

INSTITUTO TECNOLOGICO AUTONOMO DE MEXICO



FUNDAMENTOS DE DEEP LEARNING Y UNA
APLICACION PARA PREDECIR LA DEMANDA
DE ENERGIA ELECTRICA EN MEXICO

T E S I S

QUE PARA OBTENER EL TITULO DE
LICENCIADO EN MATEMATICAS APLICADAS

PRESENTA

MARIA FERNANDA MORA ALBA

ASESOR: Dr. CARLOS FERNANDO ESPONDA DARLINGTON

CIUDAD DE MEXICO

2016

Autorización

Con fundamento en el artículo 21 y 27 de la Ley Federal del Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada “**FUNDAMENTOS DE DEEP LEARNING Y UNA APLICACION PARA PREDECIR LA DEMANDA DE ENERGIA ELECTRICA EN MEXICO**”, otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Baillères Jr. autorización para que fijen la obra en cualquier medio, incluido el electrónico, y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por la divulgación una contraprestación.

MARIA FERNANDA MORA ALBA

FECHA

FIRMA

Dedicada a

Mi papá, José Mora, por creer en mí y siempre impulsarme a seguir estudiando.

Fundamentos de Deep Learning y una aplicación para predecir la demanda de energía eléctrica en México

María Fernanda Mora Alba

Para obtener los grados de Lic. en Matemáticas Aplicadas y Lic. en Actuaría
Diciembre 2016

Resumen

Deep Learning es un término que se ha vuelto muy popular en los últimos años. Empresas como Google, Apple y Facebook lo mencionan continuamente en entrevistas y notas de prensa. A pesar de esta popularidad, apenas se publicó un libro técnico sobre *Deep Learning* (hace unas semanas), lo cual es reflejo de lo nueva y fértil que es esta área de estudio. En este contexto, el primer gran objetivo de esta tesis es dar una exposición de *Deep Learning*: qué es, cómo se diferencia con las técnicas usuales de Aprendizaje Máquina, sobre qué supuestos descansa, para qué sirve, en qué áreas y tareas ha sido exitoso. Además, se encontró que *Deep Learning* sólo ha tenido impacto en industrias que usan tecnología de punta, y en específico, no ha sido aún muy explorado para modelar series de tiempo. Por otro lado, el problema de la demanda de energía eléctrica en México es relevante, pues dado que la energía no puede ser almacenada, esta se produce “al momento”, lo cual genera pérdidas si hay un exceso de oferta y escasez ante un exceso de demanda. Entonces, como segundo gran objetivo se desarrolla un modelo de *Deep Learning* usando una de sus arquitecturas más populares y exitosas (long short term memory network) con el fin de predecir la demanda de energía eléctrica en México en un esfuerzo para incentivar el enfoque cuantitativo para la toma de decisiones. Se concluye que *Deep Learning* es un paradigma de aprendizaje muy prometedor con potencial de impactar de manera significativa tanto a la industria como al gobierno.

Declaración

El trabajo en esta tesis está basado en investigación que se llevó a cabo en los Departamentos de Estadística, Matemáticas, Computación y Actuaría del Instituto Tecnológico Autónomo de México. Ninguna parte de esta tesis se ha presentado en otro lugar para algún otro grado y es mi propio trabajo a excepción de que se indique lo contrario, lo cual está referenciado en el texto.

Agradecimientos

Este trabajo es simbólico, pues representa el anhelado cierre de una etapa muy larga pero importante en mi formación académica e intelectual.

Gracias a los que hicieron posible este trabajo. Ustedes saben quiénes son.

Índice general

Autorización	II
Resumen	IV
Declaración	V
Agradecimientos	VI
Índice de figuras	IX
1. Introducción	1
1.1. Motivación	1
1.2. Contexto	3
1.3. Aplicaciones	15
1.4. Objetivos y alcance	18
2. Marco teórico	20
2.1. Fundamentos de Aprendizaje Máquina	20
2.1.1. Modelo formal básico del aprendizaje	24
2.1.2. PAC Learning	28
2.1.3. Teorema Fundamental del Aprendizaje	29
2.1.4. Complejidad Computacional de Algoritmos de Aprendizaje	30
2.2. Fundamentos de Deep Learning	32
2.2.1. Maldición de la Dimensionalidad	33
2.2.2. Constancia local y regularización	34
2.2.3. Arquitecturas profundas	35
2.2.4. Entrenamiento de redes profundas	37
2.3. Aplicaciones usuales de Deep Learning	41
2.4. Deep Learning para series de tiempo	47
2.4.1. Trabajo relacionado	48

2.4.2. Series de Tiempo	49
2.4.3. Construcción de una DRNN	51
2.4.4. Long Short Term Memory Networks	62
2.4.5. Combinación de modelos: Deep LSTM	76
3. Aplicación	78
3.1. La energía eléctrica en México	79
3.2. Descripción de la base	80
3.3. Hardware y software	82
3.3.1. Hardware	82
3.3.2. Software	82
3.4. Preprocesamiento y análisis exploratorio	83
3.5. Modelos	89
3.5.1. Sin modelo	90
3.5.2. Modelo tradicional	90
3.5.3. Modelo de Deep Learning	91
4. Resultados	95
4.1. Modelo tradicional	95
4.1.1. Regresión lineal	95
4.1.2. Red neuronal superficial	98
4.2. Modelo de Deep Learning	99
4.2.1. Caso base	102
4.2.2. Dropout(p)	103
4.2.3. Lag	104
4.2.4. Stateful	105
4.2.5. Número de bloques por capa	105
4.2.6. Tamaño del batch	107
4.2.7. Número de épocas	107
4.2.8. Número de capas	108
4.3. Impacto de los resultados	109
5. Conclusiones y trabajo futuro	110
Bibliografía	113
A. Resultados adicionales	120
A.1. Código en Keras para entrenar la LSTM	120

Índice de figuras

1.1.	¿Qué es esto?	2
1.2.	Antecedentes históricos de <i>Deep Learning</i>	4
1.3.	Autocodificador	10
1.4.	Relación de <i>Deep Learning</i> con Aprendizaje Máquina y aprendizaje de atributos.	11
1.5.	Ejemplo de cómo funciona un algoritmo de <i>Deep Learning</i> para reconocer una persona.	12
1.6.	Nivel de aprendizaje en diferentes tipos de sistemas.	14
2.1.	Proceso de Aprendizaje Máquina	22
2.2.	Maldición de la dimensionalidad	33
2.3.	Dos gráficas computacionales con profundidad diferente	35
2.4.	Conceptos simples refinan su significado	36
2.5.	Superficie de error	37
2.6.	Presencia de óptimos locales	38
2.7.	Reconocimiento de imágenes recortadas	42
2.8.	Resultados del modelo de Krizhevsky et al. de 2006	43
2.9.	Screenshots de cinco videojuegos de Atari 2600: Pong, Breakout, Space Invaders, Seaquest, Beam Rider.	44
2.10.	Ciclo de vida usual de una droga y el uso de Deep Learning en R&D.	45
2.11.	Flujo hacia adelante de una red usual.	52
2.12.	Flujo hacia adelante de una red con ciclo.	52
2.13.	Red neuronal recurrente vista como una red usual.	53
2.14.	Arquitecturas de RNN usadas en varias aplicaciones.	54
2.15.	Red neuronal recurrente vista como una red profunda tomada de [47]. .	56
2.16.	RNN agregando profundidad entre input y capa oculta	57
2.17.	RNN agregando profundidad entre capa oculta y output	58
2.18.	RNN agregándole transición profunda	59
2.19.	RNN agregándole transición profunda y atajos para el gradiente	60

2.20. RNN agregándole transición y output profundos	61
2.21. RNN apilada	62
2.22. Modificación de una RNN para hacer una LSTM	64
2.23. Superficie de error para una red neuronal recurrente simple	67
2.24. Celda de memoria c_j y sus puertas in_j , out_j , tomado de [33]	69
2.25. Bloque de memoria con una sola celda de una LSTM tomado de [22] . .	73
2.26. Bloque de memoria desagregado	74
2.27. Vista de Arquitectura parcial horizontal	74
2.28. Una LSTM network tomada de [22]	75
2.29. RNN apilada	76
2.30. LSTM apilada con 3 bloques por nivel	77
3.1. Ejemplo de Archivo de consumo y demanda descargado de [9]	80
3.2. Número total de registros en el archivo consolidado	82
3.3. Oferta y demanda diaria total 2006-2015	84
3.4. Serie de tiempo del consumo de energía eléctrica (sin transformar) . .	85
3.5. Histograma del consumo de energía eléctrica 2006-2015	86
3.6. Serie de tiempo de consumo de energía eléctrica (estacionario)	87
3.7. Serie estacionaria de consumo de energía eléctrica 2006-2015	89
3.8. Serie de tiempo de consumo de energía eléctrica (estacionaria y escalada)	91
4.1. Forma del modelo de regresión lineal múltiple en Weka	96
4.2. Regresores significativos del modelo de regresión lineal múltiple en Weka	97
4.3. Resultados del modelo de Weka	97
4.4. Parámetros de la red neuronal (interfaz de Weka)	98
4.5. Resultados de la red neuronal superficial en Weka	99
4.6. Una LSTM network tomada de [22]	100
4.7.	101
4.8. LSTM apilada	102
4.9. Efecto del tamaño del dataset en el error tomado de [58]	104
4.10. Número de parámetros para 6 bloques	106
4.11. Épocas de la LSTM en Keras	108
4.12. Resultados de la LSTM en Keras	109

Capítulo 1

Introducción

What has been will be again, what
has been done will be done again;
there is nothing new under the sun.

Ecclesiastes 1:9

1.1. Motivación

Hoy en día las computadoras son muy eficientes resolviendo problemas que son difíciles para nosotros los humanos pero que pueden ser descritos por una lista de reglas algorítmicas y/o formales: ordenar arreglos enormes de números, multiplicar matrices de gran tamaño e incluso ganar una partida de ajedrez. Tan sólo basta recordar cómo la computadora Deep Blue derrotó al campeón mundial Garry Kasparov en 1997. Por otro lado, aún no son tan eficientes para resolver tareas que son intuitivas, sencillas e incluso automáticas para nosotros los humanos: caminar, reconocer dígitos, figuras y voces [3].

Tan sólo consideremos la siguiente Figura 1.1 tomada de [62].



Figura 1.1: ¿Qué es esto?

¿Qué podemos encontrar en la figura anterior? Si le hacemos esta pregunta a una persona adulta, o incluso a un niño, es casi seguro que nos dirá que “es una rana”, a pesar de que la rana se halle camuflada. Sin embargo parece que si se le hiciera la misma pregunta a una computadora, no sería tan fácil diseñar un programa que respondiera correctamente. ¿Cuáles son los criterios que nos hicieron decidir que era una rana a pesar de que, en su mayoría, la imagen tiene hojas?

Existe ya progreso en este tipo de tareas, sin embargo éste no ha avanzado a la velocidad deseada. *Deep Learning* es un enfoque de Aprendizaje Máquina que es cada vez más usado para resolver estas tareas sencillas para nosotros pero difíciles para las computadoras.

Hay discusión en si *Deep Learning* es un nuevo paradigma de Aprendizaje Máquina, en el sentido de que escapa al enfoque “usual” de los Algoritmos de Aprendizaje en la forma en que “aprende”. O, por otro lado, en si *Deep Learning* es simplemente un conjunto nuevo de algoritmos de Aprendizaje Máquina. Existen otros grupos más drásticos aún que dicen que *Deep Learning* es únicamente otro tipo de *redes neuronales* e investigadores como Michael I. Jordan de Berkeley que argumentan que las redes neuronales y por lo tanto *Deep Learning* tienen poco qué ver con cómo el cerebro procesa la información, aprende y toma decisiones [20].

El autor de este trabajo opina que estas ideas no se contraponen y *Deep Learning* es las tres cosas: un nuevo paradigma de aprendizaje, un conjunto nuevo de algoritmos de aprendizaje y un nuevo tipo de redes neuronales. Estas ideas quedarán más claras

conforme avance este trabajo. Asimismo, a pesar de que como [20] bien dice, existe progreso a nivel neurona en la Neurociencia pero no tenemos idea clara de cómo funcionan los macroporcesos como la conciencia por ejemplo, eso no implica que no podamos usar el cerebro -que es considerado el paradigma de la inteligencia- como fuente de inspiración, guía o metáfora, o incluso usar el entendimiento que tenemos de éste para crear sistemas y algoritmos inteligentes.

1.2. Contexto

¿Cuándo surge Deep Learning?

Naturalmente, también existe debate respecto a cuándo surgió *Deep Learning*. Algunos dicen que *Deep Learning* es un área nueva con importante potencial y que incluso es relativamente fácil (en comparación de otras áreas) hacer aportaciones nuevas a la disciplina, pues hay problemas relativamente sencillos que no han sido abordados debido a la falta de especialistas en el tema. Otros dicen que no es un área nueva y que simplemente antes no tenía nombre, pero que *Deep Learning* se viene estudiando desde hace 20 años, con las redes neuronales.

Si vemos a *Deep Learning* dentro del contexto de redes neuronales artificiales podemos identificar tres corrientes de investigación según [3]:

- Cibernética (1940-1960): se logra entrenar una sola capa de neuronas.
- Conexiónismo (1980-1995): se logra entrenar una red neuronal con una o dos capas ocultas.
- Deep Learning (2006): se logran entrenar redes neuronales muy profundas, de 4 o más capas.

Lo anterior se puede resumir en la siguiente Figura 1.2, que es una reconstrucción de la figura presentada por [3].

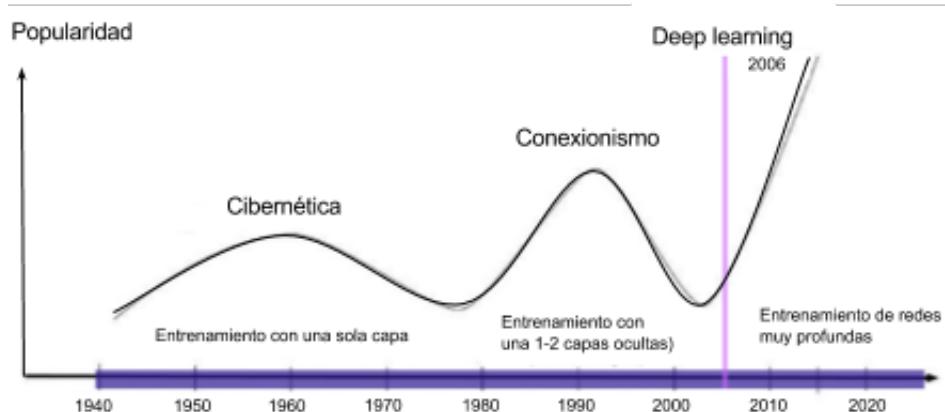


Figura 1.2: Antecedentes históricos de *Deep Learning*

Como se puede observar, los antecedentes de *Deep Learning* han reflejando diferentes enfoques de investigación y en ocasiones han sido populares y en otras no tanto, pero es claro que *Deep Learning* es heredero de las redes neuronales. Por otro lado, es importante mencionar que el trabajo de investigación que logró el resurgimiento de las redes neuronales en 2006 fue el de “A fast learning algorithm for deep belief nets” de Hinton y Osindero [29]. Es trabajo mostró que, en contra de lo que se creía anteriormente, sí era posible entrenar eficientemente un algoritmo de *Deep Learning*.

¿Por qué *Deep Learning* se volvió tan popular?

Ahora bien, resulta interesante también preguntarse porqué se da el resurgimiento de las redes neuronales a través de *Deep Learning*, como puede apreciarse en la Figura 1.2, en donde podemos ver que según [3] la popularidad de *Deep Learning* está muy por encima de la alcanzada por las redes neuronales de los años anteriores. Compañías como Google, IBM, Microsoft, Facebook, Yahoo, Baidu dedican cantidades importantes de recursos para hacer investigación en *Deep Learning* e incluso adquieren compañías más pequeñas que hacen *Deep Learning*. Un caso sonado fue la adquisición de DeepMind Technologies por Google en 2014, ahora llamada Google DeepMind. En una consulta realizada en enero de 2015 la página <https://angel.co/deep-learning-2> se enlistaron 88 startups de *Deep Learning*.

¿Porqué es tan popular *Deep Learning*? [14] da una explicación satisfactoria, mencionando que:

“Tres importantes razones por las cuales *Deep Learning* ha ganado tanta popularidad son el incremento en las habilidades de procesamiento de los GPGPUs (General-Purpose Computing on Graphics Processing Units¹), el incremento sustancial de la cantidad de información y los continuos avances en machine learning y en específico en el procesamiento de información. Estos avances han permitido que los algoritmos de *Deep Learning* puedan exitosamente explotar funciones complejas, compuestas y no-lineales que puedan aprender representaciones jerárquicas y distribuidas de las *rasgos* y hacer uso efectivo tanto de datos etiquetados como no etiquetados”.

En tres palabras: **poder de cómputo, datos masivos y mejores algoritmos**. Los algoritmos de *Deep Learning* son intensivos en la cantidad de datos y recursos computacionales que necesitan para ser entrenados. Adicionalmente, los algoritmos de *Deep Learning* pueden usar aprendizaje en dos niveles: primero se pre-entrena la red y posteriormente se entrena, lo cual aumenta aún más los requerimientos computacionales.

En los capítulos posteriores, en donde se discutirá quedará claro de manera explícita porqué esto es así. Específicamente, en el **Capítulo 2** se dedican secciones para discutir:

1. La cantidad de datos que se requieren para entrenar al algoritmo, la idea es que estos algoritmos tienen muchas capas que deben ser entrenadas, lo cual requiere no sólo cuantiosos recursos computacionales sino también datos.
2. Los órdenes computacionales de los tiempos de ejecución de los algoritmos de *Deep Learning*.
3. Cómo es que los algoritmos de *Deep Learning* pueden realizar aprendizaje en dos niveles: *Aprendizaje de atributos* (para aprender las características relevantes) como *Aprendizaje Máquina* usual (para predecir).

¿Qué es *Deep Learning*?

Hemos discutido el contexto histórico y mencionado de manera vaga el término *Deep Learning*. Ahora bien, ¿Qué es *Deep Learning*? El objetivo de esta sección es dar una definición de alto nivel de *Deep Learning*, sin entrar en detalles técnicos. La exposición matemática se verá a fondo en el **Capítulo 3**.

Existen muchas definiciones diferentes de qué es *Deep Learning*. [14] presenta una

¹Se refiere a el uso de los GPUs para computar cosas que generalmente hace el CPU en vez de usarlos exclusivamente para fines gráficos.

recopilación de varias definiciones que ayudan a introducir una definición de alto nivel.

- **Definición 1:** Es un conjunto de técnicas de Aprendizaje Máquina que explotan muchas capas de información no lineal para la extracción y transformación supervisada o no supervisada, y para el análisis de patrones y clasificación.
- **Definición 2:** Un subcampo de Aprendizaje Máquina que está basado en algoritmos para aprender muchos niveles de representación que permitan modelar relaciones complejas en los datos. Entonces, las características de alto nivel y los conceptos se definen en términos de unos de más bajo nivel. A esa jerarquía de características se le denomina arquitectura profunda. La mayor parte de estos modelos están basados en aprendizaje no supervisado de representaciones. (Wikipedia sobre *Deep Learning*, marzo 2012.)
- **Definición 3:** Un subcampo de Aprendizaje Máquina que está basado en aprender varios niveles de representación los cuales corresponden a una jerarquía de características, factores o conceptos, en donde los conceptos de alto nivel están definidos a partir de conceptos más bajos. *Deep Learning* es parte de una familia más amplia de métodos de Aprendizaje Máquina basados en aprendizaje de *representaciones*. Por ejemplo, una observación (una imagen) puede ser representada de muchas maneras, pero algunas representaciones hacen más fácil el aprender tareas de interés de los ejemplos. En este sentido, la investigación en esta área intenta definir qué es lo que hace una buena representación y cómo pueden ser aprendidas. (Wikipedia sobre *Deep Learning* en febrero 2013).
- **Definición 4:** *Deep Learning* es un conjunto de algoritmos de Aprendizaje Máquina que intentan aprender en muchos niveles de abstracción. Típicamente usa redes neuronales artificiales. Los diferentes niveles en estos modelos estadísticos aprendidos corresponden a distintos niveles de conceptos, en donde los conceptos de más alto nivel se definen a partir de niveles más bajos, y los mismos conceptos de bajo nivel pueden usarse para definir muchos otros conceptos de más alto nivel. Wikipedia http://en.wikipedia.org/wiki/Deep_learning sobre *Deep Learning* en octubre 2013.
- **Definición 5:** *Deep Learning* es una nueva área de investigación en Aprendizaje Máquina, que ha sido introducida con el objetivo de posicionar Aprendizaje Máquina cerca a sus metas iniciales: Inteligencia Artificial. *Deep Learning* comprende aprender múltiples niveles de representación y abstracción que ayuden a hacer sentido de los datos como imágenes, sonido y texto. <https://github.com/lisa-lab/DeepLearningTutorials>

Adicionalmente añadimos la siguiente definición que fue consultada en Enero 2016 de Wikipedia:

- **Definición 6:** *Deep Learning* es una rama de Aprendizaje Máquina basada en un conjunto de algoritmos que intentan modelar abstracciones de alto nivel en los datos usando múltiples capas de procesamiento con estructuras complejas, o bien compuestas de múltiples transformaciones no lineales. Es parte de una familia más amplia de métodos de Aprendizaje Máquina basados en aprendizaje de representaciones de los datos. Por ejemplo, una imagen puede ser representada de muchas maneras: como un vector de intensidades por pixel, o como un conjunto de regiones con una forma o curvatura particular. Algunas representaciones hacen más fácil el aprender tareas como el reconocimiento de rostros o de voces. (Wikipedia sobre *Deep Learning* en enero 2016).

Como podemos ver, la definición de *Deep Learning* ha ido refinándose con el tiempo; sin embargo todas las definiciones antes presentadas tienen los siguientes elementos en común:

- Modelos con múltiples capas o etapas de procesamiento de información no lineal.
- Métodos para el aprendizaje supervisado o no supervisado para la representación de atributos usando capas sucesivas y más abstractas.

Además, todas las definiciones notaban que *Deep Learning* está embebido en *Aprendizaje Máquina*. Entonces, de manera inmediata surge la siguiente pregunta, que será abordada intuitivamente en este capítulo y en el **Capítulo 2** se responderá con toda formalidad:

¿Qué es el Aprendizaje Máquina?

Las personas diariamente adquieren inmensas cantidades de información que traducen en conocimiento sobre el mundo. Dicho conocimiento es intuitivo, subjetivo y por lo mismo es difícil de enunciar en términos de reglas formales. Por otro lado, las computadoras deben capturar y replicar de manera natural este conocimiento si quieren simular la inteligencia humana. Uno de los retos más importantes en Inteligencia Artificial es cómo hacer que una computadora adquiera este tipo de conocimiento sobre el mundo [3] y para ello tiene dos enfoques: basado en el *conocimiento*, y basado en la *experiencia*.

El enfoque basado en conocimiento intenta traducir el conocimiento sobre el mundo en términos de lenguajes formales que le permiten razonar sobre enunciados usando reglas de inferencia lógica. Si bien algunos proyectos que usan este enfoque han tenido algunos éxitos, ninguno ha sido concluyente.

El segundo enfoque, basado en la experiencia, permite que las computadoras adquieran su propio conocimiento. Aprendizaje Máquina tiene que ver con cómo una máquina usa el segundo enfoque, es decir, cómo una máquina *aprende*. ¿Qué significa que *aprenda*? Intuitivamente, significa que las máquinas sean capaces de identificar patrones en los datos, pero no sólo eso, sino que el conocimiento derivado de detectar dichos patrones mejore conforme se va adquiriendo más experiencia. Al final esto se traduce en mejores predicciones. Formalmente la respuesta a *¿qué es un algoritmo de aprendizaje?* no es trivial y merece su propio capítulo, el segundo.

Un ejemplo de un algoritmo de Aprendizaje Máquina aplicado a riesgo de crédito es la regresión logística para *predecir* el tipo de cliente: bueno o malo. El usuario *construye* un perfil del cliente: sexo, sueldo, buró de crédito, estado civil, número de hijos, escolaridad, otras deudas contraídas, etc. Supongamos que se cuenta con un historial de muchos clientes con esta información capturada y además sabe quiénes de esos clientes fueron “buenos” o “malos”; entonces separan en conjunto de entrenamiento y prueba y ajusta un modelo de regresión logística en el conjunto de entrenamiento para predecir el tipo de cliente: bueno o malo. En medio de este proceso puede haber muchos detalles técnicos: remuestreo, balanceo de muestras, etc, pero los omitiremos por no ser relevantes para lo que queremos ilustrar. Lo importante es el modelo de regresión logística parametrizado que se convierte en un *modelo de predicción* del tipo de cliente. Entonces, cuando un cliente nuevo llega, se llena su perfil de cliente y basándose en el modelo logístico ve si lo califica como bueno o como malo. Si se clasifica como bueno, entonces se le da el crédito, y si no, no se le da. Además, como es un Algoritmo de Aprendizaje, si se le dan nuevos clientes con sus respectivos perfiles, entonces puede incorporar esa nueva información y mejorar las predicciones del modelo.

El ejemplo anterior fue ilustrativo, pero *¿en dónde queda Deep Learning?* Recordar que dijimos que el *usuario* tiene previamente diseñado un perfil del cliente. Pero, *¿Cómo supo el cliente que esas características: sexo, sueldo, buró de crédito, estado civil, otras deudas contraídas son las que determinan si el cliente es bueno o malo?* Este perfil del cliente es lo que se conoce en *Deep Learning* como *representación* y cada una de las características del cliente es un *atributo*. Los *atributos* pueden verse como los factores de influencia de los datos. Si en vez de darle los *atributos* a la regresión logística le diéramos otra información del cliente, como su acta de nacimiento, el reporte de buró de crédito completo, etc, posiblemente la regresión logística no haría un buen trabajo.

En general, gran cantidad de problemas pueden ser resueltos satisfactoriamente por algoritmos de Aprendizaje Máquina simples siempre y cuando se les provea de buenas representaciones. Esto los hace depender *fuertemente* de las *representaciones* y *atributos* que se les den y sus desempeños pueden variar enormemente. No obstante, en muchos problemas no es fácil determinar cuáles son los *atributos* relevantes. Considerar el problema de determinar si hay una cara en una imagen. *¿Cuáles son los atributos?*

Para dar una propuesta quizá serviría responder antes ¿qué es una cara? Sí, las caras tienen ojos, nariz y boca, pero ¿qué pasa si la cara está de perfil? Sólo se mira la nariz, la boca de perfil y en vez de ojos, las pestañas. Adicionalmente las narices son muy variadas, las hay aguileñas, respingadas, chatas, y no se diga de los ojos. Entonces, nada más determinar si hay ojos, boca y nariz, es decir, determinar los *atributos* es una tarea compleja.

Dado que hay muchas tareas de este tipo, en donde es muy difícil determinar qué tipo de *atributos* son los importantes o es difícil medirlos, un camino es utilizar *Aprendizaje Máquina* dos veces: primero para encontrar los atributos y después para predecir la existencia o no de una cara usando estos *atributos* encontradas.

Para la primera parte surge algo que se conoce como *aprendizaje de representaciones* (representation learning por su nombre en inglés).

¿Qué es el aprendizaje de atributos?

Como vimos, determinar manualmente los *atributos* puede ser complicado y muy tardado, pueden pasar incluso décadas para que una comunidad de expertos acuerden ciertos *atributos*. Por otro lado, un algoritmo de *Representation Learning* puede tomarse minutos, días o meses para descifrarlos [3].

El *autocodificador* es el ejemplo típico de *aprendizaje de representaciones* en donde un *codificador* recibe un input, lo codifica en una nueva representación y luego un *decodificador* lo decodifica y lo transforma en un output (ver Figura 1.3). El objetivo es encontrar la codificación tal que la diferencia entre el input y el output sea la menor. La codificación, que es la *nueva representación* debe intentar separar los factores de variación o de influencia que mencionamos anteriormente: ¿cuáles son las variables que hacen que observemos lo que estamos observando y no otra cosa? Es importante mencionar que muchas veces estos factores no son observables directamente; más aún, quizás estos factores son abstracciones y pueden no tener un significado intuitivo.

Adicionalmente el *aprendizaje de atributos* puede tener algunas propiedades deseables, como simplicidad de procesamiento o entendimiento: el codificador nos da una representación que es más fácil de procesar o entender.

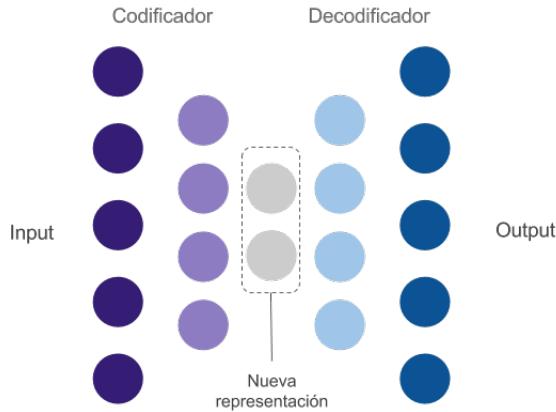


Figura 1.3: Autocodificador

Otro ejemplo de este tipo de aprendizaje es el bien conocido *análisis de componentes principales* con el que casi todos estamos familiarizados y por lo mismo no se ahondará en este tema.

En muchas aplicaciones la dificultad surge cuando muchos factores de variación influencian a otros factores de influencia y es difícil saber cuál es la propiedad *verdadera* del objeto con respecto a cada factor de manera individual. Por ejemplo, supongamos que tenemos que tenemos los factores de influencia color y hora del día para varios coches. Pero sucede que en la noche un coche azul marino puede confundirse con uno negro, o en el día un coche plata puede confundirse con uno blanco [3].

Para terminar esta sección vale la pena observar la siguiente Figura 1.4 que nos muestra que *Deep Learning* como área de estudio es subconjunto de Aprendizaje Máquina y de aprendizaje de atributos.

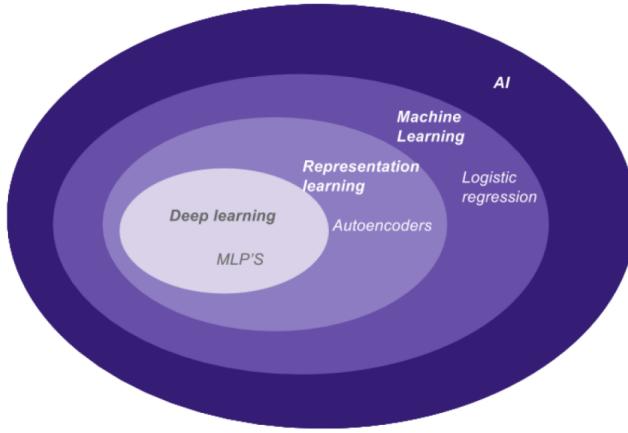


Figura 1.4: Relación de *Deep Learning* con Aprendizaje Máquina y aprendizaje de atributos.

¿Cuál es la ventaja de usar algoritmos de *Deep Learning*?

La tarea es entonces *desenredar* esos factores de variación, encontrar los de más alto nivel y desechar los que no nos sirven o nos hacen ruido. Esta tarea no es fácil y en muchas aplicaciones es tan difícil encontrar estos factores como resolver el problema original (clasificación, regresión, etc). Es aquí cuando entra *Deep Learning*, pues introduce un tipo especial de representaciones, las cuales están expresadas en términos de otras más simples, es decir, de manera jerárquica. Esto permite construir conceptos complejos a partir de conceptos simples.

La siguiente Figura 2.4 tomada de [3] muestra cómo un algoritmo de *Deep Learning* construiría el concepto de persona usando conceptos más simples: primero al algoritmo le entran los pixeles (input). Los pixeles acumulados permiten distinguir bordes (1era capa oculta), lo cual se pasa a la 2da capa oculta que permite identificar a partir de los bordes, esquinas y contornos. La 3er capa ya es capaz de distinguir objetos individuales como orejas, boca y ojos. Finalmente en la capa de salida se identifica al objeto usando toda la información previa, que en este caso es una persona.

