Energy Consumption Forecast
A Recurrent Neural Networks Approach

Junghanss, Juan Cruz*

Junio 2021

Abstract

El presente trabajo ofrece una reinterpretación en español de las principales obras académicas sobre los pronósticos de series de tiempo con Machine Learning junto con dos aplicaciones prácticas y sus respectivos códigos. El objetivo es distinguir los alcances de los métodos estadísticos tradicionales de los de Deep Learning y cómo se integran en proyectos reales. Lazzeri (2020) cubre un gran espectro del know-how de las soluciones end-to-end profesionales que se ofrecen en la industria de la ciencia de datos. Como ejemplo principal aborda un pronóstico del consumo de energía con Redes Neuronales Recurrentes, que inspirará la primer aplicación práctica de este trabajo. Zhang et. al. (2020) desarrolla de principio a fin todas las herramientas de Deep Learning, su análisis cuántitativo y aplicación en código. Ng (2018) concentra brevemente las mejores prácticas recomendadas para administrar un proyecto de Machine Learning. Por último, Hyndman et. al (2018) profundiza en los principales métodos estadísticos clásicos para el pronóstico de series de tiempo, uno de los cuales (ARIMA) es empleado para la segunda aplicación práctica: un pronóstico del consumo de energía agregado por tipo de usuario en Argentina, entre 2012 y 2020.

*jcjunghans22@ucema.edu.ar

Contents

1 Introducción

El presente trabajo ofrece una reinterpretación en español de las principales obras académicas sobre los pronósticos de series de tiempo con Machine Learning junto con dos aplicaciones prácticas y sus respectivos códigos. El objetivo es distinguir los alcances de los métodos estadísticos tradicionales de los de Deep Learning y cómo se integran en proyectos reales. Lazzeri (2020) cubre un gran espectro del know-how de las soluciones end-to-end profesionales que se ofrecen en la industria de la ciencia de datos. Como ejemplo principal aborda un pronóstico del consumo de energía con Redes Neuronales Recurrentes, que inspirará la primer aplicación práctica de este trabajo. Zhang et. al. (2020) desarrolla de principio a fin todas las herramientas de Deep Learning, su análisis cuántitativo y aplicación en código. Ng (2018) concentra brevemente las mejores prácticas recomendadas para administrar un proyecto de Machine Learning. Por último, Hyndman et. al (2018) profundiza en los principales métodos estadísticos clásicos para el pronóstico de series de tiempo, uno de los cuales (ARIMA) es empleado para la segunda aplicación práctica: un pronóstico del consumo de energía agregado por tipo de usuario en Argentina, entre 2012 y 2020.

Las secciones 1 y 2 hacen un análisis cualitativo acerca del Machine Learning, el Forecasting en series de tiempo y Soluciones Data-Driven relacionadas.

La sección 3 aborda un análisis más abarcativo de una arquitectura básica de una Red Neural Recurrente, preparando al lector para entrar en detalles más técnicos en la siguiente sección.

La sección 4 desarrolla como ejemplo práctico de una RNN la aplicación del presente trabajo: un pronóstico del consumo de energía.

La sección 5 aborda de manera más breve un ejemplo práctico del método econométrico de ARIMA aplicado a datos del consumo de energía en Argentina.

Finalmente, la sección 6 hace un repaso general de las conclusiones y comentarios finales.

Los apéndices ofrecerán el análisis de corte cuantitativo correspondiente a cada sección.

Para comenzar, haremos una distinción preliminar necesaria sobre los fundamentos del Machine Learning y los avances en la industria hasta el año corriente 2021. Cabe destacar que la industria de la Inteligencia Artificial (denotada como AI en inglés), el Machine Learning (ML) y sus disciplinas subyacentes como Business Inteligence (BI), Data-Driven Solutions, etc. son campos en los que acontecen enormes avances académicos y profesionales año tras año, siendo de las industrias con mayor innovación e impacto transversal (en el resto de rubros y mercados).

1.1 Aspectos del Machine Learning

El Machine Learning (aprendizaje automático en español) es la rama de la AI que abarca los estudios de informática y cálculo estadístico aplicado con el fin de desarrollar soluciones a problemas de cualquier disciplina o mercado que permitan mejorar la eficiencia y eficacia de los actuales procesos, escalando en el tiempo. En otras palabras, soluciones que sean mejores a las actuales (o nuevas) en áreas donde antes no era posible llevar a cabo un proceso de tal forma.

Un ejemplo muy común es el avance del Machine Learning en el mercado de la Seguridad Electrónica (bien denominado CCTV desde sus inicios, pero que ha quedado obsoleto el nombre). Si uno se remonta a dicho mercado en el año 2010, vería que apenas estaba transicionando a pasos firmes en cámaras de seguridad que emplearan altas resoluciones HD (es decir, Full HD 1920x1080 o como mucho UHD - Ultra HD), ya que antes los productos de videovigilancia eran de tecnología analógica (resoluciones de hasta 900 Tevelineas, algo que hoy no existe más). Sin embargo, ya para el año 2016 aproximadamente, tan solo 6 años después, se incursionó en el área de Machine Learning dentro de las cámaras de seguridad IP, con el fin de poder adicionar soluciones de reconocimiento facial o de patentes de vehículos. Empleando Deep Learning (aprendizaje automático profundo, una rama del ML), marcas de origen asiático como Hikvision o Dahua, y de origen americano como Axis, Pelco, entre otras, alcanzaron soluciones de analítica de video inteligente. En la actualidad no hay DVR's (videograbadoras) de gama media a baja que no dispongan de la opción de reconocimiento facial.

Este ejemplo, que es uno muy visible, no es el único. El Machine Learning tuvo avances en numerosas industrias, desde la medicina e industrias con procesos productivos, hasta mercados de servicios como los financieros.

A continuación, podemos hacer una breve reseña de los tipos de soluciones características del Machine Learning:

1. Supervised Learning: El aprendizaje supervisado (en inglés Supervised Learning) es una técnica que consta de proveer, al modelo, tanto el input, como el output deseado. En otras palabras, los datos están etiquetados o identificados tanto en las variables explicativas (parámetros o features), como en las variables explicadas (lo que será el output). Esto implica que nuestro modelo podrá interpretar que $X_i \Longrightarrow Y_i$ y encontrar patrones en la relación que hay entre los parámetros y las variables explicadas. Así, cuando uno luego posee un modelo ya entrenado, puede testearlo con un set de datos distinto en el cual el algoritmo debería identificar como output los Y_i. El objetivo del aprendizaje supervisado es estudiar muchos ejemplos etiquetados como estos y luego poder hacer predicciones sobre puntos de datos futuros, como, por ejemplo, asignar precios de venta precisos a otros autos usados que tengan características similares al usado durante el proceso de entrenamiento. Se llama aprendizaje supervisado porque los científicos de datos supervisan el proceso de un algoritmo que aprende del conjunto de datos de entrenamiento.

   A su vez, el Supervised Learning puede tomar dos formas:

   - Regresión: para problemas numéricamente continuos, es decir, cuando necesitamos modelar el patrón y ser capaces de explicar con un numéro específico como resultado el comportamiento de los datos. Se usa, por ejemplo, para las siguientes aplicaciones: estimar la demanda de un producto, predecir el retorno de inversión, los precios de viviendas , etc.
   - Clasificación: para problemas numéricamente discretos, es decir, cuando los datos en los cuales hay patrones se pueden clasificar en categorías. El algoritmo puede responder preguntas sencillas de dos opciones (es decir, binarias), como sí o no, verdadero o falso, por ejemplo: ¿Esta persona en la foto está feliz o no?

Sin embargo, hay también clasificaciones de múltiples categorías, llamadas multi-class classification. Las preguntas que puede responder son un poco más complejas, por ejemplo en clasificación de sentimientos para reseñas/críticas de restaurantes de 1 a 5 estrellas. El algoritmo debe ser capaz de identificar si un comentario corresponde a 1/2/3/4/5 estrellas.

2. Unsupervised Learning: El aprendizaje no supervisado (en inglés, Unsupervised Learning) es una técnica que, a diferencia del supervised learning, consume como input datos que solamente están identificados por las variables explicativas (es decir, las características), pero no hay un output en concreto para estos. El modelo devolverá como resultado agrupaciones por similitud que capturan la información más importante del conjunto de datos. Hay distintos tipos de unsupervised learning, como el análisis por agrupaciones (clusters), detección de anomalías y analísis de componentes principales (PCA):

   - Cluster analysis: es un tipo de aprendizaje no supervisado que separa puntos de datos similares en grupos intuitivamente identificables. Se suele usar, por ejemplo, para segmentar clientes en negocios, predecir o identificar las preferencias de los clientes, etc.
   - Anomaly detection: la detección de anomalias implica identificar o predecir comportamientos raros o poco usuales, es decir, puntos de datos que no son "normales". Básicamente el enfoque es que el algoritmo aprenda cómo se ve la actividad normal (usando datos históricos de cierta variable) y luego así podrá identificar cuando un punto es significativamente diferente. Como ejemplos, se puede considerar la detección de fraude, la predicción de riesgo, etc.
   - Principal component analysis (PCA): este método, por medio de métodos algebraicos, reduce la dimensionalidad del espacio de características con menos variables (no correlacionadas), que se denominan componentes principales. Es completamente útil cuando se requiere responder preguntas sobre las relaciones entre variables.

3. Reinforcement Learning: La técnica de Reinforcement Learning (o aprendizaje por refuerzo) es distinta a las dos anteriores en el sentido de que el modelo toma como input información parcial de las variables explicativas (X_i) y las variables explicadas (Y_i), pero con la finalidad de poder realizar una secuencia de decisiones. Para esto, debe aprender interactivamente por prueba-error usando el feedback de cada intento y consecuente experiencia. Entre los ejemplos más populares, las aplicaciones del reinforcement learning son el control de señales de tráfico, la optimización de reacciones químicas, las recomendaciones personalizadas de noticias para un usuario, etc.

   Los elementos principales de un problema de Reinforcement Learning son:

   - Environment (contexto): ambiente en el cual opera el agente, por ejemplo: una partida de ajedrez.
   - State (estado): situación actual del agente.
   - Reward (recompensa): feedback del
   - Policy (política o plan): método para mapear el estado a un curso de acción.

- Value (valor): Recompensa futura que un agente recibiría por tomar cierta acción bajo cierto estado.

Ejemplos de algoritmos más utilizados en Reinforcement Learning son Q-Learning y SARSA (State-Action-Reward-State-Action).

4. Deep Learning: Por último, cuando hablamos de Deep Learning o aprendizaje profundo nos referimos a una de las técnicas más populares en ML que abarca la utilización de modelos con arquitecturas más complejas. Pueden ser construidos con Neural Networks, de una dimension relativamente grande, es decir, que poseen muchas capas (layers) y neuronas. A su vez, dentro de las redes neuronales es posible encontrar distintos tipos, como las Convolucionales (CNN) o Recurrentes (RNN), de las cuales hablaremos en mayor detalle más adelante. En la sección 3 donde abordaremos en profundidad sobre RNN, distinguiremos también en detalle las diferencias entre Machine Learning y Deep Learning.

1.2 Aspectos del Forecasting de Series de Tiempo

Una pregunta importante que surge en esta subsección y sirve como puntapie es: ¿Por qué el pronóstico de series de tiempo es actualmente un área fundamental de investigación entre industrias?

Un ejemplo del uso de soluciones de predicción de series de tiempo sería la simple extrapolación de una tendencia pasada para pronosticar las temperaturas de la próxima semana. Otro ejemplo sería el desarrollo de un modelo estocástico lineal complejo para predecir el movimiento de las tasas de interés a corto plazo. Los modelos de series de tiempo también se han utilizado para pronosticar la capacidad de picos de demanda de las aerolíneas, la demanda de energía estacional, futuras ventas online en rubros con e-commerce y muchos casos más.

Estos son solamente los ejemplos más comunes, pero hay más.

El método utilizado para producir un modelo de Forecasting de series de tiempo puede implicar el uso de un modelo determinista simple, como una extrapolación lineal, o bien el uso de enfoques de Deep Learning más complejos.

Debido a su aplicabilidad a muchos problemas de la vida real, como la detección de fraudes, el filtrado de spam en correos electrónicos, las finanzas y el diagnóstico médico, y su capacidad para producir resultados procesables, el Machine Learning y los algoritmos de aprendizaje profundo ganaron mucha atención en los últimos años. En general, los métodos de Deep Learning se han desarrollado y aplicado a escenarios de predicción de series de tiempo univariantes (de una sola variable), donde la serie de tiempo consiste en observaciones únicas registradas secuencialmente en incrementos de tiempo iguales. Sin embargo, a menudo por este motivo los modelos de Deep Learning univariantes se han desempeñado peor que los métodos clásicos econométricos de pronóstico, como el exponential smoothing y el autoregressive integrated moving average (ARIMA), entre otros. Esto condujo a una idea general errónea de que los modelos de aprendizaje profundo son ineficientes en escenarios de predicción de time series, así es que habría que optar por arquitecturas más complejos como

CNN (Convolutional Neural Networks) o RNN (Recurrent Neural Networks), dentro del área de Deep Learning.

1.2.1 Datos de Series de Tiempo

Los datos de series de tiempo (time series data) son datos ordenados cronológicamente, capturados por lo general con fechas y para cada una respectivas valores de las variables del modelo. Los formatos, en Python, pueden ser por lo general *TimeStamp*, *Date* o *Time*, entre otros. Sus componentes son los siguientes:

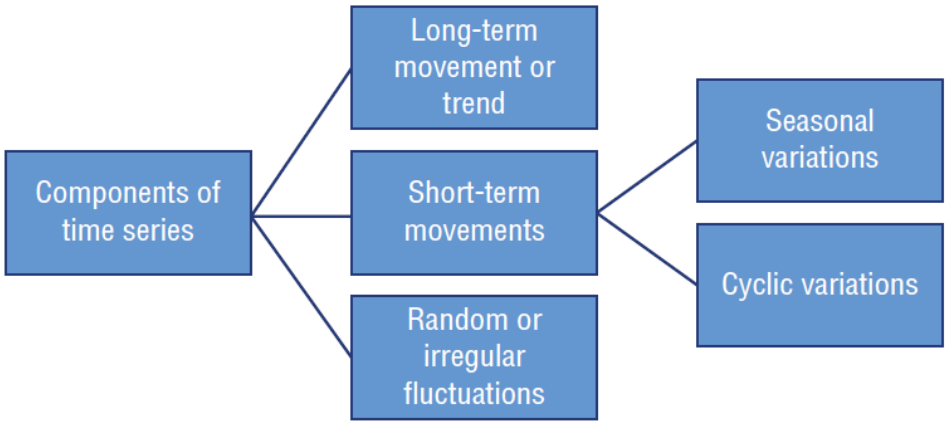


Figure 1: Componentes de las series de tiempo

Las características más comunes dentro de los time series datasets son:

- Tendencia (movimiento de largo plazo): son los movimientos generales que se pueden observar a lo largo de una serie de tiempo en un intervalo prolongado. Pueden ser tendencias alcistas, bajistas o inclusive estables, que permanecen inalteradas en promedio a lo largo del tiempo. De todas maneras, siempre debería notarse un tendencia principal en todo el conjunto de datos.
- Estacionalidad (movimiento de corto plazo): son fluctuaciones que responden a un mismo patrón en un período menor a un año generalmente, esto quiere decir, que son fluctuaciones fijas en períodos conocidos. Surgen a raíz de diferentes cuestiones sociales (como temporada de vacaciones y festividades), temporadas específicas del año y condiciones climáticas. Estos factores juegan un papel importante en las variaciones estacionales, como por ejemplo la venta de paraguas e impermeables en temporada de lluvias y la venta de aires acondicionados en verano.
- Ciclos (movimiento de corto plazo): son patrones recurrentes que existen cuando los datos exhiben alzas y bajas que no son por períodos fijos. Un período completo es un ciclo, pero un ciclo no tendrá una duración predeterminada de tiempo, incluso si la duración de la fluctuación es usualmente mayor que un año.
- Componente estocástico: este tipo de fluctuaciones son el componente errático de una serie de tiempo, que no se pueden controlar ni predecir. Como ejemplos, sabemos que de manera aleatoria existe la posibilidad de sufrir un terremoto, una inundación, un huracán o una pandemia, como sucedió con el COVID-19.

Un buen ejemplo para distinguir las diferencias entre fluctuaciones ciclicas y estacionales es la Figura 2 a continuación:

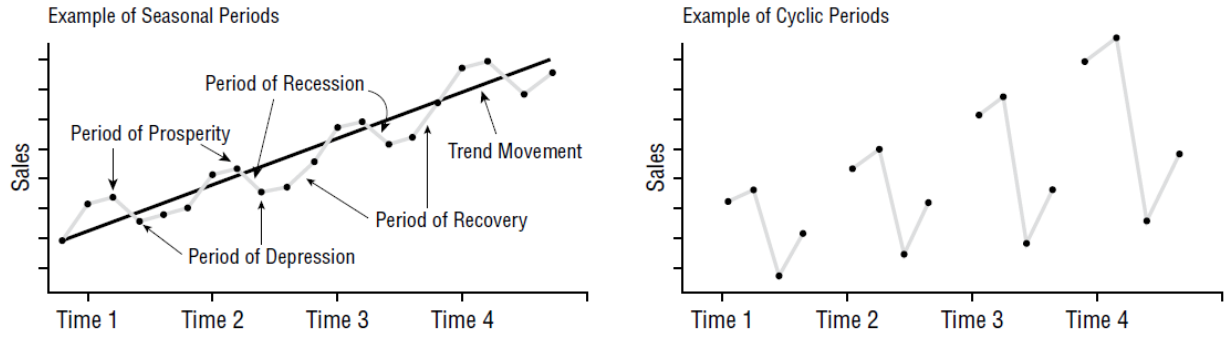


Figure 2: Diferencias entre Ciclo y Estacionalidad

En Data Science se suele referir a los tres componentes de corto y largo plazo como *señales* (signals) en los datos de series de tiempo, ya que son indicadores deterministicos que se derivan de los datos en sí. Por el contrario, el último componente aleatorio, es una variación arbitraria en los datos que, como mencionamos, no se puede predecir porque es independiente de los otros tres indicadores. Por esa razón se suele referir a este como *ruido* (noise) en los datos.

Es muy importante que el científico de datos pueda identificar cada componente presente en los datos de time series para poder construir una solución eficaz de forecasting, mejorando la precisión del algoritmo. Generalmente construyendo parámetros (features) a partir de dichas fluctuaciones se logra mejorar la precisión.

Con el fin de identificar cada componente, una práctica habitual es aplicar un proceso de descomposición para eliminar los respectivos efectos sobre los datos. En la Figura 3 podemos apreciar la coexistencia de todos los componentes recién mencionados.

Explicado esto, surge también otro concepto importante en los datos de series de tiempo que puede mejorar la precisión de los resultados de los algoritmos: la *estacionariedad*. Stationarity, en inglés, significa que los parámetros de una serie de tiempo no cambian con los períodos. En otras palabras, propiedades básicas de una distribución de un time series dataset, como la media y la varianza, se mantienen constante a lo largo del tiempo. Por este motivo, los procesos estacionarios de series de tiempo son mucho más fácil de analizar y modelar porque el supuesto subyacente es que sus propiedades no dependen del tiempo y serán iguales en el futuro, tal como lo han sido en períodos pasados.

Generalmente, hay dos tipos de estacionariedad: la fuerte y la débil. Una serie de tiempo se caracteriza de poseer *fuerte estacionariedad* cuando todos sus parámetros estadísticos no cambian en el tiempo. En contraste, una serie de tiempo se caracteriza de tener *débil estacionariedad* cuando solamente su media y sus funciones de auto-covarianza no cambian en el tiempo.

Alternativamente, series de tiempo que exhiben cambios en los valores de los datos, sea por tendencia o estacionalidad, claramente no son estacionarias, y en consecuencia, no son fáciles

de modelar y predecir. Para obtener resultados precisos de un pronóstico se requiere que las series no estacionarias sean transformadas en estacionarias.

Sin embargo, hay casos donde relaciones no lineales que son desconocidas no pueden determinarse por métodos clásicos como autoregresión, medias móviles y ARIMA. Esta información puede ser muy útil a la hora de armar modelos de Machine Learning, y puede ser usada en ingeniería de características (feature engineering) y procesos de selección. En la realidad, por ejemplo, muchas series económicas están lejos de ser estacionarias, ya que si se visualizan en su forma original, aunque ajustadas por estacionalidad inclusive, van a mostrar fluctuaciones por tendencia, ciclos y otras características no-estacionarias.

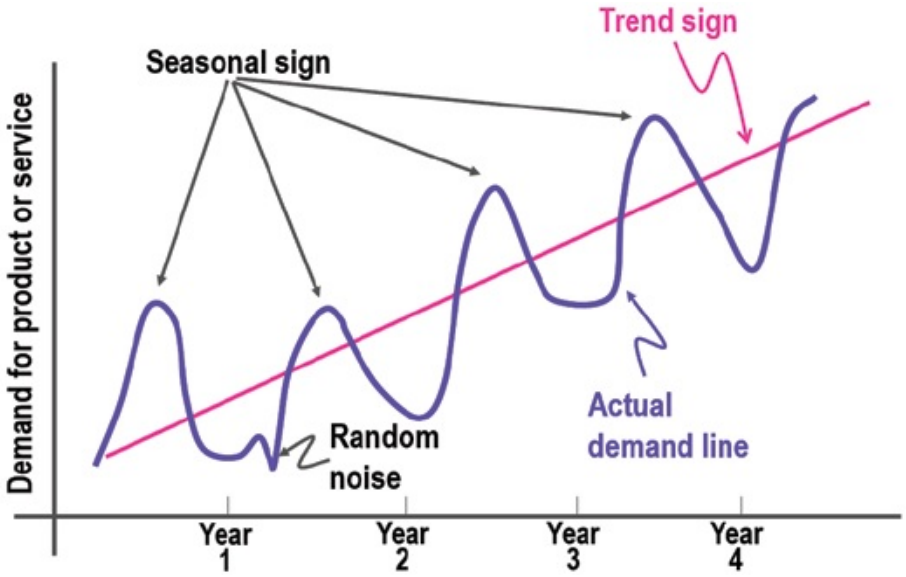


Figure 3: Ejemplo de una serie de tiempo

1.2.2 Elementos Esenciales del Forecasting

Para cualquier científico de datos, previo a comenzar a construir un modelo predictivo, resulta recomendable analizar y tener definidos los siguientes aspectos:

- Inputs y Outputs del modelo: Un aspecto importante es evaluar qué datos hay disponibles y qué se quiere predecir del futuro. Los inputs son datos históricos que alimentan el modelo y los outputs son los resultados de la predicción en el futuro. Por ejemplo, los datos del consumo de energía de los últimos 7 días, colectados por un sensor en la red eléctrica, serían los Inputs, mientras que los valores predichos del consumo de energía para el día siguiente se definen como Outputs.
- Granularidad de los datos: El grado de granularidad en la previsión de los datos representa el nivel de detalle más profundo de valores capturados para cada período. La granularidad está relacionada con la frecuencia a la cual los valores de la serie de tiempo son colectados: usualmente, en escenarios de IoT (Internet of Things)[1], los científicos de datos necesitan manejar datos que fueron recolectados por sensores cada unos segundos

[1]IoT, o en español, Internet de las Cosas, se define normalmente como un grupo de dispositivos que se conectan a internet y son capaces de recolectar, compartir y almacenar datos, entre otras cosas.

de diferencia. Esto implica por supuesto grandes volumenes de datos por el nivel de detalle que poseen.

- Horizonte del modelo de Forecasting: El horizonte del modelo de predicción es el largo a definir, en términos de tiempo futuro, por el cual se prepararán los pronósticos. Estos generalmente varían desde horizontes de corto-plazo (menos de 3 meses), hasta horizontes de largo-plazo (más de 2 años inclusive). El Forecasting de corto plazo es empleado usualmente para micro-objetivos como planificación de requerimientos y materia prima, presupuestos y planeamiento de otros tipos. De manera contraria, el Forecasting de largo plazo se usa para predecir objetivos a nivel macro, es decir, que puedan guiar planes como de diversificación de productos, niveles de venta, publicidad y otro tipo de planificación más estructural.

- Los parámetros endógenos y exógenos del modelo de Forecasting: En negocios y ciencias sociales, los términos económicos de endogeneidad y exogeneidad se usan para describir factores internos y externos, respectivamente, que afectan a la producción, la eficiencia, el crecimiento y la rentabilidad. Parámetros endógenos son variables inputs que poseen valores que se determinan por otras variables del sistema, por ejemplo: si se desea predecir el precio futuro del gas, una variable a incluir podrían ser los viajes por vacaciones o temporada, ya que los valores del precio del gas pueden subir cuando aumenta la demanda cíclica. Por otro lado, parámetros exógenos son variables inputs que no son influenciadas por otras variables del sistema. Las principales características de este tipo de parámetros son:

  - Están fijas cuando ingresan al modelo y se toman como dadas.

  - Pueden influenciar a las variables endógenas.

  - No se determinan ni se explican por el modelo.

  En el ejemplo anterior del precio del gas, variables exógenas podrían ser el precio de las reservas de petróleo, conflictos sociopolíticos, etc.

- Los Parámetros Estructurados y No Estructurados del modelo: Los datos estructurados comprenden claramente tipos de datos ya definidos, cuyos patrones hacen que sean fácilmente investigables, mientras que datos no estructurados comprenden datos que no son fácilmente investigables, como pueden ser archivos de audio, video y publicaciones de redes sociales. Los datos estructurados (structured data en inglés) residen por lo general en bases de datos relacionales y los no-estructurados tienen cierta estructura interna, pero no está definidaa través de modelos o esquemas predefinidos por datos.

  Generalmente, en un contexto de series de tiempo, datos no estructurados no presentan patrones sistemáticos dependientes del tiempo, mientras que los datos estructurados sí, como tendencia y estacionalidad.

- La Naturaleza Univariable o Multivariable del modelo predictivo: Datos Univariables se caracterizan por ser explicados por una sola variable, como lo indica el nombre. No deben lidiar con causas o relaciones. Sus propiedades descriptivas pueden ser identificadas en algunas estimaciones como tendencia central (media, mediana y moda), dispersion (rango, varianza, maximo, mínimo, cuartil y desviación estándar), y la distribución de frecuencia. El análisis de datos univariables es conocido por su limitación

a la hora de determinar relaciones entre dos o más variables, correlaciones, comparaciones, causas, explicaciones y contingencias entre vriables. No provee generalmente mayor información en variables dependientes e independientes.

En contraste, Datos Multivariables y su análisis resulta no solo mejor para considerar muchas características en un modelo, sino también para traer luz a los efectos de variables exógenas.

El modelo de Forecasting en cuestión podrá ser Univariable o Multivariable. Ejemplos de series de tiempo multivariables son los modelos de ARIMA, de los cuales ampliaremos en mayor detalle luego.

Una última aclaración, que vale la pena realizar, es que el número de variables puede diferir entre inputs y outputs, por ejemplo, los datos de un modelo no necesariamente tienen que ser simétricos. Se puede tener múltiples variables como input y una única variable output a predecir. Cuando abordemos Redes Neuronales Recurrentes podremos hacer la distinción entre modelos sequence-to-sequence, sequence-to-one, etc.

- La Estructura Single-Step o Multi-Step del modelo de Forecasting: Hay modelos que predicen el valor de una sola observación en un período futuro. Esto se denomina one-step forecast (predicción de un paso), ya que se predice un solo período futuro $(t + 1)$. En contraste a esta estructura mencionada, hay problemas multi-step (predicciones de varios pasos), donde el objetivo es predecir una secuencia de valores a lo largo del futuro, es decir, n períodos $(t + n$, para n períodos futuros). Muchos problemas de series de tiempo requieren de una predicción secuencial de valores, usando solamente los valores observados en el pasado. Ejemplos pueden ser: rendimientos agrícolas, precios de acciones en la Bolsa, el volumen de tráfico y movlidad, el consumo de energía eléctrica, entre otros. Hay, al menos, cuatro estrategias popularmente usadas en predicciones multi-step:

  - Directa: el método directo requiere de crear un modelo separado para cada período de tiempo predicho. Por ejemplo, en el caso de predecir el consumo de energía en las próximas dos horas, necesitaríamos un modelo para predecir el consumo en la primera hora $(t + 1)$ y otro modelo separado para la segunda hora $(t + 2)$.

  - Recursiva: también puede ser manejada recursivamente la predicción multi-step, donde un solo modelo de series de tiempo se construye para predecir el siguiente período, y las consiguientes predicciones son computadas usando predicciones previas. Por ejemplo, en nuestro caso del forecasting del consumo de energía en las próximas dos horas, solamente necesitaríamos desarrollar un único modelo one-step. Este se usaría para predecir el consumo en la primera hora $(t + 1)$, y con este valor predicho como input se usa para predecir la segunda hora $(t + 2)$.

  - Híbrida Directa-Recursiva: este método híbrido combina las ventajas de cada uno de los dos. Como ejemplo, un modelo distinto se construye para cada período de tiempo futuro, sin embargo, cada modelo puede usar a su vez como input la predicción del otro modelo para predecir $(t + 2)$. En nuestro ejemplo, se construirían dos modelos, pero el output del primero se usa para el segundo.

  - Multiple Output (Multisalida): la estrategia de multisalida requiere desarrollar un modelo que sea capaz de predecir la secuencia completa, en un solo cómputo. Por

ejemplo, en nuestro contexto de la demanda de energía eléctrica para las próximas dos horas, necesitaríamos de un modelo capaz de predecir las próximas dos horas $(t+1, t+2)$ en un solo procesamiento de los datos.

- Valores Contiguos o No Contiguos de la serie de tiempo: Una serie de tiempo que presenta intervalos temporales consistentes (por ejemplo, cada 5 minutos, cada 2 horas, etc.) entre cada dato, son definidas como *Contiguas*. Por otro lado, series de tiempo que no son uniformes a lo largo del tiempo entre períodos se definen como *No Contiguas*: sin embargo, muchas veces la razón de esta característica se debe a valores faltantes (missing) o corruptos (corrupt). Es importante comprender las razones por las cuales los datos se extravían:

  - Faltantes al azar (missing at random): sucede cuando la propensión a que un punto falte no está relacionada con los datos faltantes, pero si se relaciona con algunos datos observados.

  - Faltantes completamente al azar (missing completely at random): el hecho de que ciertos valores falten no tiene nada que ver con su valor hipotético y con los valores de las otras variables.

  - Faltantes no por azar (missing not at random): dos posibles razones dependen de los valores hipotéticos del dato faltante o del valor de otra variable. Los métodos para tratar con missing values serán analizados en la próxima sección.

1.2.3 Diferencias entre Time Series Analysis y Forecasting

Los científicos de datos suelen aprovechar el **análisis de series de tiempo** por las siguientes razones:

- Adquirir conocimientos claros y desarrollar la intuición sobre los patrones y estructuras subyacentes de los datos de series de tiempo históricas.

- Aumentar la calidad de la interpretación de las características de las series de tiempo para informar mejor el dominio del problema.

- Procesar previamente y realizar feature engineering (ingeniería de parámetros) de alta calidad para obtener un conjunto de datos históricos más rico y profundo.

El time series analysis se utiliza para muchas aplicaciones, como el control de procesos y calidad, estudios de servicios públicos, análisis de censos, entre otros ejemplos. Por lo general, se considera el primer paso analizar y preparar los datos de la serie de tiempo previo al paso de construcción del modelo, que se denomina propiamente **pronóstico de series de tiempo**. El time series forecasting implica tomar modelos de Machine Learning, entrenarlos con datos históricos de series de tiempo y consumirlos para pronosticar valores futuros de la serie. Resumiendo, los principales pasos en el proceso de forecasting son:

1. Alimentar el algoritmo de Machine Learning con datos.
2. Usar estos datos para entrenar el modelo.
3. Testear y montar (deploy) el modelo.

4. Consumir el modelo desplegado para realizar una tarea automática de predicción. En otras palabras, llamar y usar el modelo para recibir las predicciones realizadas por el modelo.

Por último, cabe destacar que las habilidades y herramientas necesarias para cada actividad pueden variar un poco. Para analizar datos, se requiere principalmente de manejo de herramientas como Excel, SQL para trabajar con bases de datos, Power BI para hacer reportes y conocimientos generales de estadística inferencial, que pueden ser implementados en Python, R o SAS, entre otros.

Por otro lado, un científico de datos requiere de habilidades como manejo de bases de datos en SQL y NoSQL, herramientas de Big Data como Hadoop y Spark, buen nivel de programación en Python, R, Julia o Scala, entre otros lenguajes. Este último rol implica una involucración mayor en matemática, estadística avanzada e informática.

