UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DSIIC

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Adaptación de skip-gramas a modelos conexionistas del lenguaje

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Adrián Palacios Corella

Directores: María José Castro Bleda
Francisco Zamora Martínez

15 de septiembre de 2014

A mis padres, que me siguen apoyando día tras día para que encuentre mi lugar en este mundo.

A mi hermano, porque me siento honrado de ser la meta visible durante su trayectoria.

A Hamdia, por sus ánimos en momentos en los que me sentía decaído.

A mis amigos, por ayudarme a mantener mi cordura.

A todos ellos, por quererme tal y como soy. Gracias.

A Francisco Zamora y Salvador España, por sus aportaciones a este trabajo y su paciencia a la hora de contestar los e-mails que este torpe estudiante envía a las tantas de la madrugada.

A María José Castro, por sus sugerencias sobre el trabajo y la ayuda recibida para conseguir becas y ofertas de empleo.

A los tres, por su apoyo incondicional durante mis estudios. Gracias.

Resumen

El objetivo del Modelado de Lenguaje es crear modelos de lenguaje capaces de estimar distribuciones de probabilidad sobre las palabras de un lenguaje natural. Estos modelos forman parte de los sistemas usados para resolver tareas como el Reconocimiento del Habla o Traducción Automática.

Nuestro interés se centra en los modelos conexionistas del lenguaje, los cuales han ido ganando popularidad a lo largo de los años. Este tipo de modelos, pese a su efectividad, continúan necesitando avances en distintas partes de su arquitectura para mejorar su eficiencia.

En este trabajo planteamos la adaptación de las técnicas de skip-gramas a los modelos conexionistas del lenguaje, que consisten en omitir palabras del contexto durante la fase de entrenamiento de los modelos. Estas técnicas podrían mejorar los resultados del modelo o reducir los costes temporales para su estimación.

Las técnicas y modelos requeridos para este estudio han sido implementados sobre la herramienta APRIL, utilizando los lenguajes C++ y Lua. El desarrollo del trabajo se validará sobre esta herramienta con experimentos de evaluación de modelos de lenguaje y tareas de traducción automática.

Palabras clave: procesamiento lenguaje natural, inteligencia artificial, modelos de lenguaje, redes neuronales, modelos conexionistas, n-gramas, skip-gramas, april, reconocimiento del habla, reconocimiento de escritura, traducción automática.

Abstract

The goal of Language Modeling is to create language models that represent a probability distribution over natural language words. These models are part of the systems used to solve Speech Recognition or Machine Translation tasks.

Our interest is centered on connectionist language models, which have gained popularity over time. These models, which are very effective, still need progress in some parts of its architecture to improve its efficiency.

In this work we propose the adaptation of skip-gram techniques to connectionist language models. These techniques allow the model to skip words from the context in the training phase. This may improve its results or lower the temporal cost for its estimation.

The required techniques and models for this study have been implemented in APRIL using the programming languages C++ and Lua. This development will be validated with experiments related to language model evaluation and machine translation tasks.

Índice general

Resumen III

Abstract V

Glosario XVII

1. Introducción 1

1.1. Motivación . . . 1
1.2. Fundamentos de los modelos de lenguaje . . . 2
1.2.1. Reconocimiento de formas . . . 3
1.2.2. Sistemas automáticos de reconocimiento/traducción . 5
1.2.3. Medidas de evaluación . . . 6
1.3. El toolkit APRIL . . . 9
1.4. Objetivos . . . 10
1.5. Estructura de la memoria . . . 10

2. Modelos de lenguaje 13

2.1. Modelos de lenguaje estadísticos . . . 13
2.2. Modelos de n-gramas . . . 14
2.2.1. Modelos de n-gramas por conteo . . . 15
2.2.2. El problema de la dispersión de datos . . . 15
2.2.3. Técnicas de suavizado . . . 17
2.2.4. Modelos de n-gramas basados en características . . . 21
2.3. Modelos de skip-gramas . . . 23
2.4. Modelos conexionistas de lenguaje (NN LMs) . . . 24
2.4.1. Codificación distribuida de las palabras . . . 25
2.4.2. Arquitectura de un NN LM . . . 26

3. Diseño orientado a consultas masivas 31

3.1. Partición de los modelos de lenguaje . . . 31
3.2. Estructuras de datos empleadas en los modelos de lenguaje . 32
3.2.1. La estructura KeyScoreBurdenTuple . . . 32
3.2.2. La estructura KeyScoreMultipleBurdenTuple . . . 33
3.2.3. Trie de palabras . . . 33
3.2.4. El formato Lira . . . 33

3.2.5. Tabla hash 34
3.2.6. La clase Token 34
3.2.7. La clase FunctionInterface 35
3.3. Jerarquía de clases 35
3.3.1. La clase LMModel 37
3.3.2. La clase LMInterface 37
3.3.3. La clase HistoryBasedLM 38
3.3.4. La clase HistoryBasedLMInterface 38
3.3.5. La clase HistoryBasedNgramLiraLM 40
3.3.6. La clase HistoryBasedNgramLiraLMInterface 40
3.3.7. La clase BunchHashedLM 40
3.3.8. La clase BunchHashedLMInterface 41
3.3.9. La clase FeatureBasedLM 41
3.3.10. La clase FeatureBasedLMInterface 41
3.3.11. La clase NNLM 42
3.3.12. La clase NNLMInterface 42
3.4. Scripts en Lua para el diseño de experimentos 42
3.4.1. Binding 43
3.4.2. Ejemplos 45

4. Adaptación de skip-gramas a modelos conexionistas del lenguaje 49
4.1. Modelos estadísticos de skip-gramas 49
4.2. Herramientas para la manipulación de datos 50
4.2.1. Dados 50
4.2.2. Tokens 54
4.2.3. La clase TokenVector 54
4.2.4. Filtros de datos 54
4.3. Adaptación de los algoritmos de entrenamiento y evaluación en NN LMs 55
4.3.1. Modificación probabilística de patrones 55
4.3.2. Generación de tablas de constantes softmax 56
4.3.3. Skipping NN LMs en jerarquías de modelos 58

5. Experimentación 61
5.1. Variación de la PPL en Skipping NN LMs 61
5.1.1. Corpus 61
5.1.2. Configuración experimental 62
5.1.3. Resultados experimentales 63
5.2. Emulación de modelos de orden inferior con Skipping NN LMs 65
5.2.1. Corpus 65
5.2.2. Configuración experimental 65
5.2.3. Resultados experimentales 66
5.3. Rescoring de listas nbest 67

5.3.1. Corpus 67
5.3.2. Configuración experimental 68
5.3.3. Resultados experimentales 68

6. Conclusiones y trabajos futuros 71
6.1. Conclusiones 71
6.2. Publicaciones 72
6.3. Ampliaciones futuras 73

A. El algoritmo Backpropagation 85
A.1. El problema del aprendizaje 85
A.2. Estructura de las redes neuronales 86
A.3. Algoritmos de aprendizaje para ANN 88
A.3.1. Métodos basados en el descenso por gradiente 88
A.3.2. Métodos de segundo orden 88
A.4. Introducción al algoritmo BP 89
A.4.1. El problema del aprendizaje 90
A.4.2. Derivada de la función de la red 91
A.4.3. Demostración formal del algoritmo BP 94
A.4.4. Aprendiendo con el BP 95
A.5. Forma matricial del BP 100
A.6. El algoritmo BP con momentum 102
A.7. El algoritmo BP con weight decay 102
A.8. Complejidad algorítmica del BP 103
A.9. Inicialización de pesos 103
A.10.Sobreentrenamiento 104

B. El formato Lira 105
B.1. Campos del formato Lira 106
B.2. Ejemplo 106

C. Artículo IberSPEECH 2014 109

Índice de figuras

1.1. Esquema general de un clasificador. La variable x representa los datos del entorno, mientras que $c(x)$ representa la etiqueta de clase que el clasificador asigna a x. 3
1.2. Esquema general de un sistema automático de reconocimiento. . . 6

2.1. Un ejemplo de la codificación distribuida de las palabras de un vocabulario en un espacio bidimensional. Palabras con la misma funcionalidad se concentran en determinadas áreas. 26
2.2. La arquitectura de un 4-grama NN LM durante la fase de entrenamiento, con $h_i = w_{i-1}, w_{i-2}, w_{i-3}$ [Zamora-Martínez (2012)]. 27
2.3. La arquitectura de un 4-grama NN LM durante la fase de evaluación, con $h_i = w_{i-1}, w_{i-2}, w_{i-3}$ [Zamora-Martínez (2012)]. 28

3.1. Ejemplo de un trie de palabras con un vector de tamaño máximo 8, creado a partir del conjunto de n-gramas {a, gc, ga, d}. Cada nodo contiene una tupla $\langle x, y, z \rangle$, en donde x es el índice del padre, y es el “timestamp” y z es la palabra de transición [Zamora-Martínez (2012)]. 34
3.2. Tablas hash para la agrupación de consultas. La tabla exterior relaciona claves de contexto con tablas que, a su vez, relacionan palabras con estructuras capaces de almacenar los resultados de múltiples consultas. 35
3.3. Diagramas de clases para los Modelos y las Interfaces. Cada clase derivada de la clase `LMModel` tiene asociada la clase derivada `LMInterface` correspondiente. 36

4.1. Un filtro de skips procesa un token de entrada con tres muestras. El token de salida contiene estas muestras modificadas pseudoaleatoriamente por el filtro, introduciendo la etiqueta ⟨NONE⟩ en algunas palabras de las muestras. 55
4.2. Esquema de una jerarquía de NN LMs de distinto orden. Los modelos de mayor orden delegan las consultas en modelos de menor orden cuando no pueden encontrar las constantes de normalización en sus tablas. 58

4.3. Esquema de una jerarquía de Skipping NN LMs del mismo orden. Cada Skipping NN LM maneja una tabla de constantes de normalización softmax diferente, generadas introduciendo en las muestras un número de skips por la izquierda variable. 60

5.1. PPL de los Skipping NN LMs medida sobre el conjunto de validación variando la máscara de skips, junto con la PPL de referencia de las NN LMs sin skips. Los ejes verticales de las cuatro gráficas muestran la PPL, y la máscara de skips aparece en los ejes horizontales usando notación decimal inversa. Las gráficas de abajo corresponden, de derecha a izquierda, a los modelos de 4-gramas, trigramas y bigramas. 63
5.2. Resultados de PPL para las jerarquías de modelos con NN LMs (normal) y Skipping NN LMs (con skips) para el conjunto de validación (izquierda) y test (derecha). 67

6.1. La arquitectura de un 4-grama NN LMs con caché durante la fase de entrenamiento. El vector $\bar{C}$ es una bolsa de palabras que incluye información acerca del contexto [Zamora-Martínez (2012)]. 75

A.1. Red neuronal como una caja negra. 85
A.2. Diagrama de redes hacia-delante, una capa a capa y otra general. . 88
A.3. Tres sigmoides (con c=1, c=2 y c=3). 90
A.4. Los dos lados de una unidad de proceso. 92
A.5. Sumatorio y función de activación separadas en dos unidades. . . . 92
A.6. Composición de funciones. 92
A.7. Adición de funciones. 93
A.8. Pesos en las aristas. 93
A.9. Traza del BP, para resolver el problema de la *xor*. Configuración inicial de la red. 96
A.10. Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 1/4. 97
A.11. Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 2/4. 97
A.12. Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 3/4. 97
A.13. Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 4/4. 98
A.14. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 1/6 (Cálculo de la derivada de la neurona de salida). . 98
A.15. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 2/6 (Ajuste de los pesos conectados a la neurona de salida). 98
A.16. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 3/6 (Cálculo de la derivada de la neurona oculta). . . 99
A.17. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 4/6 (Ajuste del peso umbral de la neurona oculta). . . 99

A.18. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 5/6 (Ajuste del primer peso de la neurona oculta). . . 99
A.19. Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 6/6 (Ajuste del segundo peso de la neurona oculta). . 99
A.20. Traza del BP, para resolver el problema de la *xor*. Configuración final de la red. 100

B.1. Ejemplo de un modelo de lenguaje de n-gramas representado en forma de autómata. Las transiciones con una línea discontinua indican un suavizado por back-off [Zamora-Martínez (2012)]. 105

Índice de tablas

2.1. Frecuencias de bigramas para un conjunto de palabras del vocabulario de nuestro interés. 16
2.2. Frecuencias de unigramas para un conjunto de palabras del vocabulario de nuestro interés. 16
2.3. Probabilidades de bigramas para un conjunto de palabras del vocabulario de nuestro interés. 16

4.1. Dado el orden de los n-gramas, la probabilidad se reparte de forma uniforme entre todas las posibles máscaras. 51
4.2. Dado el orden de los n-gramas y la probabilidad de no insertar ningún skip $p = 35\,\%$, la probabilidad $(1 - p)$ se reparte de forma uniforme entre todas las posibles máscaras con al menos un skip. . 51
4.3. Dado el orden de los n-gramas y la probabilidad de no insertar ningún skip $p = 50\,\%$, la probabilidad $(1 - p)$ se reparte de forma uniforme entre todas las posibles máscaras con al menos un skip. . 51
4.4. Dado el orden de los n-gramas, el número de skips que se introduce sigue estas distribuciones multinomiales. 52
4.5. Dado el orden de los n-gramas, la probabilidad se reparte de forma uniforme entre todas las posibles máscaras de skips viejos. Las máscaras de skips se representan en notación decimal inversa. . . . 53

5.1. Estadísticas del corpus LIBW. Se muestra el número de líneas y palabras para cada parte, además del porcentaje de palabras fuera de vocabulario (Out-Of-Vocabulary words). 62
5.2. La distribución multinomial empleada en el entrenamiento de Skipping NN LMs para la primera parte experimental. 62
5.3. Resultados de PPL para IAM-DB en validación (izquierda) y test (derecha). Para los modelos con skip, la PPL se calcula con 0 skips y combinaciones de skips que simulan un modelo de orden inferior. 64
5.4. Estadísticas de la parte en inglés del corpus News. Se muestra el número de líneas y palabras para cada parte. 65
5.5. La distribución uniforme empleada en el entrenamiento de Skipping NN LMs para la segunda parte experimental. 66
5.6. Resultados de PPL para las jerarquías de modelos con NN LMs (normal) y Skipping NN LMs (con skips) para los conjuntos de validación (izquierda) y test (derecha). 67

5.7. Estadísticas de la parte bilingüe del corpus News. Se muestra el número de líneas y palabras para cada parte. 68
5.8. Resultados de BLEU y TER con los conjuntos de los años 2008, 2009 y 2010 del corpus News para los dos tipos de jerarquía de modelos (**N**ormal y **S**kipping) con diferente orden máximo de modelo en la jerarquía. 69
5.9. Líneas base establecidas por los decodificadores Moses y APRIL-PB para los corpus del año 2009 y 2010. 69

A.1. Cálculo realizado por una neurona. 86
A.2. Función a aprender en el problema de la *xor*. 96

Glosario

ANN Redes Neuronales Artificiales o “Artificial Neural Nets”

API Interfaz de Programación de Aplicaciones o “Application Programming Interface”

BLEU “Bilingual Evaluation Understudy”

BP Algoritmo de Retropropagación del Error o “Backpropagation”

CPU Unidad Central de Procesamiento o “Central Processing Unit”

GPU Unidad de Procesamiento Gráfico o “Graphics Processing Unit”

IA Inteligencia Artificial o “Artificial Intelligence”

LC Lingüística Computacional o “Computational Linguistics”

ME Entropía Máxima o “Maximum Entropy”

MLE Estimación por Máxima Verosimilitud o “Maximum Likelihood Estimation”

MLP Perceptrón Multicapa o “Multilayer Perceptron”

MSE Error cuadrático medio o “Mean Square Error”

NN LM Modelo de Lenguaje de Redes Neuronales o “Neural Network Language Model”

PLN Procesamiento del Lenguaje Natural o “Natural Language Processing”

PPL Perplejidad o “Perplexity”

SIMD Single Instruction Multiple Data

TER “Translation Edit Rate”

Notación para los modelos de lenguaje

$C(x)$	Número de ocurrencias del evento x.
$\|\Omega\|$	Tamaño del vocabulario.
λ_i	Peso del modelo i-ésimo.
α	Factor de retroceso.
$Z(x)$	Término para la normalización.

Notación para el algoritmo Backpropagation

θ	Umbral de una neurona.
η	Factor de aprendizaje del BP (learning rate).
μ	Factor del "momentum" del BP.
α	Valor del Weight Decay.

Capítulo 1

Introducción

En este primer capítulo realizaremos una breve introducción al campo del Procesamiento del Lenguaje Natural con el propósito de entender los conceptos básicos y los motivos por los cuales se ha desarrollado este estudio. Se presentará también el toolkit APRIL, el software sobre el cual se han desarrollado las ideas expuestas en nuestra investigación y con el que se han llevado a cabo los experimentos que demuestran su validez. Seguidamente, se mostrarán los objetivos que este trabajo persigue y que serán revisados al final, con la finalidad de comprobar cuáles han sido alcanzados y cuáles no. Por último, se explicará cómo ha sido estructurada esta memoria, para orientar al lector sobre sus contenidos y facilitar su lectura.

1.1. Motivación

El Procesamiento del Lenguaje Natural o "Natural Language Processing" (PLN) es un campo de la Inteligencia Artificial o "Artificial Intelligence" (IA) y la Lingüística Computacional o "Computational Linguistics" (LC), cuyo interés se centra en la comunicación entre humanos y máquinas por medio del lenguaje natural. Las principales tareas del PLN, como por ejemplo el Reconocimiento del Habla, el Reconocimiento de Escritura o la Traducción Automática, pueden ser catalogadas como algunas de las tareas más complejas dentro del ámbito de la informática.

En la actualidad, los modelos estadísticos son los más empleados en tecnologías del lenguaje. Este tipo de modelos desplazaron a los modelos basados en reglas de los años 80 debido a sus mejores resultados, los cuales se deben, en parte, al aumento de la capacidad computacional de los ordenadores y la creciente disponibilidad de recursos digitales para estimar estos modelos. La aproximación estadística hace uso de la regla de Bayes para crear un modelo compuesto por:

- Un modelo que calcula la verosimilitud de la muestra en base a una hipótesis.
- Un modelo que calcula la probabilidad a priori de dicha hipótesis.

En los sistemas de reconocimiento automático del habla, por ejemplo, el primer modelo es denominado modelo acústico. El modelo acústico se encarga de asignar una puntuación a los datos observados respecto a una hipótesis, indicando cómo de bien se ajusta la señal de voz entrante a esa conjetura. El otro modelo se encarga de calcular la probabilidad a priori de la hipótesis considerada. A este segundo modelo lo llamamos modelo de lenguaje. Combinando los valores suministrados por ambos modelos, se obtiene una medida cuantitativa de las hipótesis consideradas por el sistema respecto a la muestra.

Sin duda alguna, los modelos de lenguaje estadísticos más éxitosos son los modelos de n-gramas [Jelinek (1997), Goodman (2001)]. Estos modelos requieren poco esfuerzo de cómputo para ser estimados y ofrecen unos resultados destacables. Por estas razones, no resulta extraño que hayan acaparado la atención de los expertos durante mucho tiempo, incidiendo la investigación en desarrollar métodos de suavizado. Estos métodos de suavizado sirven par reducir el impacto que la dispersión de datos tiene sobre los modelos de n-gramas.

Sin embargo, existen otros tipos de modelos que no sufren por la dispersión de datos y que han sido ignorados hasta el momento [Huang et al. (2012), Arisoy et al. (2012)]. Este es el caso de los modelos de lenguaje basados en redes neuronales (NN LMs). Estos modelos son capaces de mejorar los resultados obtenidos por los modelos de n-gramas [Schwenk and Gauvain (2002), Bengio et al. (2003), Castro and Prat (2003), Schwenk et al. (2006), Schwenk and Koehn (2008)], aunque al estar basados en redes neuronales, la construcción de los modelos tiene un coste espacial y temporal mayor. La abundancia de dispositivos con procesadores multinúcleo y arquitecturas SIMD ha ayudado a reducir estos costes considerablemente. Otro factor decisivo para la viabilidad de estos modelos ha sido el desarrollo de técnicas para optimizar su entrenamiento y evaluación [Schwenk et al. (2006), Emami and Mangu (2007), Park et al. (2010)], además de mejorar sus resultados.

La complejidad de los NN LMs es superior a la de muchos modelos, por lo que es posible diseñar mejoras en numerosas partes de su estructura. Es por ello que, a pesar del incremento de empleo de estos modelos en los sistemas de reconocimiento, todavía queda mucho por explorar. Esto nos motiva para desarrollar nuevas técnicas que permitan mejorar los resultados cualitativos de los NN LMs. Nuestro principal objetivo será adaptar las técnicas de skip-gramas para su uso en los NN LMs. Estas técnicas fueron empleadas originalmente para combatir la dispersión de datos en los modelos de n-gramas. La adaptación de las técnicas de skip-gramas se realizará sobre los Fast NN LMs [Zamora et al. (2009)], unos modelos que ya incorporan otras técnicas para mejorar su eficiencia y efectividad.

1.2. Fundamentos de los modelos de lenguaje

Los modelos de lenguaje se estudian dentro del campo del PLN, un campo derivado de la IA y la LC. En esta sección estudiaremos sus fundamentos en relación al campo de la IA, desde su planteamiento como un sistema de clasificación hasta su papel en un sistema automático de reconocimiento/traducción.

1.2.1. Reconocimiento de formas

La IA es un campo que se bifurca en dos ramas:

- La IA clásica, basada en la lógica y centrada en desarrollar sistemas inteligentes que apliquen técnicas de aprendizaje deductivas. En estos sistemas, el conocimiento se obtiene a partir de un experto.
- El reconocimiento de formas, basada en la teoría de decisión estadística y centrada en desarrollar sistemas inteligentes que apliquen técnicas de aprendizaje inductivas. En estos sistemas, el conocimiento se obtiene a partir de los ejemplos.

El objetivo del reconocimiento de formas es desarrollar sistemas capaces de reconocer su entorno a partir de datos obtenidos mediante los sensores apropiados. Estos sensores pueden ser sensores ópticos, acústicos, termales, etc. En este contexto, reconocer significa clasificar como una de las C clases o categorías posibles con una probabilidad de error mínima.

Un sistema clasificador está compuesto por tres módulos:

- El **módulo de preprocesado**, que se encarga de recoger los datos del entorno y aplicar los filtros apropiados antes de transmitir estos datos a...
- El **módulo de extracción de características**, que se encarga de calcular un vector de características a partir de los datos preprocesados y pasar este vector a...
- El **módulo de clasificación**, que se encarga de aplicar un algoritmo que determine la clase de los datos de la muestra a partir del vector de características que lo representa.

En la figura 1.1 podemos observar el esquema de un sistema clasificador como el descrito.

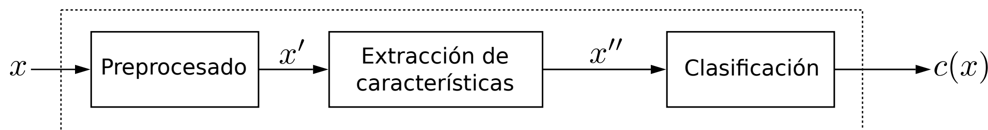

Figura 1.1: Esquema general de un clasificador. La variable x representa los datos del entorno, mientras que $c(x)$ representa la etiqueta de clase que el clasificador asigna a x.

Las técnicas que se aplican en los módulos de preprocesado y extracción de características suelen ser dependientes de la tarea. En cambio, la técnica que se aplica en el módulo de clasificación suele adaptarse a la técnica de clasificación óptima, el clasificador de Bayes:

$$c^*(x) = \underset{c=1,\ldots,C}{\arg\max}\; P(c \mid x) \tag{1.1}$$

Este clasificador selecciona aquella clase cuya probabilidad de ser la clase de la muestra x es la más alta de entre todas. La probabilidad de error al usar este clasificador para la muestra x es:

$$P(\text{error} \mid x) = 1 - \max_{c=1,\ldots,C} P(c \mid x) \tag{1.2}$$

Es decir, la probabilidad de error del clasificador de Bayes (error de Bayes) es igual a la probabilidad de que la muestra x sea de otra clase.

El enfoque tradicional consiste en reescribir la fórmula del clasificador de Bayes para expresarla en términos de probabilidades a priori y probabilidades condicionales [Duda et al. (1999)]. Partiendo de la fórmula del clasificador de Bayes:

$$c^*(x) = \underset{c=1,\ldots,C}{\arg\max}\; P(c \mid x) \tag{1.3}$$

La regla de Bayes expresa que:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{1.4}$$

Aplicando la regla de 1.4 a la fórmula 1.3, obtenemos:

$$c^*(x) = \underset{c=1,\ldots,C}{\arg\max}\; \frac{P(c)P(x \mid c)}{P(x)} \tag{1.5}$$

Si tenemos en cuenta que nuestro objetivo es maximizar el término completo, entonces podemos desechar el denominador para conseguir:

$$c^*(x) = \underset{c=1,\ldots,C}{\arg\max}\; P(c)P(x \mid c) \tag{1.6}$$

Existen dos razones por las cuales nos interesa expresar el clasificador de Bayes tal y como se muestra en la fórmula 1.6:

- La probabilidad a priori $P(c)$ puede obtenerse fácilmente a partir de las muestras. Típicamente, esta probabilidad se estima como $P(c) = \frac{N_c}{N}$, en donde N_c es el número de muestras de la clase c y N es el número total de muestras.
- La probabilidad condicional $P(x \mid c)$ puede obtenerse también a partir de las muestras, aunque el procedimiento para efectuar esta obtención suele ser más complicado. Existen distintos algoritmos para obtener esta distribución de probabilidad, como por ejemplo la Estimación por Máxima Verosimilitud o "Maximum Likelihood Estimation" (MLE), los cuales asignan una probabilidad a la muestra en función de la clase para la cual se esté ejecutando el algoritmo.

De esta forma, es posible crear un modelo completo a partir de un conjunto de muestras para aplicarlo en un sistema de clasificación.

En el siguiente apartado relacionaremos los términos de la ecuación 1.6 con los modelos de los sistemas automáticos de reconocimiento/traducción.

1.2.2. Sistemas automáticos de reconocimiento/traducción

La finalidad de un sistema automático de reconocimiento/traducción es obtener la transcripción de una secuencia de palabras de un vocabulario Ω a partir una secuencia de vectores de características que la representan. Si denotamos con $\bar{x} = x_1, x_2, \ldots, x_{|\bar{x}|}$ a la secuencia de longitud $|\bar{x}|$ de vectores de características de entrada al sistema, y con $\bar{y} = y_1, y_2, \ldots, y_{|\bar{y}|}$ a la secuencia de palabras, de longitud $|\bar{y}|$, obtenidas como salida del sistema, entonces podemos sustituir las variables de la ecuación 1.3 para adaptar el clasificador de Bayes a un sistema de reconocimiento/traducción:

$$\hat{\bar{y}} = \arg\max_{\bar{y} \in \Omega^+} P(\bar{y} \mid \bar{x}) \tag{1.7}$$

Y de forma equivalente, podemos aplicar la igualdad 1.4 a esta ecuación para obtener:

$$\hat{\bar{y}} = \arg\max_{\bar{y} \in \Omega^+} P(\bar{x} \mid \bar{y}) P(\bar{y}) \tag{1.8}$$

Después de aplicar la regla de Bayes, nos encontramos de nuevo con los dos modelos que comentábamos al principio de este capítulo:

- El modelo que indica la verosimilitud de la muestra: $P(\bar{x} \mid \bar{y})$.
- El modelo de lenguaje, que indica la probabilidad a priori de la hipótesis: $P(\bar{y})$.

Estos modelos se obtienen en una primera **fase de entrenamiento** supervisado, en la cual el sistema recibe un corpus etiquetado, de forma que se conoce la salida para cada entrada del corpus. A partir de este corpus, se estiman los modelos necesarios, que posteriormente se emplearán durante el proceso de búsqueda para obtener la respuesta más probable a muestras cuya salida sea desconocida.

Los modelos estimados en la anterior fase se emplean en la **fase de evaluación**, en la cual el sistema recibe una muestra y produce la salida más probable. Si las muestras están etiquetadas, entonces es posible medir el error cometido por el sistema.

La figura 1.2 muestra el esquema general de un sistema automático de reconocimiento/traducción. En ella, pueden distinguirse claramente ambas fases, y la relación de cada componente con los módulos. La complejidad del sistema depende de la tarea que se está afrontando, por lo que algunos sistemas más sofisticados pueden incluir módulos adicionales.

Tipos de sistemas

Las características de un sistema automático de reconocimiento/traducción dependen de la tarea que se esté abordando. En este trabajo nos centramos en dos tipos de tareas:

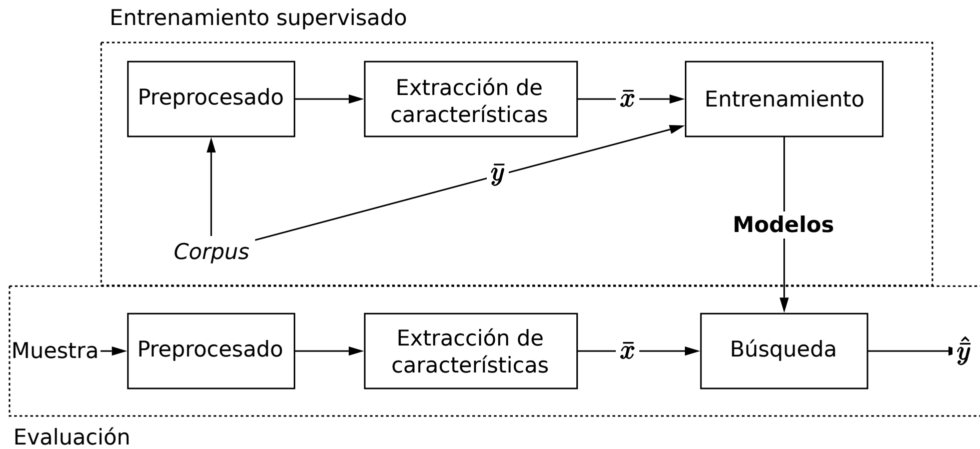

Figura 1.2: Esquema general de un sistema automático de reconocimiento.

- Reconocimiento de secuencias: La entrada es una secuencia de vectores de características extraídos a partir de una secuencia de datos, y la salida es una secuencia de palabras que son la transcripción de la secuencia de entrada. La longitud de las transcripciones siempre es menor o igual a la longitud de los vectores de características, es decir, $|\bar{y}| \leq |\bar{x}|$. El proceso de decodificación es monótono y puede diferenciarse según el problema que trata de resolver, lo cual define las propiedades de las secuencias de entrada:
  - Reconocimiento automático del habla: La entrada es una secuencia de vectores de características extraídos a partir de una señal de voz.
  - Reconocimiento automático de escritura on-line: La entrada es una secuencia de vectores de características extraídos a partir de la información sobre la posición y orientación de un lápiz electrónico respecto a una tableta.
  - Reconocimiento automático de escritura off-line: La entrada es una secuencia de vectores de características extraídos a partir de imágenes de texto a las cuales se les ha aplicado algún preproceso.
- Traducción automática: La entrada es una secuencia de palabras y la salida es otra secuencia de palabras en un idioma diferente, que es la traducción a ese idioma de la secuencia de palabras de la entrada. La longitud de ambas secuencias de palabras no tiene ningún tipo de restricción entre ellas. Al contrario que en los anteriores sistemas, el proceso de decodificación no es monótono, ya que los alineamientos de las secuencias de entrada y salida no están forzados.

1.2.3. Medidas de evaluación

La evaluación de un modelo de lenguaje dentro de un sistema automático de reconocimiento/traducción puede realizarse de dos formas:

- Calculando la perplejidad de los modelos de lenguaje estimados sobre un corpus.
- Calculando el error cometido por el sistema completo de reconocimiento/-traducción al usar los modelos de lenguaje estimados.

Ambas medidas se correlacionan entre ellas, de forma que reducir la perplejidad mediante el uso de un determinado modelo de lenguaje también reduce el error cometido por el sistema de reconocimiento/traducción [Klakow and Peters (2002)]. En este trabajo, al centrar nuestro interés en los modelos de lenguaje, usaremos la perplejidad como principal medida. En la última fase experimental incorporaremos los modelos de lenguaje a un sistema de traducción, por lo que también usaremos las medidas BLEU y TER para evaluar estos modelos.

Perplejidad

La Perplejidad o "Perplexity" (PPL) [Brown et al. (1992)] es una medida usada en la teoría de la información para evaluar la capacidad de predicción de una distribución de probabilidad sobre una muestra o un conjunto de muestras. En el ámbito que nos ocupa, los modelos de lenguaje son una distribución de probabilidad sobre sentencias o textos completos.

La PPL se define como la exponencial de la entropía normalizada por el número de palabras. Dado un conjunto de referencia $\mathcal{R}$, el modelo de lenguaje nos permite calcular la probabilidad de la frase $\bar{r}^{(i)}$, $P(\bar{r}^{(i)})$ para todo $i = 1, 2, \ldots, K$. La entropía del modelo de lenguaje sobre el conjunto de referencia puede calcularse como:

$$H(\mathcal{R}) = -\sum_{i=1}^{K} \log\ P(\bar{r}^{(i)}) \tag{1.9}$$

Nótese que en la fórmula 1.9 descartamos el término $P(\bar{r}^{(i)})$ que acompaña al término de la log probabilidad en la fórmula clásica de la entropía. Esto puede hacerse si consideramos que el lenguaje natural es una fuente ergódica.

Sea $L = \sum_{i=1}^{K} \left(\|\bar{r}^{(i)}\| + 1 \right)$ el número total de palabras del conjunto de referencia (sumamos 1 para tener en cuenta el fin de frase), entonces podemos definir la perplejidad como:

$$PPL = \exp\ \left(\frac{H(\mathcal{R})}{L} \right) \tag{1.10}$$

La PPL es una medida intuitiva que puede interpretarse como la media del número de elecciones disponibles a la hora de predecir cada palabra del conjunto de referencia. De forma equivalente, el valor de PPL representa el número de palabra que debería tener un modelo de lenguaje con una distribución uniforme para obtener el mismo resultado con el corpus de referencia.

El siguiente ejemplo nos ayudará a clarificar el concepto de PPL. Supongamos un lenguaje cuyas frases están compuestas por dígitos decimales aleatorios: $0, 1, \ldots, 8, 9$. Como no disponemos de información a priori, al tratar de predecir

el siguiente dígito escogeremos uno al azar de entre las 10 posibilidades. La PPL de este modelo de dígitos es, por tanto, 10.

Se dice que un modelo de lenguaje es mejor que otro si se consigue reducir la perplejidad para un mismo conjunto de frases, ya que es capaz de predecir mejor las palabras del conjunto. En el ejemplo anterior usábamos un modelo de unigramas para predecir el siguiente dígito, aunque cualquier modelo que usara información sobre la distribución de dígitos (como por ejemplo, que el número 5 nunca aparece detrás del número 3) sería capaz de dar mejores predicciones, reduciendo así la PPL del modelo de dígitos sobre el conjunto.

BLEU

El "Bilingual Evaluation Understudy" (BLEU) es una medida introducida por [Papineni et al. (2002)] y considerada la medida más extendida en traducción automática. El BLEU es una media geométrica del número de n-gramas (secuencias de n palabras) que comparten la referencia y la transcripción, para valores de $n = 1, 2, 3, 4$. Dado que es una media geométrica, si el resultado de una de las cuentas es 0, entonces el resultado total de la medida es 0, por lo que no puede aplicarse a nivel de frase.

