Aprendizaje Automático para la Selección de Grupos de Control en Evaluación de Impacto

por

Benjamín Bas Peralta

Presentado ante la **FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN** como parte de los requerimientos para la obtención del grado de Licenciado en Ciencias de la Computación de la **UNIVERSIDAD NACIONAL DE CÓRDOBA**

Marzo, 2025

Directores: Dr. Martín Domínguez

Mgter.David Giuliodori

Resumen: (...)

Palabras Clave: Evaluación de Impacto, Grupo de Control, Aprendizaje Automático, Redes Neuronales, Series de Tiempo.

Abstract: (...)

Keywords: Impact Assesment, Control Group, Machine Learning, Neural Networks, Time Series.

Agradecimientos

Índice general

Capítulo 1

Introducción

Capítulo 2

Marco Teórico

En este capítulo, presentamos los conceptos teóricos necesarios para entender el contexto del problema y cuál es la solución que planteamos.

Primeramente, explicamos la Evaluación de Impacto, que es el campo de aplicación sobre el que vamos a trabajar. Describimos qué es una Evaluación de Impacto, en qué consiste, cuáles son los problemas involucrados, y las distintas técnicas que se utilizan para llevarla a cabo. Al final de esta sección, presentamos el método de *Propensity Score Matching*, que es el que tomaremos como parámetro para evaluar nuestra solución propuesta.

Luego, nos abordamos al campo de la Inteligencia Artificial, enfocándonos particularmente en el campo de los algoritmos de Aprendizaje Automático. Explicamos cómo funcionan estos y cuáles son los elementos presentes en ellos haciendo especial enfásis en los algoritmos de Aprendizaje Supervisado, dentro de los cuales se enmarca nuestra propuesta.

A continuación, hablamos sobre un *modelo* particular dentro del Aprendizaje Automático que ha cobrado particular importancia en los últimos años: las Redes Neuronales Artificiales. Partimos explicando las de tipo *Feedforward* para Luego introducir otras más específicas como son las Convolucionales y las Recurrentes, de las cuales haremos uso en nuestros experimentos.

Por último, y para combinar el método de solución propuesto junto con el área de aplicación, hablamos sobre las series de tiempo. Mencionamos qué son y cuáles son los parámetros involucrados en ellas, y repasamos los últimos trabajos llevados a cabo que utilizan Inteligencia Artificial para resolver el problema de clasificación de series de tiempo.

2.1. Evaluación de Impacto

2.1.1. ¿Qué es una Evaluación de Impacto?

Los programas y las políticas de desarrollo suelen estar diseñados para cambiar resultados, como aumentar los ingresos, mejorar el aprendizaje o reducir las enfermedades [4]. Saber si estos cambios se logran o no es una pregunta crucial para las políticas públicas [4], y el método para responderla es lo que se conoce como **evaluación de impacto**.

Una evaluación de impacto mide los cambios en el bienestar de los individuos que se pueden atribuir a un proyecto, un programa o una política específicos [4]. Su sello distintivo radica en que pueden proporcionar **evidencia** robusta y creíble sobre si un programa concreto ha alcanzado o está alcanzando sus objetivos [4].

Estas evaluaciones ponen un fuerte énfasis en los resultados y buscan responder una pregunta específica de causa y efecto: ¿cuál es el impacto (o efecto causal) de un programa? Más precisamente, intentan identificar y cuantificar los cambios directamente atribuibles al tratamiento [4] sobre un conjunto de **variables de resultado** o **de interés** en un conjunto de individuos [1]. Estas variables son aquellas sobre las cuales se espera que el programa tenga un efecto en los beneficiarios [1]. Podrían ser por ejemplo la estatura y peso de los individuos, la cantidad de empleados o de ventas en una empresa, o indicadores de salud de un paciente.

Por lo tanto, el objetivo final de la evaluación de impacto consiste en establecer lo que se conoce como **efecto del tratamiento**, que es la diferencia entre la variable de resultado del individuo participante en el programa en presencia del programa, y la variable de resultado de ese individuo en ausencia del programa [1]. Sin embargo, es evidente que la respuesta a qué habría pasado con los beneficiarios si no hubieran recibido el tratamiento se refiere a una situación que no es observable. Este resultado hipotético se denomina **contrafactual** y es lo que se debe estimar en cualquier evaluación de impacto.

Algunas de las principales razones por las que se debería promover el uso de estas evaluaciones como herramientas de gestión provienen del hecho que permiten mejorar la rendición de cuentas, la inversión de recursos públicos o la efectividad de una política, obtener financiamiento, como así también probar modalidades de programas alternativos o innovaciones de diseño [4] y revelar la realidad de muchas políticas públicas para de esta forma contribuir a la fiscalización mediática [1].

Un ejemplo claro de por qué son necesarias las evaluaciones de impacto es el que describe Howard White con respecto al Programa Integrado de Nutrición en Bangladesh (PINB) [27]. Este programa identificaba, mediante mediciones en campo, a los niños desnutridos y los asignaba a un tratamiento que incluía alimentación suplementaria a los menores y educación nutricional a las madres [1]. Inicialmente, el programa fue considerado como un éxito ya que los datos de monitoreo mostraban caídas importantes en los niveles de desnutrición. El Banco Mundial decidió, con base en esta evidencia y previo a cualquier tipo de evaluación, aumentar los recursos destinados al programa. Sin embargo,

las primeras evaluaciones de impacto, realizadas por el Grupo Independiente de Evaluación del mismo Banco Mundial y por la ONG inglesa *Save the Children*, mostraron que la mejoría de los indicadores de los beneficiarios era similar o inferior a la de otros niños con características comparables que no hacían parte del programa [1]. Estos resultados reflejaron que las percepciones de los administradores del programa y de las entidades financiadoras eran erradas, y sugirieron algunos correctivos al programa [1].

Otro ejemplo que evidencia la importancia de estas evaluaciones es proporcionado por [9], donde se evaluó el efecto de un programa de preescolar comunitario en zonas rurales de Mozambique, lanzado en 2008 por la organización Save the Children. El programa consistió específicamente en el financiamiento de la construcción, equipamiento y entrenamiento de 67 aulas en 30 comunidades [9]. El objetivo principal era verificar si un desarrollo en la infancia temprana contribuía a preparar a los niños para la escuela y las etapas posteriores de la vida. Los resultados mostraron no solo que la inscripción a la escuela primaria se incrementó en un 24 % en las comunidades tratadas, sino también que los niños beneficiarios dedicaban un promedio de 7.2 horas adicionales por semana a actividades escolares y redujeron el tiempo destinado a trabajos en las granjas familiares y a asistir a encuentros comunitarios. Además, la participación en el programa preescolar mostró mejoras consistentes en habilidades motoras finas, cognitivas, y de resolución de problemas [9]. Todos estos resultados contribuyeron a reforzar la idea de que los programas preescolares constituyen una política prometedora para mejorar la preparación escolar de niños pobres y desfavorecidos en áreas rurales de África.

Habiéndonos enfocado en el *qué* es una evaluación de impacto, a continuación, presentamos *cómo* se lleva a cabo una, detallando los elementos presentes y los problemas que pueden surgir al hacerlo.

2.1.2. Estimación del Tratamiento

Como mencionamos anteriormente, el resultado deseado al llevar a cabo una evaluación de impacto es el efecto del tratamiento, y se obtiene como la diferencia entre el resultado observado y el contrafáctico. Ahora bien, marco teórico estándar para formalizar este problema se conoce como **modelo de resultados potenciales** o **modelo causal de Rubin** [24].

Formalmente, se definen dos elementos para cada individuo $i = 1, ..., N$, donde N denota la población total:

- Por un lado, el indicador de tratamiento D, tal que $D_i = 1$ implica que el individuo i participó del tratamiento, y $D_i = 0$ en caso contrario.

- Por otro lado, para las variables de resultado se define $Y_i(D_i) = (Y_i|D_i)$ - se lee como "el valor de Y_i *dado* D_i". De esta forma, $Y_i(1)$ es la variable de resultado si el individuo i es tratado, e $Y_i(0)$ es la variable de resultado si el individuo i no es tratado. Estos valores son los resultados potenciales.

Con esto, el **efecto del tratamiento** para un individuo i se puede escribir como:

$$\tau_i = Y_i(1) - Y_i(0) = (Y_i|D_i = 1) - (Y_i|D_i = 0) \tag{2.1}$$

Según esta fórmula, el impacto causal (τ) de un programa (D) en una variable de resultado (Y) para un individuo i es la diferencia entre la variable con el programa ($Y_i(1)$) y la misma variable sin el programa ($Y_i(0)$).

De nuevo, el problema fundamental de la evaluación de impacto es que se intenta medir una variable en un mismo momento del tiempo para la misma unidad de observación pero en dos realidades diferentes. Sin embargo, en la realidad, solo se observa uno de los dos resultados potenciales $Y_i(1)$ o $Y_i(0)$, pero no ambos. Es decir, en los datos queda solamente registrado $Y_i(1)$ si $D_i = 1$ e $Y_i(0)$ si $D_i = 0$; no se dispone de $Y_i(1)$ si el individuo no fue tratado ($D_i = 0$), ni tampoco de $Y_i(0)$ si el individuo fue tratado ($D_i = 1$). De esta manera, el **resultado observado de** Y_i se puede expresar como:

$$Y_i = D_i Y_i(1) + (1 - D_i)Y_i(0) = \begin{cases} Y_i(1) \text{ si } D_i = 1 \\ Y_i(0) \text{ si } D_i = 0 \end{cases} \tag{2.2}$$

Si nos concentramos en las unidades tratadas, en la Ecuación 2.1, el término $Y_i(0) = (Y_i|D_i = 0)$ representa la situación contrafactual, es decir *cuál habría sido el resultado si la unidad no hubiera participado en el programa*. Al ser imposible observar directamente el contrafactual, es necesario **estimarlo**.

La forma más directa de solucionar este problema sería hallando un "clon perfecto" para cada uno de los individuos beneficiarios del programa pero que no lo hayan recibido, lo cual resulta bastante complicado. Por lo tanto, el primer paso para lograr esta estimación consiste en **desplazarse desde el nivel individual al nivel del grupo** [4], concentrando el análisis en el **impacto** o **efecto *promedio***.

En primera instancia, se puede estimar el **impacto promedio del programa *sobre la población*** (o ATE, del inglés *Average Treatment Effect*):

$$ATE = \mathbb{E}\left[Y_i(1) - Y_i(0)\right] \tag{2.3}$$

donde el operador $\mathbb{E}$ representa la esperanza de una variable aleatoria.

El ATE se interpreta como el cambio promedio en la variable de resultado cuando un individuo escogido al azar pasa aleatoriamente de ser participante a ser no participante [1], es decir representa el efecto esperado del tratamiento si toda la población recibiera el tratamiento.

Sin embargo, en la mayoría de los casos, el tratamiento solo está disponible para un subconjunto de la población, ya sea por restricciones presupuestarias o criterios de elegibilidad, entre otras razones. Además, también ocurre que no todos los que fueron seleccionados para recibirlo efectivamente participan. Es por esto que suele ser más relevante estimar el efecto únicamente en quienes efectivamente recibieron el tratamiento.

Por lo tanto, se define el **impacto promedio del programa *sobre los tratados*** (o *ATT*, del inglés *Average Treatment Effect on the Treated*), que es en general el parámetro de mayor interés en una evaluación de impacto, y representa el efecto esperado del tratamiento en el subconjunto de individuos que fueron efectivamente tratados:

$$ATT = \mathbb{E}\left[Y_i(1) - Y_i(0)|D_i = 1\right] = \mathbb{E}\left[Y_i(1)|D_i = 1\right] - \mathbb{E}\left[Y_i(0)|D_i = 1\right] \tag{2.4}$$

Es decir, el *ATT* es la diferencia entre la media de la variable de resultado en el grupo de los participantes y la media que hubieran obtenido los participantes si el programa no hubiera existido [1]. Ahora bien, ¿cómo podemos intentar estimar esta última situación?

2.1.3. El Grupo de Control como Estimador del Contrafactual

En la ecuación 2.4, el término $\mathbb{E}\left[Y_i(1)|D_i = 1\right]$ es un resultado observable mientras que $\mathbb{E}\left[Y_i(0)|D_i = 1\right]$ es el promedio contrafactual que debemos aproximar. Para lograrlo, se construye lo que se conoce como **grupo de control** o **de comparación**, formado por individuos que no participan del programa pero que, idealmente, son estadísticamente similares [4] al **grupo de tratamiento**, compuesto por quienes sí recibieron el programa. La Figura 2.1 ilustra esta distinción. A partir de ahora, denotaremos con C_i a un individuo i que pertenezca al grupo de control.

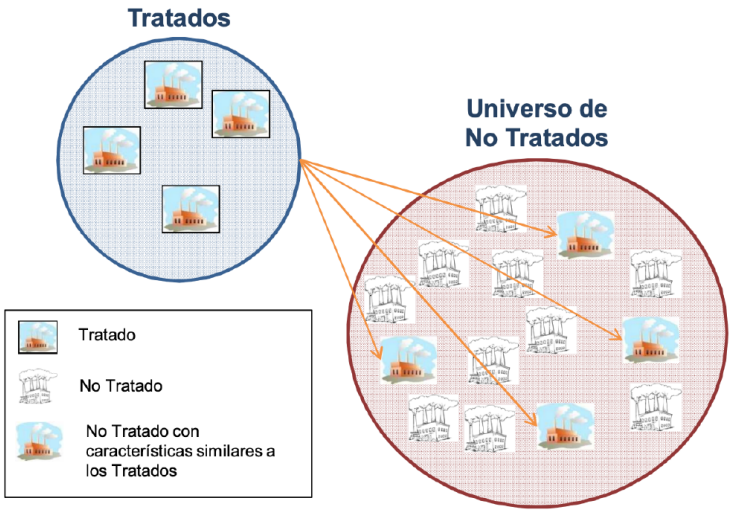


Figura 2.1: El grupo de control debería estar formado idealmente por individuos que no recibieron el tratamiento pero que en promedio poseen características similares a los tratados.

Por lo tanto, en la práctica, para obtener una buena estimación del *ATT*, el reto está en definir un *buen* grupo de control, es decir uno que sea estadísticamente idéntico al de

tratamiento, en promedio, en ausencia del programa [4]. Si se lograra que los dos grupos sean idénticos, con la única excepción que un grupo participa del programa y el otro no, entonces sería posible estar seguros que cualquier diferencia en los resultados tiene que deberse al programa [4].

Para que un grupo de comparación sea válido, debe satisfacer lo siguiente [4]:

- Las características *promedio* del grupo de tratamiento y del grupo de comparación deben ser idénticas en ausencia del programa. Cabe resaltar que no es necesario que las unidades individuales en el grupo de tratamiento tengan clones perfectos en el grupo de control.
- No debe ser afectado por el tratamiento de forma directa ni indirecta.
- El grupo de comparación debería reaccionar de la misma manera que el grupo de tratamiento si fuera objeto del programa.

Cuando el grupo de comparación no produce una estimación precisa del contrafactual, no se puede establecer el verdadero impacto del programa. A continuación, se presentan dos situaciones en las que esto ocurre.

Comparaciones antes-después

Este tipo de comparaciones intenta establecer el impacto de un programa a partir de un seguimiento de los cambios en los resultados en los participantes del programa a lo largo del tiempo. Consideran el contrafactual como el resultado para el grupo de tratamiento antes que comience la intervención, momento también conocido como **línea de base**. Si bien una ventaja de este método es que no es necesario contar con una grupo de control, esta comparación supone que si el programa no hubiera existido, el resultado para los participantes del programa habría sido igual a su situación antes del programa, lo cual en la mayoría de los tratamientos no puede sostenerse [4].

Comparaciones de inscritos y no inscritos: Sesgo de autoselección

Como explicamos anteriormente, el término $\mathbb{E}\left[Y_i(0)|D_i = 1\right]$ de la Ecuación 2.4 representa el contrafactual, que no es observable. Sin embargo, lo que sí se puede medir es la variable de resultado entre los no inscritos, aquellos individuos que no participaron del programa, es decir $\mathbb{E}\left[Y_i(0)|D_i = 0\right]$.

Por lo tanto, podríamos tomar a todo el conjunto de los no participantes como grupo de control y solucionar el problema del contrafactual utilizando $\mathbb{E}\left[Y_i(0)|D_i = 0\right]$ como estimador, es decir podríamos asumir lo siguiente:

$$\mathbb{E}\left[Y_i(0)|D_i = 1\right] = \mathbb{E}\left[Y_i(0)|D_i = 0\right] \tag{2.5}$$

Esto quiere decir que el valor esperado de la variable de resultado en ausencia del programa ($\mathbb{E}[Y_i(0)]$) es idéntico para el grupo de tratados ($D_i = 1$) y para el de no tratados ($D_i = 0$). Indica que los individuos que participan en el programa no son sistemáticamente distintos de los que no lo hacen [1] o, en otras palabras, que la decisión de participar en el programa no está determinada por factores que también influyen en la variable de resultado.