El ecosistema de lenguajes de programación, librerias y herramientas entre Data Analysis y Data Science se puede mostrar en la siguiente Figura:

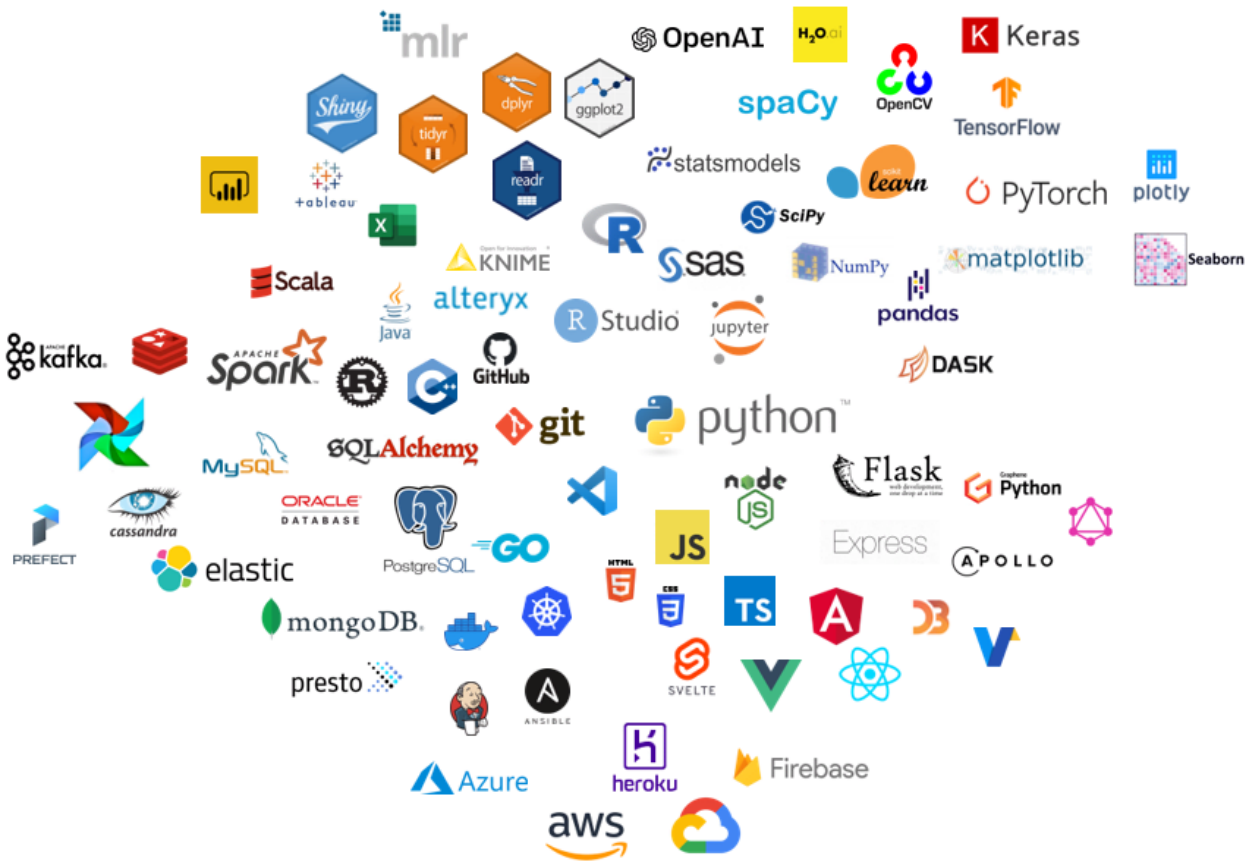


Figure 4: Ecosistema General del Manejo de Datos

2 Diseño de Soluciones Profesionales End-to-End de Forecasting

Hoy en día, el pronóstico de series de tiempo se realiza para una gran variedad de aplicaciones, que incluyen el pronóstico del clima, la predicción de terremotos o desastres naturales, la astronomía, finanzas e ingeniería de control, entre otras. En muchas aplicaciones modernas y del mundo real, la predicción de series de tiempo utiliza tecnologías informáticas, incluida la nube (cloud), la inteligencia artificial y el aprendizaje automático, para crear e implementar soluciones end-to-end, en otras palabras, de un extremo a otro. Para resolver problemas de negocios reales en una determinada industria, es esencial tener un esquema bien estructurado, para que los científicos de datos puedan usarlo como guía y aplicarlo para resolver escenarios del mundo real.

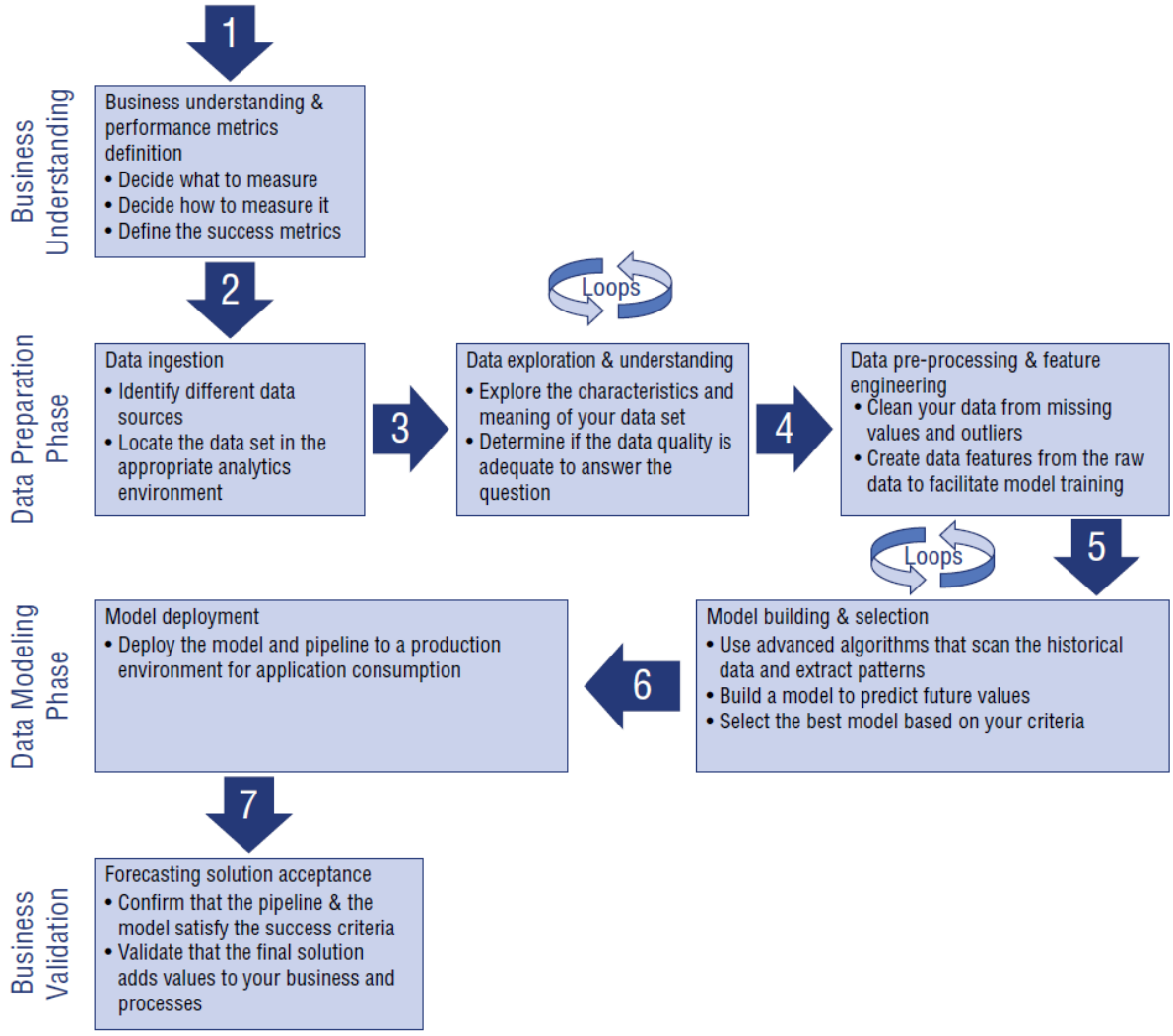


Figure 5: Esquema de un Time Series Forecasting

2.1 Esquema General para Time Series Forecasting

Un esquema o plantilla (en inglés, template) para predicción de series de tiempo es un conjunto de tareas que conllevan desde definir un problema de negocio hasta tener el modelo predictivo ya montado y listo para ser consumido externamente por el usuario final. El esquema de la Figura 5 es un marco ágil e interactivo para entregar una solución eficiente, que contiene una destilación de las mejores prácticas y estructuras que facilitan la implementación exitosa de una solución de forecasting. Como se puede observar, el esquema consiste de diferentes pasos o procesos:

1. Entendimiento del negocio y definición de las métricas de performance
2. Ingestión de datos
3. Exploración de datos y entendimiento
4. Pre-procesamiento de datos e Ingeniería de Parámetros
5. Construcción del modelo y selección
6. Montaje del modelo
7. Aceptación final de la solución

En las subsiguientes secciones profundizaremos en detalle sobre cada proceso del esquema.

2.1.1 Entendimiento del Negocio y Performance Metrics

El (primer) paso donde se definen las métricas de desempeño y se busca comprender el negocio describe el aspecto empresarial específico que debemos entender y considerar antes de tomar una decisión de inversión. Se debe calificar el problema empresarial en cuestión para garantizar que el análisis predictivo y el Machine Learning sean realmente efectivos y además, aplicables. Para muchas organizaciones, la falta de datos no es un problema. Es en realidad todo lo contrario: a menudo hay demasiada información disponible para tomar una decisión clara sobre el futuro. Con tantos datos para clasificar, las organizaciones necesitan una estrategia bien definida para aclarar los siguientes aspectos comerciales:

- ¿Cómo puede la solución de Forecasting ayudar a las organizaciones a transformar el negocio, mejorar costos, y conducir a una mejor operación general?
- ¿Tienen las organizaciones un propósito y visión bien definida y que esté claramente articulado sobre lo que buscan cumplir en el futuro?
- ¿Cómo pueden las organizaciones recibir apoyo de los principales stakeholders para llevar el enfoque de Forecasting y la visión data-driven hacia otras áreas de la empresa?

En este proceso es importante definir preguntas para calificar (o descalificar) soluciones potenciales a un problema u oportunidad específica. Por ejemplo, si una empresa de retail se está enfrentando a costos crecientes y no puede ofrecer precios competitivos, una de las tantas preguntas a resolver puede ser si "la compañía puede reducir sus operaciones sin comprometer la calidad". En este y tantos ejemplos, las principales tareas a llevar a cabo son:

- Definir Objetivos (business goals): las empresas tienen que trabajar con expertos de la industria y otros stakeholders para entender e identificar los problemas.

- Formular las preguntas correctas: las empresas tienen que formular preguntas tangibles que definen a los objetivos, y que la solución de Forecasting pueda atacar.

Luego, para traducir de manera exitosa esta visión y objetivos en resultados, el siguiente paso es establecer métricas de performance: primero hay que decidir qué y cómo medirlo, y luego definir la métrica.

- Decidir qué medir: es importante recolectar los datos correctos. Consideremos el ejemplo de *"predictive maintenance"*, una técnica usada para predecir cuando una máquina (en servicio) va a fallar, para planificar los mantenimientos con anticipación. Este área resulta ser muy amplia en la variedad de objetivos (cómo identificar la causa de la falla, identificar qué partes se reemplazarán, cuando, etc.). Sin embargo, aunque las empresas disponen de muchos datos, por lo general no tienen datos suficientes del historial de fallas, por lo que hace el modelo muy dificil de implementar. Por esto resulta relevante saber qué medir y comenzar a hacerlo cuanto antes.
- Decidir cómo medirlo: las preguntas importantes en el "cómo" dependen de cuál sea el período de tiempo, la unidad de medida y los factores a incluir. Se debe identificar las variables clave que el modelo tiene que predecir. La medición puede llevarse a cabo con sensores electrónicos IoT cuando se trata de medir, por ejemplo, datos de una máquina y también puede llevarse a cabo con plataformas de software para tal fin, como por ejemplo, cuando se trata de datos de tráfico web (Google Analytics).
- Definir las métricas de éxito: luego de definir las variables clave, es importante traducir el problema de negocio a una pregunta de Data Science, y así definir las métricas que definen el éxito de dicho proyecto. Con estos datos se puede hacer un seguimiento de la solución y en cierta parte, reduce el nivel de abandono a las soluciones por parte de las empresas usuarias, que suele suceder cuando no tienen forma de validar progreso y grado de utilidad de la solución, por lo que terminan resignando el contrato de servicio.

Por otro lado, es importante que una organización entienda muy bien el valor económico de operar una solución de forecasting (de corto o largo plazo). De hecho, es importante que se entienda el valor de cada operación de predicción. En el ejemplo del pronóstico de demanda de energía, una predicción precisa del consumo en las próximas 24hs puede evitar sobre-producción o prevenir sobrecargas en el sistema energético (sea que esté conectado a la red eléctrica, con o sin sistemas de energía renovable). Esto se puede cuantificar en ahorro financiero en una base diaria. Una fórmula simple para calcular el beneficio financiera de una solución de forecasting de demanda es:

$$\frac{CA + CE + CT}{T} = \frac{VFF}{T}$$

donde CA es el Costo de Almacenamiento de los datos, CE es el Costo de Egreso (tráfico) de los datos, CT es el Costo de Transacción de la predicción y T es el número de transacciones de la predicción. El primer término es igual, en otras palabras, al cociente de VFF, el Valor Financiero del Forecast sobre T.

2.1.2 Data Ingestion

En la actualidad, las empresas recopilan grandes volúmenes de datos, tanto estructurados como no estructurados, en un esfuerzo por utilizar esos datos para descubrir información en tiempo real o casi en tiempo real que sirve para la toma de decisiones y respalda la transformación digital. La ingestión de datos es el proceso que consta de obtener e importar datos para su uso inmediato o almacenamiento en una base de datos. Estos pueden ser ingeridos de tres formas diferentes: batch, real-time y por streaming.

- Batch (lote): Un escenario común de big data es el procesamiento por lotes de datos en reposo (data at rest). En este contexto, los datos originales se cargan en el almacenamiento de datos, ya sea por la propia aplicación de origen o por un flujo de trabajo de orquestación (orchestration workflow). Luego, los datos se procesan en el lugar mediante una tarea paralelizada.
- Real-time (en tiempo real): El procesamiento en tiempo real se ocupa de flujos de datos que se capturan en vivo y se procesan con una latencia mínima para generar informes o respuestas automáticas en tiempo real (o casi en tiempo real). Por ejemplo, una solución de monitoreo de tráfico y movilidad en la vía pública en tiempo real podría usar datos de sensores para detectar grandes volúmenes de tráfico por autos. Estos datos podrían usarse para actualizar dinámicamente un mapa que muestre la congestión o iniciar otros sistemas de gestión del tráfico.
- Streaming (transmisión): Con la transmisión de datos, los datos se procesan inmediatamente a partir de los flujos de entrada de datos. Después de capturar los mensajes en tiempo real, la solución debe procesarlos filtrando, agregando y preparando los datos para su análisis.

En el caso del pronóstico de la demanda, los datos del consumo energético deben predecirse de manera constante y frecuente, y debemos asegurarnos de que los datos sin procesar fluyan mediante un proceso de ingestión de datos sólido y confiable. El proceso de ingestión debe garantizar que los datos brutos (raw data) estén disponibles para el proceso de Forecasting en el momento requerido. Eso implica que la frecuencia de ingestión de datos debe ser mayor que la frecuencia de pronóstico. En otras palabras, si nuestra solución de Forecasting generara una nueva predicción a las 8:00 am todos los días, entonces debemos asegurarnos de que todos los datos que se han recopilado durante las últimas 24 horas se hayan ingerido por completo hasta ese momento y deben incluir la última hora de datos también.

2.1.3 Exploración de datos y Entendimiento

La exploración de datos es el primer paso en el análisis de datos y generalmente implica resumir las características principales del dataset, incluido su tamaño, precisión, patrones iniciales en los datos y otros atributos. La fuente de datos sin procesar que se requiere para realizar pronósticos confiables y precisos debe cumplir con algunos criterios básicos de calidad. Aunque se pueden utilizar métodos estadísticos avanzados para compensar algunos posibles problemas de la calidad de los datos, aún debemos asegurarnos de que estamos cruzando algún umbral de calidad básico al ingerir nuevos datos.

Algunos aspectos a considerar en la calidad de raw data (datos brutos) son:

- Valores faltantes (missing values): como ya comentamos anteriormente, es la situación que refiere a mediciones específicas faltantes en la base de datos, que no fueron colectadas. El requerimiento básico en este caso es que el ratio de valores faltantes no sea mayor a 10 porciento para ningún período de tiempo dado.

$$\text{RVF} = \frac{n_M}{N} \cdot 100$$

donde RVF es el Ratio de Valores Faltantes, n_M es el número de valores faltantes (missing) y N es la cantidad total de observaciones. En línea con las razones explicadas por las que pueden faltar datos (al azar, completamente al azar y con algun patrón), existen diversas formas de tratamiento de los missing values. En los primeros dos casos (al azar y completamente al azar), es seguro poder eliminar los datos con valores faltantes dependiendo de la frecuencia de ocurrencia, mientras que en el tercer caso (cuando hay un patrón de la ausencia de los datos) eliminar dichas observaciones puede generar un sesgo en el modelo. Existen diferentes soluciones para la imputación de datos[2] según el tipo de problema que se está intentando resolver, y es difícil dar una solución general. Los métodos más comunes se pueden observar en la Figura 6.

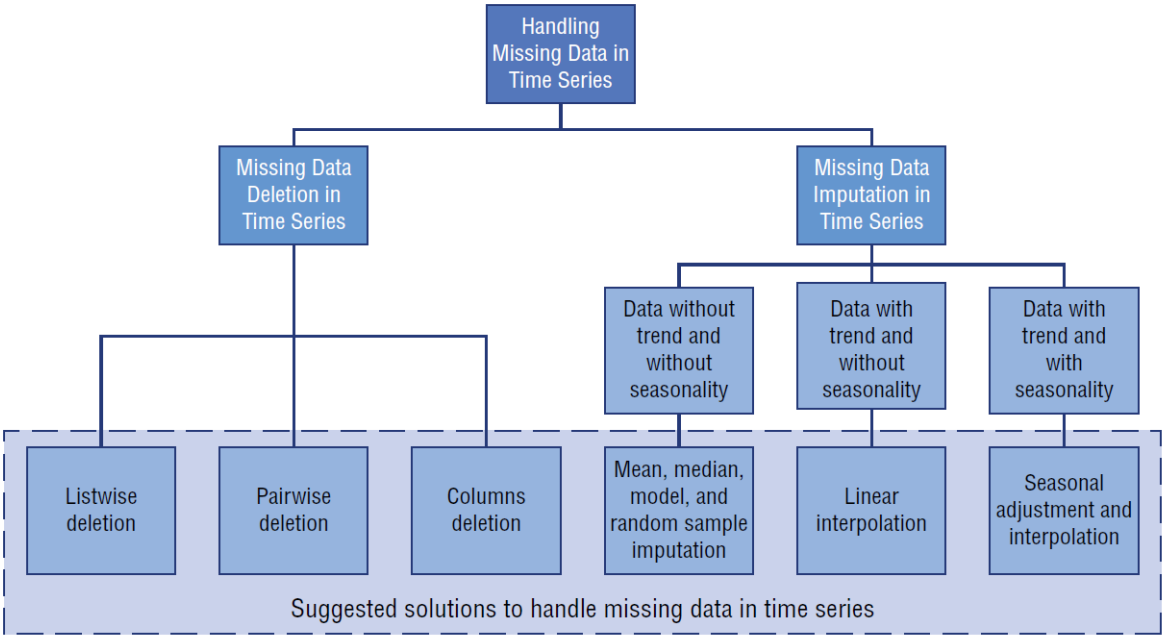


Figure 6: Tratamiento de Missing values

Como se puede observar en la Figura 6, la eliminación por lista (*listwise deletion*) elimina todos los datos de una observación que tiene uno o más valores perdidos. En particular, si los datos faltantes se limitan a una pequeña cantidad de observaciones, puede optar por eliminar esos del análisis. Sin embargo, en la mayoría de los casos es desventajoso utilizar la eliminación por lista, ya que los supuestos de que los datos faltan completamente al azar son generalmente dificiles de respaldar. En consecuencia, los métodos de eliminación por listas producen parámetros y estimaciones sesgados.

[2]La imputación de datos, en estadística, es la sustitución de valores faltantes en una observación por otros.

La eliminación por pares (*pairwise deletion*) analiza todos los casos en los que están presentes las variables de interés y, por lo tanto, maximiza todos los datos disponibles mediante una base de análisis. Un punto fuerte de esta técnica es que aumenta el poder en el análisis, pero tiene muchas desventajas. Asume también que los missing values faltan completamente al azar. Si se elimina por pares, se terminará con diferentes números de observaciones que contribuyen a diferentes partes del modelo, lo que puede dificultar la interpretación.

Eliminar columnas (*deleting columns*) es otra opción, pero siempre es mejor conservar los datos que eliminarlos. A veces, se puede descartar variables si faltan datos para más del 60 por ciento de las observaciones, pero solo si esa variable es insignificante también. Dicho esto, la imputación es siempre una opción preferida por sobre la eliminación de variables. Métodos de imputación, para series de tiempo, hay pocas opciones:

- Interpolación lineal (*Linear interpolation*): este método trabaja bien para una serie de tiempo con algo de tendencia, pero no es ajustable para datos con estacionalidad.

- *Ajuste estacional e Interpolación lineal*: este método es mejor para datos con estacionalidad y tendencia.

- *Media, Mediana y Moda*: Computando la media general, mediana o moda es un método muy básico de imputación, y es la única función probada que no aprovecha las características de la serie temporal o la relación entre las variables. Es muy rápido, pero tiene también desventajas. Una de estas es que la imputación con la media reduce la varianza en el conjunto de datos.

- Precisión de la medición (measurement accuracy): muchos datos, dependiendo el problema empresarial subyacente, deben ser medidos con mucha precisión. Mediciones poco precisas van a generar pedicciones poco precisas. Típicamente, el error de medición debería ser más bajo que 1% relativo al valor verdadero. $ME \leq 0.01$
- Tiempo de medición: es también requerido que el período de tiempo (time stamp) actual del dato colectado no se desvíe por más de 10 segundos relativo al tiempo correcto del dato.
- Sincronización: cuando se usan muchas fuentes de datos, necesitamos asegurarnos que no haya problemas de sincronización de tiempo entre estas. Esto significa que la diferencia de tiempo de un período entre una fuente de datos y otra, tampoco debería exceder más de 10 segundos.
- Latencia: como se comentó en el proceso de Ingestión de Datos, dependemos que el flujo de datos sea seguro. Para controlarlo, nos debemos asegurar que controlamos la latencia de los datos. La latencia significa la diferencia de tiempo entre que se tomó la medición del dato y el tiempo en que se cargó al dataset. En procesos automáticos es relevante poder controlarlo.

La fase de preparación de datos consiste básicamente en tomar los datos sin procesar y convertirlos (transformarlos, remodelarlos) en una forma específica para la fase del modelado de la solución. Eso puede incluir operaciones simples como usar la columna de datos sin procesar tal cual está con su valor medido real o con valores estandarizados o con operaciones más complejas como rezagando los datos. Las columnas de datos recién creadas se denomi-

nan características (features) de datos y el proceso de generarlas se denomina ingeniería de características (feature engineering). Al final de este proceso, tendríamos un nuevo conjunto de datos que se derivó de los datos sin procesar y que se puede utilizar para modelar.

2.1.4 Pre-procesamiento de datos e Ingeniería de Parámetros

El preprocesamiento de datos y la ingeniería de características es el paso en el que los científicos de datos limpian los conjuntos de datos de valores atípicos y datos faltantes, y crean características adicionales con los datos sin procesar para luego alimentar los modelos de Machine Learning. Específicamente, la ingeniería de características es el proceso de transformar datos en características para que actúen como entradas (inputs) para los modelos de ML, de modo que las características o parámetros de buena calidad ayuden a mejorar el rendimiento general del modelo. Los parámetros a crear también dependen en gran medida del problema subyacente que estamos tratando de resolver con una solución de Forecasting.

En Machine Learning, una característica es una variable cuantificable del fenómeno que se está tratando de analizar, generalmente representada por una columna en el dataset. Cada fila del dataset, para estas y otras columnas, será un feature vector. Y el conjunto de características a lo largo de las observaciones forman una matriz de features, también conocida como set de características. Así, los algoritmos trabajan con matrices numéricas o tensores, por lo que el proceso de ingeniería de características consta de técnicas para convertir raw data (datos brutos) en representaciones numéricas que puedan alimentar los algoritmos. Hay dos tipos de parámetros o características principalmente:

- Características brutas inherentes (inherent raw features) son las que ya pertenecían al dataset original y sin necesidad de manipular los datos, se extraen directamente como son.
- Características derivadas (derived features) son aquellas que se crean a partir de la manipulación de los datos o con ingeniería de características, en este último paso, los científicos de datos deben extraer características de los atributos existentes de los datos.

  Algunos de los parámetros derivados más usuales son:

  - Time Driven features: estas características se derivan de las fechas o períodos de tiempo en los datos, como pueden ser tiempo del día, día de la semana, día del mes, fines de semana, días festivos o inclusive "términos de Fourier", que son ponderaciones derivadas del período de tiempo usadas para capturar la estacionalidad en los datos.

  - Independent measurement features: las características independientes incluyen todos los elementos de los datos que queremos usar como predictores en nuestro modelo, como pueden ser lag features (rezagos), es decir, los valores de cierta variable en períodos anteriores ($t-1; t-2; t-3$, etc). y también tendencias de largo plazo, como una característica que capture la tendencia a lo largo de los años.

  - Dependent feature: los parámetros dependientes son aquellas variables que necesitamos que nuestro modelo prediga (es la variable explicada, generalmente y_i), son los labels o etiquetas.

2.1.5 Construcción del modelo y Selección

La fase de modelado es donde tiene lugar la conversión de los datos existentes en un modelo o sistema completo. En el núcleo de este proceso hay algoritmos avanzados que "escanean" los datos históricos (datos de entrenamiento o training set), extraen los patrones y construyen un modelo. Ese modelo se puede usar más tarde para predecir nuevos datos (test set) que no se han usado para construir dicho modelo. Una vez que tengamos un modelo confiable que funcione, podemos usarlo para calificar nuevos datos que estén estructurados para incluir las características requeridas. El proceso de puntuación (scoring) hará uso del modelo persistente (objeto de la fase de entrenamiento) y predecirá la variable objetivo.

En este punto, resulta relevante poder distinguir entre los datasets que son para entrenamiento, validación y testeo.

- Train data set: El data set de entrenamiento representa la cantidad total de datos que deben usarse para ajustar (fit) o entrenar el modelo de Machine Learning. A través de este, un algoritmo puede ser entrenado para aprender de datos históricos y poder, luego, predecir puntos de datos futuros.
- Validation data set: El data set de validación es una cantidad, bastante más pequeña, de datos que sirven para una evaluación (insesgada) del ajuste del modelo a los datos de entrenamiento, mientras que se calibran los hiperparámetros de dicho modelo (hyperparameter tuning). Los científicos de datos generalmente aprovechan los datos de validación para afinar (fine-tune) los hiperparámetros de un modelo de Machine Learning. Los hiperparámetros son variables adicionales de un modelo, cuyos valores se utilizan para controlar y mejorar el proceso de aprendizaje de un modelo. Por ejemplo, en el algoritmo de optimización, si fuera un Gradiente Descendiente, un hiperparámetro es la tasa de aprendizaje (learning rate) o constante con la que se mueve en dirección contraria del gradiente.
- Test data set: Este set de datos, usado para "testear" un modelo, ayuda a determinar si dicho modelo está subajustado (underfitting, es decir, ajusta mal en el set de entrenamiento y hay un grado alto de sesgo) o si está sobreajustado (overfitting, o sea, ajusta demasiado bien el set de entrenamiento y no puede generalizar luego. Hay un grado alto de varianza).

Ya explicadas las características y diferencias de cada data set, una de las decisiones importantes es cómo dividir todo el conjunto de datos disponible entre estas tres categorías. Si tenemos, por ejemplo, 100.000 datos, ¿cómo los distribuimos para poder tener estos tres tipos de set de datos? Los factores más importantes que definen esta decisión son:

- El número total de datos (samples): algunos modelos necesitan de una cantidad relativamente grande de datos para poder entrenarse, por lo que en este caso necesitaríamos priorizar un training set más grande.
- El modelo que estamos entrenando: si el modelo posee pocos hiperparámetros que deban ser calibrados, serán también fáciles de validar y ajustar, por lo que probablemente podamos reducir el tamaño de nuestro conjunto de validación. Por el contrario, si el modelo posee muchos hiperparámetros, necesitamos entonces un set de validación más grande. Asimismo, en contraste, si el modelo no posee hiperparámetros o estos son muy difíciles de ajustar, entonces no necesitamos de un conjunto de validación directamente.

De todas maneras, aunque están estos factores mencionados a considerar, una regla general que se utilizaba mucho por convención en la industria pre-big data, era de 70/15/15 o si no hubiera set de validación 70/30, aunque hoy día no se utilizan más.

El dev set debe ser lo suficientemente grande como para detectar diferencias entre los algoritmos que se están probando. Por ejemplo, si el clasificador A tiene una precisión del 90.0% y el clasificador B tiene una precisión del 90.1%, entonces en un dev set de 100 ejemplos no se podría detectar esta diferencia del 0.1%. En comparación con otros problemas de aprendizaje automático, un dev set de 100 ejemplos es pequeño. Los dev set con tamaños de 1000 a 10.000 ejemplos son comunes. Con 10.000 ejemplos, habrá mejores posibilidades de detectar una mejora del 0.1%.

Para aplicaciones un poco más maduras e importantes, por ejemplo, publicidad, búsqueda web y recomendaciones de productos, hay equipos de científicos de datos que están más motivados para lograr incluso una mejora del 0.01%, ya que tiene un impacto directo en las ganancias de la empresa. En este caso, el conjunto de dev podría ser mucho mayor que 10.000, para detectar mejoras aún más pequeñas.

¿Y qué tal el tamaño del test set? Debe ser lo suficientemente grande para brindar una gran confianza en el rendimiento general del sistema. Como mencionamos, una heurística popular había sido utilizar el 30% de los datos para el test set. Esto funciona bien cuando se tiene un número modesto de ejemplos, supongamos de 100 a 10.000 samples. Pero en la era del big data, donde ahora hay problemas de aprendizaje automático con, a veces, más de mil millones de ejemplos, la fracción de datos asignados a los conjuntos de desarrollo/prueba se ha reducido mucho. No es necesario tener conjuntos de desarrollo/prueba excesivamente grandes más allá de lo necesario para evaluar el rendimiento de sus algoritmos.

Si bien es relativo a cada problema, una referencia podría ser un test de 2 a 3% de los datos totales cuando se cuenta con cientos de miles de samples.

Por último, resta ampliar sobre un rol crítico dentro del modeling step: la evaluación del modelo. En esta parte buscamos validar el modelo y su performance con datos reales. Previamente, usamos parte de los datos disponibles para entrenar el modelo y durante la fase de evaluación, tomamos el remanente de datos para testear el modelo. Cuantificar una medida de error es clave, ya que buscamos luego afinar/calibrar el modelo y validar si el error de pronóstico está decreciendo o no. El fine-tuning puede ser realizado al modificar los hiperparámetros del modelo (que controlan el proceso de aprendizaje) o bien al agregar o remover características/parámetros (conocido como parameters sweep). En la práctica, eso significa que es posible que tengamos que iterar entre las fases de ingeniería de características, modelado y evaluación del modelo varias veces hasta que podamos reducir el error al nivel requerido.

Es importante enfatizar que el error de predicción nunca será cero, ya que nunca habrá un modelo que pueda predecir perfectamente todos los resultados. Sin embargo, existe una cierta magnitud de error que es aceptable para la empresa u organización (y lo podrá determinar un experto de la industria o business expert). Durante el proceso de validación, nos gustaría asegurarnos de que el error de predicción de nuestro modelo sea superior o igual al nivel de tolerancia elegido (por la empresa). Por lo tanto, es importante poder establecer dicho

nivel del error tolerable al comienzo del ciclo, es decir, durante la fase de formulación del problema.

Hay varias formas de medir y cuantificar el error de predicción. En concreto, existe una técnica de evaluación relevante para las series temporales y en específico para la previsión de la demanda: el MAPE. MAPE significa Mean Absolute Percentage Error (error porcentual absoluto medio). Con MAPE estamos calculando la diferencia entre cada punto pronosticado y el valor real de ese punto. Luego cuantificamos el error por punto calculando la proporción de la diferencia relativa al valor real. En el último paso promediamos estos valores. La fórmula matemática utilizada para MAPE es la siguiente:

$$\text{MAPE} = \left(\frac{1}{N} \cdot \sum_{t=1}^{N} \frac{|A_t - F_t|}{|A_t|} \right) \cdot 100$$

donde A_t son Actual Values (valores reales) y F_t son Forecasted Values (valores pronosticados). El MAPE es sensible a la escala y no debe usarse cuando se trabaja con un bajo volumen de datos. Notese que debido a que los valores reales también son el denominador de la ecuación, el MAPE no está definido cuando la demanda real es igual a cero. Además, cuando el valor real no es cero, pero es bastante pequeño, el MAPE a menudo tomará valores extremos. Esta sensibilidad de escala hace que el MAPE sea casi inútil como medida de error para data sets de bajo volumen. De todas maneras, hay también otras técnicas de evaluación que se pueden mencionar:

- Mean Absolute Deviation (MAD) - Esta fórmula mide el tamaño del error en unidades. El MAD es una buena estadística para usar cuando se analiza el error para un solo punto. Sin embargo, si se agrega el MAD a nivel de múltiples puntos, hay que ser cuidadosos con los productos de alto volumen dominando los resultados. El MAPE y el MAD son las medidas estadísticas de error más utilizadas.

$$\text{MAD} = \frac{1}{N} \sum_{i=1}^{N} |x_i - \overline{x}|$$

- Mean Absolute Deviation (MAD)/Mean Ratio - Esta es una alternativa al MAPE que se ajusta mejor para datos intermitentes y de bajo volumen también. Como se explicó, los porcentajes de error no se puede calcular cuando el valor real es igual a cero, y también pueden tomar valores extremos cuando se usan conjuntos de datos pequeños. Estos inconvenientes se agrandan cuando se comienza a promediar MAPEs a lo largo de multiples series de tiempo. El MAD/Mean Ratio busca sobrellevar este problema al dividir el MAD por la media, esencialmente reescalando el error para hacerlo comparable a lo largo de las series de tiempo de diferentes escalas.

