



UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Generación automática de memes de Internet
a través de una red neuronal profunda

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Albert Manuel Orozco Camacho

TUTOR

Dr. Ivan Vladimir Meza Ruiz

Ciudad Universitaria, CD. MX., 2018



Índice general

Agradecimientos	1
Resumen	3
1. Introducción	5
1.1. Memes de Internet y <i>memética</i>	7
1.2. Un esbozo sobre el aprendizaje profundo	9
1.3. Objetivo y metas de la tesis	12
1.4. Estructura de la tesis	13
2. Marco teórico	15
2.1. Aprendizaje automático	16
2.1.1. Las diferentes formas con las que una máquina aprende .	17
2.2. Aproximando funciones con neuronas	19
2.2.1. Inspiración a partir de la biología	20
2.2.2. El modelo de cómputo neuronal	20
2.2.3. El perceptrón de Rosenblatt	22
2.2.4. El perceptrón multicapa	26
2.2.5. Retroalimentación para aprender	28
2.3. Redes neuronales convolucionales	35

2.3.1.	La capa convolucional	36
2.3.2.	La <i>convolución</i>	39
2.3.3.	Filtros vistos como arreglos de neuronas	41
2.3.4.	La “ <i>capa</i> ” de activación	43
2.3.5.	La capa de <i>pooling</i>	43
2.4.	Redes neuronales recurrentes	45
2.4.1.	¿Cómo funciona la arquitectura recurrente?	47
2.4.2.	Propagación hacia atrás, a través del tiempo	49
2.4.3.	Redes de gran memoria de corto plazo (LSTM)	51
3.	Red neuronal para descripciones de memes	55
3.1.	Formalización del problema de aprendizaje	56
3.2.	Inception <i>V3</i>	57
3.2.1.	Factorización de convoluciones	59
3.3.	Poniendo todo junto	63
3.4.	Aprendizaje por transferencia	64
3.5.	La arquitectura en acción	67
4.	Experimentación y evaluación del desempeño de la red	69
4.1.	Estructura del conjunto de datos	70
4.2.	Generación de leyendas	72
4.3.	Experimentos	74
4.3.1.	Experimentos exploratorios	75
4.3.2.	Experimentos que involucran la afinación de una arquitectura convolucional profunda	78
4.3.3.	Experimentos que involucran una arquitectura convolucional superficial	83

<i>ÍNDICE GENERAL</i>	v
4.4. Evaluación del desempeño de la LSTM	87
5. Conclusiones	91
5.1. El <i>flujo</i> de los memes	92
5.2. Trabajo futuro	93
A. Resultados anecdóticos	99
Bibliografía	104

Agradecimientos

A mi madre, la más maravillosa del mundo, por siempre apoyarme en mis decisiones; por aprender cada día a entender lo que quiero y lo que busco.

A mi padre, el personaje con el mejor gusto musical que conozco, por amenizar mis sesiones de trabajo desde la distancia y echarme porras en mis actividades.

A mis abuelos y a mi tía, por creer en mí a pesar de la distancia.

A mis amigos: pocos pero verdaderos. Incluyendo al más fiel: mi perro.

A mi mentor de vida profesional, Dr. Ivan Vladimir Meza, por acogerme en el IIMAS desde el principio, sin siquiera saber programar, enseñarme lo que es la investigación científica, transmitirme la pasión por la inteligencia artificial, el aprendizaje profundo y el procesamiento de lenguaje natural. Por ser alguien en quien siempre puedo contar y el mejor consejero para un alumno de licenciatura. Y, simplemente, por el placer de hacer y discutir lo que nos apasiona.

Resumen

Se le llama ***meme de Internet*** a la unidad de propagación de información vía electrónica, que muchas veces utiliza bromas, chistes y/o rumores en su estructura[19]. La exorbitante popularidad del Internet ha permitido el crecimiento del uso de memes, en particular, con el formato de imagen y leyenda corta. Más aún, el problema de la generación automática de memes de Internet ha sido escasamente estudiado dentro de las disciplinas del procesamiento de lenguaje natural y visión computacional.

En este trabajo, se aborda dicho problema mediante el uso de aprendizaje profundo, una rama del aprendizaje automático que ha redefinido el *estado del arte* de diversas tareas en inteligencia artificial. Se recolectó una gran cantidad de memes de Internet, separando la imagen de su leyenda y priorizando el modelo que predomina en el sitio web `MemeGenerator.net`¹. Así, se entrenó un modelo profundo que incluye una red convolucional y una red recurrente de tipo *LSTM*. Para la implementación, se utilizaron primordialmente las bibliotecas `Tensorflow`[1] y `Keras`[5] que funcionan para el lenguaje de programación Python.

El desempeño de la misma fue evaluado por su capacidad de generar una leyenda para una imagen no perteneciente al conjunto de datos de entrenamiento. Además se midió de la perplejidad del modelo de lenguaje generado contra un corpus extraído aparte.

¹<https://memegenerator.net/>

When you plant a fertile meme in my mind you literally parasitize my brain, turning it into a vehicle for the meme's propagation in just the way that a virus may parasitize the genetic mechanism of a host cell.

Richard Dawkins [6]

1

Introducción

El siglo XXI ha traído consigo una transformación radical en las formas de comunicación entre personas. Hoy en día, el uso de redes sociales *democratiza* el acceso a la información más relevante ocurriendo en tiempo real, sin tener que esperar a que un medio masivo publique una noticia sobre ello. Cualquier persona con una cuenta de *Twitter*¹ puede tomar una fotografía, subirla a la *web* y etiquetarla de la manera en que más le convenga (mediante *hashtags*, por ejemplo). Dependiendo de muchos factores, incluyendo el alcance del *tuit* y el número de seguidores de la cuenta de Twitter, es posible que la imagen se vuelva más popular de lo que una nota publicada por algún periodista serio o famoso pueda alcanzar.

Este fenómeno también se explica por las intenciones que llevan a la gente a interactuar dentro de las redes sociales. Al usuario se le da la libertad de hacer que el contenido de su *timeline (TL)* sea tan ocioso, o tan serio, como éste quiera, siguiendo en el caso

¹Red social de intercambio de mensajes cortos. URL: <http://www.twitter.com>

de Twitter. Cuando un tuit acompañado de una imagen se vuelve *viral*, surge un fenómeno de propagación en el cual el poder del *tuitero* permite re-etiquetar la imagen sin perder la esencia de la idea original transmitida por la misma.

En general, este fenómeno no es exclusivo de Twitter y ha ocurrido en Internet desde que la comunicación entre personas se efectuaba exclusivamente por medio de correos electrónicos. Para englobar a las diversas maneras en las que se *viraliza* cierta información en la web en un solo concepto, se ha popularizado el término **meme (de Internet)**, el cual pretende *discretizar* la información cultural en unidades capaces de pasar de persona a persona, gradualmente escalar en un fenómeno social compartido e incluso *evolucionar* [19].

Paralelamente, el constante incremento en el uso de redes sociales genera un cúmulo de datos esparcidos por el Internet. De acuerdo al estudio descrito en [4] y publicado en 2016, el 46 % de la población mundial son usuarios de Internet y el 31 % son usuarios activos de alguna red social. Además, durante 2015 se produjo un aumento de usuarios equivalente a 10 % en los dos rubros descritos anteriormente. Consecuencia de esto es que a diario, el intercambio de información favorece a la riqueza y complejidad de nuevas ideas que se acumulan en grandes servidores pero cuya relevancia está empezando a ser explorada.

Dentro de la rama de las ciencias de la computación, conocida como **inteligencia artificial**, el **aprendizaje automático** (*machine learning* en inglés) propone técnicas capaces de lograr que, a partir de grandes cantidades de datos, un sistema de cómputo revele información oculta para muchos seres humanos. El auge del subconjunto de *algoritmos de aprendizaje automático* conocido como **aprendizaje profundo** (*deep learning* en inglés), trae consigo un importante adelanto en el desempeño de computadoras al realizar habilidades humanas. Dos ejemplos significativos incluye el reconocimiento de imágenes y rostros mediante visión computacional y la generación automática de texto coherente en algún idioma.

Motivado por lo establecido en los párrafos anteriores, en el presente trabajo se explorarán dos modelos de aprendizaje profundo para entrenar, identificar y, finalmente, etiquetar memes de Internet. Se considerará únicamente el caso de una imagen (la cual, en la mayoría de los casos, incluye a un solo personaje) y una *leyenda* asociada. Por lo tanto, se presentará un método para hacer que la computadora aprenda al personaje dentro del meme y la información textual que transmite; posteriormente se experimentará etiquetando memes no antes vistos por dicha computadora.

1.1. Memes de Internet y *memética*

El término *meme* fue acuñado por el biólogo Richard Dawkins en su libro “*The Selfish Gene*” de 1976. Mediante una analogía con el papel del *gen* en la evolución darwiniana, Dawkins propuso una teoría cultural en la cual el meme es visto como la unidad que se propaga por generaciones y sobrevive mediante un proceso semejante a la selección natural. Dawkins conjeturó que la teoría de la evolución de Darwin es una instancia particular de un proceso que se puede encontrar en otras áreas; en particular, es suficiente que cualquier concepto que incorpore las propiedades de *longevidad, fecundidad y fidelidad de copias* para que éste tenga un comportamiento evolutivo a través del tiempo [7].

Hoy en día, se le conoce como *meme* principalmente al objeto proveniente de Internet y que incorpora, en la mayoría de los casos, una imagen y una leyenda que cuenta algo sobre la imagen (Figura 1.1). Es importante recalcar el aspecto humorístico, muchas veces incluso irónico, que caracteriza al meme de la actualidad ya que ello contribuye a la difusión de los mismos por la web. Sin embargo, es el diseño *centrado al usuario* característico de la llamada *Web 2.0* lo que mayormente facilita la propagación de memes.

Dentro de esta estructura tecnológica, las tres propiedades adscritas por Dawkins a cualquier objeto evolutivo se satisfacen para los memes: la digitalización permite una transmisión casi sin inter-

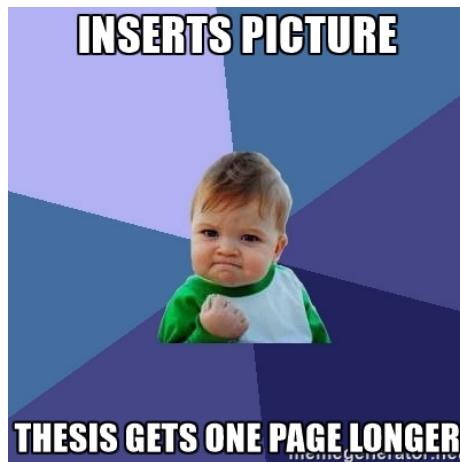


Figura 1.1: Clásico ejemplo de un personaje *memificado* junto con una de sus leyendas (descripciones). (Tomado de <http://www.memegenerator.net>.)

ferencias (fidelidad de copias), el número de copias compartidas en una unidad de tiempo es gradual – dada la facilidad de compartir información “de nodo a nodo” – (fecundidad) y el aumento en la longevidad de la información es respaldado por la capacidad de almacenamiento indefinido en los servidores de la web. *Reddit*² es uno de los sitios web con mayor flujo y contenido de memes; su eslogan “*the front page of the Internet*” resume la importancia cultural que ha alcanzado al modificar las formas de obtener y discutir información de cualquier índole. Lo que Dawkins no se imaginó en los años 70’s es que el meme se convertiría en la mejor manera de encapsular los aspectos más fundamentales de Internet [19].

La trascendencia del concepto de Dawkins provocó el surgimiento de la *memética*, la disciplina sociobiológica que extrae el concepto de evolución de la teoría de Darwin para colocar al meme como instrumento de supervivencia, un *replicador*. Originalmente, Dawkins sugirió como ejemplos de memes a frases pegadizas, tonos de audio, modas, habilidades o simplemente ideas. Más aún, según Dawkins el meme es una “unidad de información que reside en un cerebro” [6], una afirmación que sugiere la relevancia que el alcance

²Sitio web de marcadores sociales e intercambio de enlaces a contenidos de Internet. URL: <http://www.reddit.com>

de las redes sociales tiene en la población actual, incluso mayor a la que otros medios de comunicación – como la televisión o la radio – alcanzaron desde su concepción.

1.2. Un esbozo sobre el aprendizaje profundo

El ser humano es complejo. Desde una perspectiva computacional, cualquier persona realiza tareas que constituyen una inteligencia que hasta más de la mitad del siglo XX parecía inimitable. Por otro lado, si hay una capacidad para la cual el cómputo actual ha superado a la inteligencia humana es la de procesar números; la computadora es, abstractamente, una máquina que entiende muy bien cadenas de ceros y unos. La apuesta moderna consiste, entonces, en transformar las habilidades humanas en problemas numéricos para los cuales la computadora puede *aproximar* una solución.

Una de las primeras maneras de estudiar a la inteligencia humana, de manera computacional, pretende resaltar las capacidades de buscar y encontrar caminos para resolver situaciones complicadas (en juegos, por ejemplo). Al implementar algoritmos de búsqueda de soluciones, es inminente encontrarse de frente con casos en los que las capacidades de una computadora, por más nueva y equipada que sea, no sean suficientes para lograr un desempeño equiparable al del ser humano. En términos más formales, se dice que el tamaño del *espacio de estados*, el cual define el conjunto de conocimiento que puede tener un *agente racional* mientras llega a una meta, se vuelve computacionalmente intratable. Las técnicas más innovadoras, en inteligencia artificial, para tratar con problemas que involucran dichos espacios de estado, corresponden a la modelación del entorno de un agente de manera numérica y a la aproximación de resultados por medio de algoritmos probabilísticamente efectivos [17].

Si a lo establecido anteriormente se le suma la tendencia a elaborar modelos matemáticos que “*aprenden*” mediante una gran cantidad de ejemplos, encontramos a lo que se conoce como *aprendizaje automático*. Estadísticamente hablando, aprender significa que

estos modelos *convergen* a un estado deseado dependiendo de la tarea a realizar. Cuando la robustez del algoritmo (o *heurística*) numérico propuesto depende directamente de la existencia de un conjunto de datos **etiquetados**³ de *entrenamiento*, se habla de un **aprendizaje supervisado**. En ausencia de dicho conjunto de datos, es posible realizar búsquedas de patrones sobre los datos (los cuales son, digamos, *desconocidos*); a esto se le conoce como **aprendizaje no supervisado**.

El cerebro humano no es solamente una máquina que pueda procesar información. En este sentido, la inteligencia artificial se ha inspirado en la anatomía del sistema nervioso para perfeccionar al aprendizaje automático. Con esto surgen los modelos conocidos como **redes neuronales**: arquitecturas de neuronas interconectadas que, dado un conjunto de ejemplos, buscan fortalecer ciertas conexiones para construir un modelo abstracto sobre la realidad observada por un agente. Los conceptos introducidos en estos dos últimos párrafos serán profundizados en el siguiente capítulo.

El modelo más simple de una red neuronal corresponde al del perceptrón, el cual puede ser visualizado como un conjunto de neuronas que reciben la información (*de entrada*) y que se combinan en una neurona *de salida*. Posteriormente, apilando varios perceptrones en *capas*, se logran modelos más complejos. Dependiendo de la conexidad de cada capa, de la dirección hacia donde fluye la información y de la existencia de ciclos en el modelo, surge una especie de “*zoológico de redes neuronales*” (término acuñado en [22]). El éxito en la consecución de estos modelos consiste en el incremento en el número de capas de neuronas. Cuando un modelo alcanza una cantidad considerable de las mismas, se le refiere como *red neuronal profunda*.

³ Con *etiquetados* nos referimos a un conjunto de datos que posee una serie de valores de entrada al algoritmo de aprendizaje, asociados con valores de salida. Para una explicación más profunda, consultar el Capítulo 2.

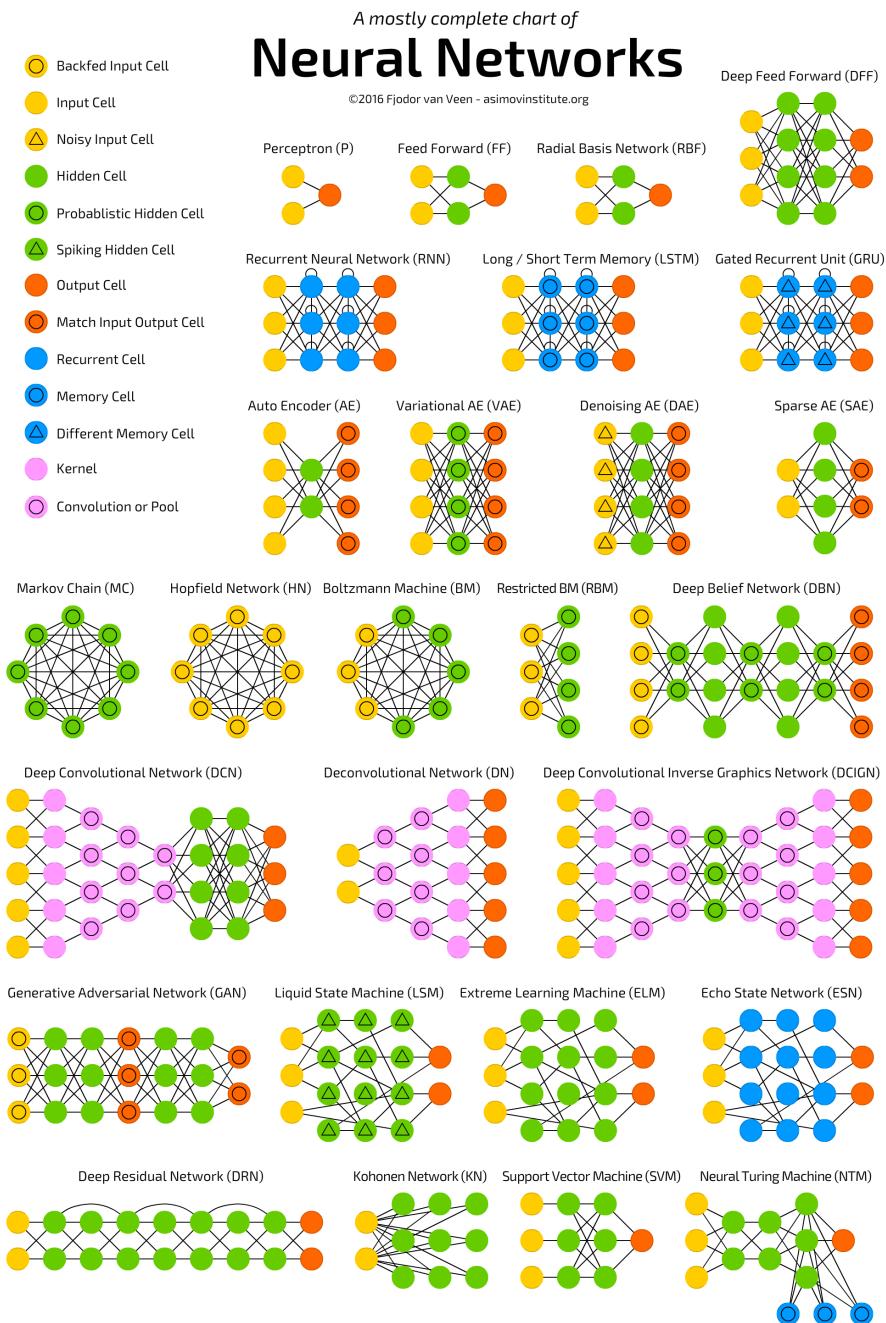


Figura 1.2: El zoológico de redes neuronales conocido hasta 2016. (Tomado de <http://www.asimovinstitute.org>.) [22]

De lo anterior, surge la sub-rama conocida como **aprendizaje profundo** y que es la que trae consigo mejoras considerables en tareas computacionalmente intratables. Entre los ejemplos que justifican este *boom* tecnológico, está el procesamiento de señales de audio, el reconocimiento de rostros, el dominio del juego de mesa chino “go”, la traducción automática de un idioma a otro, la predicción del comportamiento de una bolsa de valores, el reconocimiento de imágenes y la generación automática de descripciones de imágenes. Las dos tareas mencionadas al final son la *raison d'être* de esta tesis.

1.3. Objetivo y metas de la tesis

En el presente trabajo se expondrá el diseño y la implementación de un sistema computacional capaz de extraer las características más importantes que relacionan una imagen con su leyenda en un meme, utilizando herramientas de aprendizaje profundo. A partir de lo anterior, dada una imagen desconocida por el sistema, producir la leyenda que más haga sentido para formar un meme, con los conceptos (características) aprendidos en el primer paso.

En la consecución de lo anterior, se habrá de lograr las siguientes metas:

- Recolectar una importante cantidad de memes de Internet, clasificándolos por idioma. La mayor parte de dicho conjunto de datos será para entrenar a la red neuronal que se desarrollará, mientras que la otra será para evaluar el desempeño de la misma.
- Extraer la leyenda (texto) de cada imagen, de manera que ambas partes queden separadas pero sin perder su asociación.
- Diseñar e implementar la red neuronal profunda que extraerá las principales características de cada imagen y su leyenda asociada.

- Implementar un mecanismo para digitalizar las imágenes y que la red neuronal sea capaz de trabajar con éstas y sus leyendas asociadas.
- Entrenar la red neuronal con el conjunto de datos recolectado de Internet.
- Evaluar el desempeño de la salida de la red, asociando imagen con leyenda.

1.4. Estructura de la tesis

En el presente documento, se seguirá la siguiente estructura:

- Marco teórico. *Se cubrirán cabalmente todos los conceptos y resultados teóricos que justifican la parte experimental del trabajo.*
- Red neuronal para descripciones de memes. *Se presentará la arquitectura de la red neuronal propuesta para llevar a cabo las metas propuestas.*
- Evaluación del desempeño de la red. *Se presentarán algunas métricas utilizadas para evaluar el desempeño de arquitecturas neuronales profundas y se comparará el desempeño de la experimentación realizada con el estado del arte.*
- Conclusiones

2

Marco teórico

• **C**ÓMO es que la mente humana llega a etiquetar una imagen? A pesar de que las técnicas y algoritmos que se presentarán en esta tesis están diseñados para su implementación en una computadora, la filosofía que hay detrás de ellos se inspira en la del funcionamiento del cerebro humano. Antes de la conceptualización de la **inteligencia artificial** (IA) como disciplina científica, diversos pensadores como Ramón Llull, Thomas Hobbes y Leonardo da Vinci, ya habían propuesto la idea de *mecanizar* el razonamiento mediante una máquina apta para realizar cálculos numéricos. [17]

En 1642, Blaise Pascal construyó la primera máquina capaz de realizar cálculos (sumas y restas): la *Pascalina*. Las capacidades de ésta última fueron superadas por la máquina de Gottfried Wilhelm Leibniz, la cual, además de sumar y restar, también podía multiplicar y sacar raíces. La idea central detrás de todo lo anterior radicaba en aceptar la existencia de un conjunto de *reglas lógicas* que gobernan a la mente y al pensamiento. Esto arrancó cuando Aristóteles

formuló una serie de silogismos capaces de generar conclusiones automáticas a partir de premisas iniciales. Sin embargo, la *tradición lógica* de la IA no pudo superar dos obstáculos principales: pasar de conocimiento *informal* a términos formales y resolver un problema fuera de la teoría.[17].

2.1. Aprendizaje automático

En la literatura más tradicional de IA, se acostumbra motivar y guiar la teoría por medio de las capacidades y limitaciones de un **agente** (racional). Esto es una abstracción de un ente real, capaz de percibir información de su ambiente, procesarla y tomar decisiones. El agente será equipado del conocimiento necesario para decidir la mejor opción según lo que exista en su entorno [17].

La arquitectura interna del agente se compone de una máquina finita de *estados* (con sus variantes), cuyas transiciones están determinadas por *reglas de interpretación*. Cada vez que un agente cambia de estado, se produce una *acción*. La naturaleza de esta estructura nos lleva a reducir muchos de los problemas de la IA en búsquedas, en las cuales el agente puede o no saber *a priori* el conjunto de posibles resultados que puede encontrar.

