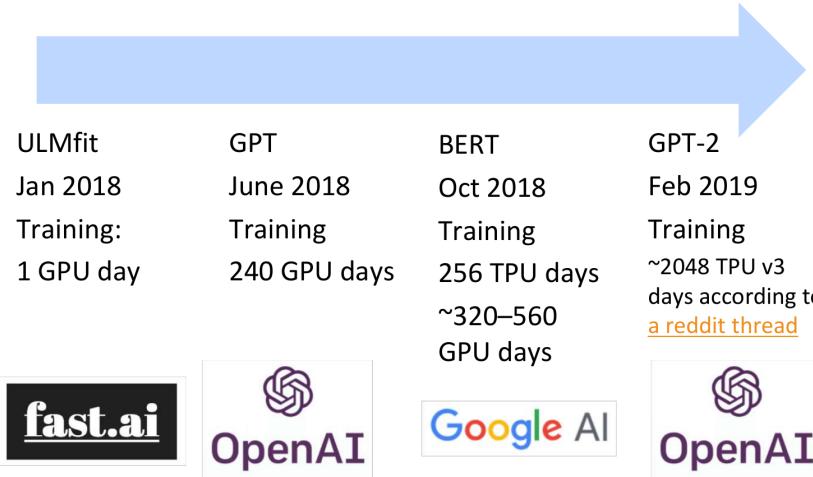
- MLM: Mask out k % of the input words, and then predict the masked words
- They always use k = 15 %.
- Too little masking: Too expensive to train
- Too much masking: Not enough context
- Next sentence prediction: To learn relationships between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

13.8. BERT sentence pair encoding

- Token embeddings: Words are divided into smaller units called word pieces, and each word piece is assigned a token embedding.
- BERT learns a segmented embedding [SEP] to differentiate between the two sentences in a pair.
- BERT utilizes positional embeddings to capture the position of each word within the sentence.

13.9. BERT Model Architecture and Training

- BERT is based on the Transformer encoder.



- The multi-headed self-attention block of the Transformer allows BERT to consider long-distance context effectively.
- The use of self-attention also enables efficient computations on GPU/TPU, with only a single multiplication per layer.
- BERT was trained on a large amount of unlabeled text data from Wikipedia and BookCorpus.
- Two different model sizes were trained:
 1. BERT-Base: 12 layers, 768 hidden units, and 12 attention heads.
 2. BERT-Large: 24 layers, 1024 hidden units, and 16 attention heads.
- The training process involved utilizing 4x4 or 8x8 TPU (Tensor Processing Unit) configurations for faster computation.
- Training BERT models took approximately 4 days to complete.

13.10. BERT model fine tuning

- Fine-tuning involves customizing the pre-trained BERT model for specific tasks.
- To fine-tune BERT, we add a task-specific layer on top of the pre-trained BERT model.
- The task-specific layer can vary depending on the task at hand, such as sequence labeling or sentence classification.
- We train the entire model, including the pre-trained BERT and the added task-specific layer, for the specific task.

All of these models are Transformer architecture models ... so maybe we had better learn about Transformers?		
ULMfit	GPT	GPT-2
Jan 2018	June 2018	Oct 2018
Training:	Training	Training
1 GPU day	240 GPU days	256 TPU days ~320–560 GPU days
		~2048 TPU v3 days according to a reddit thread
		
		



13.10.1. BERT results on GLUE tasks

- BERT was massively popular and hugely versatile; finetuning BERT led to new state-of-the-art results on a broad range of tasks.
- BERT's performance was assessed using the GLUE benchmark, a collection of diverse NLP tasks.
- The GLUE benchmark primarily consists of natural language inference tasks, but also includes sentence similarity and sentiment analysis tasks.

Example Task: MultiNLI (Natural Language Inference)

- Premise: "Hills and mountains are especially sanctified in Jainism."
- Hypothesis: "Jainism hates nature."
- Label: Contradiction

Example Task: CoLa

- Sentence: "The wagon rumbled down the road."