- Geometric Mean Relative Absolute Error (GMRAE) - Esta métrica se usa para medir la performance de la predicción out-of-sample. Es calculada al usar el error relativo entre el modelo naïve y el modelo seleccionado actualmente. Un GMRAE de 0.54 indica que el tamaño del modelo actual es solo 54 porciento del tamaño del error generado usando el modelo naïve para el mismo conjunto de datos. Como se basa en el error relativo, es mucho menos sensible a la escala que el MAPE o el MAD. Su fórmula es:

$$\text{GMRAE} = \sqrt[N]{\prod_{t=1}^{N} \left| \frac{A_t - F_t}{A_t - F_t^*} \right|}$$

- Symmetric Mean Absolute Percentage Error (SMAPE) - Esta es una variación del MAPE que se calcula usando el promedio del valor absoluto del actual sobre el valor absoluto del valor predicho (en el denominador). Esta estadística se prefiere sobre el MAPE por algunos científicos de datos. La fórmula es:

  $$\text{SMAPE} = \frac{1}{N} \sum_{t=1}^{N} \frac{|F_t - A_t|}{(|A_t| + |F_t|)/2}$$

  Esta versión con los valores absolutos en el denominador es para evitar el problema de que puede ser negativo si $A_t + F_t < 0$ o incluso indefinido si $A_t + F_t = 0$. A diferencia del error porcentual absoluto medio (MAPE), el SMAPE tiene un límite inferior y un límite superior. La fórmula anterior proporciona un resultado entre 0% y 200%. Sin embargo, un error porcentual entre 0% y 100% es mucho más fácil de interpretar. Esa es la razón por la que la fórmula a continuación se usa a menudo en la práctica (es decir, sin factor 0.5 en el denominador):

  $$\text{SMAPE} = \frac{100\%}{N} \sum_{t=1}^{N} \frac{|F_t - A_t|}{|A_t| + |F_t|}$$

  Un supuesto problema con el SMAPE es que no es simétrico, ya que los pronósticos excesivos e insuficientes no se tratan por igual. Esto se ilustra en el siguiente ejemplo aplicando la fórmula anterior:

  Overforecast (Sobreprevisión): At = 100 y Ft = 110 es un SMAPE = 4.76%

  Underforecast (Previsión insuficiente): At = 100 y Ft = 90 es un SMAPE = 5.26%. Sin embargo, solo se debe esperar este tipo de simetría para medidas que se basan completamente en diferencias y no son relativas (como el error cuadrático medio y la desviación absoluta media).

Después de seleccionar el mejor modelo para la solución de Forecasting, los científicos de datos generalmente necesitan desplegarlo/implementarlo. En la siguiente sección, analizaremos más de cerca el proceso de implementación, es decir, el método mediante el cual integra un modelo de Machine Learning en un entorno de producción existente para comenzar a usarlo y poder tomar decisiones comerciales prácticas basadas en datos (data driven decisions). Es una de las últimas etapas del ciclo propuesto.

2.1.6 Implementación del modelo

Si bien esta subsección puede ser de las más largas y abarcativas en términos técnicos, ya que refiere a tecnologías informáticas muy específicas, haremos una introducción a nivel general para poder ganar intuición al respecto.

La implementación del modelo es el método mediante el cual se integra un modelo de Machine Learning en un entorno de producción existente para comenzar a usarlo para tomar

decisiones comerciales prácticas basadas en datos. En otras palabras, un algoritmo de aprendizaje automático se convierte en un servicio web. Es una de las últimas etapas del ciclo de vida del aprendizaje automático y puede ser una de las más engorrosas también. A menudo, los sistemas de IT de una organización son incompatibles con los lenguajes tradicionales de creación de modelos, lo que obliga a los científicos de datos y a los programadores a dedicar un tiempo valioso y capacidad intelectual a reescribirlos. Nos referimos a este proceso de conversión (del algoritmo a un servicio web) como "operacionalización": poner en funcionamiento un modelo de Machine Learning significa transformarlo en un servicio consumible e integrarlo en un entorno de producción existente.

Cuando pensamos en el aprendizaje automático, centramos nuestra atención en los componentes clave del workflow, como las fuentes, procesamiento y la ingestión de datos, el modelado, entrenamiento y las pruebas, cómo diseñar nuevas funciones y qué variables usar para hacer los modelos más precisos. Todos estos pasos son importantes, sin embargo, pensar en cómo vamos a consumir esos modelos y datos a lo largo del tiempo también es un paso crítico en cada desarrollo de solución de Machine Learning. Solo podemos comenzar a extraer valor económico real y beneficios comerciales de las predicciones de un modelo cuando se haya implementado y puesto en funcionamiento. En la Figura 7 se puede visualizar el flujo de trabajo (workflow).

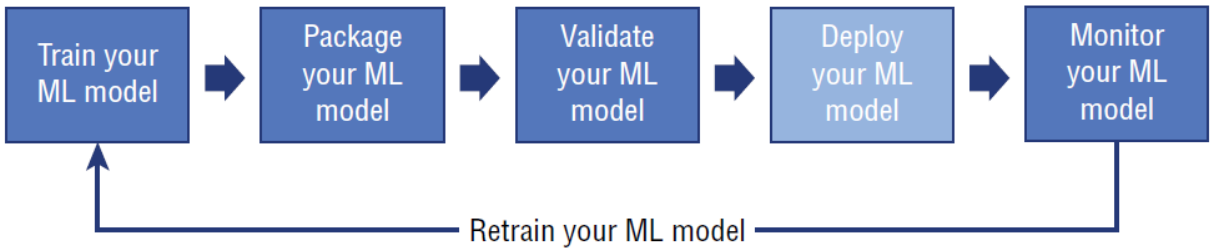


Figure 7: Workflow de Machine Learning

Las ventajas de que una implementación de un modelo sea éxitosa para las empresas que se basan en datos son:

- La implementación o montaje del modelo implica disponibilizar los modelos a clientes externos u otros equipos y stakeholders en tu empresa.
- Al ser usados por otros equipos, estos envían datos a los modelos para obtener sus predicciones, que a su vez se reintroducen en el sistema para aumentar la cantidad (y calidad) de los datos de entrenamiento.
- Una vez los usuarios pueden interactuar considerablemente con el modelo, las empresas empiezan a construir y desplegar muchos más modelos de Machine Learning.

Desde el primer día de la creación de una solución de Machine Learning, los equipos de ciencia de datos deben interactuar con sus contrapartes comerciales (business experts). Es esencial mantener una interacción constante para comprender el proceso de experimentación del modelo en paralelo a los pasos de implementación y consumo del modelo. La mayoría de las organizaciones luchan por desbloquear el potencial del Machine Learning para optimizar sus procesos operativos y lograr que los científicos de datos, los analistas y los equipos

comerciales hablen el mismo idioma.

La implementación debería basarse en los siguientes pasos:

1. Registrar el modelo: un modelo registrado es un contenedor lógico para uno o más archivos que componen el modelo. Por ejemplo, si se dispone de un modelo que está almacenado en muchos archivos, se pueden registrar como un único modelo en el espacio de trabajo. Luego del resgitro, se puede descargar (o implementar) el modelo registrado y recibir todos los archivos. La forma más común es registrar los modelos en un workspace de Azure, Amazon Web Services, Google Cloud, entre otros.
2. Preparar la implementación: para montar el modelo como un servicio web, se debe crear una configuración inferencial y una de implementación. Inferencia, o puntaje del modelo, es la fase donde el modelo implementado se usa para la predicción, más comunmente en datos de producción. Se deben especificar los programas y dependencias necesarias para servir al modelo. En la configuración de deployment, se especifican detalles sobre cómo entregar el modelo en destino.
3. Implementar el modelo en destino.

En nuestro ejemplo, al implementar una solución de Forecasting de demanda, nos interesa implementar una solución end-to-end que vaya más allá del servicio web de predicción y facilite todo el flujo de datos completo. En el momento en que solicitamos una nueva predicción, deberíamos asegurarnos de que el modelo se alimente con las variables de datos actualizadas. Eso implica que los datos brutos recién recopilados se ingieren, procesan y transforman constantemente en el conjunto de características requeridas sobre las que se construyó el modelo. Al mismo tiempo, nos gustaría que los datos previstos estén disponibles para los clientes o consumidores finales. Estos serían los pasos que pertenecen al ciclo de un pronóstico de demanda de energía:

- Millones de medidores de datos implementados generan constantemente los datos de consumo de energía en tiempo real.
- Estos datos se recopilan y cargan en un repositorio en la nube.
- Antes de ser procesados, los datos sin procesar se agregan a una subestación o nivel regional según lo definido por la empresa.
- El procesamiento de características tiene lugar y produce los datos necesarios para el entrenamiento o la puntuación del modelo; los datos del conjunto de características se almacenan en una base de datos.
- El servicio de reentrenamiento se invoca para reentrenar el modelo; esa versión actualizada del modelo se conserva para que pueda ser utilizada por el servicio web de calificación.
- El servicio web de puntuación (scoring) se invoca en un horario que se ajusta a la frecuencia de pronóstico requerida.
- Los datos previstos se almacenan en una base de datos a la que puede acceder el cliente final.

- El cliente recupera los pronósticos, los aplica de nuevo en la red y los consume de acuerdo a su uso.

Los pasos mencionados anteriormente se pueden describir en un esquema (Figura 8).

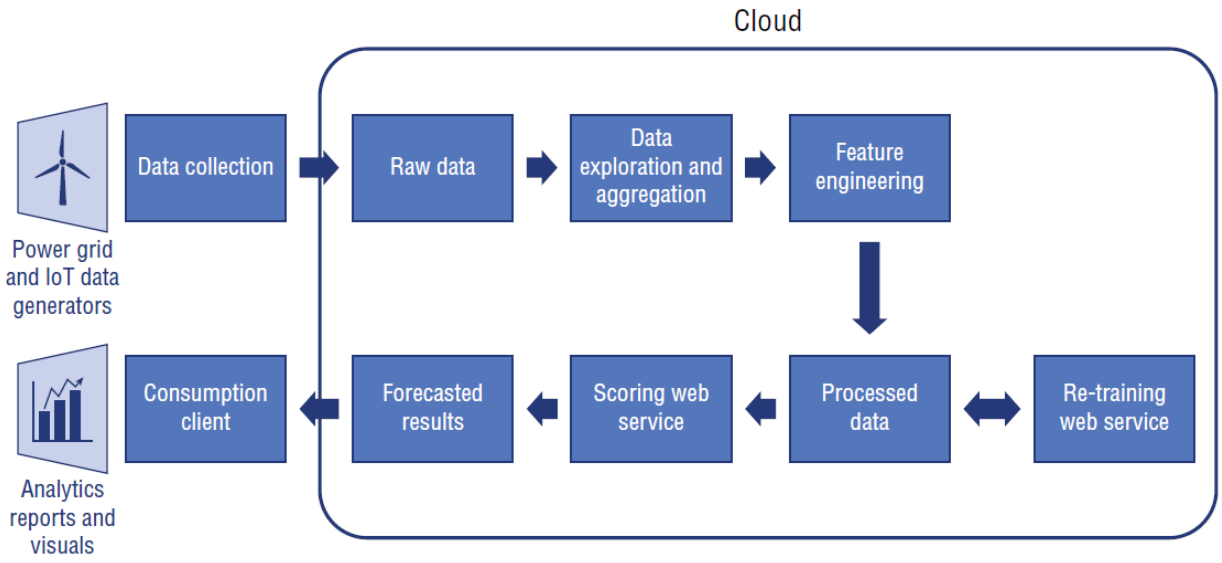


Figure 8: Solución End-to-End de Pronóstico de Demanda de Energía

Dependiendo de los requisitos comerciales, las predicciones se realizan en tiempo real o por lotes (batch basis). Para implementar modelos, se los expone con una API abierta. La interfaz permite que el modelo se consuma fácilmente desde varias aplicaciones, como websites online, spreadsheets (hojas de cálculo), dashboards, aplicaciones de un negocio y aplicaciones de backend.

2.1.7 Aceptación final de la solución

Finalmente, la última fase de nuestro esquema es la de una aceptación de la solución creada. En la última etapa del desarrollo de la solución de predicción de series de tiempo, los científicos de datos deben confirmar que la canalización (el pipeline), el modelo y su implementación en un entorno de producción satisfacen los objetivos de los clientes y los usuarios finales. Suponiendo que una organización tiene todos los ingredientes correctos, incluida una cultura organizacional adecuada, aún necesitan tener la plataforma tecnológica correcta para respaldar la productividad de los científicos de datos y ayudarlos a innovar e iterar rápidamente. Un entorno moderno de análisis basado en la nube (cloud) puede hacer muy simple las tareas de colectar, analizar y experimentar datos, para luego poner en todo en producción. Hay dos tareas principales que se abordan en la etapa de aceptación del cliente:

- Validación del sistema: confirmar que el modelo implementado y la canalización satisfacen las necesidades del cliente.
- Traspaso del proyecto: entregar el proyecto a la entidad que va a ejecutar dicho sistema en producción.

2.2 Técnicas de Modelización de Predicciones de Demanda

La capacidad de pronosticar con precisión una secuencia de datos en el futuro es fundamental en muchas industrias: finanzas, supply chain e industrias manufactureras son tan solo algunos ejemplos. Las técnicas clásicas de series de tiempo han servido para esta tarea durante décadas, pero ahora los métodos de Deep Learning, similares a los que se utilizan en Computer Vision y traducción automática de idiomas (Natural Language Processing), también tienen el potencial de revolucionar la predicción de series de tiempo.

A continuación, podemos nombrar brevemente algunas de las principales técnicas de modelización usadas para predicciones de demanda:

- Moving Average (MA) - La Media Móvil o Promedio Móvil es una de las primeras técnicas analíticas que se utilizaron en el área de pronóstico de series de tiempo, y sigue siendo hoy muy usada también en análisis preliminares y básicos. Es también el cimiento de muchas otras ténicas de Forecasting más avanzadas. Con una media movil, estamos predeciendo un dato siguiente (futuro) al promediar sobre los K puntos recientes, donde K denota el orden de la media movil (por ejemplo, orden 15, orden 30, etc). Esta técnica también tiene el efecto de suavizar el pronóstico, por lo que no es capaz de manipular magnitudes considerables de volatilidad.
- Exponential Smoothing - Esta es una familia de varios métodos que usan promedios ponderados de puntos de datos recientes para predecir el punto siguiente. La idea es asignar mayores ponderaciones a los valores más recientes y gradualmente reducir las ponderaciones para los valores más viejos en tiempo.
- Autoregressive Integrated Moving Average (ARIMA) - Procesos Autorregresivos Integrados de Medias Móviles, por su nombre en español, es otra familia usada comunmente para pronósticos de series de tiempo. Prácticamente combina métodos de autoregresión con medias móviles. Los métodos de autoregresión usan modelos de regresión, pero tomando como serie de tiempo todos los datos previos históricos, prediciendo así valores futuros. ARIMA, además, aplica metodos de diferenciación que incluyen calcular la diferencia entre un puntos de datos y usarlas para la regresión, en vez de usar los valores originales. Por último, también usa técnicas de promedios moviles, que discutimos anteriormente. La combinación de todos estos métodos, en diferentes formas, es lo que construye la familia de métodos ARIMA. Además, modelos ETS (que singifica Error, Trend, Seasonality) y ARIMA son usados ampliamente hoy día para los pronósticos de series de tiempo (ver Figura 9). En muchos casos, los modelos combinados juntos pueden acercar una solución precisa.
- General Multiple Regression - Este puede ser el enfoque de modelado más importante dentro del dominio del Machine Learning y la estadística. En el contexto de series de tiempo, usamos regresión lineal múltiple para predecir valores futuros (por ejemplo, de una demanda). En regresión, durante el proceso de entrenamiento tomamos una combinación lineal de los predictores (parámetros o variables explicativas) y aprendemos los ponderadores (weights o coeficientes β) de dichos predictores. El objetivo es producir una regresión lineal que prediga nuestro valor futuro. Los valores a predecir pueden ser continuos (valores numéricos, por ejemplo de 0 a 100) o discretos (con regresión logística, por ejemplo 0 o 1). Hay muchos métodos de regresión, algunos más simples como la regresión lineal múltiple y algunos más avanzados como árboles de decisión

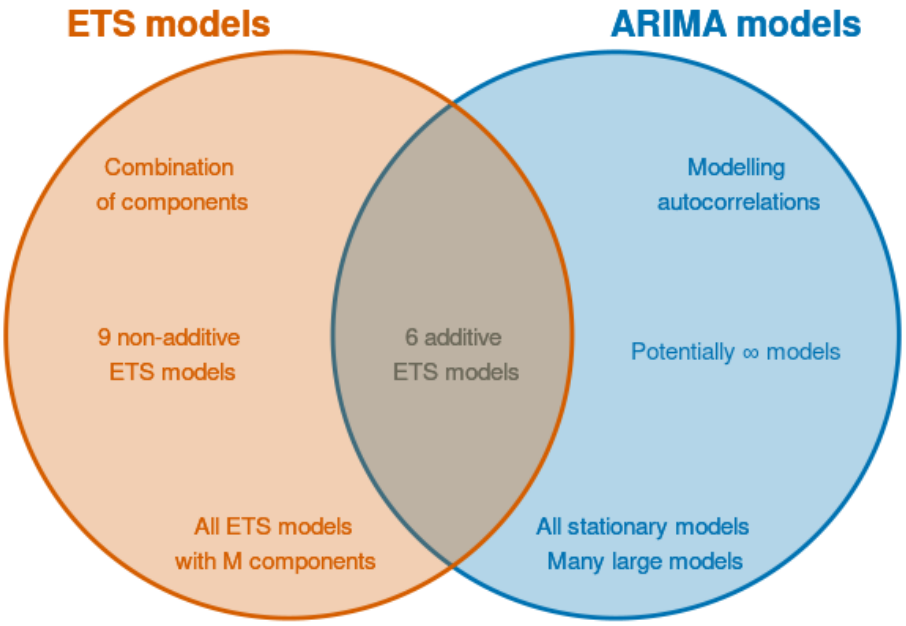


Figure 9: Relación entre modelos ETS y ARIMA

(decision trees), bosques aleatorios (random forests), redes neuronales (neural networks) y bosques aleatorios impulsados (boosted decision trees).

Debido a su aplicabilidad a muchos problemas de la vida real, como la detección de fraudes, el filtrado de correo electrónico no deseado (spam), las finanzas, el diagnóstico médico, etc. y su capacidad para producir resultados procesables, las redes neuronales de aprendizaje profundo ganaron mucha atención en los últimos años. En general, los métodos de Deep Learning se han desarrollado y aplicado a escenarios de predicción de series de tiempo univariables, donde la serie de tiempo consiste en observaciones únicas registradas secuencialmente en incrementos de tiempo iguales.

Por esta razón, a menudo las redes neuronales univariables se desempeñaron peor que los métodos de pronóstico clásicos (e ingenuos), como el suavizado exponencial y ARIMA. Esto llevó a una malinterpretación generalizada de que los modelos de aprendizaje profundo son ineficientes en escenarios de predicción de series de tiempo, y muchos científicos de datos se preguntan si es realmente necesario agregar otra clase de métodos, como redes neuronales convolucionales o redes neuronales recurrentes, a su conjunto de herramientas de series de tiempo. El aprendizaje profundo (DL) es un subconjunto de algoritmos de aprendizaje automático (ML) que aprenden a extraer estas características al representar los datos de entrada como vectores (input) y transformarlos con una serie de operaciones de álgebra lineal en una salida determinada (output). Luego, se evalua si dicha salida es lo que se esperaba usando una ecuación llamada función de pérdida (loss function). El objetivo del proceso es utilizar el resultado de la loss function de cada entrada de entrenamiento para guiar al modelo a extraer características que darán como resultado un valor de pérdida o error más bajo en la siguiente iteración. Ampliaremos sobre las implicancias en la sección 3, cuando hablemos de Recurrent Neural Networks.

2.3 Casos de Uso

En esta subsección final, vamos a comentar algunos casos de uso y además, ampliar el escenario y la intuición al respecto de nuestro caso de pronóstico: el de la demanda de energía.

En los últimos años, Internet de las cosas (IoT), las fuentes de energía alternativas y el big data se han fusionado para crear grandes oportunidades en el ámbito de la energía y los servicios públicos. Al mismo tiempo, la empresa de servicios públicos y todo el sector energético han visto cómo el consumo se ha estabilizado y los consumidores exigen mejores formas de controlar su uso de la energía. Por lo tanto, las empresas de servicios públicos y redes inteligentes tienen una gran necesidad de innovar y renovarse. Además, muchas redes eléctricas y de servicios públicos se están volviendo obsoletas y muy costosas de mantener y administrar.

Dentro del sector energético, podría haber muchas formas en las que la previsión de la demanda puede ayudar a resolver problemas empresariales críticos. De hecho, la previsión de la demanda puede considerarse la base de muchos casos de uso básicos en la industria. En general, consideramos dos tipos de previsiones de demanda de energía: a corto y largo plazo. Cada una puede tener un propósito diferente y utilizar un enfoque diferente. La principal diferencia entre los dos es el horizonte de pronóstico, es decir, el rango de tiempo en el futuro para el cual pronosticamos.

En el contexto de la demanda de energía, el pronóstico de carga (consumo) a corto plazo (STLF por Short Term Load Forecasting) se define como la carga agregada que se pronostica en un futuro próximo en varias partes de la red (o la red en su conjunto). En este contexto, el corto plazo se define como un horizonte de tiempo dentro del rango de 1 hora a 24 horas. En algunos casos, también es posible un horizonte de 48 horas. Por lo tanto, STLF es muy común en un caso de uso operativo de la red. Los modelos STLF se basan principalmente en los datos de consumo del pasado cercano (último día o semana) y utilizan la temperatura pronosticada como un predictor importante. Obtener pronósticos de temperatura precisos para la próxima hora y hasta 24 horas se está convirtiendo en un desafío menor en la actualidad. Estos modelos son menos sensibles a los patrones estacionales o las tendencias de consumo a largo plazo.

También es probable que las soluciones STLF generen un gran volumen de llamadas de predicción (solicitudes de servicio), ya que se invocan cada hora y, en algunos casos, incluso con mayor frecuencia. También es muy común ver la implantación, donde cada subestación o transformador energético individual se representa como un modelo independiente y, por lo tanto, el volumen de solicitudes de predicción es aún mayor.

El objetivo del pronóstico de energía a largo plazo (LTLF por Long Term Load Forecasting) es pronosticar la demanda de energía con un horizonte de tiempo que va desde una semana a varios meses (y en algunos casos durante varios años). Este rango de horizonte es principalmente aplicable para casos de uso de planificación e inversión. Para escenarios a largo plazo, es importante tener datos de alta calidad que abarquen un período de varios años (mínimo tres años). Estos modelos típicamente extraerán patrones de estacionalidad de los datos históricos y harán uso de predicadores externos como patrones meteorológicos y climáticos.

Es importante aclarar que cuanto más largo sea el horizonte de pronóstico, menos preciso puede ser el pronóstico. Por lo tanto, es importante producir algunos intervalos de confianza junto con el pronóstico real que permitiría a los humanos factorizar la posible variación en su proceso de planificación. Dado que el escenario de consumo de LTLF es principalmente de planificación, podemos esperar volúmenes de predicción mucho más bajos (en comparación con STLF). Por lo general, veríamos estas predicciones integradas en herramientas de visualización como Excel o PowerBI y el usuario las invocaría manualmente.

Cualquier solución avanzada analítica se basa en datos. Específicamente, cuando se trata de análisis predictivo y pronóstico, confiamos en un flujo de datos dinámico y continuo. En el caso de la previsión de la demanda de energía, estos datos pueden obtenerse directamente de medidores inteligentes o ya se pueden agregar a una base de datos local. También confiamos en otras fuentes externas de datos, como el clima y la temperatura. Este flujo continuo de datos debe estar orquestado, programado y almacenado.

3 Redes Neuronales Recurrentes

Como ya hemos mencionado anteriormente en la sección 1, el Deep Learning es una rama del Machine Learning que concentra los algoritmos con arquitecturas más complejos y "profundas", tales como las Redes Neuronales. Sin embargo, resulta interesante distinguir en detalle las principales diferencias entre ML y DL para ganar mayor intuición al respecto.

Diferencias Principales entre modelos de ML y DL		
Característica	**Machine Learning en general**	**Deep Learning exclusivamente**
Número de Data Points	Pueden usar pequeñas cantidades de datos para generar predicciones.	Son modelos *Data Hungry*: necesitan muchos datos para generar predicciones.
Requerimientos de Hardware	Pueden trabajar en máquinas low-end, sin necesidad de grandes capacidades de cómputo.	Dependen de máquinas high-end, ya que operan con matrices de altas dimensiones. Una GPU podría optimizar estos procesos.
Proceso de Parametrización (Featurization)	Requieren de parámetros definidos con precisión y creados por usuarios.	Los modelos aprenden parámetros de alto nivel de los datos y crean otros por sí mismos.
Aprendizaje	Dividen el aprendizaje en pasos pequeños. Luego combinan los resultados de cada período en un output final.	Avanzan con el proceso de aprendizaje resolviendo el problema de principio a fin.
Tiempo de Ejecución	Llevan comparativamente poco tiempo para ser entrenados, desde algunos segundos hasta pocas horas.	Usualmente, los modelos toman bastante tiempo en ser entrenados porque un algoritmo de Deep Learning posee muchas capas (layers).
Output	El output o salida es generalmente un valor numérico, como un puntaje o una clasificación, por ejemplo.	El output puede tener múltiples formatos, como texto, un sonido, etc.

Table 1: Diferencias principales entre algoritmos de Machine Learning y Deep Learning

En función de lo visto en la Tabla 1, se puede concluir que las redes neuronales (un algoritmo de la familia Deep Learning) pueden ser muy útiles para problemas de predicción de series de tiempo al eliminar la necesidad inmediata de realizar procesos de ingeniería de características (features), procedimientos de escalado de datos y hacer que las series sean estacionarias. En escenarios reales de series de tiempo, por ejemplo, pronóstico del clima, pronóstico del flujo de tráfico, etc. y escenarios de Forecasting basados en dispositivos IoT como geosensores, las estructuras irregulares de tiempo, los missing values, el ruido intenso e interrelaciones complejas en múltiples variables presentan limitaciones para los métodos clásicos de predicción que hemos mencionado. Por lo general, estas técnicas clásicas se basan en conjuntos de datos que estén limpios y completos para funcionar bien, eso implica que los valores faltantes, los valores atípicos y otras características imperfectas generalmente no se toleran. Hablando de conjuntos de datos más artificiales y perfectos, los métodos de pronóstico clásicos se basan en el supuesto de que existe una relación lineal y una dependencia temporal fija entre las variables de un conjunto de datos, y este supuesto por defecto excluye la posibilidad de

explorar datos más complejos, y probablemente relaciones entre variables más interesantes.

Por el contrario, las redes neuronales son resistentes al ruido en los datos de input y en la función de mapeo, e incluso pueden continuar con el aprendizaje y la predicción en presencia de valores faltantes. Por este motivo es que son modelos con arquitecturas más preparadas para enfrentar los problemas cada vez más díficiles de la vida real.

3.1 Aspectos fundamentales

Para nuestro contexto de predicción con series de tiempo, resulta ser que los modelos más adecuados son los que soportan secuencias de datos (Sequence Models), y dentro de estos, las Redes Neuronales Recurrentes.

Las redes neuronales recurrentes (RNN) o también conocidas como autoasociativas o feedback networks, se crearon en la década de 1980, pero recientemente ganaron popularidad debido al aumento de la potencia computacional de las unidades de procesamiento gráfico (GPU) y el avance del Machine Learning en general. Son especialmente útiles con datos secuenciales porque cada neurona o unidad puede usar su memoria interna para mantener información sobre la entrada o input anterior. Una RNN tiene bucles (loops) que permiten que la información se transmita a través de las neuronas mientras se lee la entrada.

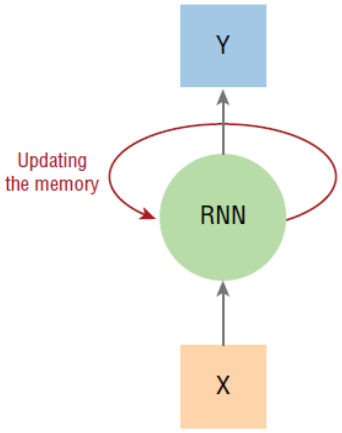


Figure 10: RNN Simple Unit

Los diferentes tipos de redes neuronales (como las feed forward networks) se basan en la idea de aprender durante el entrenamiento a partir del contexto y la historia para generar predicciones: las RNN usan la idea del "estado oculto" (*hidden state*, o bien llamado memoria) para poder generar un resultado o predicción actualizando cada neurona en una nueva unidad computacional que es capaz de recordar lo que ha visto antes.

Esta memoria se conserva dentro de la unidad, en una matriz o un vector, de modo que cuando la unidad lee un dato de entrada, también procesa el contenido existente de la memoria, combinando así toda la información. Al usar ambos (el conocimiento del input y el conocimiento de la memoria), la unidad ahora es capaz de hacer una predicción (Y_i) y actualizar la memoria en sí, como se ilustra en la Figura 10 de una RNN Simple Unit.

Las unidades RNN se describen como unidades recurrentes porque el tipo de dependencia del valor actual en el evento anterior es recurrente y puede considerarse como múltiples copias del mismo nodo, cada una de las cuales transmite un mensaje recurrente a un sucesor. Visualicemos esta relación recursiva en la Figura 11 de una arquitectura tipo.

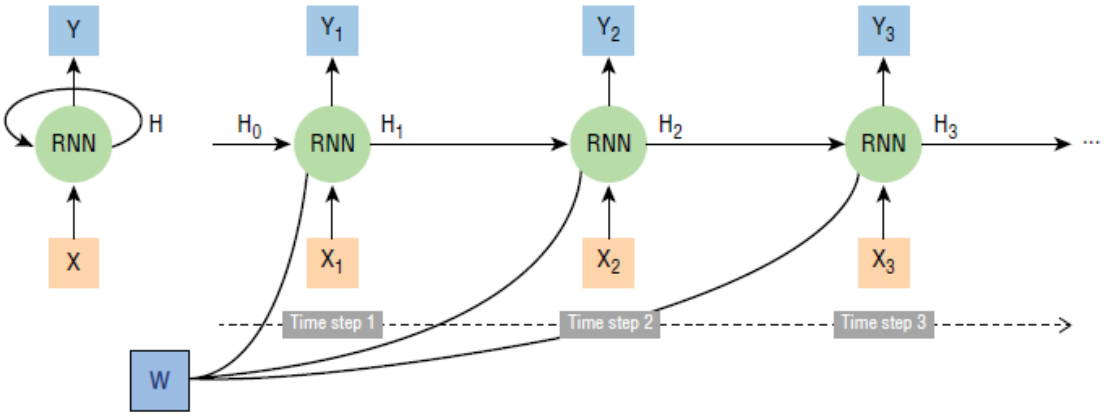


Figure 11: Arquitectura tipo de una RNN

En la Figura 11, se representa una típica arquitectura RNN. RNN tiene un estado oculto interno (*internal hidden state*), denominado H_t, que se puede retroalimentar a la red. En esta figura, el RNN procesa los valores de entrada X_i y produce los valores de salida Y_i. La estadística oculta (*hidden stat*), denotada como H_t, permite que la información se reenvíe de un nodo de la red al siguiente nodo, organizada como una secuencia o un vector. Ese vector ahora tiene información sobre la entrada actual y las entradas anteriores. El vector pasa por la función de activación tangente hiperbólica (*tanh*, podría ser otra también) y la salida es el nuevo estado oculto. La función de activación *tanh* se usa para regular los valores que fluyen a través de la red y siempre mantenerlos entre –1 y 1. Debido a esta estructura, las RNN pueden representar una buena opción de modelado para tratar de resolver una variedad de problemas secuenciales, como pronóstico de series de tiempo, reconocimiento de voz o subtítulos de imágenes, entre otras aplicaciones.

Como se puede observar, además, en la Figura 11 hay un elemento adicional denotado como W, por weights. W indica que cada unidad tiene tres conjuntos de ponderadores, uno para las entradas (X_i), otro para las salidas del ponderador del período anterior/time step (H_t), y el otro para la salida del paso de tiempo actual (Y_i). Estos valores de ponderadores o weights están determinados por el proceso de entrenamiento y pueden identificarse aplicando una técnica de optimización popular llamada "Descenso del Gradiente" (*gradient descent*).

3.1.1 Gradient Descent

En palabras simples, un gradiente es la pendiente y dirección de máximo crecimiento de una función, que se puede definir como las derivadas parciales de un conjunto de parámetros con respecto a sus inputs. Los gradientes son valores que se utilizan para actualizar los ponderadores de una red neuronal: la función representa el problema que queremos resolver utilizando redes neuronales y seleccionando una elección adecuada de parámetros para actualizar esos weights de redes neuronales.

Para calcular un descenso de gradiente[3], necesitamos calcular la función de pérdida (loss function) y su derivada con respecto a los weigths. Comenzamos con un punto aleatorio en la función y nos movemos en la dirección negativa del gradiente (ya que buscamos minimizar la función de pérdida) de la función para llegar a un punto donde el valor de la función es mínimo.

En la Figura 11, parece que estamos aplicando las mismas ponderaciones a diferentes elementos de la serie de entrada. Esto significa que estamos compartiendo parámetros entre entradas. Si no podemos compartir parámetros entre las entradas, entonces una RNN se convierte en una red neuronal clásica donde cada nodo de entrada requiere un ponderador propio y único. En cambio, las RNN pueden aprovechar su propiedad de estado oculto (hidden state) que vincula una entrada a la siguiente y combina esta conexión de entrada en una entrada en serie.