Computacionalmente hablando, el espacio de búsqueda de varios problemas interesantes en IA se vuelve intratable, por lo que no es posible simplemente codificar todos los posibles escenarios dentro de la *base de conocimientos* de un agente. El camino hacia la solución a este problema comienza con dos observaciones:

- un agente puede (y debe) ser capaz de manejar la incertidumbre de su entorno dependiendo de la verosimilitud de un evento;
- el cerebro humano pasa mucho tiempo adquiriendo conocimiento a través de experiencias que previamente *aprendió*, por ende, no es descabellado enseñar a un agente a que automáticamente se genere una descripción de su entorno.

El **aprendizaje automático** (*machine learning* en inglés) es la rama de la IA que trata con algoritmos que son capaces de tratar con información estructuralmente incierta (o ruidosa) y generar modelos internos para la toma de decisiones. Un agente equipado con la habilidad de aprender automáticamente se beneficia de grandes cantidades de ejemplos de un problema, pues su objetivo ahora es llegar al mejor modelo, estadísticamente hablando.

2.1.1. Las diferentes formas con las que una máquina aprende

La mayoría de los retos computacionales tienen como solución un programa determinístico. Su objetivo radica en que dada una entrada \vec{x} , el programador diseña en su mente un algoritmo h que calcule la salida deseada $z = h(\vec{x})$.

En cambio para estimar z , en aprendizaje automático, el programador o científico de datos propone un *modelo* h que sea *entrenado* con respecto a una función objetivo (o de error) J , optimizando así, una posible solución \hat{z} . Todo esto se hace teniendo en cuenta un conjunto de *parámetros* del modelo y un conjunto de datos muestrales \mathcal{D} .

Las maneras de hacer aprendizaje automático dependen de cómo está constituido el conjunto de datos con el que se trabaja. En **aprendizaje supervisado**, se modelan fenómenos a través de muchos ejemplos *etiquetados*, es decir, el conjunto de datos puede ser visto como

$$\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}. \quad (2.1)$$

En este caso, se dice que \mathcal{D} contiene n ejemplos etiquetados: a \vec{x} y a \vec{y} se les puede ver como instancias de posibles entradas y salidas del problema z , respectivamente. Normalmente, \vec{x} se compone de valores numéricos que fungen como *características* distintivas de la entrada. El vector \vec{y} puede contener valores de un conjunto discreto (categorías) o continuo.

Formalmente, suponiendo que $x_i \in \mathbf{X}$ e $y_i \in \mathbf{Y}$ un algoritmo

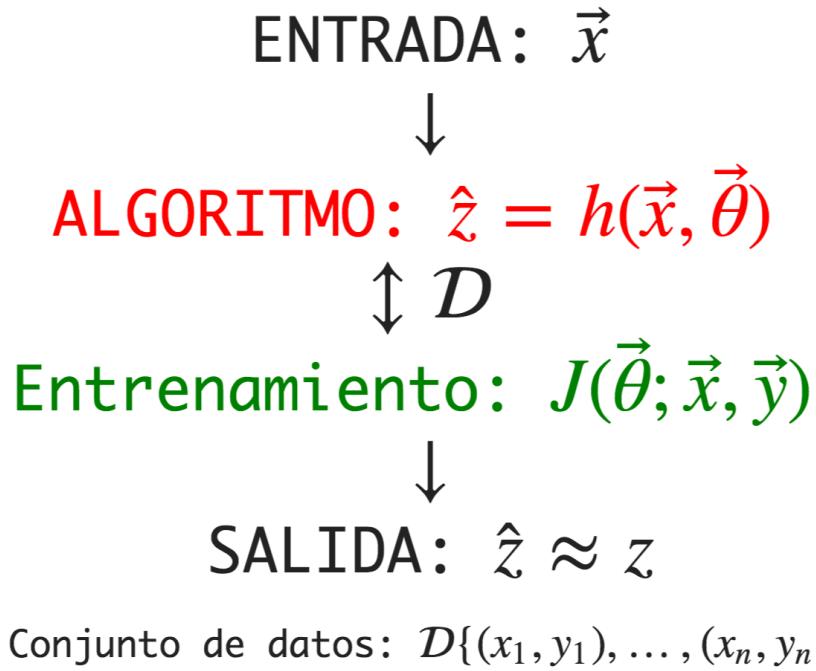


Figura 2.1: Las tres fases del aprendizaje automático supervisado. (Elaboración propia.)

h de aprendizaje automático tiene la forma

$$h : \mathbf{X} \longrightarrow \mathbf{Y}. \quad (2.2)$$

Si \mathbf{Y} es un conjunto finito, entonces decimos que el problema es de **clasiﬁcación**, mientras que si es infinito (y denso) es de **regresión**.

En la presente tesis, se trabajará únicamente con aprendizaje supervisado, pues cada uno de los memes de muestra (\mathbf{X}), está etiquetado con una o varias leyendas (\mathbf{Y}). El reto consiste en construir un modelo que sea capaz de capturar las características más importantes de la imagen y que las asocie a un *modelo de lenguaje*. Todo esto será profundizado más adelante.

Para terminar con la teoría general de aprendizaje automático, vale la pena destacar que es posible realizar **aprendizaje no supervisado**. Esto se logra mediante un conjunto de datos que ca-

rece de salidas \mathbf{Y} ; el objetivo de un agente será desvelar los patrones que comparten las características dadas, es decir, aprender a agrupar los datos mediante el uso de similitudes estadísticas. **Aprendizaje por refuerzo** es otra manera de hacer aprendizaje automático y se basa en un entrenamiento en el que el agente debe maximizar un puntaje debido a una retroalimentación dada por sus acciones. Esto último va más allá de los objetivos de esta tesis.

2.2. Aproximando funciones con neuronas

En computación, la palabra *aprendizaje* es sinónimo de “minimización de errores”, lo que lleva a cuestionarnos la existencia de una conexión entre este fenómeno estadístico con las maneras que, naturalmente, posee el cuerpo humano para comprender su medio ambiente. Concretamente, la tarea que tiene el computólogo ante sí consiste en transformar los problemas de aprendizaje automático en equivalentes compatibles con el funcionamiento del sistema nervioso humano. Dadas, las evidencias empíricas sobre el buen desempeño del cerebro humano en su cotidianidad, nos inspiramos en el comportamiento biológico del mismo para construir arquitecturas cuyo objetivo será el de aproximar funciones complejas, creando un novedoso paradigma de cómputo.

Con el fin de ilustrar al buen *performance* del cerebro humano, consideremos a las habilidades de reconocimiento perceptivo. Mientras una persona tarda de 100 a 200 ms en detectar un rostro familiar, una computadora con suficiente poder dura mucho más.[10] En contraste, se sabe que individualmente, una neurona es mucho más lenta que una compuerta lógica: mientras que ésta última tarda pocos nanosegundos en *comutar*, a la primera mencionada le puede tomar varios milisegundos en reaccionar a un estímulo.

2.2.1. Inspiración a partir de la biología

El sistema nervioso es una red *paralela* y *auto-organizada*. 86 mil millones de neuronas (aproximadamente) [25] conforman una arquitectura que funciona a través de la emisión de pulsos eléctricos y la reacción ante ellos. Dos neuronas están conectadas entre sí por medio de estructuras conocidas como *sinapsis*, a través de las cuales se transmiten señales eléctricas y químicas.

El cerebro es, además, un órgano que se adapta a las condiciones de su ambiente. Evidencia de ello es la creación de conexiones sinápticas entre neuronas (previamente desconectadas) y la modificación del mecanismo de las sinapsis existentes. Una vez que una neurona haya emitido una señal eléctrica, las adyacentes reciben la “*información*” por medio de canales de transmisión llamados *dendritas*. Estos impulsos son llevados hasta el *cuerpo* de la neurona para su procesamiento y, posteriormente, una reacción es transmitida a través del *axón* de la célula. Los organelos mencionados anteriormente constituyen las principales partes de la neurona que habrán de servir como estructuras fundamentales de las arquitecturas de aprendizaje a presentar en las siguientes secciones.[15]

2.2.2. El modelo de cómputo neuronal

Dadas varias entradas eléctricas, una neurona deberá de ajustar su reacción de manera proporcional a la intensidad de las dichas señales. Para formalizar el comportamiento de una neurona en términos matemáticos (y, por ende, computacionales), es imprescindible caracterizar las reglas que sigue una neurona para componer sus señales de entrada y manejarlas “globalmente” mediante una función. Cabe destacar que, por simplicidad y elegancia de los modelos a estudiar, resulta importante conocer la *sincronía* de la transmisión de información así como la presencia o ausencia de ciclos o bucles. Todo esto se puede englobar en un conjunto de características topológicas y algorítmicas que constituyen a una *red neuronal artificial* (ANN, por sus siglas en inglés; en adelante, abreviaremos ANN con NN).

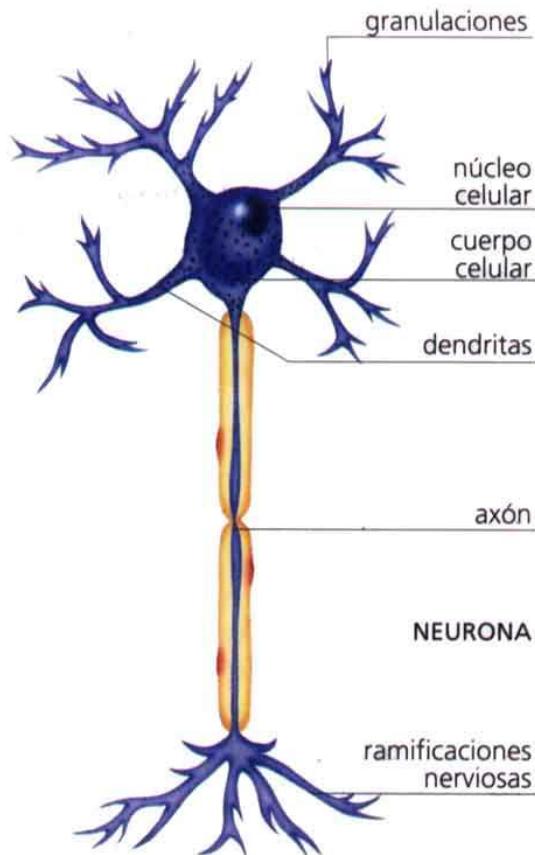


Figura 2.2: Ilustración de las partes más importantes de una neurona humana. (Tomado de (<https://psi121f.wordpress.com/2016/07/03/la-estructura-de-la-neurona-2/>))

Con respecto a la representación interna del conocimiento de una NN, estamos ante un conjunto de modelos que buscarán modelar la información de manera *asociativa*; análogamente al cerebro humano. Por ende, se requieren modelos que *relacionen* clasificaciones de objetos similares con representaciones internas similares. Con el fin de asegurarnos de ello, presentaremos más adelante una completa sección acerca de métodos para medir similitud. Una consecuencia importante de esto es que, de manera contraria, si se desea que dos objetos sean distintamente clasificados, entonces se les deben de dar dos representaciones totalmente diferentes.

Independientemente de la tarea que se desea aprender, siempre existirá alguna característica cuya importancia define el veredicto de la NN. Una forma de respaldar este hecho consiste en dedicar un gran número de neuronas a la identificación de dicha característica. Con ello, se aumenta la precisión de la NN en su toma de decisiones, contrastando con la existencia de neuronas defectuosas.

Finalmente, la existencia de un *zoológico de redes neuronales* se justifica con la tendencia que se sigue a diseñar modelos específicos para ciertas tareas. Si se sabe información *a priori* sobre los datos a procesar, es mejor integrarlas en el diseño de la arquitectura a dejar que ésta las aprenda durante el entrenamiento. Después de todo, en el cuerpo humano existen una gran cantidad de estructuras neuronales especializadas para funciones como visión y audición, muy distintas a otras presentes en el cerebro.

2.2.3. El perceptrón de Rosenblatt

El contenido de este apartado se basa principalmente de [10] y [15].

La idea de cómputo a través de redes neuronales fue tan trascendente desde su concepción, que, en 1943, McCulloch y Pitts la incluyeron en el mismo artículo que introdujo a los sistemas de transición con un número finito de estados [14]. Sin embargo, fue Rosenblatt, en 1958, quien propuso el primer modelo de aprendizaje supervisado para una NN. El perceptrón es la red neuronal más simple y, muchas veces, será uno de los bloques básicos de arquitecturas más complejas. La intuición matemática detrás de su estructura consiste en *separar linealmente* las entradas dadas en dos clases, lo cual se logra mediante un aprendizaje que va ajustando pesos de acuerdo a las salidas esperadas.

De acuerdo a la Figura 2.3, el perceptrón opera de la siguiente manera: dados los valores de entrada x_1, x_2, \dots, x_m y los pesos w_1, w_2, \dots, w_m ,

- se calcula una suma ponderada con los m pesos (*sinápticos*) del

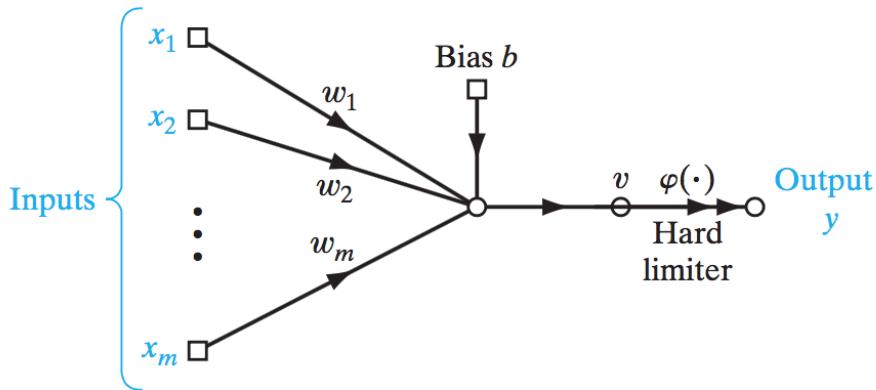


Figura 2.3: El perceptrón de Rosenblatt es un modelo gráfico cuya salida depende de las contribuciones lineales de cada una de las entradas, de un sesgo y de una función no lineal de activación. (Tomado de [10])

modelo,

- a la suma anterior, se le añade un valor de “*tendencia*” conocido como *sesgo* (*bias* en inglés); formalmente:

$$v = b + \sum_{i=1}^m w_i x_i \quad (2.3)$$

- finalmente, la salida del perceptrón se calcula aplicando una función *de activación* a \$v\$:

$$y = \Phi(v). \quad (2.4)$$

La función de activación juega el papel directo de clasificador: su salida debe decidir si la entrada pertenece, o no, a cierta clase. De ahí que en la mayoría de los casos, se trata de una función cuya imagen es \$\{0, 1\}\$ o \$\{-1, 1\}\$. Por otra parte, dado que estamos definiendo al perceptrón con operaciones aritméticas como sumas y productos, cabe recalcar que las entradas deben ser una *abstracción numérica* del elemento del entorno a clasificar; muchas veces ésta se compone de un vector de valores reales. Ello implica la existencia de un *umbral* \$U\$ (*threshold* en inglés) que divide los posibles valores

de v en dos, permitiendo su clasificación binaria. A continuación, se define la *función escalón*, valuada en $\{-1, 1\}$ (una posible función de activación):

$$\Phi(v) = \begin{cases} 1 & \text{si } v > U \\ -1 & \text{en otro caso} \end{cases} \quad (2.5)$$

Geométricamente, el perceptrón genera un *hiperplano* que, con los pesos adecuados logrará dividir las entradas de manera que cada entrada correspondiente a una cierta clase quede dentro de *una y sólo una* partición del espacio multidimensional en cuestión.

El hecho de que estamos usando funciones de activación valuadas de manera binaria nos invita a explorar el cómputo *neuronal* de las diversas funciones lógicas. Como ejemplo, está la clasificación de la función OR en la Figura 2.4. En este caso, se tienen entradas de dos dimensiones, por lo que es posible visualizarlas gráficamente; en la práctica, se trabaja con un gran número de dimensiones, de lo cual se deduce la importancia de tomarán algunos métodos de reducción dimensional para el éxito de los algoritmos de entrenamiento de modelos más complejos.

Para finalizar esta sección, cabe destacar que la función de activación (*signo*) que acaba de ser presentada no es la única (Figura 2.5). En realidad, se buscan opciones más suavizadas y, sobre todo, diferenciables con el fin de optimizar los parámetros de modelos más complejos.

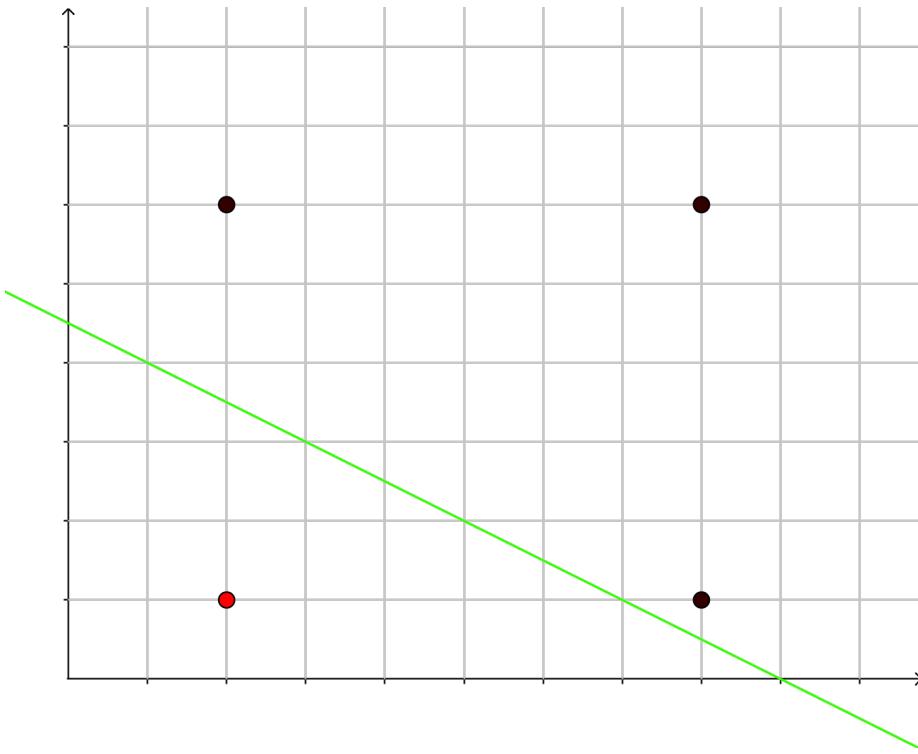


Figura 2.4: La función lógica OR, en su versión bidimensional, tiene cuatro posibles salidas: tres de ellas arrojan un valor positivo, mientras que la otra es negativa. Claramente, se trata de una función separable mediante una recta. (Elaboración propia.)

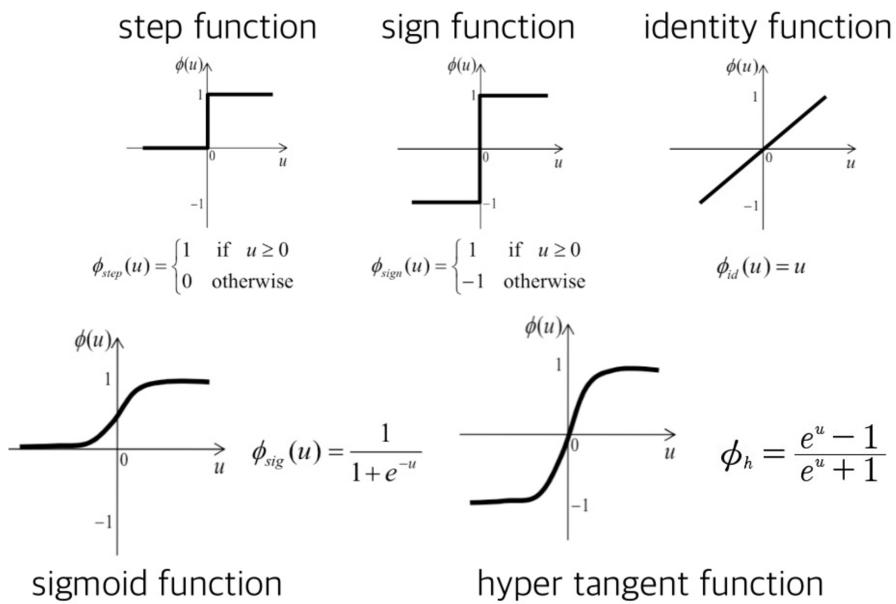


Figura 2.5: Las funciones de activación más comunes.
(Tomado de <https://www.slideshare.net/SungJuKim2/multi-layer-perceptron-back-propagation>)

2.2.4. El perceptrón multicapa

El contenido de este apartado se basa principalmente de [10] y [15].

No todos los patrones de datos son linealmente separables. Por ejemplo, la función lógica XOR, contiene salidas cuyos valores no pueden ser divididos en dos partes del plano sin ser mezclados. Por consiguiente, es necesario aumentar la capacidad de cómputo del perceptrón. Combinaremos, ahora, varios perceptrones en una *red neuronal de propagación hacia adelante*, estructurando el cómputo en distintas *capas ocultas*.

El flujo de la información irá de capa en capa, en una sola dirección hasta llegar a una *capa de salida*. Cada capa oculta consta de un conjunto de neuronas, sin conexiones entre ellas, pero totalmente conectadas a la capa inmediatamente anterior y posterior. A este modelo se le conoce comúnmente como **perceptrón multicapa** (*MLP* por sus siglas en inglés) y es el punto de partida de la rama del aprendizaje automático conocida como **aprendizaje profundo** (*deep learning* por sus siglas en inglés).

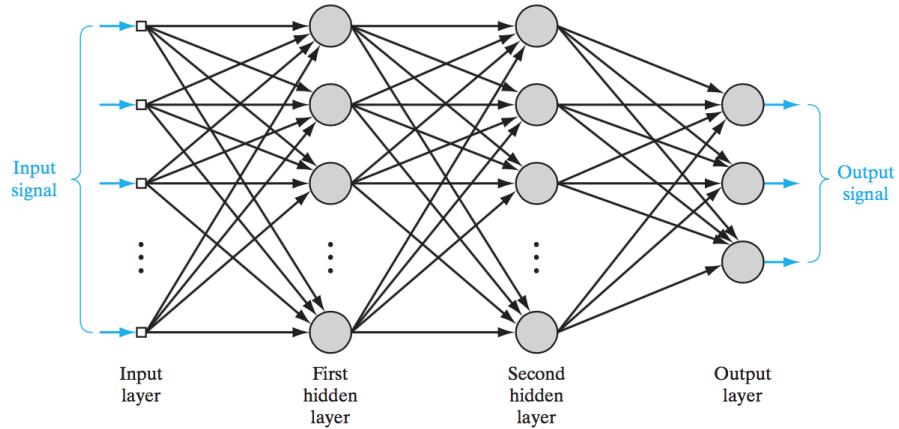


Figura 2.6: Arquitectura de un MLP con dos capas ocultas. (Tomado de [10].)

El flujo del cómputo en un MLP puede ser pensado como la composición de las capas ocultas, vistas como funciones que proce-

san los datos de entrada. Formalmente, dado un vector de entrada \mathbf{x} , la primera capa oculta h_1 calcula su valor utilizando una matriz de pesos \mathbf{W}_1 , un sesgo \mathbf{b}_1 de la siguiente manera:

$$h_1 = \sigma(\mathbf{W}_1^\top \mathbf{x} + \mathbf{b}_1), \quad (2.6)$$

donde σ es una función de activación no lineal y *diferenciable*. Las dimensiones de \mathbf{W} corresponden a la entrada y al número de neuronas en la capa h_1 . La $i+1$ -ésima capa oculta de un MLP actualiza su valor a partir de la i -ésima mediante

$$h_{i+1} = \sigma(\mathbf{W}_{i+1}^\top h_i + \mathbf{b}_{i+1}), \quad (2.7)$$

es decir, cada capa tendrá una matriz de pesos y un vector de sesgos propio. La salida, suponiendo que hay m capas ocultas, obtiene su valor con la ecuación

$$\hat{y} = \sigma(\mathbf{W}_{m+1}^\top h_m + \mathbf{b}_{m+1}). \quad (2.8)$$

Obsérvese que es posible que \hat{y} tenga más de una dimensión, entonces, cada uno de los valores de dicho vector codificará “la existencia” de alguna característica en particular. Esto es muy útil para aprender clasificadores cuyo codominio es mayor al conjunto binario.