Sentence A = The man went to the store.
Sentence B = He bought a gallon of milk.
Label = IsNextSentence

Sentence A = The man went to the store.
Sentence B = Penguins are flightless.
Label = NotNextSentence

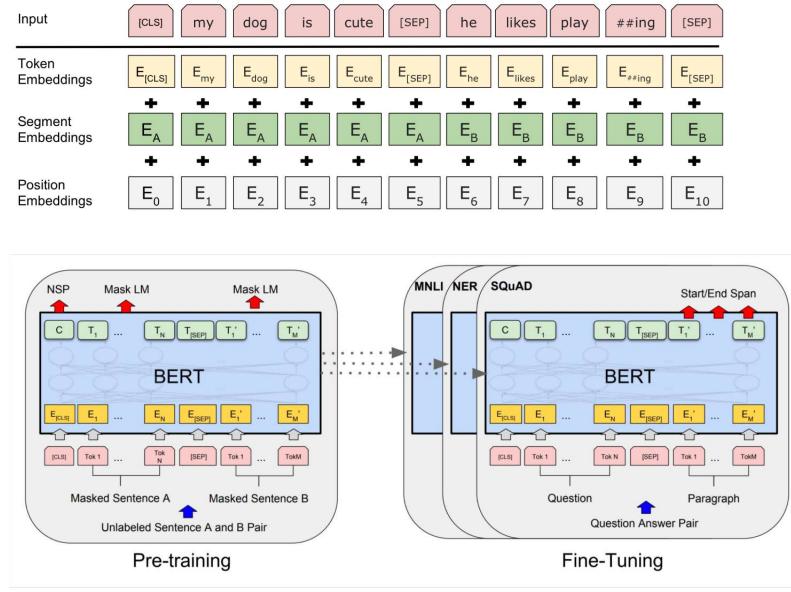
- Label: Acceptable
- Sentence: "The car honked down the road."
- Label: Unacceptable

- QQP: Quora Question Pairs (detect paraphrase questions)
- QNLI: natural language inference over question answering data
- SST-2: sentiment analysis
- CoLA: corpus of linguistic acceptability (detect whether sentences are grammatical.)
- STS-B: semantic textual similarity
- MRPC: microsoft paraphrase corpus
- RTE: a small natural language inference corpus

13.10.2. BERT Effect of pre-training task

13.11. Pre-training decoders GPT and GPT-2

- Contemporary to BERT, OpenAI introduced an alternative approach called Generative Pretrained Transformer (GPT) [Radford et al., 2018].
- The idea behind GPT is to train a large standard language model using the generative part of the Transformer, specifically the decoder.
- GPT is a Transformer decoder with 12 layers and 117 million parameters.
- It has 768-dimensional hidden states and 3072-dimensional feed-forward hidden layers.
- GPT utilizes byte-pair encoding with 40,000 merges to handle subword units.



- GPT was trained on BooksCorpus, which consists of over 7,000 unique books.
- OpenAI later introduced GPT-2, a larger version with 1.5 billion parameters, trained on even more data.
- GPT-2 has been shown to generate relatively convincing samples of natural language.

GPT-2 language model (cherry-picked) output Human provided prompt:

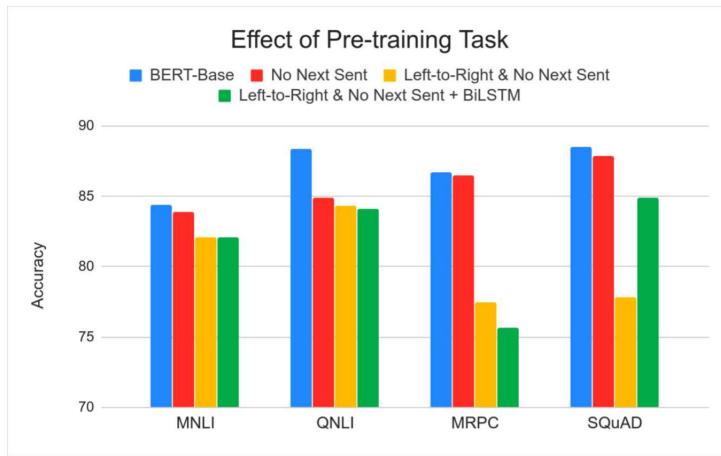
In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.
Model Completion:

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

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 _{BASE}	84.6/83.4	71.2	90.1	93.5	52.1	85.8	88.9	66.4	79.6
BERT _{LARGE}	86.7/85.9	72.1	91.1	94.9	60.5	86.5	89.3	70.1	81.9