De esta forma, podríamos calcular el ATT como:

$$ATT = \mathbb{E}[Y_i(1)|D_i = 1] - \mathbb{E}[Y_i(0)|D_i = 0] \tag{2.6}$$

Sin embargo, el supuesto de la Ecuación 2.5 se viola toda vez que la participación en el programa es una *elección* del individuo elegible [1]. La razón es que los participantes y los no participantes generalmente son diferentes, aún en ausencia del programa, y son justamente esas diferencias que llevan a que algunos individuos escojan participar y otros no. Por lo tanto, si en estos casos se toma como grupo de control a todos aquellos que no decidieron participar y se mide el impacto con respecto a ellos, puede ocurrir que la diferencia en los resultados se deba a características propias de los individuos que llevaron a unos a anotarse y a otros no.

El problema descrito se conoce como **sesgo de autoselección**, ya que los integrantes del grupo de control se *autoseleccionaron* para no participar del programa. Más concretamente, este sesgo se produce cuando los motivos por los que un individuo participa en un programa, usualmente no observables y difíciles de medir, están correlacionados con los resultados [4], y por ende, es muy probable que la variable de resultado del grupo de tratamiento y del grupo de control sean diferentes, *aún si el programa no existiera* [1]. Intuitivamente, si hay variables que explican tanto la participación como el resultado potencial, la comparación de medias puede estar atribuyendo al programa un efecto que en realidad se debe a las diferencias preexistentes entre el grupo de tratamiento y el grupo de control [1].

Un buen ejemplo de esta situación lo presenta [1], en el que se plantea un programa de nutrición infantil. Podría ocurrir por ejemplo que las madres de familias participantes del programa sean más proactivas respecto al desarrollo de sus hijos, por lo cual se preocuparon en lograr la participación en el programa. El problema de autoselección en este caso radica en que la motivación de las madres (que no se observa y es difícil de medir) afecta no solo la probabilidad de participar en el programa, sino también el estado nutricional de los niños ya que estas podrían vigilar mejor la dieta de sus hijos. De esta forma, la diferencia observada en el estado nutricional de los niños de los dos grupos podría deberse parcialmente a la diferencia en el nivel de compromiso de las madres, y no exclusivamente a que un grupo participa en el programa y el otro no.

Podemos plantear este problema más formalmente. Notemos que podemos reescribir la fórmula del ATT (2.4) de la siguiente manera:

$$ATT + \mathbb{E}[Y_i(0)|D_i = 1] = \mathbb{E}[Y_i(1)|D_i = 1] \tag{2.7}$$

Si ahora restamos $\mathbb{E}[Y_i(0)|D_i = 0]$ a ambos lados, obtenemos:

$$ATT + (\mathbb{E}[Y_i(0)|D_i = 1] - \mathbb{E}[Y_i(0)|D_i = 0]) = \mathbb{E}[Y_i(1)|D_i = 1] - \mathbb{E}[Y_i(0)|D_i = 0] \quad (2.8)$$

De esta forma, es claro que si los indivudos del grupo de tratamiento y el grupo de control son diferentes, aún en ausencia del tratamiento ($\mathbb{E}[Y_i(0)|D_i = 1] - \mathbb{E}[Y_i(0)|D_i = 0] \neq 0$), entonces la diferencia entre la media del grupo de tratamiento y la media del grupo de control (el lado derecho de la igualdad) será igual al ATT más la diferencia preexistente entre los grupos [1]. Es decir, esta comparación de medias será una combinación del efecto directo del tratamiento, capturado en el ATT, y las diferencias preexistentes, y sin información adicional, no es posible distinguir qué parte se debe a qué [1].

Dicho esto, el gran desafío de la evaluación de impacto es determinar las condiciones o supuestos bajo los cuales $\mathbb{E}[Y_i(0)|D_i = 0]$ se puede utilizar como una estimación válida del contrafactual $\mathbb{E}[Y_i(0)|D_i = 1]$ [1], y por lo tanto utilizarse para poder aproximar el valor del ATT. Asegurarse de que el impacto estimado esté libre de sesgo de autoselección es uno de los principales objetivos de cualquier evaluación de impacto, y plantea importantes dificultades [4].

Es importante en este punto identificar claramente las comparaciones que hemos mencionado hasta el momento, los cuales se encuentran en la Tabla 2.1.

Lo que se desea medir: el ATT.	$\mathbb{E}[Y_i(1) - Y_i(0)|D_i = 1]$
Lo que se observa: el promedio de la diferencia entre los resultados de los tratados y el grupo de control.	$\mathbb{E}[Y_i(1) - C_i(0)]$
El sesgo de autselección: las diferencias preexistentes que pueden existir entre el grupo de tratamiento y el de control.	$\mathbb{E}[C_i(0) - Y_i(0)]$

Tabla 2.1: Distinción entre las diferentes comparaciones presentes en una evaluación de impacto.

Teniendo en cuenta las fórmulas expuestas la Tabla 2.1, podemos enunciar de otra forma lo que dijimos anteriormente: la calidad de una evaluación de impacto dependerá de los supuestos necesarios para asegurar que no hay sesgo de selección, es decir que $\mathbb{E}[C_i(0) - Y_i(0)] = 0$.

Las reglas de un programa para seleccionar a los participantes beneficiarios, también llamadas **reglas de asignación**, constituyen el parámetro clave para determinar el método de evaluación de impacto a utilizar para dicho tratamiento. A continuación, presentamos las distintas reglas y los métodos de evaluación que se utilizan en la práctica al trabajar con cada una de ellas.

2.1.4. Evaluaciones Experimentales o Aleatorias

Este tipo de evaluaciones se utiliza cuando los beneficiarios del programa en cuestión son seleccionados de manera completamente aleatoria[1]. Este tipo de tratamientos también son llamados ensayos controlados aleatorias. La asignación aleatoria se considera la regla de oro de la evaluación de impacto ya que no solo proporciona a los administradores del programa una regla imparcial y transparente para asignar recursos escasos entre poblaciones igualmente merecedoras de ellos, sino que también representa el método más sólido para evaluar el impacto de un programa [4].

La asignación aletoria trae dos consecuencias importantes: por un lado, las unidades ya no pueden *elegir* si participar o no, lo cual **elimina el sesgo de autoselección**, y por el otro, todas tienen la misma probabilidad de ser seleccionadas para el programa. De esta forma, el resultado potencial de cada individuo es independiente de sus condiciones previas, ya que la asignación al tratamiento no está determinada por sus características, sino por el azar.

Como explicamos anteriormente, el grupo de comparación ideal sería lo más similar posible al grupo de tratamiento en todos los sentidos, excepto con respecto a su participación en el programa que se evalúa. Ahora bien, el proceso de asignación aleatoria producirá dos grupos que tienen una alta probabilidad de ser estadísticamente idénticos en todas sus características (observables y no observables), siempre que el número de unidades sea suficientemente grande [4]. Esto es así ya que en general, si esto ocurre, este mecanismo asegura que cualquier rasgo de la población se transfiere a ambos grupos. Sin embargo, en la práctica, este supuesto debería comprobarse empíricamente con los datos de línea de base de ambos grupos.

Dicho esto, lo que ocurre en estos casos es que todos aquellos individuos que no resultaron ser beneficiarios del programa conforman un grupo de control que resulta ser naturalmente un excelente estimador del contrafactual $\mathbb{E}[Y_i(0)|D_i = 1]$. Es decir, mediante este tipo de asignación, se asegura que se cumple el supuesto de la Ecuación 2.5.

Una vez que se haya asignado el tratamiento de manera aleatoria, gracias a la eliminación del sesgo de autoselección y el consecuente cumplimiento de 2.5, es bastante sencillo estimar el impacto del programa. Después de que el programa ha funcionado durante un tiempo, tendrán que medirse los resultados de las unidades de tratamiento y de comparación. El impacto del programa es sencillamente la diferencia entre el resultado promedio para el grupo de tratamiento y el resultado promedio para el grupo de comparación, es decir:

$$ATT = \mathbb{E}[Y_i(1)|D_i = 1] \ - \mathbb{E}[Y_i(0)|D_i = 0]$$

[1]Cabe aclarar que la asignación aleatoria se produce dentro de la población de unidades elegibles, que está compuesta por aquellos individuos para los cuales interesa conocer el impacto del programa [4]. Por ejemplo, si se está implementando un programa de formación para los maestros de escuela primaria en zonas rurales, los maestros de escuela primaria de zonas urbanas o los profesores de secundaria no formarían parte del conjunto de unidades elegibles [4].

La asignación aleatoria puede utilizarse como regla de asignación de un programa en dos escenarios específicos: cuando la población elegible es mayor que el número de plazas disponibles del programa, o cuando sea necesario ampliar un programa de manera progresiva hasta que cubra a toda la población elegible [4]. Además, la asignación aleatoria podría hacerse a nivel individual (por ejemplo, a nivel de hogares), o a nivel de conglomerado (por ejemplo, comunidades) [1].

Resulta claro que las evaluaciones experimentales tienen varias ventajas: por diseño, al quitarles la posibilidad de elección a los individuos sobre si participar o no en el programa, resuelven el problema del sesgo de selección, y además los resultados obtenidos son transparentes, intuitivos y no es necesario utilizar herramientas econométricas sofisticadas para alcanzarlos [1]. Esto hace que contribuyan a la transparencia de las políticas públicas y a la rendición de cuentas [1].

Sin embargo, este tipo de experimentos sufre varias limitaciones que hace que no sea tan común en la práctica. Por un lado, pueden ser costosos y de difícil implementación [1]. Esto se debe a que en general, el ensayo aleatorio se hace específicamente para evaluar una intervención, por lo que es necesario destinar recursos para la prueba piloto, la recolección de información, los seguimientos, y a veces incluso la implementación del programa [1]. Otro problema que tienen, y probablemente el más importante, es que por pertenecer al grupo de control, se *excluye* a un segmento de la población, igualmente vulnerable y elegible, de los beneficios de la intervención [1]. Además, dada la imposibilidad de negar los beneficios a un grupo de control por largos períodos, con frecuencia es imposible estimar los impactos de largo plazo [1].

2.1.5. Evaluaciones Cuasi-experimentales

Cuando la asignación aleatoria de los participantes no es factible o plantea problemas éticos, se emplean las llamadas evaluaciones o técnicas cuasi-experimentales, que son aquellas que buscan estimar el impacto causal, pero a diferencia de las experimentales, no se basan en la asignación aleatoria de la intervención [4]. En estos casos, el programa que se intenta evaluar no tiene una regla de asignación clara que explique por qué ciertos individuos se inscribieron en el programa y otros no lo hicieron, por lo que se convierte en una incógnita adicional en la evaluación, acerca de la cual se deben formular supuestos [4]. Para ello, estos métodos intentan *simular* la aleatorización de un diseño experimental controlando las diferencias entre los individuos tratados y no tratados bajo diferentes hipótesis.

Si bien los métodos cuasi-experimentales suelen ser más adecuados en algunos contextos operativos, su desventaja es que requieren más condiciones para garantizar que el grupo de comparación provea una estimación válida del contrafactual, como veremos a continuación al profundizar en algunos de ellos.

Diferencias en Diferencias

El modelo de diferencias-en-diferencias (DD) es una manera de controlar la estimación del impacto por las diferencias preexistentes entre el grupo de tratamiento y el de control, que estará formado por todos aquellos individuos elegibles que no recibieron el tratamiento. Es decir, el DD no propone una forma de construir un grupo de control adecuado sino una manera de aproximar el impacto que tenga en cuenta las diferencias entre participantes y no participantes. A grandes rasgos, combina las comparaciones antes-después con las comparaciones de inscritos y no inscritos, ambas discutidas anteriormente.

Dada una variable de resultado Y, este modelo establece el impacto como el cambio esperado en Y entre el período posterior y el período anterior a la implementación del tratamiento en el grupo de tratamiento, menos la diferencia esperada en Y en el grupo de control en el mismo período [1]. Es decir, **compara los cambios en los resultados a lo largo del tiempo** entre unidades participantes y no participantes.

Si denotamos con $Y^{(1)}$ al valor de la variable Y en la línea de base, es decir justo antes de la implementación del programa, y con $Y^{(2)}$ al valor de la misma variable en una período posterior a la implementación (también llamado período de seguimiento), entonces el impacto del programa por el método de DD estaría dado por:

$$ATT_{DD} = \left(\mathbb{E}\left[Y^{(2)}|D=1\right] - \mathbb{E}\left[Y^{(1)}|D=1\right]\right) - \left(\mathbb{E}\left[Y^{(2)}|D=0\right] - \mathbb{E}\left[Y^{(1)}|D=0\right]\right) \tag{2.9}$$

Es importante señalar que el contrafactual que se estima en este caso es el cambio en los resultados del grupo de tratamiento, y su estimación se logra midiendo el cambio en los resultados del grupo de comparación [4]. En lugar de contrastar los resultados entre los grupos de tratamiento y comparación después de la intervención, la técnica de DD estudia las *tendencias* entre los grupos de tratamiento y comparación [4].

Como explicamos con anterioridad, el principal problema de las comparaciones antes-después en el grupo de tratamiento es que se asume que lo único que hizo cambiar la variable de resultado fue el programa, cuando en realidad puede haber sido por factores externos. El DD busca solucionar este problema basándose en la idea que la diferencia en los resultados antes-después para el grupo de tratamiento controla por factores que son constantes a lo largo del tiempo en ese grupo, y la manera de capturar los factores variables en el tiempo es medir el cambio antes-después en los resultados de un grupo que no se inscribió en el programa pero que estuvo expuesto al mismo conjunto de condiciones ambientales. De esta forma, si se "limpia" la primera diferencia de otros factores variables en el tiempo que influyen en el resultado de interés sustrayendo la segunda diferencia, se habrá eliminado una fuente de sesgo [4].

Por otro lado, en las comparaciones de inscritos y no inscritos, el gran inconveniente era el sesgo de autoselección. Para entender mejor la solución que el DD propone a esto,

resulta conveniente reescribir el estimador como:

$$ATT_{DD} = \left(\mathbb{E}\left[Y^{(2)}|D=1\right] - \mathbb{E}\left[Y^{(2)}|D=0\right]\right) - \left(\mathbb{E}\left[Y^{(1)}|D=1\right] - \mathbb{E}\left[Y^{(1)}|D=0\right]\right) \tag{2.10}$$

De la Ecuación 2.10 se puede ver que lo que plantea es restarle al cambio visto en la variable entre tratados y no tratados luego de la implementación del programa ($Y^{(2)}$), las diferencias que ya pueda haber habido entre ellos en la línea de base ($Y^{(1)}$). Es decir, desde este punto de vista, se utiliza a $\mathbb{E}\left[Y^{(1)}|D=1\right] - \mathbb{E}\left[Y^{(1)}|D=0\right]$ como un estimador de las diferencias preexistentes entre el grupo de tratamiento y el grupo de control.

Ahora bien, volviendo a la Ecuación 2.9, el supuesto que permite utilizar la diferencia en el grupo de no inscritos para controlar los factores externos que afectan a los inscritos es que, en ausencia del programa, dichos factores habrían impactado a los tratados de la misma manera que a los no tratados. En otras palabras, se asume que, sin la intervención, los resultados en el grupo tratado habrían evolucionado en paralelo con los del grupo de control, aumentando o disminuyendo en la misma proporción. Esto se conoce como **supuesto de tendencias paralelas**, y es sobre el que hay que basarse al trabajar con este método ya que claramente no es posible observar qué habría ocurrido con el grupo tratado en ausencia del programa.

Propensity Score Matching

Otro enfoque para lograr una buena aproximación del contrafactual consiste en construir un grupo de comparación artificial, a partir del conjunto de individuos que no recibieron el tratamiento. En esto se basa el método conocido como **pareamiento** o **emparejamiento**, que utiliza técnicas estadísticas y grandes bases de datos para construir el mejor grupo de comparación posible sobre la base de características observables [4]. La principal utilidad de este método es que puede aplicarse en el contexto de casi todas las reglas de asignación de un programa, siempre que se cuente con un grupo que no haya participado en el mismo [4].

El supuesto que utilizan estas técnicas para basarse en características observables al construir el grupo de control es que tanto la participación en el programa como los resultados potenciales están únicamente determinados por estas. O, en otras palabras que no están determinados por variables no observables (o no medidas).