En ocasiones, al subgrafo dirigido formado por las capas ocultas h_i y h_{i+1} se le conoce como *capa densa* o *totalmente conectada* y es común encontrarla en arquitecturas de mayor complejidad y especialización. El MLP está dentro del “estado del arte” de los algoritmos de clasificación automática y será parte fundamental de los dos modelos neuronales que compone a la arquitectura de procesamiento de imágenes y generación de lenguaje de la presente tesis. La parte más importante de ello recae en su algoritmo de entrenamiento.

A veces conviene generalizar la función de activación de una capa oculta a una función multivariada, la cual esté en concordancia con las dimensiones de salida. Por ejemplo, la función *softmax* generaliza la función *sigmoide* expuesta en la Figura 2.5: sea $\mathbf{a}_{i+1}^j = (\mathbf{W}_{i+1}^\top h_i)^j + \mathbf{b}_{i+1}^j$ la j -ésima neurona de la combinación

lineal de la $i + 1$ -ésima capa, entonces

$$\text{softmax}(\mathbf{a})_j = \frac{e^{\mathbf{a}_j}}{\sum_{k=1}^K e^{\mathbf{a}_k}}, \quad (2.9)$$

donde asumimos que la salida es un vector de dimensión K .

2.2.5. Retroalimentación para aprender

El contenido de este apartado se basa principalmente de [9].

Como cualquier arquitectura de aprendizaje supervisado, a un MLP debe de asociarse una función objetivo (o de *error*) J , que le permita estructurar un entrenamiento. La no-linealidad de las funciones de activación causan que no se garantice la convexidad por parte de las funciones de error más comunes. Al no existir un método analítico para encontrar mínimos sin importar la elección de J , el entrenamiento de un MLP (y, en general, de una red neuronal profunda) se basa en métodos iterativos que van optimizando los parámetros de la arquitectura, tales como optimizadores basados en gradientes.

Para estar *ad hoc* a la práctica, presentaremos un algoritmo de *descenso por el gradiente estocástico*, el cual, a pesar de no garantizar la convergencia hacia un punto mínimo, funciona bien sabiendo escoger los valores adecuados de inicialización. Dada la salida \hat{y} establecida en la Ecuación 2.8, definimos la función de error total del MLP como

$$J(\boldsymbol{\theta}) = \frac{1}{2n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (2.10)$$

donde $\boldsymbol{\theta}$ es el conjunto de los parámetros del MLP, es decir, $\boldsymbol{\theta} := \bigcup \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^n$; n es el número de capas del MLP; e y_i es la i -ésima etiqueta de entrenamiento. A esta función se le conoce como *error cuadrático medio*; en general, una función de error J se define conceptualmente como la **entropía cruzada** de las distribuciones de los datos ($y \equiv p_{\text{DATOS}}$) y del modelo ($\hat{y} \equiv \hat{p}_{\text{MLP}}$). Esto es, la esperanza

de la verosimilitud logarítmica negativa del modelo¹:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{p_{\text{DATOS}}} \log \hat{p}_{\text{MLP}}(\mathbf{x}) \quad (2.11)$$

$$= -\sum_i (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)). \quad (2.12)$$

Esta función es particularmente útil si se desea interpretar las salidas del MLP como probabilidades, ya que, $\sum_i \hat{y}_i = 1$ y $0 < \hat{y}_i < 1$ para $1 \leq i \leq n$. Cabe destacar que ambas funciones son completamente diferenciables y los procedimientos de optimización, basados en gradientes, son invariantes.

2.2.5.1. Propagación hacia atrás

Cualquier algoritmo de optimización basado en gradientes necesita de un método (*eficiente*) para calcular las derivadas de los parámetros del modelo. En la literatura de aprendizaje profundo el cálculo de \hat{y} muchas veces se le conoce como *propagación hacia adelante*. Esto da lugar al algoritmo de **propagación hacia atrás** (o *retro-propagación*) que permite que el error resultante J se propague de adelante hacia atrás, a través de todos los parámetros del MLP. El objetivo principal de este algoritmo será el cómputo del gradiente del modelo J con respecto a los parámetros $\boldsymbol{\theta}$, es decir, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

Para empezar, el algoritmo de propagación hacia atrás se fundamenta en la regla de la cadena del *cálculo infinitesimal* para el cómputo de derivadas de composiciones de funciones: dados $x \in \mathbb{R}$, $y = g(x)$ y $z = f(g(x)) = f(y)$, la derivada de z con respecto a x se obtiene por

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (2.13)$$

Esta noción se generaliza a funciones que involucran la composición de más de dos funciones: si

- $\mathbf{x} \in \mathbb{R}^m$,

¹ Suponiendo que p_{DATOS} es una distribución *normal*, se puede demostrar la igualdad entre las Ecuaciones 2.11 y 2.10.

- $\mathbf{y} \in \mathbb{R}^n$,
- $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$,
- $f : \mathbb{R}^n \rightarrow \mathbb{R}$,
- $\mathbf{y} = g(\mathbf{x})$ y
- $z = f(\mathbf{y})$, entonces

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (2.14)$$

En forma vectorial, la Ecuación 2.14 se escribe como

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial y}{\partial x} \right)^{\top} \nabla_{\mathbf{y}} z. \quad (2.15)$$

De la Ecuación 2.15 vemos que la expresión $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ es la matriz *jacobiana* (de $n \times m$) de g , es decir, el gradiente de una variable vectorial \mathbf{x} se obtiene multiplicando dicho jacobiano por el gradiente $\nabla_{\mathbf{y}} z$. Para cada operación en el *grafo dirigido*, definido en un MLP, el algoritmo de propagación hacia atrás realizará un producto entre Jacobianos y gradientes.

Las derivaciones que restan al presente apartado, fueron basadas de [2].

En la práctica, el cómputo de estos gradientes es invariante si en vez de vectores tenemos tensores. Intuitivamente, uno puede pensar que, previamente a correr propagación hacia atrás, cualquier tensor de dimensiones $n \times m \times p$ se *aplane* a un vector de dimensión $1 \times nmp$. Tras el cálculo de los gradientes, los vectores planificados vuelven a su representación *tensorial*.

Mediante la regla de la cadena, es posible desarrollar una expresión para el cálculo del gradiente de J con respecto a todos los parámetros $\boldsymbol{\theta}$ del MLP. Sin embargo, una implementación *ingenua* involucraría que el cálculo de muchas subexpresiones se repita un número considerable de veces. En muchas ocasiones, calcular la

misma expresión más de una vez resulta una pérdida de tiempo y/o memoria, la cual muchas veces puede significar un gasto de orden exponencial con respecto al número de parámetros.

En términos formales, consideremos a u como una neurona de la capa $k + 1$ de un MLP arbitrario y que recibe como entradas las salidas de las neuronas v_1, v_2, \dots, v_m de la capa k . De acuerdo a la Ecuación 2.15, el gradiente de la función de error J con respecto a u se obtiene mediante

$$\nabla_u J = \left(\frac{\partial \mathbf{v}}{\partial u} \right)^\top \nabla_{\mathbf{v}} J, \quad (2.16)$$

donde $\mathbf{v} = (v_1, v_2, \dots, v_m)$. Este es un proceso recursivo que inicia con el cálculo del gradiente para la última capa, en donde $\frac{\partial J}{\partial J_i} = 1$ para toda neurona de salida J_i . Nótese que si para obtener J se requirieron n nodos y cada uno de éstos requiere un tiempo constante de cálculo más m arcos, entonces calcular $\nabla_u J$ requiere al menos $\Omega(m)$ cálculos y no es posible obtener los gradientes de una manera más rápida.

Remembrando la terminología establecida a partir de las Ecuaciones 2.6 y 2.7, sea

$$a_k = \mathbf{b}_k + \mathbf{W}_k h_{k-1} \quad (2.17)$$

la ecuación que define la combinación lineal que determina las salidas de la k -ésima capa de un MLP. En lo suiguiente, utilizaremos a la función de error por entropía cruzada (Ecuación 2.11) y asumimos, sin pérdida de generalidad, como funciones de activación $\sigma = \text{softmax}$ para la última capa y $\sigma = \tanh$ para las capas ocultas. Asumiendo que hay L capas en un MLP, es importante enfatizar en las siguientes observaciones:

$$\frac{\partial(-\log \hat{y})}{\partial a_{L,i}} = \hat{y}_i - \mathbb{1}_{y=i}, \quad (2.18)$$

$$\frac{\partial \tanh(u)}{\partial u} = 1 - \tanh(u)^2. \quad (2.19)$$

Resulta lógico entender que la Ecuación 2.18 calcula la diferencia que existe entre las salidas del MLP y las etiquetas del conjunto de

datos. Por otro lado, la Ecuación 2.20 calcula las derivadas parciales de la función tangente hiperbólica aplicada a cualquier tensor real u . Obsérvese que de la expresión $a_{L,i}$ nos referimos a la última capa de un MLP mediante su índice, notación que hará más fácil escribir el siguiente procedimiento.

- Para la función de error, $\frac{\partial J}{\partial J} = 1$.
- El gradiente del error con respecto a cada entrada de la última capa:

$$\frac{\partial J}{\partial a_{L,i}} = \frac{\partial J}{\partial J} \frac{\partial J}{\partial a_{L,i}} = \hat{y}_i - \mathbb{1}_{y=i}. \quad (2.20)$$

- Para cada capa, desde $k = L$ bajando hasta 1:
 - calculamos el gradiente con respecto a los sesgos:

$$\frac{\partial J}{\partial \mathbf{b}_{k,i}} = \frac{\partial J}{\partial a_{k,i}} \frac{\partial a_{k,i}}{\partial \mathbf{b}_{k,i}} = \frac{\partial J}{\partial a_{k,i}}, \quad (2.21)$$

- calculamos el gradiente con respecto a los pesos:

$$\frac{\partial J}{\partial \mathbf{W}_{k,i,j}} = \frac{\partial J}{\partial a_{k,i}} \frac{\partial a_{k,i}}{\partial \mathbf{W}_{k,i,j}} = \frac{\partial J}{\partial a_{k,i}} h_{k-1,j}, \quad (2.22)$$

- propagamos el gradiente hacia una capa inferior; para $k > 1$:

$$\frac{\partial J}{\partial h_{k-1,j}} = \sum_i \frac{\partial J}{\partial a_{k,i}} \frac{\partial a_{k,i}}{\partial h_{k-1,j}} = \sum_i \frac{\partial J}{\partial a_{k,i}} \mathbf{W}_{k,i,j}, \quad (2.23)$$

$$\frac{\partial J}{\partial a_{k-1,j}} = \frac{\partial J}{\partial h_{k-1,j}} \frac{\partial h_{k-1,j}}{\partial a_{k-1,j}} = \frac{\partial J}{\partial h_{k-1,j}} (1 - h_{k-1,j}^2). \quad (2.24)$$

2.2.5.2. Descenso por el gradiente

Una vez obtenido el diferencial por el cual se va a optimizar un MLP con respecto a la función de error, lo que procede es actualizar los parámetros de la red neuronal con una cantidad adecuada. *Descenso por el gradiente* es un algoritmo que minimiza la función de error

$J(\boldsymbol{\theta})$ de una red neuronal (en general) mediante la actualización de los parámetros $\boldsymbol{\theta} \in \mathbb{R}^d$ en dirección opuesta al gradiente $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$.

En su versión más sencilla, este algoritmo intenta encontrar un mínimo (local) mediante un proceso iterativo en el que el gradiente $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ se calcula y se actualizan *todos* los parámetros en proporción con el *meta*-parámetro η , conocido como **tasa de aprendizaje**. El número η determina el *tamaño de los pasos* que se deben tomar para llegar al mínimo local. Así,

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}). \quad (2.25)$$

La Ecuación 2.25 implica que hay que calcular los gradientes por cada parámetro del conjunto de datos para realizar *una sola* actualización. Esto deriva en un procedimiento realmente lento e incluso intratable para conjuntos de datos que no quepan en memoria. La lentitud se justifica por el cómputo redundante de varios gradientes para entradas similares antes de actualizar cada parámetro. Para sobreponerse a este problema, se da lugar al algoritmo de *descenso por el gradiente estocástico* (*SGD*, por sus siglas en inglés) en el cual se realiza una actualización por cada ejemplar $(x^{(i)}, y^{(i)})$ del conjunto de datos:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}; x^{(i)}; y^{(i)}). \quad (2.26)$$

Potencialmente, SGD permite llegar a mejores mínimos locales a los que converge la primera versión descrita. Por otro lado, es más probable que no se pueda llegar al mínimo global, dada la fluctuación que puede haber en las primeras actualizaciones. Sin embargo, se puede demostrar que si la tasa de aprendizaje es suficientemente pequeña, es casi segura la convergencia hacia un mínimo local.

En la práctica, la versión implementada en la mayoría de las bibliotecas de aprendizaje automático incorpora lo mejor de las Ecuaciones 2.25 y 2.26:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}; x^{[i:i+n]}; y^{[i:i+n]}). \quad (2.27)$$

Cada actualización propuesta por la Ecuación 2.27 se da en un rango de los datos, conocido como **mini-lote** o, simplemente, *lote*. Esto

permite que la convergencia se de de manera más estable. Además, el tamaño del lote n usualmente varía entre 50 y 256. El número de veces que el algoritmo ve el conjunto de datos entero se le llama número de **épocas**. Por otro lado, definimos al número de *iteraciones* del algoritmo, contando el número de veces que cada lote pasa por el mismo. El Procedimiento 2.1 describe una implementación del algoritmo de descenso por el gradiente utilizado en muchas bibliotecas actuales de aprendizaje automático.

```

1  for i in range(num_epocas):
2      np.random.shuffle(datos)
3      for lote in get_lotes(datos, tamano_lote=50):
4          params_grad = evaluar_gradiente(funcion_error, lote, params)
5          params = params - tasa_aprendizaje * params_grad

```

Procedimiento 2.1: Implementación del algoritmo de descenso por el gradiente utilizando mini-lotes, en lenguaje Python. Antes de iterar sobre todos los lotes, se mezclan los datos de manera aleatoria.

Visualmente el algoritmo de descenso por el gradiente se puede entender mediante la Figura 2.7. El componente restante, y que es motivo de investigación, es la manera en la que la tasa de aprendizaje se calcula. Para ello, se proponen varios *optimizadores* descritos en [16]; la elección de su uso depende fuertemente del diseño de la arquitectura en cuestión.

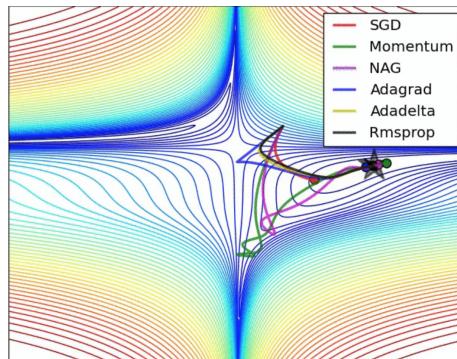


Figura 2.7: La optimización realizada por el algoritmo SGD con mini-lotes. Se muestra el comportamiento de distintos optimizadores. (Tomado de [16].)

2.3. Redes neuronales convolucionales

En 1989, el francés Yann LeCun comenzó a trabajar en los fundamentos de un novedoso modelo neuronal basado en la biología de la corteza visual del cerebro animal. Su trabajo, eventualmente, lo llevó a ser nombrado director de la investigación en IA en Facebook. Su ascenso al “*podio*” del aprendizaje profundo es producto de una revolución en la *vanguardia* del reconocimiento de imágenes a través de computadoras. Y lo más probable es que el mismo modelo continúe sobresaliendo por algunos años.

Como la mayoría de las arquitecturas neuronales, las **redes neuronales convolucionales** (*CNN's*, por sus siglas en inglés) no alcanzaron la popularidad suficiente hasta entrada la época reciente, caracterizada por una alta conectividad (a través de Internet) y un incremento significativo en la capacidad de cómputo de los dispositivos modernos. Sin embargo, LeCun ya había obtenido resultados sobresalientes hacia 1998, en tareas de reconocimiento de patrones [3]. El enfoque de LeCun, y de su equipo en *Bell Labs*, consistió en dar un modelo de mayor capacidad en *representaciones internas* que el tradicional perceptrón multicapa.

Un aspecto importante en esta arquitectura consiste en la delegación de ciertas partes de la misma a sub-problemas específicos de la tarea a resolver. Por consiguiente, cada capa de una CNN definirá uno (o varios) niveles de representación, los cuales capturan ciertas características inherentes al flujo de datos de entrada. Mediante un basto conjunto etiquetado de datos, se pueden aprender dichas abstracciones de manera supervisada. A continuación, explicaremos con detalle las capas más comunes que constituyen a una red de este tipo: empezando por las convoluciones *per se*, la capa de agrupación y la “capa de activación”.

El siguiente material se basa, principalmente, de [12] y [9].



Figura 2.8: Algunos dígitos existentes en la base de datos MNIST. (Tomado de <http://www.researchgate.net/>.)

2.3.1. La capa convolucional

Para iniciar la discusión sobre la estructura de una CNN, usaremos como ejemplo al reconocimiento computacional de imágenes; sin embargo, cabe notar que esta arquitectura es popular en tareas donde existe la necesidad de reconocer patrones sobre conjuntos de datos con topología *cuadriculada*.² Uno de los conjuntos de datos más famosos, y de mayor tradición, es la *Mixed National Institute of Standards and Technology database*, mejor conocida como MNIST. En ella se alberga una gran cantidad de dígitos escritos a mano; el problema, entonces, consiste en clasificar cualquier manuscrito dado en una de las 10 clases existentes, de acuerdo al dígito más parecido.

Como se observa en la Figura 2.8, existen distintas maneras de escribir a mano un solo dígito. Quisiéramos que nuestra solución sea lo suficientemente robusta para distinguir 4's de 9's por más ilegible que sea la letra. No tenemos idea de dónde buscar ciertas características o si el observar un cambio drástico en el color de varios pixeles contiguos nos sea significativo; todo esto es parte de lo que se deberá *aprender*. Esta incertidumbre en la detección de

²De acuerdo a [9], como ejemplos de datos con una topología cuadriculada, se incluyen series de tiempo (cuadrícula de una dimensión), audio o videos.

atributos conduce a, de alguna manera, ir en búsqueda de pequeñas porciones de la imagen que nos den una pista por dónde empezar.

En este punto cabe recordar que usualmente para una computadora, una imagen es un arreglo de tres dimensiones, cada una especificando la posición de un pixel y su color. En la práctica (y de acuerdo a las últimas tendencias de programación de redes neuronales), a esta estructura se le llama **tensor**, por lo que seguiremos la convención. Volviendo a nuestra búsqueda, necesitamos un parámetro que nos indique el mínimo de pixeles a analizar para empezar a encontrar detalles claves de la imagen. Por ello, vamos a usar un pequeño tensor “cuadrado” para recorrer toda la imagen. A dicho tensor le llamaremos **filtro** (*kernel*, en inglés). Llamaremos **zancada** (*stride*, en inglés) al número de pixeles que “saltamos” (en cualquier dimensión) al mover un filtro sobre la imagen. La salida se calcula realizando una combinación lineal sobre cierta región de la imagen ponderada con los valores del filtro y añadiendo un sesgo. Esto produce un tensor con una profundidad equivalente al número de filtros usados. En símbolos, esto corresponde a realizar lo siguiente

$$\sum_i x_i w_i + b \quad (2.28)$$

donde x_i y w_i corresponden a la i -ésima entrada de la imagen y del filtro, respectivamente, y b es el sesgo.

En ocasiones es conveniente rodear una imagen con ceros, ya que debemos garantizar que el filtro encaje exactamente dentro de la imagen conforme se va deslizando horizontal y verticalmente. A esto se le conoce como **relleno de ceros** (*zero-padding* en inglés). Los parámetros que acaban de ser presentados constituyen a una **capa convolucional**, en la cual reside la mayor carga computacional de la arquitectura. Para abreviar, en adelante nos referiremos a dicha capa con la sigla **CONV**.

La correcta elección de parámetros debe de garantizar que el filtro no salga de las dimensiones de la imagen de entrada (sujeta a relleno de ceros). Supongamos que tenemos una imagen cuadrada de dimensión $W \times W$, un filtro de dimensión $F \times F$. Además “enmar-

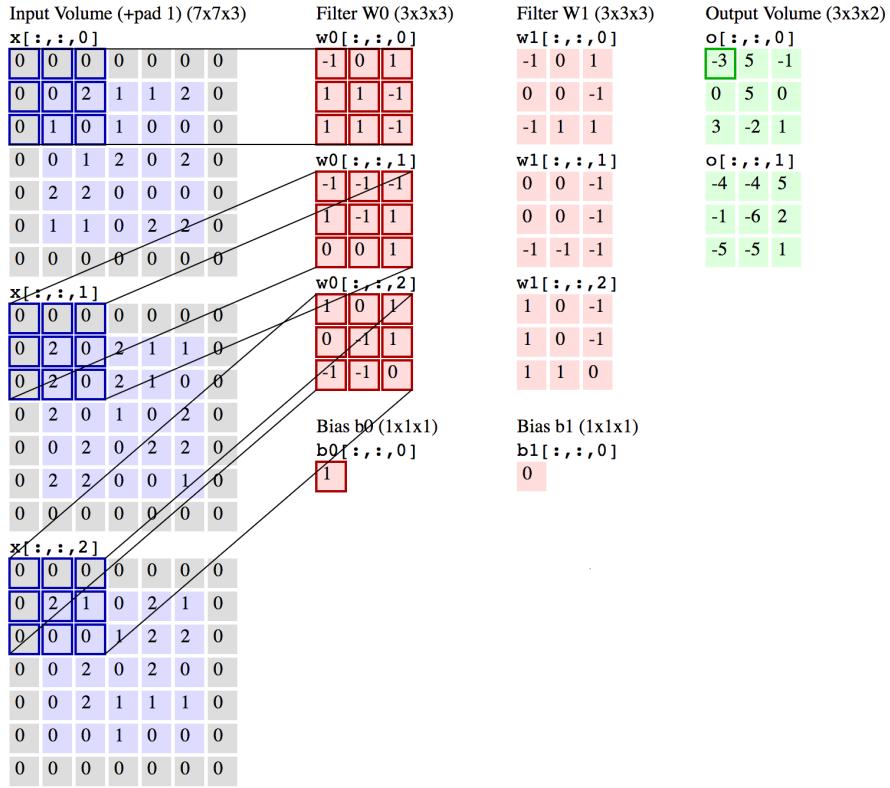


Figura 2.9: Ilustración de una pequeña capa convolucional. Cada imagen de entrada contiene tres niveles de profundidad, con los cuales trabaja el filtro (un tensor de pesos). Cabe destacar que cada valor de x está rellenando por ceros. Dado que la profundidad del volumen de la salida es sólo 2, hay igual número de filtros en total. (Tomado de <http://cs231n.github.io/convolutional-networks/>.)

camos” (rellenamos) de ceros la imagen con un grosor de P píxeles y aplicamos el filtro con una zancada Z . Entonces, la cantidad de neuronas resultantes (en una dimensión) está dada por

$$\frac{W - F + 2P}{Z} + 1, \quad (2.29)$$

donde este número es un entero positivo.

2.3.2. La convolución

Matemáticamente, este proceso se puede formalizar mediante la convolución de dos funciones reales. Si x denota a nuestro dígito desconocido y w a nuestro filtro, entonces la convolución de x con w , en un punto t , se define como:

$$c(t) := \int x(a)w(t-a)da \quad (2.30)$$

y se denota:

$$c(t) = (x * w)(t). \quad (2.31)$$

En el mundo digital, tenemos conjuntos de datos discretos, por lo que es conveniente dar una definición adaptada al respecto:

$$c(t) = (x * w)(t) := \sum_{a=-\infty}^{\infty} x(a)w(t-a). \quad (2.32)$$

Dentro del contexto de reconocimiento de imágenes, es útil especificar que la convolución se hace en dos dimensiones de las que se están contemplando. Para ello introducimos la siguiente notación: si i y j son valores que denotan la posición de un pixel, $X(i, j)$ es el valor (color) correspondiente y $K(i, j)$ denota la entrada (i, j) de nuestro filtro, definimos a la convolución C como:

$$C(t) = (K * I)(i, j) := \sum_m \sum_n I(i - m, j - n)K(m, n). \quad (2.33)$$

Muchas bibliotecas de redes neuronales utilizan la **correlación cruzada** en vez de la convolución. Semánticamente, la correlación cruzada sirve para analizar lo mismo que buscamos con la convolución; por ello, esta sutil distinción muchas veces no es notada. La definición es la siguiente:

$$C(t) = (I * K)(t) := \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (2.34)$$

Nótese que en las dos últimas definiciones se ha convenido *voltear* los índices del filtro. En la teoría, la ventaja que trae esto consiste en una mayor facilidad de probar ciertos resultados.