Las cuentas se estiman sobre el conjunto entero a evaluar. Sea $\bar{w} = w_1, w_2, \ldots, w_n \in \Omega^n$ un n-grama cualquiera, y una cadena $\bar{r} = r_1, r_2, \ldots, r_{|\bar{r}|}$, el conteo de apariciones del n-grama $\bar{w}$ en la secuencia $\bar{r}$ se define como:

$$c_n(\bar{w}, \bar{r}) = \sum_{j=1}^{|\bar{r}|-n+1} \delta(\bar{w}, (r_j, r_{j+1}, \ldots, r_{j+n-1})) \tag{1.11}$$

Sean los conjuntos $\mathcal{Y}$ de transcripciones dadas por el sistema de traducción, y $\mathcal{R}$ los conjuntos de referencias, con K frases cada uno, y sea

$$S\left(\left\{\bar{z}^{(1)}, \ldots, \bar{z}^{(K)}\right\}\right) = \sum_{i=1}^{K} |\bar{z}^{(i)}|, \tag{1.12}$$

la suma de las tallas de todas las frases de un conjunto dado, el BLEU se define como:

$$BLEU = BP \cdot \sqrt[4]{\prod_{n=1}^{4} \sum_{i=1}^{K} \sum_{\bar{w} \in \Omega^n} \text{mín}(c_n(\bar{w}, \bar{r}^{(i)}), c_n(\bar{w}, \bar{y}^{(i)})))} \tag{1.13}$$

y BP se define como:

$$BP = \begin{cases} \exp(1 - \dfrac{S(\mathcal{R})}{S(\mathcal{Y})}) & \text{sii } S(\mathcal{R}) < S(\mathcal{Y}) \\ 1 & \text{sii } S(\mathcal{R}) < S(\mathcal{Y}) \end{cases} \tag{1.14}$$

En resumen, el BLEU es la media geométrica del número de n-gramas que comparten la referencia y salida del sistema, de forma que permite el reordenamiento de secuencias enteras de palabras.

TER

El "Translation Edit Rate" (TER) [Snover et al. (2006)] es una medida que permite introducir reordenamientos entre las palabras a costa de añadir una penalización por cada uno de los reordenamientos. El TER es una extensión de la distancia de Levenshtein, que alinea de forma óptima las palabras entre la salida del sistema de traducción y la referencia, realizando operaciones de inserción (I), borrado (B), sustitución simple (S) y distorsión (D). La distancia de Levenshtein puede calcularse mediante un algoritmo de programación dinámica como:

$$TER = \frac{I + B + S + D}{|\bar{r}^{(i)}|} \cdot 100 \tag{1.15}$$

$$T(j,m,v,k) = \begin{cases} 0 & j = 0 \land m = 0 \\ T(j-1,m,v,k) + p_i & j \neq 0 \land m = 0 \\ \text{mín}_{m' \in v} \left\{ \begin{array}{l} T(j-1,m,v,k) + p_i \\ T(j,m-1,v-m',m') + p_b + d \\ T(j-1,m-1,v-m',m') + s + d \end{array} \right\} & j > 0 \land m > 0 \end{cases} \tag{1.16}$$

de forma que los valores d y s se definen como:

$$d = p_d \cdot (abs(m' - k) - 1) \tag{1.17}$$

$$s = p_s \cdot (1 - \delta(\bar{y}_j^{(i)}, \bar{r}_{m'}^{(i)})) \tag{1.18}$$

siendo $\bar{r}^{(i)}$ la referencia i-ésima del conjunto $\mathcal{R}$, $\bar{y}^{(i)}$ la transcripción i-ésima del conjunto $\mathcal{Y}$, y D el número de distorsiones. Los costes de inserción p_i, borrado p_b, sustitución p_s y distorsión p_d, son iguales a los valores por defecto en la implementación de [Snover et al. (2006)]. La implementación utilizada en este trabajo para el cálculo del TER incluye heurísticos y búsqueda en haz para reducir el espacio de búsqueda, que de otra forma sería demasiado grande para calcular el TER de forma eficiente.

1.3. El toolkit APRIL

La implementación de las estructuras y algoritmos necesarios para el desarrollo de este trabajo se realizará sobre APRIL [Zamora-Martínez et al. (2013)], un proyecto de software desarrollado por nuestro grupo de investigación [Zamora-Martínez (2005); España et al. (2007)].

APRIL (acrónimo de *A Pattern Recognizer In Lua*) es un *toolkit* que reúne un conjunto de utilidades para diversas aplicaciones del reconocimiento de formas, tales como el reconocimiento de secuencias (voz, escritura), la traducción automática, *parsing*, etc. Los lenguajes de programación empleados en el desarrollo de APRIL son Lua y C++. El código de las clases es escrito en el lenguaje C++. Luego se crea un *binding* de ese código a Lua, de forma que puedan llamarse a los métodos de las clases escritas en C++ desde código Lua. Esto facilita el diseño de

experimentos, descritos mediante *scripts* en Lua, desde los cuales pueden usarse las clases C++.

Desde sus inicios, APRIL ha sido utilizado satisfactoriamente en la ejecución de experimentos para tareas de limpieza de imágenes [Hidalgo et al. (2005); Zamora et al. (2007)], normalización de escritura contínua [España et al. (2011)] y modelado de lenguaje [Zamora et al. (2009)], entre otras. En la actualidad, APRIL es un software de código abierto cuyo repositorio se encuentra en GitHub [Zamora-Martínez et al. (2013)]. Este repositorio cuenta con un número moderado de usuarios y sus distintas ramas son actualizadas casi a diario, por lo que APRIL puede ser considerado un proyecto muy activo.

1.4. Objetivos

En este trabajo pretendemos adaptar las técnicas de skip-gramas a los modelos conexionistas del lenguaje para estudiar cómo pueden beneficiarse los NN LMs a partir de estas técnicas. Para realizar este estudio, es necesario implementar una herramienta que nos permita efectuar los experimentos previstos.

Los objetivos de este proyecto pueden dividirse en dos grandes bloques: Aquellos relacionados con el diseño de las herramientas para investigar, y aquellos relacionados con los resultados de la investigación en sí. Específicamente, estos objetivos son:

- Elaborar un diseño de la herramienta que sea eficiente, tanto espacial como temporalmente.
- Confeccionar un diseño que permita la adición y extensión de funcionalidades de una forma sencilla.
- Estudiar el efecto que el entrenamiento con skip-gramas tiene sobre la PPL obtenida en los conjuntos de evaluación.
- Considerar qué otros beneficios pueden tener las técnicas de skip-gramas en los NN LMs.

En el capítulo 6 daremos un repaso a esta lista con el fin de constatar qué objetivos hemos cumplido y en qué medida lo hemos hecho.

1.5. Estructura de la memoria

En este primer capítulo nos hemos adentrado al campo del Procesamiento del Lenguaje Natural, introduciendo aquellos conceptos que van a aparecer a lo largo de este trabajo. En el siguiente capítulo nos meteremos de lleno en los modelos de lenguaje. Mostraremos la problemática asociada al uso de determinados modelos de lenguaje y revisaremos las técnicas propuestas para resolver esos problemas. En el tercer capítulo se exhibe el diseño del paquete para modelado del lenguaje en APRIL. El cuarto capítulo enseña nuestra proposición de forma detallada. Explicaremos el razonamiento teórico detrás de esta técnica y repasaremos los

detalles de su implementación. En el quinto capítulo observaremos los resultados obtenidos en los experimentos efectuados para estudiar el efecto de las técnicas de skip-gramas en los NN LMs. En el sexto capítulo exponemos las conclusiones que se extraen a partir de los resultados experimentales. Haremos un repaso de qué objetivos se han alcanzado y cuáles no, y finalizaremos la memoria planteando una serie de trabajos futuros.

Al final de la memoria podemos encontrar tres apéndices. El apéndice A cuenta el algoritmo de retropropagación para redes neuronales generales detalladamente. El apéndice B incluye información sobre el formato Lira, un formato para el almacenamiento de modelos de lenguajes como autómatas finitos estocásticos. El apéndice C contiene el artículo que se envió al congreso IberSPEECH 2014. En este artículo se describen de forma breve los Skipping NN LMs y se exponen los resultados de nuestros primeros experimentos con estos modelos.

Capítulo 2

Modelos de lenguaje

Un modelo de lenguaje es un conjunto de mecanismos que define la estructura de un determinado lenguaje y restringe las secuencias de unidades lingüísticas permitidas en él.

Existen una gran variedad de técnicas para crear modelos de lenguaje, aunque nuestro interés se centra en aquellas técnicas estadísticas que permiten estimar los modelos de lenguaje automáticamente a partir de ejemplos.

En este capítulo repasaremos los conceptos fundamentales de los modelos de lenguaje, centrándonos en los modelos de n-gramas, los skip-gramas y los modelos conexionistas.

2.1. Modelos de lenguaje estadísticos

El objetivo de un modelo de lenguaje es calcular la probabilidad de una sentencia S compuesta por n palabras [Jurafsky (2012)]:

$$P(S) = P(w_1, w_2, w_3, \ldots, w_n) \tag{2.1}$$

Una tarea de nuestro interés, relacionada con este objetivo, es calcular la probabilidad de la próxima palabra dadas las $n-1$ palabras anteriores:

$$P(w_n \mid w_1, w_2, \ldots, w_{n-1}) \tag{2.2}$$

Cualquier modelo capaz de calcular alguna de las dos anteriores probabilidades es llamado un modelo de lenguaje.

Para calcular la probabilidad conjunta de la ecuación 2.1 podemos aplicar la regla de la cadena, convirtiéndola así en un producto de probabilidades condicionales:

$$P(w_1, w_2, w_3, \ldots, w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2) \ldots P(w_n \mid w_1, \ldots, w_{n-1}) \tag{2.3}$$

El término anterior puede expresarse de forma más compacta como:

$$P(w_1, w_2, w_3, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1, \ldots, w_{i-1}) \tag{2.4}$$

Una posibilidad para estimar los valores de estas probabilidades condicionales es contar el número de ocurrencias en el texto y normalizarlas adecuadamente. Por ejemplo, la probabilidad de la frase *"The little boy likes pizza"* se calcularía como:

$$\begin{aligned} P(The, little, boy, likes, pizza) = & P(The) \cdot P(little \mid The) \cdot \\ & \cdot P(boy \mid The, little) \cdot \\ & \cdot P(likes \mid The, little, boy) \cdot \\ & \cdot P(pizza \mid The, little, boy, likes) \end{aligned} \tag{2.5}$$

En donde $P(The)$ se estima a partir de la cantidad de veces que la palabra "The" es observada. $P(little \mid The)$ se estima a partir del número de veces que la palabra "little" aparece detrás de la palabra "The", $P(boy \mid The, little)$ se estima según las veces que la palabra "boy" aparezca detrás de las palabras "The little", etcétera.

El problema de esta aproximación es que el cálculo de estas probabilidades condicionales tiene un coste temporal elevado, y aunque este coste temporal fuese aceptable, seguiríamos sin disponer de suficientes datos como para estimar las probabilidades. Este cálculo puede ser simplificado asumiendo que la ocurrencia de una palabra en una oración depende únicamente de las k palabras previas (asunción de Markov):

$$P(w_1, w_2, w_3, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-k}, \ldots, w_{i-1}) \tag{2.6}$$

Teniendo en cuenta la misma simplificación, podemos reescribir la ecuación 2.2

$$P(w_n \mid w_1, w_2, \ldots, w_{n-1}) \approx P(w_n \mid w_{n-k}, \ldots, w_{n-1}) \tag{2.7}$$

De esta manera, la estimación y evaluación de un modelo de lenguaje se vuelven más sencillas. El cálculo de probabilidades condicionales puede efectuarse manejando una sola tabla, y la consulta de probabilidades se reduce a facilitar el valor de la casilla correspondiente a esa consulta.

Este tipo de modelos de lenguaje en donde se fija el número de palabras del contexto son los modelos de n-gramas [Jelinek (1997)], los cuales estudiaremos en el siguiente apartado.

2.2. Modelos de n-gramas

Los modelos de n-gramas son aquellos modelos de lenguaje que calculan la probabilidad de la próxima palabra en base a las $n-1$ palabras anteriores. Estos modelos son el resultado de aplicar la asunción de Markov a las probabilidades condicionales de cualquier palabra.

2.2.1. Modelos de n-gramas por conteo

Los modelos de n-gramas por conteo son el tipo de modelos de lenguaje más simple, aunque también pueden ser considerados los más exitosos. Esto es debido, en gran parte, al compromiso que un modelo de n-gramas por conteo ofrece entre eficiencia temporal y eficacia en sus resultados. Por ejemplo, es posible entrenar un modelo de lenguaje en unos pocos segundos y obtener unas buenas tasas de error o perplejidad.

Un modelo de lenguaje en el cual la probabilidad de una palabra viene dada por la frecuencia relativa de aparición de esa palabra es llamado un modelo de unigramas:

$$P(w_i \mid w_1, w_2, \dots, w_{i-1}) \approx P(w_i) \tag{2.8}$$

Un modelo de lenguaje en el cual la probabilidad de una palabra está condicionada únicamente por la anterior palabra es llamado un modelo de bigramas. Estas probabilidades se definen como:

$$P(w_i \mid w_1, w_2, \dots, w_{i-1}) \approx P(w_i \mid w_{i-1}) \tag{2.9}$$

El mismo proceso de estimación puede ser extendido a modelos que tengan en cuenta más palabras del contexto para formar modelos de trigramas, 4-gramas, 5-gramas.

Estos modelos puede ser estimados a partir de un corpus de texto contando la frecuencia de los n-gramas y aplicando una normalización. El modelo de bigramas definido en 2.9 puede definirse como:

$$P(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \tag{2.10}$$

En general, cualquier modelo de n-gramas puede estimarse como:

$$P(w_i \mid w_{i-n+1}, \dots, w_{i-1}) = \frac{C(w_{i-n+1}, \dots, w_{i-1}, w_i)}{C(w_{i-n+1}, \dots, w_{i-1})} \tag{2.11}$$

Este algoritmo para la estimación de modelos de n-gramas es conocido como la MLE. Los modelos estimados a partir de este método se denotan como $P_{MLE}(w_i \mid w_{i-n+1}, \dots, w_{i-1})$.

2.2.2. El problema de la dispersión de datos

Ahora vamos a analizar las dificultades que se presentan al estimar un modelo de n-gramas según lo explicado en el apartado anterior. La tabla 2.1 muestra las frecuencias de algunos bigramas del corpus Berkeley Restaurant Project (9222 frases):

Al normalizar estas frecuencias por los valores de los unigramas (que se muestran en la tabla 2.2), obtenemos las probabilidades de esos bigramas (véase tabla 2.3).

Podemos observar que el modelo asigna una probabilidad bastante alta a bigramas comunes de la lengua inglesa, como por ejemplo “want to” o “to eat”

Tabla 2.1: Frecuencias de bigramas para un conjunto de palabras del vocabulario de nuestro interés.

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Tabla 2.2: Frecuencias de unigramas para un conjunto de palabras del vocabulario de nuestro interés.

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

($P(to \mid want) = 0.66$), $P(eat \mid to) = 0.28$). Al mismo tiempo, el modelo asigna una probabilidad baja o nula a bigramas más extraños (habitualmente construcciones gramaticales incorrectas) como por ejemplo "want eat" o "i lunch" ($P(eat \mid want) = 0.0011$, $P(lunch \mid i) = 0$).

Por consiguiente, el modelo de lenguaje asignará unas probabilidades cercanas a las probabilidades reales de aparición. Por esta misma razón, el modelo de lenguaje asignará probabilidades nulas a bigramas poco frecuentes que no hayan aparecido en el conjunto de entrenamiento. Podemos intuir, echando un vistazo a la tabla 2.3, que es bastante probable que esto ocurra (la tabla completa es una matriz dispersa). Este es un comportamiento indeseable en un entorno real, ya que el modelo de lenguaje anula completamente la probabilidad del modelo acústico (el cual podía estar asignando un probabilidad muy alta a esa hipótesis).

Tabla 2.3: Probabilidades de bigramas para un conjunto de palabras del vocabulario de nuestro interés.

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Este problema es conocido como *La maldición de la dimensión* [Bellman and Corporation (1957)]. El lenguaje natural se compone de vocabularios muy extensos, por lo que es imposible conseguir un conjunto de entrenamiento en el cual los eventos sean observados con una probabilidad igual a su probabilidad real. Incluso si fuese posible conseguir ese conjunto de entrenamiento, sería demasiado costoso estimar un modelo de bigramas a partir de ese conjunto.

Una posible solución a este problema es utilizar técnicas de suavizado para estimar modelos de n-gramas por conteo [Chen and Goodman (1996)]. Estas técnicas sustraen una porción de la masa de probabilidad total del modelo para repartirla entre los eventos no vistos durante la fase de estimación.

2.2.3. Técnicas de suavizado

El propósito de las técnicas de suavizado es extraer una cantidad de probabilidad $P_{uniform}$ de la probabilidad P_{MLE} de los eventos observados ω en el conjunto de entrenamiento. Esta probabilidad se reparte entre todos los eventos no observados de forma uniforme:

$$P_{suavizado}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \begin{cases} P'_{MLE}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) & \text{si } C(w_{i-n+1}, \ldots, w_{i-1}, w_i) \neq 0 \\ P_{uniform}\dfrac{1}{|\Omega| - |\omega|} & \text{si } C(w_{i-n+1}, \ldots, w_{i-1}, w_i) = 0 \end{cases} \quad (2.12)$$

El modelo P'_{MLE} usado en la fórmula 2.12 es el modelo producido al extraer la masa de probabilidad $P_{uniform}$:

$$\sum P'_{MLE}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = 1 - P_{uniform} \quad (2.13)$$

Las técnicas de suavizado expuestas a continuación son fundamentalmente distintas formas de obtener los modelos P'_{MLE} y $P_{uniform}$ de la fórmula 2.12. La finalidad de todas estas técnicas es la misma: Evitar probabilidades nulas en los modelos de lenguaje, las cuales son debidas a los eventos no observados. Estas técnicas pueden catalogarse según su nivel de simplicidad o efectividad. Por lo general, cuanto más simple sea una técnica, menor será su efectividad, aunque las técnicas más complejas consiguen unos resultados parecidos.

Añade uno (Suavizado de Laplace)

El suavizado de Laplace es seguramente la técnica más simple e intuitiva de entre todas las técnicas de suavizado. Consiste en asumir que cada palabra del vocabulario se observa una vez más de lo que se observa en realidad, por lo que se elimina la posibilidad de encontrar n-gramas con probabilidad nula. El suavizado de Laplace puede formalizarse modificando la fórmula 2.11 para la estimación de n-gramas:

$$P(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \frac{C(w_{i-n+1}, \ldots, w_{i-1}, w_i) + 1}{C(w_{i-n+1}, \ldots, w_{i-1}) + |\Omega|} \quad (2.14)$$

Esta técnica, pese a resolver el problema de una forma sencilla, no puede aplicarse a tareas con un vocabulario amplio o modelos con un orden mayor que

2. Esto se debe a que la masa de probabilidad repartida entre todas las palabras es demasiado grande al trabajar con este tipo de vocabularios, por lo que las palabras que sí que se observan son tratadas de forma parecida a las palabras no observadas. No obstante, el suavizado de Laplace puede ser suficiente para tareas en donde el número de probabilidades nulas no es muy alto.

Interpolación lineal

La interpolación lineal (también conocida como interpolación simple o suavizado de Jelinek-Mercer) es una técnica que combina varios modelos de n-gramas [Jelinek (1980)]. Por ejemplo, podemos interpolar un modelo de trigramas, un modelo de bigramas y un modelo de unigramas con distintos pesos λ_i para formar un nuevo modelo:

$$\begin{aligned} P_{interpolado}(w_i \mid w_{i-2}, w_{i-1}) = & \lambda_1 P(w_i \mid w_{i-2}, w_{i-1}) + \\ & \lambda_2 P(w_i \mid w_{i-1}) + \\ & \lambda_3 P(w_i) \end{aligned} \tag{2.15}$$

Por supuesto, para que la interpolación sea lineal, se ha de cumplir que:

$$\sum_i \lambda_i = 1 \tag{2.16}$$

Estos pesos pueden ajustarse de forma manual o automática. No obstante, lo más recomendable que se ajusten de forma automática. La mayoría de toolkits para el modelado de lenguaje proporcionan *scripts* para calcular estos pesos, como por ejemplo el `compute_best_mix` de SRILM. Para realizar el ajuste de pesos automático, es necesario separar una parte de nuestro conjunto de muestras y formar un conjunto de validación. Este conjunto de validación es usado por el *script* para encontrar los pesos óptimos, los cuales serán fijados al emplear el modelo de lenguaje resultante con el conjunto de test.

Es bastante común añadir un modelo con la distribución uniforme $P_{uniform}(w_i) = \frac{1}{|\Omega|}$ con el objetivo de evitar probabilidades nulas.

Estimación de Good-Turing

La estimación de frecuencias Good-Turing trata de estimar la probabilidad de que ocurra un evento no observado a partir del número de eventos observados una sola vez.

La idea detrás de la estimación de frecuencias Good-Turing es que la probabilidad de los eventos observados está sobreestimada. Pongamos por ejemplo que una secuencia de palabras aparece una sola vez en nuestro conjunto de entrenamiento. Es entonces bastante probable que esta secuencia haya aparecido por casualidad y que la probabilidad real de que esta secuencia aparezca sea mucho menor. El descuento que se aplica en la fórmula 2.19 deja una masa de probabilidad que se reparte entre los eventos no vistos, de forma que se estima una probabilidad

más cercana a la probabilidad real, tanto para los eventos observados como los no observados.

Esta probabilidad se calcula en dos pasos:

1. Calcular el descuento para los eventos observados r veces.

2. Calcular la probabilidad de Good-Turing a partir del descuento obtenido en el anterior paso.

Antes de realizar el primer cálculo, tenemos que definir n_r como el número de n-gramas que ocurren r veces:

$$n_r = |\{w_{i-n+1}, \ldots, w_i \mid C(w_{i-n+1}, \ldots, w_i) = r\}| \tag{2.17}$$

n_0 se define como el número total de n-gramas. El descuento r^* puede calcularse a partir de n_r:

$$r^* = (r+1)\frac{n_{r+1}}{n_r} \tag{2.18}$$

Finalmente, la probabilidad de Good-Turing es igual a:

$$P^*_{GT} = \frac{r^*}{\sum_{r=1}^{\infty} r n_r} \tag{2.19}$$

Back-off de Katz

La técnica de "back-off" consiste en retroceder hacia modelos de un orden menor cuando el evento no ha sido observado [Katz (1987)]. La probabilidad que proporciona el modelo de menor orden es penalizada por un valor α. En caso de haber observado el evento, se aplica el descuento del estimador de Good-Turing. El "back-off" de Katz se formula como:

$$P_{bo}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \begin{cases} d_{w_{i-n+1},\ldots,w_i} \frac{C(w_{i-n+1},\ldots,w_{i-1},w_i)}{C(w_{i-n+1},\ldots,w_{i-1})} & \text{si } C(w_{i-n+1}, \ldots, w_i) > 0 \\ \alpha_{w_{i-n+1},\ldots,w_{i-1}} P_{bo}(w_i \mid w_{i-n+2}, \ldots, w_{i-1}) & \text{si } C(w_{i-n+1}, \ldots, w_i) = 0 \end{cases} \tag{2.20}$$

Para calcular α, lo mejor es calcular primero β, que es la masa de probabilidad restante después de aplicar el descuento:

$$\beta_{w_{i-n+1},\ldots,w_{i-1}} = 1 - \sum_{\{w_i : C(w_{i-n+1},\ldots,w_i) > 0\}} d_{w_{i-n+1},\ldots,w_i} \frac{C(w_{i-n+1}, \ldots, w_{i-1}, w_i)}{C(w_{i-n+1}, \ldots, w_{i-1})} \tag{2.21}$$

A partir de β podemos realizar el cálculo de α:

$$\alpha_{w_{i-n+1},\ldots,w_{i-1}} = \frac{\beta_{w_{i-n+1},\ldots,w_{i-1}}}{\sum_{\{w_i : C(w_{i-n+1},\ldots,w_i) = 0\}} P_{bo}(w_i \mid w_{i-n+2}, \ldots, w_{i-1})} \tag{2.22}$$

Descuento Witten-Bell

El descuento Witten-Bell es un método de suavizado parecido al estimador de frecuencias de Good-Turing, aunque en este caso se tiene en cuenta la diversidad de las palabras al calcular las probabilidades. Esta forma de calcular el descuento es razonable si consideramos que es más probable que aparezcan nuevos eventos a partir de algunos contextos que otros.

Este concepto puede ilustrarse mediante un ejemplo con bigramas. En un determinado corpus inglés, las palabras "spite" y "constant" aparecen 993 cada una. Detrás de "spite" sólo aparecen 9 palabras diferentes, siendo la palabra "of" la más frecuente (979 veces). Detrás de "constant" aparecen 415 palabras diferentes, siendo la palabra "and" la más frecuente. 286 de las 415 palabras que siguen a "constant" aparecen una única vez. El elevado número de secuencias "spite of" en el corpus se debe, lógicamente, a la construcción del inglés "in spite of". Por tanto, podemos afirmar que es más probable encontrar nuevos bigramas que empiecen por "constant" que por "spite".

Antes de formular el descuento Witten-Bell, definimos los siguientes símbolos:

- $T(w_{i-n+1}, \ldots, w_{i-1})$ es el número de palabras diferentes a la derecha de $w_{i-n+1}, \ldots, w_{i-1}$.
- $N(w_{i-n+1}, \ldots, w_{i-1})$ es el número total de palabras a la derecha de $w_{i-n+1}, \ldots, w_{i-1}$.
- $Z(w_{i-n+1}, \ldots, w_{i-1})$ es el número de n-gramas en el conjunto actual con $w_{i-n+1}, \ldots, w_{i-1}$ que no aparecen en el conjunto de entrenamiento.

El modelo suavizado devuelve una probabilidad a la que se le aplica un descuento si el evento ha sido visto con anterioridad. En caso contrario, se devuelve una cantidad relacionada con el número de palabras diferentes a la derecha del contexto. Este modelo puede formularse del siguiente modo (el contexto sobre el que operan los símbolos T, N y Z se omite por razones de espacio):

$$P_{WB}(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \begin{cases} \frac{C(w_{i-n+1}, \ldots, w_i)}{N+T} & \text{si } C(w_{i-n+1}, \ldots, w_i) > 0 \\ \frac{T}{Z(N+T)} & \text{si } C(w_{i-n+1}, \ldots, w_i) = 0 \end{cases} \tag{2.23}$$

Descuento absoluto

El descuento absoluto es una técnica de suavizado simple que consiste en aplicar un descuento fijo a la probabilidad de los eventos ya observados. La masa de probabilidad restante debida a este descuento puede repartirse sobre un modelo de menor orden.

El siguiente es un modelo de bigramas con descuento absoluto, en donde la probabilidad sobrante α se reparte sobre un modelo de unigramas en forma de constante de normalización:

$$P_{abs}(w_i \mid w_{i-1}) = \frac{\text{máx}\,(C(w_{i-1}, w_i) - d, 0)}{C(w_{i-1})} + \alpha P_{abs} \tag{2.24}$$

Suavizado de Kneser-Ney

El suavizado de Kneser-Ney es una extensión del descuento absoluto. De forma similar a como el descuento Witten-Bell tiene en cuenta la diversidad de las palabras, el descuento Kneser-Ney tiene en cuenta la diversidad del prefijo de los n-gramas.

El siguiente ejemplo nos ayudará a entender a qué nos referimos. En un determinado corpus inglés, la palabra "York" aparece un total de 477 veces. Sin embargo, la mayoría de esas veces sucede a la palabra "New" (473 veces). El modelo de unigramas le asignará una probabilidad alta, aunque es poco probable que la palabra "York" aparezca en un bigrama desconocido. Por tanto, el modelo de back-off de unigramas debería asignar a "York" una probabilidad baja.

El modelo de bigramas de Kneser-Ney define la probabilidad de continuación de una palabra como el número de palabras diferentes que preceden a esa palabra, normalizada por el número de palabras que preceden a las palabras del vocabulario:

$$P_{cont}(w_i) = \frac{|\{w_{i-1} : C(w_{i-1}, w_i) > 0\}|}{|\{(w_{j-1}, w_j) : C(w_{j-1}, w_j) > 0\}|} \tag{2.25}$$

Una palabra frecuente como "York" obtendrá una probabilidad de continuación baja debido a que prácticamente sólo ocurre en un único contexto ("New"). La masa de probabilidad sobrante λ a causa del descuento absoluto se calcula:

$$\lambda(w_{i-1}) = \frac{d}{C(w_{i-1})} |\{w : C(w_{i-1}, w) > 0\}| \tag{2.26}$$

Esta masa se reparte sobre la probabilidad de continuación:

$$P_{KN}(w_i \mid w_{i-n+1}^{i-1}) = \frac{\text{máx}\,(C(w_{i-n+1}^{i}) - d, 0)}{C(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1}) P_{cont}(w_i) \tag{2.27}$$

El algoritmo Kneser-Ney modificado aplica un descuento diferente según la cuenta del prefijo. Los tres descuentos que se aplican son:

- D_1 si la cuenta es exactamente igual a 1.
- D_2 si la cuenta es exactamente igual a 2.
- D_3 si la cuenta es mayor o igual a 3.

2.2.4. Modelos de n-gramas basados en características

Una alternativa al uso de un modelo de n-gramas por conteo, cuya principal particularidad es su simplicidad, es el uso de modelos de n-gramas basados en características. Estos modelos no realizan cuentas sobre las palabras del contexto, sino que aplican técnicas para extraer nueva información a partir de ellas u organizan la información original de otras formas.

En este apartado describiremos brevemente dos de estos tipos de modelo: Los modelos basados en árboles de decisión y los modelos de entropía máxima.

Existen otros tipos de modelos de n-gramas basados en características, aunque su exposición sería improductiva para el desarrollo de este trabajo. El lector puede encontrar otros modelos de n-gramas basados en características en [Rosenfeld (2000), Smaıli et al. (2004), Luong et al. (2013)].

Modelos basados en árboles de decisión

Los algoritmos basados en árboles de decisión fueron de los primeros en ser aplicados al modelado del lenguaje [Bahl et al. (1989)]. Los árboles de decisión realizan preguntas con respuestas binarias (verdadero o falso) al contexto de los n-gramas, con el objetivo de particionar el espacio de historias.

Los nodos hoja representan la distribución de probabilidad $P(w_i \mid w_{i-n+1}, \ldots, w_{i-1})$ que se genera a partir de los datos del conjunto de entrenamiento. Esta probabilidad suele interpolarse con la de las distribuciones de los nodos internos del árbol para reducir la varianza de la estimación. El árbol crece según las preguntas que se hacen en los nodos, seleccionando siempre aquella pregunta que más información aporta (según los principios de entropía). Las técnicas de poda y validación cruzada se usan frecuentemente después de la fase de entrenamiento para mejorar la eficiencia y los resultados del modelo.

Los resultados de los modelos basados en árboles de decisión están a la par de los modelos de n-gramas estándar, lo cual no es sorprendente, pues ambos modelos sufren de *la maldición de la dimensión*. Existen propuestas de modelos más recientes que se basan en el uso de random forests [Xu and Jelinek (2004)], los cuales combinan las predicciones de varios árboles de decisión.

Modelos de entropía máxima

Los modelos de entropía máxima [Berger et al. (1996)], en cambio, no sufren a causa de la fragmentación de datos. Esta fragmentación se evita mediente el uso de un modelo exponencial de la forma:

$$P(w \mid h) = \frac{1}{Z(h)} \cdot exp\left[\sum_i \lambda_i f_i(h, w)\right] \tag{2.28}$$

en donde λ_i son los parámetros del modelo, $Z(h)$ es un término de normalización y las funciones f_k son funciones aplicadas a la entrada y que devuelven un 0 o un 1. Dado un corpus de entrenamiento, la MLE satisface la siguiente restricción:

$$\sum_h \widetilde{P}(h) \cdot \sum_w P(w \mid h) \cdot f_i(h, w) = E_{\widetilde{P}} f_i(h, w) \tag{2.29}$$

en donde $\widetilde{P}$ es la distribución empírica del corpus. Esta estimación coincide con la distribución Entropía Máxima o “Maximum Entropy” (ME), es decir, la distribución con el valor más alto de entropía de entre todas las distribuciones y que satisface las restricciones de la ecuación 2.29.

Los modelos de entropía máxima, pese a ser “formalmente correctos” y elegantes, tienen muchos inconvenientes. Entrenar uno de estos modelos es computacionalmente costoso, debido al cálculo del término de normalización. Algunos

trabajos abordan estos modelos sin normalización [Della Pietra et al. (1992)], mientras que otros los suavizan [Chen and Rosenfeld (2000)]. Otro problema de los modelos de entropía máxima es la inducción de las características, es decir, las funciones binarias f_i. Estas funciones han de ser diseñadas manualmente, aunque la solución puede encontrarse usando métodos automáticos o semiautomáticos para la selección de características [Della Pietra et al. (1997), Rosenfeld et al. (1999)].

A pesar de estas contrariedades, la búsqueda de avances en modelos de entropía máxima continúa en marcha actualmente [Mikolov et al. (2011), Rosenberg et al. (2012)].

Los NN LMs, los cuales se presentan más adelante en este mismo capítulo y son los modelos fundamentales de este trabajo, podrían ser considerados modelos de máxima entropía [Kam and Guez (1987), Park and Abusalah (1997)]. Esto se debe a que los NN LMs están basados en redes neuronales con ciertas peculiaridades. Cada capa oculta de una red neuronal excepto la capa anterior a la capa de salida se comporta como un extractor de características (las funciones f_i a las que nos referíamos anteriormente). Las salidas de la capa de salida de un NN LM se obtienen después de aplicar la función softmax (una función con un término de normalización exponencial), mientras que la función de la red es la función de entropía cruzada con regularización L_2, lo cual convierte a la red en una variante lineal de un regresor logístico multiclase. Esta misma definición es válida para los modelos de máxima entropía, por lo que estos modelos son equivalentes a este tipo de redes.

2.3. Modelos de skip-gramas

El problema de emplear n-gramas de alto orden es que hay una probabilidad menor de haber observado el mismo contexto antes, aunque hay una probabilidad bastante alta de haber observado un contexto parecido. Los modelos de skip-gramas ponen en práctica este concepto de forma que no se tienen en cuenta algunas de las palabras del contexto [Huang et al. (1992), Ney et al. (1994), Rosenfeld (2005)].

Los modelos de skip-gramas pueden ser construidos del mismo modo que los modelos de n-gramas, empleando solamente un subconjunto de las palabras previas. Cuando estamos interesados en el contexto de un 5-grama, podemos considerar unos cuantos subconjuntos tales como $P(w_i \mid w_{i-4}, w_{i-3}, w_{i-1})$ o $P(w_i \mid w_{i-4}, w_{i-2}, w_{i-1})$ para construir los modelos de skip-gramas. En el siguiente ejemplo, podemos observar las ventajas que este modelos ofrece frente a los modelos de n-gramas tradicionales.