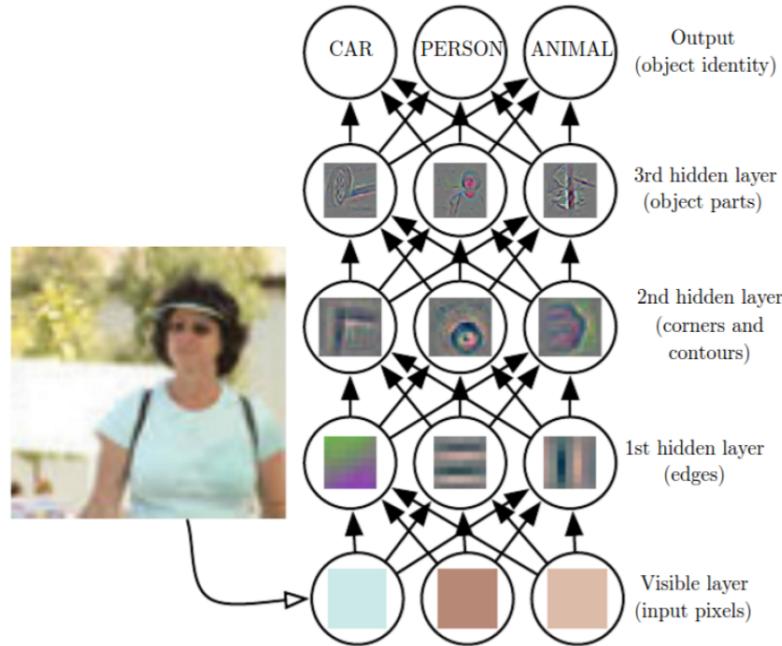


Figura 1.5: Ejemplo de cómo funciona un algoritmo de *Deep Learning* para reconocer una persona.

Es importante señalar dos cosas. Primero, que en medio de las capas que mencionamos existen otras muchas capas, las cuales no se muestran en la Figura 2.4. Esas capas intermedias son las que permiten hacer aprendizaje de atributos y a este proceso se le llama *preentrenar* al algoritmo. En este ejemplo, los atributos hallados en la fase de pre-entrenamiento fueron los bordes, las esquinas y contornos y las partes individuales. Finalmente observar que cada uno de estos atributos está definido en términos de otros atributos de manera jerárquica. Esta jerarquía de conceptos o atributos es crucial en *Deep Learning*.

Segundo, la Figura 2.4 sólo ilustra un tipo de modelo de *Deep Learning*, en específico, una *Red de creencia profunda* (Deep Belief Network). Existen numerosos modelos de *Deep Learning*, así como existen muchos tipos de redes neuronales. Cada uno tiene una arquitectura y propiedades que le permiten modelar cierto tipo de fenómenos. En el **Capítulo 2** hacemos una recopilación de los modelos más importantes de *Deep Learning*, explicamos su arquitectura y mencionamos para qué tipo de tareas generalmente se usa.

Un algoritmo de *Deep Learning* tiene al menos tres ventajas con respecto a otros algoritmos de Aprendizaje Máquina.

La primera es la capacidad de realizar **aprendizaje de atributos**. El tener una fase de pre-entrenamiento para el aprendizaje de atributos permite que no sea necesario que un experto le indique al algoritmo cuáles son los atributos importantes, lo cual no ocurre en los algoritmos usuales de Aprendizaje Máquina (excepto en redes neuronales superficiales). De hecho, como dice [57], una de las promesas de *Deep Learning* es reemplazar la selección manual de atributos con algoritmos eficientes para aprendizaje de atributos no supervisado o semi-supervisado. Posterior a la etapa de pre-entrenamiento se realiza la etapa de entrenamiento supervisado como en cualquier algoritmo usual de Aprendizaje Máquina. En este sentido podríamos decir entonces que un algoritmo de *Deep Learning* “aprende más” que un algoritmo usual de Aprendizaje Máquina. Es importante mencionar que lo anterior se refiere al caso general, pues en algunos casos se usó ReLU en las neuronas (Rectifier Linear Unit) en vez de una función sigmoide o tangente inversa sin necesidad de preentrenamiento [41].

La segunda es la capacidad de modelar fenómenos usando una **jerarquía de conceptos**. En dicha jerarquía cada concepto está definido en términos de su relación con conceptos más simples, lo que permite que el algoritmo pueda aprender conceptos complejos construyéndolos a partir de los conceptos más simples que lo forman [3].

La tercera es la capacidad de estudiar fenómenos que involucran una gran cantidad de **funciones o conceptos complejos**, por ejemplo el reconocimiento de imágenes en donde algoritmos de *Deep Learning* han sido aplicados con éxito desde el 2000, en donde usando Aprendizaje Máquina tradicional no se habían obtenido resultados tan fructíferos [39].

La Figura 1.6 muestra el nivel de aprendizaje que un algoritmo puede tener. Primero tenemos los algoritmos ordinarios basados en reglas, los cuales no tienen ningún nivel de aprendizaje. Posteriormente seguimos con los algoritmos de Aprendizaje Máquina usuales que toman un conjunto de atributos dados y realizan aprendizaje con ellos (clasificación, regresión, etc.). Luego los algoritmos de aprendizaje de representaciones que realizan aprendizaje para los atributos y luego aprendizaje sobre los atributos encontrados. *Deep Learning* añade otro nivel de aprendizaje para obtener atributos complejos a partir del aprendizaje de atributos simples, y finalmente realiza aprendizaje sobre los complejos.

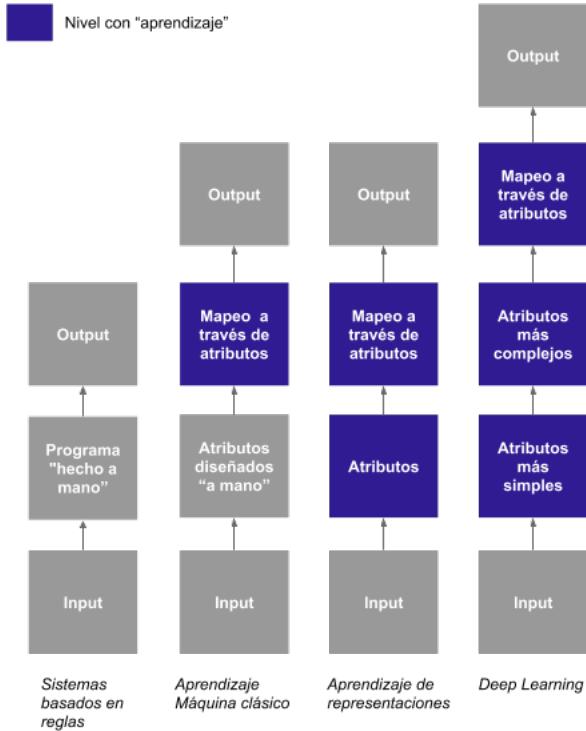


Figura 1.6: Nivel de aprendizaje en diferentes tipos de sistemas.

Superficial vs profundo

Como dice [14], la mayoría de los algoritmos de Aprendizaje Máquina consisten de una *arquitectura superficial*, en donde *superficial* no debe ser entendido como un adjetivo peyorativo. Se les denomina *superficiales* porque consisten en a lo más una o dos capas de transformaciones no lineales de los atributos. Ejemplos usuales de estos algoritmos son los Modelos de Mezcla Gaussianos (GMMs o Gaussian Mixture Models por sus siglas en inglés), máquinas de soporte vectorial (SVMs), regresiones logísticas, perceptrones multicapa (MLPs).

Estas arquitecturas superficiales han probado ser muy efectivas resolviendo cierto tipo de problemas, pero constan de poder de representación limitado y no han sido muy efectivos procesando y modelando fenómenos como el habla humana, los lenguajes naturales e imágenes [14].

La visión, la audición y el habla en los humanos sugieren la necesidad de usar arquitecturas profundas que sean capaces de extraer estructura compleja y de construir una representación interna rica en estímulos externos. [14] pone como ejemplo la producción del habla en el humano, que dice está equipada con capas de estructuras jerárquicas que transforman la información del nivel en forma de onda al nivel lingüístico. También pone como ejemplo la visión, en donde enfatiza que la percepción es jerárquica. Adicionalmente, en el cerebro se tienen de 6 a 8 capas para procesar información compleja como imágenes. Estas similitudes sugieren que si es posible desarrollar algoritmos de *Deep Learning* que sean efectivos y eficientes entonces quizás es posible lograr avances procesando este tipo de señales. Surge naturalmente la siguiente pregunta: ¿Qué tan difícil es entrenar estos modelos de manera efectiva y eficiente?

Algunas dificultades

Remontándonos a los inicios de las redes neuronales en los 70's, uno de los principales problemas que surgieron fue justamente cómo entrenar dichas redes. La respuesta surgió en los 80's con el algoritmo de *retropropagación* o propagación hacia atrás de los errores (o back-propagation en inglés) y desde entonces es un algoritmo bien conocido para entrenar las redes neuronales. Sin embargo, el algoritmo de retropropagación encontró un pobre desempeño al tratar de entrenar una red con un número no muy grande de capas ocultas usando la inicialización aleatoria usual que el algoritmo requiere [19]. En el **Capítulo 2** se hará una exposición formal de porqué sucede lo anterior y cómo se corrige el problema, pero por ahora basta mencionar que la presencia de óptimos locales (la función a optimizar es no-convexa) constituyen los principales retos al entrenar una red neuronal profunda.

1.3. Aplicaciones

A pesar de los retos que implican entrenar un algoritmo de *Deep Learning*, se han logrado resultados muy importantes en las siguientes tareas:

- Reconocimiento de imágenes
- Reconocimiento de voz
- Reconocimiento de texto
- Procesamiento de lenguaje natural

- Sistemas de Recomendación
- Descubrimiento de nuevas drogas
- Customer Relationship Management (CRM)
- Bioinformática

En el **Capítulo 2** se hace un recuento de algunas de estas aplicaciones.

Es notable que en la lista anterior aún no aparezca una aplicación destacada que esté relacionada con Finanzas, Economía, Biología, etc. De hecho las últimas tres aplicaciones son meramente experimentales, mientras que las primeras cinco han obtenido un éxito consistente y de hecho ya han permeado a la industria. Por ejemplo, el reconocedor de caras de Facebook, el traductor de Skype, el asistente Google Now de Google, etc., son implementaciones comerciales de tecnologías de *Deep Learning*. Parece ser que las industrias y empresas que usan y comercializan tecnología de punta son las que usan *Deep Learning*.

¿Por qué esto es así? Alguien podría argumentar que *Deep Learning* sirve en unas industrias y para cierto tipo de aplicaciones pero no para otras. Sin embargo esto resulta falso cuando nos preguntamos: ¿no es de interés para cualquier empresa de cualquier industria tener una aplicación de Customer Relationship Management automatizada? ¿para la industria farmacéutica no es relevante minimizar la creación de drogas que no se comercializarán? ¿no es de interés para cualquier empresa que venda productos tener un sistema de recomendación?

Pensemos en una aseguradora y sus clientes, los cuales tienen diferente perfil y por lo tanto requieren de diferentes productos: no a todos les interesaría un seguro de auto de cobertura amplia ni un seguro de vida. Muchas veces la tarea de buscar qué producto ofrecerle a cada cliente se hacen de manera manual: un agente ubica a un cliente potencial, hace una cita personal con él, luego tiene que obtener su perfil y basado “en su experiencia”, le recomienda el producto al cliente. Normalmente el cliente ni siquiera alcanza a conocer todas las opciones de producto que existen y podrían interesarle. Un sistema de recomendación podría ayudarle al agente a estrechar su espacio de búsqueda de productos de modo que pueda ofrecer un producto a la medida. El mismo ejemplo puede ser un banco y sus diferentes tarjetas y productos.

Un uso reciente de *Deep Learning* con un impacto potencial incommensurable es para el modelado de series de tiempo. Las series de tiempo son estructuras de datos que abundan y cuyo uso e importancia es generalizado e innegable, tanto en la academia como en la industria. Aparecen a la orden del día en los bancos, las casas de bolsa, aseguradoras, laboratorios, etc. para modelar el PIB, el precio de acciones, las expon-

taciones de un país, ventas de un producto, decaimiento de una molécula, demanda de bienes y servicios, etc.

En la mayoría de los casos estas series temporales son modeladas con métodos tradicionales de estadística (modelos autorregresivos). El problema de estos métodos clásicos son varios: hacen importantes supuestos sobre los datos y cuando las dinámicas son muy complejas, de altas dimensiones y con mucho ruido, no es posible describir los datos usando ecuaciones con parámetros a estimar, de modo que las predicciones serán pobres [36]. Otro enfoque más reciente es el uso de modelos de minería de datos, pero tienen la desventaja de escalar o predecir pobremente.

Por otro lado, los algoritmos superficiales, al contener un número pequeño de transformaciones no lineales, tampoco han sido capaces de modelar adecuada y consistentemente el comportamiento complejo de las series temporales [36].

El modelado de series temporales usando *Deep Learning* ha obtenido buenos resultados debido a que la jerarquía de una arquitectura profunda logra imitar la jerarquía inherente en una serie temporal (la del tiempo). En el **Capítulo 2** discutiremos cuáles son los retos que aparecen al intentar modelar una serie temporal y explicaremos con detalle matemático cómo es que una arquitectura de *Deep Learning* lograr modelar series de tiempo exitosamente.

Dada la necesidad de usar modelos más novedosos para modelar series de tiempo se pensó en aplicarlos a un problema relevante para México y se encontró que la predicción del consumo de energía eléctrica es de importancia notable. ¿Por qué? Porque primero, la energía eléctrica resulta una pieza clave en la motorización del país. Segundo, la industria eléctrica es un tema en boga debido al debate a favor o en contra de la privatización de la energía eléctrica. Tercero, la energía eléctrica no puede ser almacenada a grandes escalas, lo cual significa que debe ser generada a una tasa casi igual a la que es consumida para evitar desperdicios. Tan sólo en un período de 10 años, del 1 de enero del 2006 al 1 de enero del 2015, se previó un consumo total de 146, 265, 254 MW de energía eléctrica, pero el consumo real fue de 139, 907, 400 MW, dando una pérdida de 6, 357, 854 MW. La pérdida máxima reportada se dio el 1ero de abril del 2007, en donde se produjo un 36 % más de energía de lo realmente consumido, representando una pérdida de 14, 565 MW sólo en ese día. Por otro lado, una demanda mayor a la oferta también tiene consecuencias desafortunadas, pues el diferencial puede causar que el equipo de generación y transmisión automáticamente se apaguen para prevenir daños, pudiendo en el peor de los casos terminar en apagones. Por ejemplo, tan sólo en enero del 2009 7 días del mes la demanda fue más alta que la oferta, y el 1ero de febrero del 2009 se proyectó un consumo 3.46 % menor a lo consumido (el máximo histórico), generando una escasez de 1, 140 MW tan sólo en un día. El mecanismo que la CFE usa para llevar a cabo sus predicciones no se encontró publicado en ningún lado y tampoco se encontró documentación en la página oficial de la entidad.

Entonces, contar con un sistema preciso de predicción de la demanda de energía eléctrica es necesario para asegurarse que la generación de energía eléctrica se asemeje lo más posible a la demanda.

1.4. Objetivos y alcance

Las contribuciones de esta tesis fueron pensadas para subsanar algunas de las necesidades que se desprendieron al estudiar tanto el panorama general de *Deep Learning* como el estado de la energía eléctrica en México. Estos panoramas arrojan las siguientes conclusiones:

- Las arquitecturas de *Deep Learning* aprenden en más niveles que las arquitecturas usuales y modelan usando una jerarquía anidada de conceptos; esto permite crear modelos flexibles y poderosos, lo cual le ha valido un éxito sin precedentes en problemas que históricamente se han considerado muy complejos.
- A la fecha sólo existe un libro técnico completo publicado que unifique el trabajo hecho sobre *Deep Learning*, el cual fue publicado oficialmente apenas hace unas semanas (ver [21]); la gran cantidad de recursos disponibles del tema son artículos académicos, blogs y tutoriales y por ende el conocimiento que se tiene sobre *Deep Learning* se halla desbalanceado y fragmentado.
- *Deep Learning* sólo ha logrado permear en las industrias más innovadoras a pesar de que sus aplicaciones son del interés de cualquier industria.
- Las series temporales son estructuras de datos que abundan en las ciencias y en la industria; asimismo presentan retos para ser modeladas debido a su inherente complejidad estructural.
- *Deep Learning* ha empezado a ser usado con éxito para modelar series temporales, sin embargo las aplicaciones aún son escasas y no han tenido un impacto en la industria.
- En una era en la que la producción de combustibles fósiles va en descenso, la energía eléctrica se vuelve aún más importante.
- La predicción de la demanda de energía en México es un problema relevante: se generan pérdidas anuales considerables, y asimismo se desconocen los mecanismos de la Comisión Federal de Electricidad para llevar a cabo las predicciones actuales.
- Contar con un modelo de predicción de la demanda de energía eléctrica en México

podría ayudar a planificar la producción de energía y disminuir las pérdidas por exceso de oferta y el desabasto por exceso de demanda.

En este sentido, las contribuciones de este trabajo se pueden resumir en dos grandes objetivos:

- Dar un visión unificada, robusta, intuitiva y técnica de qué es *Deep Learning*: cómo surgió, qué es, cómo funciona, para qué sirve, cuáles son sus dificultades y sus éxitos.
- Presentar una aplicación de *Deep Learning* que no haya sido muy explorada aún (series temporales) pero con un impacto social y económico importante en México (demanda de energía).

Lo anterior permitirá un beneficio mutuo tanto para el gobierno-industria, como para el avance de *Deep Learning* como área científica.

Entonces, el alcance de este trabajo puede resumirse en: por un lado se desea abordar el problema de la demanda de energía en México desde una perspectiva cuantitativa; y que esta perspectiva cuantitativa incentive aplicaciones de *Deep Learning* que no han sido muy exploradas aún.

Capítulo 2

Marco teórico

People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world.

Pedro Domingos

2.1. Fundamentos de Aprendizaje Máquina

Según [55], dos criterios nos ayudan a determinar si un algoritmo de *Aprendizaje Máquina* es pertinente:

Complejidad: cuando queremos reproducir rutinas llevadas a cabo por personas como manejar, caminar, reconocimiento de voz y procesamiento de imágenes que son de difícil mimética por una máquina. O cuando queremos encontrar patrones en grandes cantidades de datos (astronómicos, genómicos, comercio electrónico, redes sociales), lo cual sería muy complicado de realizar por un humano.

Adaptividad: cuando necesitamos programas flexibles que se adapten a nuevos datos. Por ejemplo, programas que reconocen texto escrito a mano/spam o programas de reconocimiento de voz.

Tomando en cuenta lo anterior, los algoritmos de aprendizaje se han usado exitosamente en problemas como los que menciona [45]:

- Detección de spam
- Procesamiento de lenguaje natural
- Reconocimiento de voz y de imágenes
- Biología computacional
- Detección de fraude
- Diagnósticos médico
- Sistemas de recomendación
- etc., etc.

Como dice [45] las aplicaciones anteriores se pueden clasificar en grandes tipos de problemas de aprendizaje:

- **Clasificación:** cada individuo tiene una categoría asignada, por ejemplo, si queremos clasificar el tipo de cliente (bueno/malo); otro ejemplo es si queremos ver qué tan avanzada está el cáncer en un paciente (estadío 0, I, II, III, IV). Generalmente el número de clases es pequeño. El objetivo es predecir a qué clase pertenece un individuo dado.
- **Regresión:** el objetivo es predecir un número real para un individuo. Dicha variable real guarda una relación funcional con otras variables dadas. Ejemplos típicos son predecir el ingreso medio de una familia mexicana, los niveles de glucosa de un enfermo de diabetes, etc. Otro ejemplo son las series temporales, en donde también la variable a predecir es una variable real que guarda una dependencia temporal con ella misma.
- **Ranking:** ordenar individuos según algún criterio. Un ejemplo típico es regresar las páginas que son relevantes dada una *query* de búsqueda. Este tipo generalmente surge para diseñar algoritmos extracción de información y de procesamiento de lenguaje natural.
- **Clustering:** estos algoritmos generan conglomerados dentro de los cuales los individuos son *similares* y fuera de ellos no. Por ejemplo, en análisis de redes sociales se aplican estos algoritmos para encontrar comunidades de usuarios con intereses en común. También en investigación de mercados se buscan conglomerados de

clientes que se parezcan, se estudian las preferencias de dichos conglomerados y posteriormente se dirigen campañas publicitarias a los mismos.

- **Reducción de dimensionalidad:** dada una representación mediante variables de un individuo, se busca poder representarlo usando menos variables preservando la mayor cantidad de información sobre los individuos. En ocasiones esta técnica se usa para obtener índices. Un ejemplo son los índices de marginación que publica el CONAPO (Consejo Nacional de Población) en [12].

¿Qué significa que un algoritmo aprenda?

Grosso modo podemos decir que el *aprendizaje* es el proceso que convierte experiencia en *conocimiento*. El *Aprendizaje Máquina* a su vez es un proceso mediante el cual la computadora incorpora datos sobre un fenómeno y los convierte en experiencia; posteriormente esta experiencia se convierte en un modelo sobre el fenómeno, que a su vez genera conocimiento sobre el mismo. El proceso se puede visualizar mediante la siguiente Figura 2.1.

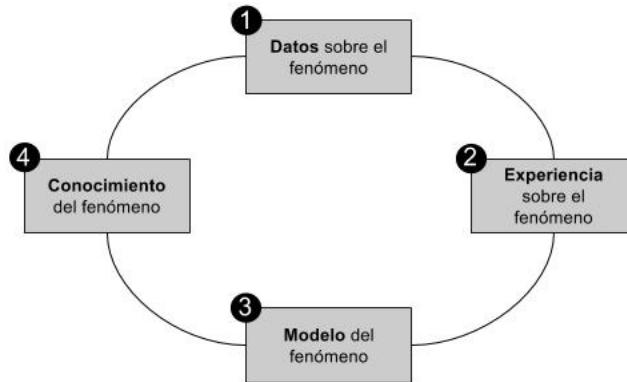


Figura 2.1: Proceso de Aprendizaje Máquina

La Figura 2.1, sin embargo, parece dar por sentado que es posible *aprender* el fenómeno y que sólo es cuestión de seguir los pasos del Proceso de Aprendizaje. No obstante, como explicaremos más adelante no es siempre así y el que algo pueda ser aprendido o no por una máquina se puede plantear y responder de manera matemática. Esto es lo que se conoce como la *Teoría del Aprendizaje Estadístico*. La idea central es la que enuncia Mitchell en su libro: “Se dice que un programa aprende de la experiencia E

con respecto a una clase de tareas T y medida de desempeño P , si su desempeño sobre las tareas en T , medidas por P , aumenta con la experiencia E' [43].

Aprendizaje Máquina es un área de estudio que se interseca de manera no vacía con muchas otras áreas, -no únicamente con Inteligencia Artificial-, como por ejemplo con Aprendizaje Estadístico, Ciencias de la Computación, Optimización, entre otras.

Además de la pregunta ¿qué fenómenos pueden aprenderse?, resulta que existen diferentes *paradigmas de aprendizaje*. No es lo mismo *aprender* a predecir clientes buenos y malos (clasificación), que *aprender* a colocar en clústers a un conjunto de individuos (clustering).

A continuación se presenta una explicación conceptual pero no exhaustiva de los diferentes *Paradigmas de Aprendizaje* según [45].

Paradigmas de aprendizaje

- *Aprendizaje supervisado*: la máquina (algoritmo) recibe una serie de ejemplos etiquetados que fungirán como los datos de entrenamiento. El objetivo es hacer predicciones para datos no vistos. Generalmente los problemas de regresión, clasificación y ranking entran en este paradigma, por ejemplo clasificación u ordenamiento de cliente, detección de spam ó predicción de series temporales.
- *Aprendizaje no supervisado*: la máquina (algoritmo) recibe datos de entrenamiento no etiquetados y hace predicciones para datos no vistos encontrando “patrones” en los datos. Dado que no hay datos etiquetados con los cuáles poder evaluar el desempeño de los modelos entonces la evaluación en este tipo de métodos es particularmente difícil. La mayoría de los conjuntos de datos naturalmente no están etiquetados, pues etiquetar es costoso y tardado. Clustering (jerárquico y no jerárquico) y reducción de dimensionalidad (componentes principales, tSNE) son ejemplos de métodos de aprendizaje no supervisado.
- *Aprendizaje semi-supervisado*: la máquina (algoritmo) cuenta con datos etiquetados y no etiquetados. La motivación de este paradigma es que en condiciones reales se cuenta con muy pocos datos etiquetados porque son caros y gran cantidad de no etiquetados. Se busca que la distribución de los datos no etiquetados ayude a alcanzar un mejor desempeño que si sólo se contara con los pocos datos etiquetados. Gran parte de la investigación actual en Aprendizaje Máquina se centra en entender las condiciones para que esto ocurra.
- *Inferencia transductiva*: la máquina (algoritmo) también recibe pocos datos eti-

quetados y otros no etiquetados que fungirán como prueba. El objetivo es predecir las etiquetas sólo para los de prueba no etiquetados. Es decir, se busca pasar de casos de entrenamiento **específicos** a casos de prueba **específicos**. Esto contrasta con el aprendizaje usual (inductivo) en donde se pasa de casos de entrenamiento específicos a reglas **generales** y posteriormente se aplican a casos de prueba. Este concepto fue introducido por Vapnik en 1990 motivado por el hecho de que la inducción requiere resolver un problema más general (inferir una función) y posiblemente computacionalmente prohibitivo antes de resolver un problema más específico que es más eficiente computacionalmente (calcular outputs para casos nuevos).

- *Aprendizaje on-line:* en este paradigma existen muchas rondas de entrenamiento y prueba. En cada ronda se recibe un dato no etiquetado, la máquina (algoritmo) hace una predicción, se le da la etiqueta real e incurre en una pérdida. El objetivo es minimizar la pérdida acumulada. A diferencia de los demás paradigmas explicados, aquí no se hace ninguna suposición sobre la distribución.
- *Aprendizaje por refuerzo:* en este paradigma también se mezclan las fases de entrenamiento y prueba. La máquina (algoritmo) recolecta la información interactuando con el ambiente con el fin de maximizar una noción de *recompensa acumulada*.
- *Aprendizaje Activo:* La máquina (algoritmo) recolecta interactivamente casos de entrenamiento pidiéndole a un oráculo que le dé datos etiquetados. El objetivo es tener un desempeño comparable al del paradigma de aprendizaje supervisado a menor costo (en términos de casos etiquetados).

En la práctica se encuentran escenarios más complejos que los paradigmas anteriores, generalmente mezclas.

Ahora se hará una exposición formal del modelo básico del Aprendizaje Máquina.

2.1.1. Modelo formal básico del aprendizaje

Aprendizaje en general

Según un compendio entre lo presentado por [55] y por [45], podemos decir que cualquier paradigma de aprendizaje requiere los siguientes elementos:

- **Dominio.-** Un conjunto Z de n individuos. Usualmente los elementos de Z se representan como vectores $z = (z_1, z_2, \dots, z_m)$ en donde z_i denota una propiedad del individuo z , $\forall i = 1, \dots, m$.

- **Modelo simple de generación de datos.**- Suponemos que los elementos de Z son generados como una muestra aleatoria por alguna densidad de probabilidad D sobre Z , de modo que son *i.i.d.*. Es importante mencionar que el *learner* (el algoritmo, la computadora) **no conoce** dicha distribución. Este supuesto es fundamental, pues si se conociera, no habría nada que aprender.
- **Conjunto de entrenamiento.**- $S \subset X$. Es el input que el *learner* tiene. Se toma como subconjunto estricto porque deben quedar individuos en X que constituirán el conjunto de prueba y /o validación, que será con los que midamos qué tan *bueno* es nuestro modelo.
- **Función de pérdida.**- Es una función \mathcal{L} que toma como entradas una función h y a Z y devuelve un número positivo:

$$\mathcal{L} : h \times Z \rightarrow \mathbb{R}^+$$

Como podemos observar, \mathcal{L} es una variable aleatoria. El objetivo general del Aprendizaje Estadístico es minimizar la esperanza de dicha función encontrando una función h^* , i.e.

$$\min_h E(\mathcal{L}(h, Z))$$

Aprendizaje supervisado

Este paradigma añade una *etiqueta* a cada individuo en Z . La idea central es que tenemos *individuos etiquetados*. Como comentamos, los problemas más importantes de *Aprendizaje supervisado* son clasificación y regresión.

Entonces:

- $Z = (X, Y)$ con Y un conjunto finito en el caso de *clasificación* en donde el caso más sencillo es en donde las etiquetas son binarias i.e. $Y = \{0, 1\}$. Entonces $Z = (X, Y)$ ahora denota a los individuos y a las etiquetas respectivamente.
- La función f mencionada arriba es tal que mapea individuos a sus etiquetas:

$$f : X \rightarrow Y \quad | \quad f(x) = y \quad \forall x \in X$$

Notar que f equivale a la *etiqueta correcta*, pero es importante mencionar que el

learner tampoco conoce esta función **ni es posible** que accese a ella.

- **Regla de predicción:** Es el *output* h del learner. Se le denomina **predictor, clasificador o hipótesis**. Surge para “acercarse” a la f desconocida:

$$h : X \rightarrow Y$$

Adicionalmente decimos que $A(S)$ es la predicción que un learner dado A hace si recibe el conjunto de entrenamiento S .

El problema en este paradigma se puede reexpresar como:

$$\min_h E(\mathcal{L}(h, (X, Y)))$$

- **Medidas de éxito:**

Para simplificar la notación, denotaremos

$$\mathcal{L}(h, (X, Y)) := \mathcal{L}_{D,f}(h)$$

En el caso del problema de **clasificación** se puede definir el error del clasificador $h : X \rightarrow Y$ como la probabilidad de que no prediga correctamente, es decir, dado $x \in X$ generado aleatoriamente de D :

$$\mathcal{L}_{D,f}(h) := P(f(x) \neq h(x))$$

En el caso del problema de **regresión** podemos definir el error como:

$$\begin{aligned} \mathcal{L}_{D,f}(h) &:= \|h(X) - f(X)\|^2 \\ &= \|h(x) - f(x)\|^2 \quad \forall x \in X, \quad x_i \sim D \end{aligned}$$

\mathcal{L} definida así se conoce como *error cuadrático*¹.

¹No perder de vista que estas dos sólo son algunas definiciones de \mathcal{L} . Son convenientes porque nos hablan del *error* incurrido al predecir a f con h

Nótese que el haber denotado $\mathcal{L}(h, (X, Y))$ como $\mathcal{L}_{D,f}(h)$ no fue a propósito: esto se hizo para hacer evidente el hecho de que ni D ni f se conocen, entonces los estimaremos usando el conjunto de entrenamiento S , que es la información sobre X que el *learner* tiene disponible. A esta estimación le llamaremos $\mathcal{L}_S(h)$.

Para el problema de **clasificación**, con la \mathcal{L} definida como arriba, un candidato natural para estimar el *error* es el error en que incurre el clasificador h en la muestra de entrenamiento. A este se le conoce como **error empírico**.

$$L_S(h) := \frac{\#\{i \in I : h(x_i) \neq y_i\}}{n} \quad \text{si } S = \{(x_i, y_i) \mid i = 1, \dots, n\}$$

Sin embargo, el hecho de que f pueda ser cualquier función puede generar *sobreajuste* al escoger predictores que se desempeñan muy bien en la muestra de entrenamiento pero muy mal en datos nuevos.

Entonces se define una clase H de predictores que se escoge a priori. Esto resuelve el sobreajuste ocasionado por no restringir el conjunto de predictores que se pueden escoger y se define lo siguiente.

Empirical Risk minimization con sesgo inductivo se define como el problema:

$$ERM_H(S) = \operatorname{argmin}_{h \in H} L_S(h)$$

donde H es un conjunto de hipótesis, h son funciones que mapean X en Y y S es una muestra de entrenamiento.