A pesar de su gran potencial, las RNN tienen algunas limitaciones, ya que sufren de memoria a corto plazo (short-term memory). Por ejemplo, si una secuencia de entrada es lo suficientemente larga, no pueden transferir información de los períodos de tiempo inicial a los finales y, a menudo, omiten información importante desde el principio de la secuencia durante el proceso de retropropagación (*backpropagation*). Nos referimos a este problema como el problema de *vanishing gradients* (desvanecimiento de gradientes), como se ilustra en la Figura 12.

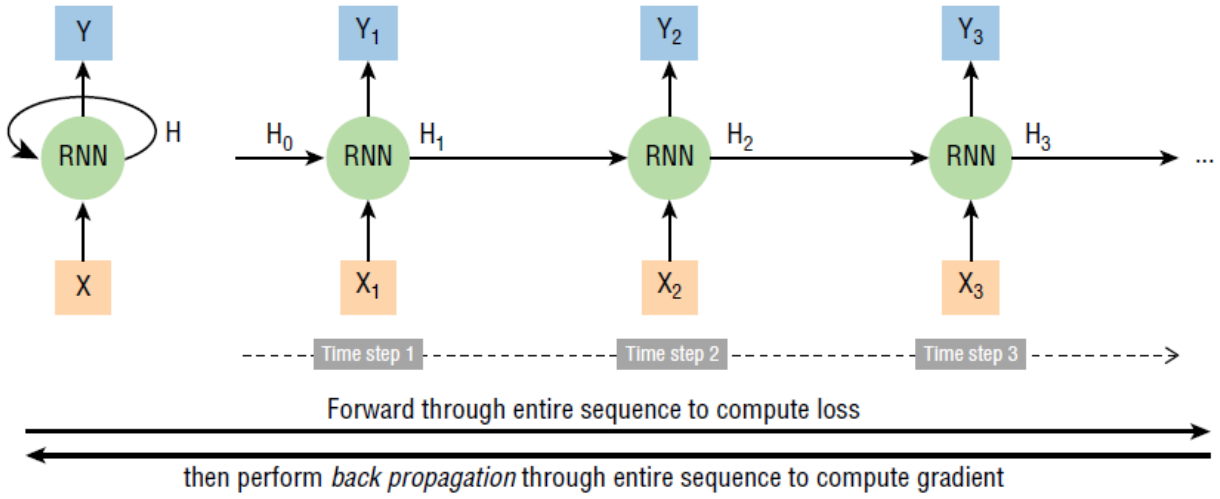


Figure 12: Backpropagation en una RNN

Durante la propagación hacia atrás (backpropagation), las redes neuronales recurrentes sufren este problema porque el gradiente disminuye (se desvanece) a medida que se propaga hacia atrás a lo largo del tiempo. Si un valor de gradiente se vuelve extremadamente pequeño, no puede contribuir al proceso de aprendizaje de la red. En otras palabras, una red recurrente simple posee el gran problema de no poder capturar dependencias a largo plazo. Esta es una de las principales razones por las que las RNN desaparecieron de la práctica hasta que se lograron excelentes resultados con el uso de una unidad LSTM (Long Short Term

[3]En el apéndice D se puede leer el desarrollo algebráico básico del Gradient Descent.

Memory) dentro de la red neuronal. La estructura LSTM resuelve este problema al introducir cambios en la arquitectura RNN sobre cómo calculan las salidas y el estado oculto utilizando las entradas: específicamente, las LSTM introducen elementos adicionales en la arquitectura, llamados "puertas" (*gates*) y "estado de celda" (*cell state*). Hay variantes de LSTM, y las discutiremos con más detalle en la siguiente sección.

3.2 Long Short-Term Memories

Las redes neuronales con LSTM son un tipo particular de RNN que tienen algunas células de estado contextual interno que actúan como células de memoria a largo o corto plazo. La salida de la red LSTM está modulada por el estado de estas células. Esta es una propiedad muy importante cuando necesitamos que la predicción de la red neuronal dependa del contexto histórico completo de todos los inputs, en lugar de solo el último input.

La celda de la LSTM toma decisiones sobre qué, cuánto y cuando almacenar y liberar información: aprenden cuando permitir que la información ingrese, salga o se elimine a través del proceso iterativo de hacer hipótesis, retropropagar el error y ajustar los weights a través del gradient descent. La Figura 13 ilustra cómo la información fluye a través de una celda y es controlada por diferentes puertas (gates).

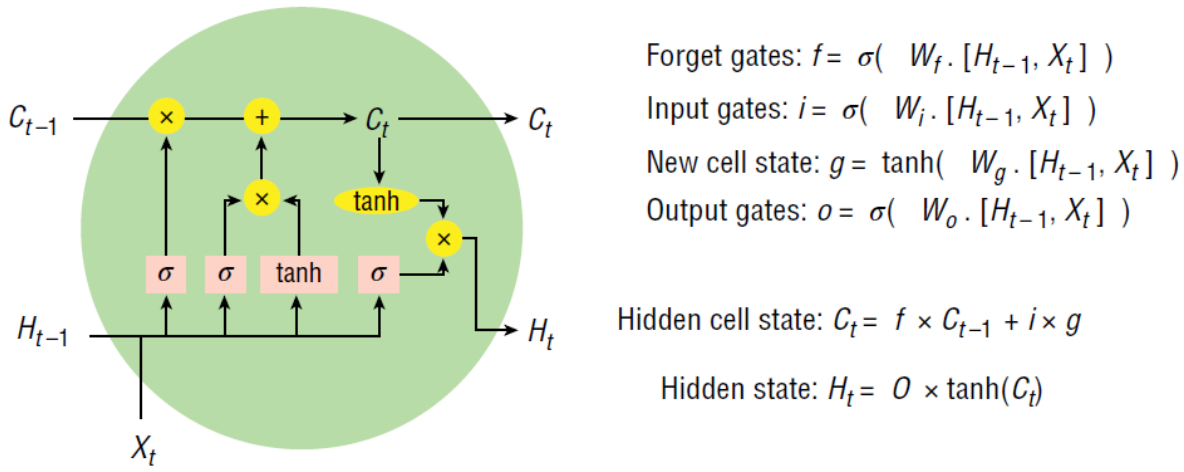


Figure 13: Backpropagation en RNN dentro de LSTM

En la Figura 13, es importante notar que las celdas de memoria LSTM aprovechan tanto la suma (*addition*, en la figura se aprecia el simbolo +) como la multiplicación (*multiplication*, en la figura $\times$) en la transformación del input y la transferencia de información. El paso de suma es esencialmente el secreto de las LSTM: conserva un error constante cuando debe propagarse hacia atrás en profundidad. En lugar de influir en el siguiente estado de la celda multiplicando su estado actual con una nueva entrada/input, suman los dos, mientras que la puerta de olvido (*forget gate*) todavía utiliza la multiplicación.

Para actualizar el estado de la celda (cell state), tenemos la "puerta de entrada" (*input gate*). La información del estado oculto anterior (H_{t-1}) y de la entrada actual (H_t) se pasa a través de la función sigmoidea (σ), para obtener valores entre 0 y 1: cuanto más cerca esté de 0 significa "olvidar", y cuanto más cerca de 1 significa "recordar". Como ya mencionamos,

las LSTM también tienen una puerta de olvido (forget gate), incluso si su especialidad es en realidad mantener y transferir información de sucesos distantes a una salida final. La puerta del olvido decide qué información debe olvidarse y qué información debe recordarse y transferirse.

Finalmente, tenemos la "puerta de salida" (*output gate*), que decide cuál debería ser el siguiente estado oculto (H_{t+1}). Primero, el estado oculto anterior (H_{t-1}) y de la entrada actual (H_t) se pasan a una función sigmoidea. Luego, el estado de celda (cell state, C_t) recién modificado se pasa a través de la función *tanh*. La salida de *tanh* se multiplica por la salida sigmoidea para decidir qué información debe conservar el estado oculto.

El ejemplo del procesamiento de video puede ser muy efectivo cuando necesitamos entender cómo funcionan las redes LSTM: en una película, lo que sucede en el fotograma actual depende en gran medida de lo que estaba en el último fotograma. Durante un período de tiempo, una red LSTM intenta aprender qué guardar y cuánto guardar del pasado y cuánta información guardar del estado actual, lo que la hace poderosa en comparación con otros tipos de redes neuronales.

Esta capacidad se puede utilizar en cualquier contexto de pronóstico de series de tiempo, donde puede ser extremadamente útil aprender automáticamente la dependencia temporal de los datos. En el caso más simple, a la red se le muestra una observación a la vez de una secuencia y puede aprender qué observaciones previas son importantes y cómo son relevantes para el pronóstico. El modelo aprende un mapeo de entradas a salidas y aprende qué contexto de la secuencia de entrada es útil para el mapeo y puede cambiar dinámicamente este contexto según sea necesario.

No es sorprendente que este enfoque se utilice a menudo en la industria financiera para construir modelos que pronostiquen los tipos de cambio, tasas de interés, precios, etc. basados en la idea de que el comportamiento pasado y los patrones pueden afectar los movimientos actuales, siendo útiles para predecir comportamiento y patrones a futuro. Por otro lado, existen desventajas con las que los científicos de datos deben tener cuidado al implementar aquitecturas de redes neuronales: Se requieren de grandes volúmenes de datos (son *data hungry*) y los modelos requieren de muchos ajustes de hiperparámetros y múltiples ciclos de optimización.

En la siguiente subsección, vamos a discutir un tipo específico de red LSTM, llamadas unidades recurrentes cerradas (*Gated Recurrent Units*): su objetivo es resolver el problema del gradiente que desvanece (vanishing gradient) que viene con una red neuronal recurrente estándar. La unidad recurrente cerrada, presentada por KyungHyun Cho y sus colegas en 2014 (Cho et al. 2014), es una variante ligeramente más optimizada de las redes LSTM que a menudo ofrece un rendimiento comparable y es significativamente más rápida de calcular.

3.3 Gated Recurrent Unit

Las GRU también se puede considerar como una variación de las LSTM porque ambas están diseñadas de manera similar y, en algunos casos, producen resultados igualmente excelentes. Las redes GRU no aprovechan el estado de la celda y utilizan el estado oculto para transferir información. A diferencia de LSTM, las redes GRU contienen solo tres puertas y no mantienen un estado de celda interno. La información que se almacena en el estado interno

de la celda en una unidad recurrente LSTM se incorpora al estado oculto del GRU. Esta información combinada se agrega y se transfiere al siguiente GRU.

Las dos primeras puertas, la "puerta de reinicio" (*reset gate*) y la "puerta de actualización" (*update gate*), ayudan a resolver el problema del gradiente que desvanece en una RNN estándar: estas puertas son dos vectores que deciden qué información se debe pasar a la salida. La característica única de ellos es que pueden ser entrenados para guardar información de hace mucho tiempo inclusive o eliminar información que es irrelevante para la predicción.

En mayor detalle, las tres puertas de una GRU son[4]:

- Update gate: esta puerta ayuda al modelo a determinar cuánto de la información pasada (de períodos/time steps previos) necesita ser pasada hacia el futuro. Esto es poderoso, ya que el modelo puede decidir copiar toda la información del pasado y eliminar por completo el riesgo del vanishing gradient.
- Reset gate: esta es otra puerta usada para determinar cuánta información del pasado olvidar.
- Current memory gate (puerta de memoria actual): Esta tercera puerta se incorpora a la puerta de reinicio (reset gate) y es usada para iniciar una no linearidad en el dato de entrada (input) y desarrollar la media cero de dicho input. Otra razón para incluirla en la puerta de reinicio es para disminuir el impacto de la información pasada en la información actual que se transmite hacia el futuro.

En conclusión, una GRU es un mecanismo muy útil para solucionar el problema del gradiente que se desvanece en RNN. Este problema ocurre en Machine Learning, en varios modelos y tipos de algoritmos.

[4]Ver apéndice G para una explicación en mayor detalle de las GRU.

4 Aplicación Práctica Nº1: Energy Consumption Forecast con RNN

Hasta el momento, se ha desarrollado en mayor o menor detalle todos los conceptos teóricos y conceptuales para ganar intuición y ser capaz de comprender la teoría subyacente de un modelo real implementado en Python. En esta sección, se extenderá la teoría a una aplicación práctica con Redes Neuronales Recurrentes. Como bien se explicó en la subsección 2.3 (Casos de Uso) sobre los aspectos de un pronóstico de consumo de energía, ahora ampliaremos en los detalles técnicos para poder llevarlo acabo. Es recomendable tener bien presente lo explicado en dicha subsección anterior, ya que la intuición sobre por qué es útil predecir la demanda de energía puede facilitar el entendimiento de ciertos aspectos de la base de datos, etc.

Para esta aplicación específica de Deep Learning, utilizaremos el set de datos públicos "Load Forecasting Data" de la competencia GEFCom2014. La consigna principal de dicha competencia de pronóstico de energía de GEFCom2014 se centró en el pronóstico probabilístico de esta. Los datos completos se publicaron como apéndice del paper GEFCom2014: (Hong et al.2016). Este set de datos consta de tres años de valores de carga eléctrica por hora y valores de temperatura entre 2012 y 2014.

El motivo por el cual se utiliza esta base de datos es por el volumen de esta. Básicamente consta de un total de 23370 puntos o datos, lo que hace posible poder utilizar un modelo de Deep Learning, especificamente de RNN. Recordemos (de la Tabla 1) que estos son Data Hungry, lo que implica que se debe disponer de un set de datos relativamente grande, ya que en caso contrario se debería terminar adoptando una técnica de Forecasting clásica (como se explicó en la subsección 2.2) que no dependen de un volumen grande de datos. Cabe destacar que para la implementación de este trabajo se intentó, originalmente, utilizar un dataset con datos de consumo eléctrico en Argentina (suministrados por CAMMESA, la Compañía Administradora del Mercado Mayorista Eléctrico), entre 2012 y 2020, pero debido a la escasa disponibilidad de mediciones, ya que se contaba solamente con datos mensuales y esto no era para nada suficiente (sacando cuentas podemos ver fácilmente que a mediciones mensuales, desde 2012 a 2020, obtenemos tan solo 96 samples).

Por esta razón es que se abordará una segunda aplicación práctica en la sección 5, que considerará una técnica clásica de forecasting, en Python, con el dataset (pequeño) de la vida real.

Por último, resulta adecuado también realizar una breve reseña sobre nuestra herramienta de implementación que son Python y las correspondientes librerías que facilitan el trabajo.

Python es actualmente una de las plataformas más dominantes para los datos de series de tiempo debido al excelente soporte de librerias o paquetes. Hay algunas librerias de Python para series de tiempo. Como se puede ver en la Figura 14, SciPy es un ecosistema basado en Python de software de código abierto para matemática, ciencias e ingenieria. Incluye módulos para estadística, optimización, integración, álgebra lineal, transformadas de Fourier,

procesamiento de señales e imágenes, y más.[5]

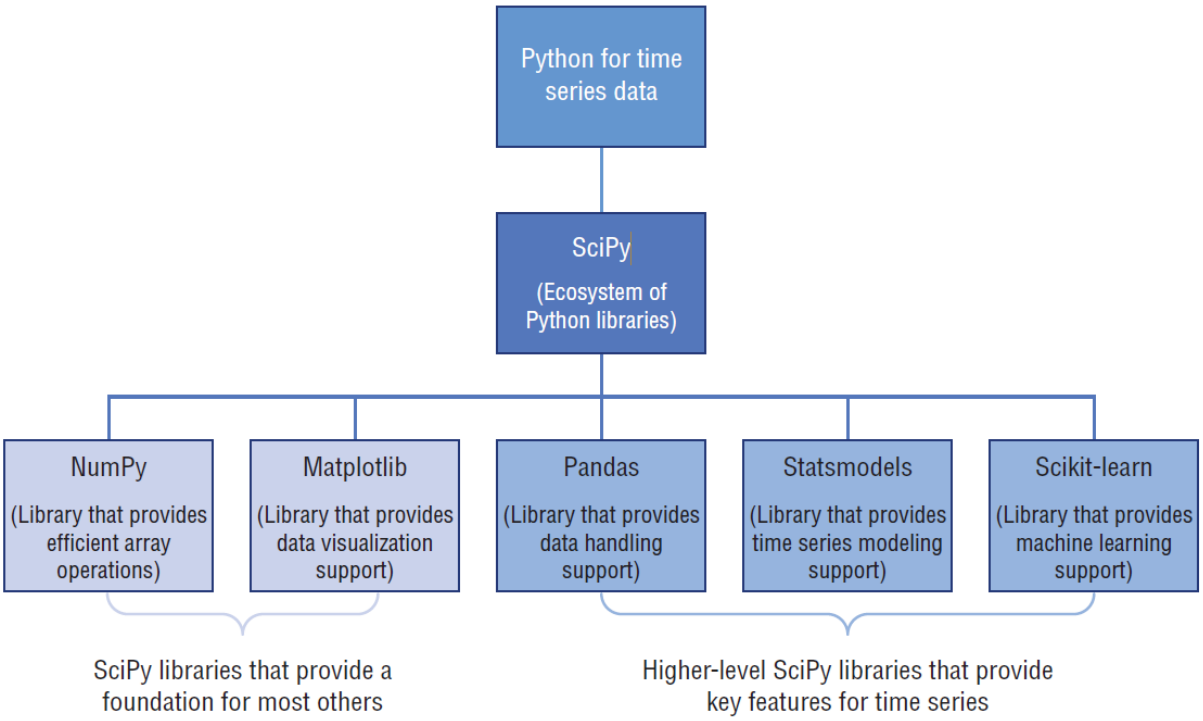


Figure 14: Ecosistema de SciPy en Python

- SciPy está diseñado para funcionar con matrices/arrays de NumPy y proporciona muchas rutinas numéricas eficientes y fáciles de usar, como rutinas para integración y optimización numéricas.
- NumPy es el paquete fundamental para computacion científica con Python. Contiene, entre muchas otras cosas, objetos tipo array de N-dimensiones, funciones de broadcasting, funciones de algebra lineal, transformada de Fourier y hasta números al azar.
- Matplotlib es una libreria de visualización de datos construida sobre matrices NumPy. Es un paquete completo para graficar datos y permite a los científicos de datos crear visualizaciones dinámicas e interactivas. Además, permite conectarse y consumir grandes cantidades de datos para crear visualizaciones ágiles.
- Pandas es un paquete muy útil y popular para importar y analizar datos. La estructura de datos principal de pandas es el DataFrame, que es una tabla de datos con filas y columnas con nombre. Pandas ofrece funcionalidades específicas de series de tiempo, como generación de rango de fechas y conversión de frecuencia, estadísticas, cambio de fecha y rezagos.
- Statsmodels es un módulo de Python que admite una amplia gama de funciones y clases estadísticas usando (pandas) DataFrames. También es utilizado por científicos de datos para realizar pruebas estadísticas y exploración de datos estadísticos.
- Scikit-learn es una libreria escrita en gran parte en Python y construida sobre NumPy,

[5]Más información se puede encontrar en scipy.org

SciPy y Matplotlib. Este paquete ofrece muchas funcionalidades útiles para Machine Learning y el modelado estadístico, incluido vector machines, random forests, k-neighbors, clasificación, regresión y agrupación en clusters.

Asimismo, tanto los modelos LSTM como GRU se pueden aplicar a la predicción de series de tiempo. Para estas implementaciones, también hay paquetes y frameworks específicos que permiten trabajar Deep Learning en Python. Nosotros utilizaremos Keras y Tensorflow.

- Keras[6] es una libreria de Python capaz de ejecutarse sobre diferentes herramientas de Deep Learning, como TensorFlow. Lo más importante es que Keras es una API que admite redes convolucionales (CNN) y redes recurrentes (RNN) y se ejecuta sin problemas en unidades de procesamiento central (CPU) y unidades de procesamiento de gráficos (GPU). Además, Keras se desarrolló teniendo en cuenta cuatro pilares:

  - Facilidad de uso y minimalismo: Keras es una API diseñada pensando en la experiencia del usuario y ofrece API consistentes y simples para construir y entrenar modelos de Deep Learning.

  - Modularidad: al usar Keras, los usuarios deben considerar el ciclo de desarrollo del modelo como un proceso modular, en el que los componentes independientes (como capas/layers neuronales, funciones de costos, optimizadores, funciones de activación, etc.) se pueden aprovechar y combinar fácilmente de diferentes maneras para crear nuevos modelos.

  - Extensibilidad: relacionado con el punto anterior, el tercer principio de Keras consiste en brindar a los usuarios la capacidad de ampliar y agregar nuevos módulos a los existentes, con el fin de mejorar el ciclo de desarrollo del modelo.

  - Trabajar con Python: como se mencionó anteriormente, Keras es una biblioteca contenedora de Python y sus modelos están definidos en código Python (keras.io).

  Básicamente, el proceso de desarrollar un modelo de Deep Learning en Keras puede dividirse en los siguientes pasos:

  1. Definir el modelo (secuencial en nuestro caso) y agregar capas/layers ya configuradas.
  2. Compilar el modelo. Especificar la loss function, los algoritmos de optimizacion y llamar la función compile() en el modelo.
  3. Ajustar (fit) el modelo. Entrenarlo en el conjunto de datos específico al llamar la función fit() en el modelo.
  4. Hacer las predicciones. Usar el modelo para generar los pronósticos en datos nuevos al llamar las funciones evaluate() y predict().

- TensorFlow[7] es una libreria de código abierto para el cálculo numérico mediante gráficos de flujo de datos (data flow graphs). Fue creado y mantenido por el equipo de Google Brain, y se publica bajo la licencia de código abierto Apache 2.0. Una de las capacidades más beneficiosas de TensorFlow es que los científicos de datos pueden crear y entrenar sus modelos con TensorFlow utilizando la API de Keras, lo que facilita la introducción a

[6] Más información en keras.io
[7] Más información en tensorflow.org

TensorFlow y el aprendizaje automático. Los nodos en el gráfico representan operaciones matemáticas, mientras que los bordes del gráfico representan las matrices de datos multidimensionales (tensores) que se comunican entre ellos. TensorFlow se creó para su uso en sistemas de investigación, desarrollo y producción debido a su capacidad de ejecutarse en sistemas de CPU única, GPU, dispositivos móviles y sistemas distribuidos a gran escala.

4.1 Procesamiento y Preparación de los Datos

En esta subsección profundizaremos sobre cómo preparar los datos de series de tiempo para una posterior implementación con arquitectura de LSTM y GRU de RNN.

Los datos brutos (raw data) se componen de filas y columnas. Cada medición se representa como una única fila de datos. Cada fila de datos incluye varias columnas (también denominadas características o parámetros):

- Marca de tiempo (*time stamp*): el campo de time stamp representa la hora real en la que se registró la medición. Debe cumplir con uno de los formatos habituales de fecha/hora. En la mayoría de los casos, no es necesario registrar el tiempo hasta el segundo nivel de granularidad. También es importante especificar la zona horaria en la que se registran los datos.
- Valor de energía (*Load value*): este es el consumo energético real en una fecha/hora determinada. El consumo se puede medir en kWh (kilovatio-hora) o en cualquier otra unidad preferida. Es importante tener en cuenta que la unidad de medida debe ser coherente con todas las medidas de los datos. En algunos casos, el consumo se puede suministrar en tres fases de potencia. En ese caso, necesitaríamos recopilar todas las fases de consumo independientes.
- *Temperatura*: la temperatura generalmente se obtiene de una fuente independiente. Sin embargo, debería ser compatible con los datos de consumo. Debe incluir una marca de tiempo como se describe anteriormente que permitirá sincronizarlo con los datos de consumo reales, no puede haber asimetría en esta. El valor de la temperatura se puede especificar en grados Celsius o Fahrenheit, pero debe mantenerse constante en todas las mediciones.

Como se explicó en subsección 2.1 sobre el preparamiento de los datos, debemos primero cargar los datos en un formato adecuado para visualizarlo, indexarlo, etc. En este caso, como nuestra fuente es un archivo en formato xlsx (Excel), lo debemos cargar como un Pandas Dataframe, así luego podemos realizar las siguientes tareas:

1. Indexar los datos en la columna time stamp para poder filtrarlos por tiempo.
2. Visualizar los datos en tablas con las columnas identificadas.
3. Identificar valores faltantes (missing values) y períodos de tiempo que puedan faltar.
4. Crear variables adicionales, rezagadas y adelantadas si hace falta.

Además, nuestros datos deben ser transformados en dos tensores X (input) e Y (output). Las dimensiones serán el número de datos, los períodos de tiempo elegidos y los parámetros. Estos representan lo que se requiere para hacer una predicción (X), mientras que la muestra

y el horizonte representan la predicción que se hace (Y). Si tomamos el ejemplo de un problema de serie de tiempo univariante (por ejemplo, si queremos predecir la siguiente cantidad de carga de energía) donde estamos interesados en predicciones de un paso o one-step (por ejemplo, la próxima hora), las observaciones en los pasos de tiempo anteriores (para ejemplo, cuatro valores de carga de energía durante las cuatro horas anteriores), las llamadas observaciones de retraso, se utilizan como entrada y la salida es la observación en el siguiente paso de tiempo (horizonte).

Primero, importamos todos los paquetes necesarios:

[1]:
```
# Importamos packages necesarios
import pandas as pd
import datetime as dt
import numpy as np
import os
import matplotlib.pyplot as plt
%matplotlib inline
```

Y realizamos ajustes de formato:

[2]:
```
# Ajustamos el formato del dataset
pd.options.display.float_format = '{:,.2f}'.format
np.set_printoptions(precision=2)
```

Ahora, como indicamos antes, cargaremos el dataset desde el archivo Excel y ajustaremos las primeras variables de tiempo:

[ ]:
```
"""Solo correr por primera vez"""
# Cargar datos desde el archivo Excel
data = pd.read_excel('GEFCom2014-E.xlsx',␣
↪engine='openpyxl', parse_dates=True)

# Crear variable Timestamp de Date y Hour
data['timestamp'] = data['Date'].add(pd.to_timedelta(data.
↪Hour - 1, unit='h'))
data = data[['timestamp', 'load', 'T']]
data = data.rename(columns={'T':'temp'})

# Eliminar time period sin load data (sin datos de␣
↪consumo)
data = data[data.timestamp >= '2012-01-01']

# Guardar a CSV
data.to_csv('energy.csv', index=False)
```

Nótese que hemos guardado en un archivo csv (comma separated values) los datos tal como necesitamos. Esto nos permitirá acceder a una base ya filtrada y de forma más directa.

Forzaremos el formato de las fechas como timestamp (de Pandas) con el argumento "parse dates" de la función read csv de Pandas.

[3]:
```
energy = pd.read_csv('energy.csv', parse_dates=['timestamp'])

# Reindexación del dataframe para que tenga un registro para cada punto del tiempo
# entre el minimo y máximo período de la serie. Esto ayuda a
# identificar los períodos faltantes (missing) en los datos.

energy.index = energy['timestamp']
energy = energy.reindex(pd.date_range(min(energy['timestamp']),
max(energy['timestamp']),
freq='H'))
energy = energy.drop('timestamp', axis=1)
```

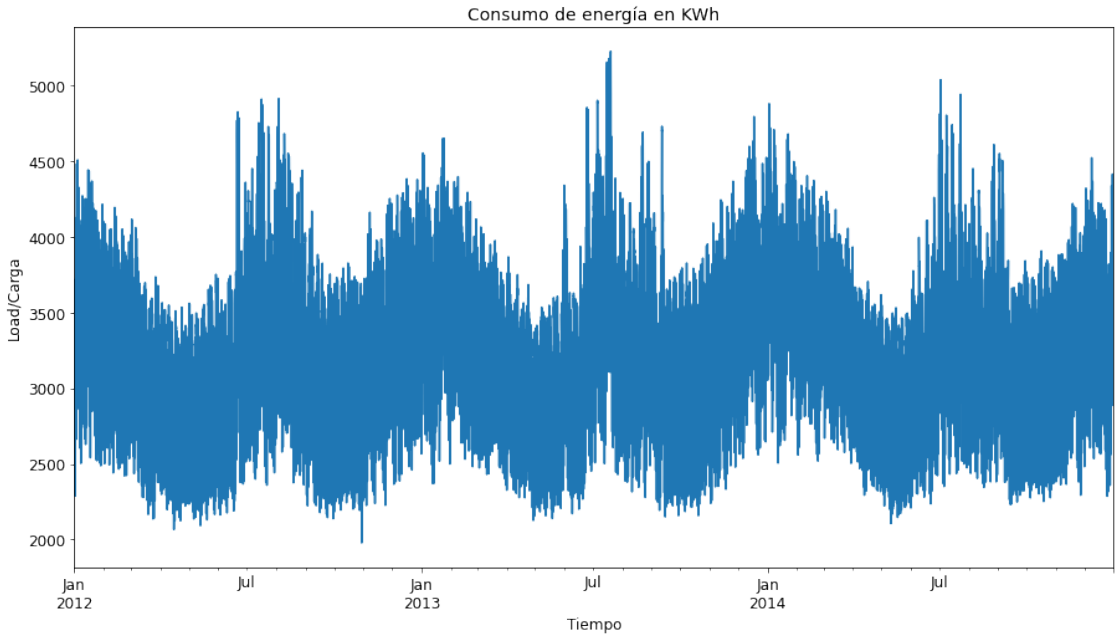
Podemos ver cómo quedó nuestro dataset:

[4]:
```
energy.head()
```

[4]:
```
                     load  temp
2012-01-01 00:00:00 2,698.00 32.00
2012-01-01 01:00:00 2,558.00 32.67
2012-01-01 02:00:00 2,444.00 30.00
2012-01-01 03:00:00 2,402.00 31.00
2012-01-01 04:00:00 2,403.00 32.00
```

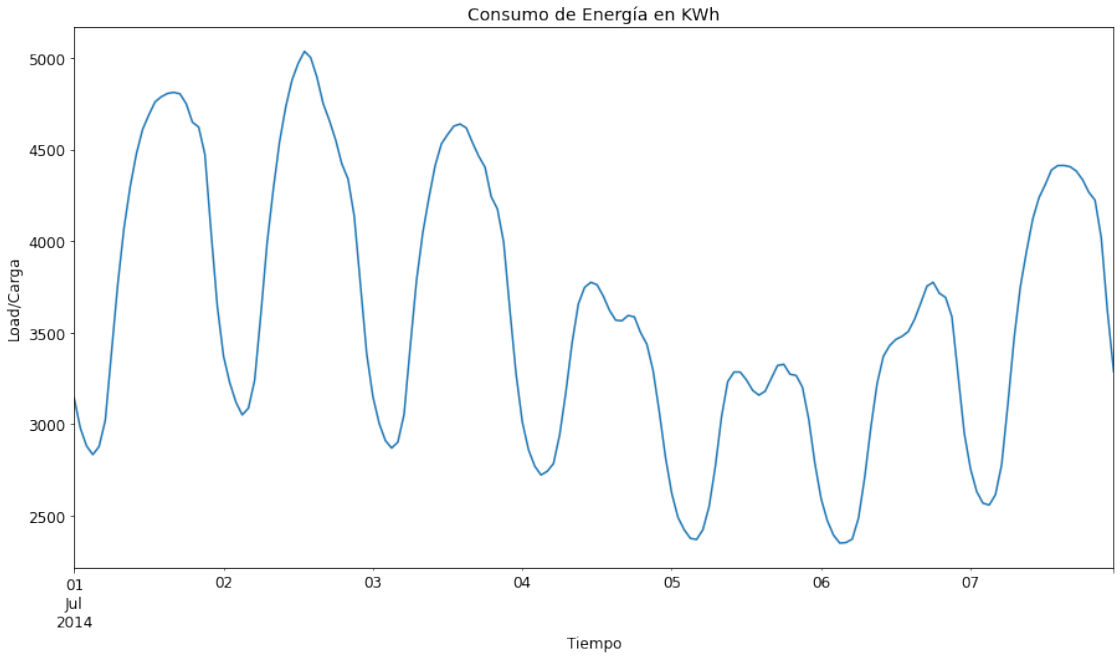
Hacemos un gráfico preliminar de los datos. Es importante identificar, en este paso, características de la serie de tiempo.

[6]:
```
# Gráfico de todos los datos disponibles de energía (entre Enero 2012 a Dec 2014) - Es lo que usaremos
energy.plot(y='load', subplots=True, figsize=(15, 8), fontsize=12, legend=None)
plt.xlabel('Tiempo', fontsize=12)
plt.ylabel('Load/Carga', fontsize=12)
plt.title('Consumo de energía en KWh', fontsize=14)
plt.show()
```



Podemos también hacer un gráfico más detallado, es decir, acotando la cantidad de tiempo. Probemos con 7 días:

[7]:
```
# Gráfico de primer semana de 2014 como ejemplo
energy['2014-07-01':'2014-07-07'].plot(y='load',␣
↪subplots=True, figsize=(15, 8), fontsize=12, legend=None)
plt.xlabel('Tiempo', fontsize=12)
plt.ylabel('Load/Carga', fontsize=12)
plt.title('Consumo de Energía en KWh', fontsize=14)
plt.show()
```



Algunas características a notar del índice de nuestro dataset:

[8]:
```
# El índice es DateTime
energy.index
```

[8]:
```
DatetimeIndex(['2012-01-01 00:00:00', '2012-01-01 01:00:
↪00',
               '2012-01-01 02:00:00', '2012-01-01 03:00:00',
               '2012-01-01 04:00:00', '2012-01-01 05:00:00',
               '2012-01-01 06:00:00', '2012-01-01 07:00:00',
               '2012-01-01 08:00:00', '2012-01-01 09:00:00',
               ...
               '2014-12-31 14:00:00', '2014-12-31 15:00:00',
               '2014-12-31 16:00:00', '2014-12-31 17:00:00',
               '2014-12-31 18:00:00', '2014-12-31 19:00:00',
               '2014-12-31 20:00:00', '2014-12-31 21:00:00',
               '2014-12-31 22:00:00', '2014-12-31 23:00:00'],
              dtype='datetime64[ns]', length=26304, freq='H')
```

Ya con nuestros datos preparados, podemos pasar al siguiente proceso, que corresponde al modelado.