13.12. What kinds of things does pretraining learn?

- Stanford University is located in _____, California. [Trivia]
- I put _____ fork down on the table. [syntax]
- The woman walked across the street, checking for traffic over _____ shoulder. [coreference]
- I went to the ocean to see the fish, turtles, seals, and _____. [lexical semantics/topic]
- Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was _____. [sentiment]
- Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the _____. [some reasoning – this is harder]
- I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21, _____. [some basic arithmetic; they don't learn the Fibonacci sequence]

13.13. Phase Change: GPT-3 (2020)

- GPT-3 is another Transformer-based Language Model (LM) that pushed the boundaries with nearly 200 billion parameters, making it the largest model at the time [Brown et al., 2020].
- It was trained on a massive corpus consisting of nearly 500 billion words.
- **In-context learning:** GPT-3 demonstrated the ability to solve various natural language processing (NLP) tasks using **zero-shot**, **one-shot** and **few-shot learning**.
- The key to this capability lies in the prompt or context provided to GPT-3.
- GPT-3 demonstrated the ability to solve various tasks without performing gradient updates to the base model.



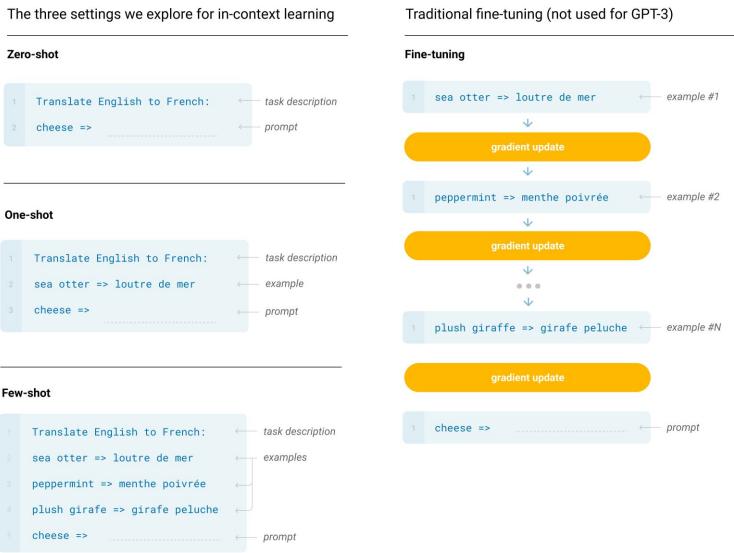
13.14. Zero-shot, One-shot, and Few-shot Learning with GPT-3

- **Zero-shot learning:** With zero-shot learning, GPT-3 can tackle tasks without any specific training. It achieves this by providing a prompt or instruction to guide its generation process. For example, by providing GPT-3 with a prompt like, “Translate this English sentence to French,” it can generate the translated sentence without any explicit training for translation tasks.
- **One-shot learning:** In one-shot learning, GPT-3 can perform a task by adding a single input-output pair to the instruction.
- **Few-shot learning:** similar idea but providing a limited number input-output pairs after the instruction in the prompt.

GPT-3 Few-shot Learning Results

13.15. Chain-of-thought Prompting

- Chain-of-thought prompting is a simple mechanism for eliciting multi-step reasoning behavior in large language models.



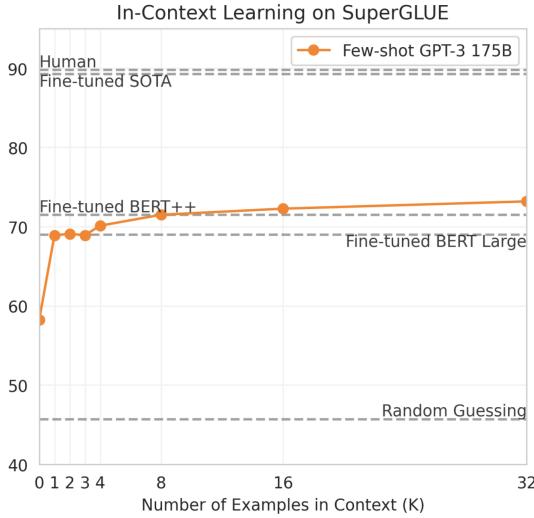
- Idea: augment each exemplar in few-shot prompting with a chain of thought for an associated answer [Wei et al., 2022]

13.16. Language Models as User Assistants (or Chatbots)

- Autoregressive Large Language Models are not aligned with user intent [Ouyang et al., 2022]
- Solution: align the language model with user intent via fine-tuning.