De esta forma, se asume que al condicionar los resultados potenciales de tratados y no tratados en determinadas características observables (pertinentes en el programa bajo estudio), el sesgo de autoselección es igual a cero. Formalmente, si denotamos con X a los rasgos observables que se están teniendo en cuenta, este supuesto se escribe de la siguiente forma:

$$\mathbb{E}\left[Y_i(0)|D_i=1,X\right] = \mathbb{E}\left[Y_i(0)|D_i=0,X\right], \tag{2.11}$$

y se lo conoce como **supuesto de Independencia Condicional** (IC).

Con esto en mente, la versión más directa de esta metodología consiste en encontrar un

"clon" *para cada individuo tratado* en el grupo de no tratados y contrastar las variables de resultado de ambos [1]. Un clon en este caso quiere decir, una vez fijadas las características que el investigador cree que explican la decisión de inscribirse en el programa, un individuo (o grupo de individuos) con exactamente las mismas características observables X.

Sin embargo, como mencionamos anteriormente, en la práctica esto resulta difícil. Si la lista de características observables relevantes es muy grande, o si cada característica adopta muchos valores, puede resultar computacionalmente complejo identificar una pareja para cada una de las unidades del grupo de tratamiento. Esta situación es conocida como *el problema de la dimensionalidad*.

Esta situación puede solucionarse utilizando un método denominado **Pareamiento por Puntajes de Propensión** (*Propensity Score Matching*, PSM), presentado por primera vez en 1983 en [22]. Consiste en emparejar individuos ya no con base en X, sino en su probabilidad estimada de participación en el programa, dadas sus características observables, es decir en:

$$P(X) = P(D = 1|X)$$

donde P denota el operador de probabilidad de un evento . Este valor es conocido como **probabilidad de participación** o **puntaje de propensión**.

De esta forma, el clon adecuado para cada individuo del grupo de tratamiento será aquel del grupo de no tratados con una probabilidad de participación en el programa suficientemente cercana [1]. La ventaja de emparejar a partir de $P(X)$, a diferencia de X, es que $P(X)$ es un número, mientras que X puede tener una dimensión muy grande [1]. Cabe aclarar que para calcular este puntaje, solo deberían utilizarse las características en la línea de base, ya que post-tratamiento, estas pueden haberse visto afectadas por el propio programa [4].

Ahora bien, puede surgir la siguiente pregunta: ¿es posible encontrar (al menos) una pareja para todos los individuos tratados? Y la respuesta es no, y para aquellos individuos no será posible estimar el impacto del programa. Lo que puede ocurrir en la práctica es que para algunas unidades inscritas, no haya unidades en el conjunto de no inscritos que tengan puntajes de propensión similares. En términos técnicos, puede que se produzca una falta de **rango o soporte común**.

La condición de soporte común (SC) establece que individuos con el mismo vector de variables X tienen probabilidad positiva de ser tanto participantes como no participantes del programa [1]. Esto es, fijado un vector de variables $\hat{X}$:

$$0 < P(D = 1|\hat{X}) < 1$$

Si esta condición no se cumple, sería posible encontrar una combinación particular de características que predice perfectamente la participación en el programa, y por tanto, no existiría un individuo que fuera un buen control (o viceversa) [1].

El SC implica que solo se utilizarán en la estimación aquellos individuos tratados para los cuales se pueda encontrar uno no tratado con un puntaje de propensión similar.

Por ejemplo, si existen individuos del grupo de tratamiento con una probabilidad de participación muy alta, pero ningún individuo no participante exhibe un puntaje tan alto, entonces estos individuos tratados se descartarán a la hora de hacer el emparejamiento.

Asumiendo que se cumplen las condiciones de IC y SC, el estimador del ATT por PSM estaría dado por:

$$ATT_{PSM} = \mathbb{E}_{P(X)|D=1} \{\mathbb{E}[Y(1)|D=1, P(X)] - \mathbb{E}[Y(0)|D=0, P(X)]\} \tag{2.12}$$

donde $\mathbb{E}_{P(X)|D=1}$ es el valor esperado con respecto a la probabilidad de participación $P(X)$, condicional en ser participante del programa [1]. Es decir, un promedio ponderado de las diferencias entre llaves, donde los ponderadores son funciones de la probabilidad de participación en el programa [1].

Los pasos a seguir a la hora de aplicar la técnica de PSM se pueden resumir en los siguientes puntos:

1. Identificar las unidades que se inscribieron en el programa y las que no lo hicieron.

2. Definir el conjunto de variables observables X que se utilizarán para calcular el puntaje de propensión $P(X)$.

3. Calcular el puntaje de propensión $P(X)$ para cada individuo, tratados y no tratados.

4. Restringir la muestra al soporte común con respecto a la distribución del puntaje de propensión.

5. Seleccionar un algoritmo de emparejamiento para emparejar a cada individuo tratado con un individuo o grupo de individuos no tratados que tenga una probabilidad de participación similar.

6. Se calcula el impacto del programa como el promedio apropiadamente ponderado de la diferencia entre la variable de resultado de los tratados y sus parejas no tratadas (Ecuación 2.12).

A continuación, se explican brevemente algunos de estos pasos.

En cuanto a la elección de variables (paso **2**) que se incluirán para calcular el puntaje de propensión, si bien es importante poder parear utilizando un gran número de características, lo es más aún hacerlo sobre la base de aquellas que **determinan la inscripción** [4]. En términos estadísticos, estas características reciben el nombre de **covariables**, **variables independientes** o **variables explicativas**, y son justamente aquellas que posiblemente predicen el resultado bajo estudio.

Cuanto más se comprenda acerca de los criterios utilizados para la selección de los participantes, en mejores condiciones se estará de construir un buen grupo de comparación [4]. Para tener una mejor comprensión, los investigadores se pueden guiar por los

modelos económicos que describen el fenómenos bajo estudio, investigaciones previas y conocimiento del diseño institucional [1]. Además, como dijimos anteriormente, estas características deben ser medidas en la línea de base, y no una vez aplicado el programa, ya que se pueden haber visto afectadas por el mismo.

Con respecto al cálculo del puntaje de propensión (paso **3**), la idea es especificar un modelo $P(D = 1|X) = f(X)$, es decir, la probabiliad de participación como una función que puede ser lineal o no lineal en las características observables de los individuos, X [1]. Con frecuencia se prefieren los modelos de regresión logística (*logit*) o de probabilidad (*probit*) [1]. En este trabajo, utilizaremos la regresión logística, por lo que resulta conveniente saber de qué se trata.

En lo que refiere al algoritmo de emparejamiento, estos suelen establecer tanto la manera en que se encuentra el individuo (o grupo de individuos) parecido a un tratamiento como así también la manera en que se pondera la diferencia a la hora de hacer la comparación. También se los llama estimadores ya que en fin, determinar una forma de estimar el ATT.

Uno de los más usados y más simples de entender es el de vecino más cercano (o *NN matching*, por su sigla en inglés, *nearest neighbor*) [1]. En el NN matching, una opción es emparejar cada individuo de tratamiento con aquel del grupo de no tratados que tenga la probabilidad de participación más cercana. Es decir, la unidad no tratada tal que la distancia entre su puntaje y el del individuo tratado sea mínima. En este caso, una vez que se haya emparejado a cada individuo tratado, el ATT se calcula como el promedio simple de las diferencias entre las variables de resultado en cada pareja.

Otra opción es dado un individuo tratado, tomar los k vecinos más cercanos del grupo de no tratados con respecto al puntaje de propensión estimado, y estos formarán el grupo de comparación para el tratado. Aquí, para calcular el impacto del programa, se puede dar igual peso a cada uno de los k vecinos, en cuyo caso el impacto para una unidad tratada estará dada por la diferencia entre la variable observada de ella y el promedio simple de las variables de resultado de los vecinos cercanos [1]. Otra alternativa es hacer un promedio ponderado de las observaciones de los vecinos, según qué tan comparable es cada vecino: así, el valor de la variable de resultado de un vecino más cercano tiene más peso que el de uno no tan cercano.

Existen algunas cuestiones a tener en cuenta al utilizar este algoritmo, como puede ser el hecho que la diferencia de puntajes entre un tratado y su vecino más cercano no sea lo suficientemente pequeña como para asegurar que ambos son muy parecidos, y aún así la diferencia entre sus variables de resultado contribuirá al cálculo del ATT.

Otros algoritmos utilizados para el emparejamiento son los de *kernel* y regresión lineal local. Estos son estimadores no paramétricos que emparejan a cada individuo del grupo de tratamiento con un promedio ponderado de (potencialmente) todos los individuos del grupo de control [1]. En principio, con estos, se puede comparar a cada individuo de tratamiento con todos los no tratados, y darles menos peso en la comparación a aquellos con probabilidades de participación más lejanas [1]. Una ventaja de ellos es que tienen

menor varianza a la hora de calcular las diferencias individuales (que vecino más cercano por ejemplo) porque usan más información [1]. Para más detalles sobre cada uno de estos, se puede consultar el Capítulo 6 de [1].

Una evaluación en la que se utilizó la técnica de PSM, particularmente con el algoritmo del vecino más cercano, fue la llevada a cabo a partir del año 2021 por la Unidad de Evaluación Integral de la Calidad y Equidad Educativa de la Ciudad Autónoma de Buenos Aires (CABA) sobre el impacto del programa "Jornada Extendida" en el clima escolar [19], aplicado sobre diferentes escuelas primarias de la ciudad. El PSM fue empleado para seleccionar las escuelas del grupo de control en base a tres características: el ISSAP (Índice de Situación Socioeconómica de los/as Alumnos/as en Escuelas Primarias), la tasa de sobreedad y la tasa de repitencia.

Como mencionamos anteriormente, la gran ventaja del método de PSM es que es flexible, pudiéndose utilizar en numerosos contextos, independientemente de la regla de asignación de un programa, por lo que se emplea con frecuencia [1].

Sin embargo, es muy importante notar que los resultados arrojados por el PSM serán confiables solamente si se cumple el supuesto de IC. Es decir, cuando existan razones para pensar que las variables no observables o no disponibles en la base de datos no son un determinante fundamental tanto de la participación en el programa como de las variables de resultado potenciales. En otros términos, cuando no haya diferencias sistemáticas en las características no observables entre las unidades tratadas y las de comparación pareadas que puedan influir en el resultado.

Cabe aclarar que el supuesto de IC no puede ser demostrado, ya que no se puede verificar si lo no observado afecta o no la decisión de participación, pero su plausibilidad puede ser argumentada basándose en el conocimiento sustantivo del área de estudio. Otra alternativa que se lleva a cabo en la práctica es calcular las diferencias en las variables observables X; si los dos grupos son demasiados diferentes en estas, esto podría ser evidencia de que es probable que también existan diferencias entre los dos grupos en características no observadas [1].

Por otro lado, si bien el cálculo del ATT se puede restringir a aquellos individuos que se encuentra en la región de soporte común entre tratados y no tratados, esta restricción puede volverse preocupante cuando hay una fracción importante de individuos tratados que quedan sin emparejar. Esto implica que existirá un conjunto de individuos para el cual no se puede decir nada acerca del efecto del programa [1].

En este trabajo, utilizaremos el PSM con el algoritmo de *[nombre del algoritmo]* como referencia para compararlo con nuestro enfoque, lo cual se explicará con más detalle en la sección 3.

A continuación, introducimos el área de Inteligencia Artificial, dentro de la cual se enmarcan los algoritmos propuestos en este trabajo.

2.2. Inteligencia Artificial

La definición más general acerca de la Inteligencia Artificial (IA) establece que es el área de las Ciencias de la Computación que se enfoca tanto en entender como en crear sistemas que simulen comportamientos *inteligentes*. Si bien existen distintas perspectivas sobre qué significa que una computadora actúe de manera inteligente, la que más ha prevalecido a lo largo de los años es aquella que refiere con la capacidad de computar cómo actuar de la mejor manera posible en una determinada situación [26].

En sus comienzos, los métodos desarrollados en el área de la IA estaban principalmente **basados en conocimiento**, es decir reglas matemáticas formales que permitían a las computadoras llevar a cabo inferencias lógicas y de esta forma resolver problemas que eran intelectualmente difíciles para los humanos [5].

Sin embargo, determinar reglas que describan la complejidad y diversidad de la realidad no era una tarea fácil. De esta manera, con el objetivo de hacer a estos sistemas más flexibles y capaces de adaptarse y entender diferentes situaciones, se transicionó hacia un enfoque en el que estos pudieran obtener su propio conocimiento aprendiendo patrones directamente a partir de los datos en lugar de depender exclusivamente de reglas predefinidas.

Este cambio de paradigma dio lugar a lo que hoy se conoce como **Aprendizaje Automático**, la subdisciplina de la IA que permite a los algoritmos mejorar su desempeño en una tarea específica automáticamente a partir de la experiencia. Diversos factores como la creciente disponibilidad de datos, el aumento en la capacidad computacional, y los avances en los algoritmos de optimización [5] han hecho que esta área sea la que mayor desarrollo e impacto ha tenido durante las últimas décadas.

Actualmente, la IA abarca una diversidad de tareas, que van desde lo general, como son las habilidades de aprendizaje, razonamiento, y percepción, entre otras; hasta lo específico, como probar teoremas matemáticos, manejar vehículos, mejorar procesos industriales o incluso diagnosticar enfermedades.

2.2.1. Aprendizaje Automático

El Aprendizaje Automático, más conocido por su nombre en inglés *Machine Learning* (ML), es un campo dentro de la IA cuyo objetivo es desarrollar técnicas que permitan que las computadoras *aprendan* automáticamente a partir de la *experiencia* - los datos -, sin la necesidad de ser explícitamente programadas para hacerlo.

Un ejemplo de estos algoritmos puede ser un clasificador de correo spam, que aprende a distinguir correos spam de regulares viendo ejemplos de cada tipo de correo. Otro ejemplo puede ser un sistema que aprenda a predecir la edad de una persona a partir de una imagen habiendo experimentado previamente algunos ejemplos de imagen-edad.

En 1997, Tom Mitchell definió en su libro Machine Learning [11] el concepto de "aprender" de la siguiente manera: "Se dice que un programa de computadora aprende de la

experiencia E con respecto a una clase de tareas T y una medida de desempeño P, si su desempeño en las tareas de T, medido por P, mejora con la experiencia E". Para tener un mejor entendimiento, nos enfocamos a continuación en cada uno de estos tres componentes.

Las tareas de ML se describen usualmente en términos de cómo el sistema debería procesar un *ejemplo* o *entrada*, entendiendo a esta como un conjunto de características (o en inglés, *features*) medidas cuantitativamente a partir de un cierto objeto o evento [5]. Por ejemplo, una entrada puede estar compuesta por los datos de un hogar, como la cantidad de habitaciones y de baños, si tiene patio o no, el tamaño de la cocina, etcétera. Dentro de las tareas que pueden ser resueltas por un sistema de ML se encuentran la clasificación, la regresión, la traducción, la detección de objetos en imágenes, la generación de nuevos datos, la detección de valores atípicos, entre muchas otras. Siguiendo con el ejemplo, podríamos querer que, en base a las features de las casa, el sistema nos provea un estimado de su valor.

La experiencia hace referencia al tipo de información que el algoritmo "puede ver" durante su proceso de aprendizaje o *entrenamiento* [3]. A esta información se la conoce como "conjunto (de datos) de entrenamiento", y es un subconjunto del *dataset*, que es simplemente el conjunto de todos los ejemplos o datos con los que se cuenta. En base a la experiencia, los algoritmos de ML se clasifican en dos grandes categorías:

- **Algoritmos de Aprendizaje Supervisado**: experimentan un dataset que contiene pares de entrada-salida, esto es cada ejemplo contiene sus características pero también su "etiqueta", que vendría a ser la "respuesta correcta" para dicha entrada, y la que se espera que el sistema aprenda. De esta forma, el algoritmo aprende una función que asigna (*mapea*) entradas a salidas. Las tareas más comunes llevadas a cabo con este tipo de aprendizaje son las de regresión, en donde la etiqueta corresponde a un valor continuo; y clasificación, en donde la solución viene dada por la categoría (dentro de un conjunto predefinido de categorías) a la que pertenece un ejemplo.

- **Algoritmos de Aprendizaje No Supervisado**: ven un dataset que cuenta solamente con features de cada entrada pero sin etiquetas, e intentan aprender automáticamente patrones y propiedades útiles de la estructura de los datos. Algunas tareas que se llevan a cabo con este tipo de aprendizaje son *clustering*, reducción de dimensionalidad, y detección de anomalías.

También existen otros tipos de algoritmos, como los de **Aprendizaje Semi-supervisado**, en donde el dataset contiene algunos ejemplos etiquetados y otros sin etiquetas; y los de **Aprendizaje por Refuerzo**, en donde el algoritmo aprende la mejor estrategia para una situación a partir de la interacción con su entorno en forma de recompensas y castigos.

Por último, para medir el desempeño de estos algoritmos, se definen métricas cuantitativas que dependen de la tarea que se esté realizando y del objetivo que se intente lograr.