4.2 Construcción del modelo

Consideremos ahora el paso para construir nuestro modelo Univariable. Será simple en términos prácticos, ya que el único input que usa es el mismo tipo de dato de output: la

carga energética (load). Esto nos permite construir conceptualmente un modelo básico e intuitivo, pero a la vez eficaz. Se podría implementar un modelo multivariable que considere la temperatura como predictor, pero la arquitectura de la red neuronal difiere para los fines que queremos. Dentro de la fase de construcción del modelo y selección, primeramente, necesitamos separar nuestro set de datos entre conjuntos para entrenamiento, validación y testeo. De esa manera, ya podremos:

1. Entrenar el modelo con el train set.
2. Con el conjunto de validación, evaluar el modelo luego de cada epoch de entrenamiento y asegurarnos que no está sobreajustando los datos de entrenamiento.
3. Al finalizar el entrenamiento, evaluar el modelo con el test set.

Nuevamente, antes que nada, importamos los paquetes necesarios para avanzar propiamente con el algoritmo de Deep Learning.

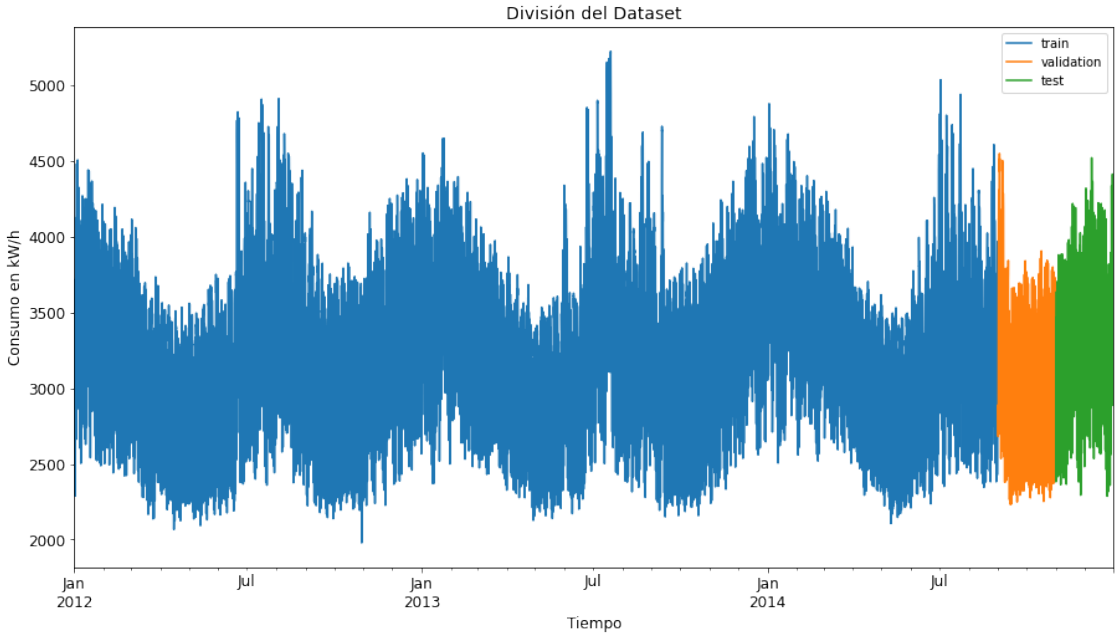
[11]:
```
import warnings
from collections import UserDict
from sklearn.preprocessing import MinMaxScaler
from IPython.display import Image

warnings.filterwarnings("ignore")
```

Crearemos ahora los sets de entrenamiento, validación y testeo:

[12]:
```
valid_start_dt = '2014-09-01 00:00:00'
test_start_dt = '2014-11-01 00:00:00'
```

[13]:
```
# plt.style.use('seaborn')
energy[energy.index < valid_start_dt][['load']].
↪rename(columns={'load':'train'}) \
.join(energy[(energy.index >=valid_start_dt) & (energy.
↪index < test_start_dt)][['load']] \
.rename(columns={'load':'validation'}), how='outer') \
.join(energy[test_start_dt:][['load']].
↪rename(columns={'load':'test'}), how='outer') \
.plot(y=['train', 'validation', 'test'], figsize=(15, 8),␣
↪fontsize=12)
plt.xlabel('Tiempo', fontsize=12)
plt.ylabel('Consumo en kW/h', fontsize=12)
plt.title('División del Dataset',fontsize=14)
plt.show()
```



Como se puede apreciar en el gráfico, identificados por colores, nuestra regla para separar los datos en train, validation, test sets fue de 86/12/12 % aproximadamente (como comentamos en la sección 2).

Por otro lado, como se muestra en el código a continuación, estamos configurando T (el número de variables de retardo) en 6. Esto significa que la entrada para cada muestra es un vector de las 6 horas previas de los valores de energía. La elección de T = 6 fue arbitraria; puede depender de recomendaciones de expertos en negocios y también al experimentar con diferentes opciones. También estamos estableciendo el horizonte en 1, ya que estamos interesados en predecir la próxima hora $(t + 1)$ de nuestro conjunto de datos:

[14]:
```
T = 6          # Hiperparámetro tamaño de la secuencia del input
HORIZON = 1    # Hiperparámetro horizonte (t+1)
```

Ahora, debemos preparar un poco más los datos para el conjunto de entrenamiento, que implicará los siguientes pasos: 1. Filtrar el conjunto de datos original para incluir solo el período de tiempo reservado para el entrenamiento. 2. Escalar la serie de tiempo de manera que los valores estén dentro del intervalo (0, 1) 3. Cambiar los valores de la serie temporal para crear un Pandas Dataframe que contenga todos los datos para un solo ejemplo de entrenamiento. 4. Desechar las muestras con valores faltantes. 5. Transformar este Dataframe en un array de numpy de dimensión (muestras, timestamps, características) para introducir en Keras

[15]:
```
                # 1. Traemos los datos para el entrenamiento del rango␣
↪de fechas valido
```

```
                train = energy.copy()[energy.index <␣
↪valid_start_dt][['load']]

                # 2. Escalamos los datos para valores entre (0, 1).
                #   Esta calibración se hace solamente en el training␣
↪set.
                #   Es para prevenir se filtre información del␣
↪validation o test sets adentro del entrenamiento!

                scaler = MinMaxScaler()
                train['load'] = scaler.fit_transform(train)

                # 3. Cambiamos el dataframe para crear los samples de␣
↪entrada (input).
                train_shifted = train.copy()
                train_shifted['y_t+1'] = train_shifted['load'].shift(-1,␣
↪freq='H')
                for t in range(1, T+1):
                train_shifted[str(T-t)] = train_shifted['load'].
↪shift(T-t, freq='H')
                y_col = 'y_t+1'
                X_cols = ['load_t-5',
                'load_t-4',
                'load_t-3',
                'load_t-2',
                'load_t-1',
                'load_t']
                train_shifted.columns = ['load_original']+[y_col]+X_cols

                # 4. Eliminamos (dropeamos) cualquier sample con missing␣
↪value
                train_shifted = train_shifted.dropna(how='any')
                train_shifted.head(5)
```

[15]:

```
                load_original  y_t+1  load_t-5  load_t-4  load_t-3  \
                2012-01-01 05:00:00           0.15   0.18      0.22     ␣
↪0.18      0.14
                2012-01-01 06:00:00           0.18   0.23      0.18     ␣
↪0.14      0.13
                2012-01-01 07:00:00           0.23   0.29      0.14     ␣
↪0.13      0.13
                2012-01-01 08:00:00           0.29   0.35      0.13     ␣
↪0.13      0.15
                2012-01-01 09:00:00           0.35   0.37      0.13     ␣
↪0.15      0.18
```

```
                     load_t-2  load_t-1  load_t
2012-01-01 05:00:00      0.13      0.13    0.15
2012-01-01 06:00:00      0.13      0.15    0.18
2012-01-01 07:00:00      0.15      0.18    0.23
2012-01-01 08:00:00      0.18      0.23    0.29
2012-01-01 09:00:00      0.23      0.29    0.35
```

Ahora, convertimos el target e input features en arrays de numpy. X tiene que estar en el shape (samples, time steps, features). Acá tenemos 23370 samples, 6 time steps (de T) y 1 feature solo (load).

[16]:
```
# 5.Transformamos el Dataframe en un vector (array) de Numpy
y_train = train_shifted[y_col].to_numpy()
X_train = train_shifted[X_cols].to_numpy()
```

[17]:
```
# Tenemos que cambiar la dimension que se compone de (23370, 6, 1)
X_train = X_train.reshape(X_train.shape[0], T, 1)
```

Verificando las shapes de los arrays vemos que:

Ahora tenemos un vector para la variable explicada del shape:

[19]:
```
y_train.shape
```

[19]:
```
(23370,)
```

La variable explicada se ve, para los primeros 3 samples, como:

[20]:
```
y_train[:3]
```

[20]:
```
array([0.18, 0.23, 0.29])
```

El tensor para el input features tiene la dimension:

[21]:
```
X_train.shape
```

[21]:
```
(23370, 6, 1)
```

[22]:
```
train_shifted.head(3)
```

[22]:
```
                     load_original  y_t+1  load_t-5  load_t-4  load_t-3  \
2012-01-01 05:00:00           0.15   0.18      0.22      0.18      0.14
2012-01-01 06:00:00           0.18   0.23      0.18      0.14      0.13
```

```
                2012-01-01 07:00:00           0.23    0.29      0.14
↪0.13      0.13

                load_t-2  load_t-1  load_t
                2012-01-01 05:00:00      0.13      0.13    0.15
                2012-01-01 06:00:00      0.13      0.15    0.18
                2012-01-01 07:00:00      0.15      0.18    0.23
```

Ahora, podemos realizar un proceso similar al del set de entrenamiento, pero al de validación:

[23]:
```
                # 1. Tomamos los datos de validación en el rango de
↪tiempo correcto
                look_back_dt = dt.datetime.strptime(valid_start_dt,
↪'%Y-%m-%d %H:%M:%S') - dt.timedelta(hours=T-1)
                valid = energy.copy()[(energy.index >=look_back_dt) &
↪(energy.index < test_start_dt)][['load']]

                # 2. Escalas la serie usando el trasnformador ajustado
↪en el set de entrenamiento:
                valid['load'] = scaler.transform(valid)

                # 3. Cambiamos el dataframe para crear los samples de
↪entrada (input).
                valid_shifted = valid.copy()
                valid_shifted['y+1'] = valid_shifted['load'].shift(-1,
↪freq='H')
                for t in range(1, T+1):
                valid_shifted['load_t-'+str(T-t)] = valid_shifted['load'].
↪shift(T-t, freq='H')

                # 4. Eliminamos (dropeamos) cualquier sample con missing
↪value
                valid_shifted = valid_shifted.dropna(how='any')
                valid_shifted.head(3)
```

[23]:
```
                load  y+1  load_t-5  load_t-4  load_t-3  load_t-2  \
                2014-09-01 00:00:00  0.28 0.24      0.61      0.58      0.
↪51      0.43
                2014-09-01 01:00:00  0.24 0.22      0.58      0.51      0.
↪43      0.34
                2014-09-01 02:00:00  0.22 0.22      0.51      0.43      0.
↪34      0.28

                load_t-1  load_t-0
                2014-09-01 00:00:00      0.34      0.28
```

```
2014-09-01 01:00:00      0.28      0.24
2014-09-01 02:00:00      0.24      0.22
```

[24]:
```
# 5. Transformamos el dataframe en un vector
y_valid = valid_shifted['y+1'].to_numpy()
X_valid = valid_shifted[['load_t-'+str(T-t) for t in␣
↪range(1, T+1)]].to_numpy()
X_valid = X_valid.reshape(X_valid.shape[0], T, 1)
```

Verificando las shapes:

[25]:
```
y_valid.shape
```

[25]:
```
(1463,)
```

[26]:
```
y_valid[:3]
```

[26]:
```
array([0.24, 0.22, 0.22])
```

[27]:
```
X_valid.shape
```

[27]:
```
(1463, 6, 1)
```

Ya preparados los conjuntos, ahora debemos construir la estructura del modelo RNN e implementarlo. Para facilitar una mejor intuición, nuestro modelo, visualmente, será de la siguiente forma:

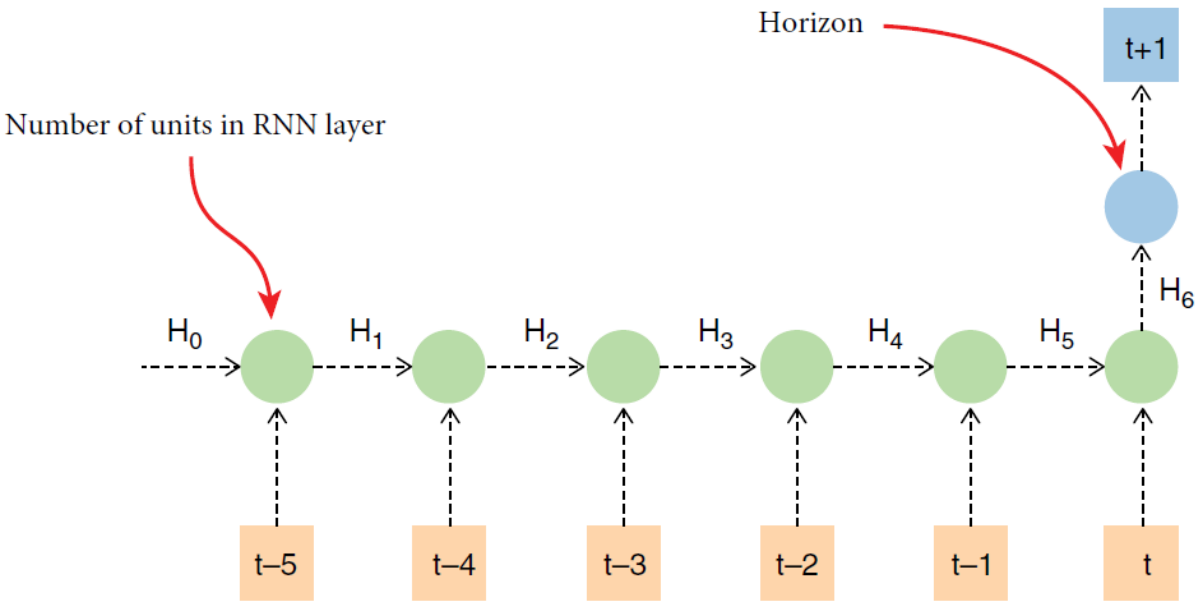


Figure 15: Estructura de un modelo RNN simple para usar en Keras

Importamos los paquetes necesarios:

[28]:
```
from keras.models import Model, Sequential
from keras.layers import GRU, Dense
from keras.callbacks import EarlyStopping
```

Calibramos los hiperparametros:

[29]:
```
LATENT_DIM = 5 # Hiperparametro de nro unidades en la␣
↪capa RNN
BATCH_SIZE = 32 # Hiperparametro: nro de samples por␣
↪mini-batch
EPOCHS = 10 # Maximo nro de veces que el algo de␣
↪entrenamiento iterará todos los samples
```

[30]:
```
model = Sequential()   # Definimos (con Keras) al modelo␣
↪como uno Secuencial
model.add(GRU(LATENT_DIM, input_shape=(T, 1))) #␣
↪Agregamos una GRU de 5 unidades (LATENT_DIM)
model.add(Dense(HORIZON)) # Dense Layer de la NN de 1␣
↪unidad
```

Usaremos Mean Squared Error como la función de pérdida (objetivo a minimizar).

[31]:
```
model.compile(optimizer='RMSprop', loss='mse')  #␣
↪Especificamos el algoritmo de aprendizaje y la funcion objetivo
```

Nótese que hemos elegido, como algoritmo de optimización, el de *RMSProp*. Básicamente, la elección se basa en una recomendación de la documentación oficial de Keras en utilizar ese algortimo específico para modelos de RNN.[8] La explicación algebraica e intuitiva de RMSProp se puede encontrar en el Anexo.

En resumen, el modelo queda como:

[32]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 5)                 120
_________________________________________________________________
dense (Dense)                (None, 1)                 6
=================================================================
Total params: 126
Trainable params: 126
Non-trainable params: 0
_________________________________________________________________
```

[8] https://faroit.com/keras-docs/2.0.2/optimizers/

Especificamos el criterio para early stopping (detener el aprendizaje). Monitorearemos la pérdida de validación (en este caso, del Mean Squared Error) en el set de validación luego de cada training epoch. Si la pérdida de validación no mejora en *min_delta* luego de *patience* epochs, detendremos el training.

[33]:
```
                earlystop = EarlyStopping(monitor='val_loss',␣
↪min_delta=0, patience=5)
                # Esta clase (EarlyStopping) es para detener el training␣
↪cuando una métrica deja de cambiar
                # Parámetros: monitoreamos el valor de la loss function,␣
↪min_delta es el delta mínimo a considerar como mejora de la f.objetivo
                # y patience es el número de epochs sin cambio alguno␣
↪luego de los cuales se detiene el training
```

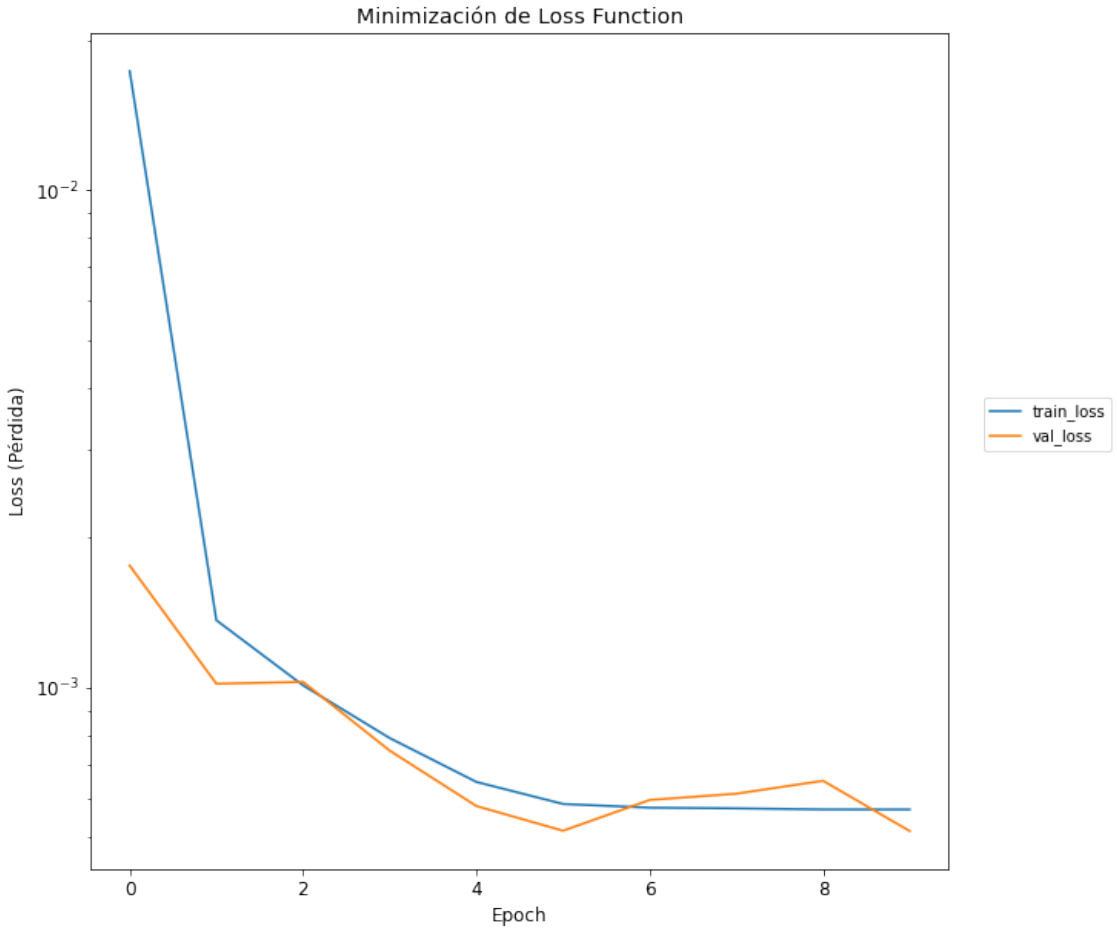
Finalmente, podemos inicializar el modelo, entrenandolo y graficando la función de pérdida, tanto para el set de entrenamiento como de validación:

[34]:
```
                # Corremos el modelo y comienza a entrenarse. Ver cómo␣
↪disminuye el "val_loss". En la próxima celda lo ploteamos
                history = model.fit(X_train,
                y_train,
                batch_size=BATCH_SIZE,
                epochs=EPOCHS,
                validation_data=(X_valid, y_valid),
                callbacks=[earlystop],
                verbose=1)
```

```
      Epoch 1/10
      731/731 [==============================] - 31s 7ms/step - loss: 0.
↪0511 -
      val_loss: 0.0018
      Epoch 2/10
      731/731 [==============================] - 4s 5ms/step - loss: 0.
↪0016 -
      val_loss: 0.0010
      Epoch 3/10
      731/731 [==============================] - 6s 8ms/step - loss: 0.
↪0011 -
      val_loss: 0.0010
      Epoch 4/10
      731/731 [==============================] - 7s 9ms/step - loss: 8.
↪3488e-04 -
      val_loss: 7.4674e-04
      Epoch 5/10
      731/731 [==============================] - 4s 5ms/step - loss: 6.
↪8352e-04 -
      val_loss: 5.7788e-04
```

```
Epoch 6/10
731/731 [==============================] - 3s 5ms/step - loss: 5.
↪8420e-04 -
val_loss: 5.1545e-04
Epoch 7/10
731/731 [==============================] - 4s 5ms/step - loss: 5.
↪7446e-04 -
val_loss: 5.9434e-04
Epoch 8/10
731/731 [==============================] - 4s 6ms/step - loss: 5.
↪6909e-04 -
val_loss: 6.1180e-04
Epoch 9/10
731/731 [==============================] - 4s 5ms/step - loss: 5.
↪7335e-04 -
val_loss: 6.4949e-04
Epoch 10/10
731/731 [==============================] - 4s 5ms/step - loss: 5.
↪6600e-04 -
val_loss: 5.1472e-04
```

[35]:
```
plot_df = pd.DataFrame.from_dict({'train_loss':history.
↪history['loss'], 'val_loss':history.history['val_loss']})
plot_df.plot(logy=True, figsize=(10,10), fontsize=12)
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss (Pérdida)', fontsize=12)
plt.title('Minimización de Loss Function', fontsize=14)
plt.legend(loc=(1.04,0.5))
plt.show()
```



4.3 Resultados

Ahora, resta evaluar el modelo. Debemos, nuevamente, preparar el set de datos para el test.

[36]:
```
                # Mismo proceso que los otros sets
                # 1. Tomamos los datos de testeo del rango correcto
                look_back_dt = dt.datetime.strptime(test_start_dt,␣
↪'%Y-%m-%d %H:%M:%S') - dt.timedelta(hours=T-1)
                test = energy.copy()[test_start_dt:][['load']]

                # 2. Escalamos los datos, igual que antes
                test['load'] = scaler.transform(test)

                # 3. Modificamos el dataframe para crear los samples de␣
↪input
                test_shifted = test.copy()
```

```
                test_shifted['y_t+1'] = test_shifted['load'].shift(-1,␣
↪freq='H')
                for t in range(1, T+1):
                test_shifted['load_t-'+str(T-t)] = test_shifted['load'].
↪shift(T-t, freq='H')

                # 4. Descartamos los samples con missing values
                test_shifted = test_shifted.dropna(how='any')

                # 5. Transformamos el dataframe en un vector
                y_test = test_shifted['y_t+1'].to_numpy()
                X_test = test_shifted[['load_t-'+str(T-t) for t in␣
↪range(1, T+1)]].to_numpy()
                X_test = X_test.reshape(X_test.shape[0], T, 1)
```

[37]:
```
                y_test.shape
```

[37]:
```
                (1458,)
```

[38]:
```
                X_test.shape
```

[38]:
```
                (1458, 6, 1)
```

Y, realizar las predicciones, es decir, el test:

[39]:
```
                # Predicción:
                predictions = model.predict(X_test)  # Usamos el método␣
↪predict() y le pasamos el único argumento obligatorio: el input vector
                predictions
```

[39]:
```
                array([[0.21],
                [0.29],
                [0.38],
                ...,
                [0.52],
                [0.46],
                [0.42]], dtype=float32)
```

Ahora realizaremos la evaluación del modelo y luego computamos el MAPE sobre todas las predicciones:

[40]:
```
                # Evaluación:
                eval_df = pd.DataFrame(predictions, columns=['t+'+str(t)␣
↪for t in range(1, HORIZON+1)])
                eval_df['timestamp'] = test_shifted.index
                eval_df = pd.melt(eval_df, id_vars='timestamp',␣
↪value_name='prediction', var_name='h')
```

```
                eval_df['actual'] = np.transpose(y_test).ravel()
                eval_df[['prediction', 'actual']] = scaler.
↪inverse_transform(eval_df[['prediction', 'actual']])
                eval_df.head()
```

[40]:

	timestamp	h	prediction	actual
0	2014-11-01 05:00:00	t+1	2,661.91	2,714.00
1	2014-11-01 06:00:00	t+1	2,930.38	2,970.00
2	2014-11-01 07:00:00	t+1	3,199.11	3,189.00
3	2014-11-01 08:00:00	t+1	3,336.47	3,356.00
4	2014-11-01 09:00:00	t+1	3,464.77	3,436.00

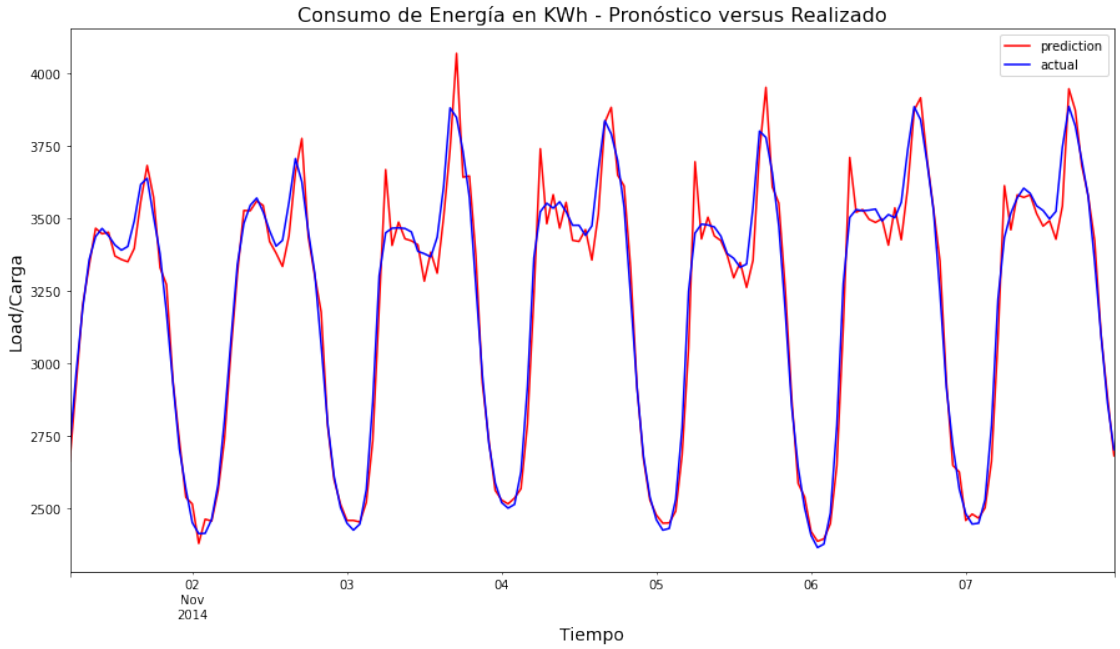
[41]:
```
                # Armamos nuestra propia función para computar el MAPE
                def mape(predictions, actuals):
                """Mean absolute percentage error"""
                return ((predictions - actuals).abs() / actuals).mean()
```

[42]:
```
                mape(eval_df['prediction'], eval_df['actual'])
```

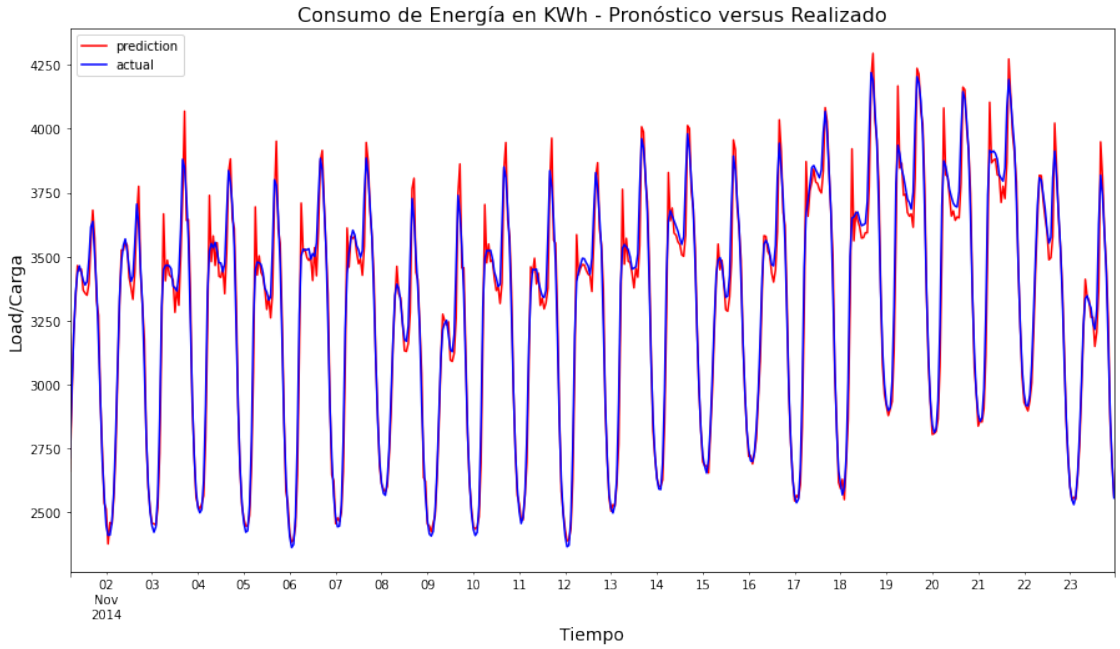
[42]:
```
                0.016104323551220903
```

Por último, graficamos la predicción versus lo realizado en diferentes rangos de tiempo así vemos en mayor o menor detalle:

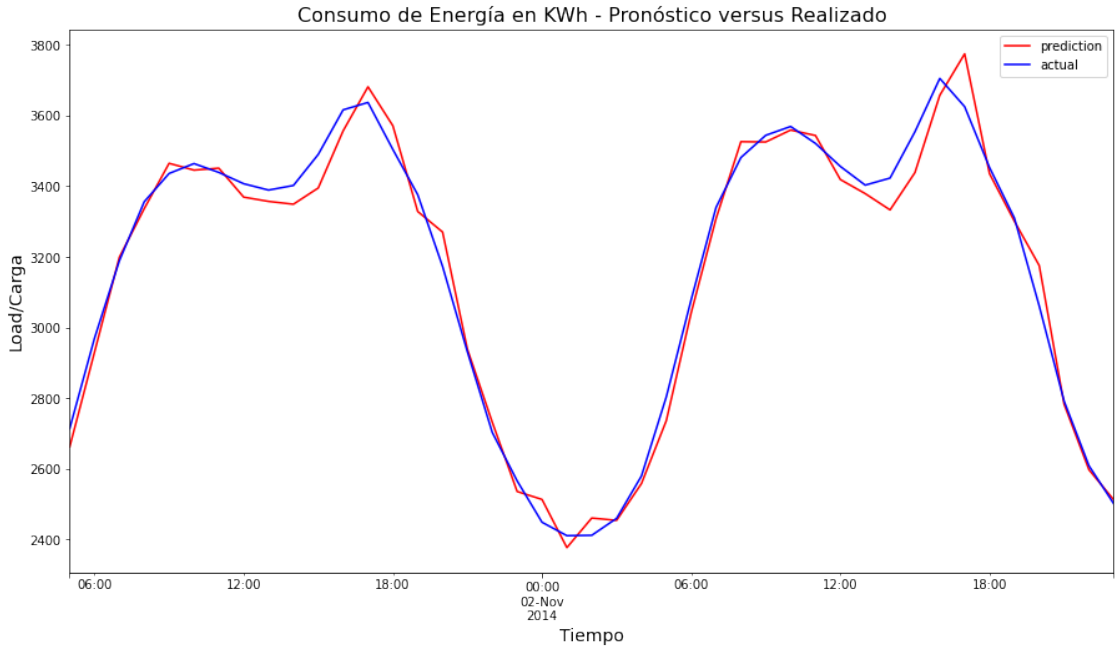
[43]:
```
                # Grafico de 1 semana
                eval_df[eval_df.timestamp<'2014-11-08'].
↪plot(x='timestamp', y=['prediction', 'actual'], style=['r', 'b'],␣
↪figsize=(15, 8))
                plt.xlabel('Tiempo', fontsize=14)
                plt.ylabel('Load/Carga', fontsize=14)
                plt.title('Consumo de Energía en KWh - Pronóstico versus␣
↪Realizado', fontsize=16)
                plt.show()
```



[45]:
```
                # Grafico de 3 semanas
                eval_df[eval_df.timestamp<'2014-11-24'].
↪plot(x='timestamp', y=['prediction', 'actual'], style=['r', 'b'],␣
↪figsize=(15, 8))
                plt.xlabel('Tiempo', fontsize=14)
                plt.ylabel('Load/Carga', fontsize=14)
                plt.title('Consumo de Energía en KWh - Pronóstico versus␣
↪Realizado', fontsize=16)
                plt.show()
```



[48]:
```
                # Grafico de 2 días
                eval_df[eval_df.timestamp<'2014-11-03'].
↪plot(x='timestamp', y=['prediction', 'actual'], style=['r', 'b'],␣
↪figsize=(15, 8))
                plt.xlabel('Tiempo', fontsize=14)
                plt.ylabel('Load/Carga', fontsize=14)
                plt.title('Consumo de Energía en KWh - Pronóstico versus␣
↪Realizado', fontsize=16)
                plt.show()
```



Así, hemos realizado nuestra predicción para los datos de consumo horario de energía, que en función de nuestra métrica error **MAPE** $= 1.61\%$ resulta ser bastante preciso out of sample. Si bien es un modelo univariable (univariate model) y de un solo paso (one step forecast: $t + 1$) resulta ser útil para los fines didácticos aquí presentes.

Si bien podríamos realizar una calibración de los hiperparámetros elegidos (hyperparameter tunning), excedería los objetivos del presente modelo, que para nuestro objetivo, ya ha dado buenos resultados con la calibración inicial.

5 Aplicación Práctica Nº2: Energy Consumption Forecast con ARIMA

Para esta aplicación, ahora sí nos valeremos de los datos[9] extraídos de la Secretaría de Energía de la República Argentina, específicamente de los datos de la Compañía Administradora del Mercado Mayorista Eléctrico S.A. (CAMMESA). Este dataset posee el consumo energético (o demanda) desde el año 2012 hasta febrero de 2020 (pre COVID-19), desglosado por categoría de usuario y región del país. Si bien el archivo .csv (comma separated values) correspondiente posee 122.263 observaciones totales, filtrando por region, usuario y agrupando repetidos (porque para un mismo usuario, de una misma región, se puede tener con distintas categorías de tarifa), llegaremos a 98 observaciones en total: exactamente la cantidad de meses desde enero 2012 hasta febrero 2020.

Con esto en mente, el objetivo de la presente aplicación es poder hacer uso de una técnica estadística clásica (Autoregressive Integrated Moving Average - ARIMA) para el caso cuando no se disponen de muchos datos (como en la aplicación Nº1, que se disponía de 23370 observaciones), y demostrar así, que aun podemos obtener resultados interesantes dentro del marco didáctico nuestro (y como potencial ejemplo práctico de la vida real, donde se requieran pronósticos con pocos datos brindados por parte del usuario).

5.1 Procesamiento y Preparación de los Datos de Argentina (2012-2020)

Primero y antes que nada, realizamos la importación de paquetes necesarios en una primera etapa.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Importamos el dataset de la misma carpeta local que la Jupyter Notebook e imprimimos un resumen del archivo al llamar a los valores unicos del dataframe (con df.nunique):

[2]:
```
# Carga del dataset
df = pd.read_csv('demanda-histrica.csv')    # Sin pasar␣
↪parametro "head=" las columnas vienen indexadas por su nombre propio!