Supongamos que hemos creado un modelo de 5-gramas en el cual hemos observado la frase "Show Stan a good time". Al usar este modelo para predecir la probabilidad $P(time \mid show, John, a, good)$, el modelo retrocedería a $P(time \mid John, a, good)$, y de ahí a $P(time \mid a, good)$, la cual tendría un probabilidad relativamente baja. Un modelo de skip-gramas de la forma $P(w_i \mid w_{i-4}, w_{i-2}, w_{i-1})$ asignaría a "time" una probabilidad mucho mayor que el modelo previo para la frase "Show Stan a good time".

Los modelos de skip-gramas pueden ser interpolados con modelos de n-gramas para obtener mejores modelos para la predicción:

$$\lambda_1 P(w_i \mid w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1}) + \lambda_2 P(w_i \mid w_{i-4}, w_{i-3}, w_{i-1}) + \lambda_3 P(w_i \mid w_{i-4}, w_{i-2}, w_{i-1}) \tag{2.30}$$

Los modelos de skip-gramas también pueden ser vistos como "una especie de modelos de n-gramas de alto orden para pobres" [Goodman (2001)], interpolando entre sí todos los modelos de skip-gramas resultantes de la extracción de una palabra del n-grama en diferentes posiciones:

$$\lambda_1 P(w_i \mid w_{i-2}, w_{i-1}) + \lambda_2 P(w_i \mid w_{i-3}, w_{i-1}) + \lambda_3 P(w_i \mid w_{i-3}, w_{i-2}) \tag{2.31}$$

El modelo representado por 2.31 está compuesto por trigramas, aunque simula el comportamiento de un modelo de 4-gramas normal.

2.4. Modelos conexionistas de lenguaje (NN LMs)

Los modelos conexionistas de lenguaje, más conocidos por el acrónimo NN LMs (Neural Network Language Models), son modelos de lenguaje basados en redes neuronales. En estas redes, el modelo de lenguaje se plantea como una función que la red neuronal intenta aprender a partir de la presentación de secuencias de palabras de un lenguaje natural. Una red neuronal entrenada de esta forma es capaz de realizar predicciones probabilísticas sobre la siguiente palabra a partir de las palabras anteriores.

Los NN LMs son capaces de proyectar las palabras en un espacio continuo, asignando a cada palabra una posición en dicho espacio. Un NN LM sigue la misma ecuación que los modelos de n-gramas (ecuación 2.7), pero a diferencia de estos, el modelo es capaz de calcular la probabilidad de todas las palabras del vocabulario dada cualquier secuencia posible de las $n-1$ palabras anteriores. Esto es posible gracias a:

- La codificación distribuida de las palabras, consecuencia de la proyección de las palabras en un espacio continuo.
- La interpolación que la red neuronal realiza ante secuencias de entrada desconocidas.

Estas dos características de los NN LMs reducen drásticamente el impacto de *la maldición de la dimensión*, de la cual hemos hablado anteriormente en este mismo capítulo. Este planteamiento teórico de los NN LMs ofrece soluciones a algunas de las deficiencias de los modelos de n-gramas estándar. No obstante, habrá que hacer frente a toda una nueva problemática que surge al modelar el lenguaje de esta forma.

La idea del aprendizaje a partir de la codificación distribuida de datos simbólicos, como las palabras, aparece ya en los inicios de los modelos conexionistas [Hinton (1986), Paccanaro and Hinton (2000)] . Esta noción también se tiene en

cuenta desde los inicios de los modelos conexionistas del lenguaje [Nakamura and Shikano (1988), Miikkulainen and Dyer (1991), Castro et al. (1999)]. Algunos trabajos proponen aprender la codificación distribuida de las palabras de forma desacoplada en un primer paso [Miikkulainen and Dyer (1991), Tortajada and Castro (2006)], aunque la aproximación estándar junta el aprendizaje de la codificación y las probabilidades [Bengio et al. (2003), Schwenk (2007)]. En este trabajo empleamos la aproximación estándar, la cual puede considerarse el resultado de muchos años de investigación en modelos conexionistas del lenguaje.

2.4.1. Codificación distribuida de las palabras

La idea de la codificación distribuida de las palabras siempre ha ido unida a las redes neuronales desde su resurgimiento en los años 80. Sin lugar a dudas, el principal defensor de esta idea ha sido Geoffrey Hinton, quien la publicó anticipadamente en sus artículos [Hinton (1986)] y [Hinton (1989)].

En una codificación distribuida de las palabras, cada palabra es representada mediante una tupla (o vector) de características que determinan el significado de esa palabra. En el caso de que una persona tuviese que elegir las características que representaran a una palabra, es muy probable que la persona escogiese cualidades gramaticales como el número o el genéro para describirla. También es probable que escogiese cualidades semánticas, como "ser un objeto animado" o "ser invisible", con el mismo objetivo. En nuestro caso, el algoritmo de aprendizaje del NN LM es el encargado de descubrir estas características y asignar el valor de esas características en cada una de las palabras del vocabulario.

Concretamente, la codificación distribuida de las palabras asigna una representación vectorial de valores reales a cada una de las palabras del diccionario. Esto equivale a decir que la codificación distribuida de las palabras asigna un punto en un espacio n-dimensional a cada una de las palabras del diccionario (siendo n igual al número de características del vector o el número de dimensiones que describen la ubicación del punto). Podemos imaginar que cada una de estas dimensiones del espacio corresponde a alguna de las características gramaticales o semánticas de las que hablábamos antes.

Lo que se espera de esta idea es que el algoritmo de aprendizaje sea capaz de asignar puntos cercanos en el espacio a palabras "parecidas"[1], al menos en algunas direcciones. De esta forma, una secuencia de palabras puede ser transformada en una secuencia de vectores de características de los que hemos aprendido. También podemos reemplazar aquellas palabras del contexto por otras palabras parecidas sobre las cuales disponemos de más información. Esto elimina el efecto que la *maldición de la dimensión* tiene sobre el modelo de lenguaje.

Con el objetivo de entender mejor este concepto, vamos a suponer que queremos generar una codificación distribuida de las palabras de un vocabulario arbitrario en un espacio bidimensional. Después de entrenar nuestro modelo, cada palabra estará codificada mediante 2 valores reales. Si dibujamos estos valores reales como puntos en una gráfica, obtendremos una gráfica parecida a la de la figura 2.1 (esta gráfica es una ejemplo, aunque podemos encontrar gráficas similares

[1] Palabras parecidas respecto a su funcionalidad.

hechas con datos reales en [Blitzer et al. (2005)], por ejemplo).

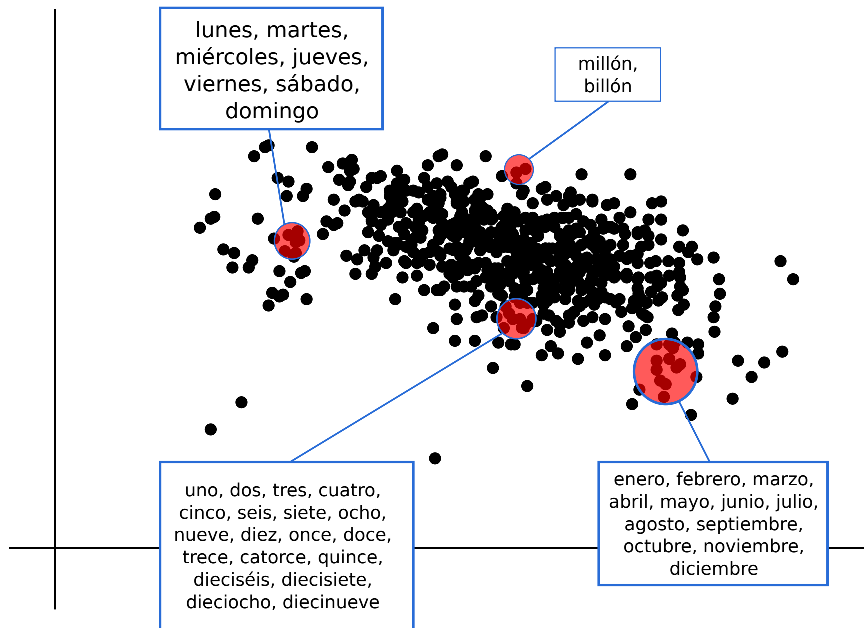

Figura 2.1: Un ejemplo de la codificación distribuida de las palabras de un vocabulario en un espacio bidimensional. Palabras con la misma funcionalidad se concentran en determinadas áreas.

Como podemos observar en la figura, el algoritmo asigna puntos cercanos a las palabras similares. Por ejemplo, los puntos correspondientes a las palabras que representan meses del año (Enero, Febrero, etc.) están contenidos por una misma zona. Lo mismo ocurre con los días de la semana, los primeros números a partir de cero, o los números que describen grandes cantidades. Estas palabras, al ser usadas habitualmente en contextos similares, tienen una funcionalidad parecida, por lo que el algoritmo determina puntos cercanos en el espacio continuo que se ha designado.

Podemos encontrar trabajos anteriores a los NN LMs que utilizaban la codificación distribida para aprender datos simbólicos [Paccanaro and Hinton (2000)], modelar datos lingüísticos [Miikkulainen and Dyer (1991)] y secuencias de caracteres [Schmidhuber and Heil (1996)]. Poco tiempo después, se demostró cómo la codificación distribuida de símbolos podría combinarse con las predicciones probabilísticas de las redes neuronales para superar a los modelos de n-gramas en tareas de modelado de lenguaje estadístico [Bengio et al. (2003)].

2.4.2. Arquitectura de un NN LM

En este trabajo nos hemos basado en la arquitectura estándar de los NN LMs presentados en [Schwenk (2007), Zamora-Martínez (2012)], que detallaremos a

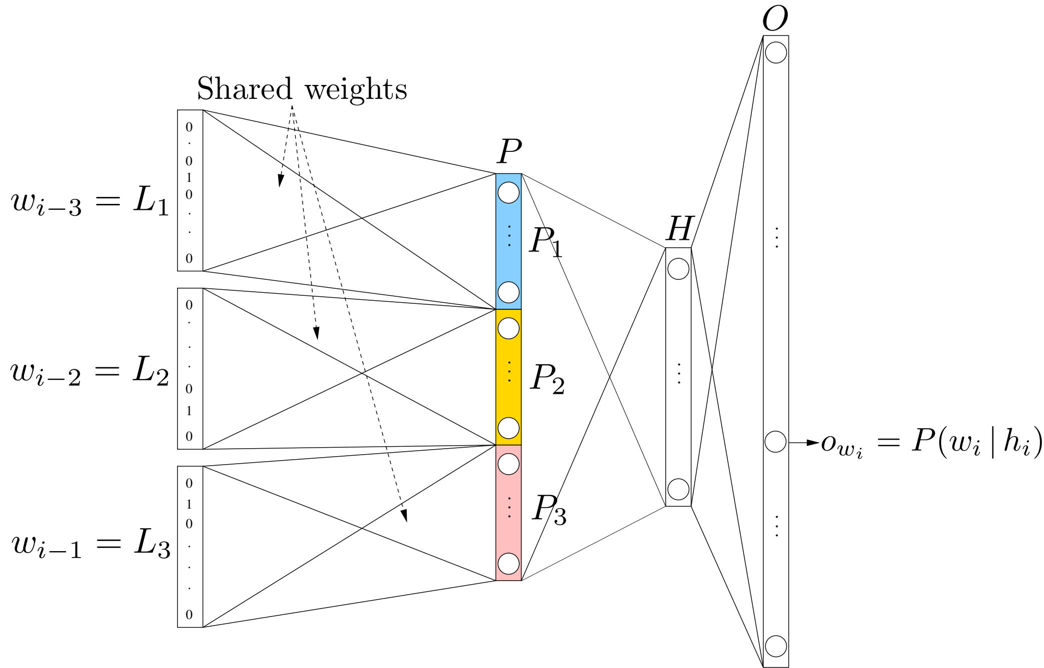

Figura 2.2: La arquitectura de un 4-grama NN LM durante la fase de entrenamiento, con $h_i = w_{i-1}, w_{i-2}, w_{i-3}$ [Zamora-Martínez (2012)].

continuación. Esta arquitectura se ilustra en la figura 2.2, y se compone de cuatro capas de neuronas:

- La capa de entrada $\bar{I}$, compuesta por $(n-1)$ grupos de $|\Omega|$ neuronas. Cada grupo de $|\Omega|$ neuronas representa una de las $n-1$ palabras del contexto, las cuales constituyen la parte conocida que condiciona la salida de la red neuronal. A cada uno de estos grupos los denominamos $\bar{L}_1, \bar{L}_2, \ldots, \bar{L}_{n-1}$, y cada uno de ellos codifica una palabra de forma local, es decir, dada una palabra i en la posición j de la entrada, la codificación de $\bar{L}_j$ será un vector de tamaño $|\Omega|$ neuronas, con un *uno* en la neurona de la posición i y *ceros* en las $|\Omega| - 1$ neuronas restantes. Esta capa se desecha en la fase de evaluación de la red, ya que es posible precomputar y almacenar en una tabla la codificación de cada entrada, tal y como se muestra en la figura 2.3.

- Cada conjunto $\bar{L}_j$ se conecta a un conjunto $\bar{P}_j$ de neuronas mediante la matriz de pesos $\boldsymbol{W}_{LP}$ y el vector de "bias" $\bar{B}_P$. Todos los conjuntos $\bar{P}_j$ tienen el mismo tamaño, y la matriz de pesos y el vector de bias son compartidos por todos los conjuntos de conexiones $\bar{L}_j$ a $\bar{P}_j$. Al conjunto resultante de concatenar todos los grupos $\bar{P}_j$ lo denominamos $\bar{P}$, la capa de proyección. Su función es codificar cada una de las $n-1$ palabras del contexto en un espacio continuo, lo cual permite tener una codificación parecida en palabras parecidas. La probabilidad condicional del modelo de lenguaje se calcula sobre este espacio continuo, lo cual permite que el suavizado de los eventos no vistos durante la fase de entrenamiento esté basado en la semejanza entre

palabras. La función de activación de la capa $\bar{P}$ es la función lineal.

- La capa de proyección se conecta a la capa oculta, formada por el conjunto de neuronas $\bar{H}$, mediante la matriz de pesos $\boldsymbol{W}_{PH}$ y el vector de bias $\bar{B}_H$. La función de activación de la capa $\bar{H}$ es la función tangente hiperbólica.
- Por último, la capa oculta se conecta a la capa de salida $\bar{O}$, formada por un total de $|\Omega|$ neuronas. La conexión se realiza a través de la matrix $\boldsymbol{W}_{HO}$ y el vector de bias $\bar{B}_O$. La función de activación de la capa de salida es la función "softmax", que permite que los valores de salida de la red sean probabilidades, después de aplicar una normalización [Bishop et al. (1995)].

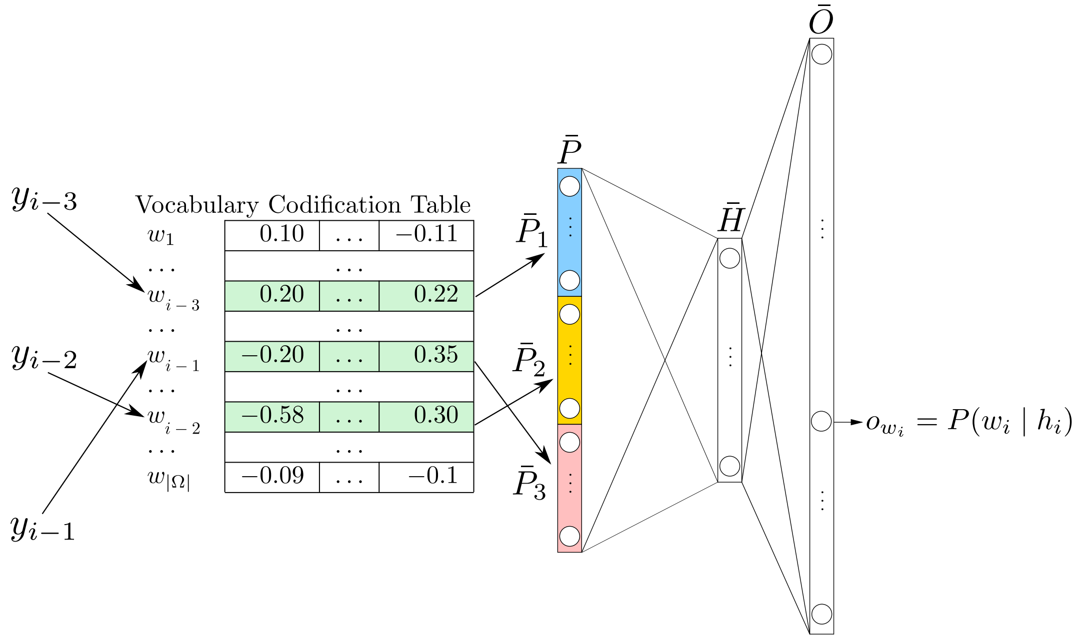

Figura 2.3: La arquitectura de un 4-grama NN LM durante la fase de evaluación, con $h_i = w_{i-1}, w_{i-2}, w_{i-3}$ [Zamora-Martínez (2012)].

Con toda esta notación podemos escribir el cómputo que realiza la red neuronal en cada capa:

$$\bar{P} = \bigcup_{i=1}^{n-1} (\bar{L}_i \cdot \boldsymbol{W}_{LP}^T + \bar{B}_P) \tag{2.32}$$

$$\bar{H} = tanh(\bar{P} \cdot \boldsymbol{W}_{PH}^T + \bar{B}_H) \tag{2.33}$$

$$\bar{A} = \bar{H} \cdot \boldsymbol{W}_{HO}^T + \bar{B}_O \tag{2.34}$$

$$\bar{O} = \frac{exp(\bar{A})}{\sum_{i=1}^{|\bar{A}|} exp(a_i)} \tag{2.35}$$

en donde a_i se refiere a la componente i-ésima del vector $\bar{A}$. El coste de la ecuación 2.32 se reduce radicalmente si se tiene en cuenta que cada entrada $\bar{L}_i$ está codificada de forma local. Por esta razón, el resultado de la multiplicación $\bar{L}_i \cdot \boldsymbol{W}_{LP}^T$ se reduce a escoger la columna de $\boldsymbol{W}_{LP}^T$ correspondiente a la única neurona activada de $\bar{L}_i$. Por tanto, el coste original $O(|\bar{L}_i| \cdot |\bar{P}_i|)$ se reduce a $O(|\bar{P}_i|)$. Hay que destacar que, habitualmente, $|\bar{L}_i| \approx |\Omega|$ y que $|\bar{P}_i| \ll |\bar{L}_i|$.

El vector de bias $\bar{B}_P$ suele ser ignorado en la literatura, de forma que no se entrena ni se tiene en cuenta en los cálculos. No obstante, en este trabajo y, en general, en nuestras herramientas, sí que se tiene en cuenta.

La red neuronal se entrena mediante el algoritmo estándar de entrenamiento de redes neuronales: El algoritmo "Error Backpropagation" o de retropropagación del error (BP). Este algoritmo se emplea en su modo estocástico y "on-line", utilizando la función de entropía cruzada como la función de error de la red. Esta función se define como:

$$e_j = -\sum_{i=1}^{|\bar{O}|} t_i \log(o_i) \tag{2.36}$$

en donde t_i es la salida deseada para la neurona i de la capa de salida, o_i es la salida calculada por la red para la misma neurona, y e_j el error calculado para la muestra de entrenamiento j. El error que comete la red neuronal en un conjunto de R muestras se mide sumando todos los errores y dividiendo después por el número de muestras:

$$E = -\frac{1}{R} \sum_{j=1}^{R} \sum_{i=1}^{|\bar{O}|} t_i^{(j)} \log(o_i^{(j)}) \tag{2.37}$$

en donde $t_i^{(j)}$ es la salida deseada para la neurona i en las muestra j, y $o_i^{(j)}$ el valor de la salida i de la red neuronal para la muestra j. El entrenamiento de la red es on-line, aunque el error se estima sobre un subconjunto de muestras de entrenamiento/validación.

Uno de los problemas de las redes neuronales es el sobreentrenamiento, que ocurre cuando la red es sometida a demasiados ciclos de entrenamiento y termina "memorizando" los datos de entrenamiento. La consecuencia del sobreentrenamiento es que la red neuronal no tiene buenos resultados ante datos nuevos, lo cual se debe a la falta de generalización sobre los datos. Para suavizar este problema, introducimos al algoritmo el término de regularización L_2 "weight decay", que añade a la función de error un porcentaje ϵ de la suma del cuadrado de los pesos de la red neuronal:

$$e_j = -\sum_{i=1}^{|\bar{O}|} t_i \log(o_i) + \epsilon \sum_{w \in \bar{\boldsymbol{W}}} \frac{w^2}{2} \tag{2.38}$$

en donde $\bar{\boldsymbol{W}} = \boldsymbol{W}_L P \cup \boldsymbol{W}_P H \cup \boldsymbol{W}_H O$. Este término fuerza que los pesos tengan valores pequeños, lo cual permite suavizar y generalizar más sobre los datos.

Se dice que el entrenamiento de la red es *estocástico* porque en cada época, en lugar de mostrar a la red el conjunto de entrenamiento completo, se muestra un subconjunto escogido de forma aleatorio y con reemplazamiento. Esto nos permite entrenar la red sólo con un subconjunto de las datos de entrenamiento disponibles, ya que el tamaño del conjunto completo podría ser excesivo para la red. Además, en la mayoría de casos sólo es necesaria una pequeña porción del conjunto de entrenamiento (pequeña si lo comparamos con el conjunto completo) para llegar a un punto de convergencia.

El entrenamiento de la red neuronal se resume en la siguientes operaciones:

$$\frac{\partial E}{\partial O} = \langle o_1 - t_1, o_2 - t_2, \ldots, o_{|\bar{O}|} - t_{|\bar{O}|} \rangle \tag{2.39}$$

$$\bar{B}_O = \bar{B}_O + \eta \frac{\partial E}{\partial O} \tag{2.40}$$

$$\boldsymbol{W}_H O = \epsilon \boldsymbol{W}_H O + \eta \bar{H}^T \frac{\partial E}{\partial O} \tag{2.41}$$

$$\delta \bar{H} = \boldsymbol{W}_H O \frac{\partial E}{\partial O} \tag{2.42}$$

$$\frac{\partial E}{\partial H} = \langle \delta \bar{h}_1 \cdot 0.5(1 - \bar{h}_1^2), \ldots, \bar{h}_i \cdot 0.5(1 - \bar{h}_i^2), \ldots \rangle \tag{2.43}$$

$$\bar{B}_H = \bar{B}_H + \eta \frac{\partial E}{\partial H} \tag{2.44}$$

$$\boldsymbol{W}_P H = \epsilon \boldsymbol{W}_P H + \eta \bar{P}^T \frac{\partial E}{\partial H} \tag{2.45}$$

$$\frac{\partial E}{\partial P} = \boldsymbol{W}_P H \cdot \frac{\partial E}{\partial H} \tag{2.46}$$

$$\bar{B}_P = \bar{B}_P + \frac{1}{n-1} \sum_{i=1}^{n-1} \eta \frac{\partial E}{\partial P_i} \tag{2.47}$$

$$\boldsymbol{W}_L P = \epsilon \boldsymbol{W}_L P + \frac{1}{n-1} \sum_{i=1}^{n-1} \eta \bar{L}_i^T \frac{\partial E}{\partial P_i} \tag{2.48}$$

En el apéndice A se encuentra explicado detalladamente el algoritmo de entrenamiento Backpropagation para redes neuronales generales.

Capítulo 3

Diseño orientado a consultas masivas

En este capítulo se describen las decisiones tomadas respecto al diseño de la herramienta. En primer lugar, se justifica la partición de los modelos de lenguaje en dos componentes distintas: El modelo y la interfaz. A continuación, exponemos las clases, estructuras y formatos auxiliares que usaremos en las clases diseñadas para representar los modelos de lenguaje. Acto seguido, mostramos la jerarquía de dichas clases, resaltando aquellos métodos que resulten de especial interés para este proyecto. Por último, explicamos cómo hemos desarrollado el "binding" de las clases escritas en código C++ a código Lua y presentamos ejemplos de scripts en Lua para la ejecución de experimentos sirviéndose de estas clases.

El diseño de clases orientado a consultas masivas se exhibe como una aportación más en este trabajo. Un buen diseño de clases puede ser costoso de implementar, pero permite ahorrar tiempo a la hora de extender la herramienta. Nuestra intención es crear un diseño de clases que permita heredar de alguna clase que ya implemente la funcionalidad básica de un modelo de lenguaje, y que mediante la sobreescritura de métodos se extienda la funcionalidad de la clase hasta el punto deseado.

3.1. Partición de los modelos de lenguaje

A la hora de implementar un conjunto de herramientas para el modelado del lenguaje, lo más lógico sería crear una clase que represente los modelos de lenguaje, como suele hacerse en la mayoría de este tipo de herramientas [Stolcke et al. (2002)]. Sin embargo, nuestra experiencia en la elaboración de estas herramientas nos ha enseñado que es mejor separar el modelo de lenguaje en dos partes: Una parte que representa el modelo de lenguaje en sí junto con sus propiedades, y otra parte que representa la interfaz del modelo, la cual nos permite interactuar con el modelo mediante la realización de consultas. Denominamos a estas partes el "Modelo" y la "Interfaz". Esta división del modelo de lenguaje en dos componentes puede aparentar ser innecesaria, por lo que a continuación vamos a tratar de justificar por qué no lo es.

En este proyecto nos hemos planteado un diseño orientado a consultas múltiples. Este diseño nos obliga a disponer de estructuras de datos adicionales que almacenen los datos de esas consultas. La principal tarea de estas estructuras de datos sería la de administrar las consultas, algo que poco tiene que ver con la funcionalidad del modelo de lenguaje. Estas estructuras, por tanto, deberían estar separadas del Modelo.

Por otra parte, estas estructuras de datos no deberían ser las encargadas de llamar a los métodos del Modelo, dado que su funcionalidad es puramente administrativa. Por esta razón, necesitamos una clase adicional que use estas estructuras de datos como un recipiente para consultas y al mismo tiempo sea capaz de realizar las consultas. Esta clase es la Interfaz. La Interfaz implementa los métodos que le permiten realizar las consultas insertadas en la estructuras de datos que las almacena, e inserta los resultados de las consultas en esta misma estructura.

El Modelo únicamente suministra a la Interfaz las estructuras de datos y parámetros que definen las propiedades del modelo de lenguaje. El Modelo delega la responsabilidad de conseguir los resultados de las consultas en la Interfaz, la cual utiliza sus métodos para dicho fin.

3.2. Estructuras de datos empleadas en los modelos de lenguaje

Antes de empezar a describir la jerarquía de clases de los modelos de lenguaje, es oportuno describir las clases, estructuras y formatos de las cuales se hace uso. No todas las clases de modelos de lenguajes hacen uso de estos elementos, sino que más bien algunos tipos de modelos están vinculados a determinadas clases. Por ejemplo, la funcionalidad del modelo `HistoryBasedLM` está supeditada a la del trie de palabras, mientras que el modelo `BunchHashedLM` emplea múltiples tablas hash para implementar el modo "bunch" de forma eficiente.

3.2.1. La estructura KeyScoreBurdenTuple

El principal cometido de la estructura `KeyScoreBurdenTuple` es almacenar los resultados de una consulta. Esta estructura no tiene ningún método, y se compone de tres campos:

- **Key**: La clave a la cual se transita a partir de la clave actual y una palabra.
- **Score**: La puntuación con la cual se transita a la siguiente clave.
- **Burden**: La "carga" del resultado, que se compone de una clave y un identificador de palabra. Este atributo no es relevante cuando se trabaja con modelos de lenguaje, aunque es necesario al trabajar en aplicaciones relacionadas como el reconocimiento de voz o el reconocimiento de escritura manuscrita.

3.2.2. La estructura KeyScoreMultipleBurdenTuple

La estructura `KeyScoreMultipleBurdenTuple` es similar a la estructura `KeyScoreBurdenTuple`, aunque en lugar de tener el atributo **Burden**, tiene un vector de **Burden**. Esto es útil para guardar el resultado de un conjunto de consultas con una misma clave y palabra, pero diferente **Burden**.

3.2.3. Trie de palabras

Un trie es una estructura de datos de tipo arbóreo para la búsqueda y recuperación de información [De La Briandais (1959)]. Por lo general, un trie almacena un conjunto de claves como una secuencia de símbolos que pertenece a un alfabeto. Los nodos del trie no almacenan información sobre las claves asociadas a esos nodos, más bien es su posición en el trie la que define la clave a la cual está asociada. El nodo raíz se asocia a la cadena vacía, y todos los descendientes de un nodo tienen un prefijo común asociado a ese nodo. Los valores asociados a las claves se almacenan en los nodos hoja del trie.

En nuestro caso, el alfabeto del trie de palabras es el vocabulario de nuestro modelo de lenguaje, y las secuencias que nos permiten acceder a los nodos hoja del trie son las secuencias de palabras vistas hasta el momento. Ningún nodo hoja guarda información adicional sobre estas secuencias. De esta forma, es posible detectar qué secuencias de palabras han aparecido hasta el momento simplemente realizando un recorrido por el trie.

El trie se implementa como una tabla hash con direccionamiento abierto sobre un vector de tamaño estático. Los nodos del trie contienen tres campos:

- El índice al padre: Un valor que identifica la posición del nodo padre en el vector.
- El "timestamp": El ciclo en el cual se ha insertado el nodo. Esto sirve para identificar la "edad" de los nodos y ahorrar tiempo en la búsqueda de secuencias. Los nodos persistentes se identifican con el valor cero en el campo "timestamp".
- La palabra: El identificador de la palabra con la cual se transita al nodo.

La figura 3.1 muestra cómo es posible representar un trie de palabras a partir de los contenidos de su vector. Un trie puede representar un modelo de lenguaje por sí mismo, aunque en este proyecto se utiliza como una estructura de datos auxiliar en la clase `HistoryBasedLM` y sus clases derivadas.

3.2.4. El formato Lira

El formato Lira es el formato alternativo al formato ARPA para la representación de modelos de n-gramas en APRIL. Un modelo de n-gramas en formato Lira puede obtenerse a partir de un modelo de n-gramas en formato utlizando el script *arpa2lira.lua* de APRIL. En el apéndice B se describe el formato Lira detalladamente.

El formato Blira, resultante de convertir un modelo de n-gramas en formato Lira a binario, puede cargarse en memoria directamente.

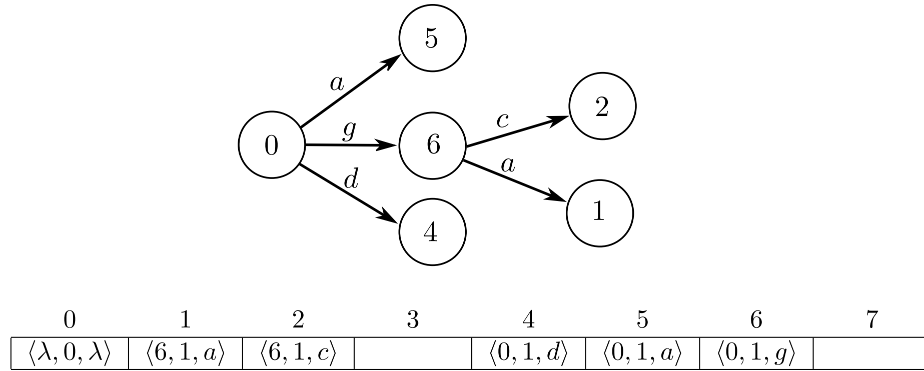

Figura 3.1: Ejemplo de un trie de palabras con un vector de tamaño máximo 8, creado a partir del conjunto de n-gramas {a, gc, ga, d}. Cada nodo contiene una tupla $\langle x, y, z \rangle$, en donde x es el índice del padre, y es el "timestamp" y z es la palabra de transición [Zamora-Martínez (2012)].

3.2.5. Tabla hash

Una tabla hash es una estructura de datos que asocia claves a valores. Una tabla hash usa una función hash para calcular el índice en el cual ubicar la casilla del valor a partir de la clave. Idealmente, una función hash asignará una casilla diferente a cada clave, aunque esta situación es prácticamente inalcanzable, por lo que la mayoría de diseños asumen que habrá colisiones hash y lo resuelven con diferentes métodos (encadenamiento separado y direccionamiento abierto son los más populares).

En este proyecto, las tablas hash se utilizan para la agrupación de consultas similares. Una tabla exterior se emplea para agrupar las consultas por clave, asociando esas claves con una tabla hash que, a su vez, agrupa las consultas con una misma palabra con la que transitar. Estas tablas hash interiores asocian las palabras a una estructura `KeyScoreMultipleBurdenTuple`, las cuales están capacitadas para guardar el resultado de esas consultas y el **Burden** de cada una. La figura 3.2 muestra un esquema sobre cómo se organizan estas estructuras.

El uso de tablas hash para la agrupación de consultas simplifica mucho el proceso de inserción de consultas, así como el proceso de recolección y supresión de resultados de consultas. El uso de otras estructuras de datos, como por ejemplo un vector dinámico, obligaría a agrupar las consultas en el momento de insertarlas o ejecutarlas, causando un sobrecoste temporal notable.

3.2.6. La clase Token

La clase `Token` representa un token, que a su vez representa un conjunto de patrones. Es una clase abstracta que declara un conjunto de métodos sin implementación que todo tipo de token debería implementar, e implementa otro

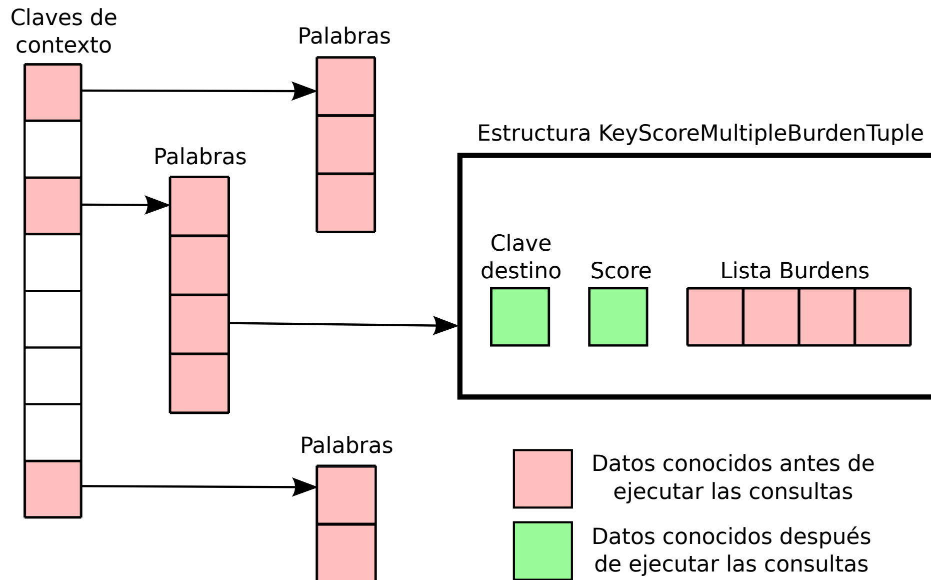

Figura 3.2: Tablas hash para la agrupación de consultas. La tabla exterior relaciona claves de contexto con tablas que, a su vez, relacionan palabras con estructuras capaces de almacenar los resultados de múltiples consultas.

conjunto de métodos que sirven para la comparación y conversión entre tokens.

3.2.7. La clase FunctionInterface

La clase `FunctionInterface` representa un filtro que genera un token a partir de otro token. Es una clase abstracta que declara la interfaz de los filtros, es decir, todos los métodos que cualquier filtro debería implementar.