Por ejemplo, en clasificación, una de las más comunes es la de exactitud (*accuracy*), que es la proporción de ejemplos para los cuales se predijo la salida correcta (dada por el dataset); aunque también existen otras como la precisión, sensibilidad, especificidad, y el puntaje F1, que pueden resultar más adecuadas según el dominio del problema. Por otro lado, en tareas de regresión, algunas métricas que se suelen utilizar son el Error Cuadrático Medio y el Error Absoluto Medio.

El objetivo fundamental del ML es que un algoritmo actúe correctamente ante nuevas entradas desconocidas, es decir que **generalice** más allá de los ejemplos del conjunto de entrenamiento. Por ello, aunque las métricas anteriores pueden calcularse durante el entrenamiento para verificar que el algoritmo esté mejorando, su verdadero desempeño se evalúa en un subconjunto del dataset distinto al de entrenamiento, denominado "conjunto de test". Como buena práctica, este conjunto debe permanecer completamente separado del proceso de aprendizaje para obtener una estimación realista de la capacidad de generalización del algoritmo.

Habiendo presentado una idea general sobre cuál es el objetivo de los algoritmos de Aprendizaje Automático, ahora nos enfocaremos en los de Aprendizaje Supervisado, que son los que usaremos en este trabajo.

2.2.2. Aprendizaje Supervisado

Como mencionamos anteriormente, en el Aprendizaje Supervisado el algoritmo aprende una función que mapea entradas a salidas a partir de un conjunto de entrenamiento compuesto por datos etiquetados. El objetivo es que al presentarle a esta función una nueva entrada no vista previamente, esta sea capaz de computar la salida que mejor se ajusta a los **patrones** aprendidos durante el entrenamiento.

Resulta conveniente introducir en este punto un término muy utilizado en el ML: **modelo**. Un modelo es simplemente una ecuación matemática, que se presenta como una forma simplificada de describir hechos de la realidad. En este contexto, será una manera de intentar capturar las relaciones entre los datos, con el objetivo de hacer predicciones basadas en los patrones aprendidos de ejemplos previos. En general, se suele utilizar la palabra modelo para referirse a la función de la que hablamos en el párrafo anterior.

Formalmente, una tarea de Aprendizaje Supervisado es la siguiente:

> Dado un conjunto de entrenamiento de N ejemplos de entrada-salida:
>
> $$(\boldsymbol{x}_1, \boldsymbol{y}_1), (\boldsymbol{x}_2, \boldsymbol{y}_2), ..., (\boldsymbol{x}_N, \boldsymbol{y}_N)$$
>
> donde cada $\boldsymbol{x}_i$ es un vector de características ($\boldsymbol{x}_i \in \mathbb{R}^n$ para algún $n \in \mathbb{N}$) e $\boldsymbol{y}_i$ es la salida correspondiente ($\boldsymbol{y}_i \in \mathbb{R}^m$ para algún $m \in \mathbb{N}$), y cada par fue

[1] Hablaremos de estas más adelante, dando sus fórmulas y lo que representan.

generado por una **función desconocida** $\boldsymbol{y} = f(\boldsymbol{x})$, descubrir una función h que aproxime a la función real f [26].
Denotaremos con $\boldsymbol{X}$ al vector de entradas del conjunto de entrenamiento, y con $\boldsymbol{Y}$ al vector de salidas, ambos de tamaño N y bien ordenados. Es decir:

$$\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N)$$
$$\boldsymbol{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_N)$$

De esta forma, el modelo es la función h, que toma un vector de entrada $\boldsymbol{x}$ y devuelve una salida $\boldsymbol{y}$, y se presenta como una **hipótesis** de f. La suposición sobre la que se trabaja es que si el modelo funciona bien para los pares de entrenamiento, entonces se espera que va a realizar buenas predicciones para nuevas entradas cuya etiqueta se desconoce.

Ahora bien, esta función h se obtiene a partir de un **espacio de funciones**, que es elegido por quien diseña el modelo. Este espacio podría ser el conjunto de funciones lineales, de polinomios de grado 2, de polinomios de grado 3, etcétera. Por lo tanto, el espacio determina la forma o "**arquitectura**" que va a tener la función h, y consecuentemente cuáles son sus parámetros, cuyo vector denotaremos con la letra $\boldsymbol{w}$, y a los que también se suele llamar "**pesos**". De esta forma, este espacio determina la familia de posibles relaciones entre entradas y salidas, y los parámetros especifican la relación particular (el modelo).

Por ejemplo, si suponemos que cada entrada $\boldsymbol{x} \in \mathbb{R}$, y establecemos como espacio el conjunto de funciones lineales, entonces la forma de h será:

$$h(\boldsymbol{x}) = w_1\boldsymbol{x} + w_0$$

y sus parámetros $\boldsymbol{w} = (w_0, w_1)$. Este caso es comúnmente conocido como regresión lineal unidimensional (ya que la entrada es simplemente un número real). Dentro de este espacio, distintas asignaciones de valores a los parámetros generan distintos modelos. Por ejemplo, tomando la asignación $\boldsymbol{w} = (13, 5)$, tenemos el modelo $h(x) = 5\boldsymbol{x} + 13$, y tomando $\boldsymbol{w} = (1.25, \pi)$, $h(\boldsymbol{x}) = \pi\boldsymbol{x} + 1.25$, entre muchos otros.

En cambio, si tomamos el conjunto de polinomios de grado 2 y seguimos suponiendo $\boldsymbol{x} \in \mathbb{R}$, entonces la forma de h será:

$$h(\boldsymbol{x}) = w_2\boldsymbol{x}^2 + w_1\boldsymbol{x} + w_0$$

y sus parámetros $\boldsymbol{w} = (w_0, w_1, w_2)$. A este caso se lo suele llamar regresión polinómica de segundo grado unidimensional.

Otro caso un poco más "complejo" podría ser en el que $\boldsymbol{x} \in \mathbb{R}^3$, es decir $\boldsymbol{x} = (x_1, x_2, x_3)$ y seguimos tomando una relación lineal. Aquí, h sería:

$$h(\boldsymbol{x}) = w_1x_1 + w_2x_2 + w_3x_3 + w_0$$

y sus parámetros $\boldsymbol{w} = (w_0, w_1, w_2, w_3)$[2].

Notemos entonces que, fijado un espacio de funciones, quienes van a determinar qué modelo es "mejor" que otro en este espacio, es decir qué modelo aproxima mejor a f, son los valores de los parámetros w_i. Por lo tanto, h en realidad es una función de la entrada $\boldsymbol{x}$ pero también de los parámetros, establecidos por el espacio de funciones elegido:

$$h(\boldsymbol{x}, \boldsymbol{w})$$

Y lo que querremos idealmente es hallar el vector de parámetros $\boldsymbol{w}^*$ que hagan que el modelo resultante cumpla:

$$f(\boldsymbol{x}) \approx h(\boldsymbol{x}, \boldsymbol{w}^*)$$

para cada $\boldsymbol{x}$ en el conjunto de entrenamiento.

Entonces, cuando un modelo se entrena o aprende, lo que realmente hace es intentar encontrar los valores de los parámetros que describan la verdadera relación entre entradas y salidas [18]. Un algoritmo de aprendizaje toma el conjunto de entrenamiento y manipula los parámetros de forma iterativa hasta que las predicciones dadas por él sean lo más cercanas posible a las etiquetas verdaderas.

Para que el algoritmo sepa cómo modificar estos parámetros para mejorar sus predicciones, necesita una forma de conocer cómo está siendo su desempeño. Para esto, se define lo que se conoce como **función de pérdida** o **función de error**, que denotaremos con la letra L y que justamente hace eso: retorna un número que resume qué tan bien o mal está funcionando el modelo con sus parámetros $\boldsymbol{w}'$ actuales, en términos de qué tan lejos están sus predicciones de las respuestas reales del conjunto de entrenamiento.

Como venimos enfatizando, lo que caracteriza al modelo en cada iteración es el valor de sus parámetros. Por lo tanto, tiene sentido tratar a la función de pérdida como una que depende de estos, es decir $L(\boldsymbol{w})$[3]. A continuación, se mencionan dos ejemplos de funciones de error:

- Una función de pérdida que se es común usar en problemas de regresión es la de Error Cuadrático Medio (ECM), dada por:

$$ECM(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \left(h(\boldsymbol{x}_i, \boldsymbol{w}) - \boldsymbol{y}_i\right)^2$$

[2]Para simplificar la notación, lo que se suele hacer es asumir que la entrada $\boldsymbol{x}$ tiene un componente adicional constante x_0 con el valor 1, es decir $\boldsymbol{x} = (x_0, x_1, x_2, x_3) = (1, x_1, x_2, x_3)$. De esta forma, podríamos escribir: $h(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w}$, donde el operador $\cdot$ representa el producto escalar. Retomaremos esta idea más adelante al hablar sobre Redes Neuronales.

[3]En realidad la función de pérdida también depende de los datos de entrenamiento, por lo que si somos estrictos deberíamos escribir $L(\{\boldsymbol{X}, \boldsymbol{Y}\}, \boldsymbol{w})$. Sin embargo, como estos datos están fijos durante todo el proceso de aprendizaje, podemos omitirlos.

- Para problemas de clasificación binaria como el que tratamos en este trabajo, vamos a tener $\boldsymbol{y}_i \in \mathbb{R}$, y se suele emplear la Entropía Cruzada Binaria (ECB), dada por:

$$ECB(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N}[\boldsymbol{y}_i \; ln(h(\boldsymbol{x}_i, \boldsymbol{w})) + (1 - \boldsymbol{y}_i) \; ln(1 - h(\boldsymbol{x}_i, \boldsymbol{w}))]$$

  donde ln es la función de logaritmo natural. A grandes rasgos, lo que mide esta función es la diferencia entre dos distribuciones de probabilidad, que en este caso son la distribución real de los datos dada por el conjunto de entrenamiento y la distribución predicha por el modelo en su estado actual.

  Para comprender un poco mejor cómo funciona de manera intuitiva, tomemos un ejemplo simple. Supongamos que tenemos una entrada $\boldsymbol{x}'$ cuya etiqueta es $\boldsymbol{y}' = 1$, y que la salida del modelo es $h(\boldsymbol{x}', \boldsymbol{w}') = 0.95$. En primera instancia, vamos a tener que el lado derecho del término de la sumatoria directamente se anula (ya que $1 - \boldsymbol{y}' = 1 - 1 = 0$). Y, para ver qué pasa con el lado izquierdo, notemos que $ln(h(\boldsymbol{x}', \boldsymbol{w}')) = ln(0.95) \approx -0.05$, por lo que multiplicando por el (-1) del comienzo de la fórmula, vamos a tener como resultado 0.05, lo cual es bajo e indica una buena predicción del modelo. Si en cambio la predicción hubiera sido $h(\boldsymbol{x}', \boldsymbol{w}') = 0.25$, tendríamos $ln(h(\boldsymbol{x}', \boldsymbol{w}')) = ln(0.25) \approx -1.39$, dando una pérdida de 1.39.

  Lo que se puede ver es que el lado izquierdo influye cuando la salida esperada para la entrada actual es 1, y el lado derecho lo hará cuando la salida esperada es 0.

En general, se asume que un valor más bajo de $L(\boldsymbol{w})$ indica un mejor desempeño del modelo, y por lo tanto el objetivo del proceso de entrenamiento se convierte en **encontrar los parámetros $\boldsymbol{w}^*$ que minimicen la función de pérdida** (en los datos del conjunto de entrenamiento):

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w})$$

donde el operador $\arg\min_\alpha g(\alpha)$ da el valor de α que minimiza $g(\alpha)$.

Si luego de la minimización la pérdida es baja, quiere decir que hemos encontrado parámetros que hacen que el modelo prediga precisamente las salidas de entrenamiento $\boldsymbol{y}_i$ a partir de los entradas $\boldsymbol{x}_i$ (con $i = 1, ..., N$). Sin embargo, como explicamos previamente, la evaluación final del modelo se debe hacer con el conjunto de test, sobre el cual se puede también calcular la pérdida.

La pregunta que surge entonces es cómo hacer para lograr esta minimización. Dependiendo del espacio de funciones y de la función de costo seleccionada, puede que exista una fórmula cerrada para $\boldsymbol{w}^*$, que se calcule analíticamente.

Sin embargo, si la función de costo tiene muchas variables (parámetros) o los modelos son complejos, resulta más útil recurrir a métodos numéricos para aproximar este mínimo.

El algoritmo por defecto que se utiliza para resolver este problema de optimización es el llamado **Descenso por el Gradiente** (DG), y es muy potente ya que se puede aplicar

a una gran variedad de funciones de pérdida [26], siempre que esta sea diferenciable (a continuación veremos por qué).

La idea del DG es ir modificando los parámetros iterativamente hasta eventualmente llegar a un "valle" de la función de pérdida. El algoritmo se basa en que la dirección dada por el gradiente[4] de una función en un punto es la de máximo crecimiento, y por lo tanto su opuesta es la de máximo decrecimiento. En este caso, la función de la cual se calcula el gradiente es la de error, con respecto a los parámetros en $\boldsymbol{w}$.

Concretamente, se empieza con un vector de parámetros $\boldsymbol{w}_0$ con valores arbitrarios que va mejorando gradualmente, tomando un paso a la vez en dirección opuesta al gradiente, con el objetivo de reducir el valor de la función de pérdida, hasta que el algoritmo *converja* a un mínimo (local o global) de la función de pérdida.

Un hiperparámetro[5] determinante en el algoritmo del DG es el tamaño de los pasos, llamado **tasa de aprendizaje**, que denotaremos con la letra η y supondremos por el momento que se mantiene constante en todo el proceso. Si el valor η es muy pequeño, entonces el algoritmo va a tener que realizar muchas iteraciones para converger [3]. Si en cambio el valor es muy alto, entonces puede ser que vayamos "saltando" de un lado a otro de un valle de la función, haciendo que el algoritmo diverja [3].

De esta forma, la regla del DG está dada por:

$$\boldsymbol{w}^{(p+1)} = \boldsymbol{w}^{(p)} - \eta \nabla L(\boldsymbol{w}^{(p)})$$

donde p indica el número de iteración actual, y $\nabla L(\boldsymbol{w}^{(p)})$ es el gradiente de la función de error con respecto a los pesos del modelo, evaluado en los valores de los pesos de la iteración actual.

Algo importante a notar es que en la forma que lo presentamos hasta ahora, el cómputo del gradiente de la función de peso involucra recorrer todas las muestras del conjunto de entrenamiento, lo cual computacionalmente es "pesado" y puede demorar el tiempo de entrenamiento. Es decir, en cada iteración, el cálculo del gradiente requiere evaluar la contribución de cada una de las muestras, lo que puede ser caro cuando N es muy grande. Para mitigar este problema, existen variantes del DG que buscan aproximar el gradiente

[4]Dada una función g de varias variables, es decir, $g(v_1, v_2, ..., v_M)$, el **gradiente** de g es el vector formado por las derivadas parciales de g con respecto a cada uno de sus parámetros:

$$\nabla g = \left(\frac{\partial g}{\partial v_1}, \frac{\partial g}{\partial v_2}, \ldots, \frac{\partial g}{\partial v_M} \right)$$

donde la notación $\frac{\partial g}{\partial v_i}$ representa la **derivada parcial** de g con respecto a v_i, que mide cómo cambia g cuando se varía v_i mientras se mantienen constantes las demás variables. Algo a notar es que el vector resultante es un vector de funciones, y cuando se lo evalúa en un punto, da la dirección de mayor crecimiento de g desde ese punto.

[5]Hablaremos más adelante sobre qué es un hiperparámetro, pero lo importante es que no es un parámetro que se aprende, como lo son los contenidos en el vector $\boldsymbol{w}$.

utilizando subconjuntos del conjunto de entrenamiento. Retomaremos este tema en la proxima sección de Redes Neuronales.

Hiperparámetros

La mayoría de los algoritmos de ML tienen **hiperparámetros**, que son ajustes que permiten controlar tanto la capacidad del modelo como el proceso de entrenamiento. Los valores de los hiperparámetros no son aprendidos por el algoritmo [5], sino que se fijan de antemano y no se modifican a lo largo del entrenamiento.

Con lo visto hasta ahora, podemos pensar en dos ejemplos: el grado del polinomio con el cual ajustar los datos, y la tasa de aprendizaje del DG. En la sección siguiente, veremos que existen varios más.

Los hiperparámetros pueden contribuir a que el modelo mejore su desempeño, por lo que surge el problema de cómo hacer para encontrar el valor óptimo de cada uno. La respuesta a esto es un proceso llamado **búsqueda u optimización de hiperparámetros**.