# Resumen de los valores únicos que puede tomar cada␣
↪columna
print(df.nunique())
```

```
anio                    9
mes                    12
agente_nemo           591
agente_descripcion    610
tipo_agente             4
region                  9
provincia              22
```

[9] Fuente del dataset: http://datos.minem.gob.ar/dataset/publicaciones-cammesa

```
categoria_area            11
categoria_demanda          2
tarifa                    37
categoria_tarifa           4
demanda_MWh           104400
indice_tiempo             98
dtype: int64
```

Como mencionamos al inicio, tenemos 9 años (2012 a 2020), 12 meses por año, 4 tipos de agentes o usuarios, 9 regiones y 22 provincias, entre otras columnas.

Si visualizamos los primeros datos del DataFrame:

[3]:
```
df.head()
```

[3]:
```
   anio  mes agente_nemo           agente_descripcion tipo_agente  \
0  2012    1    AARGTAOY   AEROP ARG 2000 - Aeroparque         GU
1  2012    1    ACARQQ3Y      ASOC.COOP.ARG. - Quequén         GU
2  2012    1    ACARSLSY  ASOC.COOP.ARG. - San Lorenzo         GU
3  2012    1    ACINROSY    ACINDAR ROSARIO EX-NAVARRO         GU
4  2012    1    ACINTBOY          ACINDAR PTA. TABLADA         GU

     region     provincia    categoria_area categoria_demanda  \
0   GRAN BS.AS.  BUENOS AIRES  Gran Usuario MEM      Gran Usuario
1  BUENOS AIRES  BUENOS AIRES  Gran Usuario MEM      Gran Usuario
2       LITORAL      SANTA FE  Gran Usuario MEM      Gran Usuario
3       LITORAL      SANTA FE  Gran Usuario MEM      Gran Usuario
4   GRAN BS.AS.  BUENOS AIRES  Gran Usuario MEM      Gran Usuario

                tarifa             categoria_tarifa  demanda_MWh  \
0  GUMAS/AUTOGENERADORES  Industrial/Comercial Grande     2222.628
1  GUMAS/AUTOGENERADORES  Industrial/Comercial Grande      536.858
```

```
                2  GUMAS/AUTOGENERADORES  Industrial/Comercial Grande      ␣
↪336.297
                3  GUMAS/AUTOGENERADORES  Industrial/Comercial Grande      ␣
↪601.066
                4  GUMAS/AUTOGENERADORES  Industrial/Comercial Grande      ␣
↪3076.618

                indice_tiempo
                0       2012-01
                1       2012-01
                2       2012-01
                3       2012-01
                4       2012-01
```

Y, para tenerlo en cuenta, nótese que la dimensión del DF es (122263, 13)

[4]:
```
df.shape
```

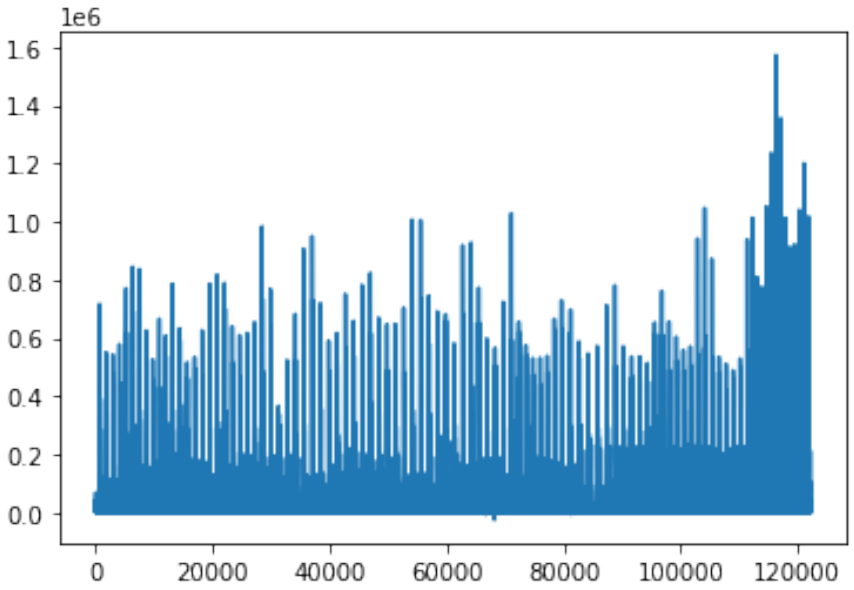
[4]:
```
(122263, 13)
```

La cantidad de 122263 corresponde a lo que comentamos antes, tendremos muchos valores debido a la numerosa cantidad de categorías/columnas, pero que en definitiva no nos dejará más que 98 obeservaciones a fin de cuentas.

Un gráfico preliminar de lo que hay para la categoría de Demanda en MWh:

[7]:
```
plt.plot(df1['demanda_MWh'])
```

[7]:
```
[<matplotlib.lines.Line2D at 0x2248838d460>]
```



Ahora, debemos quitar todos los valores que no correspondan a variables que nos interesen. Nosotros trabajaremos solamente con la categoría 'GU' que significa Gran Usuario, por ende:

[8]:
```
df1.drop(df1.loc[df1['tipo_agente']!='GU'].index, inplace=True)   # Queremos quitar los valores de tipo_agente que no vamos a usar para la prediccion
```

Luego, filtramos eliminando por región, ya que elegiremos solamente Buenos Aires para este ejercicio.

[10]:
```
df1.drop(df1.loc[df1['region']!='BUENOS AIRES'].index, inplace=True)
```

[11]:
```
df1  # Ahora lo que sucede es que se repiten valores para GU y Region
```

[11]:

	tipo_agente	region	demanda_MWh	indice_tiempo
1	GU	BUENOS AIRES	536.858	2012-01
12	GU	BUENOS AIRES	3947.187	2012-01
16	GU	BUENOS AIRES	2011.403	2012-01
31	GU	BUENOS AIRES	2703.511	2012-01
53	GU	BUENOS AIRES	2283.633	2012-01
...	...	...	...	...
122220	GU	BUENOS AIRES	471.208	2020-02
122233	GU	BUENOS AIRES	758.932	2020-02
122234	GU	BUENOS AIRES	827.217	2020-02
122235	GU	BUENOS AIRES	1400.088	2020-02
122237	GU	BUENOS AIRES	999.250	2020-02

[5352 rows x 4 columns]

Lo último que nos falta es agregar los valores de la demanda en MWh para mismo tipo de agente e índice de tiempo, recordemos que esto sucedía si había otras variables como la categoría de tarifa que los segmentaba un poco más.

[13]:
```
df1 = df1.groupby('indice_tiempo')['demanda_MWh'].sum().
↪reset_index()  # Agregamos los valores para los índices que se repiten
```

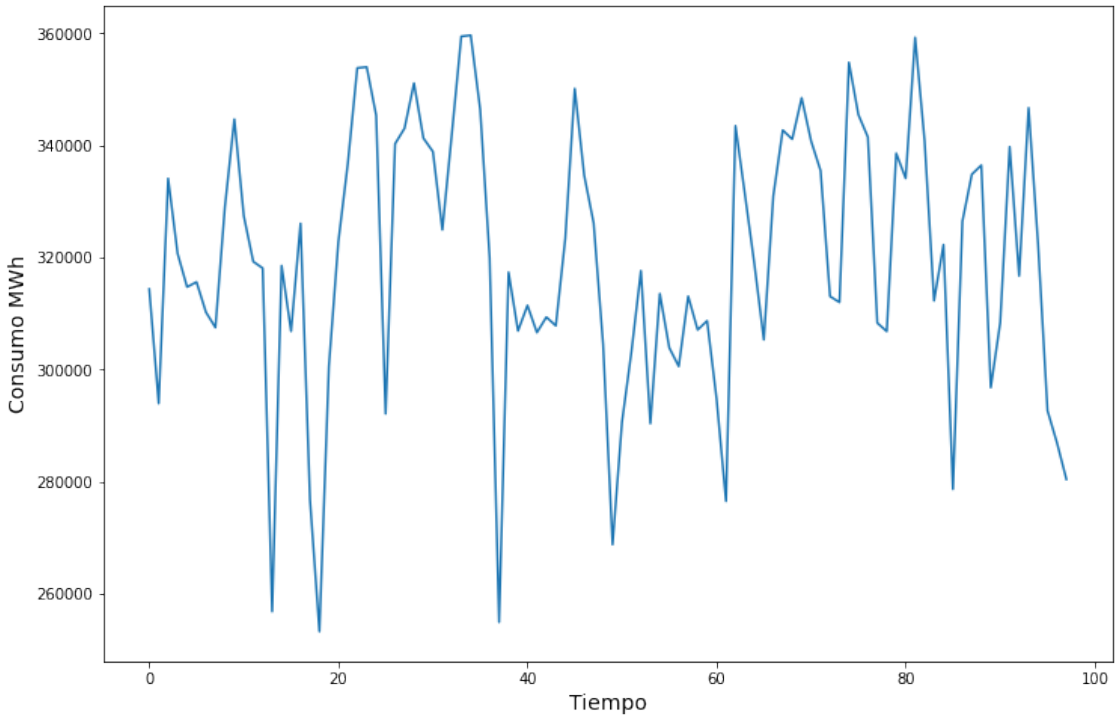
[14]:
```
df1.head()
```

[14]:
```
  indice_tiempo  demanda_MWh
0       2012-01   314345.008
1       2012-02   293941.443
2       2012-03   334085.729
3       2012-04   320688.069
4       2012-05   314725.504
```

Vemos que ahora sí, tenemos solamente un dataset más acabado en términos de columnas y valores. Solo tenemos nuestro índice (la fecha) y el valor de consumo energético en MWh asociado a este. Hagamos un gráfico preliminar de la demanda energética:

[15]:
```
fig = plt.figure(figsize=(12,8))
plt.xlabel('Tiempo', fontsize=14)
plt.ylabel('Consumo MWh', fontsize=14)
plt.plot(df1['demanda_MWh'])
```

[15]:
```
[<matplotlib.lines.Line2D at 0x22488d27040>]
```



Lo que aún resta hacer dentro del procesamiento y preparación de los datos es ajustar

el formato del índice. Como dijimos, son fechas, pero en formato "RangeIndex", que es el parametrizado por Pandas cuando no se especifica al momento de importar el archivo comma separated values. Nosotros queremos trabajar con el formato "Period" de Pandas para las fechas.

[16]:
```
print(df1.index)  # Todavía tenemos que corregir el índice, está en formato "RangeIndex"
```

```
RangeIndex(start=0, stop=98, step=1)
```

Lo que haremos es, parsear esa columna a clase DateTime de Pandas, ya que es una operación permitida por el método to_datetime de Pandas.

[17]:
```
df1.index = pd.to_datetime(df1['indice_tiempo'], format='%Y%m%')  # Primero lo pasamos a DateTime
```

Luego, usaremos el método "to_period" especificando como argumento una frecuencia (que será mensual en nuestro caso) y así hemos terminado de ajustar nuestras fechas. Nótese que hicimos este paso intermedio parseando a DateTime primero, ya que si intentabamos antes con Period no era una operación posible, dado que arrojaría error.

[19]:
```
df1.index.to_period("M")   # Luego, de DateTime a Period
```

[19]:
```
PeriodIndex(['2012-01', '2012-02', '2012-03', '2012-04', '2012-05', '2012-06',
             '2012-07', '2012-08', '2012-09', '2012-10', '2012-11', '2012-12',
             '2013-01', '2013-02', '2013-03', '2013-04', '2013-05', '2013-06',
             '2013-07', '2013-08', '2013-09', '2013-10', '2013-11', '2013-12',
             '2014-01', '2014-02', '2014-03', '2014-04', '2014-05', '2014-06',
             '2014-07', '2014-08', '2014-09', '2014-10', '2014-11', '2014-12',
             '2015-01', '2015-02', '2015-03', '2015-04', '2015-05', '2015-06',
             '2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
             '2016-01', '2016-02', '2016-03', '2016-04', '2016-05', '2016-06',
             '2016-07', '2016-08', '2016-09', '2016-10', '2016-11', '2016-12',
             '2017-01', '2017-02', '2017-03', '2017-04', '2017-05', '2017-06',
             '2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
```

```
               '2018-01', '2018-02', '2018-03', '2018-04', '2018-05',␣
↪'2018-06',
               '2018-07', '2018-08', '2018-09', '2018-10', '2018-11',␣
↪'2018-12',
               '2019-01', '2019-02', '2019-03', '2019-04', '2019-05',␣
↪'2019-06',
               '2019-07', '2019-08', '2019-09', '2019-10', '2019-11',␣
↪'2019-12',
               '2020-01', '2020-02'],
              dtype='period[M]', name='indice_tiempo', freq='M')
```

5.2 Construcción del modelo

Ahora, podemos proceder a construir nuestro modelo, que será un poco más sencillo que la aplicación Nº1. Primero, importamos el paquete necesario para esta etapa, que será ARIMA de la liberaria Statsmodels.

[20]:
```
from statsmodels.tsa.arima.model import ARIMA
```

Directamente crearemos un objeto "modelo" que será la función ARIMA con los argumentos del dataset y el orden de autoregresion. Luego, cuando usamos el método fit() estaremos corriendo el modelo con los respectivos datos.

[21]:
```
# Fit del modelo:
model = ARIMA(df1['demanda_MWh'], order=(5,1,0))
model_fit = model.fit()

# Resumen del Ajuste (fit) del modelo
print(model_fit.summary())
```

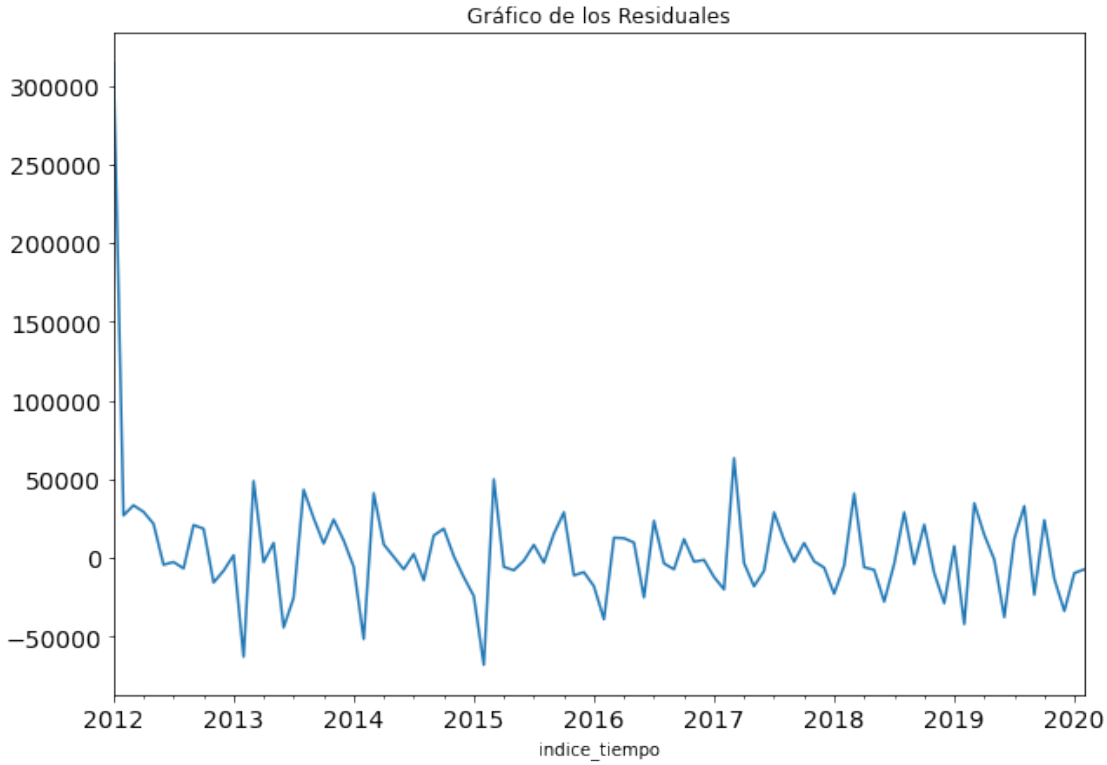
```
      SARIMAX Results
      ==========================================================================
      Dep. Variable:            demanda_MWh   No. Observations:          ␣
↪        98
      Model:                 ARIMA(5, 1, 0)   Log Likelihood             ␣
↪ -1115.694
      Date:                Sun, 13 Jun 2021   AIC                       ␣
↪  2243.388
      Time:                        23:10:00   BIC                       ␣
↪  2258.836
      Sample:                    01-01-2012   HQIC                      ␣
↪  2249.634
      - 02-01-2020
      Covariance Type:                  opg
      ==========================================================================
      coef    std err          z      P>|z|      [0.025      0.975]
      --------------------------------------------------------------------------
```

```
      ar.L1         -0.1577      0.050     -3.184      0.001      -0.255 ␣
↪    -0.061
      ar.L2         -0.0034      0.053     -0.064      0.949      -0.106 ␣
↪     0.100
      ar.L3         -0.1325      0.049     -2.712      0.007      -0.228 ␣
↪    -0.037
      ar.L4         -0.0991      0.059     -1.668      0.095      -0.215 ␣
↪     0.017
      ar.L5          0.0239      0.101      0.237      0.813      -0.174 ␣
↪     0.222
      sigma2      4.754e+08   3.26e-11   1.46e+19      0.000    4.75e+08 ␣
↪  4.75e+08
      ==============================================================================
      ===
      Ljung-Box (L1) (Q):                   1.31   Jarque-Bera (JB):
      1.39
      Prob(Q):                              0.25   Prob(JB):
      0.50
      Heteroskedasticity (H):               0.69   Skew:
      -0.16
      Prob(H) (two-sided):                  0.30   Kurtosis:
      3.49
      ==============================================================================
      ===

      Warnings:
      [1] Covariance matrix calculated using the outer product of␣
↪gradients (complex-
      step).
      [2] Covariance matrix is singular or near-singular, with condition␣
↪number
      6.78e+34. Standard errors may be unstable.
```

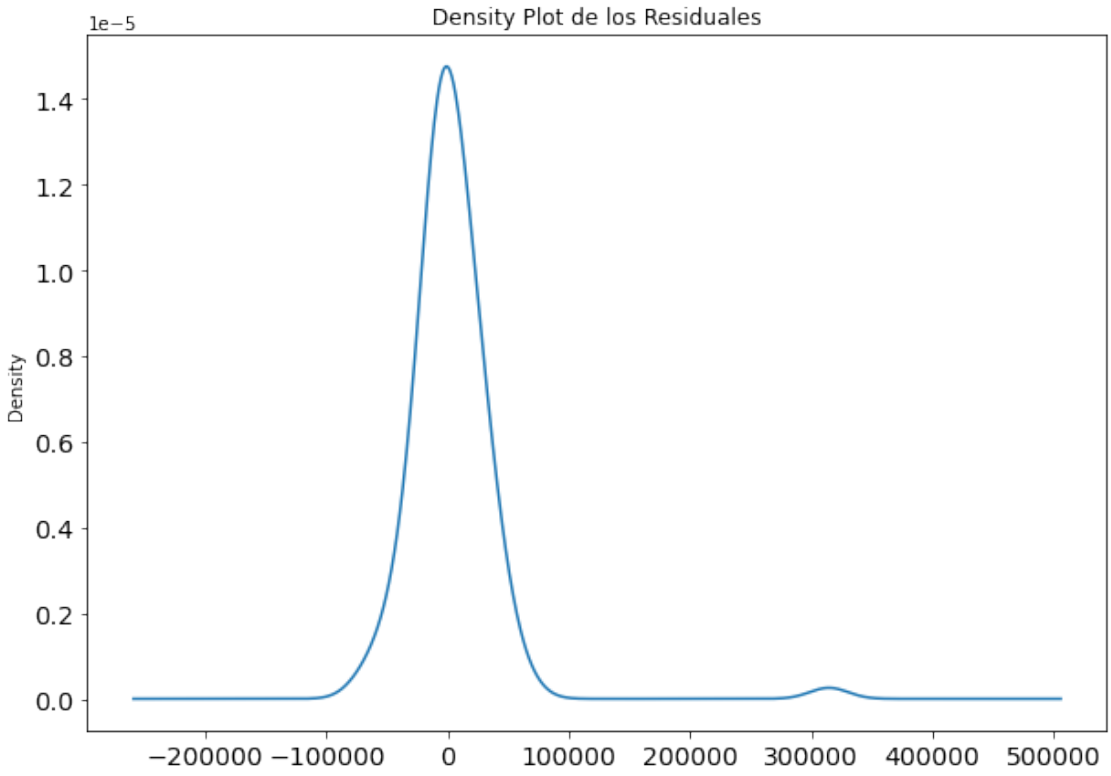
Como se ve en la opción Summary para el resumen, tenemos la salida de la ARIMA. Luego, podemos graficar los residuales:

[22]:
```
# Gráfico de los Residuales
residuals = pd.DataFrame(model_fit.resid)
residuals.plot(figsize=(10,7),fontsize=14, legend=None)
plt.title('Gráfico de los Residuales')
plt.show()
```



Y también hacer un gráfico de densidad de estos, junto con una salida de estadística descriptiva:

[23]:
```
                # Gráfico de Densidad de los Residuales
                residuals.plot(kind='kde', figsize=(10,7), fontsize=14,␣
↪legend=None)
                plt.title('Density Plot de los Residuales')
                plt.show()
                # Summary stats de los residuals
                print('Estadística Descriptiva de los Residuales: \n',␣
↪residuals.describe())
```




```
Estadística Descriptiva de los Residuales:
0
count       98.000000
mean      3933.412352
std      39590.047467
min     -67921.898848
25%      -9606.115659
50%      -2383.602218
75%      15207.273749
max     314345.008000
```

Sin embargo, nuestro objetivo primario es realizar un pronóstico que podamos comparar con datos verdaderos. Para esto, dividamos nuestro dataset en entrenamiento y test (no habrá validación para esta técnica). La regla será la tradicional de 70/30 aproximadamente. Nótese la constante de 0.66 que usamos para distinguir cuando dividir los datos en dos:

[24]:
```
# División del dataset entre training y test
X = df1['demanda_MWh']
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
```

Y ahora, correr el pronóstico para cada observación como parte de un loop, donde almacenaremos los resultados en una lista, pero imprimiremos cada valor pronosticado versus el realizado:

[25]:
```
# Validación Walk-Forward
for t in range(len(test)):
model = ARIMA(history, order=(5,1,0))
model_fit = model.fit()
output = model_fit.forecast()
yhat = output[0]
predictions.append(yhat)
obs = test[t]
history.append(obs)
print('Predicted=%f, Expected=%f' % (yhat, obs))
```

```
Predicted=338613.274520, Expected=318816.832000
Predicted=314695.514747, Expected=305314.387000
Predicted=302653.133037, Expected=330970.867000
Predicted=329927.249770, Expected=342707.866000
Predicted=343959.036635, Expected=341093.330000
Predicted=339608.417788, Expected=348462.871000
Predicted=344502.973995, Expected=340762.018000
Predicted=341756.404631, Expected=335488.979000
Predicted=335436.825118, Expected=313051.819000
Predicted=316015.015500, Expected=311989.861000
Predicted=312563.974813, Expected=354788.096000
Predicted=351933.334244, Expected=345467.312000
Predicted=349136.845816, Expected=341478.171000
Predicted=337054.830654, Expected=308299.985000
Predicted=310737.925026, Expected=306808.471000
Predicted=308136.534248, Expected=338566.771000
Predicted=338345.269236, Expected=334118.127000
Predicted=337466.561641, Expected=359252.775000
Predicted=351437.676438, Expected=340826.643000
Predicted=341886.610859, Expected=312271.248000
Predicted=313900.027930, Expected=322265.147000
Predicted=320711.012342, Expected=278631.034000
Predicted=290314.563032, Expected=326445.176000
Predicted=319536.964926, Expected=334830.024000
Predicted=338057.810143, Expected=336449.368000
Predicted=334775.419087, Expected=296781.774000
Predicted=296190.665116, Expected=308376.999000
Predicted=306456.601398, Expected=339738.643000
Predicted=340527.073862, Expected=316704.514000
Predicted=321968.952530, Expected=346688.420000
Predicted=335709.967943, Expected=322431.883000
Predicted=326868.096322, Expected=292661.430000
```

```
Predicted=296715.716248, Expected=286940.648000
Predicted=287816.967836, Expected=280401.839000
```

5.3 Resultados

Finalmente, nos resta evaluar el modelo. Para evaluar los pronósticos, en este caso, usaremos una métrica más simple a diferencia de otras, que será la media del error cuadrático:

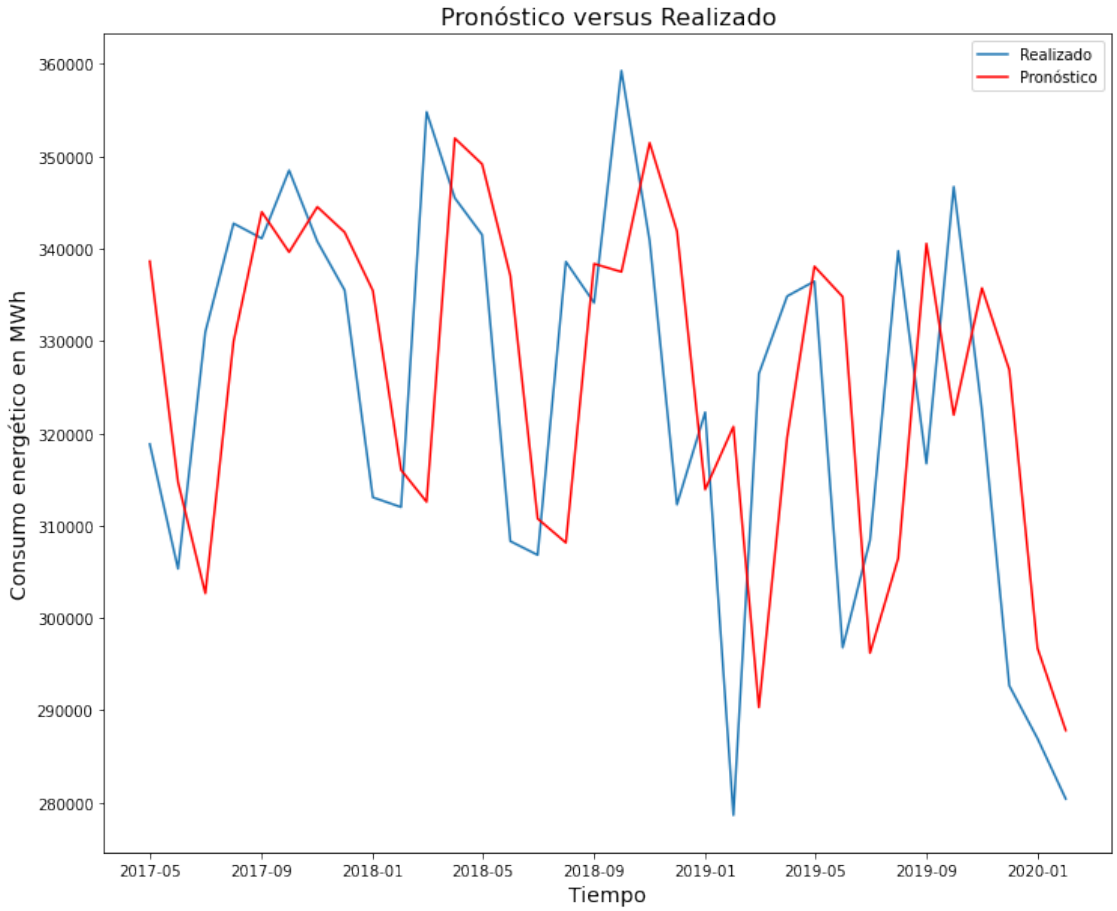
[26]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt
```

[27]:
```
# Evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test Root Mean Squared Error: %.3f' % rmse)
```

```
Test Root Mean Squared Error: 21688.928
```

Y por último, mostrar visualmente nuestros resultados pronósticados versus los realizados:

[28]:
```
# Plot forecasts against actual outcomes
fig = plt.figure(figsize=(12,10))
plt.plot(test, label='Realizado')
plt.plot(test.index,predictions, color='red',␣
↪label='Pronóstico')
plt.ylabel('Consumo energético en MWh',fontsize=14)
plt.xlabel('Tiempo',fontsize=14)
plt.title('Pronóstico versus Realizado', fontsize=16)
plt.legend()
plt.show()
```



Como se puede observar, para tratarse de un modelo alimentado con tan solo 98 observaciones, logramos capturar la variabilidad de los datos en su forma general. Aunque el error es en verdad alto relativamente, podemos asegurar las tendencias en los intervalos acotados de tiempo. A fines prácticos, esto puede ser útil si se quiere tener una idea general a largo plazo del comportamiento del consumo de energía a nivel agregado en todo Buenos Aires. Sin embargo, si uno quisiera realizar un análisis a corto plazo, como puede ser el de una distribuidora (por ejemplo, Edenor) para preveer mantenimiento de transformadores o infraestructura relacionada ante de los picos de consumo energético, este Forecast no sería para nada adecuado. Para quien esté interiorizado con otras técnicas, podemos ver un patrón similar al de una media móvil clásica, de orden k.

Una alternativa podría ser incorporar predictores como la temperatura, que tal vez permita ayudar a ajustar un poco mejor el modelo. Sin embargo, para servir los fines didácticos de este trabajo, hemos alcanzado resultados interesantes.

En la próxima sección podemos hacer los comentarios y conclusiones finales.

6 Conclusiones

En esta última sección, haremos unos comentarios finales sobre ambos métodos y sus alcances como aplicaciones de Data Science. No es posible hacer un análisis comparativo entre ambos, ya que son, en términos relativos y objetivos, aplicaciones totalmente distintas, dado que se implementan en set de datos diferentes, con métricas de evaluación y objetivos diferentes también. Dicho esto, sin embargo, podemos distinguir ciertos aspectos que son importantes a tener en cuenta como potenciales científicos de datos o usuarios.

En primer lugar, podemos notar que los métodos estadísticos tradicionales, explicados en la sección 2.2, se acomodan perfectamente para conjuntos de datos (muy) pequeños relativamente. Si comparamos el conjunto de datos de 2 años con mediciones horarias frente al conjunto de datos de Argentina de 8 años con mediciones mensuales, tenemos 23.370 y 98 observaciones respectivamente, algo que resulta completamente irrisorio si se considera la diferencia de 23.272 samples entre uno y otro.

De aquí es que podemos entender, intuitivamente, que nos enfrentaremos indefectiblemente con situaciones en las que los datos pueden faltar o que bien pueden ser poco accesibles. Sin embargo, en el repertorio de herramientas de un científico de datos debe haber métodologías aptas para dichas situaciones.

De todas maneras, cabe destacar que existen métodos para ampliar datasets cuando son relativamente díficiles de conseguir. Por ejemplo, en Computer Vision, se suele aplicar filtros, transformaciones y ajustes a las bases de datos compuestas por imágenes para clasificación. Entonces, al aplicar un filtro blur en la foto de un gato, tendríamos dos imagenes en vez de una: la original y la que incluye el filtro.