Para terminar de describir la **capa convolucional** de la arquitectura, vale la pena hablar un poco sobre la salida de (2.34). Aquí estamos caracterizando al valor de cada neurona como se describió en (2.28). Al conjunto de tensores resultantes se les conoce, convenientemente, como **mapas de activación**.

Cabe señalar que la eficiencia de la convolución es mucho mayor a la de una multiplicación de matrices (en una capa densa) y esto ocurre, principalmente, por las siguientes razones:

- En una capa densa, se consideran interacciones entre cada neurona de entrada con cada neurona de salida. Es decir, los productos matriciales involucran a cada pixel, sin importar qué lugares en específico son los que vale la pena resaltar.
- Se dice que en una capa convolucional se manejan **pesos dispersos**, los cuales corresponden a las entradas del filtro. Esto significa que después de un entrenamiento, el filtro aprenderá a interactuar con ciertos grupos de pixeles, sin conocer su distribución. Con esto es posible tomar en cuenta pequeños detalles que ocurren en una vecindad reducida de pixeles.
- En una capa convolucional, los pesos del filtro que se buscan aprender son mucho menos que los pesos que se aprendería en una capa densa: si la imagen de entrada contiene miles o millones de pixeles, entonces necesitamos cientos de pixeles en nuestro filtro para aprender los rasgos más pequeños de la misma (como bordes o puntos).

En resumen, si tenemos m entradas y n salidas en una capa, y ésta es densa, entonces habremos de realizar $O(m \times n)$ operaciones para calcular la salida. En cambio, en una capa convolucional, podemos tener una zancada Z de pixeles en el filtro, con un orden de magnitud mucho menor a n ; lo cual va a provocar que sea más eficiente calcular $O(m \times Z)$ operaciones.

2.3.3. Filtros vistos como arreglos de neuronas

Dependiendo del número de filtros escogidos para aplicarse en la capa CONV, se definirá la salida de la misma. Intuitivamente, si se usan 12 filtros en una imagen cuya dimensión de entrada es de $32 \times 32 \times 3$, uno puede imaginarse 12 imágenes procesadas por la capa CONV, como resultado del procesamiento expuesto en la Figura 2.3.3. Es decir, la dimensión de la salida sería de $32 \times 32 \times 12$.

Esto no corresponde al “mecanismo canónico” que caracteriza a una capa neuronal (densa), en donde todas las entradas participan en la decisión de la salida. Más aún, a diferencia del perceptrón multicapa, en donde existe un conjunto de pesos para cada neurona, aquí un conjunto de neuronas comparte los mismos pesos. Sin embargo, podemos ahorrarnos la discusión sobre cómo adaptar al algoritmo de propagación hacia atrás para hacer que cada neurona aprenda, transformando un poco la arquitectura:

- Si nuestra dimensión de entrada es de $32 \times 32 \times 3$, “aplanamos” dicho tensor de manera vertical, es decir, en uno de dimensión $1024 \times 1 \times 3$.
- Por cada filtro, apilamos todas las neuronas en una salida cuya altura está determinada por el valor de la zancada.
- Cada neurona de salida estará conectada a tantas neuronas de entrada como indique, otra vez, la zancada. Éstas serán continuas tras ser transformadas como indica el primer punto mencionado. Adicionalmente, podemos pensar que todas las demás neuronas de entrada están conectadas a cada salida con un peso equivalente a cero unidades.
- En este caso, los valores que constituyen al tensor de filtro estarán codificados como pesos en cada sinapsis.
- La $i + 1$ -ésima neurona de salida tendrá como primera entrada (en orden de arriba hacia abajo), a la segunda entrada de la i -ésima neurona.

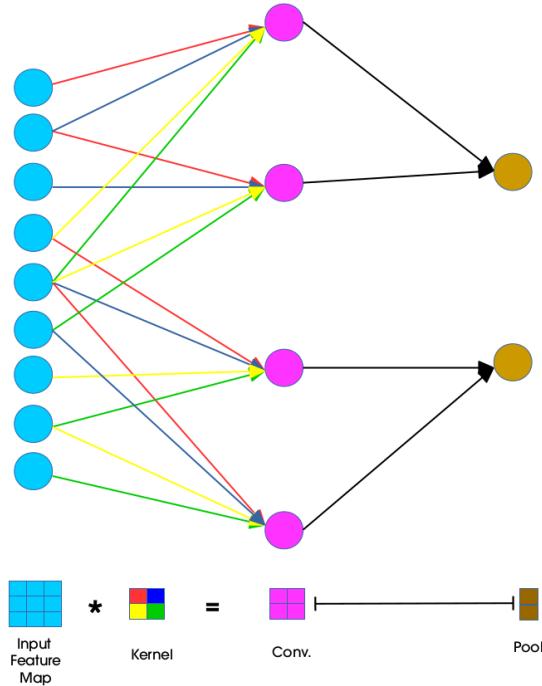


Figura 2.10: El concepto de pesos compartidos. Las sinapsis del mismo color representan conexiones con pesos iguales. En este caso, se tendría una zancada de 4 neuronas. (Tomado de <http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>.)

La capa de salida de este reacomodo corresponde a un mapa de activación “aplanado”. Si queremos un *volumen* de salida con 12 mapas, entonces habrá que visualizar un modelo en el cual tenemos 12 mapas aplanados conectados, cada uno, al tensor de entrada y sin sinapsis alguna entre cada uno de éstos. Cabe destacar que durante el entrenamiento, basta actualizar únicamente un solo conjunto de pesos que conectan a una neurona de salida con una cantidad de entradas equivalente al valor de la zancada y, posteriormente, propagar los nuevos valores a los demás pesos por cada neurona de salida; esto por cada uno de los filtros. Esto ocurre debido a que cada uno de los filtros “comparte parámetros”, como se observa en la figura .

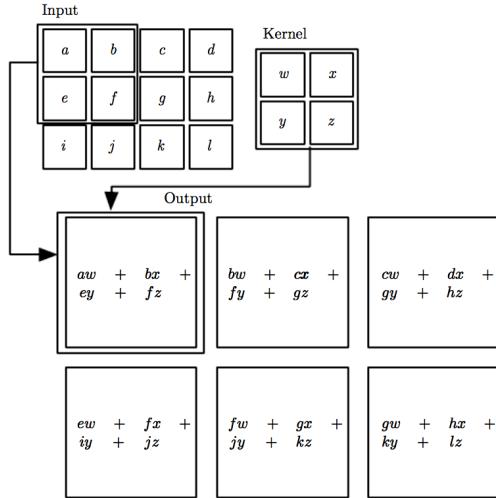


Figura 2.11: Ilustración del proceso de convolución. El filtro se desliza por la imagen de entrada y se efectúa un producto pixel por pixel.) (Tomado de [9].)

2.3.4. La “capa” de activación

Como sucede al calcular las salidas de una capa densa, aquí hay que utilizar una función no lineal (y *suave*) para finalizar la clasificación. En algunos casos, a este paso se le suele llamar **capa de no linealidad**. En [13], además de usar dicho término, se propone a la función *tanh*, sobre cada entrada de los mapas de activación, como el estándar en CNN’s. Sin embargo, también se señala que una *sigmoide rectificada* ha dado buenos resultados en reconocimiento de imágenes (sujeta a una normalización posterior); el caso particular es el de la **unidad lineal rectificada** (*ReLU*, por sus siglas en inglés):

$$f(x) = \max(0, x). \quad (2.35)$$

2.3.5. La capa de *pooling*

Un exceso de aprendizaje en la capa de convolución puede provocar un *sobreajuste* sobre el modelo. Esto se puede observar, en nuestro

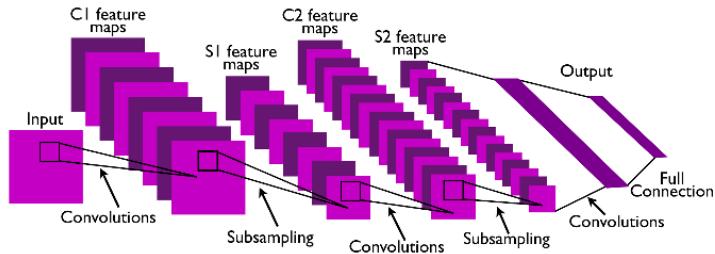


Figura 2.12: Estructura básica de una red neuronal convolucional, con dos fases de extracción de características. (Tomado de [13])

ejemplo, con una arquitectura donde cada filtro se entrene a funcionar para un grupo pequeño y exclusivo de píxeles, los cuales esperan encontrar detalles claves en lugares precisos. Es necesario, entonces, ocuparse de que una CNN *generalice* suficientemente su aprendizaje para aumentar su robustez.

Por ello, las CNN's llevan, además de capas convolucionales, algunas capas de **agrupación** (*pooling*, en inglés y usaremos esta palabra en adelante) con las cuales se puede extraer los detalles más fundamentales de cada mapa de activación. En la práctica, existen dos operaciones de *pooling* que sobresalen sobre las demás: *max-pooling* y *avg-pooling*. Para calcular cualquier capa de *pooling*, se define una vecindad en cada mapa de activación (que no exceda su zancada); acto seguido, dependiendo del *pooling* a usar, se almacena el máximo o el promedio de los píxeles abarcados en la vecindad.

Con esta capa, la arquitectura convolucional se vuelve *aproximadamente* invariante a modificaciones en la entrada. Esto permite enfocarnos más en la existencia de ciertas características en la imagen que en la posición exacta de las mismas.

El *pooling*, además, nos brinda un mecanismo de disminución de parámetros del modelo (neuronas) brindando la posibilidad de optimizar mucho más la memoria utilizada y las dimensiones de las capas.

2.4. Redes neuronales recurrentes

El contenido de este apartado se basa principalmente de [9].

Es frecuente encontrar situaciones en las que exista un orden secuencial entre los datos de entrenamiento. Esto es algo que tienen en común tareas como reconocimiento del habla o traducción automática (de idiomas). Aquí cada instancia de un conjunto de datos se puede ver como una sucesión de valores

$$\mathbf{X} = \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$$

y el objetivo es entrenar una arquitectura que nos permita predecir $\mathbf{x}^{(i+1)}$, dados $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(i)}$.

Un modelo (*estadístico*) de lenguaje es una función de probabilidad sobre cualesquiera sucesiones de símbolos de un alfabeto dado. En procesamiento de lenguaje natural (**PLN**), modelar un lenguaje consiste en la construcción de estimadores que tengan la capacidad de estructurar oraciones dado un conocimiento *a priori* ($\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(i)}$), es decir, maximizar

$$P(\mathbf{x}^{(i+1)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(i)}). \quad (2.36)$$

El lenguaje natural es inherentemente ambiguo; hecho que ha popularizado el enfoque probabilístico en PLN, dejando atrás arquitecturas puramente simbólicas cuya capacidad de cómputo es limitado. Por ejemplo, el enunciado “*Eutropio acomodó el portafolio que Porfirio tiró en el buró*” se puede interpretar de dos maneras:

- Eutropio encontró el portafolio en el buró y procedió a acomodarlo en algún otro lugar.
- Porfirio dejó tirado el portafolio; acto seguido, Eutropio lo colocó en el buró.

En un modelo de lenguaje, ¿cómo decidimos qué interpretación semántica tomar? En la práctica, esto depende del contexto

(conjunto de datos) y la tarea específicamente a resolver. En términos formales, existe más de un *árbol de análisis sintáctico* que caracteriza al enunciado anterior y el modelo debe de encontrar qué árbol tomar como interpretación.

La teoría de lenguajes formales establece que la elaboración de árboles sintácticos, de gramáticas no ambiguas, es un proceso que requiere un modelo computacional que necesariamente utilice al menos una *memoria temporal*. Pero, en general, la ambigüedad semántica nos obliga a no poder acotar el tamaño de dicha memoria, por lo que necesitamos una arquitectura *turing-completa* al menos para fines de construcción de enunciados sintáctica y semánticamente correctos.

Un **red neuronal recurrente** (*RNN*, por sus siglas en inglés) es una arquitectura que posee una memoria interna. La estructura básica de la misma se muestra en la Figura 2.13. Podemos observar que la estrategia gráfica para incorporar a dicha memoria consiste en establecer un *ciclo* entre la salida y la entrada, es decir, existirá una matriz de pesos que se entrenará y guardará un resumen de los aspectos importantes de cada símbolo de la sucesión \mathbf{X} que va procesando.

Antes de proceder con el funcionamiento de una RNN, es importante hablar sobre por qué una CNN no es un modelo adecuado para el procesamiento de datos secuenciales. La filosofía del procesamiento de imágenes en aprendizaje profundo consiste en aprender a reconocer patrones que caractericen a la instancia dada, muchas veces sin importar las relaciones que existan entre ellos mismos. Sin embargo, es inconveniente dejar de lado estas relaciones, pues la estructura gramatical del lenguaje natural implica un orden que es difícil de capturar por una CNN: habría que compartir parámetros (sinápsis) entre capas convolucionales. Más aún, dejando de lado el proceso de entrenamiento, una CNN es una máquina finita de estados que carece de una memoria temporal, pues la información solo fluye ascendentemente entre capas ignorando un posible retroalimentación de lo que se aprendió con \mathbf{x}^i al momento de procesar \mathbf{x}^{i+1} .

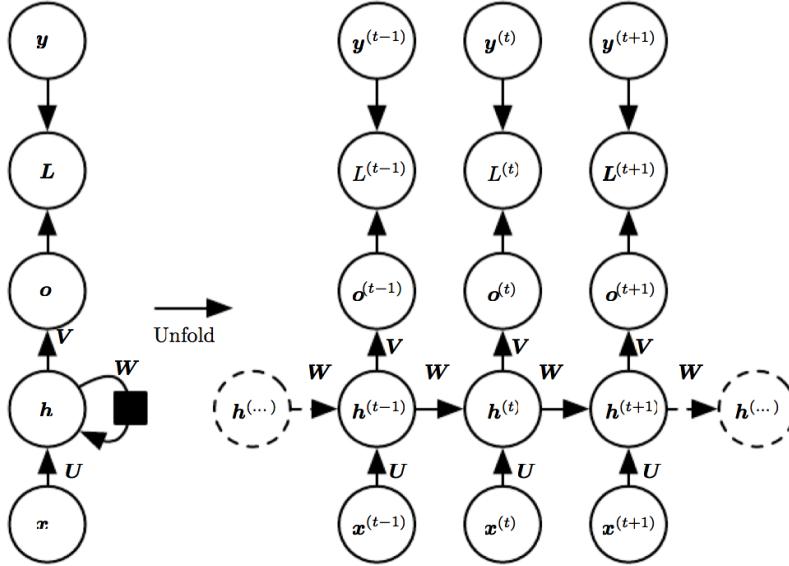


Figura 2.13: Arquitectura gráfica de una red neuronal recurrente. (Tomado de [9])

2.4.1. ¿Cómo funciona la arquitectura recurrente?

Como se mencionó anteriormente, la memoria interna de una RNN se codifica con una matriz de pesos. Se dice que estos parámetros se *comparten* a través del tiempo, conforme la red va procesando la secuencia de entradas. Formalmente, escribimos este fenómeno bajo la intuición de un sistema dinámico, en el que en un tiempo t , el estado actual del mismo depende del anterior, de la entrada actual y de los parámetros del sistema:

$$\mathbf{h}^{(t+1)} = f(\mathbf{h}^{(t)}, \mathbf{x}^{(t+1)}; \theta). \quad (2.37)$$

Se puede pensar que la función f determina qué aspectos del pasado son lo suficientemente importantes para predecir el futuro. Cabe observar que f prevalece en cualquier fase temporal, por lo que (2.37) se puede reescribir como una función que, dado un tamaño fijo τ de las sucesiones de entrada, aplica f τ veces para producir un enunciado. La *recursión* presente en (2.37) permite “desdoblar” la arquitectura de una RNN, como se muestra en 2.13. Simbólicamente, si hacemos

$\tau = 3$, podemos escribir

$$\mathbf{h}^{(3)} = f(\mathbf{h}^{(2)}, \mathbf{x}^{(3)}; \theta) \quad (2.38)$$

$$= f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}; \theta), \mathbf{x}^{(3)}; \theta). \quad (2.39)$$

Existen algunas variantes de redes recurrentes que se pueden utilizar bajo lo establecido anteriormente. La más común, y con la cual procederemos a desarrollar, se muestra en la Figura 2.13. Ahora, utilizaremos una tangente hiperbólica como función de activación σ y observamos a cada salida \mathbf{o} como la *probabilidad-log* de predicción de una clase (un símbolo en el tiempo t). Para normalizar dichas predicciones, aplicamos la función *softmax* para obtener $\hat{\mathbf{Y}}$. Dado un estado (capa oculta) inicial $\mathbf{h}^{(0)}$, para cualquier $t \in \{1, \dots, \tau\}$, las ecuaciones de actualización en la propagación hacia adelante están dadas por:

$$\mathbf{a}^{(t+1)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t)} + \mathbf{U}\mathbf{x}^{(t+1)} \quad (2.40)$$

$$\mathbf{h}^{(t+1)} = \tanh(\mathbf{a}^{(t+1)}) \quad (2.41)$$

$$\mathbf{o}^{(t+1)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t+1)} \quad (2.42)$$

$$\hat{\mathbf{y}}^{(t+1)} = \text{softmax}(\mathbf{o}^{(t+1)}) \quad (2.43)$$

donde \mathbf{b} y \mathbf{c} son los vectores de sesgo, mientras que \mathbf{U} , \mathbf{V} y \mathbf{W} son las matrices de pesos para las conexiones de las capas entrada-oculta, oculta-salida y oculta-oculta respectivamente.

Este modelo de RNN relaciona una sucesión de entrada \mathbf{X} con una salida \mathbf{Y} de misma longitud. El error total L de \mathbf{X} con \mathbf{Y} se define como la suma de los errores individuales en todos los tiempos:

$$L(\{\mathbf{x}^{(i)}\}_{i=1}^{\tau}, \{\mathbf{y}^{(i)}\}_{i=1}^{\tau}) \quad (2.44)$$

$$= \sum_t L^{(t)} \quad (2.45)$$

$$= - \sum_t \log p_{\text{modelo}}(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}) \quad (2.46)$$

donde $L^{(t)}$ es la verosimilitud (en escala logarítmica) de $\hat{\mathbf{Y}}$ dado $\{\mathbf{x}^{(i)}\}_{i=1}^t$. El término $p_{\text{modelo}}(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\})$ se obtiene a partir de la entrada de $y^{(t)}$ del tensor $\hat{\mathbf{y}}^{(t)}$.

Visualmente, este modelo neuronal ya especifica cómo es que se calculará el gradiente, para fines de entrenamiento. El costo del mismo depende totalmente de la “profundidad” dada por el parámetro τ y se discutirá a continuación.

2.4.2. Propagación hacia atrás, a través del tiempo

Aludiendo a una de sus interpretaciones empíricas, el entero positivo τ nos permite trabajar un modelo de cómputo que procesa una especie de series de tiempo. Independientemente de la elegancia que traen consigo los ciclos en una red neuronal, las versiones desdobladas resultan ser de mayor utilidad en la fase de entrenamiento. La idea clave que hay que observar es que en realidad tenemos τ unidades ocultas que pueden ser entrenadas con una versión del algoritmo de propagación hacia atrás.

Propagación hacia atrás, a través del tiempo es un procedimiento óptimo para calcular el gradiente de una RNN. Su desempeño garantiza el uso de $O(\tau)$ tanto en memoria como en ejecución, es decir, la cota mínima de recursos, dada la dependencia entre capas de neuronas y la estructura de los datos de entrenamiento. Esto implica que, incluso ejecutándose en paralelo, es imposible reducir, por lo menos, la cota inferior de memoria utilizada.

Una vez efectuada una propagación hacia adelante, procedemos a calcular el gradiente del error total L con respecto a los parámetros \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} , y \mathbf{c} , así como cada neurona $\mathbf{x}^{(t)}$, $\mathbf{h}^{(t)}$, $\mathbf{o}^{(t)}$ y $L^{(t)}$. El cálculo del gradiente de una neurona depende directamente de las neuronas a las que propaga información: por ejemplo, el valor de $\nabla_{\mathbf{x}^{(t)}} L$ depende de la suma de $\nabla_{\mathbf{x}^{(t+1)}} L$ con $\nabla_{\mathbf{o}^{(t)}} L$ pues tanto $\mathbf{x}^{(t+1)}$ como $\mathbf{o}^{(t)}$ son neuronas que dependen de $\mathbf{x}^{(t)}$. De esta manera, iniciamos la recursión calculando el gradiente con respecto al error final:

$$\frac{\partial L}{\partial L^{(t)}} = 1. \quad (2.47)$$

De acuerdo a la Ecuación 2.43, la i -ésima entrada del gradiente

$\nabla_{\mathbf{o}^{(t)}} L$ es

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbb{1}_{i,y^{(t)}}. \quad (2.48)$$

donde $y^{(t)}$ es el valor objetivo de entrenamiento en el tiempo t .

Si $t = \tau$, entonces la neurona \mathbf{h}^τ tiene como único descendiente a \mathbf{o}^τ . Su gradiente queda como

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathbf{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L, \quad (2.49)$$

en caso contrario ($1 \leq t < \tau$), este gradiente está dado por

$$\nabla_{\mathbf{h}^{(t)}} L = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\mathbf{h}^{(t+1)} L) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\mathbf{o}^{(t)} L) \quad (2.50)$$

$$= \mathbf{W}^\top (\nabla_{\mathbf{h}^{(t+1)}} L) \text{diag}(1 - (\mathbf{h}^{(t+1)})^2) + \mathbf{V}^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (2.51)$$

donde $\text{diag}(1 - (\mathbf{h}^{(t+1)})^2)$ indica el Jacobiano de la tangente hiperbólica asociada con la neurona oculta i en el tiempo $t + 1$.

Antes de enunciar las ecuaciones para calcular los gradientes con respecto a los parámetros restantes, cabe recalcar que debido a su uso (compartido) en cada uno de los tiempos t , la semántica del operador $\nabla_{\mathbf{W}} L$ no es la adecuada pues implica tomar las contribuciones a partir de todas las sinápsis de la red neuronal. En cambio, utilizaremos la notación $\mathbf{W}^{(t)}$ para referirnos a la “copia” de \mathbf{W} en

el tiempo t .

$$\nabla_{\mathbf{c}} L = \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}} \right)^\top \nabla_{\mathbf{o}^{(t)}} L = \sum_t \nabla_{\mathbf{o}^{(t)}} L \quad (2.52)$$

$$\nabla_{\mathbf{b}} L = \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}^{(t)}} \right)^\top \nabla_{\mathbf{h}^{(t)}} L \quad (2.53)$$

$$= \sum_t \text{diag}(1 - (\mathbf{h}^{(t)})^2) \nabla_{\mathbf{h}^{(t)}} L \quad (2.54)$$

$$\nabla_{\mathbf{v}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\mathbf{v}} o_i^{(t)} \quad (2.55)$$

$$= \sum_t (\nabla_{\mathbf{o}^{(t)}} L) \mathbf{h}^{(t)\top} \quad (2.56)$$

$$\nabla_{\mathbf{w}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{w}^{(t)}} h_i^{(t)} \quad (2.57)$$

$$= \sum_t \text{diag}(1 - (\mathbf{h}^{(t)})^2) (\nabla_{\mathbf{h}^{(t)}} L) h^{(t-1)\top} \quad (2.58)$$

$$\nabla_{\mathbf{u}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{u}^{(t)}} h_i^{(t)} \quad (2.59)$$

$$= \sum_t \text{diag}(1 - (\mathbf{h}^{(t)})^2) (\nabla_{\mathbf{h}^{(t)}} L) x^{(t-1)\top}. \quad (2.60)$$

Por último, ya teniendo los gradientes lo que procede para actualizar los parámetros de la RNN sería aplicar el algoritmo de descenso por el gradiente estocástico.