13.16.1. LaMDA: Language Models for Dialog Applications

- LaMDA is a language model developed by Google based on Transformer optimized for open domain dialog [Thoppilan et al., 2022].
- It has 137 billion parameters and is trained on 1.56 billion words.
- It is initially pre-trained in the same way as traditional language models (predicting words) with language models (predicting words) with a strong focus on dialog data.
- It is then fine-tuned to generate responses with respect to several other criteria.



Chain-of-thought prompting

Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ❌

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had $23 - 20 = 3$. They bought 6 more apples, so they have $3 + 6 = 9$. The answer is 9. ✓

- In order to fit LaMDA to all these criteria they worked with a large number of crowd-workers.
- These are people who manually labeled conversations from the pre-trained model.

LaMDA Optimization Criteria Quality

- Sensibleness: give meaningful answers.
- Specificity: avoid vague answers.
- Interestingness: give insightful, unexpected or witty answers.

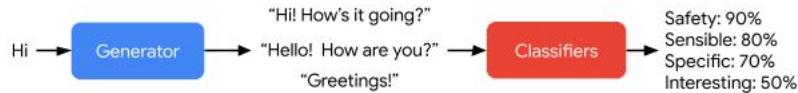
Safety

- Avoid violent language.

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.



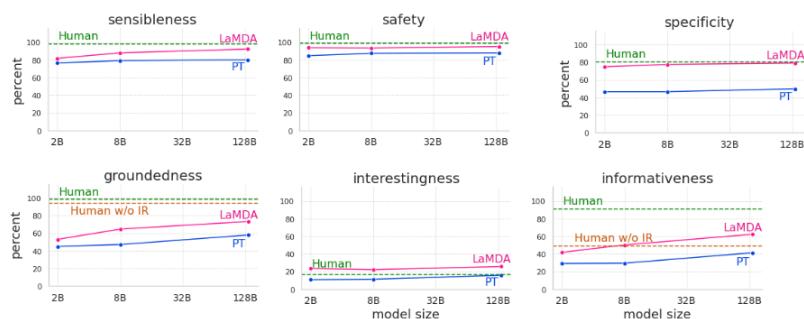
- Avoid hate speech.
- Avoid stereotyped speech.

Groundedness and Informativity

- Avoid giving answers not validated by external sources.
- Optimize the fraction of responses that can be validated in authoritative sources using search engines.

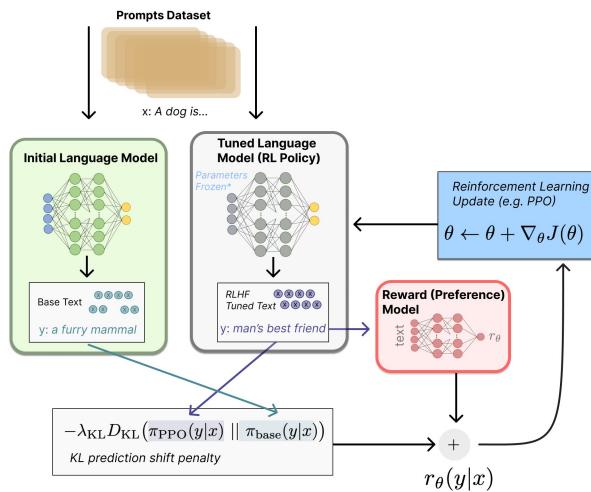
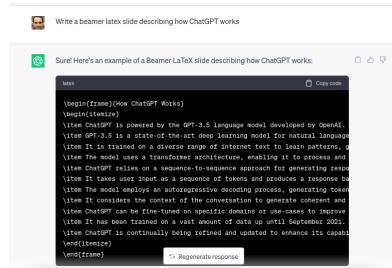
LaMDA Evaluation

- The system is compared with the original pre-trained PT model and human judgments.
- The evaluation is done by another group of people through questionnaires.



13.17. ChatGPT and RLHF

- Model similar to LaMDA launched by OpenAI at the end of 2022.
- It also uses Crowdsourcing to improve its responses, but its fine-tuning process uses Reinforcement Learning (RL), a different learning paradigm from supervised learning.
- In particular, it uses Reinforcement Learning from Human Feedback (RLHF) [Ouyang et al., 2022].
- It builds a preference model that assigns a score to a generated sentence and adjusts the language model accordingly.