Dicho esto, consideremos que si se quisiera ampliar el dataset de 98 observaciones, usando algun tratamiento de Missing Values que hemos nombrado en la sección 2.1.3, podríamos aplicar Interpolación Lineal, por ejemplo. Aunque parece atractivo, sin embargo, eso induciría un sesgamiento notable de los datos, ya que infiere en la tendencia (mes a mes) del consumo, que podría variar por otros métodos reales que nosotros no tengamos en cuenta y al utilziar un valor promedio generado por interpolación omitimos eso.

Por esa razón, al trabajar con datos de series de tiempo de determinadas industrias, no suele ser tan fácil inferir arbitrariamente qué tratamiento será mejor contemplando también que no afecte los resultados del problema real: pronosticar el consumo de energía en Gran Buenos Aires. Así es que entra en juego el rol de los business experts que deben echar luz sobre estas dudas. Solamente alguien que tenga conocimientos técnicos y comerciales de dicha industria, podrá aclarar si los hogares, industrias o el tipo de usuario que sea, cambia mucho su consumo en la primera quincena o no.

En este punto, vale la pena destacar el famoso diagrama de Venn sobre la ciencia de datos en la siguiente Figura, donde se puede notar la importancia de los tres conocimientos para abordar un problema data-driven: business domain ; math & stats ; computer science:

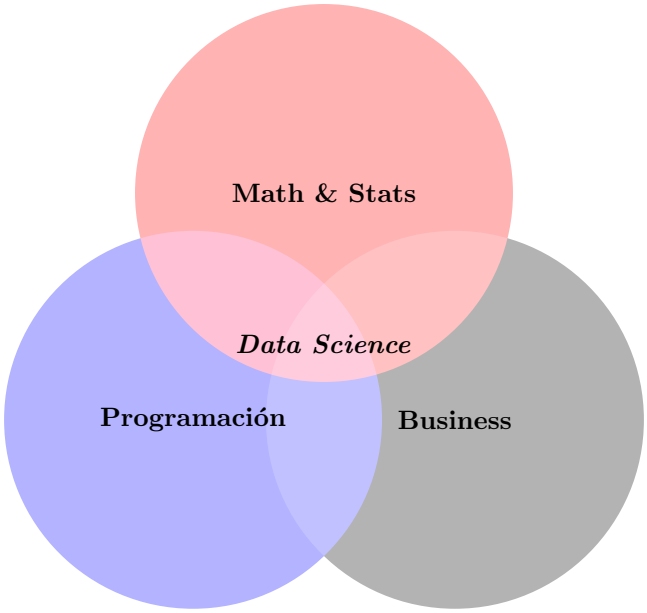


Figure 16: Diagrama de Venn de Data Science

Lo que resulta curioso, además, es que si una firma tuviera que preveer un pronóstico de demanda y solamente dispone de los datos públicos (oficiales del Estado), serían tan pocos que la reducirían a implementar solo un método clásico, poco preciso comparado relativamente a priori con métodos de Machine Learning.

Appendices

A continuación se encuentran todos los apéndices pertinentes para profundizar el álgebra (sin alcanzar niveles técnicamente difíciles) y mejorar la intuición del lector.

Appendix A Activation Functions

Las funciones de activación deciden si una neurona debe activarse o no calculando la suma ponderada y agregando más sesgo con ella. Son operadores diferenciables para transformar señales de entrada en salidas, mientras que la mayoría de ellos agregan no linealidad. Dado que las funciones de activación son fundamentales para el Deep Learning y cualquier estructura más allá de RNNs, analicemos brevemente algunas funciones de activación usadas:

1. ReLu (Rectified Linear Unit): La opción más popular, debido tanto a la simplicidad de implementación como a su buen desempeño en una variedad de tareas predictivas, es la unidad lineal rectificada (ReLU). ReLU proporciona una transformación no lineal muy simple. Dado un elemento x, la función se define como el máximo de ese elemento y 0:

$$\text{ReLU}(x) = \max(x, 0) \tag{1}$$

   De manera informal, la función ReLU retiene solo elementos positivos y descarta todos los elementos negativos estableciendo las activaciones correspondientes en 0. Para ganar algo de intuición, podemos trazar la función en un gráfico. Como se puede ver, la función de activación es lineal por partes:

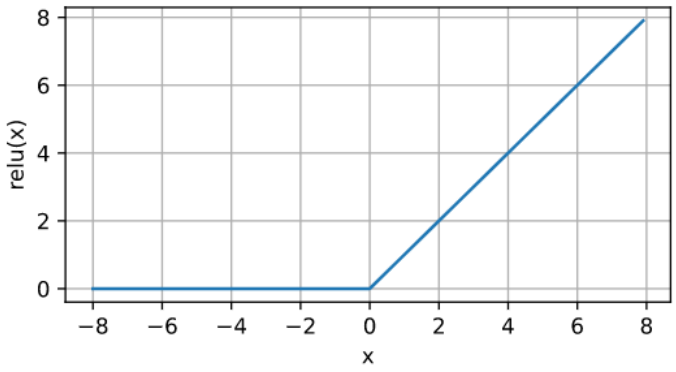


Figure 17: Función de activación ReLu

   Cuando la entrada es negativa, la derivada de la función ReLU es 0, y cuando la entrada es positiva, la derivada de la función ReLU es 1. Notese que la función ReLU no es diferenciable cuando la entrada toma un valor exactamente igual a 0. En estos casos, tomamos por defecto la derivada del lado izquierdo y decimos que la derivada es 0 cuando la entrada es 0. Podemos asumir esto último porque la entrada puede que nunca sea cero. La derivada de la función ReLU trazada a continuación se ve cómo:

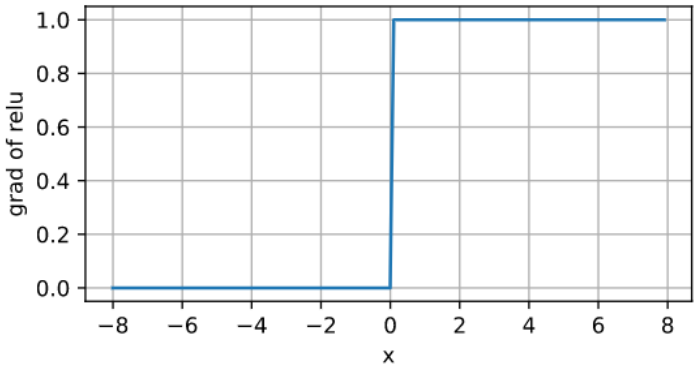


Figure 18: Derivada de la función de activación ReLu

La razón para usar ReLU es que sus derivadas se comportan particularmente bien: o desaparecen o simplemente dejan pasar el argumento. Esto hace que la optimización se comporte mejor y se mitigue el problema de los gradientes que desaparecen. Hay que tener en cuenta que hay muchas variantes de la función ReLU, incluida la función ReLU parametrizada (pReLU) [He et al., 2015]. Esta variación agrega un término lineal a ReLU, por lo que parte de la información aún se transmite, incluso cuando el argumento es negativo:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x) \tag{2}$$

2. Sigmoid: La función sigmoide transforma sus entradas, cuyos valores se encuentran en el dominio $\mathbb{R}$, en salidas que se encuentran en el intervalo (0, 1). Por esa razón, el sigmoide a menudo se denomina función de "aplastamiento" (squashing): aplasta cualquier entrada en el rango $(-\infty, \infty)$ a algún valor en el rango (0, 1):

$$\text{sigmoid}(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

En las primeras redes neuronales, los científicos estaban interesados en modelar neuronas biológicas, que se puedan disparar (fire) o no. Así, los pioneros de este campo, (desde McCulloch y Pitts, los inventores de la neurona artificial) se centraron en las unidades de umbral (conocidos en inglés como thresholding units). Una activación de umbral toma el valor 0 cuando su entrada está por debajo de algún umbral y el valor 1 cuando la entrada excede dicho umbral.

Cuando la atención se centró en el aprendizaje basado en gradientes, la función sigmoidea fue una elección natural porque es una aproximación suave y diferenciable a una unidad de umbral. Los sigmoides todavía se utilizan ampliamente como funciones de activación en las capas o layers de salida de una red neuronal, cuando queremos interpretar las salidas como probabilidades para problemas de clasificación binaria (se puede pensar en el sigmoide como un caso especial de la función softmax también). Sin embargo, el sigmoide ha sido reemplazado principalmente por el ReLU más simple y fácil de entrenar para la mayoría de uso en capas ocultas.

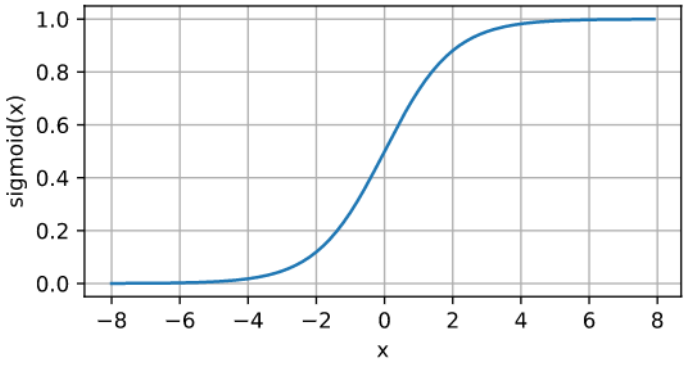


Figure 19: Función de activación Sigmoide

La derivada de la función sigmoide está dada por la siguiente ecuación:

$$\frac{d}{dx}\sigma(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)\,(1-\sigma(x)) \tag{4}$$

Notese que cuando la entrada es 0, la derivada de la función sigmoidea alcanza un máximo de 0,25. A medida que la entrada diverge de 0 en cualquier dirección, la derivada se acerca a 0.

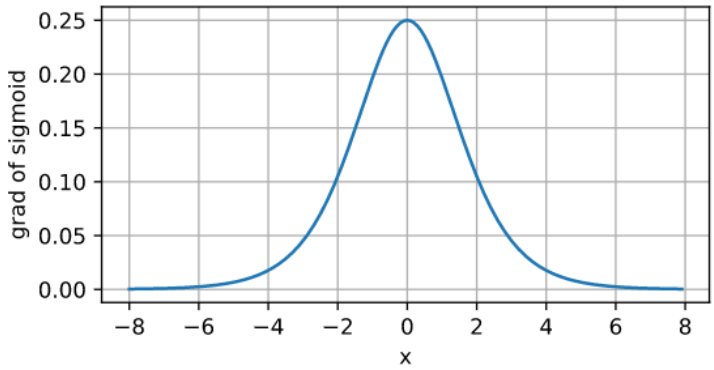


Figure 20: Derivada de la función de activación Sigmoide

3. Tanh (Tangentehiperbólica): Al igual que la función sigmoidea, la función tanh (tangente hiperbólica o hyperbolic tangent) también aplasta sus entradas, transformándolas en elementos en el intervalo entre -1 y 1:

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} \tag{5}$$

Hay que tener en cuenta que a medida que la entrada se acerca a 0, la función tanh se acerca a una transformación lineal. Aunque la forma de la función es similar a la de la función sigmoidea, la función tanh exhibe simetría puntual sobre el origen del sistema de coordenadas.

La derivada de la función tangentehiperbólica es:

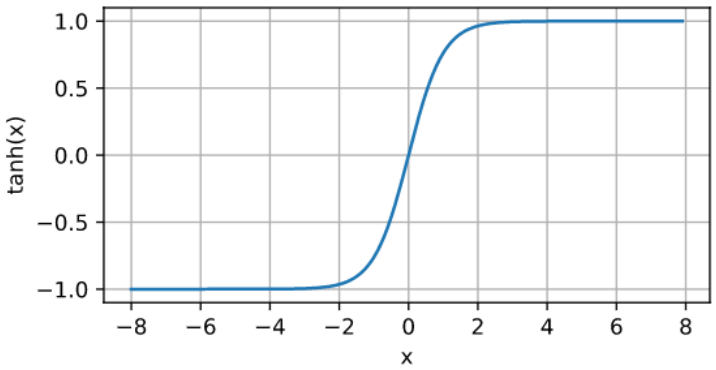


Figure 21: Función de activación Tanh

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \tag{6}$$

A medida que la entrada se acerca a 0, la derivada de la función **tanh** se acerca a un máximo de 1. Y como vimos con la función sigmoidea, a medida que la entrada se aleja de 0 en cualquier dirección, la derivada de la función tanh se acerca a 0.

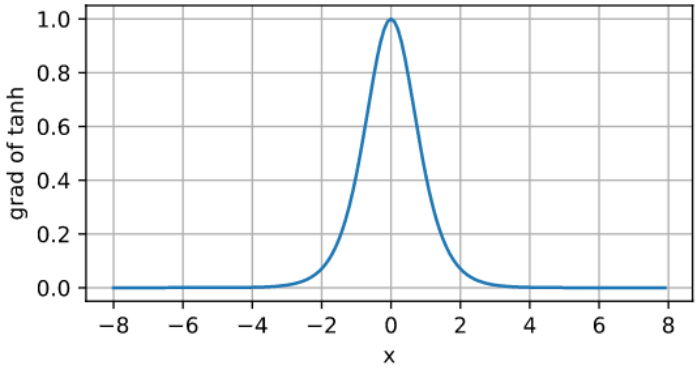


Figure 22: Derivada de la función de activación Tanh

Appendix B Neural Networks

Este apéndice tal vez sea de los más abarcativos, pero sin entrar en minuciosos detalles, dado que solamente busca brindarle una mejor intuición a quien no haya trabajado con redes neuronales.

Consideremos el ejemplo más básico, de una red neuronal de una sola capa, el clasificador Softmax. La regresión lineal es la herramienta que necesitamos cuando queremos responder preguntas como ¿cuánto? o ¿cuántos?. Si se desea predecir el precio al que se venderá una casa, o la cantidad de victorias que podría tener un equipo de futbol, o la cantidad de días que un paciente permanecerá hospitalizado antes de ser dado de alta, entonces probablemente un modelo de regresión sea lo óptimo.

Sin embargo, en la práctica, a veces estamos más interesados en la clasificación: en otras palabras, nos preguntamos ¿cuál? o ¿cuáles? Por ejemplo:

- ¿Este correo electrónico pertenece a la carpeta de correo no deseado o a la bandeja de entrada?
- ¿Es más probable que este cliente se registre o no en un servicio de suscripción?
- ¿Esta imagen representa un burro, un perro, un gato o un gallo?
- ¿Qué película es más probable que el usuario vea a continuación?

Coloquialmente, los profesionales del Deep Learning sobrecargan la clasificación de palabras para describir dos problemas sutilmente diferentes:

1. Aquellos en los que solo nos interesan las asignaciones de ejemplos a categorías (clases);
2. Aquellos en los que deseamos realizar asignaciones suaves, es decir, para evaluar la probabilidad de que se aplique cada categoría.

Comencemos con un simple problema de clasificación de imágenes. Aquí, cada entrada consta de una imagen en escala de grises de 2×2. Podemos representar cada cantidad de pixeles con un solo escalar, lo que nos da cuatro características x_1, x_2, x_3, x_4. Además, supongamos que cada imagen pertenece a una de las categorías "gato", "pollo" y "perro".

A continuación, tenemos que elegir cómo representar las etiquetas. Tenemos dos opciones obvias. Quizás el impulso más natural sería elegir $y \in \{1, 2, 3\}$ donde los números enteros representan $\{\text{perro}, \text{gato}, \text{pollo}\}$ respectivamente. Esta es una excelente manera de almacenar dicha información en una computadora. Si las categorías tuvieran algún orden natural entre ellas, digamos que intentamos predecir $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, entonces podría incluso tener sentido considerar este problema como una regresión y mantener las etiquetas en este formato.

Pero los problemas generales de clasificación no vienen con ordenaciones naturales entre las clases. Afortunadamente, los estadísticos inventaron hace mucho tiempo una forma sencilla de representar datos categóricos: la codificación one-hot (one-hot encoding). Una codificación one-hot es un vector con tantos componentes como categorías tengamos. El

componente correspondiente a la categoría de la instancia particular se establece en 1 y todos los demás componentes se establecen en 0. En nuestro caso, una etiqueta y sería un vector tridimensional, con $(1, 0, 0)$ correspondiente a "gato", $(0, 1, 0)$ a "pollo" y $(0, 0, 1)$ a "perro":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\} \tag{7}$$

Para estimar las probabilidades condicionales asociadas con todas las clases posibles, necesitamos un modelo con múltiples salidas, una por cada clase. Para abordar la clasificación con modelos lineales, necesitaremos tantas funciones como salidas tengamos. Cada salida corresponderá a su propia función afín. En nuestro caso, dado que tenemos 4 características y 3 posibles categorías de salida, necesitaremos 12 escalares para representar los ponderadores (w con subíndices) y 3 escalares para representar los sesgos (b con subíndices). Calculamos estos tres computos, o_1, o_2 y o_3, para cada entrada:

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \tag{8}$$
$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \tag{9}$$
$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \tag{10}$$

Podemos representar este cálculo con el diagrama de la red neuronal que se muestra en la siguiente Figura. Al igual que en la regresión lineal, la regresión softmax también es una red neuronal de una sola capa. Y dado que el cálculo de cada salida, o_1, o_2 y o_3, depende de todas las entradas, x_1, x_2, x_3, x_4, la capa de salida de la regresión softmax también se puede describir como una capa completamente conectada.

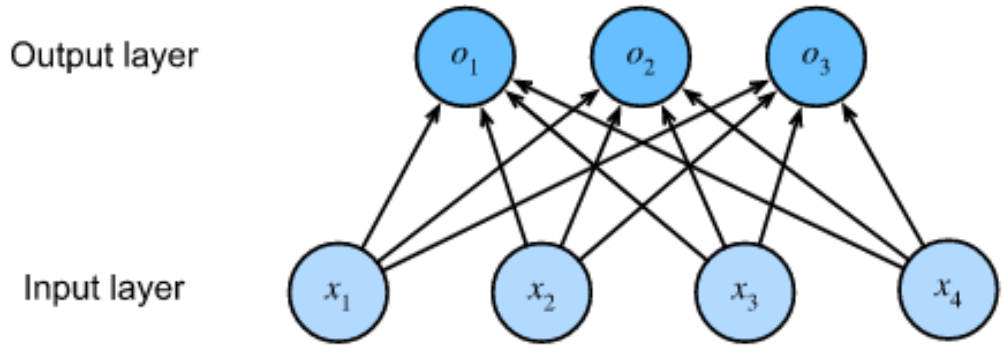


Figure 23: La regresión Softmax es una Red Neuronal "Single-Layer"

Para expresar el modelo de manera más compacta, podemos usar la notación algebraica. En forma vectorial, llegamos a $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, una forma más adecuada tanto para matemática, como para escribir código. Hay que tener en cuenta que hemos reunido todos nuestros ponderadores en una matriz de 3×4 y que para las características de un ejemplo de datos dado $\mathbf{x}$, nuestras salidas están dadas por un producto matriz-vector de nuestras ponderaciones por nuestras características de entrada más nuestros sesgos $\mathbf{b}$.

Ya introducida la intuición sobre cómo llegamos de regresiones a una red neuronal simple, podemos ampliar la arquitectura un poco más. Tengamos presente la Figura anterior de la regresión Softmax. Este modelo mapeó nuestras entradas directamente a nuestras salidas a través de una única transformación afín, seguida de una operación softmax (que no mencionamos para no entrar en detalles). Si nuestras etiquetas estuvieran realmente relacionadas con nuestros datos de entrada mediante una transformación afín, este enfoque sería suficiente. Sin embargo, la linealidad en las transformaciones afines es un supuesto muy fuerte.

¿Por qué los modelos lineales pueden dar malos resultados? Por ejemplo, la linealidad implica la suposición más débil de monotonicidad: que cualquier aumento en nuestra característica siempre debe causar un aumento en la salida de nuestro modelo (si el ponderador correspondiente es positivo), o siempre debe causar una disminución en la salida de nuestro modelo (si el ponderador correspondiente es negativo). A veces eso tiene sentido. Por ejemplo, si estuviéramos tratando de predecir si una persona pagará un préstamo, podríamos imaginar razonablemente que manteniendo todo lo demás igual (ceteris paribus), un solicitante con un ingreso más alto siempre tendría más probabilidades de pagar que uno con un ingreso más bajo. Si bien es monótona, esta relación probablemente no se asocia linealmente con la probabilidad de reembolso. Un aumento en los ingresos de 0 a 50 mil probablemente corresponda a un aumento mayor en la probabilidad de reembolso que un aumento de 1 millón a 1.05 millones. Una forma de manejar esto podría ser preprocesar nuestros datos de manera que la linealidad se vuelva más plausible, digamos, usando el logaritmo natural de los ingresos como nuestra característica.

Hay que tener en cuenta que fácilmente podemos encontrar ejemplos que violen la monotonicidad. Digamos, por ejemplo, que queremos predecir la probabilidad de muerte en función de la temperatura corporal. Para las personas con una temperatura corporal superior a 37 ° C, las temperaturas más altas indican un mayor riesgo. Sin embargo, para las personas con temperaturas corporales por debajo de 37 ° C, las temperaturas más altas indican un riesgo menor. También en este caso, podríamos resolver el problema con un **preprocesamiento** más inteligente. Es decir, podríamos usar la distancia desde 37 ° C como nuestra característica.

Pero, ¿qué pasa con la clasificación de imágenes de perros y gatos? ¿Debería aumentar la intensidad del píxel en la ubicación (13, 17) siempre aumentar (o siempre disminuir) la probabilidad de que la imagen muestre un perro? La confianza en un modelo lineal corresponde a la suposición implícita de que el único requisito para diferenciar gatos y perros es evaluar el brillo de los píxeles individuales. Este enfoque está condenado al fracaso en un mundo donde invertir una imagen preserva la categoría.

Y, sin embargo, a pesar del aparente absurdo de la linealidad aquí, en comparación con nuestros ejemplos anteriores, es menos obvio que podríamos abordar el problema con una simple solución de preprocesamiento. Esto se debe a que la importancia de cualquier píxel depende de manera compleja de su contexto (los valores de los píxeles circundantes). Si bien puede existir una representación de nuestros datos que tenga en cuenta las interacciones relevantes entre nuestras características, además de las cuales sería adecuado un modelo lineal, simplemente no sabemos cómo calcularlo a mano. Con las redes neuronales profundas,

utilizamos datos de observación para aprender conjuntamente tanto una representación a través de **capas ocultas** como un predictor lineal que actúa sobre esa representación.

Acá entra en discusión el concepto de capas ocultas (hidden layers, en inglés) en una red neuronal. Podemos superar estas limitaciones de los modelos lineales y manejar una clase más general de funciones incorporando una o más capas ocultas. La forma más sencilla de hacer esto es apilar muchas capas completamente conectadas una encima de la otra. Cada capa se alimenta de la capa superior, hasta que generamos salidas. Podemos pensar en las primeras capas $L-1$ como nuestra representación, y la capa final como nuestro predictor lineal. Esta arquitectura se denomina comúnmente *perceptrón multicapa*, a menudo abreviado como MLP. A continuación, representamos un MLP en forma de diagrama.

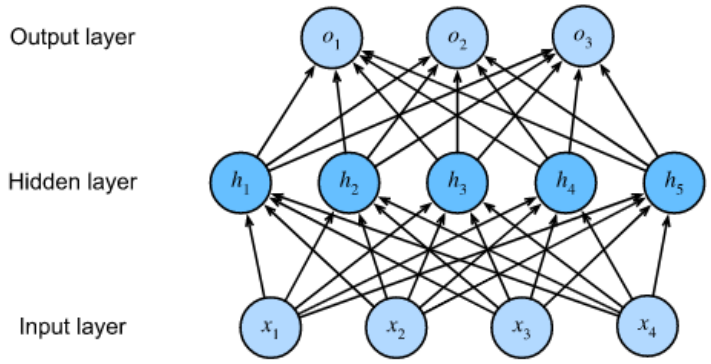


Figure 24: Un Perceptron Multicapa con capa oculta de 5 unidades

Este MLP tiene 4 entradas (x_1, x_2, x_3, x_4), 3 salidas (o_1, o_2, o_3) y su capa oculta contiene 5 unidades ocultas $(h_1, h_2, h_3, h_4, h_5)$. Dado que la capa de entrada no implica ningún cálculo, producir salidas con esta red requiere implementar los cálculos tanto para la capa oculta como para la de salida; por lo tanto, el número de capas en este MLP es técnicamente 2. Hay que tener en cuenta que estas capas están completamente conectadas. Cada entrada influye en todas las neuronas de la capa oculta y, a su vez, cada una de ellas influye en todas las neuronas de la capa de salida. Hay una interconexión completa. Sin embargo, el costo de parametrización de MLP con capas completamente conectadas puede ser prohibitivamente alto, lo que puede motivar una compensación entre el ahorro de parámetros y la efectividad del modelo incluso sin cambiar el tamaño de entrada o salida.

Por último, vamos a hacer un breve repaso de cómo pasar de un modelo lineal a uno no-lineal:

Como antes, por la matriz $\mathbf{X} \in \mathbb{R}^{n \times d}$, denotamos un minibatch de n ejemplos donde cada ejemplo tiene d entradas (características). Para un MLP de una capa oculta cuya capa oculta tiene h unidades ocultas, denotado por $\mathbf{H} \in \mathbb{R}^{n \times h}$ las salidas de la capa oculta, que son representaciones ocultas. En matemáticas o en código, $\mathbf{H}$ también se conoce como variable de capa oculta o variable oculta. Dado que las capas oculta y de salida están completamente conectadas, tenemos ponderadores de capa oculta $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ y sesgos $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$, y ponderadores de la capa de salida $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ y sesgos $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Formalmente, calculamos las salidas $\mathbf{O} \in \mathbb{R}^{n \times q}$ del MLP de una capa oculta de la siguiente

manera:

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \tag{11}$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \tag{12}$$

Nótese que después de agregar la capa oculta, nuestro modelo ahora requiere que realicemos un seguimiento y actualicemos conjuntos de parámetros adicionales. Entonces, ¿qué hemos ganado a cambio? En nuestro ejemplo de la clasificación, nada, aunque suene raro. La razón, sin embargo, es que las unidades ocultas h están dadas por una función afín de las entradas, y las salidas (previo a la softmax) son solo una función afín de las unidades ocultas. Ergo, en otras palabras, una función afín de una función afín es, en sí misma, otra función afín. Además, nuestro modelo lineal ya era capaz de trabajar bien. Esta equivalencia entre funciones podemos verla formalmente demostrando que para cualquier valor de los ponderadores (w), podemos simplemente colapsar la capa oculta, produciendo un modelo equivalente de una sola capa con parámetros $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ y $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ (el de la Figura de Softmax).

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b} \tag{13}$$

Para aprovechar el potencial de las arquitecturas multicapa, necesitamos un ingrediente clave más: una función de activación (no lineal) que se aplicará a cada unidad oculta después de la transformación afín. Las salidas de las funciones de activación (por ejemplo, la sigmoide, como hemos comentado en el Apéndice A) se denominan activaciones. En general, con las funciones de activación implementadas, ya no es posible colapsar nuestro MLP en un modelo lineal:

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \tag{14}$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \tag{15}$$

$$\tag{16}$$

Dado que cada fila en $\mathbf{X}$ corresponde a un ejemplo en el minibatch, con algún abuso de notación, definimos la no linealidad σ para aplicar a sus entradas en una forma de fila, es decir, un ejemplo a la vez. A menudo, las funciones de activación que aplicamos a las capas ocultas no son simplemente por filas, sino por elementos. Eso significa que después de calcular la porción lineal de la capa, podemos calcular cada activación sin mirar los valores tomados por las otras unidades ocultas. Esto es cierto para la mayoría de las funciones de activación.

Para construir MultiLayer Perceptrons más generales, podemos continuar apilando las capas ocultas, por ejemplo $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ y $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, una arriba de la otra, construyendo modelos cada vez más completos.

Todo lo comentado hasta el momento, junto con el apéndice A de funciones de activación, ha buscado servir de intuición (muy general) para entender un poco mejor la arquitectura

de una Red Neuronal Recurrente, que tendrá "hidden states" o estados ocultos, en vez de capas ocultas, y varios detalles diferentes más.

Appendix C Simple Recurrent Neural Networks

Consideremos una estructura de Redes Neuronales con Estados Ocultos (hidden states). Supongamos que tenemos un minibatch (lote) de datos de entrada, $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ en el período t. En otras palabras, para un minibatch de n ejemplos secuenciales, cada fila de $\mathbf{X}_t$ corresponde a un ejemplo en el período t de la secuencia. Luego, denotemos $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ como la variable oculta en el tiempo t. En nuestro caso, guardamos la variable oculta en el período anterior $\mathbf{H}_{t-1}$ e introducimos un ponderador nuevo $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ para describir cómo usar la variable oculta del período anterior en el período actual (t). Específicamente, el cálculo de la variable oculta en el período actual se determina por el input actual junto con la variable oculta del período anterior:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h) \tag{17}$$

De la relación entre las variables $\mathbf{H}_t$ y $\mathbf{H}_{t-1}$ de períodos adyacentes, sabemos que estas capturan y retienen la información histórica secuencial hasta el período actual t. Por este motivo es que se denomina a la variable oculta como estado oculto (hidden state). Como el estado oculto usa la misma definición del período anterior, pero en el actual, la computación de 17 es recurrente o recursiva. Por ese motivo, es que las redes neuronales basadas en estados ocultos y su computación recurrente se denominan Recurrent Neural Networks. Las capas que realizan el computo recursivo se denominan recurrent layers.

Hay muchas formas distintas de construir RNNs. RNNs con un estado oculto definido por 17 son muy comunes. Para el período t, el output de la capa de salida (output layer) se computa de la siguiente manera:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q \tag{18}$$

Los parámetros de la RNN incluyen los ponderadores $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, y el sesgo u ordenada de la capa oculta (hidden layer) $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, junto con los ponderadores $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ y el otro sesgo $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ de la output layer. Vale la pena mencionar que inclusive en períodos diferentes, RNNs usan siempre estos parámetros del modelo. Por consiguiente, el costo de parametrización de una RNN no crece con el número de períodos en el tiempo.

La Figura ilustra la lógica computacional de una RNN en tres períodos de tiempo adyacentes. En cualquier período t, la computación de un estado oculto se puede tratar como:

1. Concatenar el input $\mathbf{X}_t$ en el período actual t y el estado oculto $\mathbf{H}_{t-1}$ del período anterior $t-1$.
2. Alimentar el resultado de la concatenación en una capa conectada completamente (fully-connected) con la función de activación ϕ (recordar que la función de activación es algo a elegir por cada científico de datos). El output de dicha capa completamente conectada será el estado oculto $\mathbf{H}_t$ en el período actual t. En este caso los parámetro del modelo son la concatenación de $\mathbf{W}_{xh}$ y $\mathbf{W}_{hh}$, y el sesgo $\mathbf{b}_h$. El estado oculto de nuestro período

actual $\mathbf{H}_t$ será también alimentado en una capa completamente conectada de salida (fully connected output layer) para computar el resultado $\mathbf{O}_t$ del período actual t.

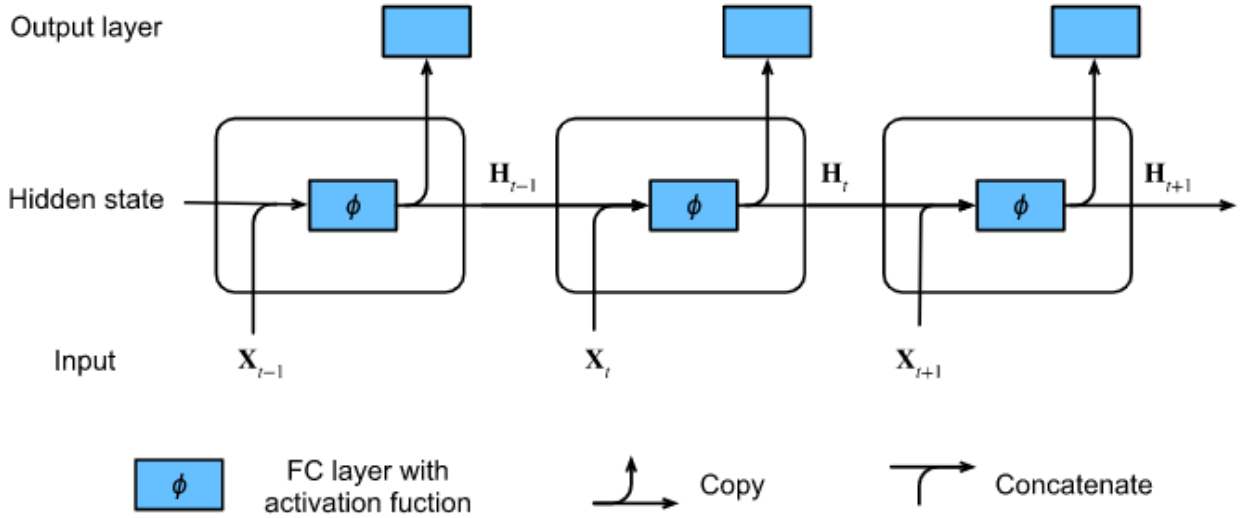


Figure 25: Lógica computacional de una estructura RNN

Nosotros mencionamos recién que el cálculo de $\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh}$ para el estado oculto es equivalente a una multiplicación de matrices de la concatenación de $\mathbf{X}_t$ y $\mathbf{H}_{t-1}$ y la concatenación de $\mathbf{W}_{xh}$ y $\mathbf{W}_{hh}$. Si definimos las matrices $\mathbf{X}$, $\mathbf{W}_{xh}$, $\mathbf{H}$ y $\mathbf{W}_{hh}$, con dimensiones para el ejemplo (3,1), (1,4), (3,4) y (4,4) respectivamente, veremos que se cumple que:

$$\underbrace{\underbrace{\underbrace{\mathbf{X}_t}_{(3,1)} \cdot \underbrace{\mathbf{W}_{xh}}_{(1,4)}}_{(3,4)} + \underbrace{\underbrace{\mathbf{H}_{t-1}}_{(3,4)} \cdot \underbrace{\mathbf{W}_{hh}}_{(4,4)}}_{(3,4)}}_{(3,4)}$$

Appendix D Gradient Descent

En este apéndice, vamos a introducir los conceptos básicos subyacentes al descenso del gradiente (Gradient Descent). Aunque rara vez se usa directamente en Deep Learning, ya que hay algoritmos de optimización más avanzados, comprender el descenso de gradiente es clave para comprender los algoritmos de descenso de gradiente estocástico (Stochastic Gradient Descent). Por ejemplo, el problema de optimización puede diferir debido a una tasa de aprendizaje (step size) demasiado grande. Este fenómeno ya se puede ver en el descenso del gradiente. Asimismo, el preacondicionamiento es una técnica común en el descenso de gradiente y se traslada a algoritmos más avanzados.