Los métodos sin implementación que la clase `FunctionInterface` declara son:

- **getInputSize**: Devuelve el tamaño del token de entrada. Si el tamaño es variable, entonces devuelve 0.
- **getOutputSize**: Devuelve el tamaño del token de salida. Si el tamaño es variable, entonces devuelve 0.
- **calculate**: Genera un token de salida a partir de un token de entrada.

3.3. Jerarquía de clases

La figura 3.3 muestra los diagramas de las clases implementadas en APRIL para representar un modelo de lenguaje y su interfaz. A continuación comentaremos la funcionalidad de estas clases y la de sus atributos y métodos más importantes.

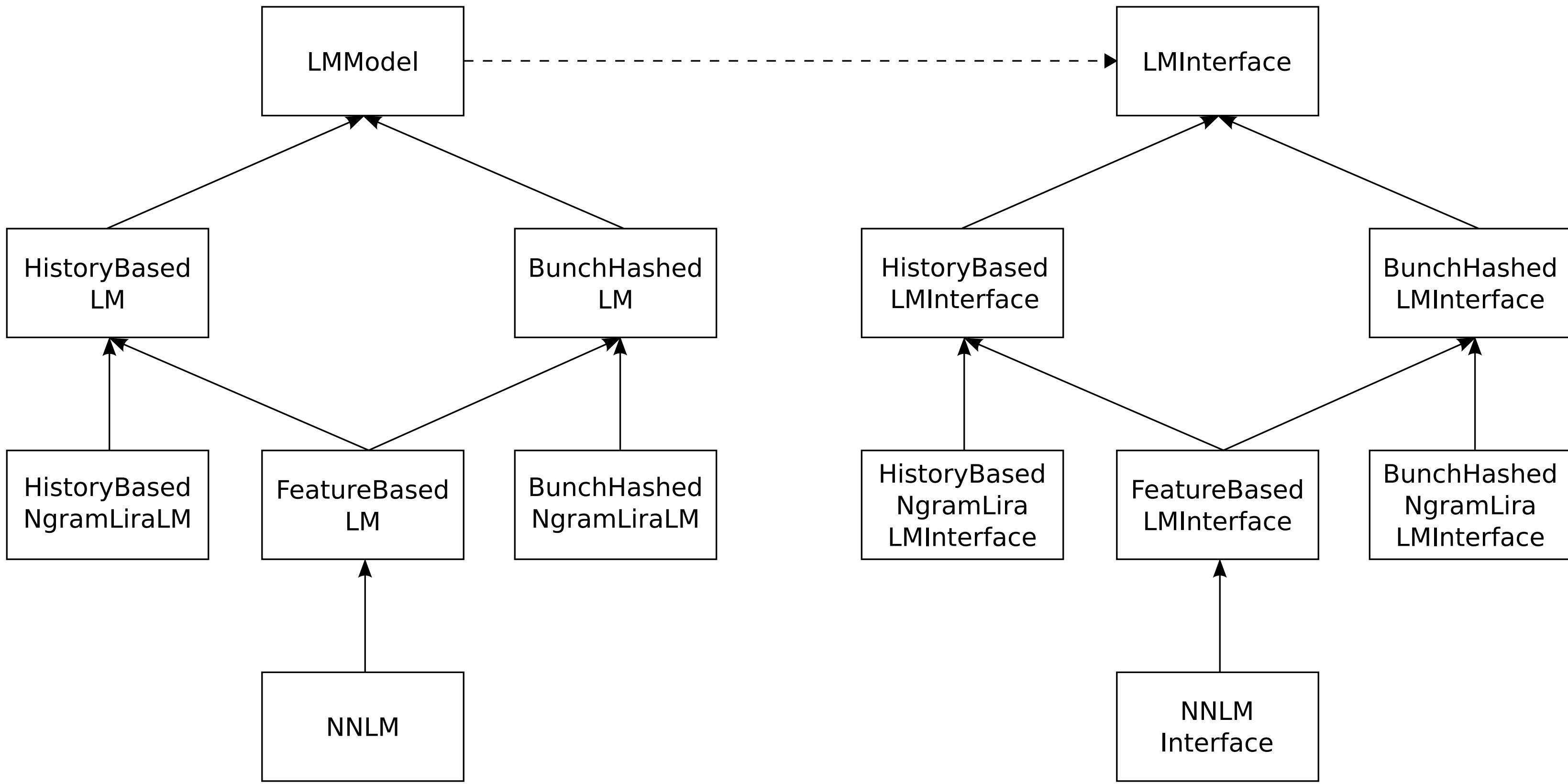

Figura 3.3: Diagramas de clases para los Modelos y las Interfaces. Cada clase derivada de la clase `LMModel` tiene asociada la clase derivada `LMInterface` correspondiente.

3.3.1. La clase LMModel

La clase `LMModel` representa un modelo de lenguaje genérico. Es una clase base abstracta que sirve para declarar los métodos básicos de cualquier modelo de lenguaje. Esta clase no contiene ningún atributo, e implementa los siguientes métodos:

- **isDeterministic**: Devuelve verdadero si el modelo de lenguaje es determinista[1]. Devuelve falso en caso contrario. Por defecto, se considera que los modelos de lenguaje son no deterministas.
- **ngramOrder**: Devuelve -1 si el modelo de lenguaje no es un modelo de n-gramas. En caso contrario, devuelve el orden del modelo de n-gramas.
- **requireHistoryManager**: Devuelve verdadero si el modelo de lenguaje requiere un `HistoryManager`. Devuelve falso en caso contrario.

La clase `LMModel` también declara un método sin implementación, el método **getInterface**, que devuelve la interfície asociada al modelo de lenguaje.

3.3.2. La clase LMInterface

La clase `LMInterface` representa la interfaz de un modelo de lenguaje genérico. Al igual que la clase que representa el modelo al cual la interfaz está asociada, esta es una clase abstracta que declara los métodos básicos que cualquier interfaz de modelo de lenguaje debería implementar.

La clase `LMInterface` declara la variable *result*, un vector de estructuras `KeyScoreBurdenTuple` en donde se almacenan los resultados de las consultas al modelo de lenguaje. Este vector se devuelve al realizar cualquier tipo de consulta, y los datos sobre esas consultas pueden extraerse de los campos de los miembros del vector.

En la siguiente lista exponemos la funcionalidad que los métodos con implementación en la clase `LMInterface` deberían ofrecer en las clases derivadas:

- **getLMModel**: Devuelve el modelo al cual está asociada esta interfaz.
- **getNextKeys**: Almacena en el vector *result* exclusivamente las claves de los estados a partir de los cuales se transita a partir de una clave y una palabra dada.
- **insertQuery**: Inserta una consulta individual.
- **insertQueries**: Inserta un conjunto de consultas.
- **getQueries**: Realiza todas las consultas insertadas hasta el momento.
- **clearQueries**: Elimina todas las consultas y los resultados de consultas.

[1]Consideramos como modelos de lenguaje no deterministas a aquellos que pueden obtener múltiples puntuaciones para un mismo contexto y palabra

- **getZeroKey**: Devuelve verdadero si se almacena la clave del estado "Zero" en la referencia que recibe como único argumento.

Los detalles sobre la implementación de estos métodos en la clase `LMInterface` no son lo suficientemente relevantes para describirlos, ya que se trata de unos métodos con funciones muy básicas (devolver un valor booleano, borrar los miembros de un vector, etc.) que en la mayoría de casos serán sobreescritos.

La clase `LMInterface` también declara múltiples métodos sin implementación. A continuación comentamos la funcionalidad que estos métodos deberían implementar en las clases derivadas:

- **get**: Realiza una consulta a partir de un clave y una palabra dada. Los resultados se guardan en el vector *result*.
- **getBestProp**: Existen dos métodos con este nombre. El primer método no tiene argumentos y devuelve la probabilidad más alta de transición del modelo. El segundo método recibe una clave y devuelve la probabilidad más alta de transición a partir del estado correspondiente a esa clave.
- **getInitialKey**: Devuelve la clave del estado inicial.
- **getFinalScore**: Devuelve la probabilidad de transitar al estado final desde el estado cuya clave se suministra como argumento.

Hay que destacar que la clave del estado inicial y la clave del estado "Zero" de un modelo de lenguaje son diferentes. Esto se debe a que ambos estados son diferentes, a pesar de que ambos sean considerados estados "iniciales". En realidad, el estado "Zero" es el estado del cual se parte y al cual se transita en caso de que se observa una palabra fuera del vocabulario. El estado al cual nos referimos como estado inicial es el estado al cual se transita con $n-1$ marcas de inicio a partir del estado "Zero-gram".

Como puede observarse, la clase `LMInterface` dispone de un conjunto de métodos a partir del cual es posible implementar toda la funcionalidad de un modelo de lenguaje. Los métodos de la clase `LMModel`, en cambio, se limitan a proporcionar los atributos y propiedades del modelo de lenguaje.

3.3.3. La clase HistoryBasedLM

La clase `HistoryBasedLM` representa un modelo de lenguaje basado en la historia reciente. Esta es una clase abstracta derivada de la clase `LMModel` que sobreescribe todos sus métodos, definiéndose como un modelo de lenguaje determinista y que requiere un `HistoryManager`.

Esta clase también añade un par de "getters" para obtener el identificador de la marca de inicio de frase y el trie de palabras desde fuera de la clase.

3.3.4. La clase HistoryBasedLMInterface

La clase `HistoryBasedLMInterface` representa la interfaz de un modelo de lenguaje basado en la historia reciente. Esta es una clase abstracta derivada de la clase `LMInterface`.

La clase `HistoryBasedLMInterface` declara un par de métodos auxiliares en la sección **protected** de la clase que son empleados en la mayoría de sus métodos:

- **getContextProperties**: Recorre el trie hacia atrás a partir de una clave para obtener el contexto y lo almacena en una referencia que se pasa como argumento. Devuelve la longitud del contexto.

- **getDestinationKey**: Devuelve la clave a la que se llega después de recorrer el trie con las palabras del contexto y el identificador de la siguiente palabra, facilitados como argumentos.

Esta clase declara 3 métodos virtuales sin implementación que las clases derivadas han de implementar:

- **privateGet**: Devuelve verdadero si es posible calcular la puntuación para una consulta con los datos suministrados. La puntuación se guarda en una de las referencias pasadas como argumentos. Devuelve falso en caso contrario.

- **privateGetFinalScore**: Devuelve verdadero si es posible calcular la puntuación para una consulta en la cual se transita con el marcador de final de sentencia. La puntuación se guarda en una de las referencias pasadas como argumentos. Devuelve falso en caso contrario.

- **privateBestProb**: Devuelve verdadero si es posible calcular la puntuación para una consulta con los datos suministrados. La puntuación se guarda en una de las referencias pasadas como argumentos. Devuelve falso en caso contrario.

La clase `HistoryBasedLMInterface` sobreescribe los siguientes métodos:

- **get**: Recupera las palabras del contexto y almacena en el vector de resultados, si es posible, la clave destino y la puntuación de la consulta. La responsabilidad de calcular la puntuación de la consulta se delega en el método **privateGet**.

- **getNextKeys**: Recupera las palabras del contexto e inserta la clave destino en el vector de resultados.

- **getZeroKey**: Devuelve verdadero después de modificar la referencia que recibe como argumento para que apunte al nodo raíz del trie.

- **getInitialKey**: Devuelve la clave del trie a la cual se llega después de transitar $n-1$ con la marca de inicio desde la raíz del trie.

- **getFinalScore**: Recupera las palabras del contexto y almacena en el vector de resultados, si es posible, la clave destino y la puntuación de la consulta con el identificador de la marca de final de sentencia. La responsabilidad de calcular la puntuación de la consulta se delega en el método **privateGetFinalScore**.

- **getBestProb**: Recupera las palabras del contexto y obtiene, si es posible, la probabilidad más alta a partir de la clave que se pasa como único argumento. Delega en **privateBestProb** la responsabilidad de calcular la puntuación de la consulta.

3.3.5. La clase HistoryBasedNgramLiraLM

La clase `HistoryBasedNgramLira` es básicamente un *wrapper* sobre un modelo de n-gramas en formato Lira. Como su nombre indica, esta es una clase derivada de la clase `HistoryBasedLM`.

3.3.6. La clase HistoryBasedNgramLiraLMInterface

La clase `HistoryBasedNgramLiraLMInterface` es la interfaz asociada al modelo `HistoryBasedNgramLiraLM`. Esta clase deriva de la clase `HistoryBasedLMInterface` y, al igual que el modelo al que se asocia, puede ser instanciada. Gracias al diseño realizado hasta el momento, la clase `HistoryBasedNgramLiraLMInterface` tan sólo ha de implementar tres métodos, los cuales no tienen implementación en la clase `HistoryBasedLMInterface`. Estos métodos son:

- **privateGet**: Obtiene la clave del modelo Lira usando las palabras del contexto y realiza la consulta a través de la interfaz del modelo Lira. Devuelve verdadero.

- **privateGetFinalScore**: Obtiene la clave del modelo Lira usando las palabras del contexto y realiza la consulta con la marca de final de sentencia a través de la interfaz del modelo Lira. Devuelve verdadero.

- **privateBestProb**: Obtiene la clave del modelo Lira usando las palabras del contexto y llama a la función **getBestProb** con la clave obtenida. Devuelve verdadero.

También se sobreescriben el constructor y el destructor de la clase, que se encargan de guardar y liberar la referencia a la interfaz del modelo Lira, respectivamente.

3.3.7. La clase BunchHashedLM

La clase `BunchHashedLM` representa un modelo de lenguaje que puede realizar consultas en modo “bunch”. Esta es una clase abstracta derivada de la clase `LMModel`, que sobreescribe todos sus métodos y se define como un modelo determinista que no requiere un `HistoryManager`.

La clase `BunchHashedLM` añade un “setter” y un “getter” para el parámetro **bunch_size**, que indica el tamaño máximo del “bunch” para el modelo.

3.3.8. La clase BunchHashedLMInterface

La clase `BunchHashedLMInterface` representa la interfaz de un modelo de lenguaje que puede realizar en consultas en modo "bunch". Esta es una clase abstracta derivada de la clase `LMInterface`.

La clase `BunchHashedLMInterface` declara un nuevo método sin implementación llamado **computeKeysAndScores**. Como su nombre indica, este método se encarga de calcular las claves destino y las puntuaciones de las consultas insertadas hasta el momento en la tabla hash del modelo. Este método también se encarga de rellenar los campos de las estructuras a las que apunta la tabla hash con los resultados de las consultas.

Esta clase sobreescribe todos aquellos métodos relacionados con las consultas:

- **insertQuery**: Inserta una tabla hash en la casilla corespondiente a la clave de la consulta en la tabla hash exterior en caso de que no exista. A continuación, inserta una estructura `KeyScoreMultipleBurdenTuple` en la casilla correspondiente a la palabra de la consulta en la tabla interior en caso de que no exista. Por último, se inserta `Burden` de la consulta en el vector de la estructura a la que apunta la palabra de la consulta.

- **getQueries**: Primero llama al método **computeKeysAndScores**, en el cual delega el cómputo de resultados de las consultas insertadas en la tabla hash. Después se encarga de recolectar los resultados de las tablas hash e ir insertándolos en el vector *result* de la clase.

- **clearQueries**: Elimina todas las consultas y resultados de la tabla hash.

3.3.9. La clase FeatureBasedLM

La clase `FeatureBasedLM` representa un modelo de lenguaje que genera un conjunto de datos a partir de los n-gramas que recibe como entrada. Esta es una clase abstracta derivada de las clases `HistoryBasedLM` y `BunchHashedLM`.

La clase `FeatureBasedLM` añade un "setter" y un "getter" para el atributo **filter**, que contiene una referencia al filtro que se aplica a los n-gramas para conseguir un nuevo conjunto de datos.

3.3.10. La clase FeatureBasedLMInterface

La clase `FeatureBasedLMInterface` representa la interfaz de un modelo de lenguaje que genera un conjunto de datos a partir de los n-gramas que recibe como entrada. Esta es una clase abstracta derivada de las clases `HistoryBasedLMInterface` y `BunchHashedLMInterface`.

La clase `FeatureBasedLMInterface` implementa el método **computeKeysAndScores** que la clase `BunchHashedLMInterface` declara como un método sin implementación. Este método se encarga de iterar sobre las tablas hash e insertar los datos de las consultas en un vector de tokens. A este vector de tokens se le aplica un filtro que genera un token nuevo, el cual se pasa como argumento al método **protected** sin implementación **executeQueries** que esta misma clase declara.

3.3.11. La clase NNLM

La clase `NNLM` representa un modelo de lenguaje basado en redes neuronales. Esta es una clase instanciable que deriva de la clase `FeatureBasedLM` y añade “setters” y “getters” para los siguientes atributos del NN LM:

- Las redes neuronales que lo componen.
- Los coeficientes de las redes neuronales.
- Las tablas de codificación de cada red neuronal.
- El identificador de las palabras “unknown”.
- El factor de suavizado para las palabras “unknown”.
- El flag que indica si la red es *on-the-fly*, es decir, si se fuerza el cálculo de probabilidades después de no encontrar la constante de normalización softmax en su tabla de constantes.

3.3.12. La clase NNLMInterface

La clase `NNLMInterface` representa la interfaz de un modelo de lenguaje basado en redes neuronales. Esta es una clase instanciable que deriva de la clase `FeatureBasedLM`.

La clase `NNLMInterface` implementa el método **executeQueries** que la clase `FeatureBasedLMInterface` declara como un método sin implementación. Este método es prácticamente un *wrapper* que llama al método **forward** de las redes neuronales que componen el NN LM y se encarga de devolver los resultados de las consultas que recibe a partir de las salidas de estas redes.

3.4. Scripts en Lua para el diseño de experimentos

La elección de un lenguaje de programación como C++ para la implementación de nuestro diseño resulta acertada, ya que el código C++ es muy eficiente. En cambio, la elección del mismo lenguaje para el diseño de experimentos no es tan acertada, sino que es más adecuado emplear un lenguaje de *scripting*. El lenguaje de *scripting* Lua es, por tanto, un buen candidato como lenguaje para el diseño de experimentos.

La implementación del proyecto April se hace con estos dos lenguajes. Las clases escritas en código C++ se enlazan a código Lua, permitiendo manejar las clases C++ de una forma más ligera para el usuario. A continuación explicaremos en qué consiste un “binding” y cómo podemos usarlo de forma exitosa para el diseño de experimentos en Lua.

3.4.1. Binding

El "binding" ("enlazado" en español) de un lenguaje de programación a una librería o un servicio de sistema operativo es una Interfaz de Programación de Aplicaciones o "Application Programming Interface" (API) que nos permite hacer uso de esa librería o servicio en un determinado lenguaje de programación. En el toolkit April, el "binding" se realiza sobre librerías de C++, organizadas por paquetes, para que las clases que declaran e implementan esa librerías puedan ser usadas desde el lenguaje Lua.

El "binding" no se realiza de forma automática, sino que ha de ser programado previamente en un fichero de "binding". Estos ficheros sirven para incluir los métodos de las clases C++ en el intérprete Lua que generamos al compilar todo el código. En los ficheros de "binding" podemos:

- Declarar el uso de ficheros de cabecera.
- Declarar el nombre de las funciones Lua que enlazan los métodos de una clase C++.
- Indicar las clases C++ que derivan de otras clases C++.
- Insertar código antes y después de llamar a los métodos de las clases C++, incluyendo los constructores y destructores, con el objetivo de preparar los argumentos de la llamada o tratar los datos que devuelve la misma.

El siguiente código corresponde a una parte del fichero de "binding" de la clase `FunctionInterface`, la clase abstracta que declara la interfaz de los filtros:

```
//BIND_HEADER_C
#include "bind_tokens.h"
//BIND_END

//BIND_HEADER_H
#include "function_interface.h"
#include "identity_function.h"
using namespace Functions;
using basics::Token;
//BIND_END

//BIND_LUACLASSNAME FunctionInterface functions
//BIND_CPP_CLASS    FunctionInterface

//BIND_CONSTRUCTOR FunctionInterface
{
  LUABIND_ERROR("Abstract class!!!");
}
//BIND_END
```

```
//BIND_METHOD FunctionInterface get_input_size
{
  LUABIND_CHECK_ARGN(==,0);
  LUABIND_RETURN(uint, obj->getInputSize());
}
//BIND_END

//BIND_METHOD FunctionInterface get_output_size
{
  LUABIND_CHECK_ARGN(==,0);
  LUABIND_RETURN(uint, obj->getOutputSize());
}
//BIND_END

//BIND_METHOD FunctionIeterface calculate
{
  LUABIND_CHECK_ARGN(==,1);
  Token *input;
  LUABIND_GET_PARAMETER(1, Token, input);
  LUABIND_RETURN(Token, obj->calculate(input));
}
//BIND_END
```

Como podemos observar en el código, en el enlazado del constructor de esta clase se ejecuta una instrucción de error que advierte que la clase `FunctionInterface` es una clase abstracta, y por lo tanto no pueden crearse instancias de esta clase.

El “binding” de los métodos virtuales con implementación contiene la declaración de una variable para guardar el resultado de la llamada al método. La sobreescritura de cualquier método virtual en una clase derivada no conlleva la sobreescritura del “binding” asociado.

La clase `SkipFunction`, que implementa los filtros skip y que comentaremos en el capítulo 4 con más detalle, sobreescribe el método **calculate** de la clase `FunctionInterface`, aunque no necesita sobreescribir el “binding”. No obstante, la clase `SkipFunction` contiene atributos adicionales que recibe en el constructor, de forma que la signatura del constructor cambia, por lo que es necesario sobreescribir su “binding”. El “binding” de la clase `SkipFunction` se reduce a sobreescribir el “binding” para su constructor:

```
//BIND_LUACLASSNAME FunctionInterface functions

//BIND_LUACLASSNAME SkipFunction functions.skip
//BIND_CPP_CLASS SkipFunction
//BIND_SUBCLASS_OF SkipFunction FunctionInterface

//BIND_CONSTRUCTOR SkipFunction
{
```

```
  LUABIND_CHECK_ARGN(==,1);
  LUABIND_CHECK_PARAMETER(1, table);
  check_table_fields(L, 1, "dice", "random", "mask_value");
  Dice *dice;
  MTRand *random;
  uint32_t mask_value;
  LUABIND_GET_TABLE_PARAMETER(1, dice, Dice, dice);
  LUABIND_GET_TABLE_PARAMETER(1, random, MTRand, random);
  LUABIND_GET_TABLE_PARAMETER(1, mask_value, uint, mask_value);
  obj = new SkipFunction(dice, random, mask_value);
  LUABIND_RETURN(SkipFunction, obj);
}
//BIND_END
```

Las funciones **LUABIND** nos permiten comprobar en tiempo de ejecución el número o tipo de los argumentos con el cual se ha llamado la función, obtener los valores de los argumentos, devolver un valor como resultado de la ejecución de un método, etc.

3.4.2. Ejemplos

Después de escribir los ficheros de "binding" y generar el intérprete, escribir scripts para describir experimentos resulta trivial si se tiene un buen manejo del lenguaje Lua. El intérprete generado es capaz de ejecutar código en Lua del mismo modo que el intérprete original, y también puede ejecutar los métodos de las clases C++ que hemos puesto en el "binding".

A continuación se muestran un par de ejemplos sencillos que ilustran cómo podemos emplear objetos C++ desde código Lua para ejecutar operaciones características en nuestro diseño para los modelos de lenguaje.

Ejemplo 1: Inserción de consultas en modo "bunch"

La inserción de consultas puede efectuarse de tres formas diferentes:

1. Insertar una consulta individual con el método **get** para obtener el resultado de la consulta al instante.

2. Insertar una consulta en modo "bunch" con el método **insertQueries**.

3. Insertar un conjunto de consultas en modo "bunch" con el método **insertQueries**.

La llamada al método **get** de la interfaz de los modelos de lenguaje ejecuta la consulta automáticamente después de ser insertada. Esto puede causar problemas de eficiencia debido al principio de localidad: El programa ejecuta dos zonas de código diferentes de forma intermitente (inserción, ejecución, inserción, ejecución...) por lo que es de esperar que se generen fallos en la memoria principal y la memoria caché. Otra excusa válida para explicar la poca eficiencia de este

procedimiento es que algunas consultas pueden repetirse, generando así un coste adicional por cada consulta que ya se haya realizado previamente para el mismo modelo de lenguaje.

Las otras dos formas de insertar consultas son más eficientes, ya que agrupan las consultas en tablas hash, evitando posibles repeticiones, y ejecutan las consultas en modo "bunch" (de forma paralela) si el modelo lo permite. No existe una gran diferencia entre ambas formas, ya que el método para la inserción de un conjunto de consultas utiliza el método para la inserción de consultas individuales. No obstante, para insertar un conjunto de consultas, primero tenemos que acumularlas dentro de una tabla, lo cual resulta poco práctico si es posible insertarlas directamente.

El siguiente trozo de código se encuentra dentro de un bucle que itera sobre las líneas de un texto e introduce las consultas de una línea entera de forma individual:

```
local words = string.tokenize(line)
local wids  = vocab:searchWordIdSequence(words, vocab:getWordId(unk_word))
table.insert(scores, 0)
local key
if use_bcc then
  key = lmi:get_initial_key()
else
  key = lmi:get_zero_key()
end
for wpos=1,#wids do
  local w = wids[wpos]
  lmi:insert_query(key, w, { id_key = #scores})
  result = lmi:next_keys(key, w)
  assert(#result == 1)
  key = result:get(1)
end
if use_ecc then
  local w = vocab:getWordId(end_word)
  lmi:insert_query(key, w, { id_key = #scores })
end
```

En esta parte del código, la variable `lmi` es la interfaz del modelo de lenguaje con el cual estamos trabajando. Después de obtener los identificadores de las palabras de la frase, los métodos **get_zero_key** y **get_initial_key** de la variable `lmi` se emplean para obtener las claves iniciales del modelo de lenguaje. Más tarde, el método **insert_query** se utiliza para insertar consultas a partir de las claves previamente adquiridas mediante el uso del método **next_keys**.

Ejemplo 2: Obtención de consultas en modo "bunch"

El siguiente fragmento de código se ejecuta de forma posterior a la inserción de consultas en código del apartado anterior:

```
result = lmi:get_queries()
for i=1,#result do
  k,p,b = result:get(i)
  scores[b]= scores[b] + p
end
lmi:clear_queries()
```

El método **get_queries** de la variable `lmi` se encarga de obtener los resultados de todas las consultas insertadas hasta el momento en la interfaz del modelo de lenguaje. Estos resultados se almacenan en el vector `result`, de forma que podemos iterar sobre este vector para obtener los campos de los resultados de las consultas. Después de usar estos datos, eliminamos todas las consultas insertadas en la interfaz hasta el momento con el método **clear_queries**.

Hay que tener en cuenta que el método **get** de la variable `result` devuelve el elemento i-ésimo del vector. Este método no debe confundirse con el método **get** de la interfaz de los modelos de lenguaje, que inserta y devuelve el resultado de un consulta.

Capítulo 4

Adaptación de skip-gramas a modelos conexionistas del lenguaje

En este capítulo primero hablaremos sobre el proceso de estimación de los modelos de skip-gramas en relación al proceso de los modelos de n-gramas estándar. Acto seguido, expondremos las herramientas que APRIL pone a nuestra disposición para la representación y modificación de datos. Algunas de estas herramientas ya se encontraban en APRIL antes de emprender este trabajo, mientras que otras han tenido que ser implementadas con el fin de desarrollarlo. Por último, detallaremos cómo se han adaptado los algoritmos de entrenamiento y evaluación de los NN LMs para considerar el caso de los Skipping NN LMs.

4.1. Modelos estadísticos de skip-gramas

Los modelos originales de skip-gramas (descritos en el capítulo 2) fueron propuestos para trabajar con modelos de n-gramas de orden superior. En este tipo de modelos, a medida que se expande el tamaño del contexto, también disminuye la probabilidad de haber visto exactamente el mismo contexto. No obstante, la probabilidad de haber visto un contexto parecido aumenta.

El procedimiento estándar para crear un modelo de skip-gramas es similar al procedimiento para crear un modelo de n-gramas. No obstante, en las muestras de entrenamiento para un modelo de skip-gramas, primero han de descartarse aquellas palabras del contexto que estén en las posiciones que previamente hemos marcado como "skippables" u omisibles. Este proceso puede llevarse a cabo sustituyendo aquellas palabras situadas en las posiciones marcadas por una máscara de bits (llamada "skipping mask" o máscara de skips) que contiene tantos bits como palabras haya en el contexto. Las palabras en las posiciones marcadas con un 1 son reemplazadas por una etiqueta especial, mientras que las palabras en las posiciones marcadas con un 0 no son manipuladas. En este trabajo usamos la etiqueta ⟨NONE⟩ para marcar las palabras omisibles.

Supongamos que estamos en medio del proceso de entrenamiento y llega una

muestra con el contexto "The little boy likes" y la máscara de skips 0101. El resultado de reemplazar las palabras por la etiqueta ⟨NONE⟩ sería "The ⟨NONE⟩ boy ⟨NONE⟩". Teóricamente, el entrenamiento de esta nueva muestra podría servir para predecir la probabilidad de la misma palabra en contextos como "The big boy eats" o "The new boy serves". En el futuro nos referiremos a las máscaras de bits según la representación decimal del código binario de la máscara, leído de derecha a izquierda. En este ejemplo, nos referiríamos a la máscara de skips 0101 como la máscara 10 en decimal.

A partir de ese proceso, el resto del algoritmo de estimación del modelo de skip-gramas es igual al de los modelos de n-gramas, que consiste en realizar el conteo y la normalización de frecuencias de los n-gramas.

4.2. Herramientas para la manipulación de datos

En esta sección analizaremos las herramientas que se han incluido en el toolkit APRIL para modificar los datos de entrada: Los dados, los tokens y los filtros. Vamos a explicar cada instrumento por separado, y en la siguiente sección explicaremos cómo se relacionan entre ellos.

4.2.1. Dados

La clase `Dice` representa un dado cargado de n caras. Decimos que es un dado cargado porque todas las caras del dado no tienen por qué tener la misma probabilidad de salir. Un dado se crea a partir de una lista con las caras que tiene el dado y otra lista, de la misma longitud, con las probabilidades de salir cada cara después de un lanzamiento de dado.

La clase `Dice` implementa un único método además de un "getter" para el atributo **outcomes**, el constructor y el destructor. El método al cual nos referimos es el método **thrown**, el cual simula el lanzamiento del dado a partir de un generador de números pseudoaleatorios [Matsumoto and Nishimura (1998)] y devuelve el resultado de lanzarlo.

Primeras distribuciones según orden de n-grama

El primer método que planteamos para insertar skips en los patrones de entrada fue una distribución cuya probabilidad de insertar skips se repartía de forma uniforme entre todas las posibles máscaras de skips. La tabla 4.1 muestra los porcentajes que se obtienen al agrupar las máscaras de skips según el número de skips que se insertan.

Esta es una distribución bastante sencilla para un primer intento, aunque también plantea otros problemas: La probabilidad de no insertar skips es demasiado baja, por lo que se sugirió fijar esta probabilidad p y repartir el resto, $(1-p)$, de forma uniforme entre todas las posibles máscaras de skips con al menos un skip.

En las pruebas preliminares de esta distribución, la probabilidad p de no insertar ningún skip se fijó al 35 %. Luego se argumentó que esta probabilidad era

Tabla 4.1: Dado el orden de los n-gramas, la probabilidad se reparte de forma uniforme entre todas las posibles máscaras.

# skips	bigrama	trigrama	4-grama	5-grama
0	50 %	25 %	12 %	6 %
1	50 %	50 %	37 %	25 %
2	–	25 %	37 %	37 %
3	–	–	12 %	25 %
4	–	–	–	6 %

demasiado pequeña para los modelos de menor orden, por lo que se aumentó al 50 %. Esta tendencia se conserva en la distribución que comentaremos a continuación. Los porcentajes que se obtienen siguiendo estas distribuciones se muestran en las tablas 4.2 y 4.3.

Tabla 4.2: Dado el orden de los n-gramas y la probabilidad de no insertar ningún skip $p = 35\,\%$, la probabilidad $(1 - p)$ se reparte de forma uniforme entre todas las posibles máscaras con al menos un skip.

# skips	bigrama	trigrama	4-grama	5-grama
0	35 %	35 %	35 %	35 %
1	65 %	43 %	28 %	17 %
2	–	22 %	28 %	26 %
3	–	–	9 %	17 %
4	–	–	–	4 %

Tabla 4.3: Dado el orden de los n-gramas y la probabilidad de no insertar ningún skip $p = 50\,\%$, la probabilidad $(1 - p)$ se reparte de forma uniforme entre todas las posibles máscaras con al menos un skip.

# skips	bigrama	trigrama	4-grama	5-grama
0	50 %	50 %	50 %	50 %
1	50 %	33 %	21 %	13 %
2	–	16 %	21 %	20 %
3	–	–	7 %	13 %
4	–	–	–	3 %

Otro problema era el hecho de que poner tantos skips como palabras tuviese el contexto tenía una probabilidad más baja a medida que aumentaba el orden de los n-gramas. Esta probabilidad puede calcularse como $\frac{(1-p)}{2^{n-1}-1}$, y equivale a un 3.33 % en un modelo con $p = 50\,\%$ y $n = 5$.

Estas formas de insertar skips no fueron empleadas finalmente para ningún experimento de esta memoria, aunque los problemas que planteaban trataron de solucionarse en la distribución multinomial que comentaremos a continuación.

Distribución multinomial según orden de n-grama

Otra posibilidad para realizar la inserción de skips es seguir una distribución multinomial según el orden de n-gramas del modelo. Esta forma de insertar skips ha sido empleada en algunos de los experimentos realizados para este trabajo, lo que la convierte en una de las más interesantes.

La probabilidad de que no se introduzca ningún skip es del 50 %, independientemente del orden de n-gramas del modelo. Esto significa que, en la mitad de ocasiones, el contexto se presentará como entrada a la red neuronal sin ser alterado de ninguna forma. En la otra mitad de ocasiones, se introducirá un determinado número de skips.

En un principio, la probabilidad de no insertar ningún skip puede parecer demasiado alta. Nuestra intención era evitar que el modelo tuviese unos malos resultados debido a una inserción excesiva de ruido (los skip-gramas pueden ser considerados una forma de ruido en el modelo). Más tarde, se comprobó que esta probabilidad era adecuada después de obtener unos resultados experimentales satisfactorios.

La probabilidad de poner algún skip (50 %) se reparte entre todas las combinaciones de máscaras de skips posibles. Si definimos la variable auxiliar k como:

$$k = \prod_{j=1}^{n-1} j \tag{4.1}$$

entonces las probabilidad de insertar un skip, a, se calcula como sigue:

$$a = \frac{0.5 \cdot k}{\sum_{i=1}^{n-1} \frac{k}{i}} \tag{4.2}$$

La probabilidad de insertar i skips es igual a la probabilidad de insertar un skip entre i.

En la tabla 4.4 puede observarse cómo se reparte esa probabilidad entre los distintos tipos de máscaras, que se agrupan según el número de skips que se introducen.

Tabla 4.4: Dado el orden de los n-gramas, el número de skips que se introduce sigue estas distribuciones multinomiales.