Source: <https://huggingface.co/blog/rlhf>

13.18. GPT-4 (2023)

- Last LM of OpenAI [OpenAI, 2023], this time able to include images in the prompt.
- Still a Transformer LM.
- Able to pass exams in several disciplines being able to process the images of the questions.
- From ChatGPT onwards, companies have stopped making public all the details of the construction of their models.

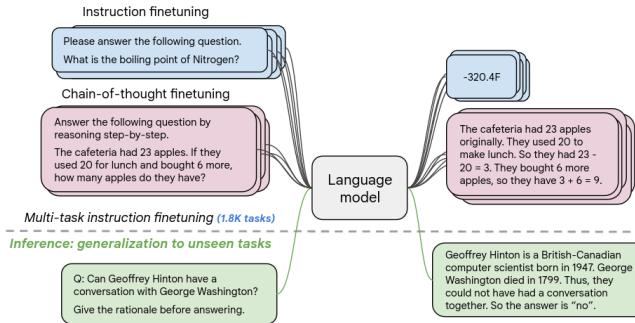
Example of GPT-4 visual input:	
User	What is funny about this image? Describe it panel by panel.
	
Source:	https://www.reddit.com/r/humor/comments/1absh5v/funny/
GPT-4	The image shows a package for a "Lightning Cable" adapter with three panels. Panel 1: A smartphone with a VGA connector (a large, blue, 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.

13.19. Instruction Fine-tuning

- A more efficient way to fine-tune Large Language Models is Instruction Fine-Tuning [Chung et al., 2022].
- Idea: collect examples of (instruction, output) pairs across many tasks and finetune an LM.
- Evaluate on unseen tasks.

13.20. Dangers of Large Language Models

The research community has raised concerns about several dangers associated with Large Language Models [Bender et al., 2021].



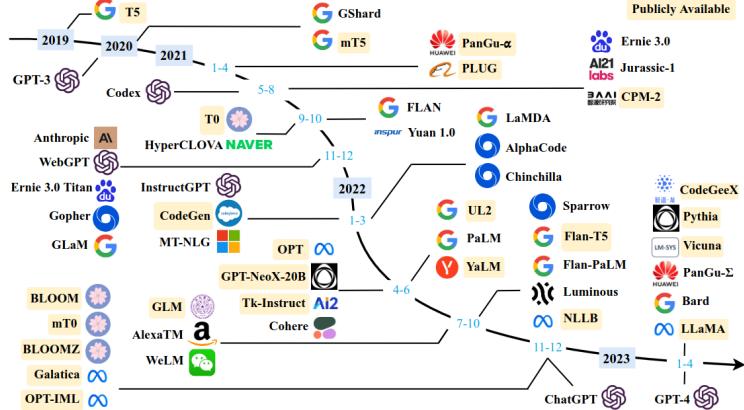
- **Hallucination:** Probabilistic language models can generate fabricated information lacking factual basis.
- **Fairness:** These models can perpetuate biases present in the training data, including toxic language, racism, and gender discrimination.
- **Copyright infringement:** Large language models may violate copyright laws by reproducing content without proper authorization.
- **Lack of transparency:** The complex nature of these models makes it difficult to interpret their predictions and understand the reasoning behind specific responses.
- **Monopolization:** The high costs of training these models create barriers for non-big-tech companies to compete.
- **High carbon footprint:** The energy-intensive training process of these models contributes to a significant carbon footprint.

13.21. Large Language Models Time-line

- As of today (2023), the development of new Large Language Models continues uninterrupted.
- A timeline of existing large language models (having a size larger than 10B) in recent years [Zhao et al., 2023].

13.22. Prompt Engineering

- Prompt engineering is a new discipline for developing and optimizing prompts to efficiently use language models (LMs).



13.23. Conclusions

- The growth in the size and power of language models has accelerated dramatically.
- It is very difficult to predict what they will do in the future.
- What can we predict with confidence?
- There will be an overload of generative models for multiple formats (text, code, image, video, virtual realities).
- There will be a plethora of agents/programs that act and make decisions by interacting with these models (medical appointments, investments, travel).

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*.
- [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*.
- [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 sub-gradient 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.
- [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.
- [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, 2008] Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd 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 *Icmi*, 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)*.
- [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.
- [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.
- [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.
- [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.