2.4.3. Redes de gran memoria de corto plazo (LSTM)

Durante el proceso de generación de lenguaje natural, es extremadamente incierto determinar qué hechos del pasado son relevantes para seguir tomando en cuenta y qué cosas valen la pena ser olvidadas. El modelo recurrente que hasta ahora ha sido presentado carece de un mecanismo que tenga la capacidad de recordar y olvidar información durante intervalos de tiempo.

Por ello, equiparemos al modelo con *compuertas recurrentes cerradas*, las cuales reemplazarán a las neuronas ocultas \mathbf{h} por un esquema que posee ciclos internos y que es capaz de acumular y olvidar información. La estructura de cada unidad es análoga a la de una compuerta lógica (en hardware) donde el flujo de datos es modificado por operaciones acumulativas, cuyo funcionamiento depende del ajuste de algunos parámetros.

Uno de los modelos con mayor popularidad es el de la **gran memoria de corto plazo** (*LSTM* por sus siglas en inglés). Mientras que la inclusión de ciclos en la arquitectura de una RNN es suficiente para codificar una memoria *a corto plazo*, una compuerta LSTM extiende el tiempo en el que la información va desapareciendo, permitiendo que el modelo guarde una memoria *a largo plazo*. El éxito de la compuerta LSTM ha permeado tareas como reconocimiento y generación de caligrafía, reconocimiento de voz, traducción automática, análisis sintáctico y etiquetamiento de imágenes (que motiva a esta tesis).

La arquitectura RNN-LSTM se caracteriza por ser superficial en cuanto al número de neuronas que separan una entrada de una salida; sin embargo, ello no implica un pobre desempeño. Una **célula LSTM** contiene

- un *bucle* interno;
- las mismas entradas y salidas de una RNN ordinaria;
- una *unidad de estado* $s_i^{(t)}$ que se conecta a sí misma a través del bucle, de manera lineal;
- una *compuerta de olvido* $f_i^{(t)}$ que controla la matriz de pesos del bucle y mantiene un valor entre 0 y 1 por medio de una activación sigmoide;
- una *compuerta de entrada* $g_i^{(t)}$ que mantiene sus valores entre 0 y 1 mediante una sigmoide;
- una *compuerta de salida* $q_i^{(t)}$ que mantiene sus valores entre 0 y 1 mediante una sigmoide.

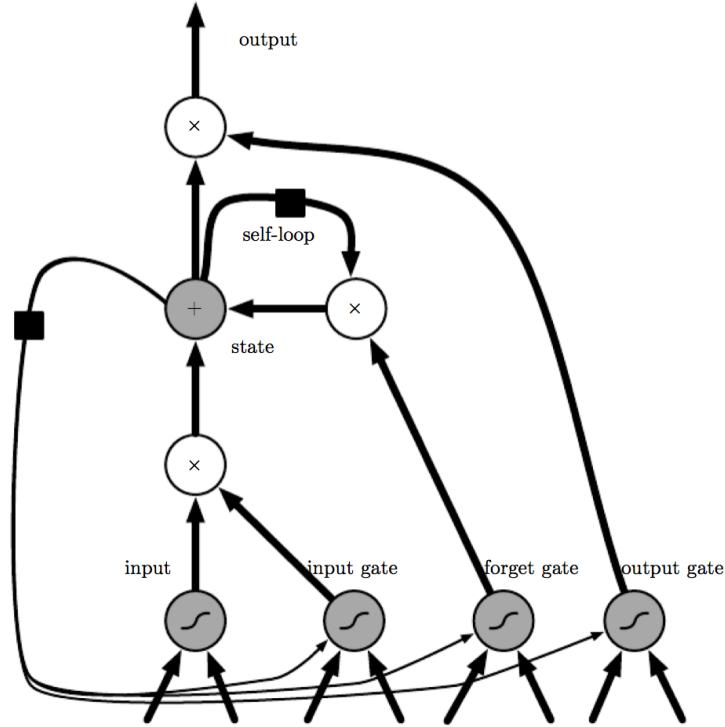


Figura 2.14: Estructura de la compuerta cerrada de una LSTM. (Tomado de [9])

Aquí hablamos de la i -ésima célula en el tiempo t . En símbolos,

$$f_i^{(t+1)} = \sigma \left(b_i^f + \sum_j U_{i,j}^f x_j^{(t+1)} + \sum_j W_{i,j}^f h_j^{(t)} \right) \quad (2.61)$$

$$s_i^{(t+1)} = f_i^{(t+1)} s_i^{(t)} + g_i^{(t+1)} + \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t+1)} + \sum_j W_{i,j} h_j^{(t)} \right) \quad (2.62)$$

$$g_i^{(t+1)} = \sigma \left(b_i^g + \sum_j U_{i,j}^g x_j^{(t+1)} + \sum_j W_{i,j}^g h_j^{(t)} \right) \quad (2.63)$$

$$h_i^{(t+1)} = \tanh(s_i^{(t+1)}) q_i^{(t+1)} \quad (2.64)$$

$$q_i^{(t+1)} = \sigma \left(b_i^O + \sum_j U_{i,j}^O x_j^{(t+1)} + \sum_j W_{i,j}^O h_j^{(t)} \right) \quad (2.65)$$

donde $\mathbf{x}^{(t+1)}$ es el vector de entrada actual y $\mathbf{h}^{(t+1)}$ es el vector de la capa oculta actual. \mathbf{b} , \mathbf{U} y \mathbf{W} son los sesgos, pesos de entrada y recurrentes para la célula LSTM, respectivamente; los análogos que contienen algún superíndice entre f , g y O corresponden a los pesos de las compuertas de olvido, entrada y salida, respectivamente.

Con estas ecuaciones, es posible realizar propagación hacia adelante sin mayor problema. Para entrenar una red LSTM, se puede usar una variante del algoritmo de propagación hacia atrás (a través del tiempo) como se proporciona en [11].

3

Red neuronal para descripciones de memes

DEJANDO de lado la teoría memética introducida por Dawkins [6], consideraremos a un *meme* de forma abstracta, separando su representación visual de su leyenda. Por consecuencia, la arquitectura neuronal que se presentará en este capítulo, tratará de procesar los siguientes tipos de datos:

- Un *tensor* con dimensiones $[w, h, 3]$ ¹, donde w y h son el ancho y el largo de una imagen, mientras que el 3 simboliza los canales de la misma (RGB).
- Un vector que contiene los índices correspondientes a las palabras de una leyenda; éstos se obtienen *mapeando* números naturales con un diccionario que reúne todas las palabras del conjunto de datos de entrenamiento.

¹ Aquí, usamos la notación del paquete NumPy del lenguaje de programación *Python* para referirnos a las dimensiones de un tensor.

3.1. Formalización del problema de aprendizaje

Los recientes avances en generación de descripciones de imágenes destacan el éxito de una CNN que obtiene una representación vectorial de las características más importantes de las imágenes de entrenamiento[23]. Vinyals, *et al* además proponen canalizar estos vectores hacia una arquitectura recurrente que, junto con las descripciones de las mismas crean un modelo de lenguaje ².

La comunidad de procesamiento de lenguaje natural se ha inspirado en el modelo de *traducción automática neuronal* que incorpora una LSTM, cuya memoria inicial está dada por los vectores salientes de la CNN, que codifica un enunciado de entrada de longitud indefinida en una representación de longitud fija para, posteriormente, “decodificar” ésta última en un enunciado objetivo. Intuitivamente, se puede pensar en una máquina abstracta que, dadas las características de una imagen intenta “traducir” las palabras S_1, \dots, S_t (un prefijo de la imagen) para obtener el sufijo S_{t+1}, \dots, S_N .

Formalmente, estamos construyendo dos distribuciones de probabilidad dadas por la CNN y la LSTM, respectivamente. El problema, en cuestión, de aprendizaje automático consiste en maximizar la *verosimilitud* de la descripción correcta de una imagen I por un enunciado S :

$$\theta^* = \arg \max_{\theta} \sum_{I,S} \log p(S|I; \theta) \quad (3.1)$$

donde θ son los parámetros del modelo. En este caso, S representa a cualquier enunciado sin restricción de longitud; para fijar su longitud, hacemos uso de la regla de la cadena probabilística para calcular la log-probabilidad de $S = S_0, \dots, S_N$:

$$\log p(S|I) = \sum_{t=0}^N \log p(S_{t+1}|I, S_0, \dots, S_t). \quad (3.2)$$

² Llamamos **modelo de lenguaje probabilístico** a una distribución de probabilidad sobre todos los enunciados formados por las palabras de un diccionario. Un buen modelo de lenguaje debe de maximizar la *probabilidad conjunta* enunciados que sean coherentes sintáctica y semánticamente.

Obsérvese que la dependencia de los parámetros θ continúa aunque no se haga explícito.

En otras palabras, proponemos usar una LSTM para estimar $p(S_{t+1}|I, S_0, \dots, S_t)$, de acuerdo al *estado del arte* actual de tareas de predicción de secuencias de símbolos. La tupla (I, S) será uno de los tantos ejemplares de entrenamiento, donde la imagen I son las características encontradas por una CNN ajustadas a un espacio vectorial. Este esquema neuronal garantiza un aprendizaje *de punto a punto*, es decir, la red tendrá la capacidad de considerar cada pixel de una imagen para determinar qué tanto contribuye en la generación de enunciados.

3.2. Inception V3

¿Cuántas capas son necesarias y suficientes?

Nos referimos a una CNN como **incrustadora** o **encajadora** (*embedder* en inglés) pues su trabajo consiste en extraer las características de una imagen y comprimirlas en un espacio vectorial fijo. Sabemos, a partir del capítulo anterior, de la capacidad de filtrar detalles de una capa convolucional; sin embargo no existe pista alguna que nos muestre con exactitud qué hay que tomar en cuenta de un conjunto de imágenes de entrenamiento (¡y sobre todo de memes!).

Este problema se reduce al de la construcción de una arquitectura suficientemente *general* que sea capaz de filtrar hasta el más mínimo detalle de una imagen. Con el fin de atacarlo, surgió la competencia anual “*Large Scale Visual Recognition Challenge*” (ILSVRC), cuyo objetivo ha sido el de mejorar en la tarea general de etiquetamiento de imágenes. Aunado a ello, el laboratorio de visión computacional de *Stanford University* comparte el conjunto de datos *ImageNet* que reúne 14,197,122 imágenes etiquetadas para el entrenamiento y la evaluación de los algoritmos participantes.

En 2014, los modelos neuronales de gran profundidad co-

menzaron a ganar popularidad en esta tarea, hecho que se consolidó con la arquitectura **Inception** [21]. A diferencia de algunas de sus predecesoras (AlexNet, VGGNet), *Inception* reduce hasta 12 veces el número de parámetros (pesos) requeridos, garantizando un desempeño computacional más óptimo. Es decir, su escalabilidad le permite incorporarse a la industria de grandes bases de datos. Por ello, se ha decidido usar a Inception como “incrustadora” de imágenes.

Aquí, vale la pena destacar que la profundidad que posee *Inception V3* es del orden de 256 capas y que algunas de sus predecesoras se caracterizaban por una profundidad de al menos un orden de magnitud menor a ella. Sin embargo, la reducción en la amplitud promedio (número de dimensiones por capa) le permite remover algunos cuellos de botella de desempeño. A continuación, se presentan algunos de los principios de diseño de Inception.

- P.1** Dos capas contiguas, con un número de neuronas conectadas menor al promedio entre cualesquiera otras, implican que la información está siendo comprimida mientras fluye hacia la capa de salida. En general, se busca capturar la mayor cantidad de detalles, a través de un número considerable de dimensiones en las primeras capas; por lo que, más o menos, el número de dimensiones debe reducirse de a poco desde el inicio hasta el final.
- P.2** Se prefiere incrementar el número de “capas de activación” por cada grupo de convoluciones-pooling, pues esto revela una cantidad mayor de características y facilita el entrenamiento.
- P.3** Es posible realizar una operación de *adición espacial*, en la cual se combinen incrustaciones con dimensiones bajas. Más aún, se pueden realizar reducciones dimensionales tras alguna adición espacial y previamente a una convolución, sin pérdida de información.
- P.4** El balance entre la amplitud y la profundidad de la red neuronal contribuye directamente a la calidad del desempeño de la misma. Sin embargo, ambas medidas deben ser incrementadas

en paralelo, para garantizar un aumento *constante* en el costo computacional de la red.

3.2.1. Factorización de convoluciones

Reducción de dimensiones, en una red convolucional, implica reducción *significativa* de pesos por capa convolucional. Una de las técnicas usadas, primero por GoogLeNet³, y luego adoptada por *Inception V3* consiste en **factorizar** (*descomponer*) una capa CONV con un filtro “grande” en varias capas CONV adyacentes que cuya suma de las dimensiones de sus respectivos filtros sea equivalente a la capa factorizada.

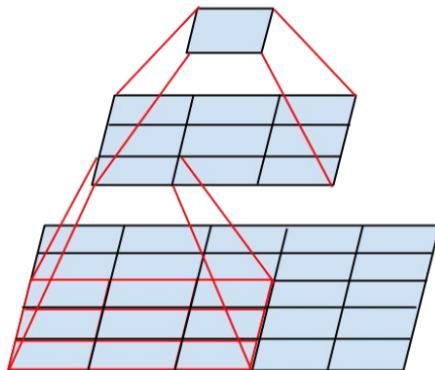


Figura 3.1: Una convolución de 5×5 es un perceptrón multicapa, *per se*. (Tomado de <https://arxiv.org/pdf/1512.00567.pdf>.)

La reducción de parámetros por capa CONV implica, además, un entrenamiento más rápido. El costo radica en aplicar más convoluciones por cada capa CONV cuyo filtro tiene dimensiones *considerablemente* grandes. Para profundizar más en esta idea, imaginemos un filtro de 5×5 . Uno de los beneficios que trae consigo el tener un filtro de tamaño “grande” (por conveniencia, aquí así lo

³ Para más información sobre la arquitectura de *GoogLeNet*, revisar <https://arxiv.org/abs/1409.4842>.

consideramos) es que la cobertura del mismo sobre una imagen le permite capturar mayores detalles de la misma. Ahora, recordemos que una capa CONV puede ser vista como un MLP, como se vio en el capítulo anterior; en realidad, ¡el filtro en cuestión es una capa totalmente conectada que pasa por toda la imagen de entrada! ¿Por qué no descomponer este MLP en varios filtros pequeños aplicados de manera *paralela* sobre la imagen? El resultado es la factorización en dos filtros de 3×3 como se muestra en la Figura 3.1.

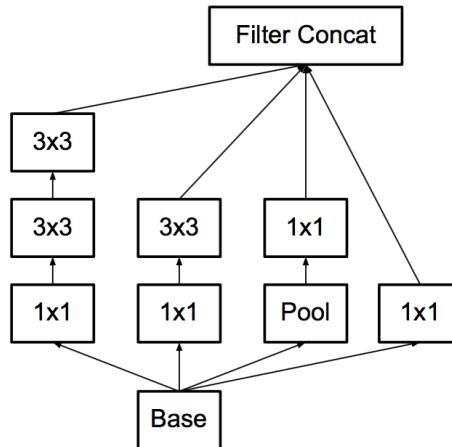


Figura 3.2: Otra manera de factorizar una convolución consiste en paralelizar el cómputo del filtro. Cabe destacar que los parámetros de cada nuevo filtro son *compartidos* entre sí, por cada nueva capa. Aquí reemplazamos un filtro de 5×5 con dos filtros de 3×3 y una capa de pooling. (Tomado de <https://arxiv.org/pdf/1512.00567.pdf>.)

En términos propios de la teoría de lenguajes de programación, podemos afirmar que una red neuronal, vista como un lenguaje formal, posee una estructura que le permite ser construida de manera *inductiva* a partir de pequeñas entidades *bien* definidas. ¿Cómo es que, entonces, se optimiza una red CNN? Minimizando la complejidad del cómputo de sus estructuras básicas. En este caso, aprovechamos el cómputo paralelo para factorizar un filtro, de donde surgen varios “hilos de ejecución” en el grafo de la misma que, deberán ser eventualmente *concatenados* en un solo tensor resultante.

A la estructura presentada en la Figura 3.2, se le denomina comúnmente como módulo *Inception*. Por otro lado, decimos que una convolución con un filtro de 1×1 procesa *localmente* una imagen de entrada pues realiza una activación (filtrado) exhaustiva a nivel de pixeles individuales. En contraste, siguiendo el principio de diseño (**P.3**), la adición espacial incorpora dos o más entradas en una, mientras que la reducción de dimensiones toma un papel clave para garantizar que no haya pérdida de información. No obstante, un incremento en tasa de procesamientos locales contra adiciones espaciales favorece la existencia de representaciones dispersas de dimensiones altas.

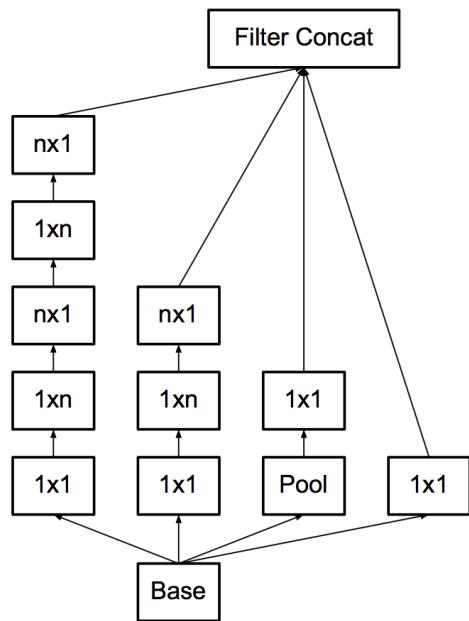


Figura 3.3: Factorización de una convolución de $n \times n$, donde $n = 7$. (Tomado de <https://arxiv.org/pdf/1512.00567.pdf>.)

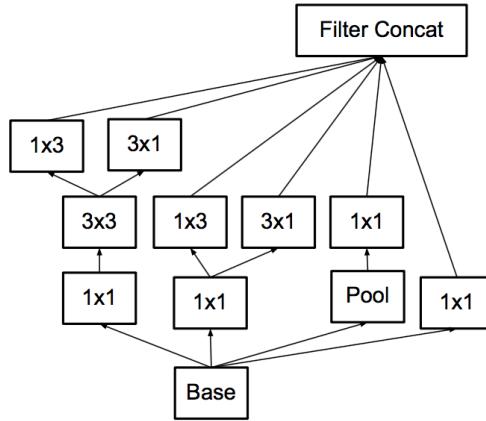


Figura 3.4: Factorización de una convolución de 8×8 . (Tomado de <https://arxiv.org/pdf/1512.00567.pdf>.)

Para concluir con esta sección, presentamos las capas que constituyen a la arquitectura Inception en la Tabla 3.1.

tipo	tamaño de filtro / zancada	tamaño de la entrada
CONV	$3 \times 3/2$	$299 \times 299 \times 3$
CONV	$3 \times 3/1$	$149 \times 149 \times 32$
CONV con relleno de ceros	$3 \times 3/1$	$147 \times 147 \times 32$
POOL	$3 \times 3/2$	$147 \times 147 \times 64$
CONV	$3 \times 3/1$	$73 \times 73 \times 64$
CONV	$3 \times 3/2$	$71 \times 71 \times 80$
CONV	$3 \times 3/1$	$35 \times 35 \times 192$
$3 \times$ Inception	Figura 3.2	$35 \times 35 \times 288$
$5 \times$ Inception	Figura 3.3	$17 \times 17 \times 768$
$2 \times$ Inception	Figura 3.4	$8 \times 8 \times 1280$
POOL	$8 \times 8/2$	$8 \times 8 \times 2048$
LINEAL	logits (log-probabilidades)	$1 \times 1 \times 2048$
MLP-SOFTMAX	clasificador	$1 \times 1 \times 1000$

Tabla 3.1: Las capas que conforman a la arquitectura convolucional *Inception V3*. Normalmente, la dimensión de la última capa se modifica dependiendo de la tarea de clasificación deseada. (Tomado de <https://arxiv.org/pdf/1512.00567.pdf>.)

3.3. Poniendo todo junto

Esencialmente, la estructura de cada unidad LSTM usada en la implementación de la RNN es la misma que la que fue presentada en la última sección del capítulo anterior. Lo único que cambia es la incorporación del tensor de la última capa de Inception ($CNN(I)$), como memoria inicial ($h^{(0)}$). Esto se ilustra en la Figura 3.5.

El entrenamiento de la LSTM se lleva a cabo *desenvolviendo* la recurrencia en un tamaño fijo de copias de la LSTM (digamos, N). En el tiempo $t + 1$, se calcula la probabilidad que tiene cada palabra en el diccionario de ser S_{t+1} a partir de $p(S_t|I, S_0, \dots, S_{t-1})$. En resumen,

$$x_{-1} = CNN(I) \quad (3.3)$$

$$x_t = W_e S_t, \quad t \in \{0, \dots, N - 1\} \quad (3.4)$$

$$p_{t+1} = LSTM(x_t), \quad t \in \{0, \dots, N - 1\} \quad (3.5)$$

donde x_{-1} indica la primera entrada de la LSTM, x_t son las entradas subsecuentes, W_e es un tensor que incrusta la palabra S_t en el mismo espacio vectorial que $CNN(\cdot)$ y p_{t+1} es el tensor de log-probabilidades por cada palabra en el diccionario en el tiempo $t + 1$. Cabe destacar que S_t se representa matricialmente como un vector codificado *en caliente*⁴.

El error de la arquitectura se calcula mediante la verosimilitud logarítmica negativa, sumando cada palabra:

$$L(I, S) = - \sum_{t=1}^N \log p_t(S_t). \quad (3.6)$$

Acto seguido, se minimiza este error con respecto a todos los parámetros de la LSTM, la última (¡ver la siguiente sección!) capa de la CNN y las incrustaciones W_e .

Finalmente, para la generación de enunciados se utiliza el

⁴ En una codificación **en caliente** (*one-hot encoding*, en inglés), se aplica la función indicadora sobre un vector con la dimensión del diccionario de palabras, dejando un 1 en el índice que corresponde a la palabra i y 0 en cualquier otro.

algoritmo de **búsqueda por haces** (*beam search* en inglés). Iterativamente, se considera el conjunto de las k mejores palabras que son candidatas para ser S_t en el tiempo t . Esto es una aproximación de

$$S = \arg \max_{S'} p(S' | I) \quad (3.7)$$

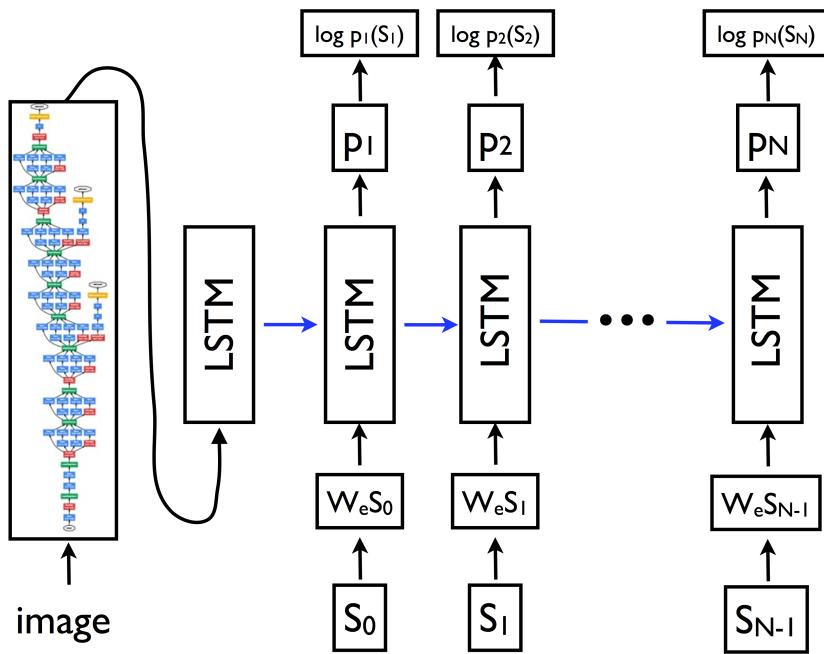


Figura 3.5: La arquitectura *Show and Tell*, propuesta por Vinyals, et al,[23] en la cual se conecta una CNN Inception con una LSTM. (Tomado de <https://arxiv.org/pdf/1411.4555.pdf>.)