Esta búsqueda se hace de forma empírica: se entrenan distintos modelos con diferentes hiperparámetros sobre el mismo conjunto de entrenamiento, se mide el desempeño de cada uno de ellos, y se elige el mejor. Ahora bien, el desempeño no se mide sobre el conjunto de test, ya que esto admitiría la posibilidad que los hiperparámetros elegidos funcionen bien para ese conjunto, pero no permitan al modelo generalizar a nuevos datos [18].

Por lo tanto, una técnica para realizar correctamente esta búsqueda consiste en construir una tercera partición del conjunto de datos, además de la de entrenamiento y la de test, llamada **conjunto de validación**. De esta forma, en la búsqueda de hiperparámetros, el modelo se sigue entrenando con el conjunto de entrenamiento, pero su performance se mide en el de validación, dejando al de test como si no existiese. Una vez que se han elegido los mejores hiperparámetros, se entrena el modelo con el conjunto de entrenamiento y se mide su capacidad real de generalización sobre el conjunto de test.

Otra estrategia para llevar a cabo esta optimización se denomina **validación cruzada** (*cross-validation*). Consiste en dividir el conjunto de entrenamiento en k partes (disjuntas), y luego entrenar el modelo k veces, cada vez utilizando $k-1$ partes para el entrenamiento y la parte restante para la validación. En este caso, habiendo seleccionado una métrica particular, el rendimiento del modelo con ciertos hiperparámetros va a estar dado por el promedio del puntaje obtenido en cada partición. Al igual que antes, una vez que se han elegido los mejores hiperparámetros, se entrena el modelo con el conjunto de entrenamiento (sin particionar) y se mide su capacidad real de generalización sobre el conjunto de test.

Algo a tener en cuenta es que aunque el espacio de hiperparámetros suele ser más pequeño que el de los parámetros del modelo en sí, puede seguir siendo lo suficientemente grande como para probar cada combinación de manera exhaustiva [18]. Ante esto, existen algoritmos de optimización de hiperparámetros que eligen inteligentemente qué combinaciones ir probando, con base en resultados anteriores [18]. Sin embargo, cabe notar que

esta parte del entrenamiento de un modelo es una de las más costosas computacionalmente y en tiempo.

Resumen

Hasta aquí hemos hablado de cómo está compuesto prácticamente cualquier algoritmo de Aprendizaje Supervisado. Los distintos elementos presentes son:

- Un **conjunto de entrenamiento**, que contiene ejemplos de entrada-salida.
- Un **conjunto de test**, que servirá para evaluar el desempeño real del modelo, y nos dará una noción de su capacidad de **generalización** una vez entrenado.
- Un **espacio de funciones**, que determina la forma de la función h y los parámetros $\boldsymbol{w}$.
- Una **función de pérdida** $L(\boldsymbol{w})$, que mide el desempeño del modelo y ayuda a guiar el entrenamiento.
- Un **algoritmo de optimización**, que busca minimizar la función de pérdida, siendo el más común el Descenso por el Gradiente.

Con esto, el objetivo es encontrar los parámetros $\boldsymbol{w}^*$ que minimicen la función de pérdida en el conjunto de entrenamiento. Para esto, se comienza con pesos $\boldsymbol{w}$ arbitarios y, haciendo uso del algoritmo de optimización, se los va modificando iterativamente hasta llegar a un mínimo (local o global) de $L(\boldsymbol{w})$.

Se trabaja sobre la hipótesis que minimizando el error en el conjunto de entrenamiento, el modelo tendrá una buena capacidad de generalización. Es decir, habiendo llegado a $\boldsymbol{w}^*$, se espera que el modelo encontrado tendrá un buen desempeño en el conjunto de test.

Como venimos enfatizando, el principal desafío del ML es hacer que un modelo tenga una buena capacidad de generalización, es decir que actúe de manera correcta ante entradas aún no vistas. Por un lado, esto depende de qué tan representativo es el conjunto de entrenamiento. Pero por otro, también depende de la elección de un espacio de funciones adecuado, cuyos modelos sean lo suficientemente *expresivos* como para capturar la distribución de los datos.

Sin embargo, más allá de la expresividad que otorgue el espacio de funciones y de la representatividad del conjunto de entrenamiento, existe otra variable presente, que es la dimensionalidad de los datos. La capacidad de generalizar se vuelve aún más complicada cuando los datos son de altas dimensiones, es decir cada entrada contiene muchas características.

En este contexto, aparecen los modelos conocidos como **Redes Neuronales**, que no solo permiten describir conjuntos de funciones complejas, expresivas, y no lineales sin la necesidad de tener un conocimiento profundo sobre la estructura subyacente de los datos, sino que también son capaces de trabajar con espacios de alta dimensión. En la sección siguiente, explicaremos el funcionamiento de estos modelos.

2.3. Redes Neuronales Artificiales

Las Redes Neuronales Artificiales (RNAs) son un modelo específico dentro del ML, cuya estructura y funcionamiento estuvieron inspirados inicialmente por el intento de ser modelos computacionales del aprendizaje biológico, es decir modelos de cómo el aprendizaje podría ocurrir en el cerebro [5]. Durante los últimos años, ha sido el área que más desarrollo e impacto ha tenido, principalmente gracias a su versatilidad, potencia y escalabilidad, cualidades que hacen que estos modelos sean capaces de enfrentar problemas grandes y complejos [3], y que sobre todo parecían extremadamente difíciles de resolver, como son el reconocimiento de voz y la detección de objetos.

A continuación, damos un breve contexto histórico y luego ahondamos en su funcionamiento.

2.3.1. Breve Contexto Histórico

Aunque su gran éxito ha sido reciente, lo cierto es que la idea de las RNAs data desde 1943, cuando Warren McCulloch y Walter Pitts presentaron en su artículo *A Logical Calculus of Ideas Immanent of Nervous Activity* [10] un modelo computacional simplificado utilizando lógica proposicional acerca de cómo funcionan las neuronas de cerebros animales en conjunto para llevar a cabo cómputos complejos [3]. Presentaron una versión muy simplificada de la neurona biológica, que solamente tenía una o más entradas binarias y una salida binaria.

Posteriormente, en 1958, Frank Rosenblatt presentó una de las formas o *arquitecturas* más simples de RNAs: el "Perceptrón" [23]. Este servía principalmente para resolver problemas de clasificación en donde los datos son linealmente separables. Sin emabrgo, su aporte más notorio fue que definió un algoritmo para el entrenamiento del Perceptrón que le permitía mejorar automáticamente sus parámetros internos para poder llegar a la solución óptima.

Más tarde, se descubrió que los problemas que no podían ser resueltos por el Perceptrón, sí podían ser resueltos "apilando" múltiples perceptrones, lo cual llevó a la invención del "Perceptrón Multicapa" (PMC), también conocido actualmente como "Red Neuronal de Propagación Directa"[6](del inglés *Feedforward Neural Networks*, FFNN) [5], las cuales conforman el punto de partida de las redes neuronales actuales.

Para explicar la idea y los elementos presentes detrás de estos algoritmos, tomaremos como referencia las redes neuronales Feedforward, y en particular las llamadas "totalmente conectadas" (*fully connected*), que definiremos a continuación.

[6]Si bien estos términos se suelen usar indistintamente, la realidad es que las redes neuronales feedforward tienen algunas diferencias con respecto al Perceptrón Multicapa.

2.3.2. Red Neuronal Feedforward

Como dijimos anteriormente, las redes neuronales son un modelo particular de Aprendizaje Automático. Aquí, las funciones hipótesis se caracterizan por incorporar **no linealidad** y toman la forma de circuitos algebraicos complejos con conexiones que pueden tener diferentes "intensidades" [26]. La idea principal en estos circuitos es que el "camino" recorrido al realizar el cómputo tenga varios pasos como para permitir que las variables de entrada puedan interactuar de formas complejas. Esto hace que sean lo suficientemente expresivos como para poder capturar la complejidad de los datos del mundo real [26].

Más concretamente, estos modelos son llamados *redes* porque el espacio de funciones que proveen está formado en realidad por la composición de varias funciones [5]. Por ejemplo, podríamos tener la composición de tres funciones $f^{(1)}$, $f^{(2)}$ y $f^{(3)}$ para formar la siguiente función:

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))) \tag{2.13}$$

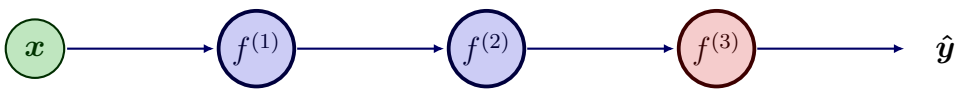
donde $\boldsymbol{x}$ es un vector de dimensión n de números reales: $\boldsymbol{x} = (x_1, x_2, ..., x_n)$, es decir $\boldsymbol{x} \in \mathbb{R}^n$.

Usualmente, se dice que las redes están organizadas en **capas**. De esta forma, en la ecuación anterior, a $\boldsymbol{x}$ se la conoce como **capa de entrada**, a las funciones $f^{(1)}$ y $f^{(2)}$ como **capa ocultas o intermedias**, y a $f^{(3)}$, que es la que produce el resultado final, como **capa de salida**. La longitud de esta cadena de funciones es la que va a dar la **profundidad** de la red.

Las RNAs tienen una capa de entrada y una de salida, pero el número de capas ocultas depende de quien la diseñe. Cuando tienen una capa oculta, se las llama "superficiales" (o poco profundas, del inglés *shallow*), y cuando tienen más de una, *profundas*. Es por esto que al hablar de redes neuronales, muchas veces se hace referencia al término **Aprendizaje Profundo**.

Recordemos que uno de los elementos presentes en el Aprendizaje Supervisado es el conjunto de entrenamiento, compuesto por pares de entrada-etiqueta. Ahora bien, lo interesante de estos modelos es que si bien este conjunto especifica qué tiene que producir la capa de salida ante cada entrada particular, no determina cuál debe ser el comportamiento de las otras capas [5]. En cambio, es el algoritmo de aprendizaje el que tiene que decidir cómo usarlas para lograr una buena aproximación de la función desconocida.

Una forma muy común y más intuitiva de pensar estos modelos es a través de grafos dirigidos cuyas flechas describen cómo están compuestas las funciones y cómo fluye la información a través de ellas. Si hay una flecha que une a dos nodos, diremos que están "conectados". Por ejemplo, la Ecuación 2.13 se representaría de la siguiente manera:



En este caso, la entrada $\boldsymbol{x}$ va a la función $f^{(1)}$, la salida de $f^{(1)}(\boldsymbol{x})$ va a $f^{(2)}$, y la salida de $f^{(2)}(f^{(1)}(\boldsymbol{x}))$ va directamente a $f^{(3)}$ para de esa forma producir el resultado final $\hat{\boldsymbol{y}} = f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$.

Es justamente el comportamiento anterior el que caracteriza a las redes neuronales de tipo ***Feedforward***: los datos y resultados fluyen en una sola dirección; cada nodo computa su resultado y se lo pasa a su sucesor (o sucesores, como veremos más adelante). En el grafo, esta situación se refleja en el hecho que no hay ciclos, por lo que las redes de este tipo se representan por medio de grafos dirigidos y acíclicos.

Ahora bien, ¿qué es exactamente una capa? En la terminología de redes neuronales, una capa es un conjunto de **unidades** o, tomando en cuenta su inspiración biológica, **neuronas**, que actúan en paralelo. Cada unidad representa una función que toma un vector y retorna un escalar, y se asemejan a las neuronas biológicas en el sentido que reciben entradas (o estímulos) de otras unidades y computan su propio valor de activación [5].

De esta forma, la capa de entrada va a tener tantas neuronas como la dimensión de los datos de entrada ($\boldsymbol{x}$). Es decir, si $\boldsymbol{x}$ es de dimensión n, entonces la capa de entrada va a tener n unidades. Sin embargo, la cantidad de neuronas de cada capa oculta depende del diseño de la red, y la de la capa de salida depende sobre todo del problema que se esté tratando de resolver. Si se trata por ejemplo de un problema de clasificación, entonces la capa de salida va a tener en general tantas neuronas como categorías existan en el dominio del problema.

Teniendo el concepto de neuronas, podemos introducir el de redes **totalmente conectadas** (*fully connected*), que son aquellas en las que cada unidad de una capa se conecta con todas las de la capa siguiente, "pasándole" su valor computado a todas ellas.

Con esto en mente, podemos concretizar un poco más el ejemplo con el que venimos trabajando suponiendo que $\boldsymbol{x}$ es un vector de dimensión 3, las capas ocultas dadas por $f^{(1)}$ y $f^{(2)}$ tienen 4 y 3 neuronas respectivamente y la capa de salida tiene 2. Así, suponiendo que nuestra red es fully connected, nuestro grafo resultaría en el de la Figura 2.2, donde el superíndice de cada nodo indica el número de capa y el subíndice hace referencia al número de neurona en esa capa.

A partir de la Figura 2.2, se puede ver que cada neurona de la capa de entrada representa un elemento del vector de entrada, pero las neuronas de tanto la capa oculta como la de salida reciben las salidas de las neuronas de la capa anterior. Veamos entonces qué hace concretamente una neurona o unidad.

Una neurona simplemente calcula una suma pesada de sus entradas, provenientes de las unidades de la capa anterior, y luego aplica una función **no lineal** para producir su salida. Esta función se denomina **función de activación**, y el hecho que sea no lineal es importante ya que de no ser así, cualquier composición de unidades podría representarse mediante una función lineal [26]. Como mencionamos anteriormente, es justamente esta no linealidad lo que permite a estos modelos representar funciones arbitrarias [26] y complejas. En general, se asume que todas las neuronas de una capa tienen la misma

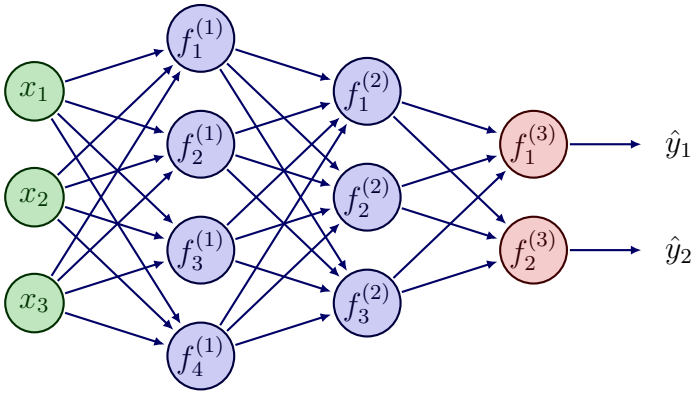


Figura 2.2: Red Neuronal de tipo Feedforward y totalmente conectada, con la capa de entrada formada por 3 neuronas, dos capas ocultas, cada una formada por 4 y 3 neuronas respectivamente, y la capa de salida, formada por dos neuronas. En este caso, la red "acepta" entradas $\boldsymbol{x} \in \mathbb{R}^3$ y produce salidas $\hat{\boldsymbol{y}} \in \mathbb{R}^2$. Adaptado de [13].

función de activación, pero puede ocurrir que diferentes capas tengan diferentes funciones de activación.

Más precisamente, una función de activación es una función no lineal que toma cualquier valor real como entrada y produce como resultado un número en un determinado rango. Algunas funciones de activación comunes son las siguientes:

- **Sigmoide**: produce un valor entre 0 y 1. Es por ello que se suele usar en problemas de clasificación binaria para que una única neurona en la capa de salida represente la probabilidad de que la entrada pertenezca a la clase "positiva" (la que corresponde a la etiqueta 1).

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

- **ReLU** (abreviatura de *Rectified Linear Unit*): produce un valor entre 0 e ∞. Se utiliza mayormente en las neuronas de capas ocultas.

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

- **Tangente hiperbólica**: produce un valor entre -1 y 1. Lo particular de esta es que mantiene el signo de la entrada.

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

Actualmente, la más utilizada es la ReLU y variantes de ella.

Para formalizar el cómputo de una neurona, es necesario introducir cierta notación. Denotaremos con:

- $s_j^{(k)}$ a la salida de la unidad j de la capa k
- $a_j^{(k)}$ a la función no lineal de la unidad j de la capa k
- $w_{i,j}^{(k)}$ la intensidad o **peso** de la conexión entre la neurona i de la capa k y la j de la capa $(k+1)$,

tenemos que:

$$s_j^{(k)} = a_j^{(k)} \left(\sum_i w_{i,j}^{(k-1)} s_i^{(k-1)} \right) \tag{2.14}$$

donde el ínidice i de la sumatoria va a recorrer todas las neuronas de la capa anterior.