Sin más, consideremos el caso para una función de una sola variable:

Tomemos una función real, continua y diferenciable $f : \mathbb{R} \to \mathbb{R}$. Aplicando una expansión de Taylor obtendremos:

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2) \tag{19}$$

Eso significa, que una aproximación de primer orden $f(x + \epsilon)$ está dada por el valor de la función $f(x)$ y su primer derivada $f'(x)$ en x. No sería poco razonable asumir que para valores pequeños de ϵ en la dirección contraria al gradiente va a disminuir a f. Para mantener las cosas simples, tomemos un valor fijo de step size $\eta > 0$ y elijamos $\epsilon = -\eta f'(x)$. Introduciendo esto en la expansión de Taylor anterior, obtendremos:

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)) \tag{20}$$

Si la derivada $f'(x) \neq 0$ no se desvanece (recordemos el problema de vanishing gradients) vamos a hacer un progreso real, ya que $\eta f'^2(x) > 0$. Asimismo, podemos elegir siempre un valor de η lo suficientemente pequeño para que los términos de orden más grande sean irrelevantes. Así, llegamos a que:

$$f(x - \eta f'(x)) \lessapprox f(x) \tag{21}$$

Eso significa, que si usamos:

$$x \leftarrow x - \eta f'(x) \tag{22}$$

Cambiando la notación:

$$x := x - \eta \nabla f(x) \tag{23}$$

para iterar x, el valor de la función $f(x)$ va a disminuir. En consecuencia, en Gradient Descent primero elegimos un valor inicial de x y una constante $\eta > 0$ y luego los usamos para iterar continuamente sobre x hasta llegar a la condición de corte (stop) o nivel de tolerancia. Por ejemplo, puede ser cuando la magnitud del gradiente $|f'(x)|$ sea suficientemente pequeña o si el número de iteraciones ya llegó cierto valor (esas condiciones son determinadas por el científico de datos).

Por otro lado, en el caso multivariable, nótese que de la ecuación (23) tendríamos el término $\nabla f(x)$ como:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top$$

donde cada derivada parcial $\partial f(\mathbf{x})/\partial x_i$ en el gradiente indica la tasa de cambio de f en $\mathbf{x}$ con respecto al input x_i. Visto gráficamente:

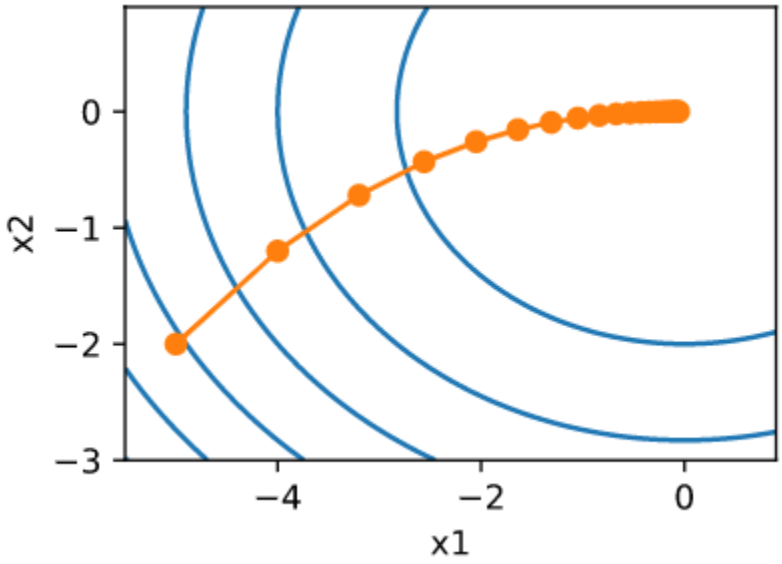


Figure 26: Ejemplo de un Gradiente Descendiente en dos variables

Appendix E Backpropagation

Comenzamos con un modelo simplificado de cómo funciona un RNN. Este modelo ignora los detalles sobre los estados ocultos y cómo se actualizan. La notación matemática tampoco distingue explícitamente escalares, vectores y matrices como solía hacerlo. Estos detalles son irrelevantes para el presente análisis.

En este modelo simplificado, denotamos h_t como el estado oculto, x_t como la entrada y o_t como la salida en el período t. Recordemos que la entrada x_t y el estado oculto pueden concatenarse para multiplicarse por una variable de ponderación (weight) en la capa oculta. Por lo tanto, usamos w_h y w_o para indicar los ponderadores de la capa oculta y la capa de salida, respectivamente. Como resultado, los estados ocultos y las salidas en cada paso de tiempo se pueden explicar como

$$h_t = f(x_t, h_{t-1}, w_h) \tag{24}$$

$$o_t = g(h_t, w_o) \tag{25}$$

donde f y g son transformaciones de la capa oculta y la capa de output, respectivamente. Por consiguiente, tendremos una cadena de valores $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ que dependen cada uno del otro vía computaciones recurrentes. La propagación hacia adelante (forward propagation) es bastante directa, lo único que necesitamos es un loop/bucle a través de (x_t, h_t, o_t) un paso por período de tiempo. La discrepancia entre el output o_t y el resultado deseado y_t se evalúa entonces por una función objetivo a través de todos los períodos T.

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t) \tag{26}$$

Para la propagación hacia atrás (backpropagation), el proceso es un poco más complicado que forward, especialmente cuando computamos los gradientes respecto a los parámetros w_h de la función objetivo L. Para ser especificos, por regla de cadena:

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial w_h} \\ &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial g(h_t, w_o)}{\partial h_t} \frac{\partial h_t}{\partial w_h}\end{aligned} \tag{27}$$

El primero y segundo factor del producto en 27 son fáciles de computar, pero el tercer factor $\partial h_t/\partial w_h$ es donde se dificulta, ya que necesitamos computar de manera recurrente el efecto del parámetro w_h en h_t. De acuerdo a la computación recurrente en 25, h_t depende de ambos h_{t-1} y w_h donde la computación de h_{t-1} también depende de w_h. Por eso, usando la regla de la cadena se llega a:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h} \tag{28}$$

Para derivar dicho gradiente, debemos asumir que las tres secuencias $\{a_t\}, \{b_t\}, \{c_t\}$ satisfacen $a_0 = 0$ y $a_t = b_t + c_t a_{t-1} \forall t \geq 1$, lo que resulta fácil mostrar que:

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i \tag{29}$$

Al sustituir a_t, b_t y c_t de acuerdo a las definiciones:

$$a_t = \frac{\partial h_t}{\partial w_h}, \tag{30}$$

$$b_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h}, \tag{31}$$

$$c_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}, \tag{32}$$

La computación del gradiente en 28 satisface que $a_t = b_t + c_t a_{t-1}$. Por eso, a través de 29 podemos reemplazar la computación recurrente de 28 con:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h} \tag{33}$$

Mientras usemos la regla de la cadena para computar $\partial h_t / \partial w_h$ recursivamente, esta cadena puede volverse muy larga siempre que t sea grande. Este problema se puede resolver (o no):

- Computación completa: Obviamente, podemos calcular la suma completa en (33). Sin embargo, esto es muy lento y los gradientes pueden explotar (blow up), ya que los cambios sutiles en las condiciones iniciales pueden potencialmente afectar mucho el resultado final. Es decir, podríamos ver cosas similares al llamado "efecto mariposa", donde los cambios mínimos en las condiciones iniciales conducen a cambios desproporcionados en el resultado. En realidad, esto es bastante indeseable en términos del modelo que queremos estimar. Después de todo, buscamos estimadores robustos que generalicen bien. Por tanto, esta estrategia casi nunca se utiliza en la práctica.
- Truncando períodos de tiempo: Alternativamente, podemos truncar la suma en (33) después de τ períodos. Esto conduce a una aproximación del gradiente verdadero, simplemente terminando la suma en $\partial h_{t-\tau} / \partial w_h$. En la práctica, esto funciona bastante bien. Es lo que comúnmente se conoce como retropropulsión truncada a través del tiempo (truncated backpropagation through time) [Jaeger, 2002]. Una de las consecuencias de esto es que el modelo se centra principalmente en la influencia a corto plazo más que en las consecuencias a largo plazo. Esto es realmente deseable, ya que sesga la estimación hacia modelos más simples y estables.
- Truncando al azar (Randomized Truncation): Por último, podemos reemplazar $\partial h_t / \partial w_h$ por una variable random que es correcta en expectativas, pero trunca la secuencia. Esto se logra usando una secuencia de ξ_t con $0 \leq \pi_t \leq 1$ donde $P(\xi_t = 0) = 1 - \pi_t$

y $P(\xi_t = \pi_t^{-1}) = \pi_t$, por lo que $E[\xi_t] = 1$. Usamos esto para reemplazar el gradiente $\partial h_t/\partial w_h$ en (33) con:

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h} \tag{34}$$

Surge de la definición de ξ_t que $E[\xi_t] = \partial h_t/\partial w_h$. Siempre que $\xi_t = 0$, la computación recurrente termina en ese período t. Esto conduce a una suma ponderada de secuencias de magnitudes variantes donde las secuencias largas son raras, pero apropiadamente sobreponderadas. Esta idea fue propuesta por Tallec y Ollivier [Tallec & Ollivier, 2017].

Appendix F Long Short-Term Memory

El desafío de preservar la información a largo plazo ha existido durante mucho tiempo. Uno de los primeros enfoques para abordar esto fue la memoria larga a corto plazo (LSTM) [Hochreiter & Schmidhuber, 1997]. Comparte muchas de las propiedades del GRU (Gated Recurrent Unity). Curiosamente, las LSTM tienen un diseño un poco más complejo que las GRU, pero son anteriores a las GRU en casi dos décadas.

Podría decirse que el diseño de la LSTM está inspirado en las puertas lógicas (logic gates) de una computadora. LSTM introduce una celda de memoria (o celda, abreviando) que tiene la misma forma que el estado oculto (hidden state), diseñada para registrar información adicional. Para controlar la celda de memoria, necesitamos una serie de puertas.

- Se necesita una puerta para leer las entradas a la celda. Nos referiremos a esta como la output gate.
- Una segunda puerta se necesita para decidir cuando incorporar datos a la celda. Esta será la puerta de entrada (input gate).
- Por último, necesitamos también un mecanismo para reestablecer el contenido de la celda, que será hecho por una puerta de olvido (forget gate).

La motivación para semejante diseño es la misma que de las GRU, es decir, poder decidir cuando recordar y cuando ignorar las entradas en el estado oculto a través de un mecanismo dedicado. Veamos como funciona esto en la práctica:

Al igual que en las GRU, los datos que ingresan a las puertas LSTM son la entrada en el período de tiempo actual t y el estado oculto del período anterior $\mathbf{H}_{t-1}$, como se ilustra en la siguiente Figura. Estos son procesados por tres capas completamente conectadas con una función de activación sigmoidea (σ) para calcular los valores de las puertas de entrada, olvido y salida. Como resultado, los valores de las tres puertas están en el rango de (0,1).

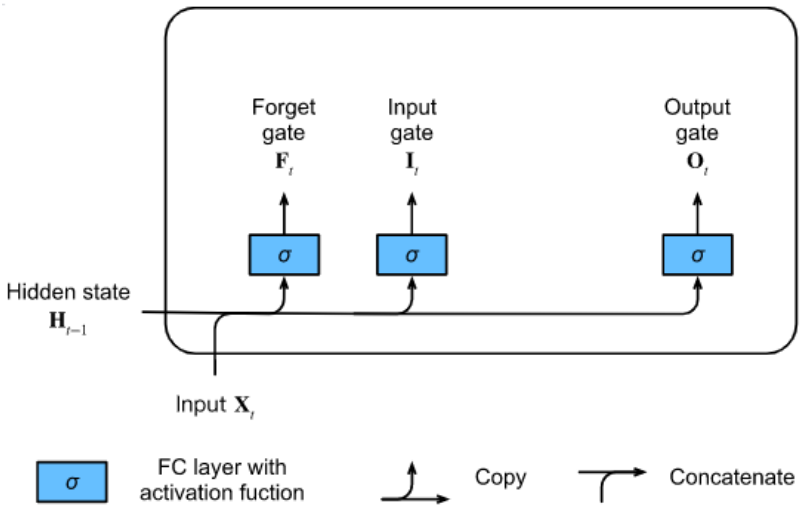


Figure 27: Computación de la input, forget y output gates en LSTM

Matemáticamente, supongamos que hay h unidades ocultas, el batch size es n, y el número de inputs es d. Por consiguiente, el input será $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ y el estado oculto del período previo es $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondientemente, las puertas en el período t se definen cómo a continuación: la puerta de entrada (input gate) es $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, la puerta de olvido (forget gate) es $\mathbf{F}_t \in \mathbb{R}^{n\times h}$ y la puerta de salida (output gate) es $\mathbf{O}_t \in \mathbb{R}^{n\times h}$. Se calculan de la siguiente forma:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i), \tag{35}$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f), \tag{36}$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o), \tag{37}$$

donde $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ y $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ son los ponderadores de los parámetros y $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ son los sesgos.

Luego, diseñamos la celda de memoria. Dado que todavía no hemos especificado la acción de las distintas puertas, primero introducimos la celda de memoria candidata $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$. Su computación es similar a la de las tres puertas, descriptas anteriormente, pero con la salvedad de que usará una función de activación **tanh** con un rango de valores (-1,1). Esto conlleva a la siguiente ecuación en el período t:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c), \tag{38}$$

donde $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ y $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ son los ponderadores de los parámetros y $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ el sesgo. Una ilustración de la celda de memoria candidata es la siguiente Figura.

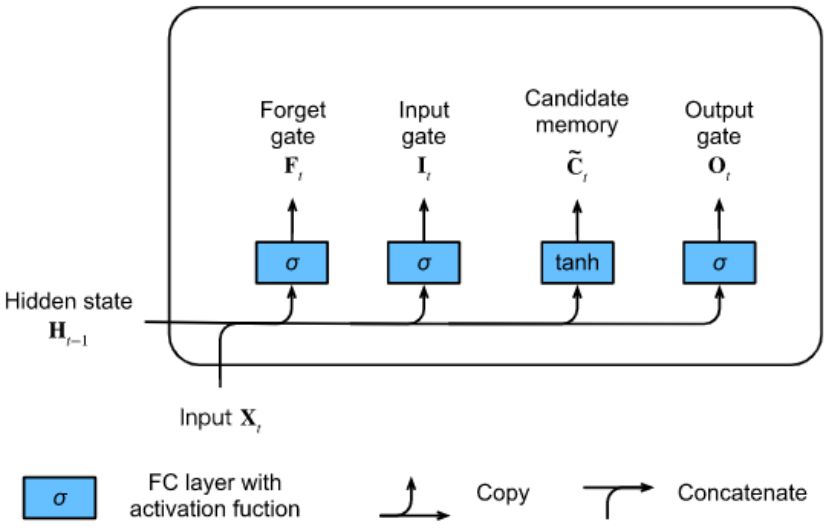


Figure 28: Computación de Memory Cell candidata

En GRUs, tenemos un mecanismo para controlar la entrada y el olvido (forgetting o skipping). Similarmente, en las LSTM tenemos dos puertas dedicadas para tales propósitos: la puerta de entrada $\mathbf{I}_t$ que controla cuánto tomamos en cuenta de los nuevos datos a través de $\tilde{\mathbf{C}}_t$, y la puerta de olvido $\mathbf{F}_t$ que aborda cuánto contenido de la celda de memoria antigua $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ retendremos. Usando el producto Hadamard (recordar que es el operador elementwise, denotado como $\odot$), llegamos a la siguiente ecuación de actualización:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \tag{39}$$

Si la puerta de olvido es siempre aproximadamente 1 y la puerta de entrada es siempre aproximadamente 0, las celdas de memoria pasadas $\mathbf{C}_{t-1}$ se guardarán con el tiempo y pasarán período de tiempo actual t. Este diseño se introduce para aliviar el problema del gradiente que desvanece (vanishing gradient) y para capturar mejor las dependencias de largo alcance dentro de las secuencias. Así, llegamos al diagrama de flujo en la Figura a continuación:

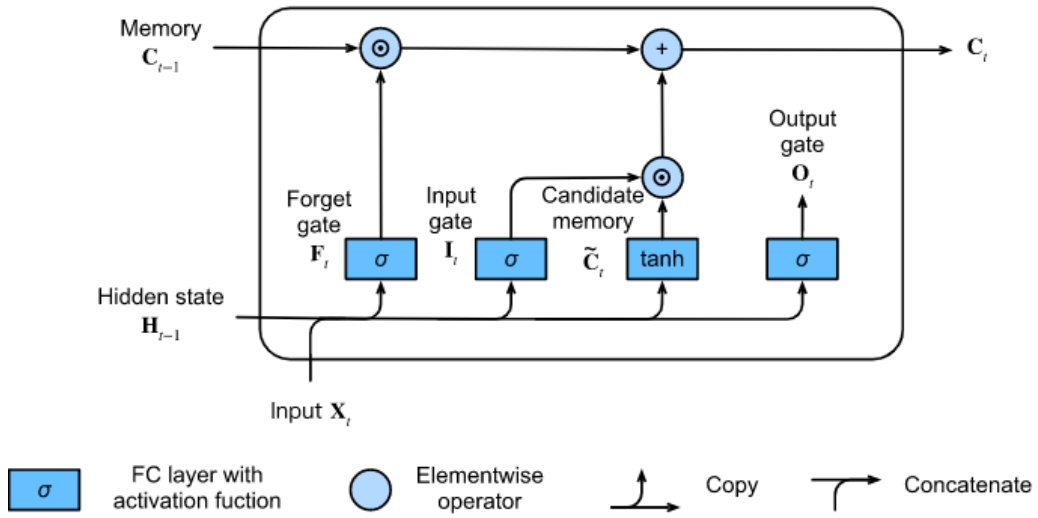


Figure 29: Computación de Memory Cell final

Finalmente, debemos definir cómo computar el estado oculto $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. Esta parte es cuando la puerta de salida entra en juego. En una LSTM es simplemente una versión cerrada de la función **tanh** de la celda de memoria. Esto asegura que los valores de $\mathbf{H}_t$ estén siempre en el intervalo (-1,1).

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t) \tag{40}$$

Siempre que la puerta de salida se aproxima a 1, efectivamente pasamos toda la información de la memoria al predictor, mientras que para una puerta de salida cercana a 0, significa que retenemos toda la información solo dentro de la celda de memoria y no realizamos ningún procesamiento adicional.

El flujo de datos se ve cómo en la Figura 21:

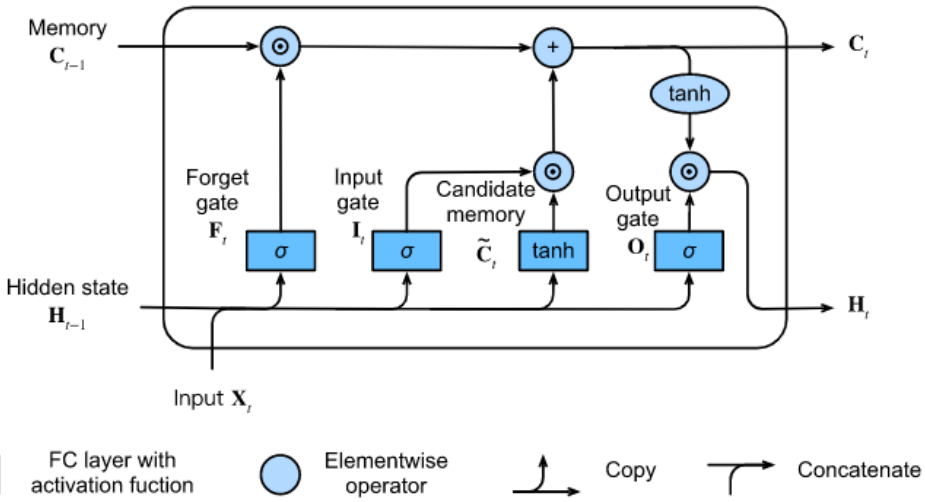


Figure 30: Computación del Hidden State

Appendix G Gated Recurrent Unit

Por otro lado, resta profundizar en las Gated Recurrent Units. En el apéndice C, de back-propagation, vimos cómo se calculan los gradientes en RNN. En particular, encontramos que los productos largos de matrices pueden conducir a gradientes que se desvanecen o explotan. Pensemos brevemente en lo que significan estas anomalías de gradiente en la práctica:

- Podríamos encontrarnos con una situación en la que una observación temprana es muy importante para predecir todas las observaciones futuras. En este caso, la influencia del primera observación es vital. Nos gustaría tener algunos mecanismos para almacenar información importante temprana en una celda de memoria. Sin tal mecanismo, tendremos que asignar un gradiente muy grande a esta observación, ya que afecta a todas las observaciones posteriores.
- Podríamos encontrarnos con situaciones en las que algunas fichas no conllevan una observación pertinente. Por ejemplo, al analizar una página web, puede haber un código HTML auxiliar que sea irrelevante para evaluar el sentimiento transmitido en la página. Nos gustaría tener algún mecanismo para omitir tales observaciones en la representación del estado latente.
- Podríamos encontrarnos con situaciones en las que existe una ruptura lógica entre las partes de una secuencia. Por ejemplo, puede haber una transición entre capítulos en un libro, o una transición entre un mercado bajista y un mercado alcista de valores. En este caso, sería bueno tener un medio para restablecer nuestra representación de estado interno.

Se han propuesto varios métodos para abordar este problema. Uno de los primeros es la memoria a corto plazo [Hochreiter & Schmidhuber, 1997]. La unidad recurrente cerrada (GRU) [Cho et al., 2014a] es una variante un poco más optimizada que a menudo ofrece un rendimiento comparable y es significativamente más rápida de calcular [Chung et al., 2014]. Por su sencillez, comencemos por el GRU.

La distinción clave entre las RNN simples y las GRU es que estas últimas admiten la activación del estado oculto. Esto significa que tenemos mecanismos dedicados para cuando se debe actualizar un estado oculto y también cuando se debe restablecer. Estos mecanismos se aprenden y abordan las preocupaciones enumeradas anteriormente. Por ejemplo, si la primer observación es de gran importancia, aprenderemos a no actualizar el estado oculto después de la primera observación. Asimismo, aprenderemos a omitir observaciones temporales irrelevantes. Por último, aprenderemos a restablecer el estado latente siempre que sea necesario.

Lo primero que debemos introducir es la puerta de reinicio (reset gate) y la puerta de actualización (update gate). Las diseñamos para que sean vectores con entradas en (0,1) de modo que podamos realizar combinaciones convexas. Por ejemplo, una puerta de reinicio nos permitiría controlar cuánto del estado anterior podríamos querer recordar. Del mismo modo, una puerta de actualización nos permitiría controlar qué parte del nuevo estado es solo una copia del estado anterior.

Comenzamos diseñando estas puertas. La Figura ilustra las entradas para las puertas de reinicio y actualización en un GRU, dada la entrada del período actual t y el estado oculto del paso de tiempo anterior $\mathbf{H}_{t-1}$. Las salidas de dos puertas están dadas por dos capas completamente conectadas con una función de activación sigmoidea (σ).

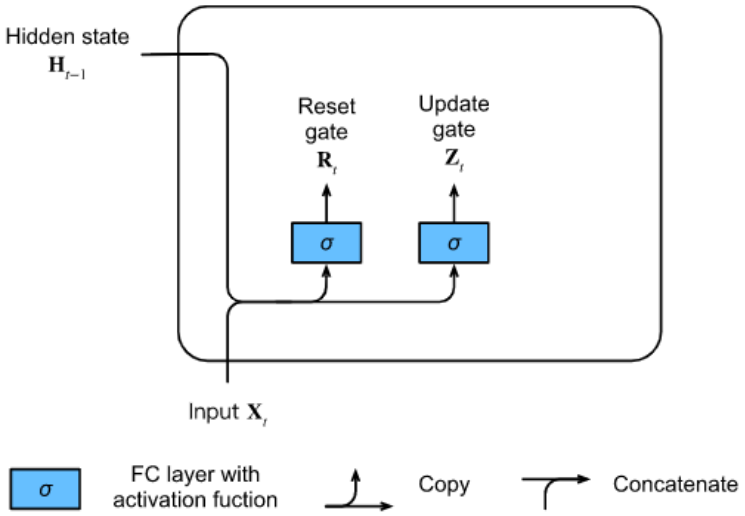


Figure 31: Computación de las Reset y Update gate en una GRU

Matemáticamente, para un período de tiempo t, supongamos que el input es un minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (número de ejemplos: n, número de entradas: d) y que el estado oculto del período anterior es $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ (número de unidades ocultas: h). Entonces, la puerta de reset o reinicio es $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ y la de update o actualización es $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ y se computan como a continuación:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \tag{41}$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z), \tag{42}$$

donde $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ y $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ son los ponderadores y $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ los sesgos. Nótese que una operación de broadcasting es empleada durante esta suma. Usamos una función sigmoidea (σ) para transformar los valores en un intervalo (0,1).

Luego, debemos integrar la puerta de reinicio $\mathbf{R}_t$ con el mecanismo de actualización de estado latente (como en la ecuación 17), que conlleva al siguiente candidato de estado oculto $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ en el período t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h) \tag{43}$$

donde $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ y $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ son los ponderadores, y $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ es el sesgo. Recordar que el símbolo $\odot$ es el operador del producto Hadamard (elementwise). Acá usamos una nolinearidad de la forma de **tanh** para asegurarnos que los valores en el estado oculto candidato permanecen en el intervalo (-1,1).

El resultado es un candidato dado que aun necesitamos incorporar la acción de la puerta de actualización (update gate). Comparado con la ecuación (17), ahora la influencia de los previos estados pueden ser reducidas con la multiplicación Hadamard de $\mathbf{R}_t$ y $\mathbf{H}_{t-1}$ en la ecuación (43). Siempre que las entradas en la puerta de reinicio $\mathbf{R}_t$ sean cercanas a 1, vamos a recuperar una RNN simple como en la ecuación (17). Visto en una figura:

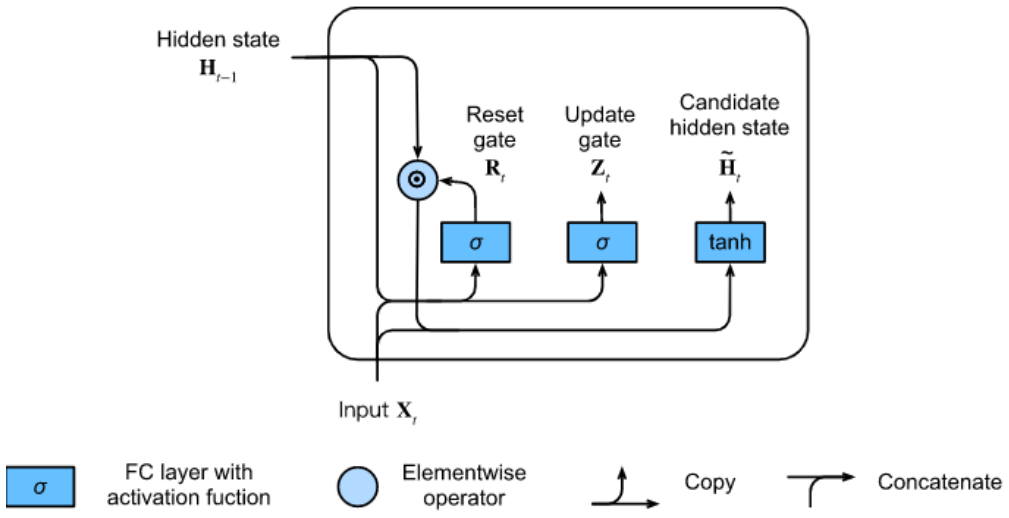


Figure 32: Computación del Hidden State candidato en una GRU

Finalmente, debemos incorporar el efecto de la puerta de actualización $\mathbf{Z}_t$. Este determina hasta qué punto el nuevo estado oculto $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ es solamente el estado oculto anterior $\mathbf{H}_{t-1}$ y en qué medida se utiliza e lnuevo estado oculto candidato $\tilde{\mathbf{H}}_t$. La puerta de actualización

$\mathbf{Z}_t$ puede ser usada con este propósito, simplemente tomando combinaciones convexas de elementos entre ambos $\mathbf{H}_{t-1}$ y $\tilde{\mathbf{H}}_t$. Esto lleva a la última ecuación de actualización para la GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t \tag{44}$$

Siempre que la puerta de actualización $\mathbf{Z}_t$ esté cerca de 1, simplemente retenemos el estado anterior. En este caso, la información de $\mathbf{X}_t$ se ignora, saltándose efectivamente el período de tiempo t en la cadena de dependencia. Por el contrario, siempre que $\mathbf{Z}_t$ está cerca de 0, el nuevo estado latente $\mathbf{H}_t$ se aproxima al estado latente candidato $\tilde{\mathbf{H}}_t$. Estos diseños pueden ayudarnos a hacer frente al problema del gradiente que desvanece (vanishing gradient) en las RNN y mejorar la captura de dependencias para secuencias grandes en el tiempo $t \to T$. Por ejemplo, si la puerta de actualización ha estado cerca de 1 para todos los períodos de tiempo de una subsecuencia completa, el antiguo estado oculto en el paso de tiempo de su comienzo se retendrá fácilmente y pasará a su final, independientemente de la longitud de la subsecuencia.

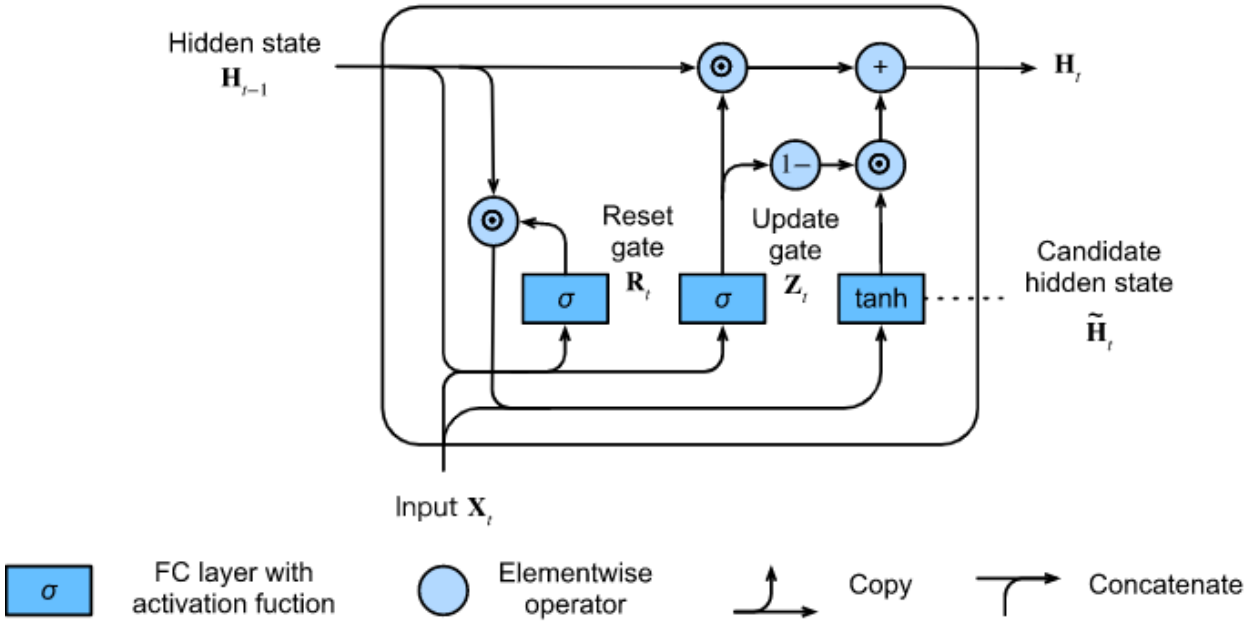


Figure 33: Computación del Hidden State en una GRU

En conclusión, GRUs tienen las siguientes ventajas distintivas:

- Las puertas de reinicio (reset gates) ayudan a capturar dependencias de corto plazo en las secuencias.
- Las puertas de actualización (update gates) ayudan a capturar dependencias de largo plazo en las secuencias.

Bibliography

[1] F. Lazzeri. Machine Learning for Time Series Forecasting with Python. Wiley, 2020.

[2] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into Deep Learning. 2020. `https://d2l.ai`.

[3] Andrew Ng. Machine Learning Yearning. 2018. `https://www.deeplearning.ai/programs/`.

[4] J. Brownlee. Introduction to Time Series Forecasting With Python: How to Prepare Data and Develop Models to Predict the Future. Machine Learning Mastery, 2017.

[5] R.J. Hyndman and G. Athanasopoulos. Forecasting: Principles and Practice. OTexts, 2018.

[6] Min Chulkim. Data science tech stack, 2020.

[7] Rob J Hyndman. Errors on percentage errors, 2014.