# skips	bigrama	trigrama	4-grama	5-grama
0	50 %	50 %	50 %	50 %
1	50 %	33 %	27 %	24 %
2	–	16 %	13 %	12 %
3	–	–	9 %	8 %
4	–	–	–	6 %

Cabe decir que esta probabilidad no se reparte uniformemente entre todas las máscaras de skips posibles, sino que la mayor parte se reparte entre las máscaras con un número bajo de skips. El siguiente script en Lua calcula cómo se reparte la probabilidad a partir n:

```
local skip_dice = nil
if use_skip then
  local N = ngram_value
  local zero_nones_prob = 0.5
  local k = factorial(N-1)
  local sum = 0
  for i=1,N-1 do sum = sum + k/i end
  local a = zero_nones_prob * k / sum
  local t = { zero_nones_prob }
  for i=1,N-1 do table.insert(t, a/i) end
  skip_dice = make_leveled_dice_mask(t)
end
```

Distribución uniforme para “skips viejos”

Nos referimos a las máscaras de skips viejos como aquellas máscaras que tienen un número cualquiera de bits a 1 por la izquierda y el resto de bits a 0. Los llamamos skips viejos porque la inserción progresiva de skips por la izquierda marca como omisibles las palabras más viejas del contexto.

La máscara de skips 1100 (3 en notación decimal inversa) en un modelo de 4-gramas es un ejemplo de máscara de skips viejos. La aplicación de dicha máscara de skips sobre el contexto “The little boy likes” daría como resultado el contexto “⟨NONE⟩ ⟨NONE⟩ boy likes”, sustituyendo las palabras “The” y “little” (las más viejas del contexto) por la etiqueta ⟨NONE⟩.

Una distribución uniforme reparte toda la probabilidad entre las posibles máscaras de skips viejos para el modelo de n-gramas de forma equitativa. La tabla 4.5 muestra cómo se reparte esta probabilidad para modelos de bajo orden.

Tabla 4.5: Dado el orden de los n-gramas, la probabilidad se reparte de forma uniforme entre todas las posibles máscaras de skips viejos. Las máscaras de skips se representan en notación decimal inversa.

máscara	bigrama	trigrama	4-grama	5-grama
0	50 %	33 %	25 %	20 %
1	50 %	33 %	25 %	20 %
3	–	33 %	25 %	20 %
7	–	–	25 %	20 %
15	–	–	–	20 %

Como ya veremos en el capítulo 5, las máscaras de skips viejos tienen una propiedad que los hace interesantes, por lo que se usan en algunos de los experimentos.

4.2.2. Tokens

Los `Tokens` se encuentran dentro del paquete con el mismo nombre del toolkit APRIL. Un token representa un conjunto de patrones, y son básicamente un *wrapper* sobre las clases `Matrix` y `DataSet` para facilitar la creación de herramientas que trabajan con distintos tipos de datos.

Así por ejemplo, los métodos pueden recibir la referencia a un token genérico como argumento o devolverlo como resultado. Ese token puede ser un conjunto números en coma flotante, números enteros o tokens. El usuario de los tokens puede recurrir a los método que permiten identificar el tipo de token con el que se está trabajando, los cuales son sobreescritos por las clases derivadas de la clase `Token`.

4.2.3. La clase TokenVector

La clase `TokenVector` añade las funcionalidades propias de un vector a la clase `Token`, con el objetivo de simplificar el tratamiento de patrones.

Los principales métodos que añade la clase `TokenVector` son:

- **push_back**: Añade un elemento a la cola del vector.
- **size**: Devuelve el tamaño del vector de elementos.
- **clear**: Elimina todos los elementos del vector.

La clase `TokenVector` se define como una plantilla, aunque implementa especializaciones para los principales tipos de datos en C++ y los tokens.

4.2.4. Filtros de datos

Las `Functions`, más conocidos como filtros de datos, se encuentran dentro del paquete `Functions` de APRIL. Un filtro se aplica sobre un conjunto de datos (contenido dentro de un token) para obtener otro conjunto de datos (también contenido por un token).

La clase SkipFunction

La clase `SkipFunction` representa un filtro de datos que inserta skips en un token. Esta clase hereda de la clase `FunctionInterface` y contiene tres atributos, que recibe en su constructor:

- Un dado: Una referencia a un objeto de la clase `Dice`.
- Un objeto *random*: Una referencia a un objeto de la clase `MTRand`.
- Un identificador: El valor con el que se marcan las palabras como omisibles, habitualmente el identificador de la palabra ⟨NONE⟩.

Estos tres atributos se utilizan en el método **calculate** para insertar skips de forma pseudoaleatoria en los contextos que se reciben como datos de entrada. Este proceso lo contaremos en detalle en la sección 4.3.1 de este mismo capítulo. Los otros dos métodos que implementa esta clase son:

- **getInputSize**: Devuelve 0, ya que el tamaño del token de entrada es variable.
- **getOutputSize**: Devuelve 0, ya que el tamaño del token de salida es variable.

La figura 4.1 muestra una posible salida después de aplicar el filtro de skips a un token de entrada.

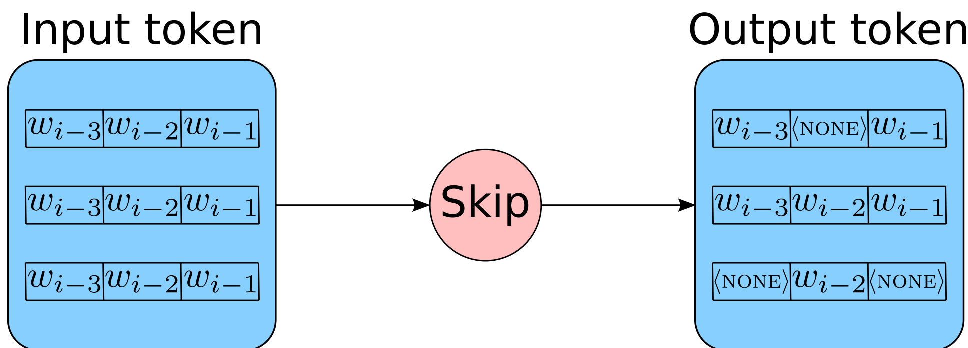

Figura 4.1: Un filtro de skips procesa un token de entrada con tres muestras. El token de salida contiene estas muestras modificadas pseudoaleatoriamente por el filtro, introduciendo la etiqueta ⟨NONE⟩ en algunas palabras de las muestras.

4.3. Adaptación de los algoritmos de entrenamiento y evaluación en NN LMs

El algoritmo Backpropagation para el entrenamiento de NN LMs no se diferencia en nada del algoritmo para entrenar los Skipping NN LMs. El algoritmo solamente ha de ser adaptado para modificar los datos de entrada, generar las tablas de constantes softmax necesarias e insertar adecuadamente los Skipping NN LMs en las jerarquías de modelos. El resto de este capítulo lo dedicaremos a explicar cómo hemos hecho estos cambios en APRIL.

4.3.1. Modificación probabilística de patrones

El primer paso para entrenar Skipping NN LMs es modificar los patrones de entrada a la red neuronal. En este entorno, los patrones de entrada son los

contextos de las consultas insertadas en las tablas hash que agrupan las consultas por claves y palabras. La función **computeKeysAndScores** de la clase `FeatureBasedLMInterface` se encarga de recopilar estos contextos y meterlos en un vector de tokens. Consecutivamente, se llama al método **calculate** del filtro asociado al modelo de lenguaje, pasando el vector de tokens como argumento. El siguiente código forma parte del método **calculate** de la clase `SkipFunction`:

```
// skip_mask es el resultado de tirar el dado
int size = word_tokens->size();
int skip_mask = dice->thrown(random);

// por cada palabra del contexto
for (int j = size-1; j >= 0; j--) {
  // si el bit esta a 1
  // marcar esa palabra como omisible
  if (skip_mask % 2)
    (*word_tokens)[j] = mask_value;
  skip_mask /= 2;
}
```

En este código, primero se lanza el dado para obtener un número entero, que corresponde a la cara del dado que queda mirando boca arriba. Ese valor es la máscara de skips que se aplicará sobre el patrón de entrada a la red. La máscara de skips se recorre examinando el último bit de la máscara y desplazando la máscara hacia la derecha. Si el último bit está a 1, entonces la palabra se reemplaza por la etiqueta ⟨NONE⟩. En caso contrario, la palabra se deja tal cual está.

El algoritmo para la modificación recorre todas las muestras, realizando una tirada de dado por cada una de ellas, y aplicando la máscara de skips sobre las palabras de la muesta.

4.3.2. Generación de tablas de constantes softmax

Uno de los inconvenientes de los NN LMs es la lentitud de estos modelos durante la fase de evaluación. Esta lentitud viene dada por la función de activación softmax asociada a la capa de salida de la red neuronal. El valor de las neuronas en esta capa de salida se calculan como:

$$o_i = \frac{exp(a_i)}{\sum_{i=1}^{|\bar{A}|} exp(a_i)} \tag{4.3}$$

en donde a_i es la activación de la i-ésima unidad de salida, y o_i es el valor de dicha salida tras aplicar la función softmax. El denominador de la ecuación 4.3 es un término de normalización para las activaciones de las neuronas de salida. Esta normalización restringe los valores de las salidas de la red neuronal a los del intervalo [0,1], lo cual permite interpretar estos valores como probabilidades de forma directa [Bishop et al. (1995)]. Otra ventaja de usar la función softmax a la

salida de la red frente a otras funciones con valores en el intervalos [0,1] (como la función sigmoide) es que la convergencia de la red es mucho más rápida.

Por tanto, la función de activación softmax es la ideal para establecer los valores de salida en un NN LM, aunque al mismo tiempo también es la culpable del peor problema de estos modelos. La constante de normalización softmax obliga a calcular todas las salidas de la red, aunque sólo nos interesen unas pocas. Este cálculo es costoso si tenemos en cuenta que, en un NN LMs estándar con una capa oculta de $|\bar{H}| = 128$ y una capa de salida $|\Omega| = 10K$ (un valor común en la literatura para aplicar la técnica de la Short-List), el número de conexiones asciende a $1.3M$. Este número resulta excesivo si tenemos en cuenta que ha de ejecutarse miles o cientos de miles de veces para guiar la búsqueda en un sistema de reconocimiento/traducción.

En estos modelos la entrada es simbólica, ya que las palabras son representadas por sus índices en una tabla con todas las palabras del vocabulario. Por esta razón, el número de entradas diferentes a la red y, por tanto, el número de constantes softmax asociadas a estas entradas es finito, aunque en la práctica no es viable almacenar todas estas constantes. En un NN LM de orden n, el número total de constantes diferentes es $|\Omega|^{n-1}$, ya que la entrada de la red neuronal recibe $n-1$ palabras del vocabulario Ω.

Una de las aportaciones de Francisco Zamora a los NN LMs es la técnica de precómputo de constantes softmax [Zamora et al. (2009)]. Esta técnica consiste en precalcular las constantes de normalización softmax más probables de aparecer durante la fase de evaluación y almacenarlas en una tabla. De esa manera, cuando se está calculando la probabilidad de una palabra en un contexto frecuente, tan sólo hemos de buscar la constante de normalización en la tabla. Un NN LM que incorpora esta técnica de precómputo de constantes de normalización softmax se denomina Fast NN LM.

En el caso de que hayamos calculado las constantes de normalización softmax para los contextos más probables, no encontraremos la constante de normalización para aquellos contextos que no se encuentren entre los más probables. En este caso se proponen dos formas de resolver este problema:

- Calcular la constante de normalización al vuelo para utilizarla después, pudiendo guardar dicha constante para el futuro si lo consideramos conveniente. A esta aproximación se la llama *on-the-fly* Fast NN LM, aunque equivale a un NN LM estándar si no fuese capaz de encontrarse en la tabla ninguna de las constantes de normalización softmax requeridas.

- Aplicar algún tipo de suavizado, como por ejemplo retroceder a un NN LM de menor orden o un modelo de n-gramas estándar. A esta aproximación se la conoce como Smoothed Fast NN LM, y consiste en tener una jerarquía de modelos en la cual los modelos de mayor orden delegan en los modelos de menor orden la tarea de calcular las probabilidades cuando no encuentran las constantes en su tabla.

Esta segunda opción es la que más nos interesa en este trabajo. A continuación, comentaremos cómo podemos emular las jerarquías de modelos con Fast NN LMs

usando un único modelo de Skipping NN LMs con distintas tablas de constantes softmax.

4.3.3. Skipping NN LMs en jerarquías de modelos

Un Smoothed NN LM representa una combinación de NN LMs y modelos de n-gramas estándar. Estos modelos no se combinan linealmente, sino que es una combinación jerárquica.

Existe un modelo de orden superior que trata de resolver todas las consultas que introducimos en el modelo. Aquellas consultas cuyas constantes no encuentra en la tabla de constantes de normalización softmax, las delega en uno de los modelos de la jerarquía de menor orden que sí pueda hacer frente a la consulta sin tener que calcular las constantes de normalización al vuelo. El Smoothed NN LM asigna la consulta a aquellos modelos de la jerarquía que sean capaces de responder a ella.

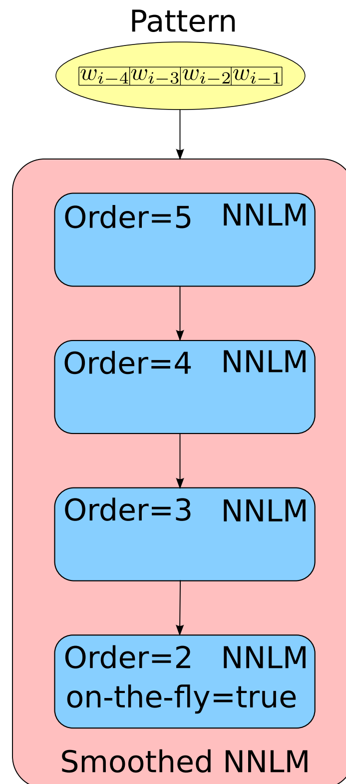

Figura 4.2: Esquema de una jerarquía de NN LMs de distinto orden. Los modelos de mayor orden delegan las consultas en modelos de menor orden cuando no pueden encontrar las constantes de normalización en sus tablas.

Normalmente, una jerarquía de modelos dispone de un Fast NN LM de 5-gramas como el modelo de orden superior. A ese modelo le siguen un Fast NN LM de 4-gramas, el cual elimina la palabra más vieja del contexto, y un Fast NN LM

de trigramas, el cual considera únicamente las dos palabras anteriores a la palabra cuya probabilidad se está consultando. Por último, se añade un modelo de bigramas a la cola de la jerarquía. Este modelo puede ser un modelo de bigramas estándar o un *on-the-fly* Fast NN LM de bigramas, los cuales son capaces de resolver todas las consultas que los modelos de mayor orden no pueden. El esquema de esta jerarquía de modelos se muestra en la figura 4.2.

En una jerarquía de modelos con Skipping NN LMs tenemos un diseño diferente. En primera instancia, tenemos un Fast NN LM de orden superior que trata de resolver las consultas que le llegan al modelo. Cuando este Fast NN LM de orden superior no encuentra las constantes de normalización softmax para alguna consulta, el modelo encarga la solución de la consulta a un NN LM del mismo orden. Este NN LM puede ser el mismo NN LM al cual se le había encargado la consulta originalmente, pero se diferencia del anterior NN LM en que la creación de su tabla de constantes softmax se hizo insertando un skip por la izquierda del contexto, eliminando así la palabra más vieja del modelo y emulando, por tanto, el retroceso a un NN LM de menor orden. Por supuesto, en la consulta también se reemplaza la palabra más vieja por la etiqueta ⟨NONE⟩, de forma que el Smoothed NN LM pueda detectar que el Skipping NN LM tiene la facultad de resolver esta consulta. El modelo inserta skips de forma incremental por la izquierda mientras ninguno de los Fast Skipping NN LMs (cuyas tablas de constantes de normalización softmax insertan $0, 1, \ldots, n-2$ skips por la izquierda del contexto) sea capaz de dar cuenta de la consulta sin recurrir al cálculo de la constante.

Un ejemplo de jerarquía de modelos con Skipping NN LMs es una jerarquía en donde todos los modelos son Fast Skipping NN LMs de 5-gramas. La tabla de constantes de normalización softmax del primer Fast Skipping NN LM se calcula de forma ordinaria. El precómputo de la tabla de constantes de normalización softmax del siguiente modelo se estima reemplazando la palabra más vieja del contexto por la etiqueta ⟨NONE⟩ en todas las muestras. La tabla para el siguiente modelo se genera igual, aunque en lugar de reemplazar únicamente la palabra más vieja por la etiqueta ⟨NONE⟩, se reemplazan un par. La tabla de constantes del último Fast Skipping NN LM de la jerarquía de modelos se crea insertando $n-2$ skips en el contexto de las muestras, es decir, tantos skips como la longitud del contexto menos uno. El esquema de esta nueva jerarquía con Skipping NN LMs se ilustra en la figura 4.3.

De esta forma, disponemos de una jerarquía de modelos cuya funcionalidad es equiparable a las jerarquías clásicas con Fast NN LMs, aunque en este caso tan sólo hemos entrenado una red neuronal y hemos generado varias tablas de constantes de normalización softmax. Esto reduce drásticamente la cantidad de tiempo necesaria para entrenar los modelos de la jerarquía.

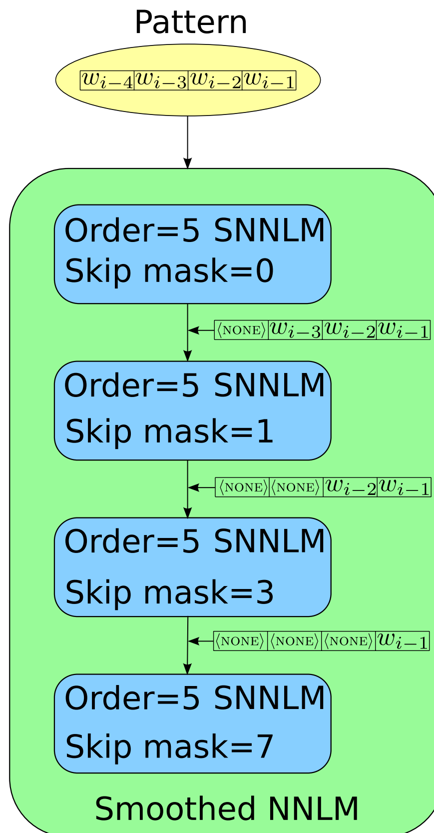

Figura 4.3: Esquema de una jerarquía de Skipping NN LMs del mismo orden. Cada Skipping NN LM maneja una tabla de constantes de normalización softmax diferente, generadas introduciendo en las muestras un número de skips por la izquierda variable.

Capítulo 5

Experimentación

En este capítulo nos centramos en el proceso de experimentación realizado en este proyecto. Empezaremos mostrando las primeras pruebas de Skipping NN LMs con el corpus LIBW, las cuales nos darán una idea sobre cómo las técnicas de skip-gramas afectan la PPL de los NN LMs o qué máscaras de skips funcionan mejor que otras. Los segundos ensayos exploran la posibilidad de emplear un único Skipping NN LM para emular una jerarquía de NN LMs entera, basándonos en los resultados de los experimentos realizados previamente y utilizando el corpus NC10 para este fin. En los últimos experimentos se repiten las mismas pruebas para una tarea de traducción automática.

5.1. Variación de la PPL en Skipping NN LMs

En este primer ensayo nuestro objetivo es realizar una experimentación exhaustiva con tal de observar cómo se comporta la PPL de los Skipping NN LMs en función de las máscaras aplicadas en la fase de evaluación. También pretendemos combinar los resultados de estas máscaras con los de los NN LMs sin skips o los modelos de n-gramas estándar, además de combinar los mejores resultados entre ellos.

5.1.1. Corpus

Esta parte experimental se ha llevado a cabo en una tarea basada en las transcripciones del conjunto de la IAM-DB [Marti and Bunke (2002)], de escritura manuscrita. Los conjuntos de entrenamiento para estimar los modelos de lenguaje son tomados del corpus LOB [Johansson et al. (1986)] (excluyendo las frases que aparecen en el conjunto de validación y test de la IAM-DB), Wellington [Bauer (1993)] y Brown [Francis and Kucera (1979)]. Nos referimos a la unión de todo este material como el corpus "LIBW". La tabla 5.1 muestra algunas estadísticas sobre este corpus.

Tabla 5.1: Estadísticas del corpus LIBW. Se muestra el número de líneas y palabras para cada parte, además del porcentaje de palabras fuera de vocabulario (Out-Of-Vocabulary words).

Corpora	# lines	# words	% OOV words
LOB + IAM training	174K	2.3M	–
Brown	114K	1.1M	–
Wellington	114K	1.1M	–
Total	402K	4.5M	–
IAM Validation	920	8762	3.18 %
IAM Test	2781	25424	2.86 %

5.1.2. Configuración experimental

Los NN LMs empleados en nuestros experimentos están completamente detallados en las publicaciones [Zamora et al. (2009), Zamora-Martínez et al. (2014)]. Estos modelos se han extendido para permitir la creación de Skipping NN LMs, tal y como se han descrito en la anterior sección.

Para entrenar tanto los NN LMs como los Skipping NN LMs, utilizamos el algoritmo Backpropagation y la función de entropía cruzada con regularización L_2 como función de la red. Basándonos en anteriores trabajos, hemos empleado una capa de proyección compuesta por 256 neuronas y una capa oculta compuesta por 200 neuronas.

La entrada de los Skipping NN LMs se altera estocásticamente, introduciendo la etiqueta ⟨NONE⟩ en cero o más posiciones arbitrarias de la capa de entrada. Para cada muestra de entrenamiento, el número de skips se muestrea a partir de una distribución multinomial. Esta distribución multinomial, que se muestra en la tabla 5.2, asigna una probabilidad del 50 % a los n-gramas sin skips, mientras que el 50 % restante se distribuye sobre los n-gramas con más de un skip siguiendo una tendencia hiperbólica. La particularidad de esta distribución es que, dado un número de skips, sus posiciones se distribuyen uniformemente.

Tabla 5.2: La distribución multinomial empleada en el entrenamiento de Skipping NN LMs para la primera parte experimental.

# skips	bigram	trigram	4-gram	5-gram
0	50 %	50 %	50 %	50 %
1	50 %	33 %	27 %	24 %
2	–	16 %	13 %	12 %
3	–	–	9 %	8 %
4	–	–	–	6 %

Una vez se hayan entrenado los Skipping NN LMs, tendremos un total de 2^{n-1} modelos de lenguaje, cada uno evaluado con una combinación diferente de skips. Estos modelos, a su vez, pueden ser combinados entre ellos para obtener nuevos modelos.

5.1.3. Resultados experimentales

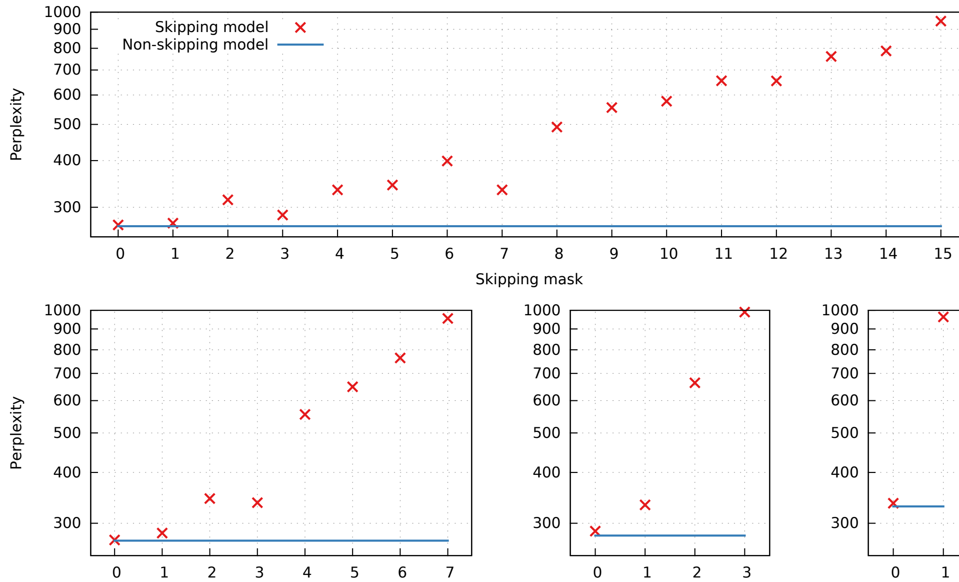

Figura 5.1: PPL de los Skipping NN LMs medida sobre el conjunto de validación variando la máscara de skips, junto con la PPL de referencia de las NN LMs sin skips. Los ejes verticales de las cuatro gráficas muestran la PPL, y la máscara de skips aparece en los ejes horizontales usando notación decimal inversa. Las gráficas de abajo corresponden, de derecha a izquierda, a los modelos de 4-gramas, trigramas y bigramas.

Para empezar, vamos a evaluar los resultados de PPL para el conjunto de validación en función de la máscara de skips (véase la figura 5.1). La PPL se ha calculado para cada una de las 2^{n-1} combinaciones de máscaras de skips. Nótese que la notación decimal usada en la gráfica es la conversión de la representación binaria invertida de las máscaras de skips, por lo que la máscara de bits debería leerse de derecha a izquierda. Recordemos que un 0 indica que no hay skips, mientras que un 1 indica que hay un skip en esa posición. De esta forma, el número skip 7 en un 5-grama se refiere a la representación binaria 1110, de forma que la probabilidad de la palabra i-ésima será calcula usando la palabra $i-1$ y tres etiquetas ⟨NONE⟩.

Los Skipping NN LMs completos tienen en cuenta la combinación de los diferentes modelos de lenguaje asociados a cada máscara de skips. Esta combinación se produce para cada orden de n-grama usando la herramienta `compute-best-mix` del toolkit SRILM [Stolcke et al. (2002)]. Los resultados obtenidos son comparables a los de los NN LMs estándar, aunque no son capaces de superarlos en ningún caso.

Como podemos observar en la figura 5.1, los Skipping NN LMs con una máscara de skips sin ⟨NONE⟩ tienen resultados tan buenos como los de los NN LMs estándar, mientras que la presencia de ⟨NONE⟩ degrada sus resultados. Este efecto

es más severo cuando más se acerca ⟨NONE⟩ a la palabra que queremos predecir.

Tabla 5.3: Resultados de PPL para IAM-DB en validación (izquierda) y test (derecha). Para los modelos con skip, la PPL se calcula con 0 skips y combinaciones de skips que simulan un modelo de orden inferior.

Model	Ngram order				
	1	2	3	4	5
5gr No skip	–	–	–	–	267
4gr No skip	–	–	–	272	–
3gr No skip	–	–	280	–	–
2gr No skip	–	330	–	–	–
5gr skip	946	334	286	272	269
4gr skip	955	337	284	273	–
3gr skip	990	333	287	–	–
2gr skip	963	336	–	–	–

Model	Ngram order				
	1	2	3	4	5
5gr No skip	–	–	–	–	309
4gr No skip	–	–	–	313	–
3gr No skip	–	–	319	–	–
2gr No skip	–	376	–	–	–
5gr skip	1019	378	326	311	309
4gr skip	1025	383	324	313	–
3gr skip	1045	377	327	–	–
2gr skip	1026	381	–	–	–

Con el objetivo de evaluar la futura capacidad de los Skipping NN LMs para simplificar los modelos presentados en [Zamora et al. (2009)], la tabla 5.3 muestra la PPL obtenida usando NN LMs sin skips y Skipping NN LMs en donde las máscaras de skips emulan modelos de igual o menor orden. Los valores de la primera fila de cada tabla son los valores de PPL para cada uno de los NN LMs. Las columnas restantes contienen los valores de cada uno de los Skipping NN LMs. El último valor de cada una de estas filas es la PPL obtenida para el conjunto sin alterar la entrada. Los valores previos se obtienen después de usar configuraciones de skips que emulan un modelo de menor orden. Por ejemplo, el valor correspondiente a un 4-grama para un 5-grama Skipping NN LM se obtiene después de usar una máscara de skips que reemplaza la palabra más alejada del contexto por la etiqueta ⟨NONE⟩. Para obtener el valor correspondiente a un trigrama reemplazamos la siguiente palabra del contexto también, y así hasta llegar al modelo de menor orden.

Podemos observar que los valores en las columnas de cada tabla son similares. Esto significa que los valores de PPL para cada uno de los NN LMs son similares a los obtenidos para los Skipping NN LMs, los cuales pueden calcular estos valores usando una configuración de skips adecuada. Por tanto, es posible entrenar un único Skipping NN LM para imitar la conducta de varios NN LMs. El retroceso en jerarquías de modelos con Skipping NN LMs es más simple y eficiente, ya que tan sólo necesitamos aplicar la máscara de skips adecuada para obtener la probabilidad del n-grama deseado.

5.2. Emulación de modelos de orden inferior con Skipping NN LMs

En este segunda fase de experimentación nuestro objetivo es corroborar los resultados obtenidos en la anterior fase y estudiar si un único Skipping NN LM es capaz de reemplazar un conjunto de NN LMs en una jerarquía de modelos. Para ello, construiremos múltiples jerarquías con NN LMs y Skipping NN LMs, y calcularemos la PPL de las jerarquías de cada tipo.

5.2.1. Corpus

Para los experimentos de esta sección usaremos el corpus News-Commentary, extraído de las tareas del *Workshop of Machine Translation* de los años 2008 y 2010 [Callison-Burch et al. (2008), Callison-Burch et al. (2010)].

Los conjuntos de entrenamiento y test para esta tarea serán los conjuntos de entrenamiento y test del News-Commentary 2010 (NC10). Como conjunto de validación utilizaremos el conjunto de desarrollo del News-Commentary 2008 (NC08). La tabla 5.4 muestra algunos datos de interés sobre este corpus.

Tabla 5.4: Estadísticas de la parte en inglés del corpus News. Se muestra el número de líneas y palabras para cada parte.

Corpora	# lines	# words
NC10 training	125.8K	2.9M
NC08 validation	2.0K	49.7K
NC10 test	2.5K	61.9K
Total	130.3K	3.0M

Una versión anterior de la herramienta APRIL se usó para entrenar NN LMs y evaluarlos en las tareas de traducción automática en el taller del año 2010 [Zamora-Martinez and Sanchis-Trilles (2010)].

5.2.2. Configuración experimental

Los NN LMs y Skipping NN LMs empleados en este experimento son los mismos que los descritos en la sección 5.1.2 de esta memoria.

Para entrenar ambos tipos de modelos, utilizamos el algoritmo Backpropagation y la función de entropía cruzada con regularización L_2 como función de la red. Cada NN LM y Skipping NN LM está compuesto por tres redes neuronales con capas de proyección de tamaños 128, 160 y 208 neuronas, y una capa oculta de 200 neuronas. Los pesos de las redes neuronales dentro del modelo son uniformes.

La entrada de los Skipping NN LMs se altera estocásticamente, introduciendo cero o más etiquetas $\langle\text{NONE}\rangle$ en las posiciones más alejadas de la palabra a predecir. Para cada muestra de entrenamiento, el número de skips que se introducen por la izquierda del contexto se muestrea a partir de una distribución uniforme. Esta distribución uniforme, que se muestra en la tabla 5.5 y que implementa el

tipo de dado descrito en la sección 4.2.1, asigna un porcentaje de probabilidad equitativo a cada cantidad de skips que puedan insertarse en el contexto.

Tabla 5.5: La distribución uniforme empleada en el entrenamiento de Skipping NN LMs para la segunda parte experimental.

mask	bigram	trigram	4-gram	5-gram
0	50 %	33 %	25 %	20 %
1	50 %	33 %	25 %	20 %
3	–	33 %	25 %	20 %
7	–	–	25 %	20 %
15	–	–	–	20 %

Después de la fase de entrenamiento, se construyen las tablas de constantes de normalización softmax para los NN LMs como se hace habitualmente y se ensamblan los modelos en una jerarquía. Para los Skipping NN LMs, las tablas de constantes de normalización se generan múltiples veces, tantas como modelos de menor orden se quieran emular, alterando los conjuntos de entrada para la generación de tablas de acuerdo a como se ha descrito en la sección 4.3.3.

5.2.3. Resultados experimentales

Vamos a evaluar los resultados de PPL para los conjuntos de validación y test obtenidos por las jerarquías de modelos con NN LMs y las jerarquías de modelos con Skipping NN LMs. Estos resultados se muestran en la tabla 5.6 y de forma gráfica en la figura 5.2.

Las jerarquías de modelos con NN LMs están compuestas por un NN LM de orden máximo, un NN LM de un orden menor al máximo, otro NN LM de un orden menor al anterior... Esta sucesión se repite hasta llegar a un NN LM de bigramas, el cual tiene activado el flag *on-the-fly* para permitir el cálculo de probabilidades al vuelo si no se encuentran las constantes de normalización softmax en su tabla.

Las jerarquías de modelos con Skipping NN LMs están compuestas por un único Skipping NN LM de orden máximo. El primer Skipping NN LM de la jerarquía emplea una tabla de constantes de normalización softmax generada de forma normal. El siguiente Skipping NN LM en la jerarquía es el mismo que el Skipping NN LM de orden máximo, aunque su tabla de constantes de normalización softmax se construye insertando un ⟨NONE⟩ por la izquierda de las secuencias usadas para la creación de la tabla, es decir, emulando un NN LM de orden inmediatamente inferior. Lo mismo ocurre en los posteriores Skipping NN LMs de la jerarquía, insertando cada vez un ⟨NONE⟩ adicional por la izquierda de los contextos usados en la generación de la tabla. El último Skipping NN LM de la jerarquía se sirve de una tabla de constantes que emula la funcionalidad de un bigrama NN LM.

Echando un vistazo a las gráficas de la figura 5.2 podemos advertir que las jerarquías de Skipping NN LMs siempre obtienen unos cuantos puntos más de PPL respecto a las jerarquías de NN LMs.

El posterior análisis de la tabla 5.6 desvela que estas ganancias en PPL nunca exceden los 5 puntos de PPL en jerarquías con un modelo de orden máximo

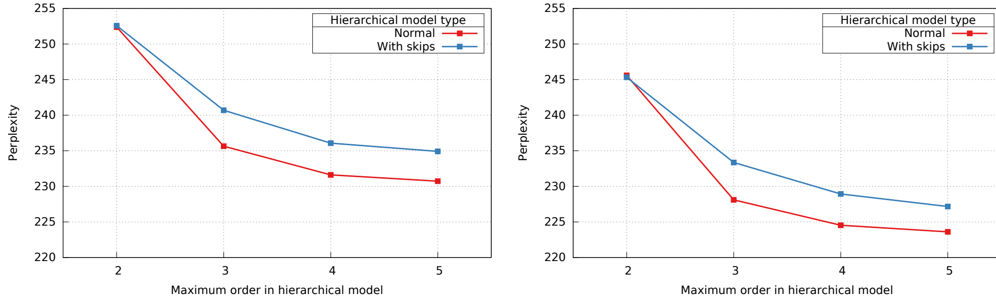

Figura 5.2: Resultados de PPL para las jerarquías de modelos con NN LMs (normal) y Skipping NN LMs (con skips) para el conjunto de validación (izquierda) y test (derecha).

Tabla 5.6: Resultados de PPL para las jerarquías de modelos con NN LMs (normal) y Skipping NN LMs (con skips) para los conjuntos de validación (izquierda) y test (derecha).

	Hierarchical model	
Max order	Normal	With skips
2	252	252
3	235	240
4	231	236
5	230	234

	Hierarchical model	
Max order	Normal	With skips
2	245	245
3	228	233
4	224	228
5	223	227

superior a 2. En ambos casos, las jerarquías con modelos de orden máximo igual a 5 son las que mejores resultados obtienen.