Volviendo a la Fórmula 2.14, si en nuestro ejemplo tomamos la neurona $f_1^{(2)}$, tenemos que su salida estará dada por:

$$\begin{aligned} s_1^{(2)} &= a_1^{(2)} \left(\sum_{i=1}^{3} w_{i,1}^{(1)} s_i^{(1)} \right) \\ &= a_1^{(2)} \left(w_{1,1}^{(1)} s_1^{(1)} + w_{2,1}^{(1)} s_2^{(1)} + w_{3,1}^{(1)} s_3^{(1)} \right) \end{aligned}$$

En las redes, se estipula que cada unidad tiene una entrada extra desde una neurona *dummy* de la capa anterior, para la cual utilizaremos el subíndice 0. El valor de salida de esta neurona se fija en 1, y el peso asociado con una neurona j de una capa k será $w_{0,j}^{(k)}$. Este peso se suele llamar ***bias***[7] y permite que la entrada a dicha neurona sea distinta de 0 incluso cuando todas las salidas de la capa anterior sean 0 [26]. Agregándolo, podemos escribir la Ecuación 2.14 de forma vectorizada:

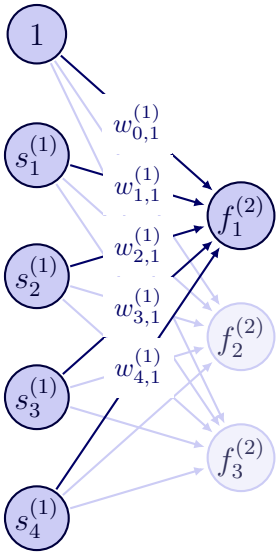
$$s_j^{(k)} = a_j^{(k)} \left(\boldsymbol{w}_j^{(k-1)} \left(\boldsymbol{s}_j^{(k-1)} \right)^T \right) \tag{2.15}$$

donde:

- $\boldsymbol{w}_j^{(k-1)}$ es el vector de todos los pesos que salen de las neuronas de la capa $(k-1)$ y se dirigen a la unidad j de la capa k (incluyendo $w_{0,j}$)
- $\boldsymbol{s}_j^{(k-1)}$ es el vector de todas las salidas de la capa anterior que se dirigen a la unidad j de la capa k (incluyendo el 1 fijo de la neurona dummy).

Con esta notación, si tomamos nuevamente a $f_1^{(2)}$, los vectores involucrados van a ser $\boldsymbol{w}_1^{(1)}$ y $\boldsymbol{s}_1^{(1)}$, dados en este caso por:

$$\boldsymbol{w}_1^{(1)} = (w_{0,1}^{(1)}, w_{1,1}^{(1)}, w_{2,1}^{(1)}, w_{3,1}^{(1)}, w_{4,1}^{(1)}),\ \boldsymbol{s}_1^{(1)} = (1, s_1^{(1)}, s_2^{(1)}, s_3^{(1)}, s_4^{(1)})$$



$$= a_1^{(2)}\left(w_{0,1}^{(1)} + w_{1,1}s_1^{(1)} + w_{2,1}s_2^{(1)} + w_{3,1}s_3^{(1)} + w_{4,1}s_4^{(1)}\right)$$

$$= a_1^{(2)}\left(\begin{bmatrix} w_{0,1}^{(1)} & w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{3,1}^{(1)} & w_{4,1}^{(1)} \end{bmatrix} \begin{bmatrix} 1 \\ s_1^{(1)} \\ s_2^{(1)} \\ s_3^{(1)} \\ s_4^{(1)} \end{bmatrix}\right)$$

Figura 2.3: Entradas a la neurona $f_1^{(2)}$, junto con sus pesos asociados. Se incluyen tanto la neurona dummy de la capa anterior como su bias. En este caso, se usan indistintamente las letras s y f para denotar a las neuronas. Adaptado de [13].

Esto se puede ver mejor gráficamente haciendo foco en dicha neurona, como lo ilustra la Figura 2.3.

Así como utilizamos vectores para describir el cómputo de una neurona, podemos emplear matrices para describir el comportamiento de toda una capa. Para ello, tomemos la siguiente notación, para lo cual resulta conveniente fijar una capa k, que tiene m neuronas:

- $\boldsymbol{s}^{(k)}$ es el vector columna formado por las salidas de la capa k. Es decir: $\boldsymbol{s}^{(k)} = (s_0^{(k)}, s_1^{(k)}, ..., s_m^{(k)})^T = (1, s_1^{(k)}, ..., s_m^{(k)})^T$, de dimensión $m \times 1$.
- $\boldsymbol{a}^{(k)}$ es la función de activación de la capa k, con una aplicación elemento a elemento. Es decir: $\boldsymbol{a}^{(k)}(x_1, x_2, ..., x_m) = (a^{(k)}(x_1), a^{(k)}(x_2), ..., a^{(k)}(x_m))$, donde $a^{(k)}$ es la función de activación de todas las neuronas de la capa k con aplicación a un número.
- $\boldsymbol{W}^{(k)}$ es la matriz de pesos que salen de la capa k. Cada fila j de esta matriz corresponde a los pesos que sale de todas las neuronas de la capa k (incluyendo el bias) y se dirigen a la neurona j de la capa $(k+1)$. Es decir cada fila es $\boldsymbol{w}_j^{(k)}$ con $j = 1, ..., n$, y n la cantidad de neuronas de la capa $(k+1)$ (sin contar la dummy, que "aparece después"). Así, esta matriz tiene dimensiones $n \times m$.

[7] En la bibliografía sobre redes neuronales, también se suele presentar a este peso como un elemento "aparte" de la neurona, no como un peso extra, y se lo denota como $b_j^{(k)}$.

Con esto, tenemos que la salida de una capa k está dada por:

$$\boldsymbol{s}^{(k)} = \boldsymbol{a}^{(k)}\left(\boldsymbol{W}^{(k-1)}\boldsymbol{s}^{(k-1)}\right)$$

Con todo esto en mente, veamos cuál es la función que describe la red neuronal presentada como ejemplo:

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \boldsymbol{s}^{(3)} \\
&= \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{s}^{(2)}\right) \\
&= \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{s}^{(1)}\right)\right) \\
&= \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\boldsymbol{W}^{(0)}\boldsymbol{s}^{(0)}\right)\right)\right) \\
&= \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\boldsymbol{W}^{(0)}\boldsymbol{x}^{T}\right)\right)\right)
\end{aligned}$$

Y si queremos profundizar aún más esta ecuación para ver dónde aparece cada peso:

$$\begin{aligned}
\hat{\boldsymbol{y}} = \; & \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\boldsymbol{W}^{(0)}\boldsymbol{x}^{T}\right)\right)\right) \\
= \; & \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\boldsymbol{W}^{(0)}\begin{bmatrix}1\\x_1\\x_2\\x_3\end{bmatrix}\right)\right)\right) \\
= \; & \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\begin{bmatrix}w_{0,1}^{(0)} & w_{1,1}^{(0)} & w_{2,1}^{(0)} & w_{3,1}^{(0)}\\ w_{0,2}^{(0)} & w_{1,2}^{(0)} & w_{2,2}^{(0)} & w_{3,2}^{(0)}\\ w_{0,3}^{(0)} & w_{1,3}^{(0)} & w_{2,3}^{(0)} & w_{3,3}^{(0)}\\ w_{0,4}^{(0)} & w_{1,4}^{(0)} & w_{2,4}^{(0)} & w_{3,4}^{(0)}\end{bmatrix}\begin{bmatrix}1\\x_1\\x_2\\x_3\end{bmatrix}\right)\right)\right) \\
= \; & \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}\left(\begin{bmatrix}w_{0,1}^{(0)}+w_{1,1}^{(0)}x_1+w_{2,1}^{(0)}x_2+w_{3,1}^{(0)}x_3\\ w_{0,2}^{(0)}+w_{1,2}^{(0)}x_1+w_{2,2}^{(0)}x_2+w_{3,2}^{(0)}x_3\\ w_{0,3}^{(0)}+w_{1,3}^{(0)}x_1+w_{2,3}^{(0)}x_2+w_{3,3}^{(0)}x_3\\ w_{0,4}^{(0)}+w_{1,4}^{(0)}x_1+w_{2,4}^{(0)}x_2+w_{3,4}^{(0)}x_3\end{bmatrix}\right)\right)\right) \\
= \; & \boldsymbol{a}^{(3)}\left(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}\left(\boldsymbol{W}^{(1)}\begin{bmatrix}a^{(1)}\left(w_{0,1}^{(0)}+w_{1,1}^{(0)}x_1+w_{2,1}^{(0)}x_2+w_{3,1}^{(0)}x_3\right)\\ a^{(1)}\left(w_{0,2}^{(0)}+w_{1,2}^{(0)}x_1+w_{2,2}^{(0)}x_2+w_{3,2}^{(0)}x_3\right)\\ a^{(1)}\left(w_{0,3}^{(0)}+w_{1,3}^{(0)}x_1+w_{2,3}^{(0)}x_2+w_{3,3}^{(0)}x_3\right)\\ a^{(1)}\left(w_{0,4}^{(0)}+w_{1,4}^{(0)}x_1+w_{2,4}^{(0)}x_2+w_{3,4}^{(0)}x_3\right)\end{bmatrix}\right)\right) \\
= \; & (\dots)
\end{aligned}$$

Las librerías que implementan estos modelos utilizan esta notación matricial ya que suelen estar optimizadas para cálculos con matrices.

Algo a notar es que en las redes neuronales, aparecen nuevos hiperparámetros, que ya no tienen tanto que ver con el proceso de entrenamiento en sí, sino más bien con el diseño y la capacidad de la red, como son la cantidad de capas ocultas, la cantidad de neuronas en cada capa oculta, e incluso la función de activación de cada capa. Para todos estos, se puede encontrar el valor óptimo utilizando las técnicas mencionadas en la sección anterior.

Entrenamiento

En el caso de las redes neuronales, los parámetros a optimizar van a ser las intensidades de las conexiones entre las neuronas, que también venimos llamando pesos. Los optimizadores que se utilizan actualmente se basan en la regla del DG, aunque con algunas mejoras.

Como mencionamos en la sección anterior, lo costoso de la regla clásica del DG presentada hasta el momento es que requiere calcular el gradiente de la función de pérdida considerando *todas* las muestras del conjunto de entrenamiento. Para reducir esta carga, en la práctica se recurre a una aproximación: en lugar de usar todas las muestras, se selecciona aleatoriamente un subconjunto de ellas, llamado **lote** (*batch*), y se calcula el gradiente usando únicamente ese lote. Luego, los pesos se actualizan con base en esta estimación parcial. Luego, se actualizan los pesos en base a este estimación. Cabe aclarar que el tamaño de lote, es decir la cantidad de ejemplos que se eligen cada vez, se mantiene fijo en todo el entrenamiento y constituye un hiperparámetro. Esta técnica suele llamarse optimización por **mini-lotes** o **estocástica**, ya que introduce cierto grado de aletoriedad en el cálculo del gradiente.

Otra mejora aplicada sobre el método del DG estándar, diseñada para acelerar el aprendizaje, es la técnica conocida como **momentum**. A grandes rasgos, consiste en actualizar los pesos mirando no solo el gradiente de la iteración actual, sino también dándole un peso a la acumulación de gradientes de las iteraciones anteriores. Esto permite sobre todo acelerar las actualizaciones cuando se encuentran sucesivos gradientes que apuntan en la misma dirección [5]. Un optimizador ampliamente utilizado actualmente, y que es el que empleamos en este trabajo, es **Adam** [8]. Este combina momentum con una tasa de aprendizaje adaptativa para cada parámetro.

Para entender cómo progresa el entrenamiento, es útil definir el concepto de definir el concepto de **época**. Una época se refiere al proceso completo en el cual el modelo se entrena utilizando **todos** los datos disponibles en el conjunto de entrenamiento una vez. A grandes rasgos, los pasos involucrados en una época son los siguientes:

1. Mezclar el conjunto de entrenamiento. Este paso en realidad es opcional pero evita que el modelo aprenda patrones no deseados debido al orden de los datos.
2. Seleccionar un lote del tamaño predefinido del conjunto de datos de entrenamiento.
3. Para cada ejemplo del lote:

   a) Computar la predicción del modelo.
   b) Calcular la pérdida.

4. Promediar el error a lo largo de todos los ejemplos del lote. Este valor escalar no se utiliza directamente en el entrenamiento, pero se emplea comúnmente para monitorear el progreso del aprendizaje.

5. Calcular el gradiente utilizando solamente las muestras del lote.

6. Actualizar los pesos.

7. Volver a 2.

8. Una vez que se han recorrido todos los lotes, finaliza la época.

Ahora bien, un paso fundamental en el entrenamiento de las redes neuronales y del que vale la pena profundizar es el del cálculo del gradiente. De hecho, no encontrar una solución eficiente para lograrlo detuvo el avance de estos modelos durante varios años. El algoritmo que vino a dar respuesta y que es el utilizado hasta hoy es conocido como **retropropagación** (del inglés *backpropagation*) y fue presentado en el año 1986 [25].

El backpropagation se basa fundamentalmente en la regla de la cadena para derivadas de funciones compuestas y se divide en dos etapas: primero, el paso hacia adelante (*forward pass*) y luego, el paso hacia atras (*backward pass*). Para entender estas etapas, supondremos que solamente una entrada es provista a la red.

En el forward pass, la red simplemente lleva a cabo una predicción para la entrada, realizando todo el cómputo intermedio necesario para llegar a producir una salida.

El backward pass comienza por calcular el error cometido por la red para la entrada dada, y consiste en propagar este error desde la capa de salida hasta la de entrada, midiendo la contribución al error de cada conexión [3]. Este paso se basa fuertemente en la idea que un pequeño cambio en uno de los pesos causa un efecto cadena sobre el cómputo restante de la red [18]. Para verlo, supongamos que tenemos una red neuronal con tres capas ocultas, que denotaremos con $\boldsymbol{h}^{(1)}$, $\boldsymbol{h}^{(2)}$ y $\boldsymbol{h}^{(3)}$ (la cantidad de neuronas en cada capa es irrelevante), y veamos cómo computaríamos el efecto de un cambio en un peso [18]:

- Para calcular cómo un cambio en un peso que se dirige a $\boldsymbol{h}^{(3)}$ modifica el valor de la pérdida, necesitamos saber (i) cómo un cambio en $\boldsymbol{h}^{(3)}$ modifica la salida $\boldsymbol{f}$ del modelo y (ii) cómo un cambio en la salida modifica la pérdida.

- Para calcular cómo un cambio en un peso que se dirige a $\boldsymbol{h}^{(2)}$ modifica el valor de la pérdida, necesitamos saber (i) cómo un cambio en $\boldsymbol{h}^{(2)}$ afecta a $\boldsymbol{h}^{(3)}$, (ii) cómo un cambio en $\boldsymbol{h}^{(3)}$ modifica la salida $\boldsymbol{f}$ del modelo y (iii) cómo un cambio en la salida modifica la pérdida.

- Para calcular cómo un cambio en un peso que se dirige a $\boldsymbol{h}^{(1)}$ modifica el valor de la pérdida, necesitamos saber (i) cómo un cambio en $\boldsymbol{h}^{(1)}$ afecta a $\boldsymbol{h}^{(2)}$, (ii) cómo un cambio en $\boldsymbol{h}^{(2)}$ afecta a $\boldsymbol{h}^{(3)}$, (iii) cómo un cambio en $\boldsymbol{h}^{(3)}$ modifica la salida $\boldsymbol{f}$ del modelo y (iv) cómo un cambio en la salida modifica la pérdida.

Mientras nos movemos hacia las primeras capas de la red, vemos que la mayoría de los términos que se necesitan ya han sido calculados en pasos anteriores, por lo que no es necesario recomputarlos. Justamente calcular los efectos de los cambios - que son básicamente las derivadas parciales - de esta manera es conocido como el backward pass.

El paso final del backpropagation es actualizar los pesos con los gradientes calculados en el backward pass y utilizando la regla del DG.

En las siguientes secciones, hablaremos sobre dos tipos particulares de redes neuronales de las que hacemos uso en este trabajo: las Redes Neuronales Convolucionales y las Redes Neuronales Recurrentes. Cada una fue diseñada originalmente para trabajar con un tipo específico de datos. Las convolucionales son ideales para procesar datos estructurados en forma de grilla, mientras que las recurrentes son adecuadas para secuencias temporales. Presentaremos la intuición sobre detrás de ellas y nos concentraremos en su aplicación sobre series de tiempo, relevante para nuestro trabajo.

2.4. Redes Neuronales Convolucionales

Como mencionamos anteriormente, las Redes Neuronales Convolucionales (RNC) son un tipo especializado de redes neuronales pensadas para el procesamiento de datos que tienen una estructura de grilla [5]. Ejemplos de estos datos son las series de tiempo, que pueden pensarse como una grilla unidimensional y las imágenes, que pueden verse como una grilla bidimiensional de píxeles.