El hecho de condicionar de antemano a $ERM_H(S)$ a un conjunto de predictores hace que lo sesguemos hacia estos. A pesar de que idealmente H debería ser escogido usando experiencia previa sobre el problema que se quiere resolver, se puede demostrar que $ERM_H(S)$ no sobreajusta.

Entonces resulta interesante preguntarse qué tipo de hipótesis H no generan sobreajuste. Parece que el hecho de restringir nuestro conjunto de hipótesis hace que disminuya nuestro sobreajuste pero a su vez aumenta el sesgo hacia estos predictores. Esto formalmente formulado se conoce como el *No free-lunch theorem* o el problema *sesgo-varianza*.

Existe un resultado teórico que nos garantiza que clases finitas que cumplen ciertas propiedades son muy buenos aproximadores. Lo enunciamos a continuación.

Teorema (Clases de hipótesis finitas).- Sea H una clase de hipótesis finita. Sea

$\delta \in (0, 1)$ y $\epsilon > 0$ y sea $m \in \mathbb{N}$ tal que $m \geq \frac{\log(|H|/\delta)}{\epsilon}$. Entonces para toda función etiqueta f , para toda distribución D tal que existe $h \in H$ con $L_{(D,f)}(h) = 0$, tenemos que para toda hipótesis ERM, h_S , se cumple que $P(L_{(D,f)}(h_S) \leq \epsilon) = 1 - \delta$.

Interpretación: Para una muestra de entrenamiento de un tamaño suficientemente grande se cumple que el ERMH sobre una clase finita de hipótesis H será probablemente aproximadamente correcta (probabilidad $1 - \delta$ y con un error de hasta ϵ). Notar que entre más pequeña sea la tolerancia de error ϵ , o bien más grande sea la precisión $1 - \delta$, entonces requerimos una muestra más grande m .

De aquí surge la idea de **probablemente aproximadamente correcto**: Probably Approximately Correct, que es una propiedad muy deseable en nuestros modelos de Aprendizaje Máquina (nuestras hipótesis) y es el fundamento teórico de la teoría del aprendizaje.

2.1.2. PAC Learning

El modelo PAC es el modelo de Aprendizaje más usado, sin embargo existen otros enfoques con criterios diferentes (por ejemplo Aprendizaje vía convergencia uniforme y Aprendizaje no uniforme).

Def.- (PAC Learnability). Una clase de hipótesis H es **PAC-learnable** si existe una función $m_H : (0, 1)^2 \rightarrow \mathbb{N}$ y un algoritmo de aprendizaje A tal que $\forall \delta \in (0, 1), \forall D$ distribución sobre X y para toda etiqueta $f : X \rightarrow \{0, 1\}$ si la realización se cumple con respecto a H, D, f , entonces si corremos el algoritmo de aprendizaje en $m \geq m_H(\epsilon, \delta)$ *i.d.* (independientes idénticamente distribuidos) datos generados por D y etiquetados correctamente por f , el algoritmo regresa una hipótesis h tal que con probabilidad de al menos $1 - \delta$ (sobre los m datos), $L_{(D,f)}(h) \leq \epsilon$. ϵ, δ son los parámetros de precisión y de confianza respectivamente. O sea, $P(L_{(D,f)}(h_S) \leq \epsilon) = 1 - \delta$.

La interpretación del teorema es la siguiente:

- ϵ nos dice qué tan lejos podemos permitir que quede el clasificador del verdadero valor y es por esto que el algoritmo se denomina aproximadamente correcto.
- Por otro lado, δ nos dice qué tan factible es que el clasificador cumpla con el requisito de precisión y por esto el algoritmo se denomina probable. Se expresa en términos probabilísticos porque los datos de entrenamiento son variables aleatorias y como tales podrían no ser representativos del Dominio.
- $m_H : (0, 1)^2 \rightarrow \mathbb{N}$ determina la complejidad de la muestra de la clase de hipótesis

H , o sea, cuántos datos de prueba necesitamos para garantizar el aprendizaje PAC. $m_H(\epsilon, \delta)$ también depende de H , pues hay un teorema que ya enunciamos que dice que para una clase finita H la complejidad de la muestra depende del $\log(|H|)$.

Observación: Si H es *learnable* entonces hay infinitud de funciones que cumplen con las propiedades (simplemente se toma $\epsilon' < \epsilon$ y $\delta' < \delta$). Entonces se toma la mínima función que cumple eso y se denota $m_H(\epsilon, \delta)$.

Tenemos un resultado que acota superiormente el tamaño de la muestra.

Corolario.- Toda clase de hipótesis finita H es PAC learnable con complejidad de la muestra m_H acotada superiormente: $m_H(\epsilon, \delta) \leq \text{roof}(\frac{\log(|H|/\delta)}{\epsilon})$.

El modelo anterior puede ser generalizado relajando ciertos supuestos de modo que sea más relevante para tareas de tipo learnable en la práctica. Los supuestos que pueden ser relajados/cambiados son:

- Existe $h \in H$ con $L_{(d,f)}(h) = 0$. Para tareas prácticas este requerimiento es fuerte y entonces surge el modelo **PAC agnóstico** en donde este supuesto se relaja.
- Etiquetas binarias: el modelo PAC se puede extender a multclases y regresión.
- Adicionalmente se puede cambiar la función de pérdida (vimos en clase otros tipos de pérdida)

2.1.3. Teorema Fundamental del Aprendizaje

Finalmente presentaremos el teorema Fundamental del Aprendizaje. Como vimos, las clases finitas son *PAC learnable*, sin embargo también hay clases infinitas que lo son. Es decir, la finitud de H es una condición suficiente pero no necesaria.

Definición (dimensión Vapnik-Chervonenkis, VC).- La **dimensión-VC** de una clase de hipótesis H se denota como $VC - dim(H)$ y es el tamaño máximo de un subconjunto C de X que puede ser generado por H . Si H puede generar conjuntos de un tamaño arbitrariamente grande decimos que H tiene **dimensión-VC infinita**.

Teorema Fundamental del Aprendizaje.- Sea H una clase de hipótesis de funciones $f : X \rightarrow \{0, 1\}$ y sea la función de pérdida la *pérdida 0-1*. Entonces los siguientes resultados son equivalentes:

- H tiene la propiedad de convergencia uniforme.

- Cualquier regla ERM es un PAC learner agnóstico exitoso para H .
- H es learnable agnóstico.
- H es PAC learnable.
- Cualquier regla ERM es un PAC learner exitoso para H .
- H tiene dimensión finita.

La demostración exhaustiva de este Teorema sale de los alcances de este trabajo, sin embargo mostramos la interpretación del mismo.

Interpretación: Este Teorema caracteriza el aprendizaje de las clases de clasificadores binarios usando la dimensión VC, en donde la dimensión VC de una clase es una propiedad combinatoria que nos dice el tamaño de muestra máximo que puede ser soportado por la clase.

Nos dice que una clase es PAC learnable si y sólo si su dimensión VC es finita y especifica la complejidad de la muestra requerida para el aprendizaje PAC. Es decir, la dimensión VC de la clase nos provee de toda la información necesaria.

Observaciones: El teorema se cumple para clasificación binaria, sin embargo el resultado se extiende para regresión con la pérdida en valor absoluto o pérdida cuadrática. Tener en cuenta que no se cumple para todas las tareas de Aprendizaje.

2.1.4. Complejidad Computacional de Algoritmos de Aprendizaje

Hemos hablado ya de la “cantidad” de información que un algoritmo necesita para aprender, ¿Pero qué sucede con la complejidad computacional, la cual siempre es relevante en cualquier algoritmo? Pues Una vez que tenemos una muestra de entrenamiento adecuada se tienen que llevar a cabo cálculos computacionales para etiquetar a los elementos de S . ¿Cuántos recursos computacionales se necesitan?

Naturalmente el tiempo que un algoritmo se tarda en correr depende de la máquina en donde se esté corriendo. Para evitar que el tiempo de corrida varíe entonces el tiempo de corrida de un algoritmo se evalúa de forma asintótica.

Por ejemplo, la complejidad computacional del algoritmo de Ordenamiento por Mezcla el cual ordena una lista de n elementos es $O(n \log(n))$ si:

- El algoritmo se puede implementar en una máquina que cumpla los requisitos de un modelo abstracto de cómputo. Es decir, el algoritmo se puede implementar.
- El tiempo de corrida del algoritmo en segundos para ordenar n elementos debe de ser a lo más $c * \log(n)$ para $n > n_0$ y $c \in \mathbb{R}$.

Definición.- Se dice que una tarea es computable si puede ser realizada por un algoritmo con tiempo de corrida $O(p(n))$ para algún polinomio p .

El argumento de $O(n)$ es n , y por ejemplo si la tarea es ordenar elementos, entonces el argumento es el número de elementos a ordenar. De igual modo si la tarea es computar un cálculo aritmético entonces el input es el número de operaciones necesarias para calcularlo. En Aprendizaje Máquina no es muy claro quién es n .

Como parte final de la teoría de aprendizaje presentada en este trabajo, definiremos formalmente la **complejidad computacional del aprendizaje**.

Como ya vimos, un Algoritmo de Aprendizaje tiene acceso a un dominio \mathbb{Z} , a una familia de predictores H , función de pérdida l , un conjunto de entrenamiento S cuyos elementos son *i.i.d* y cuya distribución es D . Adicionalmente introdujimos a los parámetros ϵ y δ para denotar el margen de error permitido y la probabilidad de no obtener muestra representativa. Entonces se debe cumplir que: $L_D(h) \leq \min_{h' \in H} L_D(h') + \epsilon$

Definición.- (La complejidad computacional de un algoritmo de aprendizaje). Este algoritmo se define en dos pasos. Primero consideramos la complejidad computacional de un Problema de Aprendizaje, el cual está determinado por (X, H, l) . Posteriormente, en el segundo paso consideramos la tasa de cambio de la complejidad computacional a lo largo de la sucesión de tareas que debe realizar.

1. Sea f función tal que $f : (0, 1)^2 \rightarrow \mathbb{N}$, una tarea de Aprendizaje (X, H, l) y un algoritmo de aprendizaje, se dice que A resuelve la tarea de aprendizaje si:
 - A termina en a lo más $cf(\epsilon, \delta)$ operaciones.
 - El output de A , denotado h_A puede ser aplicado para predecir la etiqueta de un nuevo caso de X haciendo a lo más $cf(\epsilon, \delta)$ operaciones.
 - El output de A es PAC, i.e. con probabilidad de al menos $1 - \delta$ sobre las muestras aleatorias que cumplan $L_D(h) \leq \min_{h' \in H} L_D(h') + \epsilon$.
2. Sea $(X_n, H_n, l_n)_{n=1}^{\infty}$ una sucesión de problemas de aprendizaje y A un algoritmo de aprendizaje para resolver estos problemas. Dada una función $g : \mathbb{N} \times (0, 1)^2 \rightarrow \mathbb{N}$ decimos que A se resuelve en $O(g)$ si A resuelve el problema (X, H, l) en tiempo

$O(f_n)$ para $f_n : (0, 1)^2 \rightarrow \mathbb{N}$ se define por $f_n(\epsilon, \delta) = g(n, \epsilon, \delta)$.

Decimos que A es un algoritmo eficiente con respecto a la sucesión $(X_n, H_n, l_n)_n = 1^\infty$ si se resuelve en $O(p(n, 1/\epsilon, 1/\delta))$ para algún polinomio p . Es decir, si se puede descomponer en una sucesión de problemas de aprendizaje.

La teoría de la complejidad computacional de los algoritmos de Aprendizaje Máquina sale de los alcances de este trabajo, por lo que concluiremos la exposición teórica mencionando únicamente que el aprendizaje PAC y el teorema fundamental del Aprendizaje ignoran los aspectos computacionales e incluso hay casos en donde implementar el ERM es computacionalmente duro, por lo que se recurre a perder precisión ϵ para disminuir el grado de dureza y así surge el **Aprendizaje débil**. Al lector interesado se le sugiere dirigirse a [45].

Como conclusiones podemos tener que:

- Existen criterios formales que un algoritmo debe cumplir para poder aprender.
- No todo algoritmo cumple con esos criterios. Particularmente,[45] no cubre en su libro métodos como **redes neuronales** porque argumenta que actualmente hay falta de evidencia teórica que permita incorporar estos métodos a una teoría del aprendizaje sólida. De hecho, resulta interesante cómo se logran resultados impresionantes en tareas muy complejas de inteligencia artificial con redes neuronales, pero comúnmente no se entiende por qué, cómo o bajo qué condiciones se logran estos resultados. Para una discusión teórica exhaustiva del aprendizaje y la complejidad computacional en redes neuronales dirigirse al libro [1].
- No todo algoritmo debe aprender, existen otros fines de los algoritmos.
- En la práctica muchos algoritmos que se usan (entre ellos máquinas de soporte vectorial, clasificadores lineales) han demostrado *aprender* (en el sentido que hemos discutido en este capítulo) y por eso se usan como Algoritmos de Aprendizaje.

2.2. Fundamentos de Deep Learning

Los algoritmos de Aprendizaje Máquina convencionales funcionan muy bien para ciertas tareas simples, sin embargo no han logrado tener un éxito contundente en problemas centrales de Inteligencia Artificial, tales como reconocer voz o texto [3]. Esto ha motivado el uso de algoritmos de Deep Learning, pues se ha visto que alcanzan resultados sin precedentes en estas tareas.

La intuición de porqué Deep Learning funciona es clara: la profundidad de la red permite aprender una composición de conceptos y ayuda a generar abstracciones. Se trata de aprender buenas representaciones de los datos. Esta intuición naturalmente ha nacido tras éxitos prácticos de Deep Learning.

A pesar de que aún hoy no se cuenta con una teoría sólida que fundamente los principios de Deep Learning, sí existen conceptos teóricos clave que hay que entender para obtener una idea abstracta de la estructura de los algoritmos de Deep Learning. El objetivo de esta sección es dar una descripción de ellos.

2.2.1. Maldición de la Dimensionalidad

Muchos problemas de Aprendizaje Máquina se vuelven extremadamente difíciles cuando la dimensión de los datos es alta. Más específicamente, el número de configuraciones distintas de un conjunto de variables aumenta exponencialmente con el número de variables. Esto se conoce como la **maldición de la dimensionalidad**, concepto introducido por Richard E. Bellman. Otra manera de decir esto es que conforme el número de variables aumenta, la cantidad de datos que se necesitan para generalizar crece de manera exponencial.

La siguiente Figura 2.2 tomada de [3] muestra cómo conforme la dimensionalidad aumenta, los datos deben ser más ricos. Observaciones que eran *ricas* en espacios de dimensión baja, ser vuelven *ralas* en espacios de alta dimensionalidad. La pregunta es ¿Cómo podemos entrenar modelos significativos cuando se da la maldición de la dimensionalidad?

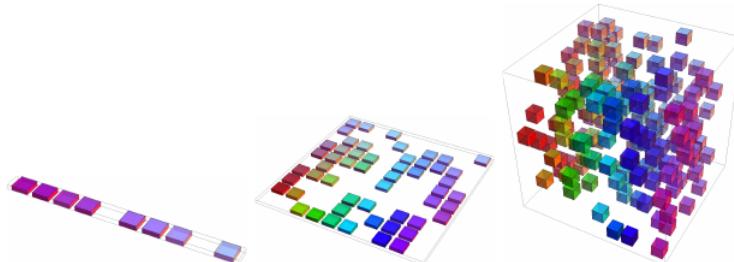


Figura 2.2: Maldición de la dimensionalidad

2.2.2. Constancia local y regularización

En la sección de *Fundamentos de Aprendizaje Máquina* vimos que para que los algoritmos de Aprendizaje Máquina puedan generalizar adecuadamente deben estar guiados por un conjunto de hipótesis H y distribuciones de los datos D que fungen como creencias “previas” sobre la función que deben aprender.

Otro tipo de creencias previas es la “smoothness prior”, que consiste en pedir que la función a aprender f^* no cambie demasiado en vecindades cercanas: $f^*(x) \approx f^*(x)$.

Este supuesto junto con los algoritmos no paramétricos usuales de Aprendizaje Máquina funcionan bien siempre y cuando existan suficientes datos en los picos y valles de las funciones, lo cual es cierto generalmente cuando la función a aprender es suficientemente suave y que cambie en pocas dimensiones (en altas dimensiones incluso una función suave puede cambiar de diferente manera en cada dimensión). Si la función además se comporta diferente por regiones, puede resultar extremadamente complicado describirla usando unos cuantos datos. La pregunta es si es posible generalizar aún cuando la función es complicada [3].

¿Es posible representar una función complicada eficientemente? ¿Es posible que la función estimada generalice bien en datos nuevos? La respuesta es sí según [3]. Un número grande de regiones, por ejemplo $O(2^k)$ puede ser definida por $O(k)$ datos siempre y cuando introduzcamos algunas dependencias entre las regiones vía supuestos adicionales acerca de la distribución subyacente de los datos. Así, es posible generalizar de manera *no-local*. Muchos algoritmos de Deep Learning dan supuestos explícitos o implícitos razonables que son necesarios para que una gran cantidad de problemas de Inteligencia Artificial puedan capturar estas ventajas.

Por otro lado, muchos algoritmos de Aprendizaje Máquina tienen supuestos fuertes sobre los datos que modelan. Tales supuestos no se les pide a las redes neuronales con el objetivo de que puedan generalizarse a una variedad mucho más rica de problemas, incluyendo aquellas de Inteligencia Artificial, cuya estructura es mucho más compleja.

La idea clave en deep learning es que asume que los datos fueron generados por la composición de factores o rasgos, potencialmente en múltiples niveles de jerarquía y se construye una *representación distribuida* de los datos. Esto permite una *ganancia exponencial* en la relación entre el número de datos y en número de regiones que se pueden distinguir. Dichas ganancias de las redes neuronales debidas a las representaciones distribuidas son un contrapeso a los retos que impone la maldición de la dimensionalidad [3].

2.2.3. Arquitecturas profundas

¿Por qué es *profundo*?

Según [3], en Inteligencia Artificial se puede definir la *profundidad* de un modelo mediante dos enfoques: la **profundidad de una gráfica computacional** o la **profundidad de una gráfica probabilística**.

El primer enfoque se basa en el número de instrucciones secuenciales que se tienen que ejecutar para evaluar la arquitectura. Esto se puede ver como la longitud de la ruta más larga a través de un flujo que describe cómo calcular cada uno de las salidas del modelo a partir de una entrada. Naturalmente, esta profundidad dependerá del lenguaje en el cual esté escrito el programa. También dependerá las funciones que se usen para dar los pasos a lo largo del flujo.

En la Figura 2.3 (tomada de [3]) muestra dos gráficas computacionales que mapean una entrada en una salida, en donde cada nodo lleva a cabo una operación. Ambas computan $\sigma(w^T x)$. La profundidad dependerá de qué se entienda por paso computacional, por ejemplo, en el modelo de la izquierda se usa suma, multiplicación y la función sigmoide, de modo que la profundidad es de tres; en el modelo de la derecha se ve a la función logística como un elemento en sí mismo, con lo cual la profundidad es de uno.

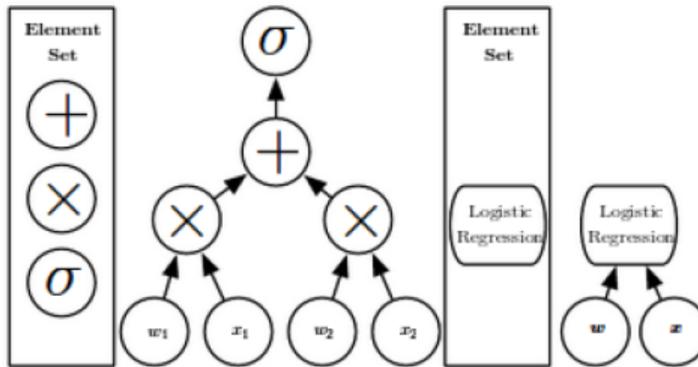


Figura 2.3: Dos gráficas computacionales con profundidad diferente

El segundo enfoque es el usado por modelos probabilísticos. Se refiere a la profundidad de la gráfica que describe cómo se relacionan los *conceptos* del modelo. Entonces, la profundidad del flujo de cálculos necesarios para calcular la representación de un

concepto puede ser mucho más profunda que los mismos conceptos. Esto se debe a que el entendimiento de conceptos simples puede refinarse ante la presencia de conceptos más complejos. Un ejemplo es la Figura 2.4 presentada en la introducción, en donde supongamos que un algoritmo sólo es capaz de detectar un ojo (porque el otro no se ve, porque está nebuloso, etc). Sin embargo, supongamos que en otra capa del algoritmo se han detectado caras, entonces la capa que sólo vio el ojo puede inferir que debe haber otro ojo, refinando así su conocimiento. Es decir, es capaz de inferir elementos a partir del *contexto* dado por conceptos más *abstractos*. En el ejemplo de la Figura la *gráfica de conceptos* tiene una profundidad de 5, sin embargo la profundidad es de la gráfica computacional contempla $5 * n$ capas suponiendo que en cada capa se hacen n cálculos.

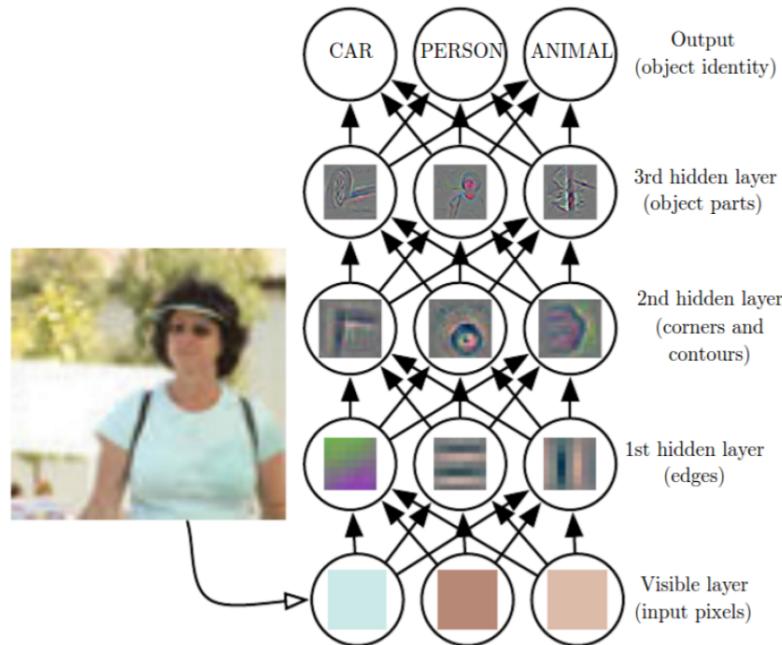


Figura 2.4: Conceptos simples refinan su significado

Debido a esto no hay una única respuesta correcta sobre la profundidad de una arquitectura, y a veces un enfoque es más relevante que otro. Tampoco hay un consenso sobre *qué tan profundo* tiene que ser un modelo para ser *profundo* [3]. No obstante, Deep Learning puede ser considerado como el uso y estudio de modelos que relizan una mayor composición de funciones o conceptos aprendidas que los modelos tradicionales de Aprendizaje Máquina. Es por esto que una red neuronal con muchas capas

intermedias se considera el modelo prototípico de Deep Learning.

2.2.4. Entrenamiento de redes profundas

Limitantes del algoritmo de retropropagación

Ya que hemos expuesto los componentes matemáticos y técnicos de las arquitecturas profundas surge naturalmente la pregunta: ¿ahora cómo las entrenamos?

Al igual que en las redes neuronales usuales, tendremos una superficie de error y el objetivo es llegar al punto mínimo de esta, o lo más *cercano posible* a este punto. Podemos visualizar la superficie de la siguiente manera en la Figura 2.5 tomada de https://en.wikipedia.org/wiki/Backpropagation#/media/File:Error_surface_of_a_linear_neuron_with_two_input_weights.png.

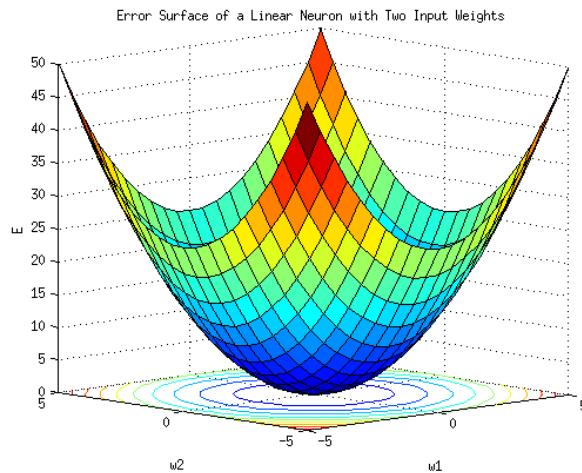


Figura 2.5: Superficie de error

En la introducción de este trabajo mencionamos que en el caso de arquitecturas profundas, el algoritmo de retropropagación tiene un pobre desempeño al entrenar. El problema radica en que este algoritmo está basado en el cálculo del gradiente, que es un método que usa información local y usualmente se inicializa en puntos aleatorios, lo cual puede ocasionar que se quede atorado en mínimos locales, incluso si se usan modificaciones al algoritmo como el *batch-mode* o el gradiente de descenso estocástico

[14]. Este riesgo aumenta conforme la profundidad de la red aumenta. Entonces en vez de tener algo como en la Figura 2.5, tenemos una una situación como la siguiente Figura 2.6:

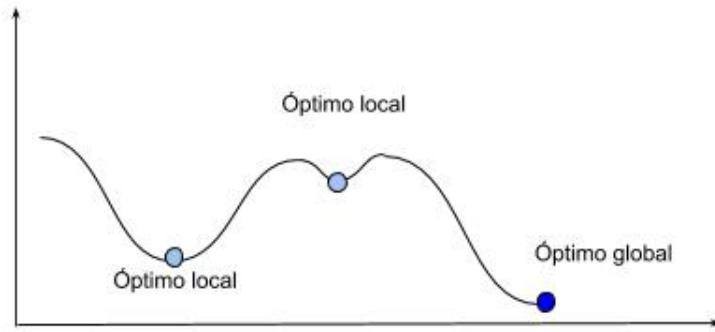


Figura 2.6: Presencia de óptimos locales

Históricamente eso hizo que las redes neuronales profundas fueran relegadas y sustituidas por modelos superficiales con funciones de pérdida convexas como las máquinas de soporte vectorial para las cuales el óptimo global es único y se puede obtener de manera eficiente [14] [46].

Además del problema de la inicialización de parámetros, el algoritmo de retropropagación ha mostrado correr muy lentamente en redes profundas, lo cual hace que pare antes de alcanzar un progreso significativo y genera un pobre ajuste en el conjunto de entrenamiento [42].

Esta tendencia no cambió hasta el surgimiento de dos trabajos muy importantes que mostraron que era posible entrenar eficientemente redes profundas usando algoritmos de aprendizaje no supervisado.

Primeras alternativas: pre-entrenamiento

La primer alternativa fue mostrada en *A fast learning algorithm for deep belief nets* por Hinton, Osindero y Teh en 2006 en donde se desarrolla un algoritmo Greedy que entrena redes neuronales profundas (en específico Deep Belief Networks) capa por capa. Usando esto se propone un modelo que da una mejor clasificación de dígitos que los mejores algoritmos usados hasta la fecha para ese fin [29].

La segunda alternativa fue presentada en *Reducing the dimensionality of data with neural networks* por Hinton y de Salakhutdinov en 2006 donde se describe una manera efectiva de inicializar los pesos de una red neuronal profunda (en específico un *autocodificador*) que permite aprender representaciones de los códigos en dimensiones más bajas y además funciona mejor que componentes principales como herramienta para reducir la dimensión de un conjunto de datos, pues se pueden usar relaciones no lineales [30]. Incluso es posible mejorar los resultados de un perceptrón multicapa si en una fase pre-entrenamiento se inicializan los pesos usando el algoritmo descrito en [30] y posteriormente se hace *fine-tuning* usando retropropagación [14].

A pesar de que históricamente los dos trabajos anteriores son los de más relevancia, también se tienen las siguientes estrategias no supervisadas para pre-entrenar arquitecturas profundas:

- Entrenar capa por capa considerando cada par de capas como un autocodificador reductor de ruido (de-noising autoencoder por su nombre en inglés) y regularizarlo asignando a un conjunto aleatorio de los nodos de entrada el valor cero [2].
- Usar un *autocodificador contractivo* con el fin de favorecer representaciones que son más robustas ante variaciones de los inputs, i.e. penalizar al gradiente por cambios en el comportamiento de las neuronas ocultas con respecto a los inputs [51].
- Máquina simétrica de codificación rala (SESM por sus siglas en inglés) cuya función se parece mucho a las Máquinas Restringidas de Boltzmann (RMBs por sus siglas en inglés) como los pilares fundamentales de las Redes de Creencia Profunda.

Y finalmente también existen estrategias supervisadas para pre-entrenar, también llamadas *pre-entrenamiento discriminativo*, que han mostrado ser exitosas e inclusive, cuando la cantidad de datos etiquetados es muy grande, se desempeñan mejor que las de pre-entrenamiento no supervisado [14].

Las alternativas anteriores consideran que es necesario hacer un pre-entrenamiento de la red para obtener resultados satisfactorios, pues de esta manera se obtiene una mejor generalización del conjunto de entrenamiento [15]. Las razones por las cuales esto es así han sido estudiadas por numerosos investigadores, destacando que las funciones a optimizar de las arquitecturas profundas tienen una mayor prevalencia de óptimos locales y el pre-entrenamiento ayuda a aliviar el problema [42] al proveer mejores puntos de inicialización.

Otras alternativas

Por otro lado, Martens[42] sugiere que más bien, las funciones de las arquitecturas profundas tienen *curvatura patológica*, lo cual hace que métodos que son ciegos a la curvatura como el algoritmo de retropropagación, no funcionen adecuadamente en las arquitecturas profundas. El algoritmo *Libre de Hessiano* de Martens[42] es un método de segundo orden que modela la curvatura local sin usar el Hessiano -el cual es computacionalmente intensivo- y que logra corregir la curvatura patológica. El método mostró ser exitoso e inclusive tuvo un mejor desempeño que el propuesto en *Reducing the dimensionality of data with neural networks* por Hinton y Salakhutdinov para el mismo conjunto de entrenamiento [42].

Otras alternativas son los *Métodos de subespacio de Krylov* que tiene la capacidad de salir de óptimos locales e incluso mostró una convergencia más rápida que el *Libre de Hessiano* [61].

Del mismo modo, el descenso de gradiente estocástico (SGD por sus siglas en inglés) generalmente es el algoritmo más eficiente cuando el conjunto de entrenamiento es grande y redundante, lo cual ocurre en la mayoría de las aplicaciones [14]. De hecho, el descenso de gradiente estocástico ha mostrado ser efectivo para parallelizar múltiples máquinas o para parallelizar múltiples GPUs usando el algoritmo de Retropropagación en forma de tubería [6]. Este algoritmo adicionalmente tiene la ventaja de poder salir de óptimos locales debido a que se muestrea un lote de observaciones para estimar el gradiente.

Otra alternativa para mejorar el desempeño de una arquitectura profunda es el uso de capas ocultas con muchas neuronas: el riesgo de quedar atrapado en un óptimo local es mayor cuando pocas neuronas son usadas. Sin embargo esta opción no fue anteriormente explorada porque, al igual que las arquitecturas profundas, las anchas eran computacionalmente prohibitivas, llevando a un reto computacional a lo largo y a lo ancho [14].

Finalmente podemos mencionar el uso de otras funciones alternativas a la sigmoidal $f(x) = \frac{1}{1+e^{-x}}$ y a la tangente inversa $g(x) = \tanh(x)$ para la activación de unidades, por ejemplo la ReLU (Rectified Linear Unit) definida por:

$$h(x) = \sum_i \sigma(x - i + 0,5) \approx \log(1 + e^x)$$

$\log(1 + e^x)$ puede ser aproximada por la función máximo, i.e. $\max(0, x)$. A esta función se le llama ReL (Rectified Linear Function).