En la siguiente parte experimental integraremos las jerarquías de modelos de lenguaje en un sistema de traducción y evaluaremos su aportación a dicho sistema. De esta forma, tendremos una idea más cercana sobre el rendimiento de los Skipping NN LMs en un sistema de traducción real.

5.3. Rescoring de listas nbest

En esta parte experimental, las jerarquías de modelos desarrolladas en la anterior parte se integran en un sistema de traducción automática con el objetivo de evaluar su impacto en una tarea de traducción.

5.3.1. Corpus

Al igual que en los experimentos de la anterior sección, vamos a usar el corpus News-Commentary, sacado de las tareas del *Workshop of Machine Translation*. En esta ocasión usaremos datos de las ediciones de los años 2008, 2009 y 2010 [Callison-Burch et al. (2008), Callison-Burch et al. (2009), Callison-Burch et al.

(2010)] de este taller.

En concreto, evaluaremos los resultados del sistema sobre los corpus de desarrollo de las ediciones 2008 y 2009, mientras que de la edición del 2010 usaremos el corpus de test. De cada uno de estos corpus usaremos los ficheros en español e inglés. La tabla 5.7 muestra datos que pueden ser de interés sobre los ficheros de este corpus.

Tabla 5.7: Estadísticas de la parte bilingüe del corpus News. Se muestra el número de líneas y palabras para cada parte.

	Spanish		English	
Corpora	# lines	# words	# lines	# words
News2008	2.0K	52.6K	2.0K	49.7K
News2009	2.5K	68.0K	2.5K	65.6K
News2010	2.5K	65.5K	2.5K	61.9K

5.3.2. Configuración experimental

El decodificador de traducción empleado en esta parte experimental es APRIL, aunque utiliza modelos basados en segmentos entrenados con Moses [Hoang et al. (2007)]. Este decodificador, al cual nos referimos como APRIL-PB, usará las jerarquías de modelos que se han creado para los experimentos sección 5.2 como modelos de lenguaje para el idioma inglés.

El modelo de lenguaje se utiliza para repuntuar las hipótesis de una lista generada por el proceso de búsqueda de APRIL-PB (listas n-best). La nueva puntuación se combina con las puntuaciones de las listas n-best en un proceso de maximización para conseguir la mejor hipótesis.

En esta tarea, el sistema de traducción se emplea para traducir los ficheros en español de cada corpus al inglés. Esta traducción se efectua por cada tipo de jerarquía (con NN LMs o Skipping NN LMs) y por cada orden máximo de modelo la jerarquía. Usando los ficheros de referencia en inglés, calculamos el BLEU y TER de las traducciones efectuadas por el sistema.

5.3.3. Resultados experimentales

Los resultados de esta tarea se muestran en la tabla 5.8. Si tomamos como referencia los valores de las jerarquías de modelos con un orden máximo igual a 5 (los más propensos a ser usados en un sistema real a causa de sus resultados en PPL), observamos unos valores casi idénticos en las traducciones de los corpus del 2009 y 2010. Solamente el corpus del año 2008 da unos resultados peores para la jerarquía de Skipping NN LMs, aunque esta diferencia no sobrepasa los 0.11 puntos de BLEU.

La tabla 5.9 muestra los resultados de los decodificadores Moses y APRIL-PB realizados en [Zamora-Martínez (2012)] como líneas base para sus experimentos de traducción automática. En ambos corpus (2009 y 2010) obtenemos mejoras significativas respecto a estas líneas de base para ambas jerarquías.

Tabla 5.8: Resultados de BLEU y TER con los conjuntos de los años 2008, 2009 y 2010 del corpus News para los dos tipos de jerarquía de modelos (**N**ormal y **S**kipping) con diferente orden máximo de modelo en la jerarquía.

	2008				2009				2010			
	BLEU		TER		BLEU		TER		BLEU		TER	
n	N	S	N	S	N	S	N	S	N	S	N	S
2	19.9	19.9	63.4	63.5	20.7	20.7	60.0	60.1	22.9	22.9	57.5	57.5
3	20.2	20.1	63.3	63.5	21.2	21.0	59.8	59.9	23.2	23.0	57.4	57.5
4	20.3	20.2	63.2	63.3	21.1	21.1	59.8	59.8	23.3	23.2	57.3	57.3
5	20.3	20.2	63.2	63.2	21.0	21.0	59.8	59.8	23.2	23.1	57.4	57.4

Tabla 5.9: Líneas base establecidas por los decodificadores Moses y APRIL-PB para los corpus del año 2009 y 2010.

	2009		2010	
System	BLEU	TER	BLEU	TER
Moses	20.4	60.3	22.6	57.8
April-PB	20.6	60.3	22.7	57.8

Teniendo en cuenta que las diferencias entre jerarquías no son significativas en ninguno de los tres corpus usados en esta tarea, es bastante seguro afirmar que el uso de Skipping NN LMs en lugar de NN LMs en las jerarquías de modelos no afecta a sus resultados. No obstante, el tiempo y los recursos dedicados a entrenar un conjunto de NN LMs sí que son mayores al tiempo y los recursos dedicados a entrenar un único Skipping NN LM.

Si bien es cierto que es posible entrenar varias redes neuronales al mismo tiempo (para NN LMs que usen una combinación de redes neuronales), entrenar todas las redes de la jerarquía en un solo ciclo de entrenamiento es bastante difícil, a no ser que se utilice un clúster de máquinas para este cometido. En cambio, un Skipping NN LM seguramente podrá entrenar todas las redes de las cuales se compone en un único ciclo de entrenamiento.

La complejidad temporal de generar las tablas de constantes de normalización softmax se aproxima en ambos casos al coste de generar la tabla para el modelo de máximo orden en la jerarquía. Esto significa que, aunque las tablas de constantes de modelos de menor orden para jerarquías de NN LMs sí que cuestan temporalmente menos de crear, el tiempo extra que gastemos en crear las tablas de constantes de menor orden para jerarquías de Skipping NN LMs no será notable si empleamos un procesador con varios núcleos.

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones extraídas a partir de los experimentos realizados y las contribuciones de este trabajo a las tareas del Procesamiento del Lenguaje Natural relacionadas con los modelos de lenguaje. Luego hablamos de las publicaciones relacionadas con este proyecto, con la intención de poder valorar el impacto de nuestra investigación adecuadamente. Por último, se sugieren nuevas vías de trabajo que surgen a partir de este proyecto.

6.1. Conclusiones

A lo largo de este trabajo, hemos desarrollado una herramienta que permita combinar los modelos de skip-gramas con los NN LMs. El resultado de esta combinación son los Skipping NN LMs que, después de un exhaustivo proceso de experimentación, han probado tener sus ventajas.

Antes de entrar en detalles, vamos a repasar los objetivos que nos marcábamos en la sección 1.4, al inicio de este trabajo:

- *Elaborar un diseño de la herramienta que sea eficiente, tanto espacial como temporalmente.*
- *Confeccionar un diseño que permita la adición y extensión de funcionalidades de una forma sencilla.*
- *Estudiar el efecto que el entrenamiento con skip-gramas tiene sobre la PPL obtenida en los conjuntos de evaluación.*
- *Considerar qué otros beneficios pueden tener las técnicas de skip-gramas en los NN LMs.*

Podemos marcar el primer elemento de nuestra lista de objetivos sin dudar que APRIL usa las estructuras de datos necesarias para implementar sus algoritmos lo más eficientemente posible. Lo hemos visto en la clase `BunchHashedLMInterface`, en donde empleamos un conjunto de tablas hash y estructuras `KeyScoreMultipleBurdenTuple` para agrupar las consultas insertadas hasta el momento y evitar repetir esas consultas posteriormente. Lo hemos visto también en la búsqueda de claves en la

clase `HistoryBasedLMInterface`, en donde los recorridos por el trie de palabras se realizan con el mínimo número de pasos. El código puede acelerarse todavía más si compilamos la versión de APRIL con la librería MKL de Intel, la cual optimiza el código para arquitecturas Intel.

El segundo elemento de la lista también lo podemos etiquetar como "cumplido". La jerarquía de clases para implementar los modelos de lenguaje, mostrada en el capítulo 3 de esta memoria, está hecha con el objetivo de que se puedan añadir y extender funcionalidades fácilmente gracias a los mecanismos de herencia y las clases base ya implementadas. Otras clases, como por ejemplo los filtros, también nos permiten crear nuevos filtros a partir de una interfaz ya declarada en la clase `FunctionInterface`. Todas estas clases, implementadas en el lenguaje C++, pueden ser usadas en el lenguaje Lua gracias al "binding", lo cual permite una rápida creación de scripts para efectuar experimentos.

El tercer elemento de la lista lo podemos marcar después de repasar toda la experimentación efectuada a lo largo del trabajo. Los primeros experimentos ya obtenían la PPL para todas las posibles máscaras de skips y distintos valores de n. Estos resultados se combinaban entre ellos y los de otro tipo de modelos para observar si se conseguían mejoras significativas. La segunda experimentación se realiza después de observar que la PPL de los Skipping NN LMs con una máscara de skips viejos es similar a la de los NN LMs de una jerarquía de modelos. Esta experimentación demuestra que, en efecto, los Skipping NN LMs de una jerarquía de modelos pueden imitar la conducta de una jerarquía de NN LMs estándar. Esta declaración se realiza en base a los resultados en PPL de la segunda fase de experimentación y los resultados en BLEU y TER del sistema de traducción automática con estas jerarquías de modelos en la tercera fase de experimentación.

Los resultados finales muestran que un Skipping NN LM puede ser usado en jerarquías de modelos con un conjunto de tablas de constantes de normalización softmax para emular el comportamiento de las jerarquías de modelos con NN LMs estándar. Esto significa que solamente es necesario entrenar un Skipping NN LM del mayor orden que pretendamos incluir en la jerarquía de modelos y generar tantas tablas de constantes de normalización softmax como modelos de menor orden queramos emular en esa jerarquía. Si tenemos en cuenta que los Skipping NN LMs pueden ahorrar una cantidad considerable de tiempo para entrenar las redes neuronales de una jerarquía de modelos, entonces podemos considerar que hemos logrado el cuarto objetivo que nos proponíamos al principio de este trabajo.

6.2. Publicaciones

Es recomendable que la presentación de un Trabajo Fin de Máster orientado a la investigación incluya al menos una publicación científica. La escasez de eventos durante el verano ha hecho que nos tengamos que conformar con presentar nuestra propuesta en un solo congreso.

A mediados de Julio enviamos el artículo *First steps towards Skipping NNLMs* al congreso internacional IberSPEECH 2014 (véase el apéndice C), aunque la aceptación del artículo quedará determinada después de hacer la entrega de esta memoria. No obstante, informaremos sobre el estado del artículo en la presentación

del TFM. .

El artículo empieza con una breve introducción a los modelos de lenguaje y un resumen sobre el estado del arte en modelos de skip-gramas y NN LMs. Después se plantean los Skipping NN LMs como una combinación de ambos modelos. A continuación se incluye la experimentación efectuada con el corpus LIBW, la cual está expuesta en la sección 5.1. Por último, se presentan las conclusiones extraídas a partir de los primeros experimentos con Skipping NN LMs y se plantean pruebas adicionales para estudiar el impacto de estos modelos.

Los resultados de emulación de modelos de orden inferior con Skipping NN LMs aplicado a tareas de traducción, presentados en las secciones 5.2 y 5.3, serán enviados a algún congreso del área. Adicionalmente, esperamos poder presentar nuevos avances en futuros eventos o revistas especializadas. Estos avances podemos alcanzarlos a partir de experimentar más con estos modelos o explorar algunas de las propuestas listadas en la sección 6.3.

6.3. Ampliaciones futuras

En este trabajo hemos observado que el uso de Skipping NN LMs puede tener sus ventajas a pesar de ser una modificación de los ya existentes NN LMs. Esto refleja el hecho de que los NN LMs son unos modelos relativamente jóvenes, capaces de mejorar su eficiencia o sus resultados mediante la alteración de alguna de sus partes.

En este trabajo nos hemos centrado en la capa de entrada de los NN LMs. Los contextos de las muestras son modificados para incluir uno o varios símbolos $\langle$NONE$\rangle$ que marcan la palabra como omisible, modificando así los patrones de entrada de la red. También podríamos modificar la entrada de otras formas, como por ejemplo, "cortando" las palabras que antes marcábamos como omisibles e insertando el resto de palabras como entrada al modelo. No es difícil dar con otras alteraciones del conjunto de entrada para estudiar cómo afectarían al rendimiento de los NN LMs.

Y de la misma manera que es posible pensar en modificaciones a la altura de la capa de entrada, también es posible hacer lo mismo sobre las otras capas u otros elementos de las redes neuronales. En definitiva, los NN LMs son unos modelos sobre los cuales todavía queda mucho por explorar.

A continuación presentamos una breve lista de posibles ampliaciones futuras sobre los Skipping NN LMs y otras ideas que han ido surgiendo a lo largo del proyecto en relación a los NN LMs:

- Uso de jerarquías de Skipping NN LMs para otras tareas: Del mismo modo que se han comparado las jerarquías de modelos con NN LMs y Skipping NN LMs mediante una tarea de traducción automática en la sección 5.3, también podríamos compararlas en otras tareas de Reconocimiento del Habla o Reconocimiento de Escritura, para estudiar el impacto de las jerarquías con Skipping NN LMs en las tasas de reconocimiento de estos sistemas.

- Estudio sobre la PPL según la frecuencia de las palabras: Una de las primeras razones por las que decidimos adaptar las técnicas de skip-gramas a los NN LMs era comprobar si estas técnicas reforzaban la generalización de los NN LMs sobre los datos, a pesar de ser unos modelos que ya de por sí tienen una gran capacidad para hacerlo, gracias a la codificación distribuida de las palabras.

  Después de realizar los primeros experimentos, observamos que las ganancias de PPL eran leves para máscaras de skips razonables, es decir, para máscaras de skips con un número moderado de *unos*. En el futuro, esperamos poder determinar cómo los skips afectan a la probabilidad de las palabras en función de su frecuencia. Nuestra conjetura es que los skips afectan de manera negativa la PPL de las palabras más frecuentes (las palabras más frecuenta se vuelven menos probables), mientras que las palabras menos frecuentes se ven beneficiadas por el uso de esta técnica (las palabras menos frecuentes seguramente son reemplazables por otras en contextos parecidos). Desgraciadamente, esto afecta negativamente a la PPL del modelo en general.

- Modelos con caché orientados a consultas masivas: Los modelos de lenguaje con caché introducen información adicional sobre el contexto del discurso que la máquina está procesando. Esta información puede ser relevante al realizar consultas sobre un documento de una temática determinada o un diálogo espontáneo, en donde la repetición de algunas frases será bastante común y podría influir positivamente en las predicciones del modelo de lenguaje.

  La extensión de los NN LM con caché [Zamora-Martinez et al. (2012)] (la figura 6.1 muestra su arquitectura) podría ser adaptada también a nuestro diseño orientado a consultas masivas. El diseño de la herramienta APRIL nos permite crear un nuevo filtro para generar los contenidos de la caché y colocar esos contenidos en un token.

  No obstante, existe toda una problemática relacionada con la generación simultánea de los contenidos de la caché que debería estudiarse en un proyecto aparte.

- Alternativas a la función softmax en la capa de salida: Como comentábamos en la sección 4.3.2, la función de activación softmax es la ideal para establecer los valores de salida en los NN LMs, aunque también es la culpable de su lentitud en la fase de evaluación, ya que es necesario calcular todas las salidas para obtener la constante de normalización softmax.

  En la actualidad, se están buscando alternativas a la función softmax en la capa de salida o formas de hacer viable esta función sin causar problemas de eficiencia en el NN LM. Algunas de las aproximaciones que queremos explorar son las siguientes:

  - NN LMs con jerarquías de palabras: Una propuesta para evitar el cálculo de todas las salidas es reemplazar la representación no estructurada del vocabulario de salida por un árbol que representa la

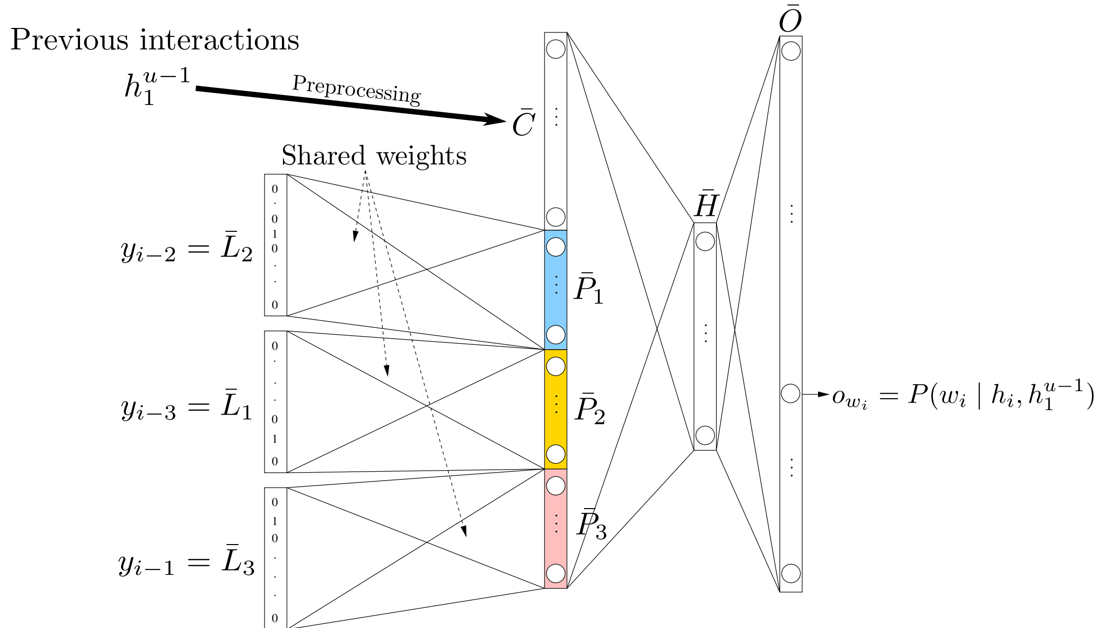

Figura 6.1: La arquitectura de un 4-grama NN LMs con caché durante la fase de entrenamiento. El vector $\bar{C}$ es una bolsa de palabras que incluye información acerca del contexto [Zamora-Martínez (2012)].

jerarquía de las palabras de este vocabulario [Mikolov et al. (2011), Mnih and Hinton (2009), Bengio et al. (2006)].

La representación jerárquica podría distribuir las palabras por categorías, campos semánticos, etc. Esta representación jerárquica puede obtener de varias formas, aunque la principal propuesta es utilizar un algoritmo de clustering para crear grupos de palabras.

De esta forma, el problema de calcular las constantes softmax se reduce a calcular las salidas de la categoría en la cual se encuentra la palabra para la cual estamos realizando la consulta. Esto reduce notablemente el tiempo de cómputo requerido, aunque del mismo modo también aumenta la complejidad del modelo.

- NN LMs con función de activación sin normalización en la capa de salida: Otra alternativa es sustituir la función softmax por cualquier otra función que no precise una normalización en su cálculo [Collobert and Weston (2008), Huang et al. (2012)]. Estas técnicas también requieren el uso de un algoritmo de clustering para relacionar aquellas palabras que sean semánticamente parecidas.

- Aproximación de probabilidades a un parámetro α: Una de nuestras ideas para librarnos de la función softmax en la capa de salida es utilizar una función alternativa que no requiera ningún tipo de normalización. No obstante, el reemplazamiento de la función softmax

en la capa de salida conlleva la adaptación de algún mecanismo para controlar los valores de salida en la misma.

Nuestra propuesta sería utilizar una función que aproxime los valores de salida a un valor arbitrario α. Esto podría conseguirse, por ejemplo, añadiendo un término que penalice el distanciamiento de los valores de salida del valor α.

De momento, esta propuesta es solamente un esbozo, por lo que habría que ver cómo incorporar esta técnica en nuestro diseño y realizar los experimentos necesarios para valorar su efectividad, al contrario que las anteriores técnicas, las cuales ya han probado su efectividad en los artículos que se citan.

Bibliografía

Arisoy, E., Sainath, T. N., Kingsbury, B., and Ramabhadran, B. (2012). Deep neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 20–28. Association for Computational Linguistics.

Bahl, L., Brown, P., de Souza, P., and Mercer, R. (1989). A tree-based statistical language model for natural language speech recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(7):1001–1008.

Bauer, L. (1993). Manual of Information to Accompany The Wellington Corpus of Written New Zealand English. Technical report, Department of Linguistics, Victoria University, Wellington, New Zealand.

Bellman, R. and Corporation, R. (1957). *Dynamic Programming*. Rand Corporation research study. Princeton University Press.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *JOURNAL OF MACHINE LEARNING RESEARCH*, 3:1137–1155.

Bengio, Y., Schwenk, H., Senécal, J.-S., Morin, F., and Gauvain, J.-L. (2006). Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer.

Berger, A. L., Pietra, V. J. D., and Pietra, S. A. D. (1996). A maximum entropy approach to natural language processing. *Computational linguistics*, 22(1):39–71.

Bishop, C. M. (1996). *Neural networks for pattern recognition*. Oxford University Press.

Bishop, C. M. et al. (1995). Neural networks for pattern recognition.

Blitzer, J., Weinberger, K., Saul, L., and Pereira, F. (2005). Hierarchical distributed representations for statistical language modeling. In Saul, L. K., Weiss, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 18*, Cambridge, MA. MIT Press.

Brown, P. F., Pietra, V. J. D., Mercer, R. L., Pietra, S. A. D., and Lai, J. C. (1992). An estimate of an upper bound for the entropy of english. *Comput. Linguist.*, 18(1):31–40.

Callison-Burch, C., Koehn, P., Monz, C., Peterson, K., and Zaidan, O., editors (2010). *Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*. Association for Computational Linguistics, Uppsala, Sweden.

Callison-Burch, C., Koehn, P., Monz, C., and Schroeder, J., editors (2009). *Proceedings of the Fourth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Athens, Greece.

Callison-Burch, C., Koehn, P., Monz, C., Schroeder, J., and Fordyce, C. S., editors (2008). *Proceedings of the Third Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Columbus, Ohio.

Castro, M. J. and Prat, F. (2003). New directions in connectionist language modeling. In *Computational Methods in Neural Modeling*, pages 598–605. Springer.

Castro, M. J., Prat, F., and Casacuberta, F. (1999). Mlp emulation of n-gram models as a first step to connectionist language modeling. In *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, volume 2, pages 910–915. IET.

Chen, S. F. and Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics.

Chen, S. F. and Rosenfeld, R. (2000). A survey of smoothing techniques for me models. *Speech and Audio Processing, IEEE Transactions on*, 8(1):37–50.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM.

De La Briandais, R. (1959). File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM.

Della Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing features of random fields. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(4):380–393.

Della Pietra, S., Della Pietra, V., Mercer, R. L., and Roukos, S. (1992). Adaptive language modeling using minimum discriminant estimation. In *Proceedings of the workshop on Speech and Natural Language*, pages 103–106. Association for Computational Linguistics.

Duda, R. O., Hart, P. E., and Stork, D. G. (1999). *Pattern classification*. John Wiley & Sons,.

Emami, A. and Mangu, L. (2007). Empirical study of neural network language models for arabic speech recognition. In *Automatic Speech Recognition & Understanding, 2007. ASRU. IEEE Workshop on*, pages 147–152. IEEE.

España, S., Castro, M., Gorbe, J., and Zamora, F. (2011). Improving offline handwritten text recognition with hybrid hmm/ann models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(4):767–779.

España, S., Zamora, F., Castro, M., and Gorbe, J. (2007). Efficient BP Algorithms for General Feedforward Neural Networks. In *Bio-inspired Modeling of Cognitive Tasks*, volume 4527, pages 327–336. Springer.

Francis, W. and Kucera, H. (1979). Brown Corpus Manual, Manual of Information to accompany A Standard Corpus of Present-Day Edited American English. Technical report, Department of Linguistics, Brown University, Providence, Rhode Island, US.

Goodman, J. T. (2001). A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434.

Hidalgo, J. L., España, S., Castro, M. J., and Pérez, J. A. (2005). Enhancement and cleaning of handwritten data by using neural networks. In Marques, J. S. et al., editors, *Pattern Recognition and Image Analysis*, volume 3522, pages 376–383. Springer-Verlag. ISSN 0302-9743.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA.

Hinton, G. E. (1989). Connectionist learning procedures. *Artificial intelligence*, 40(1):185–234.

Hoang, H., Birch, A., Callison-burch, C., Zens, R., Aachen, R., Constantin, A., Federico, M., Bertoldi, N., Dyer, C., Cowan, B., Shen, W., Moran, C., and Bojar, O. (2007). Moses: Open source toolkit for statistical machine translation. pages 177–180.

Huang, E. H., Socher, R., Manning, C. D., and Ng, A. Y. (2012). Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 873–882. Association for Computational Linguistics.

Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., Lee, K.-F., and Rosenfeld, R. (1992). *The SPHINX-II speech recognition system: an overview*. Citeseer.

Jelinek, F. (1980). Interpolated estimation of markov source parameters from sparse data. *Pattern recognition in practice*, pages 381–397.

Jelinek, F. (1997). *Statistical methods for speech recognition*. MIT press.

Johansson, S., Atwell, E., Garside, R., and Leech, G. (1986). The Tagged LOB Corpus: User's Manual. Technical report, Norwegian Computing Centre for the Humanities, Bergen, Norway.

Jurafsky, D. (2012). *Language Modeling: Introduction to N-grams*. Coursera.

Kam, M. and Guez, A. (1987). On the probabilistic interpretation of neural network behavior. In *American Control Conference, 1987*, pages 1968–1972. IEEE.

Katz, S. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 35(3):400–401.

Klakow, D. and Peters, J. (2002). Testing the correlation of word error rate and perplexity. *Speech Communication*, 38(1–2):19 – 28.

Llorens Piñana, D. (2000). *Suavizado de automatas y traductores finitos estocasticos*. Tesis doctoral en informática, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

Luong, M.-T., Socher, R., and Manning, C. D. (2013). Better word representations with recursive neural networks for morphology. In *CoNLL*, Sofia, Bulgaria.

Marti, U. V. and Bunke, H. (2002). The IAM-database: an English sentence database for offline handwriting recognition. 5:39–46.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30.

Miikkulainen, R. and Dyer, M. G. (1991). Natural language processing with modular pdp networks and distributed lexicon. *Cognitive Science*, 15(3):343–399.

Mikolov, T., Deoras, A., Povey, D., Burget, L., and Cernocky, J. (2011). Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE.

Mnih, A. and Hinton, G. E. (2009). A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088.

Moody, J., Hanson, S., and Lippmann, R. (1995). A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957.

Nakamura, M. and Shikano, K. (1988). A study of english word category prediction based on neural networks. *The Journal of the Acoustical Society of America*, 84(S1):S60–S61.

Ney, H., Essen, U., and Kneser, R. (1994). On structuring probabilistic dependences in stochastic language modelling. *Computer Speech and Language*, 8(1):1–38.

Paccanaro, A. and Hinton, G. E. (2000). Extracting distributed representations of concepts and relations from positive and negative propositions. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 2, pages 259–264. IEEE.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Park, J., Liu, X., Gales, M. J., and Woodland, P. C. (2010). Improved neural network based language modelling and adaptation. In *INTERSPEECH*, pages 1041–1044.

Park, J. C. and Abusalah, S. T. (1997). Maximum entropy: a special case of minimum cross-entropy applied to nonlinear estimation by an artificial neural network. *Complex Systems*, 11(4):289–308.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks.* Cambridge University Press.

Rojas, R. (1996). *Neural Networks - A Systematic Introduction.* Springer-Verlag.

Rosenberg, D. S., Klein, D., and Taskar, B. (2012). Mixture-of-parents maximum entropy markov models. *arXiv preprint arXiv:1206.5261.*

Rosenfeld, R. (2000). Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278.

Rosenfeld, R. (2005). *Adaptive statistical language modeling: A maximum entropy approach.* PhD thesis, IBM.

Rosenfeld, R., Wasserman, L., Cai, C., and Zhu, X. (1999). Interactive feature induction and logistic regression for whole sentence exponential language models. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 231–236.

Schmidhuber, J. and Heil, S. (1996). Sequential neural text compression. *Neural Networks, IEEE Transactions on*, 7(1):142–146.

Schwenk, H. (2007). Continuous space language models. *Computer Speech & Language*, 21(3):492–518.

Schwenk, H., Dchelotte, D., and Gauvain, J.-L. (2006). Continuous space language models for statistical machine translation. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 723–730. Association for Computational Linguistics.

Schwenk, H. and Gauvain, J.-L. (2002). Connectionist language modeling for large vocabulary continuous speech recognition. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 1, pages I–765. IEEE.

Schwenk, H. and Koehn, P. (2008). Large and diverse language models for statistical machine translation.

Smaıli, K., Jamoussi, S., Langlois, D., and Haton, J.-P. (2004). Statistical feature language model. In *Proc. ICSLP*.

Snover, M., Dorr, B., Schwartz, R., Micciulla, L., and Makhoul, J. (2006). A study of translation edit rate with targeted human annotation. In *Proceedings of association for machine translation in the Americas*, pages 223–231.

Stolcke, A. et al. (2002). Srilm-an extensible language modeling toolkit. In *INTERSPEECH*.

Tortajada, S. and Castro, M. (2006). Diferentes aproximaciones a la codificación del vocabulario en modelado de lenguaje conexionista. *Actas de las IV Jornadas de Tecnologías del Habla (págs. 59-63). Valencia-España: UPV.*

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269.

Xu, P. and Jelinek, F. (2004). Random forests in language modeling. In *Proc. EMNLP*.

Zamora, F., Castro, M., and España, S. (2009). Fast Evaluation of Connectionist Language Models. In Cabestany, J., Sandoval, F., Prieto, A., and Corchado, J. M., editors, *International Work-Conference on Artificial Neural Networks*, volume 5517, pages 33–40. Springer. 10th International Work-Conference on Artificial Neural Networks, IWANN 2009, Salamanca, Spain, June 10-12, 2009. Proceedings.

Zamora, F., España, S., and Castro, M. J. (2007). Behaviour-based clustering of neural networks applied to document enhancement. In *IWANN'07: Proceedings of the 9th international work conference on Artificial neural networks*, pages 144–151, Berlin, Heidelberg. Springer-Verlag.

Zamora-Martínez, F. (2005). Implementación eficiente del algoritmo de retropropagación del error con momentum para redes hacia delante generales. Proyecto Final de Carrera.

Zamora-Martínez, F. (2012). *Aportaciones al modelado conexionista de lenguaje y su aplicación al reconocimiento de secuencias y traducción automática.* Tesis doctoral en informática, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

Zamora-Martinez, F., Espana-Boquera, S., Castro-Bleda, M., and De-Mori, R. (2012). Cache neural network language models based on long-distance dependencies for a spoken dialog system. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4993–4996. IEEE.

Zamora-Martínez, F., España-Boquera, S., Gorbe-Moya, J., Pastor-Pellicer, J., and Palacios-Corella, A. (2013). April-ANN toolkit, A Pattern Recognizer In Lua with Artificial Neural Networks. https://github.com/pakozm/april-ann.

Zamora-Martínez, F., Frinken, V., España-Boquera, S., Castro-Bleda, M., Fischer, A., and Bunke, H. (2014). Neural network language models for off-line handwriting recognition. *Pattern Recognition*, 47(4):1642 – 1652.

Zamora-Martinez, F. and Sanchis-Trilles, G. (2010). Uch-upv english–spanish system for wmt10. In *Proceedings of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*, pages 213–217, Uppsala, Sweden. Association for Computational Linguistics.

Apéndice A

El algoritmo Backpropagation

En este capítulo se pretende exponer el algoritmo Backpropagation descrito de manera muy superficial en la introducción. Para ello, primero será necesario explicar los conceptos básicos sobre redes neuronales artificiales y cómo se relacionan éstas con el aprendizaje automático.

A.1. El problema del aprendizaje

Para estudiar cómo se relacionan las Redes Neuronales Artificiales o "Artificial Neural Nets" (ANN) con el aprendizaje, vamos a considerar una red arbitraria como una caja negra, de forma que esta red neuronal dispone de n entradas y m salidas (figura A.1). La red puede contener cualquier número de unidades ocultas y exhibir cualquier patrón de conexiones entre las unidades. Se nos proporciona un conjunto de entrenamiento $\{(x_1, t_1), \ldots, (x_p, t_p)\}$ formado por p pares ordenados de vectores, de dimensión n y m, llamados patrones de entrada y salida. Sean las funciones primitivas de cada nodo de la red continuas y diferenciables, y los pesos de los arcos números reales que parten de una configuración inicial (normalmente son escogidos aleatoriamente en un rango determinado, aunque existen diversas propuestas de algoritmos de inicialización de los pesos). Cuando se le presenta a la red el patrón de entrada x_i, se produce un salida o_i, que por lo general es diferente de la salida deseada t_i. Nuestro objetivo es hacer que o_i sea "lo más parecida" a t_i para todo $i = 1, \ldots, p$ utilizando un algoritmo de aprendizaje, en donde "parecida" viene dado por una función de error. Concretamente, queremos minimizar el valor de esta función de error de la red aplicada sobre los datos de entrenamiento, de forma que nos aproximemos más a la función objetivo de la

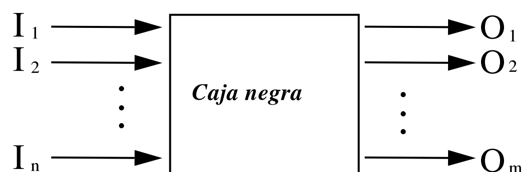

Figura A.1: Red neuronal como una caja negra.

$f(\sum_{i=1}^{n} w_i x_i - \theta) = z.$ $f(\sum_{i=0}^{n} w_i x_i) = z,$ incluyendo $-\theta$ como el peso w_0 y haciendo que $x_0 = 1$.

Tabla A.1: Cálculo realizado por una neurona.

red. Existe otro criterio relacionado con evitar el problema del sobreaprendizaje y que está relacionado con términos de regularización (weight decay) o con el uso de un corpus de validación para poder decidir un criterio de parada.

Existen diversas funciones de error para cuantificar la aproximación de la red a la función objetivo, siendo la más popular de entre ellas la función del error cuadrático medio (abreviado MSE del inglés *Mean Squared Error*). En este proyecto se pone a disposición del usuario el siguiente conjunto de funciones de error que, por las características modulares de la implementación, se podría ampliar en un futuro:

- Error cuadrático medio (MSE).
- Tangente hiperbólica.
- Entropia cruzada.
- Entropia cruzada completa.

En el capítulo del diseño del Algoritmo de Retropropagación del Error o "Backpropagation" (BP) serán comentados los aspectos relacionados con la implementación de estas funciones de error, así como la posibilidad de crear nuevas funciones a partir de una interfaz común que todas han de implementar.

A.2. Estructura de las redes neuronales

Una neurona es una unidad de proceso conectada con una serie de entradas y una única salida. La neurona produce una combinación lineal de las entradas, y propaga en su salida un valor que depende de esta combinación y la aplicación de una función de activación que, por lo general, no es lineal. Esta unidad se denomina perceptrón, y realiza las siguientes operaciones:

1. Un producto escalar del vector de entradas $\vec{x}$ por otro vector, cuyos valores representan los pesos, $\vec{w}$. A este valor se le resta un valor θ, denominado umbral. Al resultado se le denomina potencial: $\gamma = \vec{x} \cdot \vec{w} - \theta$.
2. Al potencial que hemos obtenido anteriormente, se le aplica una función de activación f no lineal, de forma que: $f(\gamma) = z$.
3. El resultado z, conseguido después de aplicar la función f, es propagado por la salida de la neurona.