Si bien las RNCs se han usado principalmente para el procesamiento de imágenes, en este trabajo nos enfocaremos en su uso para analizar series de tiempo, en particular unidimensionales. Veremos cómo su funcionamiento permite capturar patrones temporales en los datos.

Sin entrar en detalles[8], una **serie de tiempo univariada o unidimensional** se puede representar como un vector ordenado U de valores reales, en el que cada valor corresponde a una medición en el tiempo sobre un determinado fenómeno:

$$U = (x_1, x_2, ..., x_N)$$

donde N es la dimensión del vector.

Ahora bien, el término *convolucionales* proviene del hecho que lo que caracteriza a

[8]En una sección posterior, hablaremos más en profundidad sobre las series de tiempo unidimensionales.

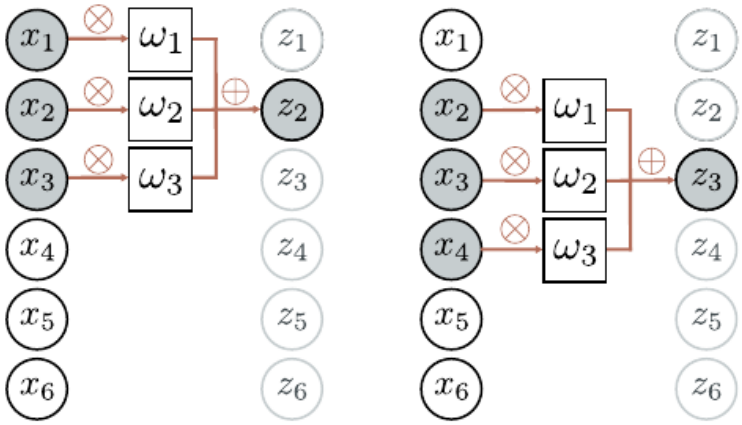


Figura 2.4: Fuente: [18]. Ejemplo de convolución unidimensional con un filtro $\boldsymbol{w}$ de tamaño 3.

estas redes es la utilización de una operación matemática llamada **convolución**[9][5]. Las capas de la red que aplican esta operación son llamadas *capas convolucionales*.

La convolución toma dos argumentos: por un lado, la *entrada* y por otro lado, un *filtro* o *kernel*. En el contexto del ML, ambos son usualmente arreglos multidimensionales de números reales, también comúnmente llamados tensores [5]. Intuitivamente, la operación consiste en "deslizar" el filtro sobre la entrada, y en cada momento calcular el producto escalar. Para verlo más claramente en el caso unidimensional, tomemos un ejemplo.

Dado como entrada a la convolución un vector $\boldsymbol{x}$, si tomamos un filtro de tamaño 3, dado por $\boldsymbol{w} = (w_1, w_2, w_3)^T$, entonces el resultado de la convolución es un vector $\boldsymbol{z}$ en donde cada componente z_i es una suma pesada de las entradas "cercanas":

$$z_i = w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1}$$

En la Figura 2.4, se puede ver cómo se computan z_2 y z_3. Un detalle a notar aquí es que al calcular z_2 y z_3, el filtro "entra" por completo en la entrada, pero si quisiéramos calcular z_1, pareciera que w_1 no puede multiplicarse por nada. Existen alternativas para solucionar este problema, que detallaremos más adelante.

Si generalizamos y tomamos un vector $\boldsymbol{x}$ de tamaño n correspondiente a la entrada y un vector $\boldsymbol{w}$ de tamaño l correspondiente al kernel, entonces el resultado de la convolución es un vector $\boldsymbol{z}$ donde:

$$z_i = \sum_{j=1}^{l} w_j x_{j+i-(l+1)/2} \tag{2.16}$$

[9]En realidad, la operación que está presente en estas redes es una relacionada con la convolución, llamada **correlación cruzada**.

[3]Por conveniencia, el primer elemento de los vectores será indexado en 1, y no en 0.

En otras palabras, para generar cada componente i de la salida, se toma el producto escalar entre el kernel $\boldsymbol{w}$ y una parte de $\boldsymbol{x}$ de largo l centrada en x_i [26].

A partir de aquí, se puede ver que aplicar una convolución resulta en una salida usualmente de menor tamaño que la entrada. La salida de la convolución suele llamarse **mapa de características** (*feature map*) [5], que resalta las áreas de la entrada que son "similares" a la característica que el filtro está tratando de capturar [3]. Para ver esto, tomemos otro ejemplo.

Supongamos que tenemos el filtro $\boldsymbol{w} = (1, 0, -1)$ (de tamaño $l = 3$). Entonces, usando la Ecuación 2.16, veamos qué calcula este filtro en cada posición i de la salida:

$$\begin{aligned} z_i &= \sum_{j=1}^{3} w_j x_{j+i-(3+1)/2} \\ &= \sum_{j=1}^{3} w_j x_{j+i-2} \\ &= w_1 x_{1+i-2} + w_2 x_{2+i-2} + w_3 x_{3+i-2} \\ &= 1 \times x_{i-1} + 0 \times x_i + (-1) \times x_{i+1} \\ &= x_{i-1} - x_{i+1} \end{aligned}$$

O sea, lo que hace es dada una posición i, calcular la diferencia entre el "vecino" izquierdo de i y el derecho. De esta forma, si para un i el valor es positivo, quiere decir que hubo una transición de un valor mayor a uno menor, y si es negativo, viceversa. En este caso, se podría decir que el filtro "detecta" regiones de la entrada en donde hubo un cambio de tendencia.

Con esto, podemos ver que un filtro se encarga de identificar una característica específica en la entrada. Y por lo tanto, diferentes filtros actuando en conjunto podrán detectar diferentes características puntuales.

Ahora que ya tenemos una intuición clara sobre qué hace una convolución, veamos cómo esta operación se traslada a las redes neuronales. Para empezar, lo que va a ocurrir ahora es que los parámetros a optimizar van a ser los pesos involucrados en cada filtro.

Las redes convolucionales son un ejemplo de red Feedforward, ya que la información fluye en una sola dirección. Ahora bien, vimos que en las redes totalmente conectadas, cada neurona de una capa está conectada con todas las de la capa anterior. En cambio, en las capas convolucionales lo que ocurre es que cada neurona está conectada con un subconjunto de neuronas (consecutivas) de la capa anterior, sobre las cuales aplicará el filtro, y que recibe el nombre de **campo receptivo**. Es importante notar que cada neurona de una capa convolucional aplica el mismo filtro, por lo tanto todas ellas comparten los mismos pesos. Esto significa que la cantidad de parámetros a aprender en una red convolucional es mucho menor que en una red totalmente conectada, y es algo que caracteriza a las RNCs. Este comportamiento se puede ver en la Figura 2.5.

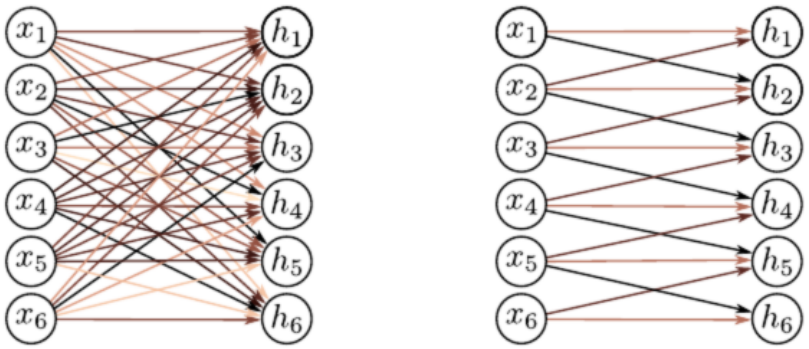


Figura 2.5: Fuente: [18]. Capas de una red fully connected (izquierda) vs capas convolucionales (derecha). Denotamos con x a la capa de la izquierda, y con h a la de la derecha, en ambos casos. En las capas de la fully connected, vemos que cada neurona de la capa x está conectada con todas las neuronas de la capa h, habiendo un total de $6 \times 6 = 36$ pesos entre ellas. En cambio, en la convolucional, asumiendo que se aplica un filtro de tamaño $l = 3$, tenemos que cada neurona de la capa h computa una suma pesada (con los mismos pesos) de $l = 3$ neuronas consecutivas de la capa x.

Con esta idea de campo receptivo, podemos pensar en una red con dos capas convolucionales seguidas y en las que se aplica un filtro de tamaño 3. Lo que va a ocurrir entonces es que las neuronas de la primera capa oculta toman una suma pesada de conjuntos de 3 neuronas consecutivas de la capa de entrada. Luego, las unidades de la segunda capa oculta toman una suma pesada de conjuntos de 3 neuronas consecutivas de la primera capa oculta, que son a su vez sumas pesadas de 3 neuronas de entrada. Por lo tanto, las neuronas de la segunda capa oculta en realidad tienen un campo receptivo de 5 neuronas. De esta forma, el campo receptivo de las unidades en las capas siguientes va aumentando [18]. Esto provoca que la primera capa oculta se concentre en características de bajo nivel, más específicas, luego la siguiente en otras de mayor nivel como resultado de integrar las anteriores, y así sucesivamente [3].

Hasta ahora vimos una capa convolucional que aplica solamente un filtro, lo que nos va a dar como resultado solamente un mapa de características. Sin embargo, una misma capa puede aplicar varios filtros (todos del mismo tamaño), cada uno de los cuales va a generar su propio feature map. Para esto, lo que se hace es darle a la capa una "profundidad" de tantos filtros como querramos aplicar sobre la entrada, obteniendo de esta forma un feature map para cada nivel profundidad, cada uno capturando una característica puntual de la imagen. Así, se suele decir que una capa tiene varios *canales*.

Algo que cabe notar en este punto es que cuando una neurona aplica el filtro correspondiente sobre las de la capa anterior, si esta última tiene varios canales, entonces el filtro es aplicado en todos ellos. Es decir, si por ejemplo la capa anterior tenía C canales y el filtro tiene tamaño l, entonces al aplicar el filtro, ahora vamos a tener una cantidad

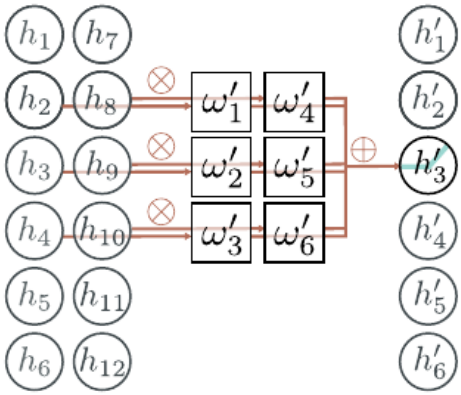


Figura 2.6: Fuente: [18]. La capa de neuronas h_i tiene dos canales: aquel formado por las neuronas h_1 a h_6 y aquel formado por las neuronas h_7 a h_{12}. La capa de neuronas h'_i aplica un filtro de tamaño $l = 3$ sobre cada uno de los canales de la capa anterior. Por lo tanto ahora termina habiendo $2 \times 3 = 6$ pesos entre cada neurona de la capa h'_i y la capa h_i.

de pesos $l \times C$. Esto se puede ver en la Figura 2.6.

Además de la cantidad de filtros en una capa, y su tamaño l, existen otros hiperparámetros de las capas convolucionales:

- El **paso** (*stride*), que representa la distancia entre dos campos receptivos consecutivos. Un stride igual a 1 quiere decir que el filtro se aplica en cada posición de la entrda.
- El **relleno** (*padding*), que controla cuántos valores "fantasma" se agregan artificialmente a los extremos de la entrada de forma tal que el tamaño de la salida de la convolución no sea tanto menor al de la entrada. En la práctica, se usa el 0 para este relleno (*zero-padding*). Con esto, se puede computar el valor de z_1 que presentamos como ejemplo anteriormente.
- La **dilatación** (*dilation*), que determina cuántos 0s se intercalan en los pesos del filtro. Con este parámetro, se puede convertir un filtro de $l = 5$ a uno "dilatado" de tamaño 3 fijando el segundo y cuarto elementos en 0. Esto permite seguir "integrando" la información de una región de la entrada de tamaño 5 pero requiriendo solamente 3 pesos para hacerlo [18].

Otros de los componentes particulares de las RNCs son las capas de *pooling*, cuyo objetivo principal es reducir el tamaño del mapa producido por la convolución [3]. Es decir, se aplican luego de la convolución y lo que hacen es reemplazar valores contiguos presentes en una determinada sección del mapa por una medida resumen de ellos, entre las cuales algunas comúnmente utilizadas son el máximo (*max pooling*) y el promedio (*average pooling*). Esta reducción de tamaño no solo disminuye la cantidad de parámetros sino que también hace a la red más robusta ante desplazamientos de la imagen [3].

Una capa convolucional típica de una RNC se compone de tres etapas: en primer lugar, se producen las convoluciones, que se encargan de producir activaciones lineales; luego, cada una de estas activaciones pasa por una función de activación no lineal; y finalmente se aplica pooling para reducir (aún más) el tamaño de la salida [5]. Y así, una Red Neuronal Convolucional profunda se compone de varias de estas capas.

Ahora bien, en este trabajo nosotros utilizaremos este tipo de redes para resolver la tarea de **clasificación de series de tiempo**, esto es: dada una serie de tiempo como entrada de la red, que esta sea capaz de predecir a qué categoría corresponde. Para ello, lo que se suele hacer en estas redes es agregar, luego de la última capa convolucional, una capa de **aplanado** (*flatten*), la cual no posee pesos aprendibles y simplemente convierte el tensor resultante de la capa anterior en uno unidimensional. Por ejemplo, si esta consiste de k feature maps de longitud n cada uno, entonces la capa de flatten dará como resultado un vector unidimensional de tamaño kn.

El paso anterior es necesario ya que las capas siguientes que se agregan para hacer la clasificación son totalmente conectadas que, como vimos anteriormente, aceptan únicamente entradas unidimensionales. Entonces, lo que ocurre es que el vector de tamaño kn es procesado por las diferentes capas densas hasta llegar a la de salida.

En problemas de clasificación con más de dos categorías, esta última capa suele tener tantas neuronas como clases haya. En cambio, para tareas de clasificación binaria, como en nuestro caso, se utiliza habitualmente una única neurona con una función de activación sigmoide, cuya salida representa la probabilidad de que la secuencia de entrada pertenezca a la clase positiva (aquella asociada con la etiqueta 1).

2.5. Redes Neuronales Recurrentes

Las Redes Neuronales Recurrentes (RNRs) son una familia de redes neuronales pensadas específicamente para trabajar con **datos secuenciales**, en donde el orden y la dinámica importan. Es por esto que han sido y continúan siendo muy útiles en tareas como análisis de series temporales y procesamiento de lenguaje natural. Se emplean por ejemplo para predecir cuáles van a ser los próximos valores en una serie de tiempo (tarea comúnmente conocida como *forecasting*), cuáles van a ser las palabras que continúan una oración, y también para clasificar series de tiempo, que es para lo que las usaremos nosotros.

Una características de las redes feedforward que presentamos hasta el momento es que no tienen memoria. Si quisieramos procesar una secuencia temporal de datos con estas redes, lo que deberíamos hacer es "mostrársela" entera como entrada, perdiendo la dependencia temporal que existe entre las distintas features, que en realidad son los pasos de tiempo de una misma secuencia.

Las RNRs proponen una forma alternativa de trabajar con datos de este tipo. Proce-

san las secuencias iterando sobre los distintos pasos de tiempo manteniendo un **estado interno** o **memoria** que contiene información relacionada con lo que ha visto hasta el momento[10]. A esto lo logran introduciendo cíclos en su grafo, que permiten que las neuronas de la red puedan tomar como entradas sus propias salidas de pasos anteriores. Esto provoca que entradas recibidas en pasos más tempranos afecten la respuesta de la red ante la entrada actual [26] .

Para comprender un poco más este comportamiento, tomemos la más simple de las RNRs, compuesta por una única neurona (oculta) que recibe la entrada correspondiente al tiempo t, produce una salida y se la envía tanto a la capa de salida como a sí misma [3], como se puede ver a la izquierda de la Figura 2.7. Así, en cada paso t, la *neurona "recurrente"* f recibe no solo la entrada x_t sino también su propia salida computada en el paso anterior, f_{t-1}. Esto se hace aún más evidente cuando "desenrollamos" la red a lo largo del tiempo, cuya representación se encuentra a la derecha de la Figura 2.7 y se asemeja a la de una feedforward.

Otro detalle a tener en cuenta en estas redes es que, como se puede ver en la Figura 2.7, en cada paso de tiempo de una misma secuencia, la red genera una salida. Es decir, si tenemos una secuencia de largo τ y nuestra red produce en cada paso de tiempo una salida de tamaño o, entonces al terminar de procesar una secuencia entera, tendremos una salida "total" de tamaño $\tau \times o$. Sin embargo, la salida en cada paso de tiempo contiene información de los pasos de tiempo previos, por lo que en general se suele utilizar la salida producida en el último paso, ya que contiene información sobre toda la secuencia.