Las principales ventajas de esta nueva función de activación es que, a diferencia de la función sigmoide, el gradiente de la función ReL no se desvanece conforme x crece. Adicionalmente, se ha mostrado que usando ReLUs es posible entrenar redes profundas sin necesidad de preentrenamiento [41].

2.3. Aplicaciones usuales de Deep Learning

Wisdom comes from only through failed experimentation.

Damian Mingle

En este capítulo se hace un recuento de las áreas y aplicaciones en las que *Deep Learning* ha alcanzado resultados exitosos. Es importante mencionar que esta lista no pretende en lo más mínimo ser exhaustiva, pero sí mostrar los resultados más relevantes.

Gran número de estas aplicaciones son rentables. Como se mencionó en la introducción, *Deep Learning* es usado en empresas que desarrollan, compran y usan tecnología de punta como Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA y NEC [3].

Reconocimiento de imágenes

Deep learning ha logrado ser aplicado con éxito en tareas cada vez más diversas. Estos éxitos empezaron desde mucho antes que su popularidad creciera en 2006 con el paper de Hinton [29] como se explicó en la introducción.

El primer logro en cuanto a reconocimiento de imágenes se refiere, se atribuye a Rumelhart et al. en 1980 con su artículo *Learning representations by back-propagating errors* [52], en donde se mostró cómo se podían reconocer objetos individuales en imágenes pequeñas y recortadas como la que se puede observar en la Figura 2.7.



Figura 2.7: Reconocimiento de imágenes recortadas

La red LeNet fue introducida por primera vez en [40] en 1998 para reconocimiento de caracteres en documentos usando redes neuronales convolucionales.

A partir de ahí las redes neuronales fueron capaces de procesar imágenes cada vez más grandes y para 2012 procesaban imágenes de alta resolución sin necesitar que la imagen estuviera recortada y podían reconocer más un sólo objeto en la misma imagen. Actualmente las redes más modernas pueden reconocer al menos 1,000 tipos de objetos diferentes en una misma imagen [3].

El punto cúspide para las redes profundas en el reconocimiento de imágenes fue en el concurso más importante de este rubro: ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). En 2006 una red de convolución profunda mostró resultados nunca antes vistos logrando un error de 15.3 % versus 26.2 % de su competidor. La siguiente Figura 2.8 tomada de Krizhevsky et al. [35] y modificada para fines de este trabajo, se muestran los resultados y puede apreciarse cómo el modelo asignaba cinco posibles categorías para el objeto y le asignaba una probabilidad a cada uno, la cual se muestra en las barras. La barra roja corresponde a la probabilidad de que el modelo le asignaba al objeto correcto y podemos observar que en las imágenes en donde el modelo fue completamente exitoso (1-4) el objeto a reconocer se hallaba completo, aislado y definido que en las imágenes en donde el modelo no fue completamente exitoso (imágenes 5-8). De hecho las figuras 7 y 8 son interesantes porque el modelo ni siquiera contempló que el objeto era un racimo de cerezas y un gato de madagascar respectivamente. En estas imágenes los objetos se pueden confundir con un perro (imagen 7), o bien se hallan inmersos en un ambiente muy poblado con otros objetos (imagen 8).



Figura 2.8: Resultados del modelo de Krizhevsky et al. de 2006

A partir de ese año los modelos de *Deep Learning* han ganado consistentemente este concurso y cada año se alcanzan menores tasas de error. Actualmente la tasa de error es de sólo 3.6% [3].

Otro problema real en donde el reconocimiento de imágenes se ha aplicado es la *Clasificación de señales de tránsito*, en donde los algoritmos de *Deep Learning* han logrado resultados superiores a los de un humano (0.54 % de error). La arquitectura usada fue una MCDNN (Multi-column deep neural network) que básicamente entrena Redes neuronales profundas en diferentes conjuntos preprocesados [8].

Reconocimiento de voz

Deep Learning también ha tenido un impacto significativo en el reconocimiento de voz. Durante los 90's las tasas de error mantuvieron un ritmo de mejora, sin embargo el progreso se estancó al llegar al 2000 y no fue hasta el 2010 cuando la introducción de modelos de *Deep Learning* lograron seguir con dicho ritmo e incluso algunos modelos lograron reducir los errores a la mitad [3].

Otras áreas y problemas de aplicación

Otra aplicación en donde *Deep Learning* ha sido exitoso es el paradigma de aprendizaje *reinforcement learning* que mencionamos en la introducción, el cual se inspira en psicología conductual. Brevemente, un agente autónomo debe aprender a hacer una tarea por ensayo y error sin tener ninguna guía humana para hacerlo y buscando maximizar su recompensa acumulada. DeepMind demostró que un sistema de *reinforcement learning* basado en *Deep Learning* (específicamente una red neuronal profunda de convolución, DCNN) es capaz de aprender a jugar los videojuegos de Atari (ver Figura 2.9 tomada de [44]) y alcanza un desempeño parecido al del humano en varios de ellos [44].



Figura 2.9: Screenshots de cinco videojuegos de Atari 2600: Pong, Breakout, Space Invaders, Seaquest, Beam Rider.

Un área de estudio muy novedosa y que empuja los límites del conocimiento son las *máquinas de Turing neuronales*. Estas arquitecturas extienden las capacidades de las redes neuronales al darles la posibilidad de acceder a recursos externos de memoria y el resultado es parecido a tener una Máquina de Turing o una arquitectura de Von Neumann diferenciable, lo cual permite que sea posible entrenarla usando el método del gradiente. Los resultados preliminares fueron presentados en el artículo *Neural Turing Machines* de Graves et al. [26] y muestran que estas arquitecturas son capaces de inferir algoritmos sencillos como copiar, ordenar y hacer llamadas de memoria asociativas ².

Otra aplicación novedosa de *Deep Learning* es en la industria farmacéutica, lo cual es sorprendente dada la reticencia a la tecnología de la que esta industria ha siempre ha sufrido. Básicamente una droga de patente pasa por 3 etapas comerciales principales: antes del lanzamiento (cuando se investiga y desarrolla químicamente la droga), comercialización (cuando se vende al público) y decaimiento (pérdida de patente) (Ver Figura 2.10). Encontrar nuevos tratamientos que sean eficaces (i.e. que ayuden significativamente a controlar, revertir o curar una enfermedad) y que además cumplan mínimos requerimientos toxicológicos y del metabolismo es una tarea complicadísima.

²Son capaces de recuperar pedazos completos de información a partir de sólo datos parciales

Esto hace que muchas drogas no pasen a la fase dos y en gran medida se debe a que las moléculas usadas son químicamente incompatibles o que juntas crean reacciones adversas en el organismo. Usualmente esto lo hacen expertos por ensayo y error, es decir, desarrollan los compuestos y ven si sirven, ocasionando que enormes cantidades de recursos se pierden en esta fase. *Deep Learning* empieza a ser usado para predecir cómo interactuarán las moléculas de drogas que se hallan en la fase de investigación, de modo que se puedan desechar los compuestos químicamente incompatibles sin tener que fabricarlos previamente [50]

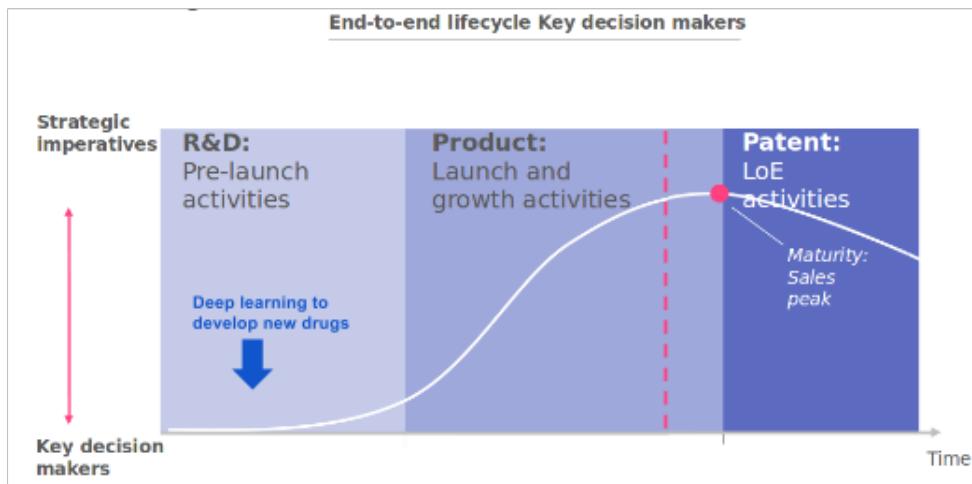


Figura 2.10: Ciclo de vida usual de una droga y el uso de Deep Learning en R&D.

Así como la escala y desempeño de las redes profundas se ha incrementado, los problemas también han incrementado de complejidad. Las redes neuronales recurrentes, en específico la Long Short Term Memory Network se han usado para modelar problemas del tipo sequence-to-sequence permitiendo tener entradas y salidas multidimensionales (antes esto no era posible). De esta manera, el aprendizaje tipo sequence-to-sequence también ha revolucionado áreas como la traducción automatizada, la lectura de comprensión y análisis de sentimiento [3].

Según [3], la complejidad de estos problemas ha alcanzado su cúspide con las Máquinas Neuronales de Turing, que extienden las capacidades de las redes neuronales al añadirles recursos de memoria externos con los cuales pueden interactuar mediante procesos de atención. El sistema combinado es análogo a una Máquina de Turing o una Arquitectura de Von Neumann pero es diferenciable de inicio a fin, con lo cual puede entrenarse eficientemente con descenso en gradiente. Los resultados preliminares muestran que

estas redes pueden inferir algoritmos simples como copiar, ordenar y asociar si se les provee de inputs y outputs adecuados de los cuales puedan aprender estas tareas [26].

Desarrollo de software

Como ya mencionamos en la introducción, los avances de deep learning han dependido en gran medida de la tecnología disponible. Esto ha empujado el desarrollo de software exclusivamente para desarrollar *Deep Learning*.

Las librerías más importantes diseñadas con este fin y que han sido usadas exitosamente para aplicaciones comerciales o académicas son las siguientes:

- **Theano**: librería de Python desarrollada por la universidad de Montreal.
- **Torch**: provee un ambiente parecido a Matlab. Basado en Lua.
- **Caffe**: pensado para ser rápido y para trabajar en módulos.
- **Deeplearning4j**: es el primer software pensado para aplicaciones comerciales para Java y Scala.
- **Tensorflow**: librería sucesora de DistBelief, fue desarrollada por Google Brain.
- **Keras**: librería de Python para usarse sobre Theano o Tensorflow.

Conclusiones

En resumen, *Deep Learning* ha tenido un crecimiento impresionante en los últimos años y ha mostrado lograr resultados que hoy son el *estado del arte*, es decir, que no han podido ser superados por otras arquitecturas.

Es importante mencionar que una arquitectura profunda no es la mejor para todos los problemas de aprendizaje. A continuación contaremos una breve anécdota que ocurrió apenas hace dos años.

Adam Gibson es el creador de DeepLearning4j, una librería abierta orientada al uso de Deep Learning en los negocios. Se dice que en una conferencia de 2014 sobre Deep Learning Gibson dijo que sugería el uso de algoritmos de Deep Learning en problemas relacionados con media (video, imágenes, sonido, texto) y para análisis de series de tiempo. Y cuando un integrante de la audiencia le preguntó si podía usar Deep Learning

para detectar fraudes, Adam le dijo de manera directa que “no quería verlo ahí (en la conferencia)”.

El éxito de *Deep Learning* para realizar reconocimiento de imágenes, voz y texto normalmente se atribuye a la *jerarquía* inducida a través de las múltiples capas del modelo. Cada capa procesa y resuelve una parte de la tarea y la pasa a la siguiente capa a modo de pipeline [28]. Si dicha jerarquía no está presente en el fenómeno a modelar, posiblemente no sea necesario usar una arquitectura profunda.

Entonces al parecer, no todos los problemas son aptos para ser modelados con *Deep Learning*. Sin embargo, no siempre es posible saber a priori cuándo una arquitectura profunda es pertinente o no y el objetivo tampoco es saberlo a priori, pues gran parte del avance científico se logra vía ensayo y error. Sin embargo, es crucial por un lado, conocer la arquitectura de los modelos, cómo funcionan, cómo se aplican, cómo se miden sus resultados, y por otro lado también es crucial entender cuál es el problema o fenómeno que queremos modelar, cuáles son los modelos que se han usado previamente y qué resultados han alcanzado con el fin de decidir si realmente vale la pena usar modelos más complejos.

No obstante, las áreas y problemas que sí son objeto de aplicación son incontables y *Deep Learning* está llena de retos y oportunidades excitantes que seguramente nos llevarán a terrenos insospechados en los años venideros.

2.4. Deep Learning para series de tiempo

Time is eternity that sees its own implementations.

Plato

Además de las áreas y aplicaciones en las que *Deep Learning* ha sido exitoso, que ya comentamos en el capítulo anterior, existen áreas que tienen potencial de ser impactadas de manera significativa y cuya investigación actualmente se encuentra en pleno crecimiento.

Deng et al. [14] identificaron tres áreas de oportunidad en 2013³:

³Notar que debido a que *Deep Learning* tiene un avance acelerado, dichas áreas pueden ya no ser las mismas.

- Procesamiento de texto y lenguaje natural.
- Recuperación de información.
- Procesamiento de información multimodal y temporal.

2.4.1. Trabajo relacionado

Varias de las aplicaciones que mencionamos en el capítulo pasado suponen datos que no están indexados por el tiempo, es decir, datos que son estáticos. ¿Qué pasa con datos indexados por una variable temporal? ¿Son los algoritmos de *Deep Learning* capaces de modelar este tipo de información? Si la respuesta es “sí”, ¿específicamente qué arquitecturas? ¿son exitosos?

Las arquitecturas profundas usadas en el *Procesamiento de texto y lenguaje natural*, han empezado a usarse para modelar series de tiempo y es una aplicación que ha ganado atención reciente según [36]. Se hizo una búsqueda de literatura a finales de enero 2016 y se encontraron al menos los siguientes artículos que han usado *Deep Learning* para modelar series temporales.

1. *Factored Conditional Restricted Boltzmann Machines for Modeling Motion Style*. Taylor & Hinton, 2009 [60]. Tiene 152 citas y es un artículo que presenta un modelo que añade interacciones temporales entre unidades visibles basado en una *Conditional Restricted Boltzmann Machine* pero reduciendo el tiempo de $O(n^3)$ a $O(n^2)$ para modelar el movimiento humano.
2. *Training recurrent neural networks*. Sutskever, 2013 [59]. Tiene 45 citas y en la sección 3 de este trabajo el autor desarrolla RBMs (Restricted Boltzmann Machines) con interacciones temporales entre las unidades.
3. *Deep Learning for Time Series Modeling*. Längkvist, Lars Karlsson & Amy Loutfi, 2014 [36]. Tiene 19 citas y presenta una revisión de los avances recientes en Deep Learning y en Feature Learning para problemas de series de tiempo señalando los retos intrínsecos en el modelaje de datos temporales mediante algoritmos de Deep Learning.
4. *Parsing Natural Scenes and Natural Language with Recursive Neural Networks*. Socher, 2011 [56] usa autocodificadores recursivos para hacer parseo. No es estrictamente series de tiempo pero es trabajo relacionado.
5. *Training and Analyzing deep recurrent neural networks*. Hermans & Schrauwen, 2013 [28]. Tiene 44 citas y muestra porqué una Red neuronal recurrente no pro-

funda no logra capturar la estructura temporal y propone el uso de una red del mismo tipo pero profunda.

6. *Deep Recurrent Neural Networks for Time-series prediction*, 2014 [49]. Se usa una red neuronal recurrente profunda con un tiempo más largo de retropropagación en el tiempo como una solución para modelar sistemas de órdenes altos y para predecir el modelo se aplica una convulsión elíptica.

Se encontró que en varios de los artículos se usaba una red neuronal recurrente con algunas variantes. Intuitivamente, estos modelos en teoría son capaces de modelar series de tiempo porque su arquitectura incorpora ciclos, lo cual hace que puedan modelar las dependencias temporales de una serie de tiempo. Si bien es cierto, estas redes padecen problemas al modelar dependencias temporales altas. En las secciones posteriores se ahondará en este punto y se verá cómo modificar esta arquitectura para corregir este detalle.

Para cerrar esta sección observamos que la mayoría de los trabajos existentes se enfocan en entrenar algoritmos de *Deep Learning* y no ahondan en su arquitectura básica [28] ni en explicar porqué usaron el modelo que usaron. Por lo mismo en una sección que se presenta más adelante veremos porqué estas arquitecturas (DRNN) modelan bien datos temporales partiendo de una red neuronal recurrente y se dará una descripción matemática precisa de estas.

Cómo preámbulo y para entender la importancia de las series temporales daremos una introducción a los modelos que históricamente se han usado para modelar series de tiempo.

2.4.2. Series de Tiempo

La mayoría de los datos tienen (o nuestro cerebro cree que tienen) un componente temporal, se mida o no. Desde procesos naturales (las ondas de sonido, el clima, el día y la noche, la descomposición de una partícula) hasta artificiales (mercados financieros, PIB de un país). Incluso un objeto que pase de un estado a otro se puede pensar como que en un tiempo t_0 tenía una propiedad y en un tiempo t_1 tenía otra.

Entonces, encoentramos series de tiempo en variadas áreas: finanzas, salud, economía, telecomunicaciones, biología, etc. Básicamente cualquier fenómeno que tenga un componente temporal y que se mida en un período, en teoría puede representarse como una serie temporal. En todas estas aplicaciones se busca encontrar patrones, detectar comportamientos atípicos y predominantemente predecir el comportamiento futuro.

No ha de sorprendernos entonces que el análisis de series de tiempo haya sido objeto de estudio durante décadas y que sea considerado por muchos investigadores de Minería de Datos como uno de los diez problemas más desafiantes debido a las propiedades únicas que el tiempo le impone [36].

Modelos usuales para series temporales

Algunos enfoques tradicionales para modelar series de tiempo son los siguientes [36]:

- Los modelos de estadística clásica autorregresivos o de Box-Jenkins en donde se estiman los parámetros de un modelo de series de tiempo propuesto por el investigador.
- Sistemas dinámicos lineales o no lineales.
- Modelos de Markov ocultos.
- Modelos basados en Minería de datos.

Sin embargo, cuando se tienen series de tiempo complejas, de altas dimensiones y con mucho ruido no es posible describir los datos usando ecuaciones con parámetros a estimar, pues las dinámicas son muy complejas o incluso desconocidas [36]. Esto sin mencionar los supuestos importantes que los modelos de estadística clásica imponen a los datos. Por otro lado, modelos como los basados en minería de datos lo que hacen es reacomodar la serie temporal para eliminar la columna que mide el componente temporal; al hacer eso añaden varias variables objetivo retrasadas (*lagged variables*) y además otras variables adicionales para modelar la tendencia y la estacionalidad. Al añadir todas estas variables los modelos corren en riesgo de sufrir de la *maldición de la dimensionalidad* (mencionada en la sección 2.2), pues los datos se vuelven ralos en un espacio de tan alta dimensión.

Por otro lado, las redes neuronales superficiales (no profundas), que también pueden catalogarse como modelos de Minería de datos, al contener un número pequeño de operaciones no lineales tampoco tienen la capacidad de modelar adecuadamente estas series [36]. Una opción es generar *rasgos* (rasgos en inglés) robustos que capturen la estructura compleja. El problema de esto es que, como vimos en la introducción de este trabajo, desarrollar *rasgos* hechos a la medida es caro, tardado y requiere tener conocimiento sobre los datos [36].

La red neuronal recurrente (RNN), generalmente se ha usado para modelar *sucesiones*-en texto, en voz, etc- de manera razonablemente exitosa. Las sucesiones, más abstrac-

tamente, son una serie de valores ordenados ascendentemente por los naturales \mathbb{N} . Las series de tiempo pueden verse entonces como una sucesión ordenada por el tiempo. Entonces, parece razonable pensar que las RNN pueden aplicarse para modelar series temporales. Sin embargo, estudios han mostrado que una RNN no tiene la capacidad de incorporar la jerarquía que es introducida por el tiempo ($t_0 < t_1 < t_2 < \dots$), ni las dependencias a largas distancias [28]. El objetivo es entonces tener una arquitectura que atienda a estas deficiencias. Para lograr esto, propondremos dos esquemas que son casos especiales de una red neuronal recurrente, pero sin los problemas anteriormente señalados:

1. Red neuronal recurrente profunda (DRNN)
2. Long Short Term Memory Network (LSTM)

En cualquier caso, el primer paso para construir cualquiera de las dos propuestas es introducir a la RNN. Primero discutiremos la DRNN y posteriormente la LSTM.

2.4.3. Construcción de una DRNN

Una RNN puede verse DRNN si se despliega en el tiempo, pues al hacer esto la arquitectura crece considerablemente de tamaño al incrementar sus capas intermedias. Sin embargo, el artículo *How to construct deep recurrent neural networks* de Pascanu et al. [47] trata de extender la noción de RNN a una DRNN; encuentra tres maneras de hacerlo y concluye que añadirle profundidad a las redes recurrentes mejora su desempeño.

Comencemos explicando primero intuitivamente qué es una RNN y después daremos la definición matemática.

Red neuronal recurrente (RNN)

Supongamos que vemos una película y queremos clasificar a los escenas de alguna manera. Escenas posteriores estarán definidas en términos de las anteriores. No parece claro cómo una red neuronal normal lograría hacer esto, pues el flujo de alimentación es hacia adelante únicamente (ver Figura 2.11).



Figura 2.11: Flujo hacia adelante de una red usual.

Necesitamos incluir algún tipo de *memoria* en la neurona que haga que la información persista durante todo el proceso del entrenamiento. Entonces consideremos el siguiente modelo de red de la Figura 2.12.

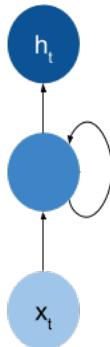


Figura 2.12: Flujo hacia adelante de una red con ciclo.

Esta es la característica que diferencia a las redes neuronales recurrentes de las usuales: *los ciclos*. Notemos que una red neuronal recurrente puede ser vista como una red neuronal usual si se desenreda en el tiempo, pues simplemente es una copia de sí misma múltiples veces (ver Figura 2.15)

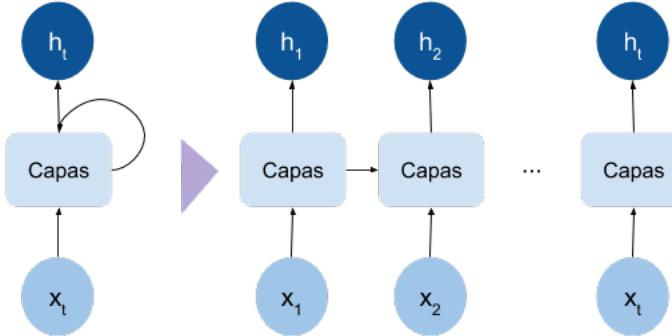


Figura 2.13: Red neuronal recurrente vista como una red usual.

La Figura 2.15 permite ver que la red neuronal recurrente parece ser apta para procesar información secuencial. Y de hecho lo es: esta arquitectura ha sido usada con mucho éxito para reconocimiento de voz e imagen, modelaje de lenguaje y traducción justamente por su habilidad para modelar datos que están relacionados de manera secuencial. De hecho, las redes neuronales recurrentes son los pilares de los modelos sequence-to-sequence mencionados anteriormente, que consisten en dos redes neuronales recurrentes: un codificador que procesa el input y un decodificador que genera el output [7].

Más aún, la Figura 2.14 (ejemplo tomado de [34]) ilustra cómo trabaja esta arquitectura para cada variante de input y output. Los rectángulos son vectores. Nótese que en ningún caso hay restricciones en cuanto al tamaño de las sucesiones porque en la fase de transformación recurrente (en donde tenemos los ciclos) puede ser aplicada tantas veces como se desee, incrementando el tamaño de la sucesión.

- Arquitectura 1 - Uno a uno: es la tradicional, sin RNN; mapea un input de tamaño fijo a un output de tamaño fijo. Es la típica de clasificación.
- Arquitectura 2 - Uno a muchos: RNN. Un ejemplo es tener de input una imagen y de salida una oración conformada por palabras.
- Arquitectura 3 - Muchos a uno: RNN. Análisis de sentimientos, en donde una oración se clasifica y expresa un sentimiento.
- Arquitectura 4 - Muchos a muchos: RNN. Traducción de máquina, en el input se lee una oración en inglés y en el output saca la oración en francés.
- Arquitectura 5 - Muchos a muchos: RNN. Clasificación de un video en donde

queremos etiquetar a cada escena del video.

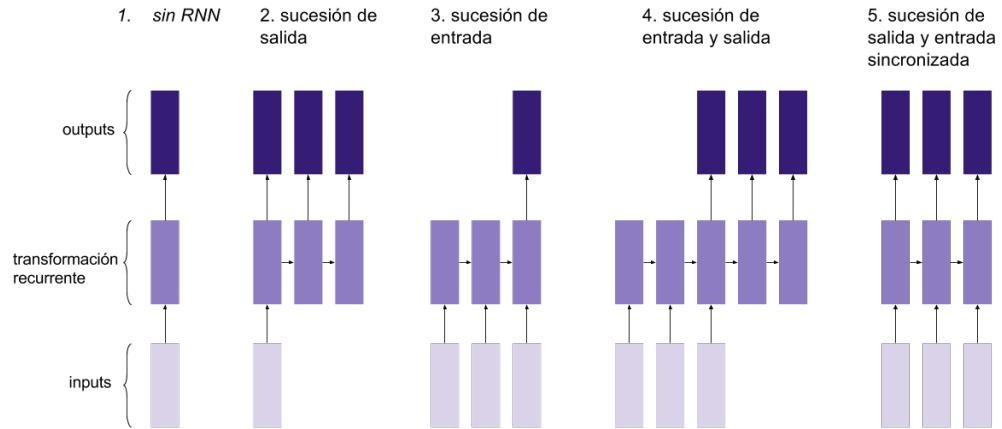


Figura 2.14: Arquitecturas de RNN usadas en varias aplicaciones.

Ahora que tenemos una idea intuitiva de cómo funcionan y para qué sirven, las introduciremos matemáticamente. La notación fue tomada de [47].

Definición 2.4.1 Una red neuronal recurrente (RNN) es un red neuronal que simula un sistema dinámico discreto que tiene una entrada x_t , una salida y_t y un estado oculto h_t que pueden ser vectores, de la siguiente manera:

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) \\ y_t &= f_o(h_t) \end{aligned}$$

en donde f_h es una función de transición de estado y f_o es la función de salida. Cada función se parametriza por un par de parámetros θ_h, θ_o .

Posteriormente procedemos como usualmente lo hacemos, a minimizar la función de costo a partir de una muestra de entrenamiento.

Sea $D = \{(x_1^{(n)}, y_1^{(n)}), \dots, (x_{T_n}^{(n)}, y_{T_n}^{(n)})\}_{n=1}^N$ un conjunto de N sucesiones de entrenamiento en donde cada muestra n se toma sobre el conjunto de tiempo $\{0, \dots, T_n\}$ (notar que esto implica que las sucesiones pueden ser de tamaños distintos, como observamos anteriormente).

Entonces, los parámetros θ_h, θ_o pueden ser estimados minimizando la siguiente función de pérdida:

$$J(\theta_h, \theta_o) = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} d(y_t^{(n)}, f_o(h_t^{(n)}))$$

en donde $h_t^{(n)} = f_h(x_t^{(n)}, h_{t-1}^{(n)})$ y $h_0^{(n)} = 0$ y $d(\cdot, \cdot)$ es una distancia en el espacio métrico \mathbb{R} (usualmente la euclídea o la entropía cruzada).

Definición 2.4.2 Una red neuronal recurrente convencional es un red neuronal recurrente (RNN) en donde las funciones de transición y de salida f_h, f_o se definen de la siguiente manera:

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) = \phi_h(\mathbf{W}^\top h_{t-1} + \mathbf{U}^\top x_t) \\ y_t &= f_o(h_t, x_t) = \phi_o(\mathbf{V}^\top h_t) \end{aligned}$$

En donde \mathbf{W}, \mathbf{U} y \mathbf{V} son las matrices de *transición*, de *entrada* y de *salida* respectivamente, y ϕ_h, ϕ_o son las funciones no-lineales usuales de las redes neuronales (usualmente ϕ_h es una sigmoide o la tangente hiperbólica). Entonces, los parámetros para esta red pueden ser estimados usando el algoritmo de *gradiente estocástico de descenso* (SGD) en donde el gradiente de la función de costo escogida, $J(\theta_h, \theta_o)$, se calcula usando retropropagación a lo largo del tiempo (estas técnicas ya fueron explicadas en la sección anterior).

Ya tenemos los rudimentos básicos para construir una red neuronal recurrente profunda (DRNN). Ahora explicaremos para qué la queremos y veremos cuatro maneras de construirlas a partir de la RNN.

De una red neuronal recurrente a una profunda

Según [47], *Deep Learning* tiene como hipótesis fundamental que un modelo profundo y jerárquico puede ser exponencialmente más eficiente al representar algunas funciones que uno superficial. Dicha hipótesis ha sido confirmada por varios estudios teóricos y empíricos. Por ejemplo, [38] mostró que redes profundas generativas pero estrechas (en específico las redes de creencia profunda, Deep Belief Networks) no requieren más parámetros que los modelos superficiales para ser aproximadores universales de funciones. Otro ejemplo teórico es el de [13] que mostró que una red profunda de suma-

producto puede requerir exponencialmente menos unidades para representar una función, que una red superficial de suma-producto. Un ejemplo práctico es el presentado por [31], en donde usando una técnica llamada *random dropout* (que omite una unidad oculta al azar en cada paso de entrenamiento) se logra sobrepassar el desempeño de modelos superficiales como las mezclas de Gaussianos. [47] hipotetizó que lo mismo podría ocurrirle a las redes neuronales recurrentes profundas.

Como explica [47], en una red neuronal prealimentada (feedforward neural network) la *profundidad* se define en términos del número de capas no lineales entre input y output. Debido a que las redes recurrentes contienen ciclos, la definición de *profundidad* no es análoga. Por ejemplo, cuando una RNN es *profunda* si se “desdobra” en el tiempo como en la Figura 2.15, pues un camino computacional (computational path) entre un input en un tiempo $i < t$ al output en un tiempo t cruza varias capas no lineales debido a los ciclos.

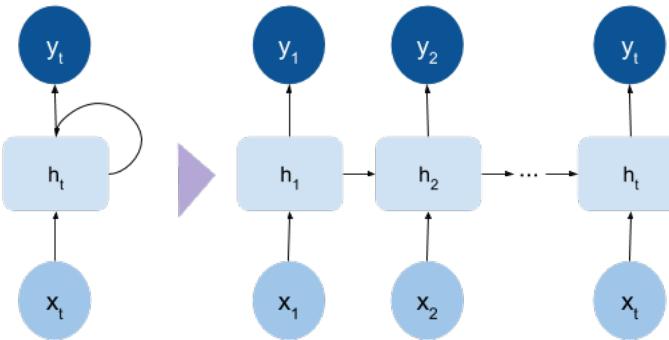


Figura 2.15: Red neuronal recurrente vista como una red profunda tomada de [47].

Sin embargo, según [47], un análisis detallado de lo que hace una RNN en cada tiempo individualmente, muestra que algunas transiciones no son profundas, sino más bien resultados de una proyección lineal seguida de una no-linealidad *elemento-a-elemento*. Es decir, en la Figura 2.15 puede verse que las transiciones $h_i \rightarrow h_{i+1}$, $x_{i+1} \rightarrow h_{i+1}$ y $h_{i+1} \rightarrow y_{i+1}$ son *superficiales* porque no hay capas intermedias ocultas no lineales.