Para simplificar los cálculos, se añade a la neurona una nueva entrada x_0, conectada a un valor fijo 1, de manera que el peso w_0 realiza la función del umbral θ que se resta en el punto 1.

En este proyecto podemos hacer uso de las siguientes funciones de activación:

- **Lineal**: El valor resultante no cambia, de forma que $f(x) = x$.
- **Sigmoide**: La función más popular de las ANN. El intervalo de valores resultantes se sitúa entre el 0 y el 1. El factor c, empleado en el cálculo de la función, hace que la sigmoide se parezca a la función escalón para valores altos de c (véase la figura A.3), algunas versiones de BP permiten variar este factor pero, en general, se puede dejar fijo. El cálculo se realiza de la siguiente forma: $s_c(x) = \frac{1}{1+e^{-cx}}$.
- **Tangente hiperbólica**: Esta función tiene la misma forma que la función sigmoide, pero proporciona valores entre $[-1, 1]$. Es posible relacionar ambas funciones de forma que, mediante una transformación lineal, se convierta una en la otra. La función se calcula de esta forma:
  $f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  En el caso en que $s_c = s_1$, la transformación se obtiene como: $tanh(x) = 2s_1(2x) - 1$.
- **Softmax**: La ventaja de esta función frente a las otras es que se normaliza la salida de las neuronas de forma que la suma de todas ellas sea igual a 1, lo que suele dar mejores resultados cuando la salida de la red se interpreta como una probabilidad condicionada a los datos de la entrada. Dado un conjunto de n neuronas en la red, con una única salida, la fórmula para el cálculo de la salida k es: $f(x_k) = e^{x_k} / \sum_{k'=1}^{n} e^{x_{k'}}$.

Una ANN se forma gracias a la conexión de un grupo de neuronas. La topología clásica agrupa las neuronas por capas, de forma que las neuronas de una capa están conectadas solamente con neuronas de la anterior y de la siguiente capa. A este tipo de ANN se las conoce como red "capa a capa", y también como Perceptrón Multicapa o "Multilayer Perceptron" (MLP). Pero es posible generalizar más la topología de una red, conectando algunas neuronas con otras neuronas que no estén en la anterior o la siguiente capa, de forma que la red se convierta en un grafo acíclico. A estos tipos de redes se las denomina red hacia-adelante general (véase la figura A.2).

Una red neuronal se ajusta modificando los valores de los pesos de las conexiones entre estas capas. Para poder modificarlos, pueden utlizarse varias técnicas, siendo las más populares de todas los métodos basados en descenso por gradiente. El gradiente se refiere al del error sobre los datos de entrenamiento respecto al espacio de pesos de la red. Este gradiente se calcula en base a una función de error (derivada), por lo que es necesario que las funciones de activación de las neuronas sean derivables. En el fondo, este algoritmo es un caso particular del clásico descenso por gradiente para localizar el mínimo local de una función con ciertas propiedades matemáticas.

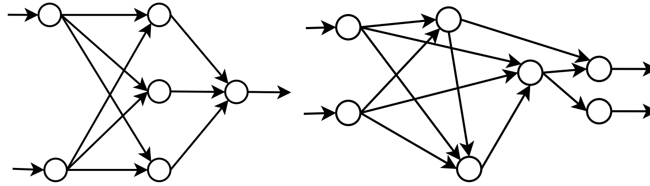

Figura A.2: Diagrama de redes hacia-delante, una capa a capa y otra general.

A.3. Algoritmos de aprendizaje para ANN

Los algoritmos de aprendizaje para redes neuronales se pueden clasificar básicamente en dos tipos: El primer tipo es el de los métodos basados en el descenso por gradiente y el segundo tipo es el de métodos de segundo orden, los cuales suelen ser variaciones del primer tipo de métodos.

A continuación se presentan ambos tipos y se muestran los algoritmos representativos de éstos.

A.3.1. Métodos basados en el descenso por gradiente

Los métodos más populares de entre los basados en el descenso por gradiente son el algoritmo BP y sus variantes. Este tipo de métodos se basan en definir una función de error y calcular el gradiente respecto a los pesos de la red, los cuales serán modificados según la dirección del gradiente, tratando de encontrar así un mínimo en la función de error.

Las variantes más importantes del algoritmo son:

- BP en modo “batch”: El error de la red se calcula como la media de los errores producidos en cada muestra de entrenamiento, por lo que la actualización de los pesos se realiza después de cada época.

- BP en modo “on-line”: El error de la red se calcula como el error producido por una muestra individual, actualizando así los pesos después de visitar cada muestra.

- BP con momentum: Se trata de una variación de los anteriores modos en la cual se añade un factor llamdo momentum (representado por μ) a la fórmula. Este factor contribuye a suavizar el entrenamiento al sumar un porcentaje del valor de los pesos en la iteración anterior a los de la iteración actual.

A.3.2. Métodos de segundo orden

En esta sección se presentan algunos de los métodos de segundo orden más populares, que tratan de acelerar la convergencia mediante el uso de nuevas técnicas:

- Algoritmo *Quickprop*: Este algoritmo procede de la misma forma que el algoritmo BP, pero se diferencia en el cálculo del incremento de los pesos.

El algoritmo asume que la función de error tiene un superfície (localmente) parecida a una función cuadrática, la cual trata de aproximar mediante el cálculo de la derivada de la función de error en la iteración anterior y la iteración actual. Una vez hecha la aproximación, se modifican los pesos buscando el mínimo de la función cuadrática, por lo que se produce un descenso mucho más rápido hacia el mínimo de la función de error.

- Algoritmo de *Newton*: Para el estudio de este algoritmo, se requiere un nuevo operador, la *matriz Hessiana*:

$$H = \frac{\partial^2 E}{\partial wji \partial w_{lk}}.$$

  Esta matriz está formada por la derivada segunda de la función de error respecto a cada uno de los pesos. Es posible calcular, a partir de la inversa de esta matriz, la dirección exacta del mínimo de la función de error E. El problema de este método es que el cálculo exacto de la matriz inversa de H tiene un coste computacional de $O(W^3)$, en donde W es el número de pesos en la red.

- Algoritmos *quasi-Newton*: Este tipo de algoritmos se basan en el algoritmo de *Newton*, pero el cálculo de la inversa de la matriz H se efectúa de forma aproximada. De este modo, se acelera la velocidad del algoritmo reduciendo el coste computacional, que pasa a ser $O(W^2)$, pero que introduce errores de imprecisión en el cálculo. La parte restante del algoritmo sigue siendo exactamente igual.

A.4. Introducción al algoritmo BP

El algoritmo BP es un algoritmo de aprendizaje supervisado que se usa para entrenar redes neuronales artificiales. El algoritmo trata de encontrar un punto mínimo en una función de error mediante la técnica del descenso por gradiente. Los conceptos que se van a presentar han sido extraídos de los libros [Rojas (1996); Bishop (1996); Ripley (1996)].

El algoritmo aplica una función de activación a las neuronas de cada capa después de ser calculadas. Estas funciones de activación necesitan ser derivables, de modo que al realizar paso de actualización de los pesos podemos efectuar el cálculo inverso de las funciones de activación en cada capa.

Existen unas cuantas funciones de activación para el algoritmo BP muy populares. De entre ellas, vamos a escoger la función sigmoide para explicar cómo funciona este algoritmo. La función sigmoide se define como:

$$s_c(x) = \frac{1}{1 + e^{-cx}}$$

El valor c es una constante que ha de tener un valor mayor que 0. Este valor modifica la forma de la función, de manera que cuando $c \to \infty$ la función sigmoide se asemeja a la función escalón en el origen de coordenadas. En la figura A.3

podemos observar como cambia la forma de la función sigmoide en función del valor elegido para c.

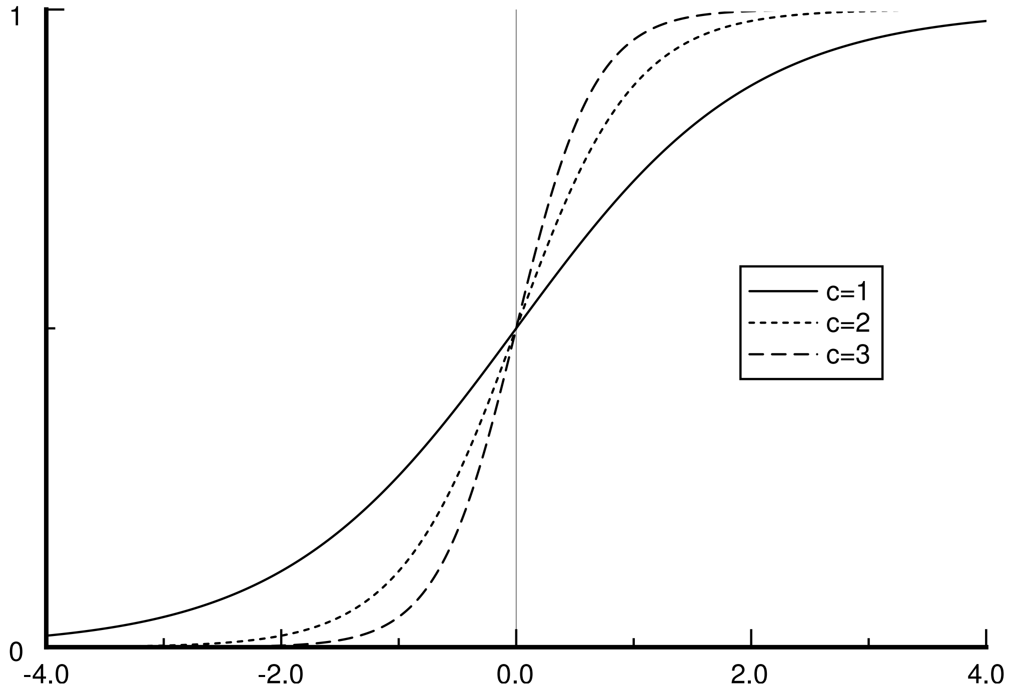

Figura A.3: Tres sigmoides (con c=1, c=2 y c=3).

Con el objetivo de simplificar las expresiones, a partir de ahora asumiremos que $c = 1$, de manera que $s_1(x)$ pasa a ser $s(x)$. El hecho de realizar esta simplificación no supondrá ningún cambio en los resultados. La función $s(x)$ quedará como:

$$s_1(x) = \frac{1}{1 + e^{-x}}$$

La derivada de esta función respecto a x será la siguiente:

$$\frac{d}{dx}s(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = s(x)(1 - s(x))$$

A.4.1. El problema del aprendizaje

Las redes hacia-adelante pueden verse como un grafo computacional cuyos nodos son unidades de cómputo y cuyas aristas dirigidas transmiten información numérica de un nodo a otro. Cada unidad de cómputo es capaz de calcular una función por lo general no lineal de sus entradas, de manera que la red calcula una composición de funciones que transforma un espacio de entrada en un espacio de salida. La función que realiza esta transformación es llamada función de la red. El problema del aprendizaje consiste en encontrar la combinación óptima de pesos de forma que la función de la red se aproxime lo más posible a una cierta función

f. No obstante, esta función f es desconocida para nosotros, pero disponemos de ejemplos que la definen implícitamente. Por tanto, como ya se ha explicado de forma general en el primer apartado de este capítulo, necesitamos ajustar los parámetros de la red (pesos) para minimizar un función de error determinada, que nos da una medida de lo buena que es la aproximación a la función f.

Después de minimizar la función de error para el conjunto de entrenamiento, al presentar nuevos patrones de entrada a la red, desconocidos hasta el momento, se espera que la red interpole y acierte con la salida. La red debe aprender a reconocer si los nuevos patrones de entrada son similares a los otros patrones ya aprendidos y producir una salida similar para ellos.

Ahora vamos a extender la red para que cada vez que llegue un nuevo patrón, se calcule la función de error E de forma automática para el mismo. Para ello, conectamos cada unidad de salida a un nuevo nodo que evalúa la función $\frac{1}{2}(o_{ij} - t_{ij})^2$, en donde o_{ij} y t_{ij} representan la componente j del vector de salidas o_i y del objetivo t_i, respectivamente. La salida de estas m unidades es recolectada por un nodo final que suma todas las entradas recibidas, de manera que la salida de este nodo es la función E_i. La misma extensión de la red ha de ser construida para cada patrón t_i. La salida de la red extendida es la función de error E.

Ahora tenemos una red capaz de calcular el error total para un conjunto de entrenamiento. Los pesos de la red son los únicos parámetros modificables para minimizar la función de error E. Dado que E es calculada a partir de la composición de funciones de los nodos, E será una función continua y derivable de los l pesos $w_1, w_2, \ldots, w_l$ que componen la red. Podemos entonces minimizar E utilizando un proceso iterativo de descenso por gradiente. El gradiente se define como:

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_l}).$$

Cada peso es actualizado por el incremento:

$$\triangle w_i = -\eta \frac{\partial E}{\partial w_i}, \text{ para } i = 1, .., l.$$

En donde η representa el factor de aprendizaje, un parámetro de proporcionalidad que define la longitud del paso de cada iteración en la dirección del gradiente.

Con esta extensión de la red original, el problema del aprendizaje se reduce a calcular el gradiente de la función de la red respecto a los pesos que la componen. Una vez tengamos un método para el cálculo de este gradiente, podemos ajustar los pesos de forma iterativa, esperando encontrar un mínimo local de la función de error E en la cual $\nabla E = 0$.

A.4.2. Derivada de la función de la red

Vamos a dejar de lado todo lo visto hasta el momento, para centrarnos en encontrar la derivada de la función de la red extendida respecto a cada uno de los pesos, o dicho de otro modo, como obtener ∇E.

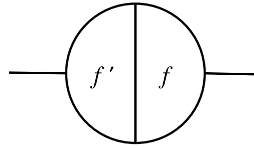

Figura A.4: Los dos lados de una unidad de proceso.

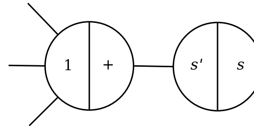

Figura A.5: Sumatorio y función de activación separadas en dos unidades.

La red puede realizar dos pasos: El paso hacia-delante y el paso de retropropagación. En el paso hacia-delante, las unidades calculan una función de sus entradas y propagan el resultado en la salida. Al mismo tiempo, la derivada de la función es calculada para la misma entrada, y el resultado se guarda para su posterior uso. Podemos ver las unidades como separadas en dos partes, una con el resultado obtenido al aplicar la función y otra con el resultado de la derivada, como se ilustra en la figura A.4. En el paso de retropropagación, las unidades utilizarán este último resultado para calcular la derivada de la función de la red.

Ahora separaremos la unidad en dos partes independientes (figura A.5). La primera es la que calcula, para una unidad k, el sumatorio de todas sus entradas, obteniendo $\gamma_k = \sum_i w_{ki} \cdot x_{ki}$. La segunda calculará la función primitiva de la misma unidad, obteniendo: $f(\gamma_k) = z_k$.

De esta forma, tenemos tres casos en los que estudiar cómo calcular la derivada:

Primer caso: Composición de funciones

La composición de funciones se consigue mediante la unión de la salida de una unidad con la entrada de otra. La función que calcula el primer nodo es g, y la que calcula el segundo nodo es f, de forma que se realiza la composición $f(g(x))$. El resultado de esta composición es el obtenido por la salida de la última

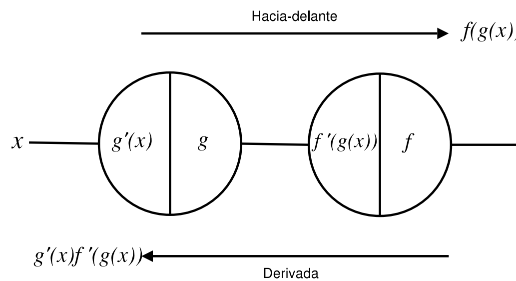

Figura A.6: Composición de funciones.

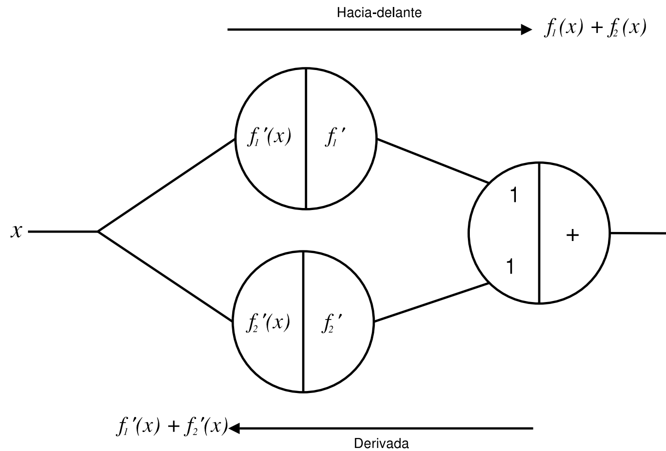

Figura A.7: Adición de funciones.

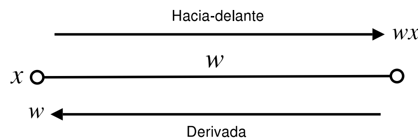

Figura A.8: Pesos en las aristas.

unidad. De forma paralela, cada unidad guarda el valor de la derivada en su lado izquierdo. En el paso de retropropagación entra un 1 por la parte derecha de la última unidad, que es multiplicado por el valor guardado en el lado izquierdo de cada unidad. El resultado obtenido es transmitido hacia la siguiente unidad por la izquierda. Por tanto, el cálculo que la red está efectuando es $f'(g(x))g'(x)$, que es la derivada de $f(g(x))$. Cualquier secuencia de composición de funciones puede ser evaluada de esta manera, y su derivada puede ser obtenida el paso de retropropagación.

Segundo caso: Adición de funciones

En este caso, tenemos una serie de nodos en paralelo conectados a un nodo de salida que realiza la suma de todas sus entradas. Para dos nodos de entrada, que calculan las funciones f_1 y f_2, la suma calculada por el nodo de salida en el paso hacia-delante será $f_1(x)+f_2(x)$. La derivada parcial de la función de adición respecto a cualquiera de sus entradas es 1. En el paso de retropropagación entra un 1 por la parte derecha de la unidad encargada de realizar la suma, por lo que el resultado final de la red será $f_1'(x) + f_2'(x)$, que es la derivada de la función evaluada en el paso hacia-delante. Se puede demostrar, mediante inducción, que la derivada de la adición de cualquier número de funciones puede ser tratada del mismo modo.

Tercer caso: Pesos en las aristas

Las aristas pueden tener un peso asociado que podría ser tratado de la misma forma que un caso de composición de funciones, pero hay una forma diferente de tratar con ellas que resulta ser más fácil. En el paso hacia-delante, la información entrante x es multiplicada por el peso w, de forma que el resultado es wx. En el paso de retropropagación, el valor 1 es multiplicado por el peso de la arista. El resultado es w, que es la derivada de wx respecto a x.

A.4.3. Demostración formal del algoritmo BP

Ahora ya podemos formular el algoritmo BP y probar por inducción que funciona en una red hacia-delante arbitraria con funciones de activación en los nodos. Vamos a asumir que estamos tratando con una red que únicamente tiene una unidad de entrada y otra de salida.

Consideraremos una red con una única entrada real x y la función de red F. La derivada $F'(x)$ es calculada en dos fases:

- Hacia-delante: La entrada x se introduce en las entradas de la red. Las funciones de activación y sus derivadas son evaluadas en cada nodo. La derivada es guardada.
- Retropropagación: La constante 1 es introducida por la unidad de salida y la red se ejecuta hacia atrás. La información pasa de nodo a nodo y es multiplicada por el valor guardado en la parte izquierda de cada nodo. El resultado recogido en la unidad de entrada es la derivada de la función de la red respecto a x.

Para demostrar que el algoritmo es correcto aplicaremos un razonamiento inductivo. Supongamos que el algoritmo BP funciona para cualquier red hacia-delante con n nodos. Ahora podemos considerar una red hacia-delante con $n+1$ nodos. El paso hacia-delante es ejecutado en primera instancia y el resultado obtenido en la única unidad de salida es la función de la red F evaluada con x. Las m unidades conectadas a esa unidad de salida producen $F_1(x), \ldots, F_m(x)$ por su salida, y teniendo en cuenta que la función de activación de la unidad de salida es φ, tenemos que:

$$F'(x) = \varphi(w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x))$$

Por lo que la derivada de F con x es:

$$F'(x) = \varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x))$$

En donde s representa $F(x)$. Si introducimos un 1 por el nodo salida de la red y ejecutamos el paso de retropropagación, podremos calcular la derivada de las funciones $F_1(x), \ldots, F_m(x)$. Si en lugar de introducir un 1, introducimos la constante $\varphi'(s)$ y la multiplicamos por los pesos asociados a los nodos, entonces obtendremos $w_1 F_1'(x)\varphi'(s)$ para el nodo que calcula $F_1(x)$ en la unidad de entrada, y lo mismo para los otros nodos hasta el nodo m. El resultado de ejecutar el algoritmo de retropropagación será, por tanto:

$$\varphi'(s)(w_1F_1'(x) + w_2F_2'(x) + \cdots + w_mF_m'(x))$$

Que es la derivada de F evaluada con x. Nótese que la introducción de las constantes $w_1\varphi'(s), \ldots, w_m\varphi'(s)$ en las m unidades conectadas la unidad de salida puede ser realizada introduciendo un 1 en la unidad de salida, multiplicándolo por el valor almacenado $\varphi'(s)$ y distribuyendo el resultado a las m unidades conectadas a través de las aristas con pesos $w_1, \ldots, w_m$. De hecho, estamos ejecutando la red hacia atrás tal y como el algoritmo BP exige, con lo cual se prueba que el algoritmo funciona para $n+1$ nodos y la demostración concluye.

A.4.4. Aprendiendo con el BP

Recuperemos ahora el problema del aprendizaje en las redes neuronales. Queremos minimizar una función de error E, la cual depende de los pesos de la red. El paso hacia-delante es efectuado de la misma forma en que se ha hecho hasta el momento, pero ahora vamos a guardar, además del valor de la derivada en el lado izquierdo de cada unidad, el valor de la salida en el lado derecho. El paso de retropropagación será ejecutado en la red extendida que calcula la función de error y fijaremos nuestra atención en uno de los pesos, que denominaremos w_{ij}, cuyas arista relaciona la unidad i con la unidad j. En el primer paso calcularemos $x_j = \sum o_i w_{ij}$, de manera que x_j será el valor que llegue a la unidad j. También tendremos que $o_j = f_j(x_j)$, siendo f_j la función de activación que se aplica en el nodo j. La unidad i guardará en su salida el valor o_i. Las derivadas parciales de E se calculan como:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = o_i\frac{\partial E}{\partial x_j} = o_i f_j'(x_j)\frac{\partial E}{\partial o_j} = o_i\delta_j, \text{ con } \delta_j = f_j'(x_j)\frac{\partial E}{\partial o_j}.$$

La primera igualdad es una consecuencia de la dependencia entre E y los pesos con los que se relaciona en las salidas. La segunda se obtiene a partir de $x_j = \sum o_i w_{ij}$. Denominaremos a δ_j el *error retropropagado* para el nodo j. Por tanto, la información que cada nodo ha de guardar para poder realizar estos cálculos son:

- La salida o_i del nodo en el paso hacia-delante.
- El resultado acumulativo tras ejecutar el paso de retropropagación en ese nodo, δ_i.

Para concluir el cálculo es necesario definir cúal va a ser el valor de δ_j para cada unidad de la red. Podemos observar que $\delta_j = f_j'(x_j)\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial x_j}$. De esta expresión, podemos calcular $f_j'(x_j)$ para cualquier unidad, ya que el valor x_j es conocido para todas las unidades durante el paso de retropropagación. Despejaremos cúal debe ser el valor del otro factor de la expresión, distinguiendo entre dos casos. En las unidades de salida de la red, $\frac{\partial E}{\partial o_j}$ se calcula de forma directa. Para las unidades de las capas ocultas, teniendo en cuenta la unidad j y las K que le suceden $(j \to K)$:

$$\frac{\partial E}{\partial o_j} = \sum_{k:j \to K} w_{jk} \frac{\partial E}{\partial x_k} = \sum_{k:j \to K} w_{jk} \delta_k.$$

Una vez todas las derivadas parciales han sido calculadas, podemos añadir a cada peso w_{ij} el incremento:

$$\triangle w_{ij} = -\eta o_i \delta_j$$

Con todo esto, ya tenemos definido el funcionamiento del algoritmo para redes neuronales generales, las cuales son capaces de describir cualquier topología hacia-delante.

A continuación se ofrece una traza paso a paso del algoritmo completo para una muestra de entrenamiento del problema consistente en aprender la función lógica "o exclusiva" o *xor*. El problema *xor* es un problema clásico en el cual se entrena una red para aprender la función mostrada en el cuadro A.2.

i_1	i_2	t_1
0	0	0
0	1	1
1	0	1
1	1	0

Tabla A.2: Función a aprender en el problema de la *xor*.

En la siguiente traza se muestra la ejecución del algoritmo BP para la muestra (0, 1).

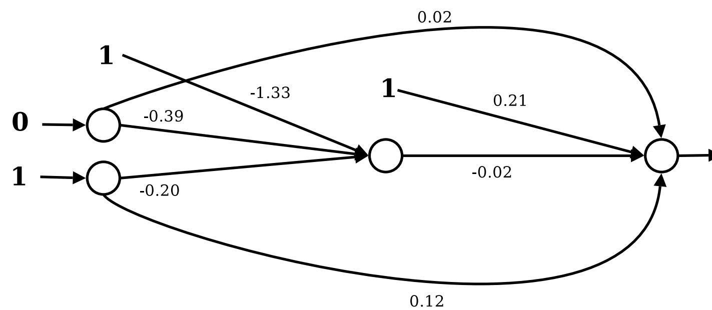

Figura A.9: Traza del BP, para resolver el problema de la *xor*. Configuración inicial de la red.

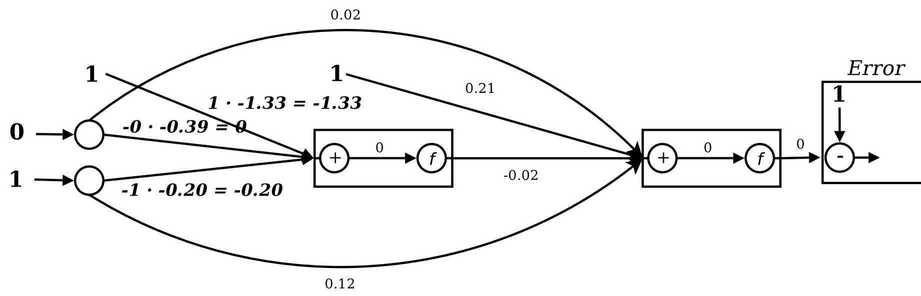

Figura A.10: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 1/4.

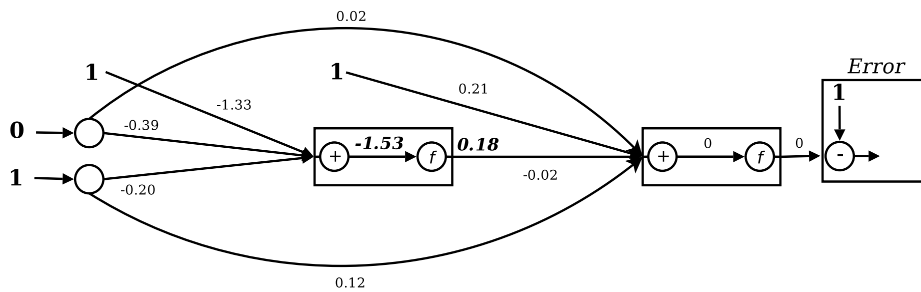

Figura A.11: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 2/4.

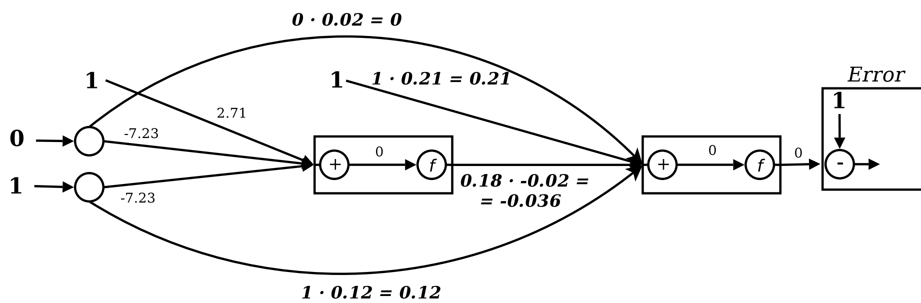

Figura A.12: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 3/4.

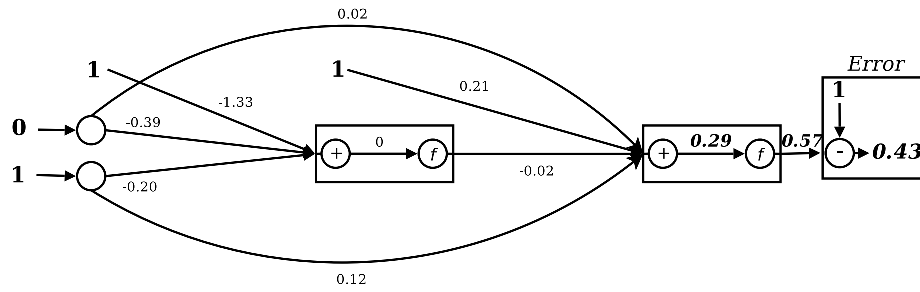

Figura A.13: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 4/4.

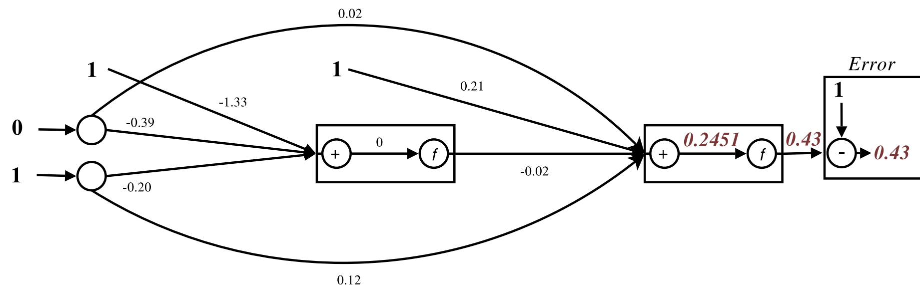

Figura A.14: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 1/6 (Cálculo de la derivada de la neurona de salida).

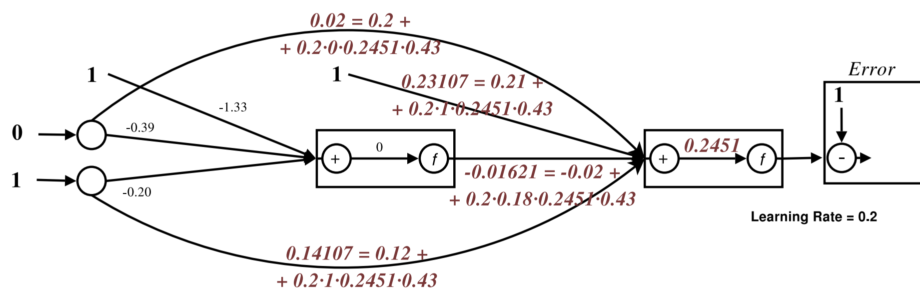

Figura A.15: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 2/6 (Ajuste de los pesos conectados a la neurona de salida).

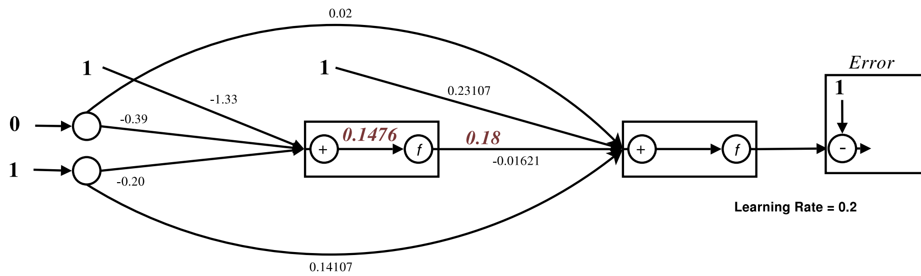

Figura A.16: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 3/6 (Cálculo de la derivada de la neurona oculta).

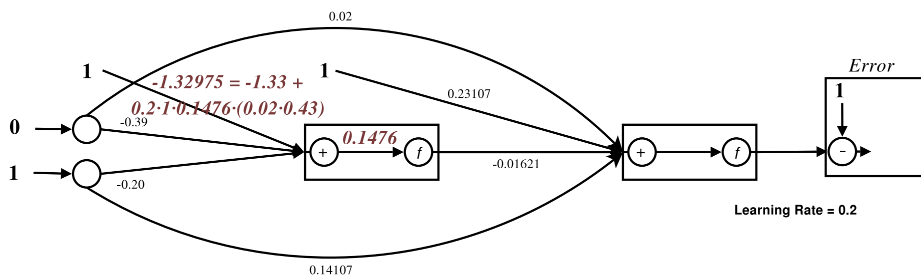

Figura A.17: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 4/6 (Ajuste del peso umbral de la neurona oculta).

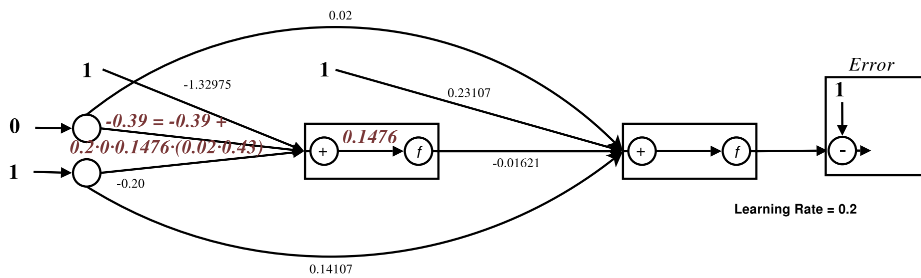

Figura A.18: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 5/6 (Ajuste del primer peso de la neurona oculta).

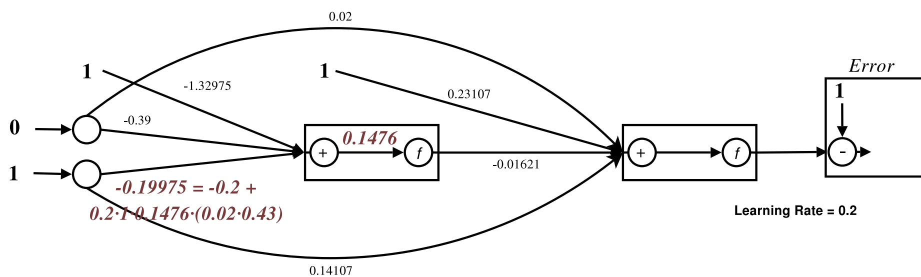

Figura A.19: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 6/6 (Ajuste del segundo peso de la neurona oculta).