3.4. Aprendizaje por transferencia

En cada una de las capas de una CNN *profunda*, como las de la Tabla 3.1, se abstraen distintas características del conjunto de datos aprendido. ¿Por qué colocar capas MLP tras sucesiones de CONV? Otra interpretación de lo que pasa dentro de una capa *densa* consiste en pensar en un cambio de espacio vectorial. Cuando la dimensión de la salida es menor que la de la entrada, podemos afirmar que el

espacio “objetivo” comprime la información contenida en las capas previas. Así es como la última capa de una arquitectura convolucional preserva la información adquirida tras el entrenamiento.

Asumiendo una reducción considerable del error de entropía cruzada y, tras una existosa evaluación a través de distintas métricas⁵, la red convolucional por sí sola pasa a ser un excelente mecanismo de extracción de características de un conjunto de datos. Para entrenar una red de la profundidad de *Inception V3* se requiere una cantidad de imágenes que muchas veces excede los 30 GB⁶ en tamaño total. Además, el hecho de reunir semejante tamaño de imágenes es, en sí, un tema de gran importancia para las ciencias de datos y un trabajo difícil de lograr.

Por ello, la comunidad de **código abierto** (*open source* en inglés) suele compartir los pesos de los parámetros que usaron para entrenar una arquitectura profunda con un conjunto de datos como *Imagenet*.

Dada la riqueza de dicho conjunto de datos, resulta completamente viable suponer que los atributos más importantes que comprende un meme de Internet serán identificados por las capas convolucionales.

Consecuentemente, nos ahorraremos el entrenamiento de una arquitectura convolucional profunda “desde cero” y utilizamos los vectores resultantes de un lote de entrenamiento como memoria inicial de la LSTM. Por otro lado, el nivel de abstracción que adquiere una CNN entrenada con *Imagenet* es demasiado amplio, en comparación, con lo que necesitamos para obtener una buena representación vectorial de un meme⁷. En este caso, es posible utilizar la distribución probabilística (multivariada) resultante de *Inception V3*

⁵ Las métricas de evaluación de una CNN se discuten en la Sección 4.4.

⁶ Desde la concepción del concurso ILSVRC (ver Sección 3.2), se motivó a la comunidad a trabajar con un conjunto de imágenes de entrenamiento de aproximadamente 47 GB. Para más información, consultar <http://image-net.org/challenges/LSVRC/2014/download-images-5jj5.php>.

⁷ Un meme obtenido de <https://memegenerator.net> consiste básicamente en el rostro de un personaje (“viralmente”) famoso centrado en la imagen. Aprender un conjunto de datos obtenido de este sitio se reduce, entonces, a localizar las características más importantes del rostro del personaje en cuestión.

para mejorar la generalización de una distribución que se enfoque en reconocer atributos de memes.

Hablamos, ahora, de un área de estudio conocida como **aprendizaje por transferencia**. A partir del aprendizaje de una distribución P_1 , ¿qué tanto ésta puede ser explotada para aprender otra distribución P_2 ? Para un conjunto de datos como *Imagenet*, es común que las primeras capas de un arquitectura profunda se especialicen en detectar pequeños atributos: desde bordes, cambios de brillo, texturas y partes sombreadas hasta estructuras geométricas básicas (Figura 3.6). Tiene sentido, entonces, utilizar la especialización de estas capas para extraer las características más fundamentales de un conjunto de memes.

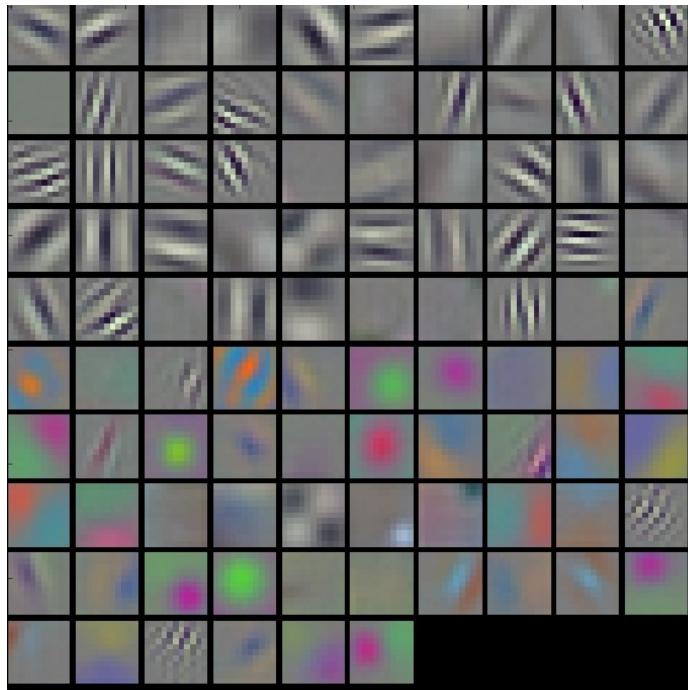


Figura 3.6: Así se ven los filtros entrenados de la primera capa de una CNN profunda. En este caso se trata de una AlexNet. La literatura suele conocer a los resultados de estos filtros como “*características generales*”. (Tomado de <http://cs231n.github.io/understanding-cnn/>.)

El procedimiento consecuente a la idea del aprendizaje por

transferencia consiste en **afinar** la red neuronal, continuando con el algoritmo de propagación hacia atrás. Esto se hace con un nuevo conjunto de datos (el de memes). Usualmente, se propaga el error únicamente sobre algunas de las últimas capas (más *abstractas*), del modelo; si se vuelven a entrenar todas las capas, cabe la posibilidad de llegar a un sobreajuste [26].

El enfoque tomado para este trabajo consiste en experimentar con una red *Inception V3* como extractora de características. Más aún, se probará afinando dicha arquitectura con el conjunto de datos reunido. El tamaño del mismo permite hacer dicha afinación.

3.5. La arquitectura en acción

Antes de pasar a describir los experimentos realizados para esta tesis, vale la pena echar un vistazo formal a cómo funciona el modelo propuesto. De acuerdo a las elecciones de implementación (descritas en el siguiente capítulo), la arquitectura tiene tres modos de operación: **entrenamiento, inferencia y evaluación**. Éstos dos últimos consisten en hacer predicciones, utilizando el algoritmo de búsqueda por haces para generar un enunciado o añadiendo métricas de evaluación, respectivamente.

En cualquiera de estos modos de operación, una entrada del modelo se puede entender como un par ordenado (I, S) , donde

- I es un tensor que representa a una imagen (cuadrada) con dimensión fija (n), es decir, posee tres dimensiones de la forma $[n, n, 3]$;
- S es un vector cuya dimensión se define por el tamaño del vocabulario obtenido a partir de todas las palabras que conforman el conjunto de datos.

Usualmente, el vocabulario se construye enumerando todas las palabras existentes (sin repeticiones); S está formado, entonces,

por los índices numéricos de las respectivas palabras en el vocabulario. En el modo de **entrenamiento**, el algoritmo de descenso por el gradiente estocástico sugiere utilizar lotes de datos, por lo que los valores de las dimensiones tanto de I como de S suelen ser de la forma `[tamaño_lote, n, n, 3]` y `[tamaño_lote, tamaño_vocabulario]`, respectivamente.

En los tres modos, se procede con el cálculo de $CNN(\cdot)$. En el modo de **entrenamiento** se sigue hasta el cálculo dado por la Ecuación 3.5 y se propaga el error sobre la LSTM desenvuelta. Tanto en **inferencia** como en **evaluación** se utiliza la LSTM en su forma recurrente, dado que será necesario utilizar el tensor resultante de la Ecuación 3.5.

4

Experimentación y evaluación del desempeño de la red

EN este capítulo se detallarán los experimentos hechos con la arquitectura expuesta anteriormente. La metodología llevada a cabo consistió en reunir la mayor cantidad posible de datos para seguir con los procedimientos descritos en la Sección 3.5. Los alcances de los experimentos serán descritos más adelante, dejando en claro las restricciones de tiempo (para la realización de la tesis) y capacidad de equipo de cómputo disponible.

Con respecto a lo último mencionado, para los cómputos de mayor desempeño se utilizó una **tarjeta gráfica** (*GPU*, por sus siglas en inglés) *GeForce GTX 1080 Ti* de la marca *NVIDIA*¹. Es común que en aprendizaje profundo se usen este tipo de componentes, que originalmente se construyen para el mundo de los videojuegos.

¹ Para mayor información con respecto al modelo *GeForce GTX 1080 Ti*, consultar el sitio <http://la.nvidia.com/graphics-cards/geforce/pascal/la/gtx-1080-ti>.

4.1. Estructura del conjunto de datos

Como ya fue mencionado anteriormente, los datos se obtuvieron del sitio `MemeGenerator.net`². La información recabada en este sitio es reunida a través de usuarios alrededor del mundo, quienes de manera libre tienen la posibilidad de generar un nuevo meme. El sitio, además, agrupa a los memes en personajes (Figura 4.1) y los jerarquiza en base a su popularidad³.

En total se reunieron **4379** personajes del sitio web antes mencionado. Por cada uno de ellos el número de leyendas usadas para entrenar varía según los resultados obtenidos en cada experimento. Considerando que es necesario dividir un conjunto de datos en subconjuntos de entrenamiento y validación⁴, podemos afirmar que el tamaño de los datos de entrenamiento es pequeño en relación a otros conjuntos como *ImageNet*.

Mediante una búsqueda *por profundidad*, a través del árbol de páginas web, definido por cada uno de los personajes del sitio, uno encuentra una gran cantidad de leyendas separadas de la imagen de su personaje asociado. Es decir, por cada personaje se extrae una imagen y tantas leyendas como sea posible (Figura 4.4).

Para llevar a cabo este fin, se programó un **rastreador web** (*web crawler*) para llevar a cabo la búsqueda de la información y la acumulación de los datos. Se escribió un programa en el lenguaje de programación **Python** (versión 3.6)⁵ con la biblioteca *Scrapy* (versión 1.4.0)⁶, la cual organiza hilos de ejecución concurrentes para obtener el contenido de las URL's necesarias mediante peticiones GET de HTTP. Al final, se organizaron los datos bajo la jerarquía presente en la Figura 4.2.

²<https://memegenerator.net>

³ Más aún, cualquier usuario puede crear libremente un personaje nuevo, lo que habilita la idea de buscar y agrupar los memes de acuerdo a su personaje.

⁴ Usualmente esta división se realiza muestreando aleatoriamente del 70 al 80 % de los datos para entrenamiento y del 30 al 20 % para validación.

⁵<https://www.python.org>

⁶<https://scrapy.org>



Figura 4.1: Ejemplos de personajes populares del sitio <https://memegenerator.net>. La mayoría de ellos gozaba de gran popularidad entre los años 2009 y 2013; la evolución de la viralidad de los mismos y el surgimiento de otros sitios web para compartir memes son algunas causas que pueden explicar su “extinción”. (Tomado de <https://memegenerator.net>.)

	caption	img_url	language
0	meme generator users y u no give me more upvotes?	https://memegenerator.net/img/instances/250x25...	English
1	steve jobs y u no respawn?!	https://memegenerator.net/img/instances/250x25...	English
2	commercial y u no same volume as show!?	https://memegenerator.net/img/instances/250x25...	English
3	KONY Y u no take justin bieber	https://memegenerator.net/img/instances/250x25...	English
4	TED y u no tell us how you met their mother	https://memegenerator.net/img/instances/250x25...	English
5	Victoria y u no tell us your secret?!	https://memegenerator.net/img/instances/250x25...	English
6	Google Y U NO LET ME FINISH TYPING?	https://memegenerator.net/img/instances/250x25...	English
7	pink floyd y u no need no education?	https://memegenerator.net/img/instances/250x25...	English
8	universal remote y u no work on universe?	https://memegenerator.net/img/instances/250x25...	English

Figura 4.3: De esta manera se guardaron cada una de las leyendas de cada personaje.

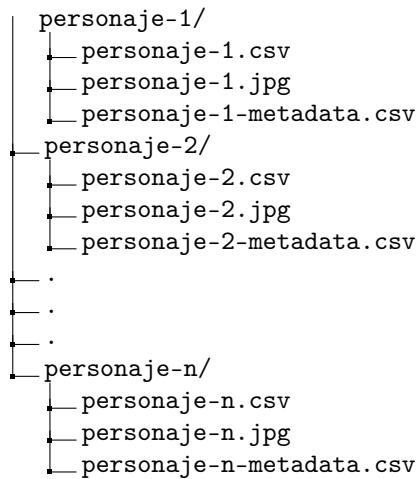


Figura 4.2: Como se observa, los datos se guardaron bajo el formato CSV, asociando cada leyenda con la URL de su meme asociado y su idioma de origen (Figura 4.3). Se estima que un 90 % de datos está en inglés.

4.2. Generación de leyendas

El modo **inferencia** del modelo consiste en generar una leyenda para una imagen de entrada. Dada la manera con la cual se representa el modelo de lenguaje aprendido del conjunto de datos de entrenamiento, sabemos que por cada palabra hay una distribución de probabilidad que evalúa qué tan viable es que cada palabra del vocabulario sea la que sigue. Por lo tanto, muchas veces conviene observar los k enunciados que *mejor* describen una imagen, maximizando la probabilidad conjunta entre cada una de las palabras que los componen (en orden).

La **búsqueda por haces** (*beam search*, en inglés) es un algoritmo que logra calcular enunciados de “*máxima verosimilitud*”. El procedimiento incorpora a un agente cuyo objetivo es encontrar el camino de mayor *peso* posible en una máquina de estados, en la cual solo tiene conocimiento del estado actual y los pesos para llegar a los vecinos del mismo. La salida del algoritmo muestra los k enunciados más viables para cierta imagen. El agente, entonces registra de manera paralela las k palabras de mayor probabilidad



Figura 4.4: La ilustración de la parte superior constituye a un meme que integra a un personaje con su leyenda y es el objeto que se propaga a través de Internet. Los datos que se reunieron se separaron como se indica en la ilustración de la parte inferior. (Tomado de <https://memegenerator.net>.)

que sucedan a la palabra anterior.

Normalmente, esto se programa mediante k hilos de ejecución en paralelo. Cada uno se inicializa aleatoriamente con las k palabras de inicio de mayor probabilidad. En el paso t , cada hilo vuelve a calcular los k siguientes mejores estados (un total de k^2 estados) y, al final, el agente se queda con los mejores k para proceder al paso $t + 1$. En este caso, se consideraron los $k = 3$ mejores enunciados para cada imagen.

4.3. Experimentos

Los experimentos realizados obedecen a lo sugerido por la teoría presentada en los dos capítulos anteriores. Se buscó seguir la metodología sugerida por el *estado del arte* ([24]). Por ello, se eligió trabajar con el lenguaje de programación Python (versión 3.6) y las bibliotecas **Tensorflow** (versión 1.3.0)⁷ y **Keras** (versión 2.0.9)⁸.

Ambas bibliotecas son obra *reciente* de la división de código abierto de *Google*. *Tensorflow* surgió como una iniciativa para compartir la manera en que dicha empresa despliega sus proyectos que involucran aprendizaje profundo. El paradigma que se sigue consiste en definir un grafo dirigido de *tensores* como nodos, en el cual fluirá la información⁹; acto seguido, se “compila” el modelo y se levanta una sesión para ejecutarlo. Así, *Tensorflow* utiliza la sintaxis de *Python* para construir redes neuronales en un mayor nivel.

Por otro lado, *Keras* se define a sí misma como una interfaz de programación de aplicaciones (*API*) de alto nivel, que utiliza como motor de ejecución a *Tensorflow*. Es decir, el programador es capaz de escribir código que defina una red neuronal y se ejecute secuencialmente. *Keras*, además, facilita la tarea de definir un mo-

⁷<https://www.tensorflow.org>

⁸<https://keras.io>

⁹En cada nodo del grafo, se definen *operaciones* (lectura y escritura de archivos, operaciones matriciales, etc.) y se da la posibilidad de especificar si van a correr en un GPU, si se cuenta con uno.

delo profundo con una sintaxis más amigable que la de *Tensorflow* y configura algunos *hiper-parámetros* de ésta, con el fin de facilitar el prototipado de redes neuronales profundas.

4.3.1. Experimentos exploratorios

Vinyals, *et al* presentan tanto en [23] como en [24] el *estado del arte* de los modelos neuronales capaces de generar descripciones a partir de imágenes. Por ende, el primer paso experimental consistió en replicar el entrenamiento completo, usando el conjunto de datos *ImageNet*.

Utilizando pesos pre-entrenados¹⁰ de Inception V3, se realizó el proceso de generar un tensor de incrustaciones de las imágenes de ImageNet en un espacio vectorial. Acto seguido se alimentó una LSTM con dicho tensor, como memoria inicial, y se entrenó utilizando las leyendas asociadas a cada imagen.

Dentro de las ventajas consecuentes de este experimento, está el tener una implementación de una LSTM para ser entrenada de nuevo, con cualquier memoria inicial. El desempeño del entrenamiento se ilustra en la Figura 4.5. Es importante destacar que, dadas las altas dimensiones con las que se trabaja en cada capa, se estima que se requieren alrededor de 1 millón de épocas para que el error dado por la entropía cruzada se estabilice en un valor mínimo (este comportamiento es común para la mayoría de los experimentos subsecuentes).

El objetivo principal de este primer experimento era familiarizarse con el uso de las implementaciones y la tecnología necesaria para realizar aprendizaje profundo. Por ende, se decidió dejar a un lado la evaluación del desempeño de este modelo neuronal, mediante un conjunto de datos independiente del de entrenamiento y valida-

¹⁰ *Tensorflow* provee un conjunto de modelos previamente entrenados bajo un subconjunto de *ImageNet* (<http://www.image-net.org/challenges/LSVRC/2012/>). Esto ahorra tiempo en la vectorización de imágenes. Para mayor información, consultar el sitio <https://github.com/tensorflow/models/tree/master/research/slim#tensorflow-slim-image-classification-library>.

ción.

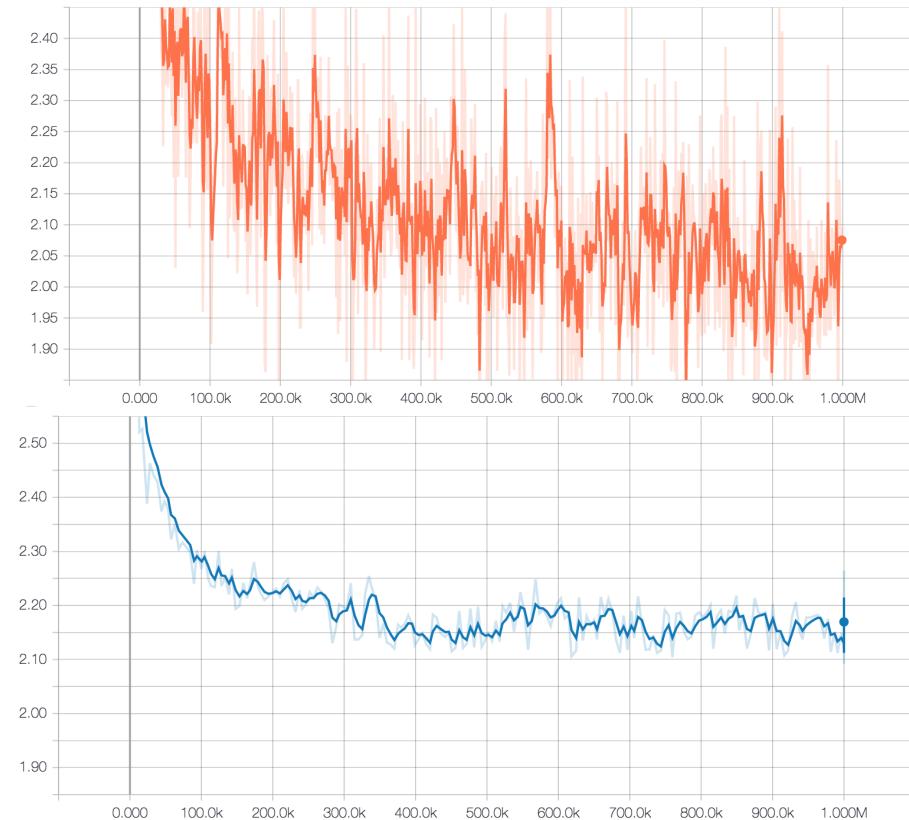


Figura 4.5: Función de error de entropía cruzada para el primer experimento exploratorio. En el gráfico de la parte superior se muestra el error a través de cada época del entrenamiento, mientras que el de la parte inferior se calculó por medio de un conjunto de datos de *validación* (de menor tamaño que el de entrenamiento). Ambos gráficos indican el valor de la entropía cruzada (eje de las ordenadas) a través del tiempo (eje de las abscisas). (Fuente: elaboración propia.)

Tenemos, ahora, una CNN que es “experta” en etiquetar imágenes con detalles muy generales. Vale la pena, entonces, probarla con el conjunto de datos de memes. Buscamos ver, de entrada, que sea capaz de distinguir entre dos imágenes distintas (mediante representaciones vectoriales muy diferentes), que además se refleje en la discrepancia entre las leyendas generadas. Para ello, diseñamos

un experimento en el que entrenamos una LSTM con una memoria inicial dada por los códigos convolucionales obtenidos a partir de **97** personajes del conjunto de datos. Estos personajes pasarán por la red *Inception V3* pre-entrenada del experimento anterior.

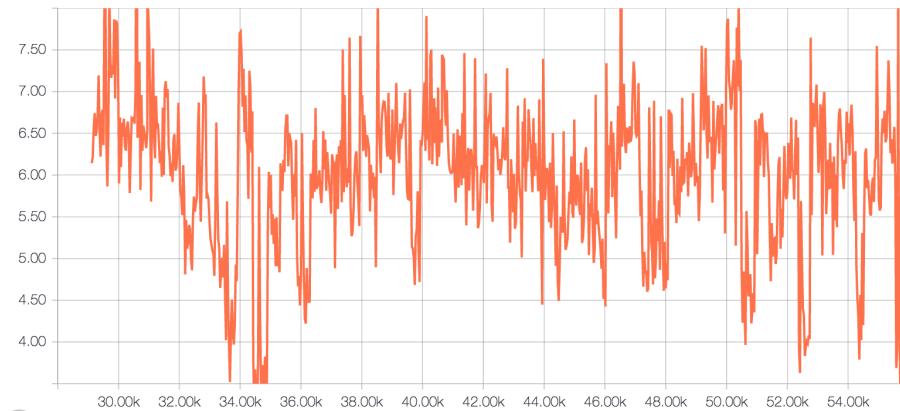


Figura 4.6: Función de error para el experimento que utiliza la CNN *Inception V3* con pesos pre-entrenados, sin ser afinada. (Fuente: elaboración propia.)

Cada personaje posee un promedio de 700 leyendas, pues el número total de leyendas por personaje varía según la popularidad del mismo. Cabe destacar que el vocabulario construído tuvo alrededor de 20 mil palabras distintas. El desempeño fue deficiente tanto en entrenamiento (Figura 4.6) como en resultados *anecdóticos*. Las siguientes observaciones fueron notables:

- los códigos convolucionales resultantes del paso por la CNN fueron indistinguibles entre una imagen y otra, dejando en claro la necesidad de realizar una afinación);
- lo anterior provocó que las leyendas fueran todas iguales para imágenes distintas;
- el elevado tamaño del vocabulario no favoreció al modelo para poder asignar probabilidades, condicionales, al momento de generar frases (repetición palabras de manera contigua).

El bajo desempeño de este experimento motivó a *no* realizar una evaluación de los resultados pero trajo consigo dos hipótesis

para considerarse. La primera de ellas consiste en afinar la red *Inception V3* con el conjunto de datos de memes, pues se cree que el modelo implícito en sus parámetros pre-entrenados explora detalles mucho más generales de los requeridos para clasificar imágenes como las mostradas en la Figura 4.1. Dado el tamaño del conjunto de datos, se plantea, de igual manera, como hipótesis, la viabilidad de que una CNN más **superficial** (menos profunda) tenga éxito al generar códigos convolucionales.