Entonces [47] propone considerar esas transiciones por separado en 4 maneras diferentes de añadir *profundidad* a una red neuronal recurrente. En todos los casos la idea es agregar capas intermedias ocultas no lineales a lo largo de cada transición. A continuación se presenta la descripción de cada red y posteriormente se estudian las consecuencias que conlleva esta modificación.

Deep Input-to-Hidden Function: $x_t \rightarrow h_t$

En trabajos pasados se ha visto que representaciones de alto nivel de las redes profundas tienen a “desenredar” mejor que los inputs iniciales, los factores de variación subyacentes. [47] conjeturan que en el caso de las redes neuronales recurrentes, estas representaciones de alto nivel harían más fácil aprender la estructura temporal entre tiempos sucesivos porque la relación entre rasgos abstractos generalmente puede ser expresada más fácilmente. Los autores ponen como ejemplo los *word embeddings* de procesamiento de lenguaje natural (Natural Language Processing), que permiten relacionar vecinos cercanos (palabras) mediante operaciones algebraicas. Entonces, este enfoque está en sintonía con la práctica de reemplazar el input con rasgos extraídos que permitan mejorar el desempeño de los algoritmos de Aprendizaje Máquina.

La Figura 2.16 muestra cómo se vería esta arquitectura. $g(x_t)$ indica que se está transformando el input x_t de alguna manera.

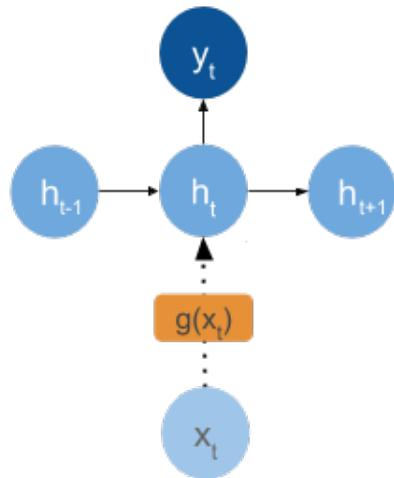


Figura 2.16: RNN agregando profundidad entre input y capa oculta

Deep Hidden-to-Output Function: $h_t \rightarrow y_t$

Añadirle profundidad a la transición de $h_t \rightarrow y_t$ puede ayudar a “desenredar” los factores de variación de la capa oculta de tal forma que sea más fácil predecir el output. También puede ayudar a resumir la historia de los inputs anteriores más eficientemente.

Para definir matemáticamente este modelo, retornemos a la **Definición 2.4.2** en donde se definió la capa de salida como $y_t = f_o(h_t) = \phi_o(\mathbf{V}^\top h_t)$. Como no hay restricción para la forma funcional de f_o [47] propone aproximar esta función mediante un perceptrón de L capas intermedias, pues tiene propiedades de aproximador universal, es decir:

$$y_t = f_o(h_t) = \phi_o(\mathbf{V}_L^\top \phi_{L-1}(\mathbf{V}_{L-1}^\top \phi_{L-2}(\dots \phi_1(\mathbf{V}_1^\top h_t))))$$

en donde ϕ_i es la función no-lineal elemento a elemento y \mathbf{V}_i es la matriz de pesos para la i -esima capa. Esta RNN con una función de transición formada por composiciones de funciones (multicapa) es llamada RNN con output profundo (DO-RNN).

La función f_o también puede ser un modelo generativo condicional como una Máquina de Boltzmann o un “neural autoregressive distribution estimator” como proponen [5] y [37].

La Figura 2.17 muestra cómo se vería esta arquitectura. El recuadro amarillo muestra en dónde estaría la modificación de la RNN para hacerla profunda, es decir, el perceptrón multicapa ϕ_o .

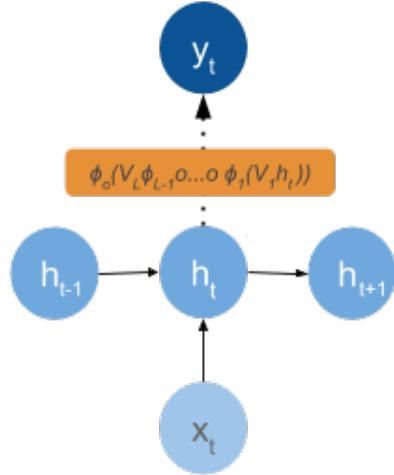


Figura 2.17: RNN agregando profundidad entre capa oculta y output

Deep Hidden-to-Hidden Transition: $h_t \rightarrow h_{t-1}$

También es posible añadirle profundidad a la transición entre capas ocultas. [47] argumenta que f_h debe construir un nuevo resumen que dependa de x_t, h_{t-1} debe ser altamente no-lineal. Esto permitiría que h_t se adapte rápidamente a cambios en el input pero preservando un resumen útil del pasado. Esta no linealidad naturalmente no puede ser modelada usando modelos lineales generalizados, sin embargo también se podría usar un perceptrón multicapa con una o más capas ocultas.

Para definir matemáticamente este modelo, retornemos a la **Definición 2.4.2** en donde se definió la capa oculta como $h_t = f_h(x_t, h_{t-1})$. Como no hay restricción para la forma funcional de f_h [47] propone aproximar esta función mediante un perceptrón de múltiples capas, es decir:

$$h_t = f_h(x_t, h_{t-1}) = \phi_h(\mathbf{W}_L^\top \phi_{L-1}(\mathbf{W}_{L-1}^\top \phi_{L-2}(\dots \phi_1(\mathbf{W}_1^\top h_{t-1} + \mathbf{U}^\top x_t))))$$

en donde ϕ_l es la función no-lineal elemento a elemento y \mathbf{W}_l es la matriz de pesos para la l -esima capa. Esta RNN con una función de transición formada por composiciones de funciones (multicapa) es llamada RNN con transición profunda (DT-RNN).

La Figura 2.18 muestra cómo se vería esta arquitectura. El óvalo amarillo representa el perceptrón multicapa.

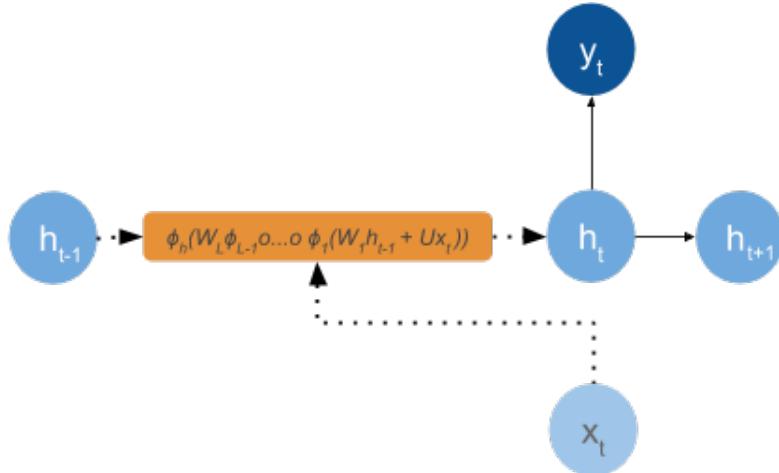


Figura 2.18: RNN agregándole transición profunda

Es importante notar que agregar transiciones profundas tiene un potencial problema: incrementa el número de pasos no lineales que el gradiente tiene que atravesar cuando se propaga a través del tiempo, lo cual puede ocasionar que el modelo no sea capaz de capturar dependencias a largo plazo [4]. Una alternativa es agregar conexiones que provean de atajos que el gradiente pueda tomar cuando es propagado hacia atrás en el tiempo. En la siguiente sección veremos un modelo que fue construido justamente para capturar estas dependencias a largo plazo, la Long Short Term Memory Network.

La Figura 2.19 muestra cómo se vería esta arquitectura si añadimos estos atajos. Las líneas punteadas rojas muestran los atajos.

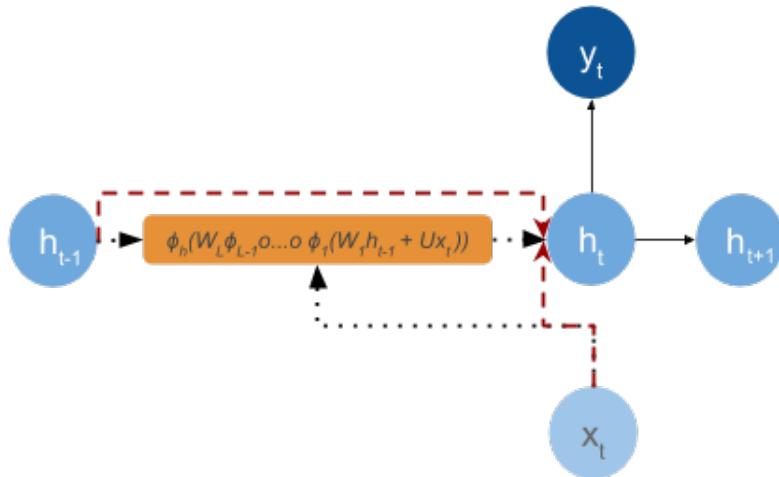


Figura 2.19: RNN agregándole transición profunda y atajos para el gradiente

También se pueden agregar profundidades en varios lugares de la red simultáneamente. La Figura 2.20 muestra cómo se vería una arquitectura con profundidad tanto en transición como en output.

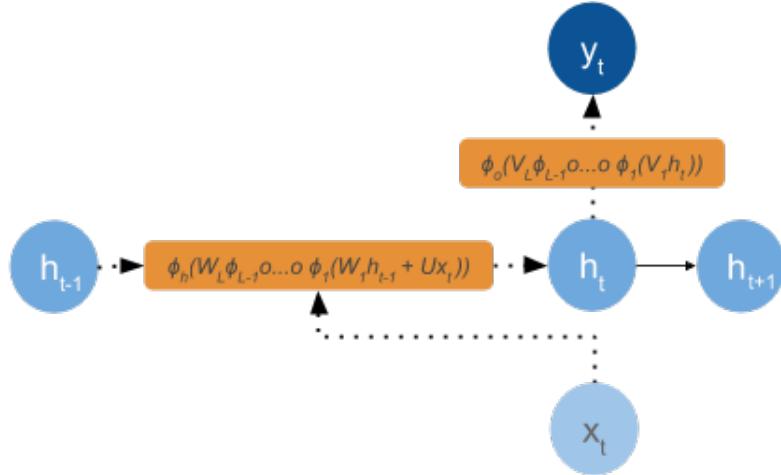


Figura 2.20: RNN agregándole transición y output profundos

Stack of Hidden States

A una RNN también se le puede agregar profundidad apilando múltiples capas recurrentes. Este modelo recibe el nombre de “stacked RNN”(sRNN). La idea es que cada capa recurrente opere en diferentes niveles.

La Figura 2.29 muestra cómo se vería esta arquitectura si añadimos una capa. Si observamos capa por capa, podemos ver que la transición del estado oculto $z_{i-1} \rightarrow z_t$ sigue siendo superficial, así que también se le podría añadir profundidad a esta transición mediante un perceptrón multicapa para que pueda representar más familias de funciones. Sin embargo, para que cada capa tuviera esta propiedad, se tendría que añadir un perceptrón por capa, lo cual podría complicar el modelo. En este sentido, podemos decir que el sRNN y el DT-RNN son ortogonales, pues podemos tener las propiedades de ambos en un mismo modelo. Por otro lado, este modelo tiene la ventaja de que para un input dado x_t , se pueden tener muchos niveles de recurrencia, lo cual no pasa en los modelos anteriores.

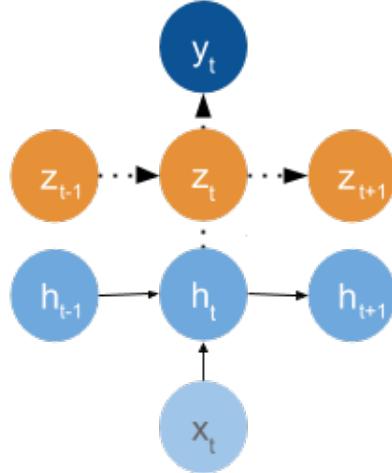


Figura 2.21: RNN apilada

Para definir matemáticamente este modelo, notemos que tiene múltiples niveles de funciones de transición definidos como:

$$h_t^{(l)} = f_h^{(l)}(h_t^{(l-1)}, h_{t-1}^{(l)}) = \phi_h(\mathbf{W}_l^\top h_{t-1}^{(l)} + \mathbf{U}_l^\top h_t^{(l-1)})$$

en donde $h_t^{(l)}$ es el estado oculto del l -esimo nivel en el tiempo. Observar que cuando $l = 1$ el estado se calcula usando x_t en vez de $h_t^{(l-1)}$. Los estados ocultos de los niveles se pueden calcular recursivamente a partir del estado base $l = 1$. Una vez que el estado más arriba es calculado, el output puede ser calculado de manera natural usando $y_t = f_o(h_t) = \phi_o(\mathbf{V}^\top h_t)$. Se pueden añadir atajos también en este modelo, por ejemplo, otra manera es usar todos los estados ocultos para calcular el output. O también cada estado oculto puede depender del input según [23].

2.4.4. Long Short Term Memory Networks

Ahora presentaremos la segunda propuesta alternativa: Long Short Term Memory Networks.

Motivación

Como ya vimos, las redes neuronales recurrentes son teóricamente fascinantes, pues son arquitecturas ricas y flexibles que permiten modelar problemas muy diversos. El problema de estas arquitecturas es su *entrenamiento*.

El método usual para entrenar las redes neuronales recurrentes es la propagación hacia atrás a través del tiempo (backpropagation through time o BPTT), que simplemente es una generalización de la propagación hacia atrás para redes tipo feed-forward a través del tiempo, ya que vimos que una red neuronal recurrente puede verse como una feed-forward si se desdobra en el tiempo. Otro método más computacionalmente intensivo es el Real-time Recurrent Learning (RTRL) [63]. Ambos métodos tienen las desventajas de que las señales del error que fluyen de atrás para adelante en el tiempo tienen *a explotar* o a *desaparecer*: la evolución temporal del error propagado hacia atrás depende exponencialmente del tamaño de los pesos. Cuando los errores *explotan* los pesos pueden oscilar y cuando *desaparecen* aprender dependencias largas toma un tiempo prohibitivo o no funciona [33] (o dicho de otra manera, el gradiente del error desaparece exponencialmente rápido con el tamaño del lag entre eventos importantes). Esto hace que las redes neuronales recurrentes no sean buenas incorporando dependencias temporales a grandes distancias (incluso si se inicializan los pesos aleatoriamente no podemos saber si un output dado va a depender de un input muy antiguo).

A veces esta dependencia no es relevante pero otras veces sí. Por ejemplo, en el procesamiento de lenguaje natural, si se entrenara una red neuronal recurrente para predecir la siguiente palabra en esta oración: “El cielo es _____”, entonces no existe dependencia de muchos lags hacia atrás, es decir, no es necesario entender mucho contexto para poder decir que la palabra faltante es *azul*. Sin embargo, si consideramos la siguiente oración “Entonces decidí que lo mejor era regresar a _____”, no es muy claro qué hay que poner en el espacio en blanco: ¿un lugar físico?, ¿un verbo?, ¿un país?, de modo que es claro que se necesita más contexto (i.e. dependen de más lags). Lo mismo puede ocurrir en series de tiempo: ¿Cuál es la dependencia de lags relevante en series de tiempo? ¿Hasta qué evento pasado es relevante considerar? Estas cuestiones no se saben a priori, pero una arquitectura que se comporte bien ante dependencias temporales grandes es muy deseable.

Las *Long Short Term Memory Networks* se introdujeron en 1997 en [33] para evitar estos problemas. Según los autores, esta arquitectura puede considerar datos con un lag en exceso de 1000 incluso en el caso de sucesiones de entrada ruidosas (de ahí el nombre de *long*) sin perder las capacidades de considerar también dependencias en el corto plazo (de ahí el nombre de *short*). Esto se logra con un algoritmo de descenso en gradiente eficiente que tenga un *flujo de error constante* (no explosivo ni desvaneciente) a lo largo de estados internos de unas unidades especiales y cortando el gradiente en ciertos

puntos específicos sin alterar el flujo del error [32].

De hecho, parte clave del éxito de las redes neuronales recurrentes para modelar problemas tan diversos como los que se han discutido, es el uso específicamente de las Long Short Term Memory networks (LSTM), que pueden verse como un caso especial de las RNN. Estas arquitecturas alcanzaron los mejores resultados en reconocimiento de escritura [24] y en 2009 ganaron la competencia de escritura a mano llamada ICDAR. También se han usado para detección automática de habla y eran parte esencial de la arquitectura que alcanzó un récord en 2013 con un error de 17.7% en el conjunto de datos clásico TIMIT. Como escribió Jürgen Schmidhuber en su blog, 2016 las compañías tecnológicas más importantes del mundo como Google, Apple, Microsoft y Baidu están usando LSTMs como parte fundamental de sus productos [54].

La Figura 2.22, muestra que la zona de la arquitectura que se modifica en la RNN para llegar a la LSTM está en las capas intermedias h_t .

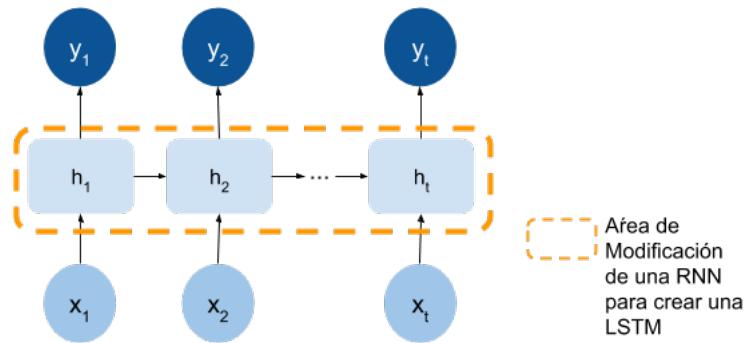


Figura 2.22: Modificación de una RNN para hacer una LSTM

Primer Arquitectura: Hochreiter y Schmidhuber, 1997

A continuación explicaremos la primer arquitectura de una LSTM introducida por Hochreiter y Schmidhuber en 1997. Esta sección fue enteramente tomada del artículo original: [33].

Posteriormente discutiremos algunas modificaciones que se hicieron hasta llegar a la LSTM que se usa actualmente.

Entrenamiento: BPTT usual

Primero conviene explicar a detalle a qué nos referimos con que los errores decaigan exponencialmente en una red neuronal recurrente si son entrenados usando propagación hacia atrás de los errores a través del tiempo (BPTT). Posteriormente explicaremos cómo es que se logra el flujo de error constante.

El output de la unidad k en el tiempo t se denota por $d_k(t)$. Usando error cuadrático medio, la k -ésima señal de error está dada por:

$$e_k(t) = f'_k(\text{net}_k(t))(d_k(t) - y^k(t)) \quad \text{con} \quad y^i(t) = f_i(\text{net}_i(t))$$

$y^i(t)$ es la activación de una unidad i que no es de entrada con función de activación diferenciable f_i .

$\text{net}_i(t)$ es el input de la red en la unidad i , w_{ij} es el peso de la conexión de la unidad j a la unidad i :

$$\text{net}_i(t) = \sum_j w_{ij} y^j(t-1)$$

Y finalmente la señal del error propagada hacia atrás de alguna unidad no de salida j es:

$$e_j(t) = f'_j(\text{net}_j(t)) \sum_i w_{ij} e_i(t+1)$$

La contribución a la actualización del peso w_{jl} está dada por $\alpha e_j(t)y^l(t-1)$, en donde α es la tasa de aprendizaje y l es una unidad arbitraria conectada a la unidad j .

En [33] se discute el análisis realizado por Hochreiter en 1991 para probar el decaimiento exponencial de los errores y es el siguiente. Suponer que tenemos una red totalmente conectada cuyos índices correspondientes a unidades de no entrada van de 1 a n . Enfoquémonos primero en el flujo de error local de la unidad u a la unidad v . BPTT nos dice que el error en una unidad arbitraria u en el tiempo t se propaga hacia atrás en el tiempo para q unidades de tiempo a una unidad arbitraria v . Esto modificará el error por el siguiente factor:

$$\frac{\partial e_v(t-q)}{\partial e_u(t)} = \begin{cases} f'_v(\text{net}_v(t-1))w_{uv} & q = 1 \\ f'_v(\text{net}_v(t-q)) \sum_{l=1}^n \frac{\partial e_l(t-q+1)}{\partial e_u(t)} w_{lv} & q > 1 \end{cases} \quad (2.4.1)$$

Haciendo $l_q = v$ y $l_0 = u$ obtenemos por inducción:

$$\frac{\partial e_v(t-q)}{\partial e_u(t)} = \sum_{l_1=1}^n \cdots \sum_{l_{q-1}=1}^n \prod_{m=1}^q f'_{l_m}(net_{lm}(t-m))w_{l_ml_{m-1}}$$

La ecuación anterior significa que si $|f'_{l_m}(net_{lm}(t-m))w_{l_ml_{m-1}}| > 1 \quad \forall m$ (por ejemplo con una función lineal f_{l_m}) entonces el producto más grande aumenta exponencialmente con respecto a q . Es decir, el error *explota* y señales de error que conflictúan y llegan a la unidad v pueden ocasionar pesos oscilatorios y aprendizaje inestable, tal como lo menciona [33]. Intuitivamente podríamos decir que si los pesos son grandes entonces el gradiente explota.

Por otro lado, si $|f'_{l_m}(net_{lm}(t-m))w_{l_ml_{m-1}}| < 1 \quad \forall m$ entonces el producto más grande *disminuye* exponencialmente con respecto a q . Esto es, el error se *desvanece* y el aprendizaje en un tiempo razonable es imposible. Intuitivamente podríamos decir que si los pesos son chicos, el gradiente desaparece.

Es importante mencionar que las redes tipo feed-forward pueden lidiar con estos problemas porque cuentan con unas pocas capas ocultas. Sin embargo, como una red neuronal recurrente es una red feed-forward con muchas capas ocultas, el efecto exponencial sí es significativo.

La Figura 2.23 siguiente tomada de *On the difficulty of training Recurrent Neural Networks* de Pascanu, Mikolov y Bengio [48] muestra gráficamente las dificultades anteriormente señaladas. Se muestra una red neuronal recurrente con una única capa oculta y se puede apreciar la existencia de paredes con alta curvatura. Las flechas azules sólidas muestran trayectorias estándar que el descenso en gradiente podría tomar. Cuando los gradientes explotan, la curvatura también explota en la misma dirección y tenemos una pared como la que se muestra en la Figura 2.23, y el gradiente se aleja completamente de la zona crítica. Las flechas azules punteadas muestran qué podría pasar si el gradiente se reescalara a un tamaño fijo cuando su norma es superior a un threshold (el flujo de error constante que mencionamos arriba).

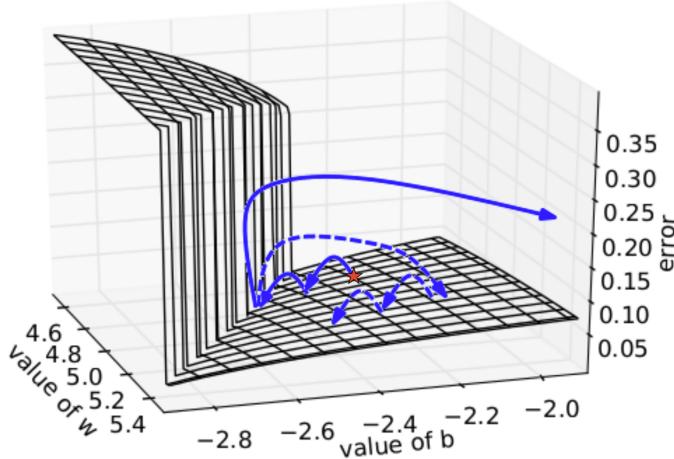


Figura 2.23: Superficie de error para una red neuronal recurrente simple

Alternativas de entrenamiento: preliminares

Para evitar señales de error que se desvanezcan, la pregunta es: ¿Cómo podemos tener un flujo de error constante a través de una unidad j con una única conexión a sí misma? Según las ecuaciones vistas anteriormente, en el tiempo t , el flujo de error hacia atrás de la unidad j es $e_j(t) = f'_j(\text{net}_j(t))e_j(t+1)w_{jj}$, de modo que para tener un flujo de error constante a lo largo de j requerimos que $e_j(t) = f'_j(\text{net}_j(t))w_{jj} = 1$. Se escoge la constante 1 porque sirve incluso para lags potencialmente infinitos.

Si se integra la ecuación anterior $e_j(t)$ anterior, nos queda que para un $\text{net}_j(t)$ arbitrario, $f_j(\text{net}_j(t)) = \frac{\text{net}_j(t)}{w_{jj}}$. O sea que f_j tiene que ser lineal y la activación de la unidad j tiene que ser constante. Esto [33] le llama *el carrusel del error constante*:

$$y_j(t+1) = f_j(\text{net}_j(t+1)) = f_j(w_{jj}y^j(t)) = y^j(t)$$

Esto se puede lograr haciendo que $f_j(x)$ sea la función identidad, es decir, que $\forall x$, $f_j(x) = x$ y $w_{jj} = 1$.

El problema con este enfoque es que supone que j solamente estará conectada consigo misma, pero j estará conectada con otras unidades, lo cual conlleva a los siguientes dos problemas (que no son exclusivos de este método, sino de todos los que usan gradiente)

descritos por [33]:

- Conflicto de peso que entra: Supongamos que el error total puede reducirse cambiando a la unidad j en respuesta a cierto input i y manteniendo activa j hasta que se pueda calcular el output deseado. Si i no es cero, dado que el mismo peso debe usarse para guardar ciertos inputs e ignorar otros, w_{ij} frecuentemente recibirá señales de actualización de peso que entran en conflicto (j es lineal) pues: estas señales tratarán de hacer que w_{ij} participe guardando el input (al cambiarse a j) y que proteja el input (previniendo que j se apague por inputs irrelevantes). Este conflicto hace que el aprendizaje sea difícil, y se requiere un mecanismo más sensible al contexto que ayude a controlar las *operaciones de escritura* a través de los *pesos de entrada*.
- Conflicto de peso que sale: asumir que j está activa y guarda un input previo. Sea w_{kj} un peso que sale. El mismo w_{kj} tiene que usarse tanto para obtener el contenido de j en ciertos tiempos, como para prevenir que j ofusque a k en otros. Mientras j sea distinta de cero, w_{kj} atraerá señales de actualización de peso contradictorias que se generarán cuando se procese la sucesión: estas señales tratarán en ciertos tiempos de hacer que w_{kj} participe tanto en accesar a la información guardada en j y en otros tiempos proteger a la unidad k de ser perturbada por j . Por ejemplo, [33] menciona que muchas tareas tienen errores que dependen de lags cortos que pueden reducirse en etapas tempranas del entrenamiento. Sin embargo, en etapas posteriores del entrenamiento j podría repentinamente empezar a causar errores que se pueden evitar en situaciones que parecían ya estar controladas al intentar participar en reducir errores difíciles con lags muy grandes. Igual que en el caso anterior, esto hace que el aprendizaje sea difícil, y nuevamente se requiere de un mecanismo para controlar las *operaciones de lectura* a través de los *pesos de salida*.

Más aún, los conflictos de pesos de entrada y salida no son exclusivos de lags grandes, sino que también ocurren con lags cortos. Sin embargo, los efectos son más perniciosos con los lags largos, pues a medida que el lag aumenta pasan dos cosas: 1. la información guardada debe protegerse contra perturbación por períodos más largos, 2. más outputs correctos requieren de protección contra la perturbación.

Debido a la discusión presentada, este enfoque no funciona en general. [33] introduce la *Long Short Term Memory* para sortear estas dificultades.

LSTM: primer arquitectura

Entonces queremos una arquitectura que permita un flujo de error constante mediante unidades especiales conectadas a sí mismas que no tengan las desventajas recién pre-

sentadas. Para lograr esto retomaremos el *carrusel del error constante* introducido en la sección anterior y le añadiremos características especiales.

Una **unidad de puerta de entrada** multiplicativa se introduce para proteger de perturbaciones por inputs irrelevantes a los contenidos de la memoria guardados en j . De la misma forma una **unidad de puerta de salida** multiplicativa se introduce para proteger a otras unidades de la perturbación por contenidos irrelevantes de la memoria guardados en j .

La unidad resultante se denomina **celda de memoria** y se muestra en la Figura 2.24 tomada directamente de [33].

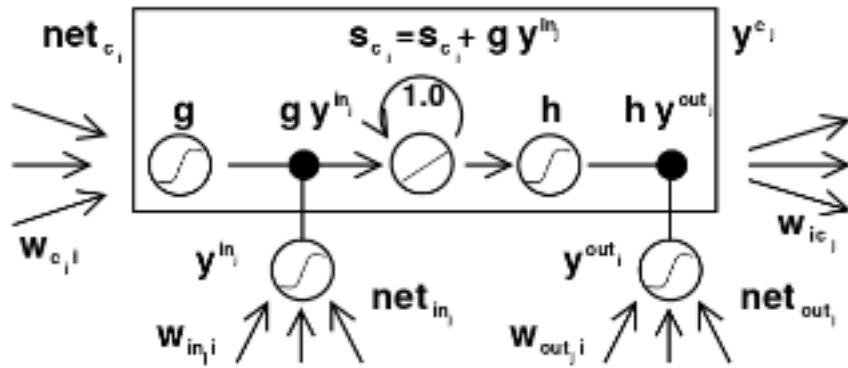


Figura 2.24: Celda de memoria c_j y sus puertas in_j , out_j , tomado de [33]

Cada *celda de memoria* c_j se construye alrededor de una unidad central lineal con una auto-conexión fija (el carrusel). Adicionalmente a net_{c_j} , c_j tiene como input una unidad multiplicativa out_j (la *puerta de salida*) y otra unidad multiplicativa in_j (la *puerta de entrada*). La activación de in_j y de out_j en el tiempo t se denota como $y^{in_j}(t)$ y $y^{out_j}(t)$, respectivamente. Entonces tenemos que para la *puerta de salida*:

$$y^{out_j}(t) = f_{out_j}(net_{out_j}(t))$$

en donde

$$net_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1)$$

Y por otro lado para la *puerta de entrada*:

$$y^{in_j}(t) = f_{in_j}(net_{in_j}(t))$$

en donde

$$net_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1)$$

Y adicionalmente

$$net_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1)$$

Los índices u puedes corresponder a unidades de input, unidades de puerta, celdas de memoria o incluso unidades ocultas si existieran. Todas estas unidades pueden tener información útil sobre el estado actual de la red. Un ejemplo es una puerta de entrada puede usar inputs de otras celdas de memoria para decidir si guarda cierta información en su celda de memoria. Incluso puede haber autoconexiones $w_{c_j c_j}$. La topología de la red es completamente customizable.

Al tiempo t , el output de a celda de memoria c_j , $y^{c_j}(t)$, se calcula como sigue:

$$y^{c_j}(t) = y^{out_j}(t)h(s_{c_j}(t))$$

En donde el *estado interno* $s_{c_j}(t)$ está dado por:

$$s_{c_j}(0) = 0, \quad s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t)g(net_{c_j}(t)), \quad t > 0$$

Entonces, la función diferenciable g aplasta net_{c_j} , la función diferenciable h escala los outputs de la celda de memoria que se calculan del estado interno s_{c_j} .

En resumen, necesitamos **puertas** porque para evitar conflictos de pesos de entrada, la puerta de entrada in_j controla el flujo de error a los pesos de entrada de celda de memoria c_j , es decir, $w_{c_j i}$. Por otro lado, para evitar los conflictos de pesos de salida, la puerta de salida, out_j , controla el flujo de error desde las conexiones de salida de la unidad j . Es decir, la red puede usar in_j para decidir cuándo *guardar o descartar* información en la celda de memoria c_j y out_j para decidir cuándo *acceder* a la celda

de memoria c_j y cuándo evitar que otras unidades sean perturbadas por c_j .