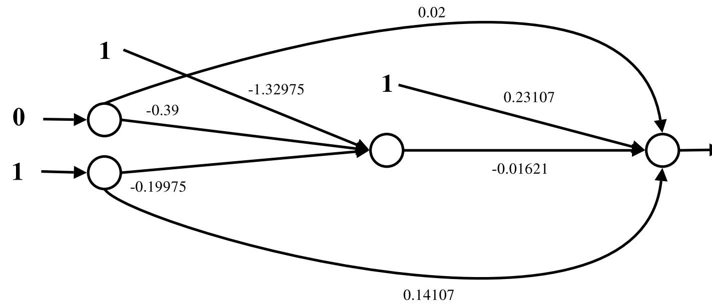

Figura A.20: Traza del BP, para resolver el problema de la *xor*. Configuración final de la red.

La iteración del algoritmo BP con la muestra (0, 1) finaliza aquí. En este punto, se supone que la red proporcionará salidas más cercanas a la salida deseada para la muestra (0, 1). Por lo general, será necesario un mayor número de presentaciones de muestras con tal de obtener una mejor aproximación a las salidas deseadas.

A.5. Forma matricial del BP

El enfoque empleado a lo largo de este capítulo con grafos computacionales para la prueba del algoritmo BP puede sernos útil para el caso de las redes con una topología general. No obstante, en otro tipo de redes como las redes capa a capa, puede resultar de interés utilizar diferentes enfoques, como es el caso de la forma matricial del BP. Esta forma matricial nos permite organizar los elementos de la red para sacar un mayor provecho de los procesadores vectoriales u otras máquinas especializadas en el álgebra lineal.

Para mostrar su funcionamiento, se empleará una red con una capa de entrada, una capa oculta y otra de salida (con n, k y m unidades). La capa de entrada se representa con un vector o de dimensiones $1 \times n$, y las capas siguientes se denotan con o^i, en donde i es la posición que toman las capas después de la capa de entrada. Las matrices de pesos son representadas con W_i, siendo W_1 la matriz de conexiones entre la capa o y la capa o^1, cuyas dimensiones son $n \times k$, lógicamente.

El paso hacia-delante producirá $o^2 = s(o^1 W_2)$ en el vector de salidas, en donde $o^1 = s(oW_1)$. Las derivadas almacenadas en el paso hacia-delante por las k unidades ocultas y las m unidades de salida pueden ser escritas como la matriz diagonal:

$$D_2 = \begin{pmatrix} o_1^2(1-o_1^2) & 0 & \cdots & 0 \\ 0 & o_2^2(1-o_2^2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^2(1-o_m^2) \end{pmatrix}$$

La misma matriz se puede confeccionar para las derivadas entre la capa de entrada y la capa oculta:

$$D_1 = \begin{pmatrix} o_1^1(1-o_1^1) & 0 & \cdots & 0 \\ 0 & o_2^1(1-o_2^1) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_k^1(1-o_k^1) \end{pmatrix}$$

El vector e, que guarda las derivadas de las diferencias entre las salidas obtenidas y las salidas deseadas, se define como:

$$e = \begin{pmatrix} (o_1^2 - t_1) \\ (o_2^2 - t_2) \\ \vdots \\ (o_m^2 - t_m) \end{pmatrix}$$

El vector m-dimensional δ^2, que representa el *error retropropagado* por las unidades de salida, se obtiene mediante la expresión:

$$\delta^2 = D_2 e$$

El vector k-dimensional δ^1, que representa el *error retropropagado* por las unidades ocultas, se obtiene mediante la expresión:

$$\delta^1 = D_1 W_2 \delta^2$$

Los incrementos de las matrices W_1 y W_2 vienen dados por:

$$\triangle W_2^T = -\eta \delta^2 o^1$$

$$\triangle W_1^T = -\eta \delta^1 o$$

Es fácil generalizar estas ecuaciones para el caso de una red capa a capa con l capas. La expresión para la obtención de los *errores retropropagados* se puede definir de forma recursiva:

$$\delta^i = D_i W_{i+1} \delta^{i+1}, \text{ para } i = 1,..,l-1.$$

O podemos desplegar la expresión para obtener:

$$\delta^i = D_i W_{i+1} \cdots W_{l-1} D_{l-1} W_l D_l e$$

Los incrementos de las matrices de pesos se calculan de la misma forma para el caso con l capas. Este punto de vista resultará ser esencial para la elaboración de este proyecto, y será ampliado más adelante para permitir el cálculo de varias entradas de forma simultánea, consiguiendo así una aceleración notable al utilizar librerías especializadas en el cálculo de productos de matrices.

A.6. El algoritmo BP con momentum

Ya se ha mencionado que una de las variantes del algoritmo BP es el algoritmo BP con momentum.

La introducción del factor momentum pretende suavizar el ajuste de los parámetros, evitando así algunos problemas del algoritmo clásico. El momentum provoca un efecto de inercia en el entrenamiento de la red, evitando cambios bruscos al realizar la actualización de los pesos.

El cálculo de los pesos en el algoritmo clásico se realizaba conforme a la siguiente fórmula:

$$\triangle w_{ij} = -\eta o_i \delta_j.$$

A esta fórmula pretendemos añadirle el momentum, que será representado por μ a partir de ahora. Como el momentum se define como un factor del incremento de los pesos, es necesario introducir dos valores para cada peso: w_{ij}^k representa el valor del peso en la posición ij para la iteración actual, mientras que $w_i^{k-1}j$ representa el valor de ese mismo peso en la iteración previa. La diferencia entre estos valores estará determinando el incremento del peso, por lo que la fórmula queda como sigue:

$$\triangle w_{ij}^k = -\eta o_i \delta_j - \mu(w_{ij}^k - w_{ij}^{k-1})$$

El valor μ debería tener valores entre $[0, 1]$, dado que se define como un porcentaje. Resulta obvio que si $\mu = 0$, nos encontramos de nuevo con el algoritmo BP clásico.

El uso de esta técnica tiene sus ventajas y desventajas. La principal ventaja es que se logra suavizar el entrenamiento de la red. No obstante, la inclusión de los valores necesarios para actualizar los pesos en la red exige que se guarden los pesos de la iteración anterior, por lo que se requiere una porción mayor de memoria a la hora de realizar el entrenamiento.

Otra ventaja que nos proporciona el momentum es que puede propiciar el descubrimiento de nuevas zonas en la función de error a partir de una misma inicialización, pudiendo encontrar así nuevos locales mínimos mejores que los anteriores.

A.7. El algoritmo BP con weight decay

Los algoritmos de poda se utilizan para ajustar las arquitecturas de las redes neuronales a los problemas que se tratan de resolver. Estos algoritmos tratan de detectar qué partes de la red no contribuyen al cálculo del BP, y se dividen principalmente en dos tipos: Los algoritmos de poda de unidades y los algoritmos de poda de pesos. En esta sección se presenta el BP con weight decay [Moody et al. (1995)], uno de los algoritmos de poda de pesos más populares, que cae dentro del grupo de algoritmos con factor de penalización.

En este algoritmo se introduce el término weight decay (representado por α) en la fórmula para la actualización de los pesos del algoritmo BP clásico:

$$\triangle w_{ij}^{k} = -\eta o_i \delta_j - \alpha w_{ij}^{k-1}$$

El término completo que se ha añadido a la fórmula representa la resta de una porción del peso en la anterior iteración al peso en la iteración actual. Concretamente, el weight decay indica qué porcentaje del peso en la anterior iteración se restaría al peso en la iteración actual.

Al restar este término en la actualización de los pesos, aquellos pesos que no sean reforzados de forma significativa por los incrementos efectuados en el paso de retropropagación se aproximarán más rápidamente hacia el valor 0, produciendo así un efecto de poda en el peso.

De la misma forma que ocurría con el momentum, hemos de ser capaces de recuperar el valor que el peso tenía en la anterior iteración para poder realizar este cálculo, por lo que tendremos que guardar los pesos de la iteración anterior cada vez que queremos actualizar los pesos. Si se pretende aplicar el algoritmo BP con momentum para el entrenamiento de una red, no es una mala idea usar también la técnica del weight decay, ya que de todos modos hemos de guardar los pesos en la iteración anterior.

A.8. Complejidad algorítmica del BP

El entrenamiento se realiza en dos pasos: El paso hacia-delante y el paso de retropropagación. El primer paso tiene un coste de $O(W)$, siendo W el número de pesos de la red. El paso de retropropagación calcula primero las derivadas de las neuronas, lo cual tiene un coste de $O(N)$, siendo N el número de neuronas en la red. A continuación, se actualizan los pesos según el valor obtenido en la fórmula del BP. Por tanto, el coste final del algoritmo será de $O(2(W + N))$, pero como $N \leq M$, nos queda un coste lineal con el número de conexiones en la red, $O(W)$.

Este coste no es reducible ya que es necesario visitar todos los pesos de la red para modificarlos. No obstante, es posible acelerar la ejecución del algoritmo en implementaciones especializadas para una determinada topología. Esto es lo que se pretende hacer en este proyecto mediante el uso de arquitecturas Single Instruction Multiple Data (SIMD) que las Unidad Central de Procesamiento o "Central Processing Unit" (CPU) y Unidad de Procesamiento Gráfico o "Graphics Processing Unit" (GPU) actuales implementan. Gracias a estas arquitecturas, seremos capaces de paralelizar procesos como la aplicación de las funciones de activación sobre las neuronas de las capas o la modificación de pesos en el paso de retropropagación.

A.9. Inicialización de pesos

El valor inicial de los pesos es un factor clave en la obtención de un mínimo local de la función de error de la red. Por esto mismo, la capacidad de poder reproducir un mismo experimento con una misma inicialización resulta bastante importante. Para lograr esta reproducibilidad, hemos utilizado un generador de números pseudo-aleatorios que pueda guardar y recuperar su estado. El generador

que se utiliza en este proyecto es el *Mersenne Twister* [Matsumoto and Nishimura (1998)]. Este generador nos permite la reproducción exacta de experimentos a partir de una misma semilla, como también nos permite su reanudación a partir del estado en el que se encontraba cuando fue detenido.

A.10. Sobreentrenamiento

El sobreentrenamiento (también conocido como *overfitting*) es uno de los problemas que presentan las técnicas del reconocimiento de formas. El sobreentrenamiento se produce cuando los modelos para el reconocimiento se ajustan demasiado al conjunto de muestras con el cual se realiza el entrenamiento, de forma que el modelo no consigue generalizar las clases lo suficiente y falla en la clasificación durante la fase de test cuando le presentamos muestras que no estaban en el conjunto de entrenamiento.

Por este motivo, los conjuntos de muestras suelen ser separados en tres subconjuntos de muestras, cada uno de los cuales tiene un objetivo:

- Conjunto de entrenamiento: Es el subconjunto de muestras con el cual se pretende entrenar la red.

- Conjunto de validación: Es el subconjunto de muestras con el cual se pretende ajustar los parámetros de entrenamiento de la red. Este subconjunto no es usado por el algoritmo de entrenamiento, por lo que los resultados de clasificación son más fiables. A veces se pueden considerar 2 conjuntos de validación (si hay datos suficientes): uno para elegir el criterio de parada para una red concreta y otro para elegir la mejor red (en caso de probar diversas topologías o configuraciones) antes de realizar la medición del test.

- Conjunto de test: Es el subconjunto de muestras con el cual se realiza la clasificación con la red ya entrenada para obtener el error final de la red y se utilizaría una única vez cuando se ha decidido la mejor red (topología y pesos).

Existen variantes de esta idea que realizan diversas particiones de la parte de entrenamiento y validación (conocidos como “leaving one out”, etc.) cuando el tamaño de estos conjuntos no es suficientemente grande. La idea consiste en realizar diversas particiones, entrenar con ellas y promediar los resultados.

Para realizar un entrenamiento, es recomendable fijar un criterio de parada relacionado con el conjunto de validación, consiguiendo así evitar el sobreentrenamiento en el caso de que las muestras de los conjuntos para el entrenamiento y la validación tengan la suficiente variabilidad. El entrenamiento suele detenerse cuando el error en validación no ha mejorado durante un número determinado de presentaciones. Al conseguir un buen resultado de clasificación respecto al conjunto de validación es más probable que los resultados de clasificación para el conjunto de test sean buenos.

Apéndice B

El formato Lira

El formato Lira representa un modelo de lenguaje de n-gramas como un autómata finito estocástico [Llorens Piñana (2000)], lo cual permite representar modelos de lenguaje interpolados o suavizados por back-off de una forma compacta y eficiente.

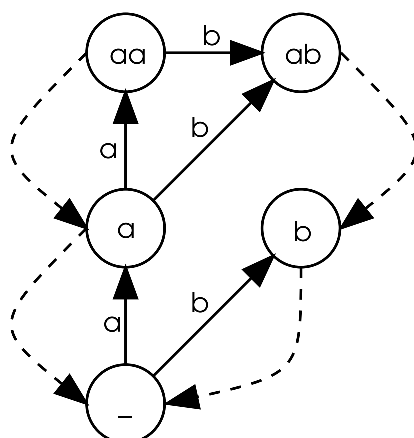

Figura B.1: Ejemplo de un modelo de lenguaje de n-gramas representado en forma de autómata. Las transiciones con una línea discontinua indican un suavizado por back-off [Zamora-Martínez (2012)].

El algoritmo de Viterbi [Viterbi (1967)], que se utiliza para la búsqueda de la secuencia de palabras que maximizan la probabilidad, tan sólo necesita un identificador del estado del automáta que representa al contexto de la siguiente hipótesis. Este identificador es una clave que va actualizándose conforme el sistema procesa hipótesis. Una clave representa una secuencia de $n-1$ palabras. Cada hipótesis tiene una clave, y cuando una hipótesis se extiende con una palabra nueva, entonces se transita en el autómata con dicha palabra y el estado destino es la nueva clave. Este proceso permite que la búsqueda en el modelo de lenguaje sea muy eficiente. La figura B.1 muestra un ejemplo de representación de un modelo de n-gramas como un autómata.

B.1. Campos del formato Lira

El formato Lira describe la información sobre el autómata finito estocástico que representa en los siguientes campos:

1. Tamaño del vocabulario.
2. Palabras del vocabulario.
3. Orden máximo del n-grama.
4. Número de estados.
5. Número de transiciones.
6. Límite máximo de probabilidad de transición.
7. Cantidad de estados tienen con un número diferente de transiciones.
8. Listado del número de estados que tienen un número diferente de transiciones.
9. Estados inicial, final y más bajo.
10. Lista de transiciones por back-off, con las relaciones de estado origen, estado destino, probabilidad de transición y probabilidad máxima al transitar.
11. Lista de transiciones, con las relaciones de estado origen, estado destino, identificador de la palabra con la que se transita y probabilidad de transición.

B.2. Ejemplo

El siguiente código es un ejemplo de un fichero en formato Lira completo. Al comienzo de cada campo puede encontrarse un comentario que describe el campo que va a continuación:

```
# number of words and words
5
<s>
</s>
a
b
c
# max order of n-gram
3
# number of states
8
# number of transitions
18
```

```
# bound max trans prob
15.299730
# how many different number of transitions
6
# "x y" means x states have y transitions
1 0
2 1
2 2
1 3
1 4
1 5
# initial state, final state and lowest state
3 0 7
# state backoff_st 'weight(state->backoff_st)' [max_transition_prob]
# backoff_st == -1 means there is no backoff
0 7 0.000000 -0.998529
1 5 -0.810930 -0.405465
2 5 -0.405465 -0.405465
3 7 -0.456758 -0.510826
4 7 -0.233615 -1.098612
5 7 0.171850 -0.693147
6 7 16.298259 15.299730
7 -1 -1 -0.998529
# transitions
# orig dest word prob
1 0 2 -0.405465
2 6 4 -0.405465
3 2 3 -0.510826
3 6 4 -1.60944
4 0 2 -1.09861
4 5 3 -1.09861
5 0 2 -1.38629
5 5 3 -2.07944
5 6 4 -0.693147
6 0 2 -1.94591
6 1 3 -1.25276
6 6 4 -1.94591
6 4 5 -1.25276
7 3 1 -1e+12
7 0 2 -1.55814
7 5 3 -0.998529
7 6 4 -1.15268
7 4 5 -2.25129
```

Apéndice C

Artículo IberSPEECH 2014

First steps towards Skipping NNLMs

A. Palacios-Corella[1], F. Zamora-Martínez[2], S. España-Boquera[1], and M.J. Castro-Bleda[1]

[1] Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Valencia, Spain
[2] Departamento de Ciencias Físicas, Matemáticas y de la Computación, Universidad CEU Cardenal Herrera, Alfara del Patriarca (Valencia), Spain

Abstract. Statistical language modeling greatly suffers from the effects of data sparsity which is tackled by means of smoothing techniques. Continuous space language models are able to interpolate unseen word histories but new problems and challenges arise, as a very high computational cost during evaluation of N-gram probabilities, due to the softmax normalization constant. Several approaches to study how to reduce this computational cost have been proposed in the literature. This work tries to improve the use of pre-computed softmax normalization constants tables by including the Skipping N-grams technique into NN LMs and describes some experiments conducted on IAM-DB corpus to validate the viability of the proposed technique. The skipping for NN LMs works as regularization, but additionally helps to simplify the use of precomputation of softmax normalization constants, as will be shown in the preliminary experiments of this paper.

1 Introduction

The estimation of a-priori probabilities of word sequences is one of the main purposes of language models (LMs). They play a key role in many natural language processing applications such as speech or text recognition, machine translation, part-of-speech tagging, or information retrieval. The most commonly used LMs are statistical N-gram models [6, 4], which only consider the $N-1$ previous words to estimate the LM probability for a sequence of words of length $|W|$:

$$p(w_1 \dots w_{|W|}) \approx \prod_{i=1}^{|W|} p(w_i | w_{i-n+1} \dots w_{i-1}) \quad (1)$$

The use of N-grams is usually restricted, in practice, to low orders, as is the case of trigrams. Although trigram LMs work well in practice, there are many improvements over this simple model, including higher-order N-grams, smoothing techniques, skipping models, clustering models or cache models [4].

The estimation of these models are based on counting occurrences of word histories in a training corpus. A more recent alternative to the classical "count-based" N-grams are those based on a continuous representation of the lexicon

using connectionist approaches, as is the case of NN LMs based on multilayer perceptrons [15, 18, 19].

NN LMs do not require the use of explicit smoothing techniques usually employed in count-based N-grams (e.g., backing-off), but important computational issues appear when using large vocabularies, majorly due to output softmax activation function. Short-list [15] and pre-computation of softmax normalization constants [18, 19] allow to reduce significantly this computational cost.

This paper describes the first steps of this research and its main contribution is to study whether the connectionist skipping N-gram LMs can help to improve NN LMs performance in any way.

2 Related language models

2.1 Skipping N-gram LMs

N-grams suffer from data sparsity making it necessary the use of smoothing techniques. Skipping N-grams [5, 14, 13, 10, 16, 4] can improve the generalization ability of standard smoothing techniques. The idea is that the exact context will have not probably been seen during training, but the chance of having seen a similar context (with gaps that are skipped over) increases as the order of the N-gram does.

Let us explain the idea with an example: suppose that the sentence *"The little boy likes pizza very much"* appears in the training data and we are trying to estimate a 5-gram. The training sentence has contributed to the estimation of p(pizza|the little boy likes). Unfortunately, this sentence cannot help much in the estimation of p(pizza|the little girl likes). The usual technique of backing-off would consist of using lower order N-grams. In this case, we would need to descend until p(pizza|likes). By skipping some words from the past history, the training sentence is useful to estimate p(pizza|the little — likes). For instance, the probability of p(pizza|the little boy eats) would benefit from p(pizza|the little boy —) whereas backing-off would require to descent until unigrams as far as the example sentence is concerned.

Skipping N-grams are not only based on skipping words from the past history but also on the combination of different ways of performing these skips. Each different way of skipping words can be considered a lower order LM in some way and the Skipping N-grams are a mixture of them. For example, the representation of several skipped trigrams (at most two context words are not skipped) may approximate a higher order N-gram using less resources, which explains why some authors have considered Skipping N-grams as a *poor man's* alternative to higher order N-grams. Nevertheless, our emphasis here is that they can also be useful to improve NN LMs probability computation.

2.2 NNLMs

NN LMs are able to learn a continuous representation of the lexicon [15, 18, 19]. A scheme of a NN LM is illustrated in Figure 1 where a multilayer perceptron

(MLP) is used to estimate $p(w_i|w_{i-n+1}\dots w_{i-1})$. There is an output neuron for each word w_i in a vocabulary Ω', a subset of the most frequent words of the task vocabulary Ω, allowing to increase computation of output layer.[3] The input of the NN LM is composed of the sequence $w_{i-n+1},\dots,w_{i-1}$. A local encoding scheme would be a natural representation of the input words, but it is not suitable for large vocabulary tasks due to the huge size of the resulting NN. To overcome this problem, a distributed representation for each word is learned by means of a projection layer. The mapping is learned by backpropagation in the same way as the other weights in the NN. After the projection layer, a hidden layer with non-linear activation function is used and an output layer with the softmax function will represent the N-gram LM probability distribution. The projection layer can be removed from the network after training, since it is much more efficient to replace it by a pre-computed table which stores the distributed encoding of each word. In order to alleviate problems with rare words, the input is restricted to words with frequency greater than a threshold K in training data.

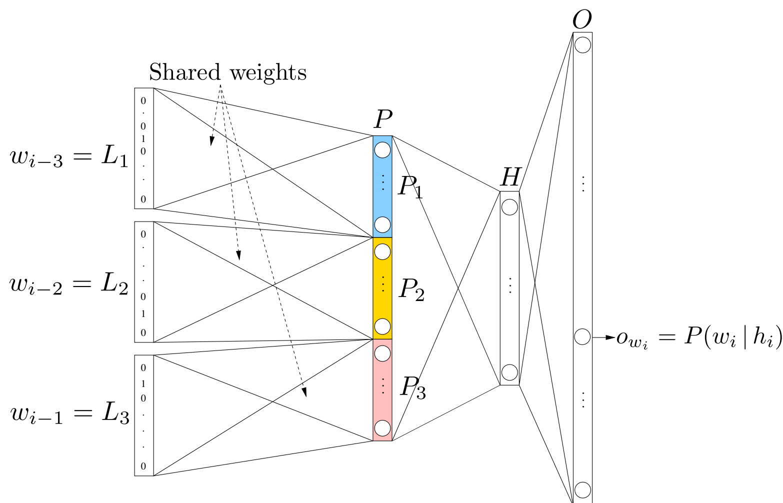

Fig. 1: The architecture of a 4-gram NN LM during training. In this example, the history of word w_i is represented by $h_i = w_{i-1,}, w_{i-2}, w_{i-3}$.

Pre-computation of softmax normalization constants [18] has been proposed as a practical solution to the high computational cost of NN LMs output layer. Following our previous work, in order to compute 4-gram probabilities by using NN LMs, lower order NN LMs are needed to compute N-gram probabilities for

[3] Relating to the output layer, several techniques exist to reduce its computational cost [8, 15, 18, 2, 12]. Short-list approach [15] is the one used in this work and requires to combine the NN LMs with another LM for words out of the short-list.

contexts which softmax normalization constant has not been pre-computed. In this way, the model performance decreases but a significant time speed-up is obtained. One drawback of this approach is that it forces the training of several NN LMs for different N-gram orders.

3 Skipping NNLMs

Since skipping techniques consist of learning several LMs based on different ways of removing words from the past history, a natural way of introducing Skipping NN LMs is by training a sole MLP but replacing random positions in context history by a new special symbol ⟨NONE⟩. There exist 2^{N-1} different ways of replacing some of the $N-1$ words by ⟨NONE⟩. After training, 2^{N-1} models are available, each one with a fixed pattern of skipped positions. The perplexity (PPL) of each possible skipping model will be evaluated in this paper. Additionally, skipping the proper positions, it is possible to convert a 5-gram NN LM into a 4-gram, 3-gram and so on. Softmax normalization constants can be pre-computed for this unique model using LM training data. When a constant is not found, instead of using a totally different model with lower order, it is possible to use the same model but with skipping positions which reduce its order. We have evaluated this approach, a comparison of PPL results of skipped lower order models and *true* lower order NN LMs will be performed in this paper.

It is worth mentioning that the technique described in [11], also coined as continuous skip-ngram model by their authors, is not related to this work despite the similarity in nomenclature: the main purpose of [11] consists in learning a continuous representation of the lexicon of a task in a way that captures the semantic similarity w.r.t. the task. This representation could be used, for instance, in NN LMs. Their continuous skip-ngram models are log-bilinear classifiers which receive a word at the input layer and try to predict the neighboring (past and future) words.

4 Experimental setup

The NN LMs used as baseline for our experiments are fully described in [18, 19]. These models will be extended to construct the Skipping NN LM as described in the previous section. The experiment is conducted on a task based on the transcriptions of the IAM-DB [9] handwritten dataset. The training material to estimate the LMs is taken from LOB [7], Wellington [1] and Brown [3] corpora. Validation and test sets were taken from IAM-DB corpus. Since IAM-DB texts are extracted from LOB corpus, IAM-DB validation and test sentences have been removed from LOB text data. PPL results will be presented for validation and test parts of the IAM-DB.

Backprogation algorithm and L2 regularized cross-entropy loss function are used to train NN LMs and Skipping NN LMs. A projection layer with 256 neurons and a hidden layer with 200 neurons has been used based in previous works. For Skipping NN LMs, the input is stochastically perturbed, introducing

the ⟨NONE⟩ symbol in zero or more random positions of the input layer. For every training pattern, the number of skips is sampled from a multinomial distribution. Given a number of skips, the positions of them are uniformly distributed. The multinomial distribution is defined to assign a probability of 50% for no skips at all, whereas the other 50% is distributed over one, two or more skips following a hyperbolic trend (see Table 1).

Table 1: The number of skips is sampled from the following multinomial distributions, given N-gram order.

# skips	bigram	3-gram	4-gram	5-gram
0	50%	50%	50%	50%
1	50%	33%	27%	24%
2	–	16%	13%	12%
3	–	–	9%	8%
4	–	–	–	6%

Once the model is trained, 2^{N-1} different LMs can be build, having each one a different combination of skips or ⟨NONE⟩ tokens in its input layer. Although one of the main benefits of Skipping NN LMs is to emulate lower order N-grams in order to greatly simplify the speed-up technique based on memorizing softmax normalization constants, in this work we focus on: first, investigating these emulation capabilities, and, second, computing the probabilities of the first N1 words of a sentence (for higher order N-grams) before the complete word history is available.

5 Experimental results

First, an evaluation of the PPL trend for validation data depending on the none tokens has been performed (see Figure 2). PPL is computed for all of the 2^{N-1} possible skipping positions. To better interpret the results, let us remark that the skipping number is a right-to-left bits mask where "0" indicates no skip and "1" indicates a skip. So, the skip number 7 in a 5-gram refers to the binary representation 1110, meaning that probability of N-gram at position i is computed by using word $i-1$ and three ⟨NONE⟩ tokens. The complete Skipping NN LMs models consider the combination of the different LMs associated to the skipping masks. We have performed this combination, for each N-gram order, using the `compute-best-mix` tool from SRILM toolkit [17]. The obtained results are very similar and competitive with those of standard NN LMs but they are not able to outperform this baseline.

As can be observed in Figure 2, the Skipping NN LMs with a skipping mask without ⟨NONE⟩ performs as well as standard NN LMs while the presence of ⟨NONE⟩ downgrades the results and this effect is more pronounced when ⟨NONE⟩ is close to the word to be predicted.

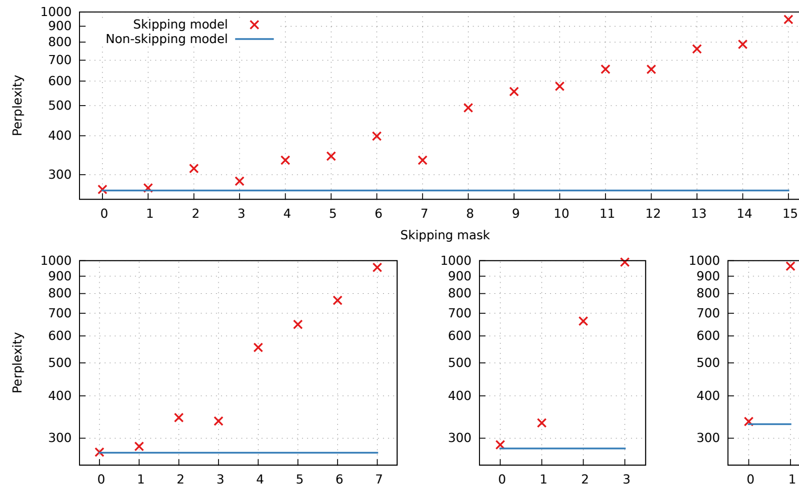

Fig. 2: PPL measured on validation set for Skipping NN LMs varying the skipping mask together with the baseline PPL of the non-skipping NNLM. The vertical axes of the four plots show the PPL and the skipping mask appears on the horizontal axes using the binary notation described in the paper. Upper plot corresponds to 5-grams. Lower plots correspond, from left to right, to 4-grams, trigrams and bigrams.

In order to evaluate the future ability of Skipping NN LMs to simplify the work presented at [18], Table 2 shows the PPL obtained by using no skipping NN LMs and Skipping NN LMs where the skipping positions are set to model equal or lower orders. Values from the first row of each table are the PPL values for each one of the NN LMs. The remaining rows contain the values from each of the Skipping NN LMs. The last non-void value of these rows is the PPL for the set without perturbing its input. The previous values are obtained after using skipping configurations that emulate a lower order ngram. For example, the 4-gram value for 5-gram Skipping NN LM is obtained after using the skipping mask that replaces the furthest word from the context by ⟨NONE⟩. To get the trigram value we replace the next word from the context too, and so on.

We can observe that the column values for each table are quite similar. This means that the PPL values for each one of the NN LMs are similar to the ones obtained for the Skipping NN LMs which can compute them using the adequate skipping configuration. Therefore, it is possible to train a single Skipping NN LM to imitate the behaviour of several NN LMs. Backing off is simpler and more efficient on Skipping NN LMs, since we just need to apply the required mask to our N-gram and get the probability as usual.

Table 2: PPL results for IAM-DB validation (left) and test (right). For skip models the PPL has been computed for 0 skips and for skip combinations which simulate a lower order model.

Model	Ngram order				
	1	2	3	4	5
5gr No skip	–	–	–	–	267
4gr No skip	–	–	–	272	–
3gr No skip	–	–	280	–	–
2gr No skip	–	330	–	–	–
5gr skip	946	334	286	272	269
4gr skip	955	337	284	273	–
3gr skip	990	333	287	–	–
2gr skip	963	336	–	–	–

Model	Ngram order				
	1	2	3	4	5
5gr No skip	–	–	–	–	309
4gr No skip	–	–	–	313	–
3gr No skip	–	–	319	–	–
2gr No skip	–	376	–	–	–
5gr skip	1019	378	326	311	309
4gr skip	1025	383	324	313	–
3gr skip	1045	377	327	–	–
2gr skip	1026	381	–	–	–

6 Conclusions and future work

This work is, to the best of our knowledge, the very first attempt to integrate the well known technique of Skipping N-grams into NN LMs. NN LMs are capable of learning distributed representations which might explain that non additional gain is obtained by including the skipping technique. Coming back to the example of *the little boy* which *likes pizza*, from Section 2.1, it is possible that in a large corpus the contexts of words *boy* and *girl* are similar enough to make it possible for the MLP to learn a similar representation for these words so that the effect of skipping is diminished.

On the other side, the capability of Skipping NN LMs to emulate lower order NN LMs makes them very suitable for greatly simplifying the speed-up technique based on pre-computation of softmax normalization constants [18] since these models rely on lower order models when a constant is not found.

As a future work, we plan to investigate the effect of this technique in larger corpora to give more support to the preliminary results presented here and to study the effect of the new LM in the overall error of a recognition system.

Acknowledgments

This work has been partially supported by the Spanish Government TIN2010-18958.

References

1. Bauer, L.: Manual of Information to Accompany The Wellington Corpus of Written New Zealand English. Tech. rep., Department of Linguistics, Victoria University, Wellington, New Zealand (1993)

2. Bengio, Y., Senecal, J.S.: Adaptive importance sampling to accelerate training of a neural probabilistic language model. IEEE Transactions on Neural Networks 19(4), 713–722 (2008)
3. Francis, W., Kucera, H.: Brown Corpus Manual, Manual of Information to accompany A Standard Corpus of Present-Day Edited American English. Tech. rep., Department of Linguistics, Brown University, Providence, Rhode Island, US (1979)
4. Goodman, J.T.: A Bit of Progress in Language Modeling - Extended Version. Tech. Rep. MSR-TR-2001-72, Microsoft Research, One Microsoft Way Redmond, WA 98052 (2001)
5. Huang, X., Alleva, F., Hon, H.W., Hwang, M.Y., Lee, K.F., Rosenfeld, R.: The SPHINX-II speech recognition system: an overview. Computer Speech and Language 7(2), 137–148 (1993)
6. Jelinek, F.: Statistical Methods for Speech Recognition. Language, Speech, and Communication, The MIT Press (1997)
7. Johansson, S., Atwell, E., Garside, R., Leech, G.: The Tagged LOB Corpus: User's Manual. Tech. rep., Norwegian Computing Centre for the Humanities, Bergen, Norway (1986)
8. Le-Hai, S., Oparin, I., Alexandre, A., Gauvaing, J.L., Franois, Y.: Structured Output Layer Neural Network Language Model. In: Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). vol. 11, pp. 5524–5527 (2011)
9. Marti, U.V., Bunke, H.: The IAM-database: an English sentence database for offline handwriting recognition. International Journal of Document Analysis and Recognition 5, 39–46 (2002)
10. Martin, S., Hamacher, C., Liermann, J., Wessel, F., Ney, H.: Assessment of smoothing methods and complex stochastic language modeling. In: Proc. 6th European Conference on Speech Communications and Technology (Eurospeech). vol. 5, pp. 1939–1942 (1999)
11. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. CoRR abs/1301.3781 (2013)
12. Mnih, A., Kavukcuoglu, K.: Learning word embeddings efficiently with noise-contrastive estimation. In: Advances in Neural Information Processing Systems 26, pp. 2265–2273 (2013)
13. Ney, H., Essen, U., Kneser, R.: On structuring probabilistic dependences in stochastic language modeling. Computer Speech and Language 8(4), 1–38 (1994)
14. Rosenfeld, R.: Adaptative statistical language modeling: A maximum entropy approach. Ph.D. thesis, Carnegie Mellon University (1994)
15. Schwenk, H.: Continuous space language models. Computer Speech and Language 21(3), 492–518 (2007)
16. Siu, M., Ostendorf, M.: Variable n-grams and extensions for conversational speech language modeling. IEEE Trans. Speech and Audio Processsing 8(1), 63–75 (2000)
17. Stolcke, A.: SRILM: an extensible language modeling toolkit. In: Proceedings of the International Conference on Spoken Language Processing (ICSLP). pp. 901–904 (2002)
18. Zamora-Martínez, F., Castro-Bleda, M., España-Boquera, S.: Fast Evaluation of Connectionist Language Models. In: International Work-Conference on Artificial Neural Networks, LNCS, vol. 5517, pp. 33–40. Springer (2009)
19. Zamora-Martínez, F., Frinken, V., España-Boquera, S., Castro-Bleda, M., Fischer, A., Bunke, H.: Neural network language models for off-line handwriting recognition. Pattern Recognition 47(4), 1642 – 1652 (2014)