4.3.2. Experimentos que involucran la afinación de una arquitectura convolucional profunda

Para aprovechar la especialización que posee *Inception V3*, de reconocer patrones simples en imágenes, afinamos dicho modelo con un clasificador de memes. Dado que el conjunto de datos presentado en la Sección 4.1 no está dividido en “categorías” de memes, resulta difícil realizar una tarea de aprendizaje supervisado con una CNN.

Por otro lado, dadas las características de las imágenes de los memes, existe un patrón más o menos definido (rostro del personaje centrado en la imagen) que puede ser explotado contra la tarea general de aprender a clasificar *ImageNet*. Esto nos brindó la solución para poder afinar a *Inception V3*: colocar una capa MLP, al final de la red, con salida de dos dimensiones para decidir si la entrada es, o no es, un meme.

Las imágenes “*no memes*” utilizadas en este punto se tomaron aleatoriamente muestreando 4379 de *ImageNet*. Se trabajó bajo la premisa de que la profundidad de la red alcanzará para generalizar la identificación de las características presentes en un meme. Después de todo, nuestro conjunto de datos posee imágenes de mucho menor complejidad que las existentes en *Inception V3*. En total, se dejaron 3065 imágenes (memes y no memes) para entrenamiento, 657 para validación y 657 para evaluación; el resultado de este proceso se ilustra en la Figura 4.7. El desempeño fue bastante favorable, ya que se minimizó considerablemente el error de la entropía

cruzada. Además, con el conjunto de evaluación se corroboró dicho rendimiento mediante los gráficos de las métricas de *precisión*¹¹ y *error medio absoluto*¹² (Figura 4.8).

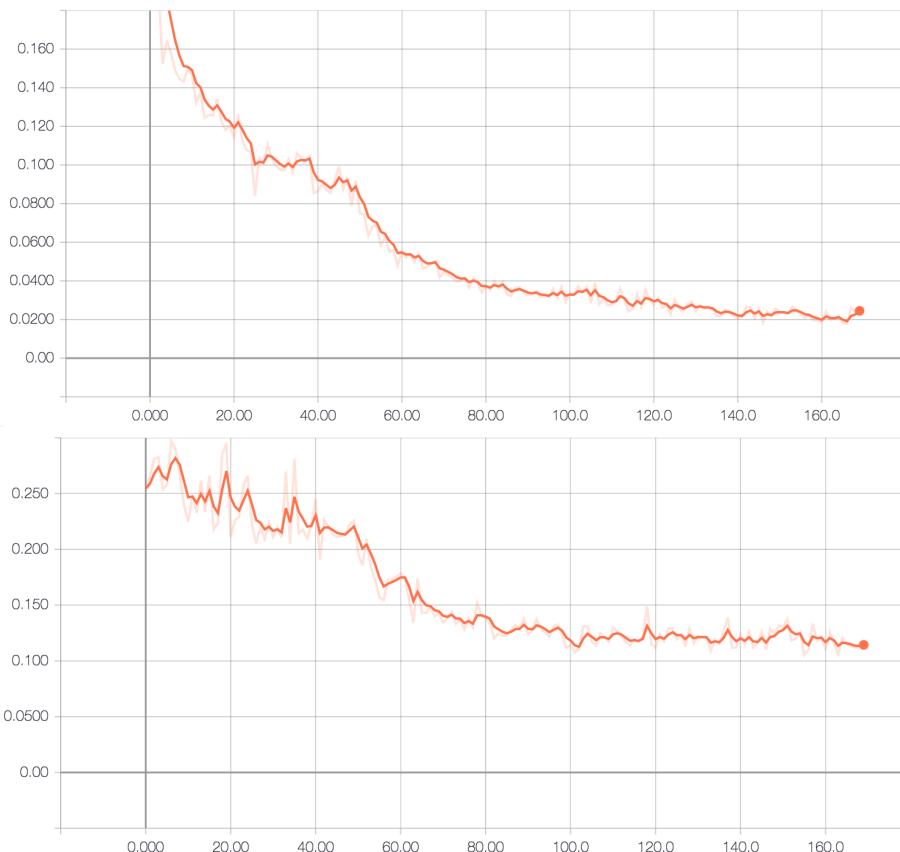


Figura 4.7: Funciones de error de entrenamiento y validación (gráfico de arriba y de abajo, respectivamente) para la afinación de *Inception V3*. Esto es, entrenar un clasificador entre lo que es un meme y lo que no es. Se quita la última capa y se añade un MLP con dos dimensiones de salida. (Fuente: elaboración propia.)

¹¹ La **precisión** (*accuracy*, en inglés) puede ser entendida como el porcentaje de aciertos que tiene el modelo al intentar predecir un lote de datos. Se busca, entonces, que su valor esté muy cercano a 1.

¹² El **error medio absoluto** es la diferencia promedio que existe entre los valores estimados por un modelo (\hat{Y}) y los valores reales de un conjunto de datos (Y). Esto se calcula con la expresión

$$\frac{\sum_{i=1}^n \hat{Y}_i - Y_i}{n}.$$

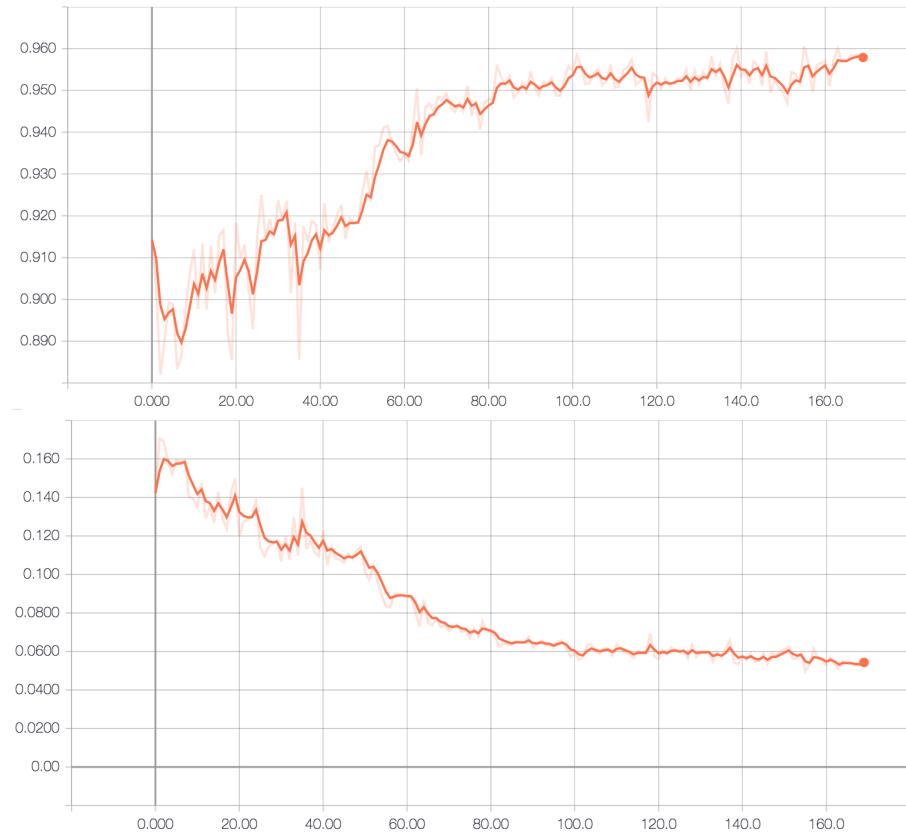


Figura 4.8: Dos funciones de evaluación para la afinación de *Inception V3*. El gráfico de la parte superior muestra la *precisión* con la que el modelo realiza sus clasificaciones, mientras que el gráfico de la parte inferior muestra el *error medio absoluto* entre las clasificaciones hechas por el modelo contra las verdaderas. Ambos gráficos se generaron a partir de un muestreo del 10 % de los datos disponibles. (Fuente: elaboración propia.)

Una vez teniendo afinada una red neuronal profunda, surge la inquietud de ver si esto ayuda a mejorar el primer experimento en el que se entrenó la LSTM usando memes. Por ello se usaron los mismos datos que los que se usaron para dicho experimento pero con la *Inception V3* previamente afinada. Esto provocó que entre cualesquiera dos imágenes diferentes, existieran dos códigos convolucionales con valores suficientemente distintos; lo que implica que las leyendas generadas serán también distintas.

El tensor de imágenes y leyendas de entrada (I, S) se fue construyendo en orden, de manera que todas las leyendas de un solo personaje permanecían en posiciones contiguas. Dada la cantidad desmedida de leyendas, esto provocó que tras ciertas iteraciones se sobreentrenara el modelo sobre un mismo lote. Así, al probar el modo de **inferencia** del mismo, se podía observar claramente una tendencia por repetir la “manera de hablar” de un cierto personaje (Tabla A.1). Iteraciones más tarde, esto cambiaba para que el modelo comenzaría a repetir la manera de hablar de otro personaje. Una de las evidencias del sobreajuste se ilustra en la Figura 4.9.

A pesar que las leyendas generadas aparentan tener sentido, el sobreajuste que se dio en varios momentos del entrenamiento motivó a realizar un mayor trabajo de procesamiento previo. En particular, se destaca la importancia de restringir el tamaño del vocabulario, así como de mezclar aleatoriamente los tensores de memoria inicial que alimentarán la LSTM. Esta última acción garantiza que no se repitan ciertas frases un número de veces elevado en cada lote de entrenamiento, por lo que se espera una reducción en el sobreajuste. Por lo tanto, se omitió la evaluación de este experimento, esperando resultados más apegados a la realidad de la distribución del conjunto de entrenamiento.

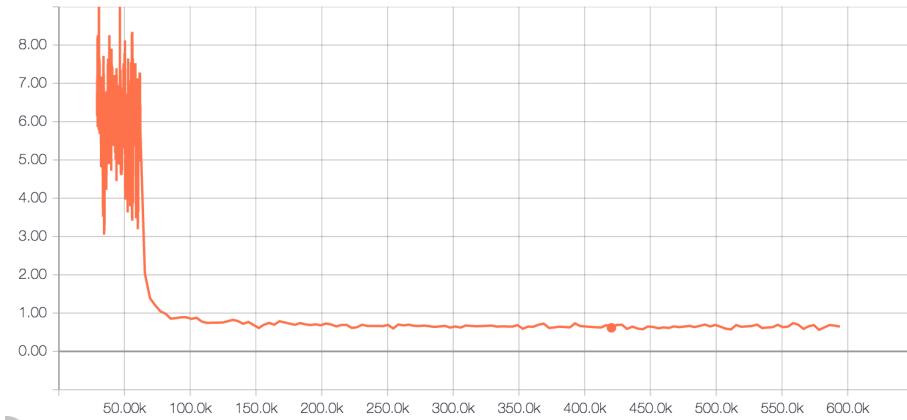


Figura 4.9: Función de error para el entrenamiento de la LSTM, dándole como estado inicial un tensor generado a partir de la red *Inception V3* afinada. (Fuente: elaboración propia.)

Para refinar el desempeño de la CNN afinada, el siguiente experimento se diseñó muestreando aleatoriamente **3416** personajes. Además, se eliminaron todas las palabras que no aparecen al menos 5 veces en todo el conjunto de datos y se usaron únicamente 5 leyendas para cada personaje, con el fin de reducir el tamaño del vocabulario. Por consiguiente el vocabulario se redujo a 4528 palabras.

Previo al entrenamiento, se mezclaron los datos de manera aleatoria en el tensor (I, S); esto provocó la diversificación en la manera con la que se generan leyendas para imágenes que no aparecen en el conjunto de entrenamiento. El comportamiento de la función de error se muestra en la Figura 4.10.

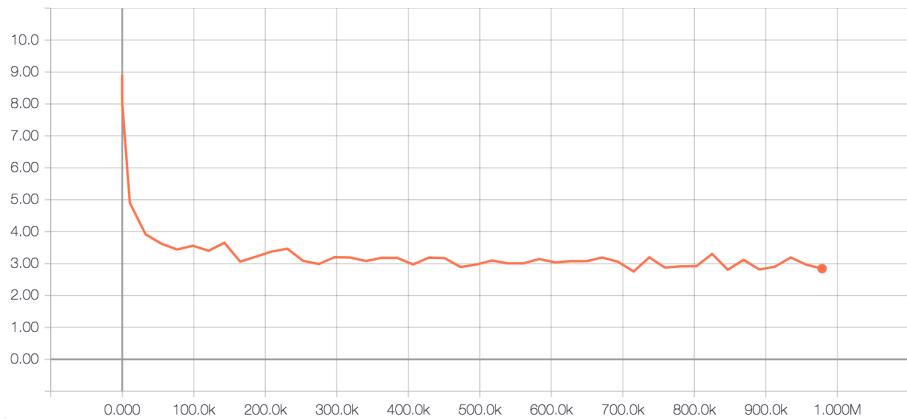


Figura 4.10: Función de error para el experimento que involucra la red *Inception V3* afinada y 5 leyendas por personaje en el entrenamiento. (Fuente: elaboración propia.)

Qualitativamente, la Figura 4.10 muestra una curva descendente más suave, lo que hace pensar que el modelo, en efecto, aprendió a generalizar ciertos detalles del conjunto de datos. En promedio, podemos afirmar que el error de la entropía cruzada baja, con los cambios hechos y da cierto grado de confianza debido a la mezcla aleatoria de las leyendas. Por otro lado, queda la duda de qué pasaría si el tensor de códigos convolucionales es generado a partir de una CNN más superficial, algo que es válido plantearse debido al número de imágenes disponibles.

Este último experimento será el que mejor desempeño trae de los que usaron la red *Inception V3* afinada. Efectuar métricas de evaluación como *precisión* y *error medio absoluto* es una práctica poco común en tareas que involucran generación de texto: la calidad de las leyendas generadas debe ser comparada con un modelo de lenguaje real. Por ello, en la Sección 4.4 presentamos una evaluación basada en una métrica común para el procesamiento del lenguaje natural. La Tabla A.2 muestra algunos resultados anecdóticos de este experimento.

4.3.3. Experimentos que involucran una arquitectura convolucional superficial

Aunque afinar una CNN profunda, previamente entrenada, es un procedimiento justificado por la literatura, el número máximo de imágenes que conforman el conjunto de datos sugiere otro tratamiento. Esto indica que entrenar una red neuronal desde cero no es una mala apuesta, después de todo, de acuerdo a [26].

Ahora, presentamos el entrenamiento de una red neuronal superficial (*bastante menos profunda*). Se usaron los mismos datos que para la afinación de *Inception V3*. La arquitectura de esta red se muestra en la Tabla 4.1, el desempeño del entrenamiento se ilustra en la Figura 4.11, mientras que la evaluación del modelo se encuentra en la Figura 4.12.

¹³ Las técnicas de **regularización**, en aprendizaje automático, añaden una restricción adicional al problema en cuestión para evitar que los valores del modelo propuesto se ajusten completamente al conjunto de datos de entrenamiento. Es decir, evitar el *sobreajuste* y favorecer la generalización hacia datos no observados durante el entrenamiento.

tipo	tamaño de filtro	número de filtros
CONV	3×3	32
CONV	3×3	16
POOL	2×2	-
DROPOUT	25 % de las neuronas se ignoran	-
MLP	128 unidades de salida	-
DROPOUT	50 % de las neuronas se ignoran	-
MLP	2 unidades de salida	-

Tabla 4.1: Arquitectura utilizada para entrenar una CNN superficial. Las capas **DROPOUT** constituyen una popular técnica de *regularización*¹³en la que se descarta un porcentaje dado de las neuronas de entrada con el fin de evitar un sobreajuste sobre el conjunto de datos en cuestión.

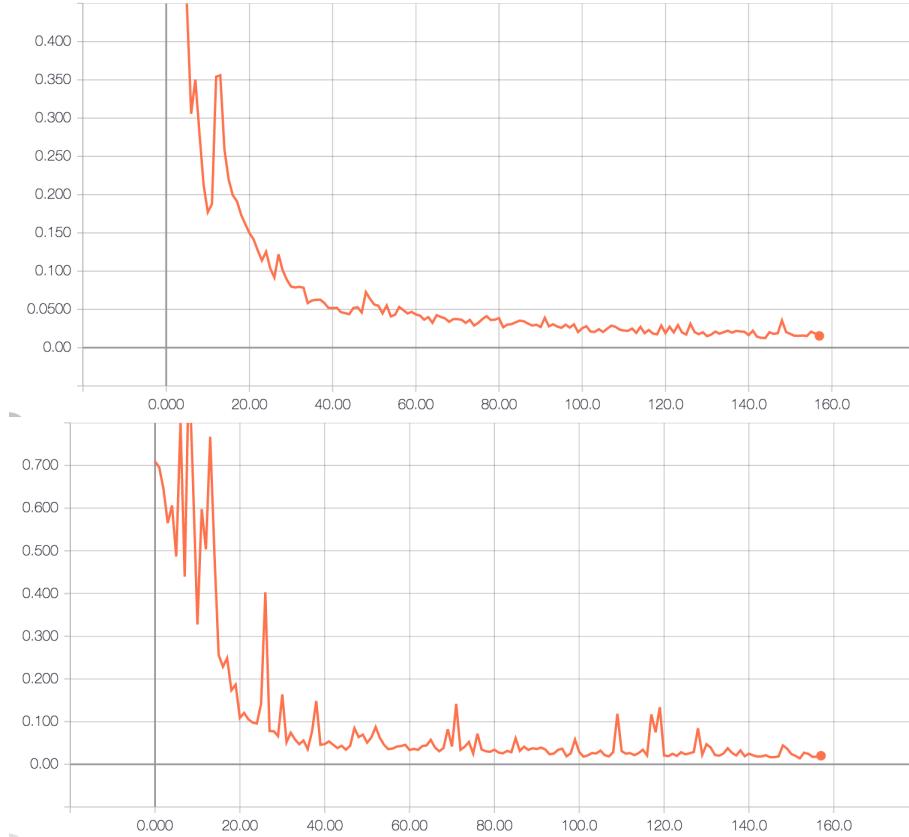


Figura 4.11: Funciones de error de entrenamiento y validación (gráfico de arriba y de abajo, respectivamente) de la CNN superficial. Obsérvese que solo se requirieron alrededor de 150 épocas de entrenamiento para llegar a un valor óptimo; esto dado el tamaño del conjunto de datos (*memes* y *no memes*) que es bastantes órdenes de magnitud más pequeño el conjunto de datos de entrenamiento presentado en la Sección 4.1. (Fuente: elaboración propia.)

Procedemos, entonces, a replicar el último experimento realizado en la sección anterior. Sin embargo, esta vez utilizamos la CNN superficial entrenada sobre memes y no memes. El entrenamiento arrojado (Figura 4.13) fue muy similar a lo observado en la Figura 4.10; no obstante, se logró una mejora en la calidad de las leyendas sin ser un detalle anecdótico muy considerable (Tabla A.3).

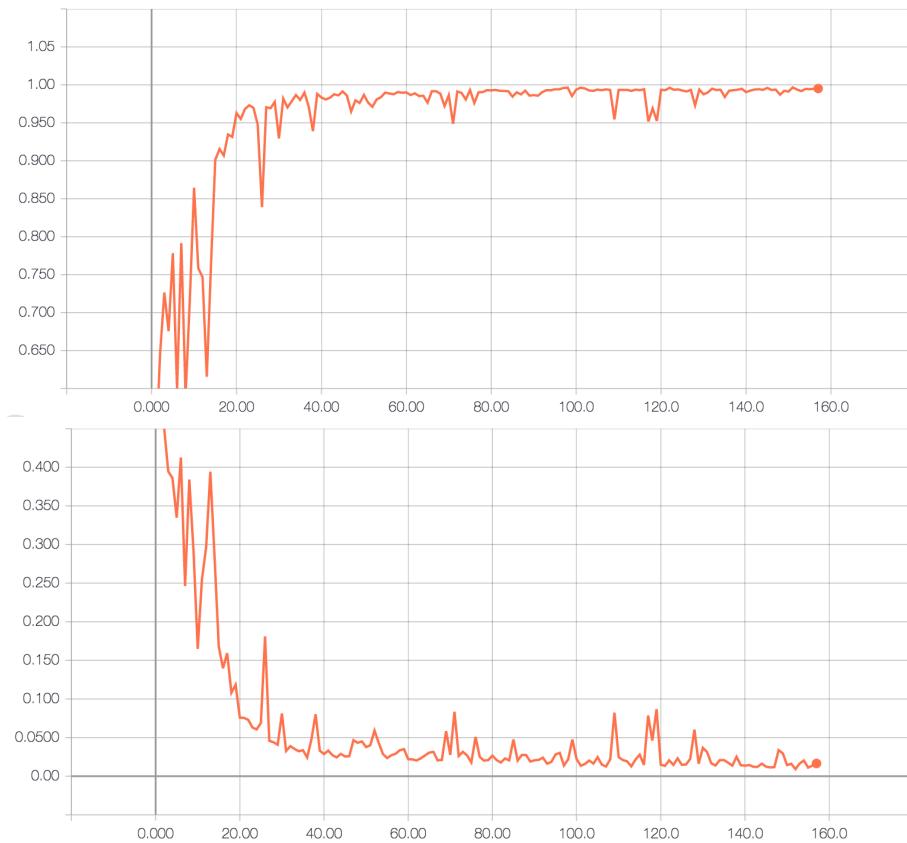


Figura 4.12: Dos funciones de evaluación para el modelo convolucional superficial. El gráfico de la parte superior muestra la *precisión* con la que el modelo realiza sus clasificaciones, mientras que el gráfico de la parte inferior muestra el *error medio absoluto* entre las clasificaciones hechas modelo contra las verdaderas. Ambos gráficos se generaron a partir de un muestreo del 10 % de los datos disponibles. (Fuente: elaboración propia.)

Hasta este punto, podemos decir que hemos construido dos modelos convolucionales capaces de distinguir entre cualesquiera dos

personajes. La tarea pendiente recae en hallar los (*híper*)-parámetros necesarios para que las leyendas generadas tengan más sentido a juicio de un ser humano. Por ello, reportamos un último experimento que involucra un mayor número de palabras por personaje.



Figura 4.13: Función de error para el entrenamiento de la LSTM, usando 5 leyendas por personaje, un vocabulario reducido y códigos convolucionales generados a partir de una CNN superficial. (Fuente: elaboración propia.)

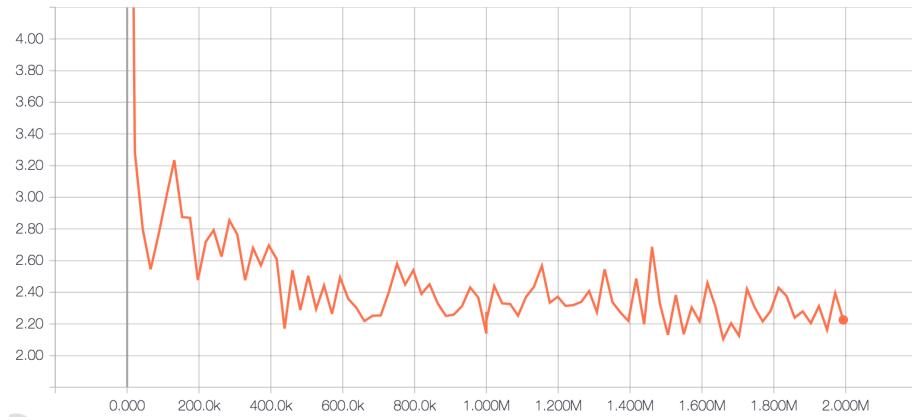


Figura 4.14: Función de error para el entrenamiento de la LSTM, con códigos convolucionales dados por una CNN superficial y 20 leyendas por imagen. Como se observa, existe una ligera disminución en el error resultante al final del entrenamiento, con respecto al obtenido en experimentos anteriores. (Fuente: elaboración propia.)