Naturalmente, señales de error atrapadas dentro del *carrusel de error constante* de una celda de memoria no pueden cambiar, pero diferentes señales de error fluyendo hacia la celda de memoria en diferentes tiempos vía la *puerta de salida* podrían estar superimpuestas. La puerta de salida tendrá entonces que aprender qué errores mantener en su carrusel, escalándolos apropiadamente. La puerta de entrada tendrá que aprender cuándo liberar esos errores, de igual forma escalándolos apropiadamente. Entonces, las puertas abren y cierran el acceso al flujo de error constante a lo largo del carrusel.

No siempre se requieren ambas puertas. [33] muestra un par de experimentos en donde es posible sólo usar puertas de entrada. Sin embargo menciona que incluir también puertas de salida ayuda al aprendizaje.

Para terminar la explicación de la arquitectura de la LSTM, [33] discute los siguientes puntos relativos a la configuración y optimización de la misma:

- **Topología de la red:** Usaron redes con una capa de entrada, una capa escondida y una capa de salida. La capa escondida está completamente conectada y contiene celdas de memoria y puertas. La capa escondida también puede contener unidades convencionales que funjan como inputs de las puertas y las celdas de memoria. Todas las unidades de todas las capas, a excepción de las puertas tienen conexiones dirigidas a todas las unidades en la capa de abajo o a todas las capas de arriba.
- **Bloques de celdas de memoria:** Al conjunto de S celdas de memoria que comparten la misma puerta de entrada y la misma puerta de salida se denominan “bloque de celdas de memoria de tamaño S ”. Estas estructuras facilitan el guardado de información. Dado que cada bloque tiene dos puertas, al igual que una única celda, entonces esta estructura puede ser más eficiente.
- **Aprendizaje:** Los autores usaron una variante del algoritmo RTRL, el cual captura apropiadamente la dinámica causada por las puertas de entrada y salida. No obstante, para asegurarse de no tener una propagación hacia atrás del error que decaiga a lo largo de los estados internos de las celdas de memoria como sucede en BPTT, los errores que llegan a los inputs de las celdas de memoria ($net_{c_j}, net_{in_j}, net_{out_j}$) no se propagan hacia atrás más, pero sí sirven para cambiar los pesos que vienen después. Solamente en las celdas de memoria los errores se propagan hacia atrás a través de los estados internos s_{c_j} . Es decir, una vez que una señal de error llega a un output de una celda de memoria, se escala mediante la activación de una puerta de salida y h' . Entonces en el carrusel de la celda de memoria puede fluir hacia atrás indefinidamente sin ser escalada. Sólo cuando deja la celda de memoria a través de la puerta de entrada y g , es escalada mediante la activación de la puerta de entrada y g' . Entonces sirve para cambiar los pesos

entrantes antes de que sea truncada.

- **Complejidad computacional:** Solamente se necesita guardar y actualizar las derivadas $\frac{\partial s_{c_j}}{\partial w_{il}}$, de tal forma que el algoritmo es muy eficiente con una complejidad de actualización de $\mathcal{O}(W)$, en donde W es el número de pesos. Entonces este algoritmo tiene la misma actualización de complejidad por unidad de tiempo que el *BPTT* para redes recurrentes completamente conectadas, mientras que el RTRL normal tiene mucho peor desempeño. Sin embargo este algoritmo tiene la ventaja de que es *local en el tiempo y en el espacio*, de modo que no es necesario guardar valores de activación observados durante el procesamiento de la sucesión en una pila de tamaño ilimitado.

LSTM: modificaciones a la primer arquitectura

Como hemos discutido, las LSTM en su forma original introducida por [33], sólo tenían dos tipos de puerta: de entrada y salida. Posteriormente se añadieron las **puertas que olvidan** (forget gates) por Gers et al. [17]. Estas puertas tienen el objetivo de que las celdas de memoria puedan resetearse a sí mismas cuando la red necesita *olvidar* ciertos inputs previos, liberando recursos. Sin estas puertas, [17] se encontró que el estado de las redes podía crecer indefinidamente y eventualmente fallar.

Por otro lado, las **conexiones tipo “peephole”** se añadieron para mejorar la habilidad de la LSTM para aprender tareas que requerían timing y conteo preciso de los estados internos [18]. Estas conexiones conectan la salida de la celda de memoria con los tres tipos de puertas (entrada, salida y olvido) y también tienen pesos asociados.

A la nueva red con estas dos modificaciones: *puerta de olvido* y *conexiones peephole* se le llama **modelo extendido**.

La Figura 2.25 tomada de [22] muestra un bloque típico con estas dos modificaciones. En medio se muestra la celda (cell). Las tres puertas: input, output y forget son unidades no lineales de suma con funciones squashing f , usualmente logísticas con rango en $[0,1]$ (cercano a 0 significa puerta cerrada y cercano a 1 puerta cerrada). Estas unidades reciben información desde fuera del bloque y también desde dentro mediante la celda usando las conexiones tipo “peephole”. Las puertas controlan las activaciones usando multiplicaciones (los círculos negros). Por ejemplo, la puerta de entrada multiplica el input ya sea por 0 ó 1 y así decide si entra o no el input a la celda. Análogamente la puerta de salida decide si el output sale del bloque o no. La puerta de olvido decide si multiplicar o no el estado anterior de la celda (es decir, decide si resetear la celda o no). Finalmente, las funciones g, h son las activaciones usuales del input y el output respectivamente y usualmente son tangentes inversas o logísticas. Notar que los únicos

outputs que salen del bloque hacia el resto de la red son las que salen por la puerta de salida y los únicos inputs que entran son los que pasan por la puerta de entrada. Finalmente, tomar en cuenta que un bloque puede tener varias celdas, esta Figura que mostramos sólo tiene una.

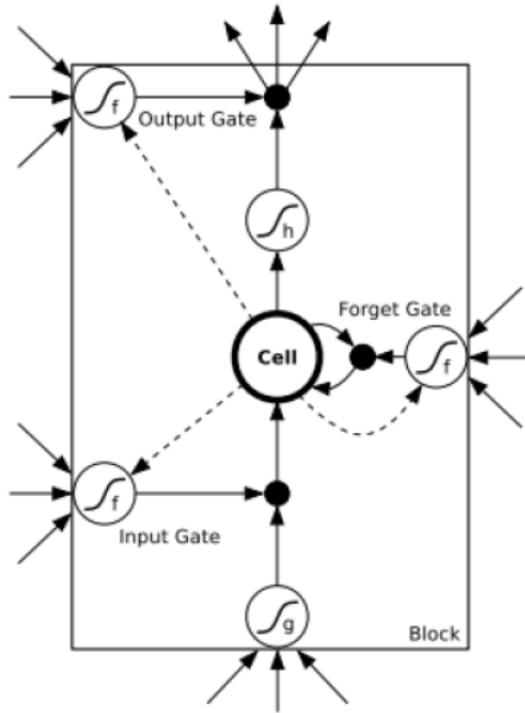


Figura 2.25: Bloque de memoria con una sola celda de una LSTM tomado de [22]

Otra visualización útil para entender la arquitectura de estas redes se muestra en la Figura 2.26 tomada de https://en.wikipedia.org/wiki/Long_short-term_memory, en donde ahora el bloque no se presenta como una figura aislada, sino que se incorpora un poco más a la arquitectura completa.

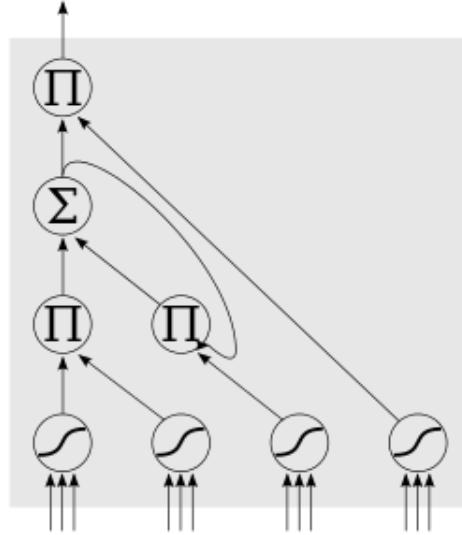


Figura 2.26: Bloque de memoria desagregado

También podemos ver la Figura 2.27 también tomada de https://en.wikipedia.org/wiki/Long_short-term_memory en donde se muestra una visualización horizontal de la arquitectura.

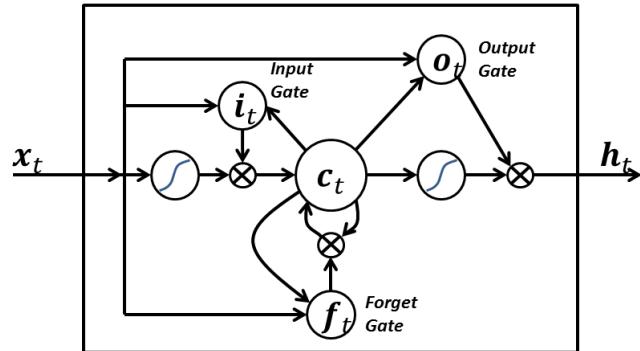


Figura 2.27: Vista de Arquitectura parcial horizontal

Debido a que el *bloque* es la única modificación a la RNN y que ésta se compone de unidades de suma y multiplicación y las conexiones entre ellas, entonces el modelo de las

LSTM networks es muy flexible, nuevas arquitecturas pueden ser creadas fácilmente. La Figura 2.28 tomada de [22] muestra un ejemplo de cómo se vería una LSTM network, es decir, incorporando el bloque de la Figura anterior a la red completa (pero no se muestran todas las conexiones). La red tiene cuatro unidades de input y cinco unidades de output. Cada bloque tiene cuatro inputs pero sólo un output.

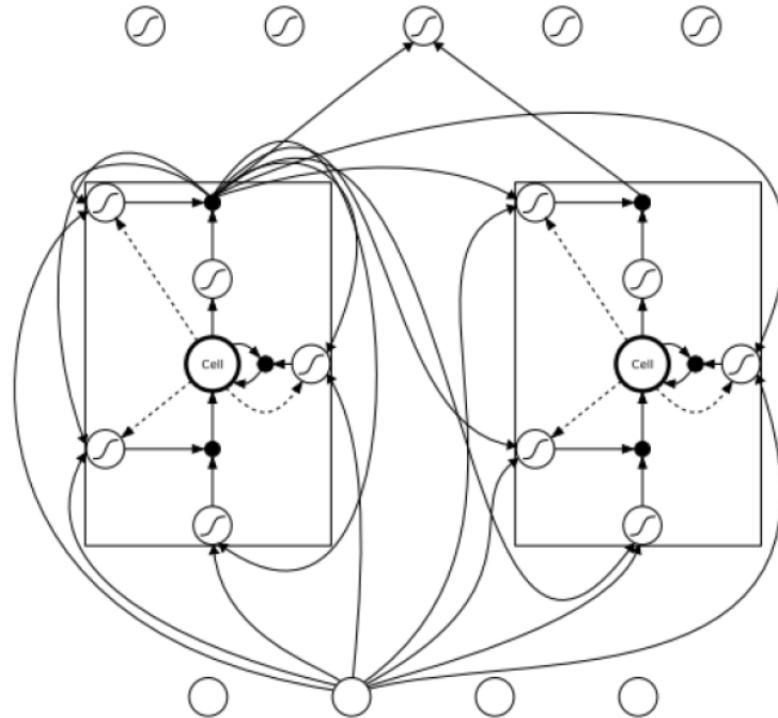


Figura 2.28: Una LSTM network tomada de [22]

Inclusive se han propuesto nuevos modelos para el bloque [22]. Sin embargo, [22] argumenta que el *modelo extendido* es suficiente para el etiquetado usual de sucesiones, así que ese es el modelo que usaremos.

Finalmente, la última modificación a la primer versión concierne al entrenamiento. Como vimos, en la primer versión de [33] el entrenamiento se hacía con una mezcla de ETRL y BPTT. Solamente el gradiente de la celda se propagaba a través del tiempo y el gradiente para otras conexiones recurrentes se truncaba, es decir, en esta versión no

se usaba el gradient exacto para el entrenamiento. La modificación se hizo en [25] en donde se usó el *BPTT completo* con las modificaciones arriba señaladas y se presentaron resultados sobre el dataset de TIMIT. Usar BPTT completo tenía la ventaja que los gradientes podían ser revisados usando diferencias finitas, lo que hizo que los resultados fueran más confiables [27].

2.4.5. Combinación de modelos: Deep LSTM

Las RNN y por lo tanto, las LSTM networks son arquitecturas profundas en el sentido de que pueden ser consideradas como redes tipo feed-forward cuando se desenrollan en el tiempo [53]. Sin embargo en las secciones anteriores vimos varias maneras de añadirle profundidad a una RNN en otro nivel y así construir *deep recurrent neural networks* [47].

La propuesta de este trabajo es combinar las dos alternativas que vimos (deep RNN y LSTM) para tener así tener una **deep LSTM**. Por un lado, usaremos una LSTM para evitar los problemas de las RNN con el gradiente que ya discutimos anteriormente. Por otro lado, usaremos el último enfoque mencionado para construir DRNN: *apilado* como la de la Figura 2.29 .

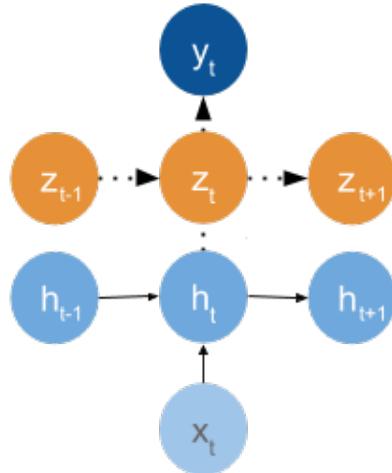


Figura 2.29: RNN apilada

Entonces, construiremos una **LSTM apilada**, es decir, tendremos una RNN apilada pero de tipo LSTM. Es decir, nos quedará algo como lo de la Figura 2.30.

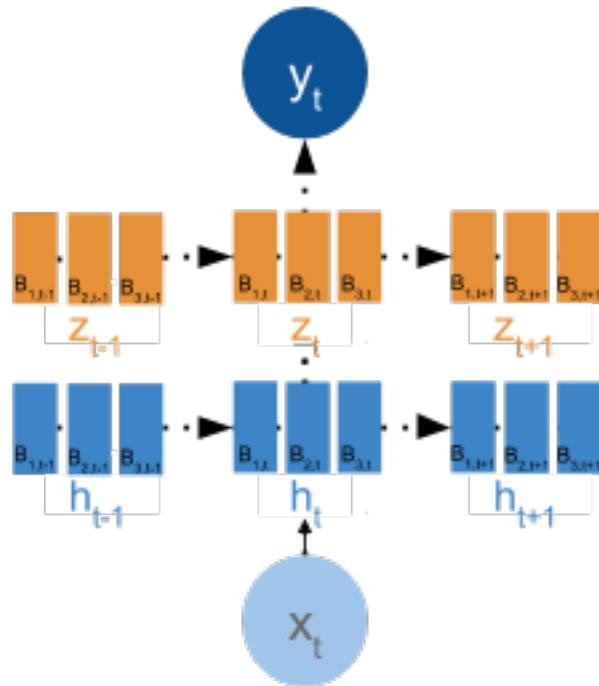


Figura 2.30: LSTM apilada con 3 bloques por nivel

Capítulo 3

Aplicación

The scientist describes what is; the engineer creates what never was.

Theodore von Kármán

La energía eléctrica puede ser fácilmente transportada y transformada en otros tipos de energía, sin embargo no puede ser almacenada a grandes escalas y consumida a demanda. Entonces, la energía eléctrica debe ser generada a una tasa casi igual a la que es consumida.

En caso de que la oferta de energía sea mayor a la demanda, entonces el diferencial se pierde, generando pérdidas. Según las cifras oficiales consultadas en la *Consulta de archivos de costo unitario y capacidad promedio prevista y realizada diaria* [10] de la Comisión Federal de Electricidad en México, tan sólo en un período de 10 años, del 1 de enero del 2006 al 1 de enero del 2015, se previó un consumo total de 146,265,254 MW de energía eléctrica, pero el consumo real fue de 139,907,400 MW, dando una pérdida de 6,357,854 MW.

Por otro lado, una demanda mayor a la oferta también tiene consecuencias desafortunadas, pues el diferencial puede causar que el equipo de generación y transmisión automáticamente se apaguen para prevenir daños, pudiendo en el peor de los casos terminar en apagones. A pesar de que en el período de 10 años señalado se tuvo una producción agregada diaria mayor al consumo real, esto no significa que no exista una subproducción para algunas de las más de 200 subestaciones distribuidas en el país,

que son las unidades encargadas de suministrar la energía a los poblados cercanos.

Entonces, contar con un sistema sofisticado de **predicción de la demanda de energía eléctrica** es necesario para asegurarse que la generación de energía eléctrica se asemeje lo más posible a la demanda.

3.1. La energía eléctrica en México

El Centro Nacional de Control de Energía (Cenace) es el organismo mediante el cual la Comisión Federal de Electricidad garantiza la seguridad, calidad y economía del suministro en el Sistema Eléctrico Nacional. Se hacen públicos los costos totales de corto plazo del sistema, las capacidades y costos unitarios previstos e incurridos del despacho de carga, de las centrales generadoras públicas y privadas que entregan energía al sistema para el suministro de la demanda en todo el país [9].

Según las Estadísticas Generales de [11], la Comisión Federal de Electricidad opera su abastecimiento en un esquema descentrado, en el que participan, aproximadamente, 361 centros compradores en todo el país, mismos que dependen jerárquicamente de las diferentes Direcciones de la Entidad. Asimismo, opera con un Comité Central de Adquisiciones, Arrendamientos y Servicios dependiente de la Dirección de Administración y conducido por la Gerencia de Abastecimientos, que tiene como principal función dictaminar sobre la procedencia de las excepciones a la licitación pública para las contrataciones que realice el Corporativo y para el caso de compras consolidadas. Adicionalmente, CFE cuenta 40 Comités Regionales, 32 dependientes de la Dirección de Operación y 8 que dependen de la Dirección de Proyectos de Inversión Financiada, que tienen como principal función dictaminar sobre la procedencia de las excepciones a la licitación pública para las contrataciones que realicen las unidades administrativas adscritas a cada área regional.

En [9] se muestran estadísticas sobre los siguiente rubros:

- Costos totales de corto plazo incurrido: Permite consultar y descargar la información correspondiente a los CTCP incurridos hasta el mes anterior.
- Costo unitario y capacidad promedio: Permite consultar y descargar la información correspondiente a los costos unitarios y capacidades promedio previstas y las realizadas al día.
- Consultas: Permite realizar una solicitud sobre los costos unitarios y capacidades previstos y realizados por central generadora.

- Documentos: Publicación de documentos varios, comunicados, avisos, etcètera, relacionados con los CTCP, capacidades y costos unitarios del despacho de carga.
- Comentarios y sugerencias sobre el funcionamiento de la sección Control y Despacho de Comentarios y sugerencias sobre la sección: Aquí podrás enviarnos algún comentario sobre la solicitud de información que registraste o bien alguna sugerencia sobre el funcionamiento de este módulo.

Se consultó el rubro *Costo unitario y capacidad promedio* para poder observar la oferta y demanda de energía eléctrica en México.

3.2. Descripción de la base

De [9] se descargaron más de 110 archivos en formato .xlsx correspondientes a más de 10 años de operación ya que no era posible descargar un archivo consolidado. Cada archivo contiene el consumo mensual real y previsto de energía eléctrica de cada una de las estaciones del país. Los archivos también tienen información sobre el costo previsto unitario previsto, sin embargo en la mayoría de los casos este dato tiene el valor de cero, así que no se pudo considerar para el análisis.

A continuación se muestra una vista del archivo correspondiente a julio 2010:

	A	B	C	D	E	F
1	Día	Central	Costo unitario previsto \$/MWh	Capacidad promedio prevista MW	Costo unitario realizado \$/MWh	Capacidad promedio realizada MW
2	1/1/2010	ACIPI	0	192	0	206.312
3	1/2/2010	ACIPI	0	192	0	217.761
4	1/3/2010	ACIPI	0	192	0	213.849
5	1/4/2010	ACIPI	0	192	0	211.218
6	1/5/2010	ACIPI	0	192	0	212.433
7	1/6/2010	ACIPI	0	192	0	211.036
8	1/7/2010	ACIPI	0	192	0	208.807
9	1/8/2010	ACIPI	0	192	0	210.899
10	1/9/2010	ACIPI	0	192	0	207.041
11	1/10/2010	ACIPI	0	192	0	204.773
12	1/11/2010	ACIPI	0	192	0	203.087
13	1/12/2010	ACIPI	0	192	0	203.887
14	1/13/2010	ACIPI	0	192	0	203.521
15	1/14/2010	ACIPI	0	192	0	202.2
16	1/15/2010	ACIPI	0	192	0	202.625

Figura 3.1: Ejemplo de Archivo de consumo y demanda descargado de [9]

Como se puede observar en la figura 3.1, los archivos tienen 6 columnas:

- **Día**: contiene el día en formato mm/dd/aaaa.
- **Central**: Muestra la central generadora y suministradora de la energía eléctrica.
- **Costo unitario previsto \$/MWh**: muestra el costo unitario previsto, sin embargo la mayoría de los registros tienen un valor de cero.
- **Capacidad promedio prevista (MW)**: muestra la capacidad promedio considerada medida en Megawatts y por lo tanto esta cantidad también corresponde a la energía eléctrica producida.
- **Costo unitario realizado \$/MWh**: muestra el costo unitario previsto, sin embargo la mayoría de los registros también tienen un valor de cero.
- **Capacidad promedio realizada MW**: muestra la energía eléctrica consumida medida en Megawatts, por lo tanto corresponde a la demanda real.

Entonces, las columnas consideradas fueron las siguientes: Día, Central, Capacidad promedio prevista (MW) y Capacidad promedio realizada (MW).

Los más de 110 archivos se convirtieron a *.csv* y se consolidaron en uno solo utilizando dos rutinas en Bash:

```
for i in `echo *.xlsx`; do
    libreoffice --headless --convert-to csv "$i" ;
done

for i in `echo *.csv`; do
    cat $i | sed "1~d" >> output.csv
done
```

El archivo consolidado *output.csv* tiene casi un millón de registros, para ser exactos 964,975 registros:

```
sophie@sophie-ThinkCentre-M93p:~/tesis_lic/tesis/data/data total$ cat output.csv
| wc -l
964975
```

Figura 3.2: Número total de registros en el archivo consolidado

3.3. Hardware y software

3.3.1. Hardware

Dado que el conjunto de datos final no es demasiado grande, entonces no es necesario el uso de GPUs, contrario a lo que se había creído al inicio.

Las características del hardware utilizado son las siguientes:

- Memoria: 7.6 GB
- Procesador: Intel® Core™ i7-4790 CPU @ 3.60GHz x 8
- Gráficos: Intel® Haswell Desktop
- Tipo Sistema Operativo: Linux de 64 bits
- Disco: 471.6 GB

3.3.2. Software

El software utilizado para hacer el modelo de Deep Learning fue *Keras*, una librería de Deep Learning para Theano y para Tensorflow: <https://keras.io/>.

Según el sitio web, *Keras* es una librería de alto nivel para redes neuronales. Está escrita en Python y es capaz de correr sobre Tensorflow o Theano. *Keras* se desarrolló para permitir la experimentación rápida con modelos de redes neuronales profundas, tales

como redes neuronales convolucionales o Long Short Term Memory Networks. Tras probar Theano y *Keras*, el autor se dio cuenta de que es mucho más fácil implementar un modelo en *Keras*, mientras que se necesitan varios cientos de líneas de código en Theano para implementar una red neuronal recurrente, en *Keras* se necesitan menos de 10. Sin embargo Theano es más personalizable y permite construir modelos más variados y al gusto del programador. No obstante, para el programador no experto en modelos de Deep Learning y para el objetivo y alcance de este trabajo se consideró que *Keras* era adecuado.

El sitio <https://keras.io/> menciona que es conveniente usar Keras si se necesita una librería de Deep Learning que:

- Permite una fácil y rápida implementación (modularidad, minimalismo y extensibilidad)
- Soporte redes neuronales convolucionales y recurrentes y combinaciones de estas
- Corra tanto en CPU como en GPU

Para comparar el modelo de Deep Learning se hicieron también dos modelos tradicionales usando un enfoque de Minería de Datos. Estos modelos fueron implementados en *Weka* versión 3.8 <http://www.cs.waikato.ac.nz/ml/weka/>. Weka es una librería de Java con modelos de Aprendizaje Máquina enfocados a tareas de Minería de Datos. Contiene herramientas para el preprocesamiento, clasificación, regresión, clusterización, series de tiempo y visualización. Es de código abierto y fue desarrollada en la Universidad de Waikato en Nueva Zelanda hace varios años y nuevas actualizaciones son generadas periódicamente (la última en abril del 2016 y es la que se usa en ese trabajo).

3.4. Preprocesamiento y análisis exploratorio

En la siguiente figura 3.3 se muestra el detalle de la Capacidad promedio realizada (azul) y la capacidad promedio prevista (naranja) y claramente se puede ver que diariamente hay una pérdida, la cual se puede observar en la línea gris y corresponde a los 6,357,854 MW agregados mencionados en la introducción de este trabajo.

Se observa un dato atípico entre enero 2007 y enero 2008 con valor de 25,708.346 MW que está claramente muy por debajo del promedio observado. No se sabe si hubo un error en la captura o el dato es real. Es importante que esta observación puede crear ruido en las predicciones y que un análisis más exhaustivo requeriría un tratamiento especial, sin embargo estas consideraciones quedan fuera de los objetivos de este

trabajo.

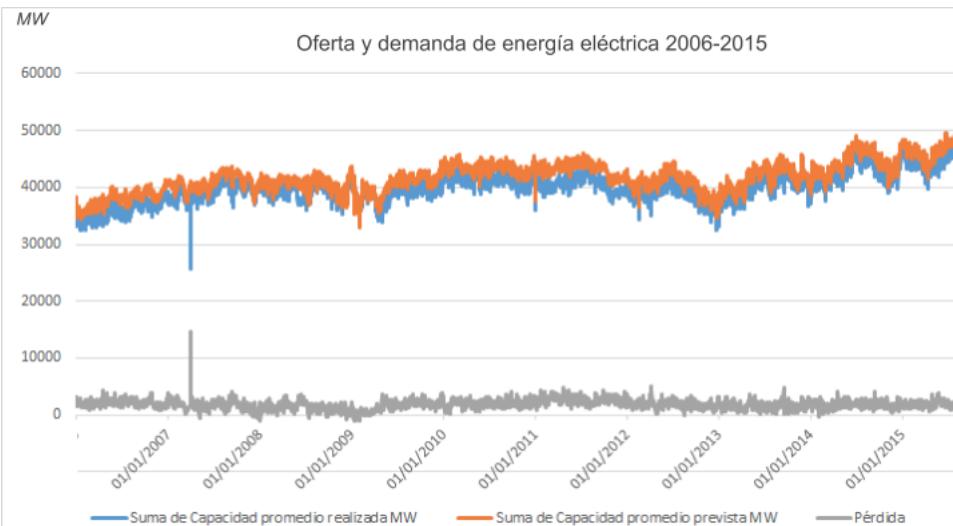


Figura 3.3: Oferta y demanda diaria total 2006-2015

Cada registro anterior tiene una Central asociada como se pudo observar en la Figura 3.1, con lo cual la Serie de Tiempo más bien es un Panel de Datos, pues se tienen observaciones temporales para diferente individuos, en este caso Centrales. Con el objetivo de ver si se podía hacer un modelo de predicción por estación se desagregó la base a nivel estación. Al desagregar la base y ver cuántas observaciones se tenían por cada central, se pudo observar que eran muy pocas, alrededor de 300 por estación, lo cual es muy poco para poder entrenar un modelo; incluso al graficar los datos para algunas estaciones aleatorias se podía observar un comportamiento muy errático. Por lo tanto, se tomó la decisión de que, dado que no se tienen suficientes observaciones por estación, el modelo predictivo se haría para los datos agregados diarios por un período de 10 años, es decir, alrededor de 350 datos por año que corresponden a la demanda diaria en todo el país. Esto nos da un total de 3,499 observaciones para entrenar el modelo, que si bien no es lo más deseado para entrenar un modelo de *Deep Learning*, aún así puede entrenarse el modelo y comparar los resultados con otros modelos. En el futuro se buscarán conjuntos de datos con un número mucho mayor de observaciones. La figura ?? muestra el total del consumo de energía eléctrica, pero ignorando el formato fecha, solamente enumerando las observaciones.

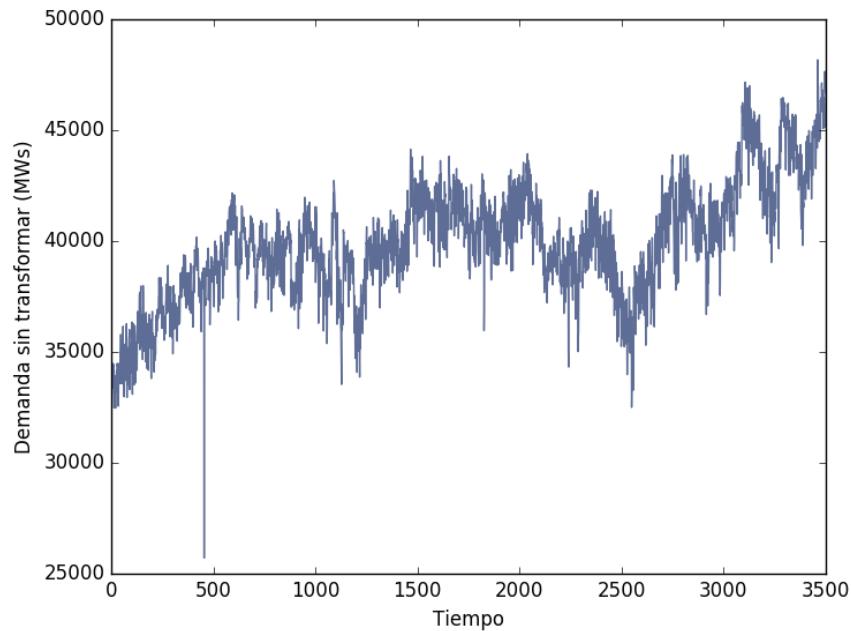


Figura 3.4: Serie de tiempo del consumo de energía eléctrica (sin transformar)

A continuación en la Figura 3.5 se muestra un histograma de los datos y puede verse un sesgo a la derecha, lo cual se esperaba ya que hay una tendencia ligeramente creciente:

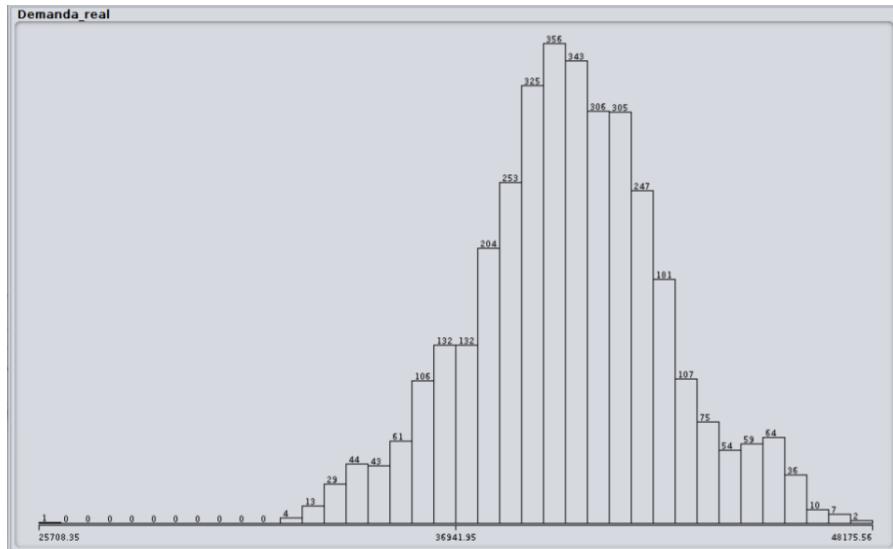


Figura 3.5: Histograma del consumo de energía eléctrica 2006-2015

Las estadísticas básicas son las siguientes:

Estadística	Valor
Mínimo	25,708.34
Máximo	48,175.56
Media	39,984.96
Desviación estándar	2,604.19

Cuando se modelan series de tiempo, es conveniente aplicar algunas transformaciones que permitan tres cosas:

- **Estabilizar la varianza:** esto se puede lograr aplicando la función \ln a todos los datos.
- **Hacer estacionarios los datos:** se refiere a lograr una varianza, medias y covarianzas constantes. Los modelos estadísticos para series de tiempo tienen como supuesto fundamental que los datos son estacionarios. Esta hipótesis no es fundamental en otros enfoques (como minería de datos), sin embargo aún así se recomienda hacer estacionarios los datos. Claramente la 3.3 muestra que esto no es así. Hay varias técnicas para lograr esto pero la que usamos fue aplicar *diferencias*. Con una diferencia bastó para estacionarizar los datos.

- **Quitar la tendencia:** se refiere a quitar las tendencias crecientes o decrecientes de la serie de tiempo. En la Figura 3.3 se puede observar que hay una ligera tendencia creciente (es natural, pues esperamos que la demanda de energía aumente con el paso del tiempo ya que la población tiende a hacerlo también). Generalmente al hacer los datos estacionarios también la tendencia se elimina.

Estas transformaciones se realizaron en el lenguaje de programación R. A continuación en la Figura 3.6 se muestra la gráfica de la serie ya transformada. Notemos que ahora la serie parece oscilar alrededor de una media constante y la varianza también se ve constante. sin embargo, nuevamente llama la atención la observación atípica mencionada anteriormente, la cual es el número 456 con un valor de 25,708.346 MW. Las transformaciones no sólo no lograron eliminar el ruido de esta observación, sino que además lo hicieron más notorio.

```
library(ts)
plot(diff(log(ts(new_elec))))
```

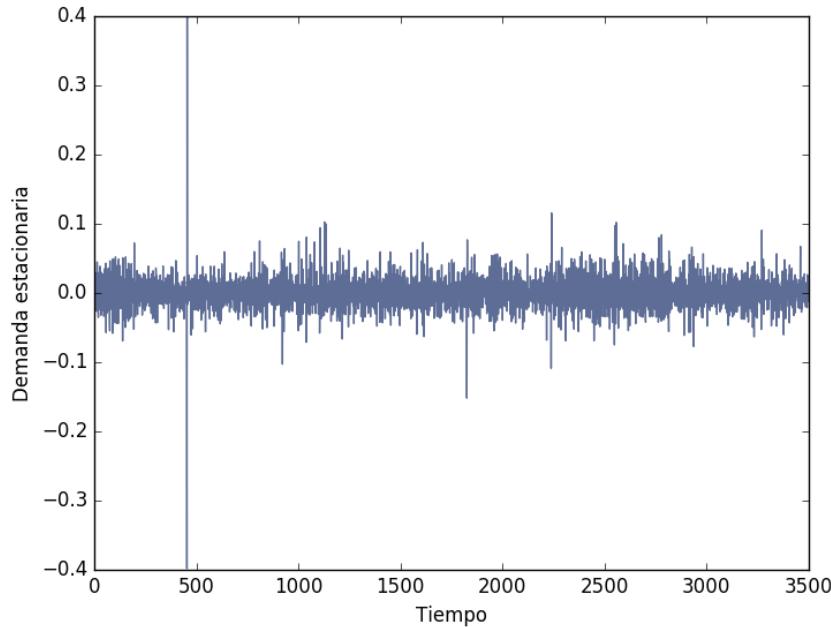


Figura 3.6: Serie de tiempo de consumo de energía eléctrica (estacionario)

Finalmente, para tener un criterio más robusto para la estacionariedad, aplicamos el **Test de Dickey-Fuller** para hacer una prueba de Hipótesis sobre la estacionariedad de la serie. La hipótesis nula es que “la serie no es estacionaria”. Entonces hay que ver si tenemos suficiente evidencia para rechazar la hipótesis nula y decir que “la serie es estacionaria”:

$$\mathbf{H_0 : La \ serie \ no \ es \ estacionaria} \quad \mathbf{H_1 : La \ serie \ es \ estacionaria}$$

La línea de código en R y los resultados del test son los siguientes y se puede ver que se rechaza la hipótesis nula, así que la serie es estacionaria:

```
#Prueba de hipotesis
adf.test (diff(log(ts(new_elec))), alternative="stationary", k=0)

#resultados
In adf.test (diff(log(ts(new_elec))), alternative = "stationary",
p-value smaller than printed p-value
Augmented Dickey-Fuller Test
data: diff(log(ts(new_elec)))
Dickey-Fuller = -64.804, Lag order = 0, p-value = 0.01
alternative hypothesis: stationary
```

Y finalmente, si volvemos a graficar el histograma de los datos, podemos ver que los datos casi forman una campana, lo cual muestra que ya no hay tendencia:

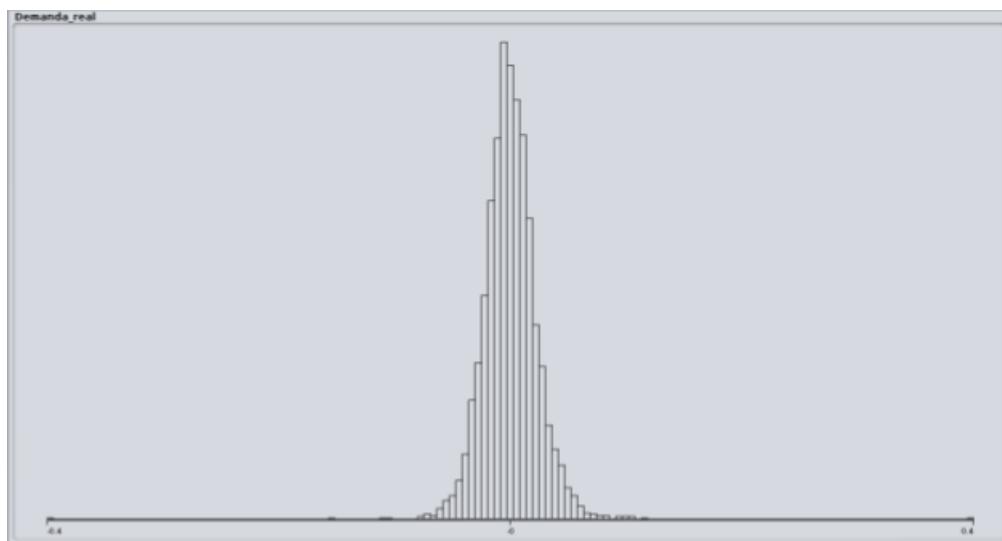


Figura 3.7: Serie estacionaria de consumo de energía eléctrica 2006-2015

Una vez que los datos están listos para aplicar el modelo, se procedió a decidir algún enfoque para seleccionar los conjuntos de **entrenamiento** y **prueba**. Debido a que estamos trabajando con una serie temporal -lo cual significa que hay correlación entre las observaciones (al menos temporal)-, esta tarea no es tan sencilla como lo es en instancias usuales de Aprendizaje Máquina, en donde se asume independencia entre las observaciones de tal forma que se puede hacer un corte puntual en entrenamiento y prueba, o bien usar validación cruzada. Hay diversas técnicas para llevar a cabo este corte, sin embargo acen fuera de los objetivos de este trabajo, así que como el objetivo del modelo era la predicción (i.e. a futuro), se tomaron los primeros datos para hacer el entrenamiento y los últimos para hacer la prueba. La distribución fue del usual 80 % y 20 %. Es decir, los primeros 2,800 datos para el entrenamiento y los restantes para prueba.

3.5. Modelos

Para la parte del modelo se consideraron dos clases: 1) Modelo tradicional y 2) Modelo de Deep Learning. Ambos modelos se compararon con los datos actuales (sin modelo), que corresponden a la proyección hecha y reportada por al Comisión Federal de Electricidad.

La medida del error en los dos modelos fue la raíz cuadrada del error cuadrático medio (RMSE por sus siglas en inglés):

$$RMSE = \sqrt{MSE} = \sqrt{E[(y - \hat{y})^2]}$$

3.5.1. Sin modelo

Corresponde a la columna *Capacidad promedio prevista (MW)* de la base de datos y representa la predicción hecha por la CFE y la producción real.

3.5.2. Modelo tradicional

Como se mencionó arriba, se usó la librería de Weka para el enfoque tradicional de minería de datos. Se probaron dos modelos:

- Regresión lineal múltiple
- Redes neuronales superficiales: un perceptrón de una sola capa

Los modelos de minería de datos no consideran observaciones dependientes, es decir, en donde el valor de una variable para cierto individuo depende del valor en otro individuo. Las series de tiempo naturalmente no son independientes: se asume que el valor de la serie temporal en t depende del valor de la serie en $t - 1$, $\forall t > 0$.

Entonces, para poder usar un enfoque de minería de datos en series temporales univariadas es necesario eliminar el orden temporal por renglones. Es necesario reacomodar la base de datos para que el problema pueda formularse como un problema de *regresión*. Para esto será necesario agregar *variables retrasadas* (lagged en inglés):

$$y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \dots + \alpha_m y_{t-m}$$

Por lo tanto, cada renglón se tiene que reescribir de esta manera. Además posiblemente será necesario añadir ciertas variables para quitar la tendencia de la serie. Entonces se puede proceder a resolver el modelo como un problema usual de regresión ya sea usando regresión lineal múltiple o redes neuronales superficiales.

Weka hace esa reescritura de la base de datos automáticamente y sólo es necesario configurar los parámetros de cada modelo.

3.5.3. Modelo de Deep Learning

Para poder entrenar una red neuronal, ya sea superficial o profunda, hay que escalar los datos en $(0, 1)$. Hay varias maneras de hacer esto, la transformación que se usó fue:

$$X_{std} = \frac{(X - X.\min)}{(X.\max - X.\min)}$$

Que es una alternativa a la transformación de media cero y varianza unitaria. Una vez definida la transformación, la aplicamos a los datos.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

La Figura 3.8 muestra las observaciones escaladas.

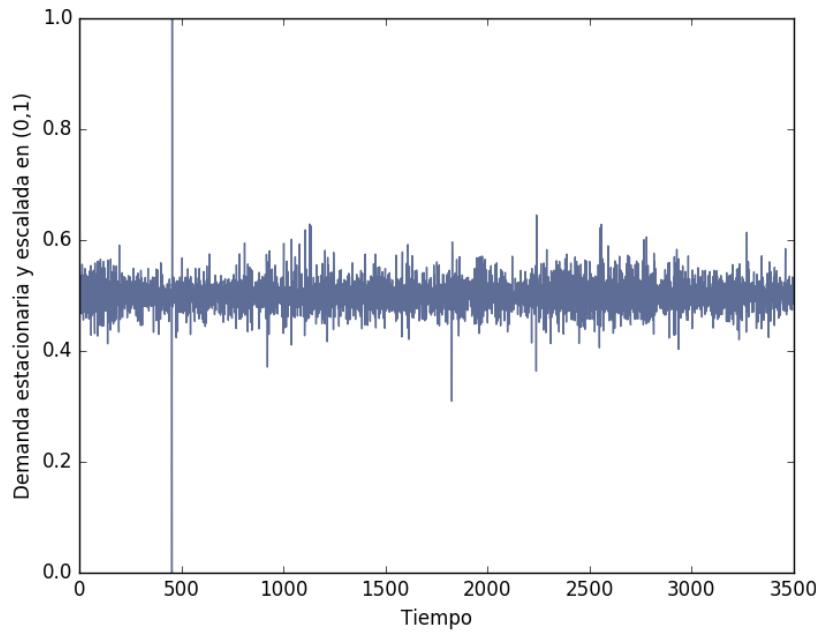


Figura 3.8: Serie de tiempo de consumo de energía eléctrica (estacionaria y escalada)

Para aplicar un modelo de *Deep Learning* en una serie de tiempo, primero es necesario reacomodar los datos, pues el racional para modelar series de tiempo con modelos de *Deep Learning* es parecido al de minería de datos: debemos transformar nuestro problema de un problema de series de tiempo a uno de regresión considerando las variables retrasadas de nuestra variable objetivo. A diferencia de Weka, que automáticamente reacomoda los datos, Keras no lo hace. Una función en Python para hacer esto que tiene como parámetro el número de lags que queremos considerar es:

```
import numpy
import pandas

def lagged_dataset(dataset , look_back=1):
    dataX , dataY = [] , []
    for i in range(len(dataset)-look_back-1):
        a = dataset[ i :( i+look_back) , 0]
        dataX.append(a)
        dataY.append(dataset[ i + look_back , 0])
    return numpy.array(dataX) , numpy.array(dataY)
```

Esta función nos regresa por un lado los predictores (las variables retrasadas) y por otro la variable objetivo (la variable sin retrasar).

Como se mencionó, se usó *Keras*¹ para construir el modelo de Deep Learning. En la sección anterior se discutió que se usaría una **Deep Long Short-Term Memory Network**.

Keras está construida por los siguientes módulos:

- **MODELOS.** Existen dos maneras de construir modelos en Keras: los secuenciales (Sequential) y la clase que se usa con la API. Los últimos son muy flexibles: dado un tensor de entrada y uno de salida se puede construir un modelo. En este trabajo se usarán los secuenciales, pues provee una estructura sencilla para construir modelos recurrentes. Para ver más a detalle la documentación, así como ver los métodos en común de ambos modelos, consúltese <https://keras.io/models/about-keras-models/>.

El modelo *Sequential* (ver más en <https://keras.io/models/sequential/>), es una pila de lineal de capas y se le van añadiendo capas parametrizables, por ejemplo: *core* (activation, dense, dropout, merge) y ya propiamente *arquitecturas* como: recurrentes, convolucionales (incluyendo pooling), embeddings, normalizadoras, de

¹<https://keras.io/>

ruido, etc. Dentro de las recurrentes están las LSTM, GRU-Gated Recurrent Unit y SimpleRNN. Para ver todas las capas, consultar <https://keras.io/layers/about-keras-layers/>.

En nuestro modelo usaremos: **core (activation y dense)** para la activación de las unidades y para el output, y **recurrentes (LSTM)** para la arquitectura. Para ver la documentación de la clase *recurrente*, ver <https://keras.io/layers/recurrent/>.

Finalmente, para disminuir el *overfitting* que las arquitecturas profundas presentan se usó la técnica de dropout introducida por [58]. La idea es seleccionar aleatoriamente ciertas unidades y eliminarlas incluyendo sus conexiones durante el entrenamiento. Se tiene que seleccionar la proporción de unidades a eliminar: *keras.layers.core.Dropout(p)*.

1. **Dimensiones de los datos:** en nuestro ejemplo la longitud del input es del tamaño del **lag** a considerar y la dimensión del input es de 1, pues es una serie de tiempo univariada.
2. **Compilación:** aquí se escoge el método de *optimización* a usar (gradiente estocástico, rmsprop, adagrad, customizable, etc). Para ver la lista completa ver <https://keras.io/optimizers/>. Aquí también se escoge la función de pérdida, por ejemplo para problemas de clasificación *categorical_crossentropy* o *mse* para problemas de regresión. Escogimos **Mean Squared Error**, pues nuestro problema es de series de tiempo, es decir, de **regresión**. Para ver otras funciones de pérdida, ver <https://keras.io/objectives/>. Finalmente se pueden agregar métricas para evaluar el modelo, como la precisión en problemas de clasificación, ver <https://keras.io/metrics/>.
3. **Entrenamiento:** se entrena el modelo usando la función *fit*. Como parámetros hay que incluir los datos (son los predictores), etiquetas (son los valores a predecir), número de épocas (*nb_epoch*), tamaño del lote (*batch_size*). En nuestro caso, los datos son *trainX* (las m variables retrasadas $y_{t-1}, y_{t-2}, \dots, y_{t-m}$), las etiquetas son las variables sin retraso, y_t que se guardaron en *trainY* y serán las que se buscará predecir. El número de épocas fue de 100 y el tamaño del lote fue de 1.

A continuación se muestra el código en Keras de lo anterior:

```
#LSTM en Keras para modelar serie de tiempo univariada
#Paquetes
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
```

```
from keras.layers import Dropout
#Modelo
model = Sequential()
model.add(LSTM(5, input_length=lag, input_dim=1))
#Optimizacion
model.compile(optimizer='rmsprop', loss='mse')
#Entrenamiento
model.fit(trainX, trainY, nb_epoch=1,
batch_size=1, verbose=2, shuffle=False)
```

- **OTROS.** *Callbacks, Preprocesamiento, Datasets, Aplicaciones, Backend, Inicializaciones, Regularizadores, Restricciones, Visualización, Scikit-learn API y utils:* funcionalidades adicionales para construir un modelo más robusto, interactivo y con posibilidad de integrarlo a un pipeline de producción. Para ver más de estas funcionalidades, revisar <https://keras.io/getting-started/sequential-model-guide/>.

El código completo se puede encontrar en el **Apéndice**.

Capítulo 4

Resultados

The absence of evidence is not the evidence of absence.

Carl Sagan

En el capítulo pasado se explicaron los dos tipos de modelo que se usarían, modelo tradicional con Minería de Datos y modelo de Deep Learning. En esta sección mostramos y discutimos sus resultados.

4.1. Modelo tradicional

4.1.1. Regresión lineal

La forma funcional del modelo lineal multivariado es de la Figura 4.1 siguiente. La variable objetivo es la demanda real de electricidad sin retraso. Se puede ver cómo Weka añade regresores al modelo: 12 variables retrasadas hasta doce unidades de tiempo (lineales), índices artificiales de tiempo (polinomios de primer, segundo y tercer grado), así como combinaciones de productos de variables retrasadas con índices de artificiales de tiempo. Entonces, el modelo sólo es lineal con respecto a las variables retrasadas.

```

Scheme:
    LinearRegression -S 0 -R 1.0E-8 -num-decimal-places 4

Lagged and derived variable options:
    -F Demanda_real -L 1 -M 12

Relation:     demanda_energia_est
Instances:   3498
Attributes:  1
              Demanda_real

Transformed training data:

Demanda_real
ArtificialTimeIndex
Lag_Demanda_real-1
Lag_Demanda_real-2
Lag_Demanda_real-3
Lag_Demanda_real-4
Lag_Demanda_real-5
Lag_Demanda_real-6
Lag_Demanda_real-7
Lag_Demanda_real-8
Lag_Demanda_real-9
Lag_Demanda_real-10
Lag_Demanda_real-11
Lag_Demanda_real-12
ArtificialTimeIndex^2
ArtificialTimeIndex^3
ArtificialTimeIndex*Lag_Demanda_real-1
ArtificialTimeIndex*Lag_Demanda_real-2
ArtificialTimeIndex*Lag_Demanda_real-3
ArtificialTimeIndex*Lag_Demanda_real-4
ArtificialTimeIndex*Lag_Demanda_real-5
ArtificialTimeIndex*Lag_Demanda_real-6
ArtificialTimeIndex*Lag_Demanda_real-7
ArtificialTimeIndex*Lag_Demanda_real-8
ArtificialTimeIndex*Lag_Demanda_real-9
ArtificialTimeIndex*Lag_Demanda_real-10
ArtificialTimeIndex*Lag_Demanda_real-11
ArtificialTimeIndex*Lag_Demanda_real-12

```

Figura 4.1: Forma del modelo de regresión lineal múltiple en Weka

Los regresores significativos son todos las variables retrasadas, así como algunas combinaciones de índices y variables retrasadas, además de una constante. Llama la atención la dificultad para interpretar el modelo: por ejemplo, todos los lags tienen asignados un peso negativo, lo cual podría interpretarse como que el consumo de energía eléctrica en el pasado afecta negativamente el del presente. Sin embargo esto es contraintuitivo con el hecho de que el consumo tiende a aumentar con el tiempo, hallazgo que discutimos en la sección final del capítulo anterior. Esto podría explicarse porque algunas combinaciones de índices y variables retrasadas tienen asignados pesos positivos y porque la constante es positiva. sin embargo, la explicación de fondo es que los datos están transformados, entonces para tener una idea real de si la predicción tiene una tendencia creciente tendríamos que transformar los datos a las variables originales. No obstante, como podemos ver, los regresores del modelo no son tan sencillos de interpretar.

```

Demanda_real:
Linear Regression Model
Demanda_real =
-0.5136 * Lag_Demanda_real-1 +
-0.3873 * Lag_Demanda_real-2 +
-0.3346 * Lag_Demanda_real-3 +
-0.2588 * Lag_Demanda_real-4 +
-0.2696 * Lag_Demanda_real-5 +
-0.2033 * Lag_Demanda_real-6 +
-0.087 * Lag_Demanda_real-9 +
-0.1223 * Lag_Demanda_real-10 +
-0.1187 * Lag_Demanda_real-11 +
-0.1537 * Lag_Demanda_real-12 +
-0 * ArtificialTimeIndex^2 +
0 * ArtificialTimeIndex^3 +
0.0001 * ArtificialTimeIndex*Lag_Demanda_real-1 +
0 * ArtificialTimeIndex*Lag_Demanda_real-3 +
0.0001 * ArtificialTimeIndex*Lag_Demanda_real-7 +
-0 * ArtificialTimeIndex*Lag_Demanda_real-8 +
0.0008

```

Figura 4.2: Regresores significativos del modelo de regresión lineal múltiple en Weka

Los resultados del modelo de regresión se muestran en la Figura 4.3 y puede verse que el *RMSE* es de 0,0202 para el conjunto de entrenamiento de 2,798 datos y de 0,0189 para el conjunto de prueba de 700 datos.

```

==== Evaluation on training data ====
Target 1-step-ahead
=====
Demanda_real
N 2786
Root mean squared error 0.0202

Total number of instances: 2798

==== Evaluation on test data ====
Target 1-step-ahead
=====
Demanda_real
N 700
Root mean squared error 0.0189

Total number of instances: 700

```

Figura 4.3: Resultados del modelo de Weka

4.1.2. Red neuronal superficial

Los parámetros de la red neuronal fueron los siguientes. Sólo se usó una capa intermedia para hacer énfasis en que no es una arquitectura profunda.

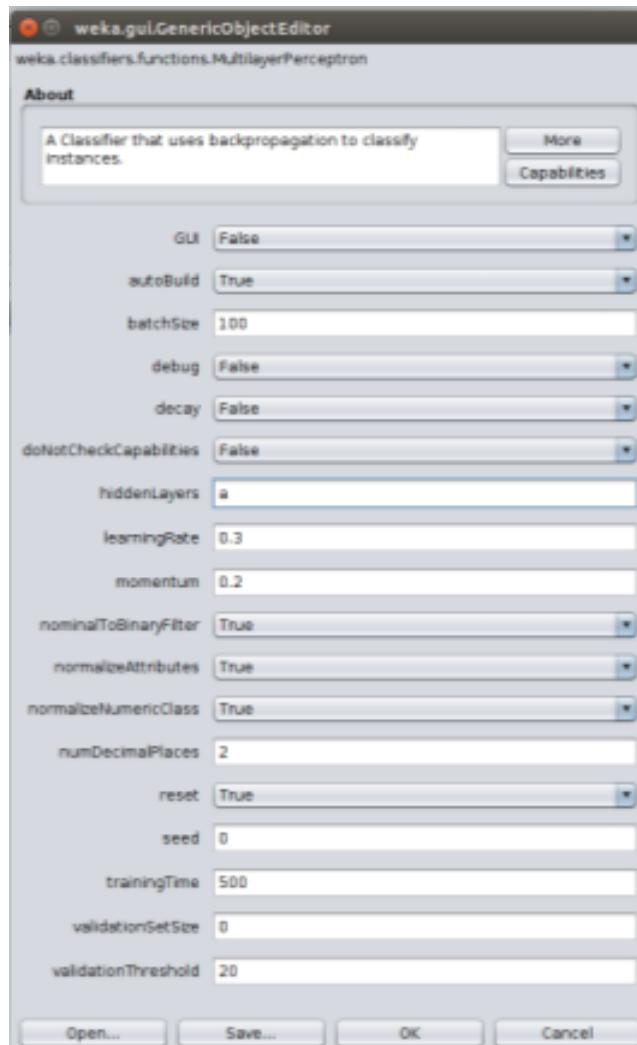


Figura 4.4: Parámetros de la red neuronal (interfaz de Weka)

Los resultados de la red superficial fueron de 0,0277 para los datos de entrenamiento y de 0,0279 para el de prueba, lo cual muestra que el error no empeoró mucho y que el modelo es relativamente bueno generalizando.

```

    === Evaluation on training data ===
    Target          1-step-ahead
    =====
    Demanda_real
      N           2786
      Root mean squared error   0.0277

    Total number of instances: 2798

    === Evaluation on test data ===
    Target          1-step-ahead
    =====
    Demanda_real
      N           700
      Root mean squared error   0.0279

    Total number of instances: 700
  
```

Figura 4.5: Resultados de la red neuronal superficial en Weka

4.2. Modelo de Deep Learning

No se sabe a priori cuál es la combinación óptima de parámetros para una LSTM y tampoco hay un método de optimización que los calcule directamente. Así que para hacer el *tunning* de la LSTM se tomaron en cuenta las siguientes variables:

- **Dropout(p):** Técnica razonablemente general para evitar el overfitting que hacen las redes neuronales profundas. Según [58] (y como ya habíamos mencionado en el capítulo anterior de *Aplicación*), esta técnica consiste en desechar aleatoriamente unidades y sus conexiones durante el entrenamiento. El parámetro $0 \leq p \leq 1$ se refiere al porcentaje de unidades que desecharemos. Por supuesto, deseamos valores distintos de los extremos, pues si $p = 0$ entonces no desecharemos nada (no usamos *dropout*); y si $p = 1$ entonces estaríamos desecharando todas las unidades, es decir, no tendríamos unidades en la capa considerada.
- **Lag(m):** Retraso considerado, es decir, cuántas variables considerar como predictores de la variable objetivo X_t (consumo de energía eléctrica en t). O sea, hay que determinar cuál es el valor óptimo de $m \in \mathbb{N}$ en la siguiente ecuación:

$$X_t = X_{t-1} + X_{t-2} + \dots + X_{t-m}.$$

- **Stateful(True/False):** Se refiere a si estamos usando el último estado de cada muestra en el índice i en un batch como el estado inicial para la muestra i en el batch siguiente. Esta variable es booleana. Entonces stateful=true indica que sí se tiene esta memoria, y Stateful=false indica que no. Por default en Keras esta variable es falsa. Ver <https://keras.io/layers/recurrent/> para más documentación.
- **Número de bloques por capa:** Se refiere a cuántos bloques consideraremos en cada capa de la LSTM. Recordar la Figura 4.6 del capítulo 2, Sección 2.4.4 Long Short Term Memory Networks, en donde podemos observar que sólo hay dos bloques en la capa intermedia.

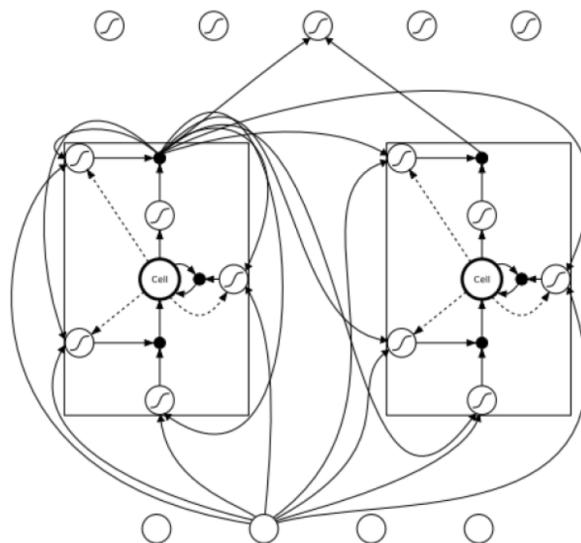


Figura 4.6: Una LSTM network tomada de [22]

- **Tamaño del batch:** Se refiere al tamaño B del lote $\{i_1, i_2, \dots, i_B\}$ del gradiente estocástico. Conforme B aumenta se llega en menos iteraciones al óptimo, sin embargo cada iteración es más costosa. La línea rosa de la Figura 4.7 muestra trayectorias de las direcciones de descenso usuales de un batch de tamaño B . Entre más pequeño sea B , más erráticas serán. La línea roja podría representar el gradiente estocástico tomando todos los datos, claramente se llega en menos itera-

ciones al óptimo que cuando tomamos un tamaño de batch pequeño, pero corremos el riesgo de sobreajustar. Por eso consideraremos tamaños de batch pequeños en el tuning de la LSTM.

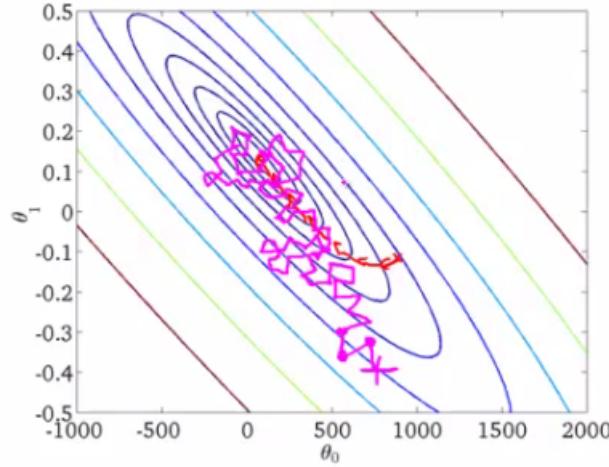


Figura 4.7:

- **Número de épocas:** Se refiere al número de pases completos a los datos.
- **Número de capas:** Profundidad. Como comentamos en el capítulo anterior de *Aplicación* construiremos un modelo profundo apilando varias LSTM como lo muestra la Figura 4.8. Esta red tiene 2 LSTM apiladas. El nivel de profundidad, como vimos, depende de cómo se calculen las operaciones de la red.

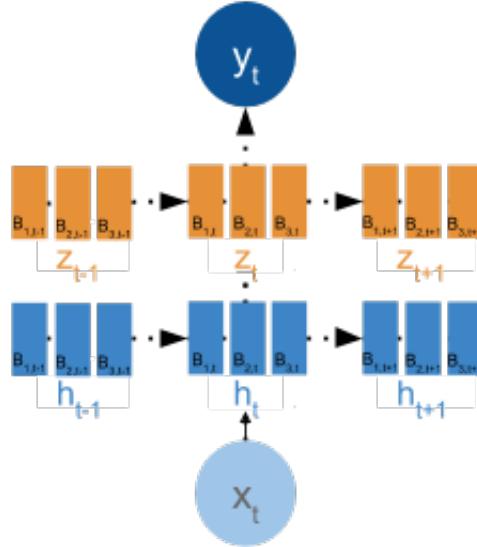


Figura 4.8: LSTM apilada

- **Semilla:** Se usa la misma para garantizar la reproducibilidad de los experimentos.

Lo primero que se hizo fue fijar la semilla. La semilla escogida para todos los experimentos fue: 103596 (la clave única del autor).

Posteriormente se partió de un caso base y se fueron variando alguno de los parámetros anteriores *ceteris paribus*. Idealmente se tendrían que variar varios parámetros a la vez, sin embargo lo que se hizo es regresar a hacer *tunning* nuevamente a ciertos parámetros para los que no se había tenido un resultado contundente.

Finalmente, agregamos a cada corrida el número de parámetros totales (pesos) que tiene que estimar cada LSTM con la configuración dada de variables.

4.2.1. Caso base

El caso base considerado se pensó tomando parámetros “conservadores”, y a partir de este se irán variando los parámetros:

Lag	Batch	Bloques	Capas	Stateful	Dropout	Épocas	train RMSE	test RMSE	Paráms
3	1	4	1	sí	no	100	0.1955	0.19459	101

El caso base se irá actualizando conforme tengamos resultados y se pondrá como comparador en el primer renglón siempre.

4.2.2. Dropout(p)

Para ver si usando Dropout(p) el sobreajuste disminuye y por lo tanto vemos una ganancia en el RMSE en el conjunto de prueba, se probó con $p = 0,5$ y se agregó `model.add(Dropout(0,5))` al código después de construir la LSTM.