Con los mismos datos que para el experimento anterior y con la CNN superficial, se realizó el entrenamiento cuyo desempeño se ilustra en la Figura 4.14. Ahora se aumentó a 20 el número de leyendas por imagen. En cuanto a detalles anecdóticos, la Tabla A.4 ilustra una generación de leyendas con un poco de más sentido cuando hay dos palabras distintas contiguas. Resumimos los detalles técnicos de todos los experimentos presentados en la Tabla 4.2.

tipo de experimento	número de personajes	número de leyendas por personaje
Réplica de entrenamiento de <i>Inception V3</i> , como en [24]		Conjunto de datos <i>ImageNet</i>
LSTM a partir de <i>Inception V3</i>	97	700, en promedio
Afinación de <i>Inception V3</i>		3065 memes y 3065 no memes
LSTM a partir de <i>Inception V3</i> afinada	97	700, en promedio
LSTM a partir de <i>Inception V3</i> afinada	3416	5
Entrenamiento de una CNN superficial		3065 memes y 3065 no memes
LSTM a partir de la CNN superficial	3416	5
LSTM a partir de la CNN superficial	3416	20

Tabla 4.2: Resumen de los detalles técnicos presentados en esta sección.

4.4. Evaluación del desempeño de la LSTM

En el contexto del procesamiento del lenguaje natural, es común preguntarse qué tan *buenos* son los enunciados generados con respecto a un lenguaje referencia. En el caso que concierne a esta tesis, tratamos con un lenguaje informal, generado a partir de la popularidad que adquieren ciertas frases dentro del Internet. Sin embargo, es posible discernir entre un enunciado que hace sentido empírico a uno que combina palabras sin patrón alguno.

Las métricas propuestas en [24] sugieren el uso de un corpus lingüístico de referencia, con el cual se compare la similitud entre las estructuras gramaticales generadas por el modelo con enunciados “reales”. Por ello, hemos construido un corpus lingüístico formado por leyendas en el conjunto de *evaluación* para fines de realizar una comparación con los resultados emitidos por la LSTM.

En particular, analizamos qué tan bien aprende el modelo a predecir la t -ésima palabra, a nivel probabilístico. Dada la distribución de probabilidad p sobre los enunciados realizados a partir de un alfabeto Σ^{14} , definimos la **perplejidad** $PP(S)$ del modelo como

$$PP(S) = p(s_1 s_2 \dots s_n)^{-\frac{1}{n}}, \quad (4.1)$$

donde $S = s_1, s_2, \dots, s_n \in S^*$. Desarrollando la Ecuación 4.1, tenemos que

$$PP(S) = \sqrt[n]{\frac{1}{p(s_1 s_2 \dots s_n)}} \quad (4.2)$$

$$= \sqrt[n]{\prod_{i=0}^n \frac{1}{p(s_{i+1} | s_1 s_2 \dots s_i)}} \quad (4.3)$$

La Ecuación 4.3 se obtuvo aplicando la *regla de la cadena* de las probabilidades conjuntas a la predicción de cada palabra s_i . Lo que este razonamiento nos indica es que minimizar la perplejidad de un enunciado S con respecto a un modelo p , es lo mismo que maximizar la probabilidad de S bajo p .

Intuitivamente, la *perplejidad* del modelo nos dice el número de posibles candidatos que tiene el modelo para la palabra s_{i+1} , dadas s_1, s_2, \dots, s_i . Si la *perplejidad* es minimizada, entonces el modelo aprende con éxito a descubrir patrones de secuencias de palabras.

Queremos comparar los dos *mejores* modelos entrenados, es decir, el mejor de los que codifican imágenes con la arquitectura convolucional superficial contra el mejor de los que codifican con *Inception V3*. Llamémoslos **Modelo A** y **Modelo B**, respectivamente. Para exponerlos a un conjunto de imágenes ajenas a los datos de entrenamiento, se tomaron 20 “nuevos memes” (la mayoría, provenientes de sitios web con gran popularidad actual¹⁵), 20 de las imágenes del conjunto de datos de evaluación y 20 imágenes nuevas de *ImageNet* (“no memes”). Esta distinción se hizo con el fin de

¹⁴ Es decir, un modelo de lenguaje.

¹⁵ <https://www.reddit.com> y <https://me.me>.

mostrar qué tanto aprendió el modelo acerca de los memes y qué tanto generalizó en detalles.

Los promedios de las perplejidades arrojadas por cada experimento de evaluación se muestran en la Tabla 4.3. Observamos que la profundidad de la CNN del **Modelo B** brinda un mejor contexto a la LSTM de los atributos que constituyen a la imagen de entrada, por lo que se justifica la discrepancia entre las perplejidades del **Modelo A** y del **Modelo B**.

Recordemos que la CNN del **Modelo A** fue entrenada bajo la tarea de distinguir el conjunto de datos de memes con un subconjunto de *ImageNet*. Por ello, muchas abstracciones posiblemente identificadas por la CNN del **Modelo B** no logran ser reflejadas en las salidas de la CNN del **Modelo A**. Más aún, como la CNN del **Modelo B** estuvo previamente entrenada con los datos de *ImageNet*, de los cuales surgen los “no memes”, es entendible que dicho modelo haya logrado un desempeño similar con “no memes” y memes de evaluación. En contraste, los nuevos memes constituyen una clase de imágenes que no fue parte tanto del pre-entrenamiento como de la afinación del **Modelo B**.

Los resultados de la evaluación pueden ser consultados de manera interactiva en el **repositorio de código fuente**¹⁶ de esta tesis.

Datos	Perplejidad promedio vs corpus evaluación, usando el Modelo A	Perplejidad promedio vs corpus evaluación, usando el Modelo B
Nuevos Memes	331.14	74.66
Memes Evaluación	241.78	47.19
No Memes	342.68	30.17

Tabla 4.3: Promedios de las perplejidades arrojadas al evaluar los dos *mejores* modelos contra el *corpus* formado mediante las leyendas de los memes extraídos de Internet.

¹⁶ El repositorio se encuentra en <https://github.com/alorozco53/Deep-Meme-Captioner/blob/evaluate/memes/testing.md>.

5

Conclusiones

AS tendencias actuales en cuanto a desarrollo de software compuesto por inteligencia artificial, se inclinan a favorecer arquitecturas que involucran redes neuronales en un rol preponderante. Por ello, esta tesis se desarrolló respetando los métodos que indica el *estado del arte* en cuestión de procesamiento de imágenes y generación de lenguaje natural.

El advenimiento de modelos como *Inception V3* en generación de descripciones de imágenes fue uno de los precursores más importantes para el crecimiento del *aprendizaje profundo*. Como consecuencia inmediata, muchos problemas de inteligencia artificial pasaron a depender fuertemente de la existencia y extracción de grandes cantidades de datos. Por otra parte, es evidente que este proceso es, en sí, un área en pleno desarrollo para todo el que se dedique a las *ciencias de datos*¹. En particular, aquí experimentamos con un

¹ Las *ciencias de datos* son un concepto moderno que engloba los métodos interdisciplinarios que involucran estadística, probabilidad, investigación de operaciones y optimización para el procesamiento de grandes cantidades de información.

conjunto de datos de tamaño pequeño y con una estructura distinta a *ImageNet*.

El estudio de los memes de Internet es un área poco explorada dentro de aprendizaje profundo. A pesar de haber podido reunir un cúmulo de datos clasificados por personaje, en la actualidad este orden ya no se sigue. Haciendo una analogía al concepto propuesto por Richard Dawkins, el meme es una unidad de información que evoluciona y se transforma: hoy en día la popularidad de un meme en una red social se extingue más rápido que hace 4 o 5 años.

5.1. El *flujo* de los memes

El problema de la generación de leyendas para memes se trató estrechamente desde un punto de vista de aprendizaje automático. Si bien existen algoritmos de optimización que incorporan procesos evolutivos de la información, la intención fue explorar la construcción de una arquitectura capaz de *extraer características* y procesarlas de manera numérica. De ahí que se decidió utilizar modelos convolucionales “*convencionales*” sin conocimiento previo al personaje.

Vista como tarea de clasificación de imágenes, la extracción de códigos convolucionales resultó tener éxito tanto con modelos profundos como superficiales. Entonces, ¿qué tan necesaria es una arquitectura de gran profundidad para procesar un conjunto de datos de memes de Internet? Para contestar, de manera general a esta pregunta se requiere continuar con la recolección y procesamiento de datos, incluyendo los memes que gozan de popularidad hoy en día. Tras los experimentos realizados, proponemos que una estructura adecuada involucraría jerarquizar nuevos memes en categorías dependiendo de los personajes involucrados.

La generación de descripciones para imágenes es una tarea que, en aprendizaje profundo, se ha reducido a la optimización de un modelo de lenguaje a partir de representaciones vectoriales de un vocabulario. En este contexto, tratamos con un conjunto de tex-

tos cortos y con un nivel de lenguaje bastante informal: cualquier persona es capaz de etiquetar un personaje de *MemeGenerator* sin importar si el texto hace sentido para los demás *internautas*. Por ello es que es complicado analizar si un meme de Internet es correcto, en comparación con otros *córpores* en inglés. Teniendo esto en mente, se justifica la existencia de un alto porcentaje de leyendas generadas sin sentido pero se destaca la reducción que tuvo la perplejidad al evaluar los modelos (Tabla 4.3) como evidencia de que hubo un aprendizaje.

El modelo recurrente con unidades LSTM es el *estado del arte* actual en generación de lenguaje natural a través de redes neuronales. Debido a que cada personaje posee una personalidad independiente a la mayoría, las leyendas de entrenamiento suelen ser distintas para cada uno de éstos. Con el número de leyendas por personaje usadas para entrenar los modelos de mejor desempeño (5 y 20), resulta totalmente predecible el hecho de no haber podido capturar la “forma de hablar” de algún personaje en particular. En cambio vemos reflejadas algunas frases cortas que exhiben las maneras con las que se lleva a cabo la comunicación a través de Internet. Esto se ve reflejado en resultados como los de la Tabla A.2.

5.2. Trabajo futuro

Todo el seguimiento que se le dio al conjunto de datos de memes fue inspirado en un método de aprendizaje supervisado. En este rubro, se proponen algoritmos completamente dependientes de grandes cantidades de datos para su desempeño; no obstante, es natural preguntarse a partir de qué punto es necesario extraer datos para realizar aprendizaje automático. El entrenamiento que se le da a una red neuronal de cierta profundidad permite que ésta sea utilizada posteriormente como *sistema experto*, ahorrándose así, el gasto de recursos que implica entrenar una red de mayor profundidad.

En el problema de agrupación de datos (*clustering*, en inglés) recae gran parte de la investigación realizada en aprendizaje

no supervisado. En el caso de las imágenes, la confianza adquirida en los modelos (profundos) previamente entrenados con conjuntos de datos como *ImageNet* nos brindan un método que ya ha sido explorado con satisfacción en trabajos como [8]. Aprovechando la “correlación”, que en teoría deberían de tener los códigos convolucionales de dos imágenes similares, se puede cuantificar la cercanía de ambos mediante métricas de distancia² en un espacio vectorial. La agrupación de memes puede traer como consecuencia una mejora en la elaboración de etiquetas, pues previo al entrenamiento de una LSTM, se pueden asociar nuevas características (estados iniciales) al corpus de entrenamiento.

A mediados de 2012, el sitio web 9gag³ subió considerablemente de popularidad como una plataforma de intercambio *humorístico* a partir de memes. Muchos de los personajes engendrados dentro de **MemeGenerator** evolucionaron y “cobraron vida” en 9gag. En particular esto se vio reflejado a partir del formato de memes basado en una historieta corta, como el de la Figura 5.1. En este caso, ya no basta con generar una leyenda que describa a toda la imagen, pues se trata de una sucesión de eventos. Visto de manera formal, se buscaría modelar la interacción entre dos pares (i_1, s_1) , (i_2, s_2) provenientes de un conjunto de datos de entrenamiento $I \times S$ de imágenes asociadas con leyendas. Una opción interesante para este problema consiste en adaptar la arquitectura *Seq2Seq* [20] a la producción de leyendas para cada uno de los cuadros de la historieta.

Seq2Seq es una arquitectura que utiliza dos LSTM’s en situaciones donde hay dos modelos de lenguaje involucrados (posiblemente distintos uno del otro). Como se observa en la Figura 5.2 el rol de una LSTM es **codificar** una secuencia en vectores (parecido a los códigos convolucionales) que sirvan de memoria inicial para la otra LSTM que **decodificará** para producir otra secuencia que siga probabilísticamente a la primera mencionada.

El éxito de este modelo se ha comprobado principalmente en traducción automática de idiomas y, más recientemente, en la

² Por ejemplo, distancia euclíadiana, similitud coseno o distancia Manhattan

³ <https://9gag.com>.



Figura 5.1: Un meme tomado de <https://9gag.com> que ejemplifica el formato de historietas popularizado por dicho sitio web. Algunas variantes de este formato se siguen utilizando en la actualidad.

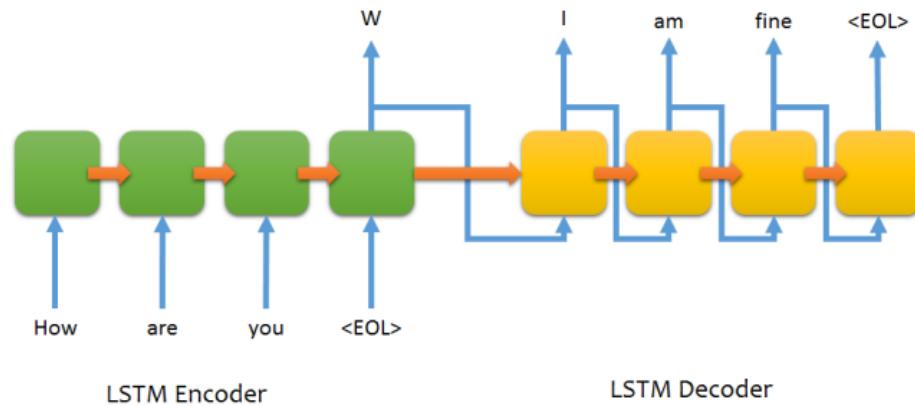


Figura 5.2: Arquitectura *Seq2Seq* aplicada al aprendizaje de modelos de lenguaje para un agente conversacional. Típicamente, se utilizan conjuntos de datos que enseñen al agente a responder preguntas que haría un usuario. (Tomado de <https://github.com/farizrahman4u/seq2seq.git>).

elaboración de agentes conversacionales (*chatbots*). Para incorporar este modelo neuronal en etiquetamiento de historietas de memes, la intuición necesaria consiste en considerar un agente racional que está tratando de etiquetar cada uno de los cuadros a partir del anterior y de los códigos convolucionales extraídos (como memorias iniciales).

El meme es la forma de comunicación de la era actual. Richard Dawkins propuso el concepto y los avances tecnológicos dieron la plataforma para llevarse a cabo. Así como pueden estudiarse los contenidos de un programa de televisión para entender los gustos de la gente, el intercambio de ideas a través de memes permite el desarrollo de estrategias que potencialmente influyan en el comportamiento en una gran masa de población. Por otro lado, el contenido de la información compactada en un meme significa que detrás hay un mecanismo efectivo de compresión de datos que merece ser estudiado tanto en aprendizaje automático como en teoría de la información.

Si bien es cierto que muchos de los resultados anecdóticos de esta tesis no fueron del todo satisfactorios, hay otras maneras de interpretar el conocimiento incrustado en los tensores multidimensionales.

sionales que produce un modelo neuronal. Mediante algoritmos de reducción de dimensiones, combinados con la idea de *clustering* presentada anteriormente, la LSTM se vuelve en un método interesante para analizar la semántica con la que un agente expuesto al Internet comprende el contenido de una red social.

También es importante considerar la volatilidad que existe dentro del *estado del arte* de la inteligencia artificial. En la presente década se dieron los medios para el resurgimiento del aprendizaje profundo, pero ello no significa que hayamos llegado a la *panacea* algorítmica. Más concretamente, existen fuertes críticas hacia las redes convolucionales, las cuales han ido creciendo a pesar de que aún no se tenga un modelo que las supere. Geoffrey Hinton, uno de los *padres* del aprendizaje profundo, ha sido uno de los entusiastas de este movimiento, sobre todo al publicar un nuevo modelo neuronal para el procesamiento de imágenes [18]. Inmediatamente surge la inquietud de si este nuevo modelo puede mejorar el desempeño expuesto en esta tesis y para un conjunto de datos semejante al de memes con el que se trabajó.

Para finalizar, vale la pena recordar el famoso “teorema” del **no almuerzo gratis** (*no-free lunch theorem*) que tanto caracteriza al aprendizaje automático hoy en día: si un modelo M tuvo éxito para una tarea X , no significa que éste va a dar respuesta a otra tarea Y . Hasta ahora, el *boom* del aprendizaje profundo ha traído grandes avances en inteligencia artificial especializada por tareas particulares. Si buscamos extender la comprensión de entes de información, como memes, caemos dentro de la inteligencia artificial general, un campo de estudio muchas veces teórico el cual prioriza el entendimiento y la construcción de máquinas similares al ser humano. ¿Vale la pena estudiar memes? ¿Qué sería de nuestra existencia en el siglo XXI si no fuera por las mentes más curiosas del siglo XX ?

“*Ars longa, vita brevis.*”

— Hipócrates



A

Resultados anecdóticos

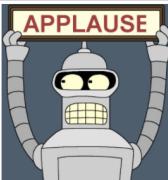
Imagen	Leyenda 1	Leyenda 2	Leyenda 3
	one does not simply have one	one does not simply have a meme	one does not simply have one one
	i got a meme	i have a meme	i am a good

Tabla A.1: Resultados *anecdóticos* del entrenamiento de la LSTM, con previa afinación de *Inception V3*. Como el orden en los tensores de entrenamiento está determinado por cada personaje, ocurrió un sobreajuste durante ciertas épocas contiguas. Esto se muestra mediante la repetición de frases muy características en la elaboración de leyendas. (Fuente: elaboración propia.)

Imagen	Leyenda 1	Leyenda 2	Leyenda 3
	thesis thesis thesis tony banner banner banner	thesis thesis thesis tony banner banner banner	thesis thesis thesis banner tony banner banner
	imperialist imperialist imperialist gates gates attack	imperialist imperialist imperialist gates gates chad	imperialist imperialist imperialist gates attack chad

Tabla A.2: Resultados *aneclóticos* del entrenamiento de la LSTM, con previa afinación de *Inception V3* y reducción del vocabulario. Se puede apreciar la repetición de palabras, como principal característica. (Fuente: elaboración propia.)

Imagen	Leyenda 1	Leyenda 2	Leyenda 3
	trains trains trains trains carefully	oitrains trains trains trains carefully	trains trains trains trains carefully
	kurt virgin born doing bed bed dreams dreams	kurt virgin born doing bed bed chill confused	kurt virgin born doing bed bed dreams dreams

Tabla A.3: Resultados *aneclóticos* del entrenamiento de la LSTM, a partir de una CNN superficial. Al igual que con la red *Inception V3*, hay una considerable repetición de palabras. El desempeño mostrado sugiere pensar que la generalización de las imágenes pertenecientes al conjunto de datos no requiere de un modelo que explore, con gran detalle, sus características más intrínsecas. (Fuente: elaboración propia.)

Imagen	Leyenda 1	Leyenda 2	Leyenda 3
	blacks trade vacation vs pure pure	blacks trade vacation vacation pure pure	blacks trade vacation vs pure magic
	avoid avoid surrender easily easily easily	avoid avoid surrender easily easily easily	avoid avoid avoid surrender easily easily

Tabla A.4: Resultados *anecdóticos* del entrenamiento de la LSTM, a partir de una CNN superficial y 20 leyendas por personaje. Aquí cabe destacar que cuando una palabra sucede a otra, si éstas son distintas, la lógica entre *categorías gramaticales* suele hacer sentido a juicio de un ser humano. Por ejemplo, la aparición de un artículo antes de un sustantivo o verbos a la mitad de la leyenda. (Fuente: elaboración propia.)

Bibliografía

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu y X. Zheng. TensorFlow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. 2015. URL: <http://tensorflow.org/>.
- [2] Y. Bengio. Introduction to multi-layer perceptrons (feedforward neural networks). electrónico. Mayo de 2010. URL: <http://www.iro.umontreal.ca/~pift6266/H10/notes/mlp.html>.
- [3] Y. Bengio. Learning deep architectures for ai. *Foundations and trends in machine learning*, 2009.
- [4] D. Chaffey. Global social media research summary 2016. electrónico. 2016. URL: <http://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/>.
- [5] F. Chollet. Keras. <https://github.com/fchollet/keras>. 2015.
- [6] R. Dawkins. *The selfish gene*. Oxford University Press, New York, NY, USA, 3ra ed., 2006.
- [7] K. Distin. *The selfish meme*. Cambridge University Press, New York, NY, USA, 1ra ed., 2005.
- [8] A. Dundar, J. Jin y E. Culurciello. Convolutional clustering for unsupervised learning. *Corr*, abs/1511.06241, 2015. arXiv: 1511.06241. URL: <http://arxiv.org/abs/1511.06241>.
- [9] I. Goodfellow, Y. Bengio y A. Courville. *Deep learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [10] S. Haykin. *Neural networks and learning machines*. Prentice Hall, New Jersey, USA, 3ra ed., 2009.
- [11] S. Hochreiter y J. Schmidhuber. Long short-term memory. *Theoretical aspects of neural computation (tanc 97)*:1735 -1780, 1997. URL: <http://www.bioinf.jku.at/publications/older/2604.pdf>.

- [12] A. Karpathy. Convolutional neural networks: architectures, convolution / pooling layers. electrónico. Mayo de 2017. URL: <http://cs231n.github.io/convolutional-networks/>.
- [13] Y. LeCun, K. Kavukcuoglu y C. Farabet. Convolutional networks and applications in vision. *Iscas 2010*, 2010. DOI: 10 . 1109 / ISCAS . 2010 . 5537907.
- [14] W. S. McCulloch y W. Pitts. A logical calculus in the ideas immanent in nervous activity. *Bulletin of mathematical biophysics*, 5:115-133, 1943.
- [15] R. Rojas. *Neural networks: a systematic introduction*. Springer-Verlag, Berlin, Deutschland, 1ra ed., 1996.
- [16] S. Ruder. An overview of gradient descent optimization algorithms. *Corr*, abs/1609.04747, 2016. arXiv: 1609 . 04747. URL: <http://arxiv.org/abs/1609.04747>.
- [17] S. J. Russell y P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, New Jersey, USA, 3ra ed., 2010.
- [18] S. Sabour, N. Frosst y G. E Hinton. Dynamic Routing Between Capsules. *Arxiv e-prints*, oct. de 2017. arXiv: 1710 . 09829 [cs . CV].
- [19] L. Shifman. *Memes in digital culture*. MIT Press, Cambridge, MA, USA, 1ra ed., 2014.
- [20] I. Sutskever, O. Vinyals y Q. V. Le. Sequence to sequence learning with neural networks. *Corr*, abs/1409.3215, 2014. arXiv: 1409 . 3215. URL: <http://arxiv.org/abs/1409.3215>.
- [21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens y Z. Wojna. Rethinking the inception architecture for computer vision. *Corr*, abs/1512.00567, 2015. URL: <http://arxiv.org/abs/1512.00567>.
- [22] F. v. Veen. The neural network zoo. electrónico. Sep. de 2016. URL: <http://www.asimovinstitute.org/neural-network-zoo/>.
- [23] O. Vinyals, A. Toshev, S. Bengio y D. Erhan. Show and tell: A neural image caption generator. *Corr*, abs/1411.4555, 2014. URL: <http://arxiv.org/abs/1411.4555>.
- [24] O. Vinyals, A. Toshev, S. Bengio y D. Erhan. Show and tell: lessons learned from the 2015 MSCOCO image captioning challenge. *Corr*, abs/1609.06647, 2016. arXiv: 1609 . 06647. URL: <http://arxiv.org/abs/1609.06647>.
- [25] B. Voytek. Brain metrics: are there really as many neurons in the human brain as stars in the milky way? electrónico. Mayo de 2013. URL: http://www.nature.com/scitable/blog/brain-metrics/are_there_really_as_many.
- [26] J. Yosinski, J. Clune, Y. Bengio y H. Lipson. How transferable are features in deep neural networks? *Corr*, abs/1411.1792, 2014. arXiv: 1411 . 1792. URL: <http://arxiv.org/abs/1411.1792>.