Como se puede apreciar en la tabla, el error aumentó considerablemente, de 0.19459 a 0.51033. Se probó también con $p = 0,2$ y $p = 0,8$ pero los resultados fueron los mismos.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochas	train RMSE	test RMSE	Paráms
3	1	4	1	sí	no	100	0.1955	0.19459	101
3	1	4	1	sí	sí, $p = 0,5$	100	0.51081	0.51033	101
3	1	4	1	sí	sí, $p = 0,2$	100	0.51081	0.51033	101
3	1	4	1	sí	sí, $p = 0,8$	100	0.51081	0.51033	101

Vimos que *Dropout* es una técnica razonablemente general para evitar el overfitting que hacen las redes neuronales profundas, y al revisar el artículo original [58] de Srivastava et al. (2014) se encontró que el tamaño del dataset tiene un efecto en la efectividad del dropout. La Figura 4.9 muestra que para datasets muy pequeños el dropout no sólo no ayuda, sino que perjudica y el error (en este caso de clasificación) incrementa al usar Dropout. Los autores explican que el modelo tiene suficientes parámetros y muy pocos datos y puede hacer overfitting en el conjunto de entrenamiento, a pesar del ruido añadido por el Dropout. Es decir, aplicar Dropout en conjuntos de datos muy pequeños no evita que la red sobreajuste. Nuestro conjunto de datos, al ser tan pequeño, cae en este caso.

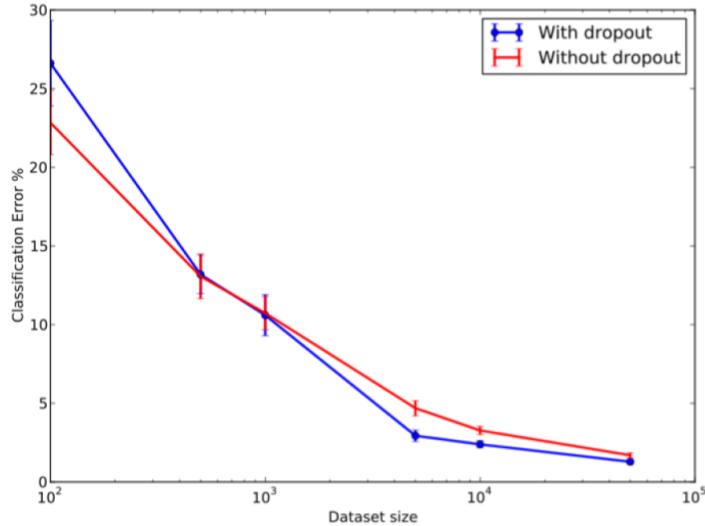


Figura 4.9: Efecto del tamaño del dataset en el error tomado de [58]

Como trabajo futuro, conviene revisar el trabajo de Gal et al. publicado en 2015 [16] en donde se discute cómo es que la técnica de *Dropout* ha fallado cuando se aplica a redes recurrentes. Usando técnicas bayesianas se explica el concepto de *Dropout* y se proveen insights de cómo se debería usar esta técnica para redes recurrentes. Las modificaciones al *Dropout* usual mejoraron el caso base para redes recurrentes.

4.2.3. Lag

Ahora se procedió a variar el tamaño del lag. Podemos ver que cuando se hace $lag = 9$ se alcanza el mejor desempeño *ceteris paribus*.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochas	train RMSE	test RMSE	Paráms
3	1	4	1	sí	no	100	0.1955	0.19459	101
6	1	4	1	sí	no	100	0.19535	0.19455	101
12	1	4	1	sí	no	100	0.1953	0.19456	101
9	1	4	1	sí	no	100	0.1953	0.19448	101
5	1	4	1	sí	no	100	0.19539	0.19459	101
4	1	4	1	sí	no	100	0.19543	0.1946	101
2	1	4	1	sí	no	100	0.19554	0.1945	101

4.2.4. Stateful

Se cambió el parámetro *Stateful* a falso y el error disminuyó. Se tenía un verdadero para aumentar la memoria entre batches de la red. No se tiene una explicación en términos de la arquitectura LSTM de porqué es mejor tener *Stateful = False*.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochas	train RMSE	test RMSE	Paráms
9	1	4	1	sí	no	100	0.1953	0.19448	101
9	1	4	1	no	no	100	0.18717	0.1863	101

4.2.5. Número de bloques por capa

Ahora variamos el número de bloques que tendremos en cada capa. Como podemos observar, el mejor desempeño se alcanza para *Bloques = 6*. Notar también cómo cambia el número de parámetros. Cuando consideramos muy pocos bloques, por ejemplo 2, el error empeora considerablemente.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochas	train RMSE	test RMSE	Paráms
9	1	4	1	no	no	100	0.18717	0.1863	101
9	1	5	1	no	no	100	0.09628	0.09506	146
9	1	6	1	no	no	100	0.0593	0.0572	199
9	1	7	1	no	no	100	0.13066	0.12997	260
9	1	2	1	no	no	100	0.059893	0.59891	35

Para el caso en donde *Bloques = 6*, tenemos la siguiente Figura 4.10 que muestra el número de parámetros a optimizar, es decir, los pesos y sesgo de la red:

```

    print('El numero de parametros en cada capa es: ')
>>> _____
Layer (type) ..... Output Shape ..... Param # ..... Connected to
=====
lstm_9 (LSTM) ..... (1, 6) ..... 192 ..... lstm_input_9[0][0]
=====
dense_9 (Dense) ..... (1, 1) ..... 7 ..... lstm_9[0][0]
=====
Total params: 199
=====
>>> ..... Desescalando los datos...
>>> ..... Evaluando modelo en entrenamiento y prueba:
>>> Train Score: 0.05930 RMSE
>>> Test Score: 0.05720 RMSE

```

Figura 4.10: Número de parámetros para 6 bloques

Notar que si ahora cambiamos Stateful=true el modelo mejora:

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochs	train RMSE	test RMSE	Paráms
9	1	6	1	no	no	100	0.0593	0.0572	199
9	1	6	1	sí	no	100	0.0414	0.03823	199

Se volvió a generar el experimento con *Stateful = True* cambiando el número de bloques para ver si se podía encontrar alguna explicación del comportamiento del parámetros Stateful y se encuentra que en general es mejor tener Stateful=False, menos para el caso de Bloque=6.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochs	train RMSE	test RMSE	Paráms
9	1	6	1	sí	no	100	0.0414	0.03823	199
9	1	2	1	sí	no	100	0.6206	0.62054	35
9	1	3	1	sí	no	100	0.48907	0.48899	64
9	1	4	1	sí	no	100	0.1953	0.19448	101
9	1	5	1	sí	no	100	0.17823	0.17749	146
9	1	7	1	sí	no	100	0.12611	0.12541	260

4.2.6. Tamaño del batch

Como explicamos en la sección de arriba, aumentar tamaño del batch lleva más rápido al óptimo (en número de iteraciones) para un mismo número de épocas. Sin embargo se corre el riesgo de sobreajustar. Mantendremos el tamaño del batch lo más pequeño posible, y como nuestro conjunto de datos es pequeño, realmente no vemos un impacto en el tiempo de corrida del algoritmo.

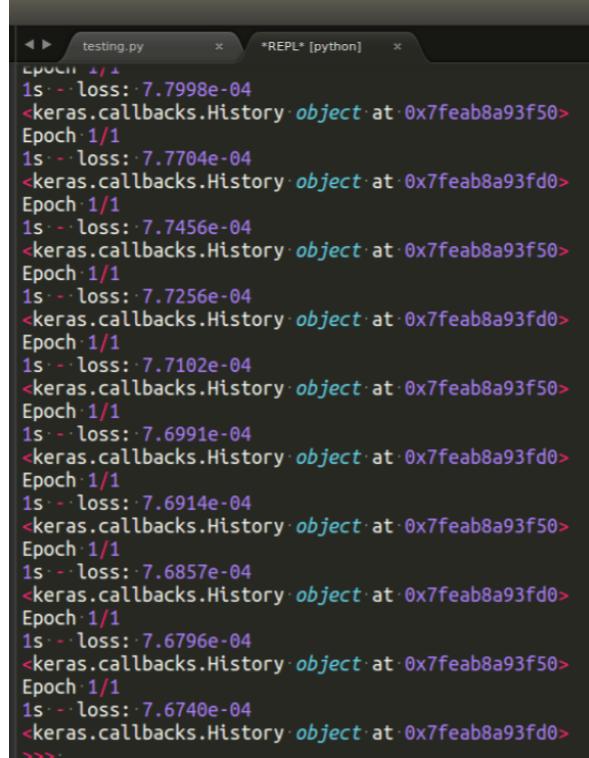
Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochs	train RMSE	test RMSE	Paráms
9	1	6	1	sí	no	100	0.0414	0.03823	

4.2.7. Número de épocas

Finalmente podemos ver que a mayor número de épocas el error mejora, pues la estimación es más precisa al hacer más pases completos por los datos.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochs	train RMSE	test RMSE	Paráms
9	1	6	1	sí	no	100	0.0414	0.03823	
9	1	6	1	sí	no	200	0.01987	0.01812	199

En la Figura 4.11 se muestra un fragmento de cada una de las épocas.



The screenshot shows a Jupyter Notebook interface with two tabs: 'testing.py' and '*REPL* [python]'. The code in 'testing.py' is an LSTM training loop. It prints the epoch number (1/1) and the loss value for each of 10 epochs. The loss starts at 7.7998e-04 and decreases steadily to 7.6740e-04. The output ends with a '...>' indicating more data.

```
Epoch 1/1
1s - loss: 7.7998e-04
<keras.callbacks.History object at 0x7feab8a93f50>
Epoch 1/1
1s - loss: 7.7704e-04
<keras.callbacks.History object at 0x7feab8a93fd0>
Epoch 1/1
1s - loss: 7.7456e-04
<keras.callbacks.History object at 0x7feab8a93f50>
Epoch 1/1
1s - loss: 7.7256e-04
<keras.callbacks.History object at 0x7feab8a93fd0>
Epoch 1/1
1s - loss: 7.7102e-04
<keras.callbacks.History object at 0x7feab8a93f50>
Epoch 1/1
1s - loss: 7.6991e-04
<keras.callbacks.History object at 0x7feab8a93fd0>
Epoch 1/1
1s - loss: 7.6914e-04
<keras.callbacks.History object at 0x7feab8a93f50>
Epoch 1/1
1s - loss: 7.6857e-04
<keras.callbacks.History object at 0x7feab8a93fd0>
Epoch 1/1
1s - loss: 7.6796e-04
<keras.callbacks.History object at 0x7feab8a93f50>
Epoch 1/1
1s - loss: 7.6740e-04
<keras.callbacks.History object at 0x7feab8a93fd0>
...>
```

Figura 4.11: Épocas de la LSTM en Keras

4.2.8. Número de capas

Ahora se trabajará con la profundidad de la red.

Con dos capas, se muestran los resultados del entrenamiento en la Figura 4.12 y podemos ver que este modelo es suficiente para mejorar al modelo de regresión lineal.

Se puede ver que al agregar Dropout de 0.8 en la primer capa, el desempeño no empeora demasiado, a diferencia de lo que veíamos más arriba.

Lag	Batch	Bloques	Capas	Stateful	Dropout	Epochs	train RMSE	test RMSE	Paráms
9	1	6	1	sí	no	200	0.01987	0.01812	199
9	1	6	2	sí	no	200	0.02005	0.01855	511
9	1	6	2	no	no	200	0.01969	0.01836	511
9	1	6	2	no	sí, p=0.8	200	0.02090	0.01845	511
9	1	6	3	no	sí, para RNN	200	0.02469	0.02032	823
9	1	6	5	no	sí, para RNN	100	0.02468	0.02031	1447
9	1	6	5	no	sí, para RNN	200			1447
9	1	6	5	sí	sí, para RNN	200			1447

```
>>> >>> ...: ...: ...: ...: ...: ...: >>> >>> ...: Desescalando los datos...
>>> >>> >>> >>> ...: >>> Evaluando modelo en entrenamiento y prueba:
>>> >>> Train Score: 0.02027 RMSE
>>> >>> Test Score: 0.01860 RMSE
```

Figura 4.12: Resultados de la LSTM en Keras

Podemos notar que el orden en el desempeño de los modelos con respecto a el RMSE es el siguiente:

1. Modelo de Deep Learning: $RMSE = 0,0186$
2. Modelo de Regresión Lineal $RMSE = 0,0189$
3. Modelo de red neuronal superficial $RMSE = 0,0279$

Creemos que el desempeño del modelo de Deep Learning es bastante bueno considerando que sólo se usaron 3,500 datos. También creemos que su desempeño puede mejorar considerablemente si se incrementa el tamaño del conjunto de datos, pues de esta manera la arquitectura tendría oportunidad de modelar dependencias temporales mucho más complicadas, lo cual se espera que sea así por la habilidad de los modelos de deep learning para modelar conceptos abstractos y jerárquicos.

4.3. Impacto de los resultados

Capítulo 5

Conclusiones y trabajo futuro

Live as if you were to die tomorrow.
Learn as if you were to live forever.

Mahatma Gandhi

En este trabajo se dio un recorrido general de Deep Learning. Se pueden mencionar los siguientes hallazgos:

- Deep Learning tiene larga y rica historia desde hace varias décadas.
- El incremento del poder computacional, del número de datos y de la complejidad de los problemas que se desean resolver ha permitido e incentivado su uso.
- Deep Learning ha tenido resultados sin precedentes en varias áreas y problemas muy complejos, sin embargo su uso en industria y gobierno es limitado.
- El fundamento teórico de Deep Learning es limitado y aún no se ha logrado incorporar de manera coherente, sólida y natural a la teoría del aprendizaje.
- Una razón de porqué los algoritmos de Deep Learning funcionan mejor que los algoritmos usuales de Aprendizaje Máquina en problemas difíciles, es porque la profundidad de la red permite aprender representaciones abstractas y jerárquicas de los datos, y de esta manera es posible construir conceptos complejos a partir de simples.

- Entonces, Deep Learning asume que los datos fueron generados por la composición de factores o rasgos, potencialmente en múltiples niveles de jerarquía y se construye una *representación distribuida* de los datos. Esto permite una *ganancia exponencial* en la relación entre el número de datos y en número de regiones que se pueden distinguir. Esta ganancia es un contrapeso a los retos que impone la maldición de la dimensionalidad.
- Deep Learning es capaz de representar una función compleja más eficientemente que los algoritmos de aprendizaje máquina usuales. Así, Deep Learning permite generalizar de manera no-local, en nuevos datos.
- Muchos algoritmos de Deep Learning dan supuestos explícitos o implícitos razonables que son necesarios para que una gran cantidad de problemas de Inteligencia Artificial puedan capturar las ventajas de una representación distribuida.
- El entrenamiento de redes profundas es difícil principalmente por dos razones: no-linealidad y no-convexidad. Aún existen detalles incomprendidos en este aspecto y parte de la investigación en Deep Learning versa sobre esto.

Posteriormente, se estudiaron las redes neuronales recurrentes como una alternativa a modelos usuales para modelar series tiempo. Se explicó cómo es que las redes neuronales recurrentes sufren del problema de gradiente que se desvanece y explota y esto hace que su entrenamiento en ciertos casos sea muy difícil. Para subsanar este problema, se usó una Long Short Term Memory por su habilidad para modelar dependencias temporales lejanas y porque esta arquitectura se entrena satisfactoriamente a diferencia de las redes recurrentes usuales. Vimos cómo esta arquitectura ha sido usada con gran éxito para problemas de procesamiento de texto, de voz y de imágenes alcanzando resultados sin precedentes.

Se usó una LSTM para modelar los consumos de energía eléctrica en el período 2006-2015 y así poder tener un modelo de predicción del mismo. Se hizo un tuning del modelo con respecto a lo siguiente: *Dropout, Lag, Statefulness, Número de bloques por capa, Tamaño del batch, Número de épocas, Número de capas*.

Se comparó con modelos tradicionales de minería de datos (regresión y perceptrón de una sola capa). Se encontraron resultados satisfactorios, pues el modelo de Deep Learning superó a los otros y si se compara con la energía efectivamente producida por la CFE (caso “sin modelo”), pudimos observar que el modelo de LSTM es mejor. Por lo tanto, habría una generación de valor en términos de menos desperdicio de energía si se usara el modelo propuesto para predecir el consumo.

Sin embargo existe gran área de oportunidad, pues la serie que se usó no es suficientemente grande para poder explotar modelos de deep learning. Sin embargo, el ejercicio

ejemplificó que es factible usar modelos de deep learning como alternativas para modelar series de tiempo. Un conjunto de datos más grande también implicaría contar con mucho mayor poder de cómputo, con lo cual actualmente no se cuenta (y no era uno de los objetivos del trabajo).

Como trabajo futuro se puede enlistar lo siguiente:

- Se desearía contar con un conjunto de datos mucho más grande, esto quizás permitiría notar claramente las fortalezas de los algoritmos de deep learning para modelar series temporales. Naturalmente para lograr este punto también tendríamos que contar con mayor poder de cómputo, en específico usar GPUs.
- Se desearía contar con una serie de tiempo multivariada para poder modelar y explotar las dependencias temporales de varios fenómenos simultáneamente a lo largo del tiempo.
- Al hacer el *tunning* de la LSTM vimos que la famosa técnica de *Dropout*, que permite aminorar el sobreajuste de las redes profundas, no funcionó para nuestro modelo. Desearíamos revisar el trabajo de Gal et al. publicado en 2015 [16] en donde se discute cómo modificar la técnica de *Dropout* para redes recurrentes.
- Se desearía entender que atributos de la serie está encontrando y generando hasta llegar la máximo nivel de abstracción de relaciones más locales a entender grandes bloques en la serie
- Se desearía mejorar la partición del conjunto original en entrenamiento y prueba, pues como vimos, para series temporales no se puede aplicar validación cruzada directamente debido a la dependencia secuencial de las observaciones.
- Debido a la falta de teoría sobre *Deep Learning*, se desearía hacer un *survey* de todo el trabajo existente para conocer realmente el estado actual del área y poder encontrar y sugerir una línea de investigación teórica futura.

Bibliografía

- [1] Anthony, Martin and Peter L Bartlett (2009), *Neural network learning: Theoretical foundations*. cambridge university press.
- [2] Bengio, Yoshua (2009),
Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2, 1–127.
- [3] Bengio, Yoshua, Ian J. Goodfellow, and Aaron Courville (2015),
Deep learning., URL <http://www.iro.umontreal.ca/~bengioy/dlbook>. Book in preparation for MIT Press.
- [4] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi (1994),
Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5, 157–166.
- [5] Boulanger-Lewandowski, Nicolas, Yoshua Bengio, and Pascal Vincent (2012),
Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*.
- [6] Chen, Xie, Adam Eversole, Gang Li, Dong Yu, and Frank Seide (2012),
Pipelined back-propagation for context-dependent deep neural networks. In *INTERSPEECH*.
- [7] Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014),
Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- [8] Cireşan, Dan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber (2012),
Multi-column deep neural network for traffic sign classification. *Neural Networks*,

- 32, 333–338.
- [9] de Electricidad, Comisión Nacional (2016),
Consulta de archivos de costo unitario y capacidad promedio prevista y realizada diaria. URL http://www.cfe.gob.mx/Proveedores/4_Informaciongeneral/Control_despacho_energia/Paginas/Control-y-despacho-de-energia-.aspx.
 - [10] de Electricidad, Comisión Nacional (2016),
Consulta de archivos de costo unitario y capacidad promedio prevista y realizada diaria. URL <http://app.cfe.gob.mx/Applicaciones/OTROS/costostotales/ConsultaArchivoCostosyCapacidades.aspx>.
 - [11] de Electricidad, Comisión Nacional (2016),
Consulta de archivos de costo unitario y capacidad promedio prevista y realizada diaria. URL http://www.cfe.gob.mx/Proveedores/1_Adquisicionesarrendamientosyservicios/Paginas/Estadisticasgenerales.aspx.
 - [12] de Población, Consejo Nacional (2012),
México Índices de marginación. URL http://www.conapo.gob.mx/es/CONAPO/Indices_de_Marginacion.
 - [13] Delalleau, Olivier and Yoshua Bengio (2011),
Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems*, 666–674.
 - [14] Deng, Li and Dong Yu (2014),
Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7, 197–387.
 - [15] Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio (2010),
Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11, 625–660.
 - [16] Gal, Yarin (2015),
A theoretically grounded application of dropout in recurrent neural networks. *arXiv preprint arXiv:1512.05287*.
 - [17] Gers, Felix A, Jürgen Schmidhuber, and Fred Cummins (2000),
Learning to forget: Continual prediction with lstm. *Neural computation*, 12, 2451–2471.

- [18] Gers, Felix A, Nicol N Schraudolph, and Jürgen Schmidhuber (2002), Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3, 115–143.
- [19] Glorot, Xavier and Yoshua Bengio (2010), Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, 249–256.
- [20] Gomes, Lee (2014), Machine-learning maestro michael jordan on the delusions of big data and other huge engineering efforts. URL <http://spectrum.ieee.org/robotics/artificial-intelligence/machinelearning-maestro-michael-jordan-on-the-delusions-of-big-data-and-other-huge-engine>
- [21] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016), Deep learning., URL <http://www.deeplearningbook.org>. Book in preparation for MIT Press.
- [22] Graves, Alex (2012), Neural networks. In *Supervised Sequence Labelling with Recurrent Neural Networks*, 15–35, Springer.
- [23] Graves, Alex (2013), Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.
- [24] Graves, Alex, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber (2009), A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31, 855–868.
- [25] Graves, Alex and Jürgen Schmidhuber (2005), Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18, 602–610.
- [26] Graves, Alex, Greg Wayne, and Ivo Danihelka (2014), Neural turing machines. *arXiv preprint arXiv:1410.5401*.
- [27] Greff, Klaus, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber (2015), Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*.
- [28] Hermans, Michiel and Benjamin Schrauwen (2013),

- Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*, 190–198.
- [29] Hinton, Geoffrey E, Simon Osindero, and Yee-Whye Teh (2006),
A fast learning algorithm for deep belief nets. *Neural computation*, 18, 1527–1554.
 - [30] Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006),
Reducing the dimensionality of data with neural networks. *Science*, 313, 504–507.
 - [31] Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov (2012),
Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.
 - [32] Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber (2001),
Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.
 - [33] Hochreiter, Sepp and Jürgen Schmidhuber (1997),
Long short-term memory. *Neural computation*, 9, 1735–1780.
 - [34] Karpathy, Andrej (2015),
The unreasonable effectiveness of recurrent neural networks. URL <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.
 - [35] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012),
Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.
 - [36] Langkvist, Martin, Lars Karlsson, and Amy Loutfi (2014),
A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42, 11–24.
 - [37] Larochelle, Hugo and Iain Murray (2011),
The neural autoregressive distribution estimator. In *AISTATS*, volume 1, 2.
 - [38] Le Roux, Nicolas and Yoshua Bengio (2010),
Deep belief networks are compact universal approximators. *Neural computation*, 22, 2192–2207.
 - [39] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). *Nature*, 521, 436–444.
 - [40] LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998),
Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86, 2278–2324.

- [41] Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013), Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30.
- [42] Martens, James (2010), Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 735–742.
- [43] Mitchell, Thomas M (1997), Machine learning. *New York*.
- [44] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013), Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
- [45] Mohri, M., A. Rostamizadeh, and A. Talwalkar (2012), *Foundations of machine learning*. MIT Press.
- [46] Nocedal, Jorge and Stephen Wright (2006), *Numerical optimization*. Springer Science & Business Media.
- [47] Pascanu, Razvan, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio (2013), How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.
- [48] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013), On the difficulty of training recurrent neural networks. *ICML* (3), 28, 1310–1318.
- [49] Prasad, Sharat C and Piyush Prasad (2014), Deep recurrent neural networks for time series prediction. *arXiv preprint arXiv:1407.5949*.
- [50] Ramsundar, Bharath, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande (2015), Massively multitask networks for drug discovery. *arXiv preprint arXiv:1502.02072*.
- [51] Rifai, Salah, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio (2011), Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 833–840.
- [52] Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1988), Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1.

- [53] Sak, Hasim, Andrew W Senior, and Françoise Beaufays (2014),
Long short-term memory recurrent neural network architectures for large scale
acoustic modeling. In *INTERSPEECH*, 338–342.
- [54] Schmidhuber, Jürgen (2015),
Recurrent Neural Networks why use recurrent networks at all? and why use a
particular deep learning recurrent network called long short-term memory or lstm?
URL <http://people.idsia.ch/~juergen/rnn.html>.
- [55] Shalev-Shwartz, S. and S. Ben-David (2014), *Understanding machine learning: from theory to algorithms*. Cambridge University Press, New York.
- [56] Socher, Richard, Cliff C Lin, Chris Manning, and Andrew Y Ng (2011),
Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, 129–136.
- [57] Song, Hyun Ah and Soo-Young Lee (2013),
Hierarchical representation using nmf. In *Neural Information Processing*, 466–473, Springer.
- [58] Srivastava, Nitish, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan
Salakhutdinov (2014),
Dropout: a simple way to prevent neural networks from overfitting. *Journal of
Machine Learning Research*, 15, 1929–1958.
- [59] Sutskever, Ilya (2013), *Training recurrent neural networks*. Ph.D. thesis, University
of Toronto.
- [60] Taylor, Graham W and Geoffrey E Hinton (2009),
Factored conditional restricted boltzmann machines for modeling motion style.
In *Proceedings of the 26th annual international conference on machine learning*,
1025–1032, ACM.
- [61] Vinyals, Oriol and Daniel Povey (2011),
Krylov subspace descent for. *arXiv preprint arXiv:1111.4259*.
- [62] Vitale, Alessandro (2015),
What google cloud vision api means for startups. URL <https://medium.com/@alevitale/what-google-cloud-vision-api-means-for-deep-learning-startups-cd39226922e5#.u796cwr5q>.
- [63] Williams, Ronald J and David Zipser (1995),
Gradient-based learning algorithms for recurrent networks and their computatio-

nal complexity. *Back-propagation: Theory, architectures and applications*, 433–486.

Apéndice A

Resultados adicionales

A.1. Código en Keras para entrenar la LSTM

También puede ser consultado en Github

```
#MARIA FERNANDA MORA ALBA
#LSTM PARA PREDICCIÓN DE SERIES DE TIEMPO
#24 NOVIEMBRE 2016

import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import os
os.chdir("/home/sophia/tesis_lic/tesis/tesis_final/final")
#os.chdir("/home/sophie/tesis/data/final_code")

#=====PREPROCESAMIENTO=====
```

```
#  
  
# Fijamos la semilla para tener reproducibilidad  
numpy.random.seed(103596)  
# Cargando y explorando dataset original con datos no estacionarios  
  
print('Graficamos la serie de tiempo: ')  
dataframe_noest = pandas.read_csv('data_noest.csv', usecols=[0],  
                                    engine='python')  
plt.plot(dataframe_noest, color="#5e6d96")  
plt.ylabel('Demanda sin transformar (MWs)')  
plt.xlabel('Tiempo')  
#plt.show()  
  
#dataset = dataset.astype('float32')  
  
# Hacemos los datos estacionarios en R e importamos el csv  
print('Graficamos la serie de tiempo estacionaria: ')  
dataframe = pandas.read_csv('data_est.csv', usecols=[0],  
                            engine='python')  
plt.plot(dataframe, color="#5e6d96")  
plt.ylabel('Demanda estacionaria')  
plt.xlabel('Tiempo')  
#plt.show()  
# guardamos los valores solamente  
dataset = dataframe.values  
  
# escalamos el dataset a (0,1) para poder usar una red neuronal  
# Usamos esta transformacion:  $X_{std} = (X - X.\min(axis=0)) / (X.\max(axis=0) - X.\min(axis=0))$  como una alternativa a media cero  
# y varianza unitaria.  
scaler = MinMaxScaler(feature_range=(0, 1))  
#fit a los datos y luego los transformamos  
dataset = scaler.fit_transform(dataset)  
  
#visualizamos la serie de tiempo estacionaria y escalada,  
# con esta trabajaremos  
print('Graficamos la serie de tiempo estacionaria y escalada en  
      (0,1): ')  
plt.plot(dataset, color="#5e6d96")  
plt.ylabel('Demanda estacionaria y escalada en (0,1)')
```

```

plt.xlabel('Tiempo')
#plt.show()

print('Creamos_dataset_de_entrenamiento_y_prueba... ')
# dividimos en entrenamiento y prueba. (to be refined for time
# series prediction)
train_size = int(len(dataset) * 0.8)
#test_size = len(dataset) - train_size
train, test = dataset[0:train_size ,:],
              dataset[train_size:len(dataset),:]

# Reformular como X=t and Y=t+1
# hay que convertir el vector de valores de la serie de tiempo en
# una matriz para poder aplicar algoritmos de prediccion
def create_dataset(dataset, lag=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-lag-1):
        a = dataset[i:(i+lag), 0]
        dataX.append(a)
        dataY.append(dataset[i + lag, 0])
    return numpy.array(dataX), numpy.array(dataY)

#Cambiamos el tamano de la ventana
lag = 9
trainX, trainY = create_dataset(train, lag)
testX, testY = create_dataset(test, lag)

# Actualmente los datos estan en la forma de [samples, features]
# Reformular el input para que sea [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))

#----- MODELO -----
# Creacion y entrenamiento de la LSTM network
# La red tiene una capa visible con 1 input y el output debe ser 1
# valor (la prediccion)
# Los parametros de la red son:
#   - Tamano del lag
#   - Numero de bloques
#   - Profundidad

```

```

#      - Dropout
#      - Tamano del batch
#      - Numero de epochas
#      - Funcion de activacion

print( 'Creando_arquitectura_LSTM... ' )
batch_size = 1
model = Sequential()

#lag=input_length , input_dim=1
##Stateful=true porque no se resetea en cada tiempo
model.add(LSTM(6, batch_input_shape=(batch_size, lag, 1),
               stateful=True, return_sequences=False ))
#model.add(Dropout(0.8))
#model.add(LSTM(6, return_sequences=True, stateful=True ))
#model.add(Dropout(0.5))
#model.add(LSTM(6, return_sequences=True, stateful=True ))
#model.add(LSTM(6, return_sequences=True), stateful=True)
#model.add(Dropout(0.5))
#model.add(LSTM(6, return_sequences=True, stateful=True ))
#model.add(LSTM(6, return_sequences=True), stateful=True)
#model.add(Dropout(0.5))
#model.add(LSTM(6, return_sequences=False, stateful=True))
#model.add(Dropout(0.5))

#output layers. Size=1 (1 timestep)
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
print( 'Entrenando_modelo... ' )
for i in range(200):
    model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size,
              verbose=2, shuffle=False)
    model.reset_states()
print( 'El_numero_de_parametros_en_cada_capa_es: ' )
model.summary()

#-----EVALUACION-----
#-----Predicciones-----
#print( 'Haciendo_predicciones: ' )Pre
trainPredict = model.predict(trainX, batch_size=batch_size)

```

```

model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
# Invertimos predicciones a sus unidades originales
print('Desescalando los datos ...')
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# Calculamos el RMSE

print('Evaluando modelo en entrenamiento y prueba : ')
trainScore = math.sqrt(mean_squared_error(trainY[0],
                                           trainPredict[:,0]))
print('Train Score: %.5f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0],
                                         testPredict[:,0]))
print('Test Score: %.5f RMSE' % (testScore))



---




---


VISUALIZACION


---


Graficando datos originales y estimados por el modelo :
Shift predicciones de train para graficarlas
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[lag:len(trainPredict)+lag, :] = trainPredict
Shift predicciones de test para graficarlas
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(lag*2)+1:len(dataset)-1, :] = testPredict
Graficamos reales y predicciones
plt.plot(scaler.inverse_transform(dataset))
#dataset=np.exp(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()

Ahora regresamos a las unidades originales de la base de datos y
graficamos. Para esto hay que aplicar exponencial para quitar la
transformacion logaritmo que estabilizaba la varianza de los datos
(eso se hizo en R)

```