Cada neurona realiza lo mismo que en una red feeforward, en el sentido que computan una suma pesada de sus entradas y aplican una función de activación sobre esta. Sin embargo, a partir de la Figura 2.7, podemos notar algunas particularidades con respecto a las redes recurrentes:

- Por un lado, cada neurona tiene un peso extra además del que se aplica sobre la entrada x_t ($w_{x,f}$) y el bias correspondiente ($w_{0_x,f}$). Este peso extra corresponde al que se aplica sobre la salida de la neurona en el paso anterior f_{t-1}, denotado con $w_{f,f}$.

- Por otro lado, la red utiliza los mismos pesos $w_{x,f}$, $w_{f,f}$ y $w_{f,y}$, y biases $w_{0_x,f}$ y $w_{0_f,y}$ en *todos* los pasos de tiempo.

Con esto en cuenta, vamos a tener que el cómputo llevado a cabo por la red en cada paso de tiempo t está dado por:

$$f_t = a_f\left(w_{x,f}x_t + w_{f,f}f_{t-1} + w_{0_x,f}\right) \tag{2.17}$$

$$y_t = a_y\left(w_{f,y}f_t + w_{0_f,y}\right) \tag{2.18}$$

[10]Cabe aclarar que este estado interno se reinicia entre el procesamiento de diferentes secuencias.

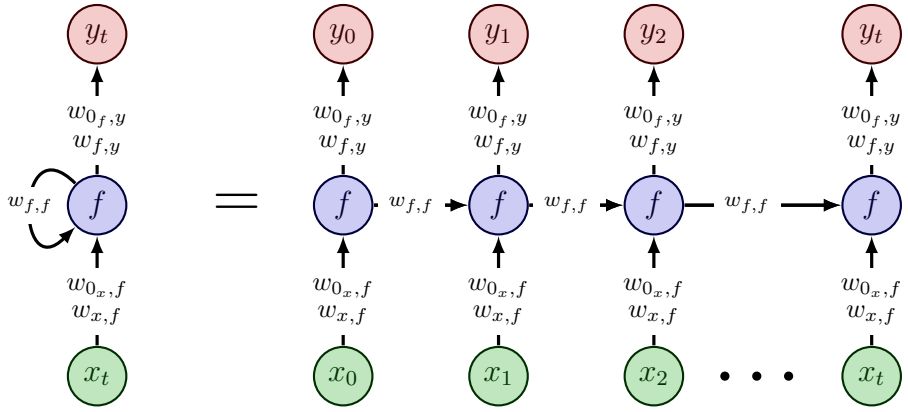


Figura 2.7: A la izquierda, se ve la más simple de las Redes Neuronales Recurrentes, con solamente la capa de entrada, una capa oculta formada por una neurona con su respectiva conexión recurrente, y la capa de salida. A la derecha, se ve la misma red pero "desenrollada" a lo largo del tiempo. Asumiremos por simplicidad que tanto la entrada como la salida en cada momento t son simplemente números reales, es decir $x_t, y_t \in \mathbb{R}\ \forall t$. Denotamos con $w_{x,f}$ al peso que va desde la entrada a la neurona oculta, con $w_{0_x,f}$ al asociado a la neurona dummy con valor fijo en 1 que está en la entrada, con $w_{f,y}$ al que va desde la neurona oculta hasta la capa de salida, con $w_{0_f,y}$ al bias entre la oculta y la de salida, y con $w_{f,f}$ al peso que aplica la neurona sobre su propia salida del paso anterior.

donde a_f denota la función de activación de la capa oculta y a_y la de la capa de salida. Algo que vale la pena aclarar es que para el primer paso de tiempo x_0, lo que sería la salida del paso anterior se establace manualmente, por lo general con el valor 0.

Nos concentremos en la salida de la neurona recurrente en el tiempo t, es decir en f_t, dejando la salida de la red y_t de lado. A partir de las Ecuaciones anteriores, podemos ver lo que venimos explicando: f_t es una función de tanto la entrada en el tiempo actual x_t y de su salida en el paso anterior f_{t-1}. Entonces, si tomamos un $t' > 0$ fijo, vamos a tener que:

- $f_{t'}$ es una función de $x_{t'}$ y de $f_{t'-1}$, pero
- $f_{t'-1}$ es a su vez una función de $x_{t'-1}$ y de $f_{t'-2}$, pero
- $f_{t'-2}$ es a su vez una función de $x_{t'-2}$ y de $f_{t'-3}$,
- y así sucesivamente

Este comportamiento hace que $f_{t'}$ sea una función de todas las entradas vistas desde $t = 0$, constituyendo de esta forma una especie de *memoria*, que se suele llamar **estado oculto** de la neurona. En este caso, como la red solamente tiene una capa oculta con una neurona recurrente, el estado oculto de la neurona coincide con el estado oculto de la red.

De la misma forma en que lo hicimos anteriormente para las redes feedforward, podemos empezar a complejizar esta red agregando varias neuronas recurrentes en la capa oculta, cada una con su propio ciclo, como se puede ver en la Figura 2.8.

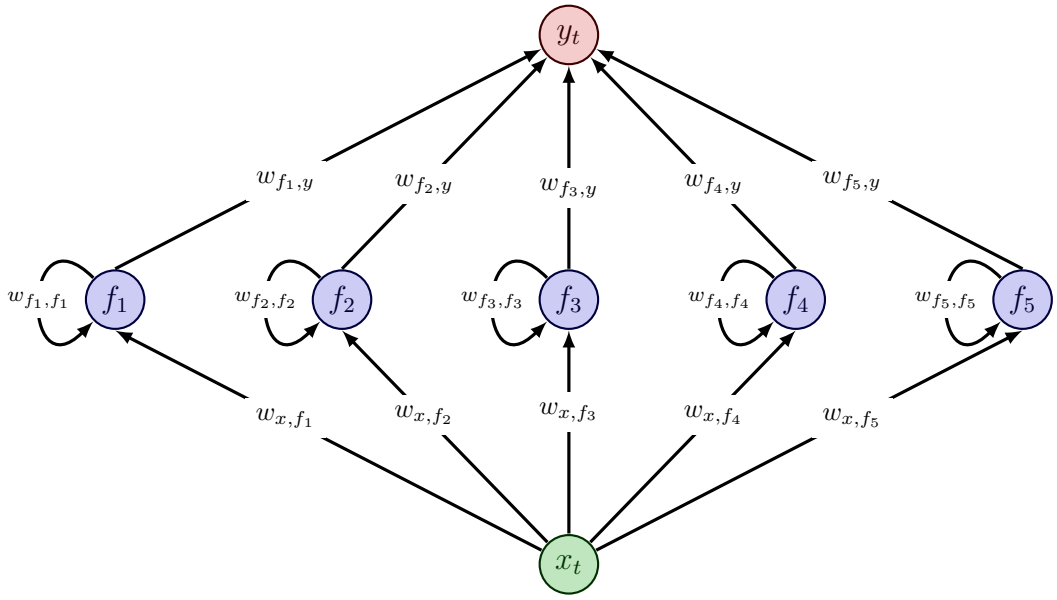


Figura 2.8: Red Neuronal Recurrente con una capa oculta de 5 neuronas recurrentes, cada una con su ciclo. Similarmente al Diagrama anterior, denotamos con w_{x,f_i} al peso que va desde la capa de entrada a la neurona oculta f_i, con w_{f_i,f_i} al peso de cada neurona oculta a sí misma, y con $w_{f_i,y}$ al que va desde la neurona oculta f_i a la capa de salida ($i = 1, 2, 3, 4, 5$). En este caso, omitimos los biases.

Con esto, vamos a tener que el estado oculto de la red va a ser un vector formado por el estado oculto de las 5 neuronas. O sea, si denotamos con f_t al estado oculto de la red en el tiempo t y con f_{i_t} al estado oculto de la neurona oculta i, en el tiempo t vamos a tener que:

$$f_t = (f_{1_t}, f_{2_t}, f_{3_t}, f_{4_t}, f_{5_t})$$

Y también podemos volver a utilizar la notación matricial para describir el comportamiento de la red de manera general:

$$\begin{aligned} f_t &= \boldsymbol{a}_f \left(\mathbf{W}_x x^T + \mathbf{W}_f f_{t-1}^T\right) \\ y_t &= \boldsymbol{a}_y \left(\mathbf{W}_y f_t\right) \end{aligned}$$

donde, siendo k el número de neuronas en la capa oculta y n el tamaño de cada paso de tiempo de la secuencia:

- $\boldsymbol{a}_f$ es la función de activación de la capa oculta, aplicada elemento a elemento, es decir $\boldsymbol{a}_f(z_0, z_1, ..., z_m) = (a_f(z_0), a_f(z_1), ..., a_f(z_m))$.

- $\mathbf{W}_x$ es la matriz de tamaño $k \times (n+1)$ de pesos que salen desde la capa de entrada hasta la capa oculta, incluyendo el bias.
- x^T es el vector transpuesto de la entrada, junto con el 1 del bias, es decir x^T es de tamaño $(n+1) \times 1$.
- $\mathbf{W}_f$ es la matriz diagonal de tamaño $k \times k$ de pesos recurrentes. En cada elemento de la diagonal i se encuentra el peso que va desde la neurona i a sí misma, es decir w_{f_i,f_i}.
- f_{t-1}^T es el vector transpuesto del estado oculto de la red en el paso de tiempo anterior, es decir es de tamaño $k \times 1$.
- $\boldsymbol{a}_y$ es la función de activación de la capa de salida, aplicada elemento a elemento, al igual que $\boldsymbol{a}_t$.
- $\mathbf{W}_y$ es la matriz de tamaño $o \times (k+1)$ de pesos que van desde la capa oculta hasta la capa de salida, incluyendo el bias, donde o es el tamaño de la salida.

Al igual que antes, se pueden agregar más capas de neuronas recurrentes a la red, permitiendo de esta manera que cada capa tenga su propio estado oculto. En este caso, lo que va a ocurrir es que el estado oculto producido por las neuronas de una capa va a ser la entrada de las neuronas de la siguiente capa, estando las capas totalmente conectadas entre sí.

Para entrenar este tipo de redes, el truco está en "desenrollarlas" como vimos anteriormente y luego utilizar el algoritmo de backpropagation presentado en la sección de Redes Neuronales. Esta estrategia es conocida como ***backpropagation through time***.

Ahora bien, estas redes así como las presentamos sufren de un problema conocido como **el problema del gradiente desvaneciente**, que las hace incapaces de capturar dependencias de largo plazo al procesar secuencias de muchos pasos de tiempo.

En general, y no solo en estas redes, el problema del gradiente desvaneciente ocurre cuando, al calcular el gradiente de la función de pérdida con respecto a un parámetro determinado, este resulta ser muy pequeño. Como consecuencia, la actualización de dicho parámetro durante el proceso de entrenamiento es insignificante, lo que puede provocar que la red aprenda muy lentamente o incluso que deje de entrenarse por completo.

2.5.1. Long Short-Term Memory

Las redes conocidas como *Long Short-Term Memory* (LSTM) son un tipo particular de RNRs que fueron introducidas en el año 1997 [6] para solucionar el problema del gradiente desvaneciente que tenían las RNRs estándar, haciéndolas capaces de aprender dependencias de largo plazo. Han demostrado funcionar de manera efectiva en una gran variedad de problemas, y los mayores avances obtenidos con redes recurrentes se han logrado utilizando esta arquitectura [15].

Como hemos visto hasta el momento, todas las redes recurrentes tienen una estructura de cadena de "módulos" o "bloques" [15] - las neuronas recurrentes -, es decir, unidades que se repiten a lo largo de la secuencia temporal. En una RNR estándar, este módulo es simplemente una aplicación de una función de activación, particularmente la tangente hiperbólica, sobre una suma pesada de sus entradas. En las LSTM, estas neuronas se complejizan para incorporar mecanismos que permiten mantener una memoria a largo plazo en la red. Las llamaremos "celdas" o neuronas LSTM.

El principal componente que introducen las LSTM para poder retener información por largos períodos de tiempo es el **estado de celda**. La particularidad de este estado es que a medida que "fluye" por la red, se mantiene prácticamente intacto, salvo por algunas interacciones lineales [15], pero no es afectado por ningún peso o bias, lo que permite justamente solucionar el desvanecimiento del gradiente.

Así, vamos a tener dos estados que interactúan entre sí para realizar predicciones:

- Por un lado, el **estado de celda**, que representa la **memoria a largo plazo** de la red, y no es afectado por ningún peso o bias.
- Por otro lado, el **estado oculto** (del que hablamos previamente), que representa la **memoria a corto plazo** de la red, y sí es calculado por medio de sumas pesadas.

Ahora bien, con esto, la idea es que la red aprenda qué almacenar en el estado de celda, qué descartar y qué leer de él [3], y a esto lo hace a través de otros nuevos elementos llamados **compuertas**. En particular, se introducen tres tipos de compuertas en la celda LSTM:

- **La compuerta de olvido $\boldsymbol{f}$** (del inglés *forget*): determina si cada elemento del estado de celda es recordado, copiándolo al paso siguiente, u olvidado, fijándolo en 0.
- **La compuerta de entrada $\boldsymbol{i}$** (del inglés *input*): determina cuál va a ser la información que se va a utilizar para actualizar el estado de celda a partir de la entrada actual y del estado oculto actual.
- **La compuerta de salida $\boldsymbol{o}$** (del inglés *output*) determina qué partes de la memoria de largo plazo son transferidas a la memoria de corto plazo, el estado oculto.

De esta forma, las compuertas son una manera de regular el paso de información [15]. La salida de cada compuerta es un vector de valores todos en el rango $[0, 1]$. Estos se obtienen como la salida de una pequeña red neuronal totalmente conectada en la que las entradas son los datos del paso de tiempo actual y el estado oculto anterior, y cuya capa de salida aplica una función de activación sigmoide.

A continuación, detallamos las operaciones producidas por una celda LSTM en un paso de tiempo. Utilizaremos la siguiente notación para un tiempo t:

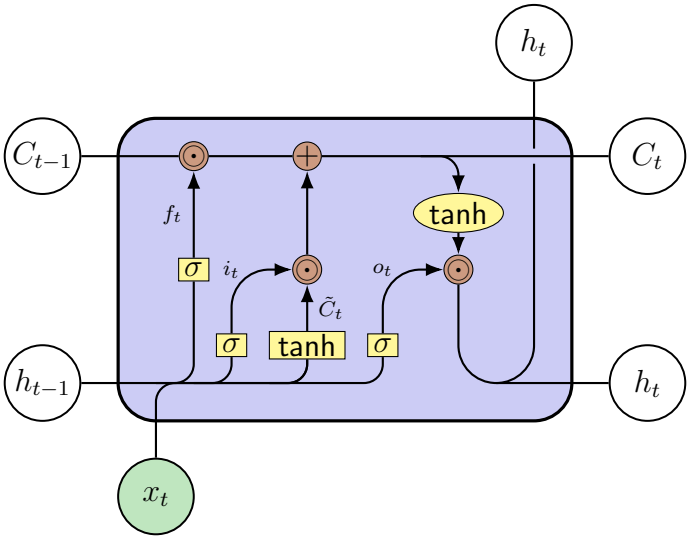


Figura 2.9: Diagrama de una celda LSTM. σ representa la función sigmoide, tanh la tangente hiperbólica, el signo "+" una suma, y el signo $\odot$ el producto de Hadamard, también conocido como "multiplicación elemento a elemento".

- x_t para el vector de entrada.
- h_t para el estado oculto.
- C_t para el estado de celda.
- $\tilde{C}_t$ para el "potencial" estado de celda (veremos a continuación qué significa esto).
- f_t, i_t, o_t para las salidas de las compuertas de olvido, entrada y salida respectivamente.

Cabe aclarar que h_t, C_t, $\tilde{C}_t$, f_t, i_t y o_t son todos vectores de la misma dimensión. Con estos elementos, podemos dar un diagrama de una celda LSTM, que se encuentra en la Figura 2.9.

Y ahora, como se puede deducir a partir de la Figura 2.9, vamos a tener diversas matrices de pesos involucradas, que todas se aprenden durante el entrenamiento. Dejando de lado sus dimensiones, y asumiendo que todas incluyen el bias, las denotaremos de la siguiente forma:

- $\boldsymbol{W}_{x,f}$ y $\boldsymbol{W}_{h,f}$ para los pesos de la compuerta de olvido.
- $\boldsymbol{W}_{x,i}$ y $\boldsymbol{W}_{h,i}$ para los pesos de la compuerta de entrada.
- $\boldsymbol{W}_{x,o}$ y $\boldsymbol{W}_{h,o}$ para los pesos de la compuerta de salida.
- $\boldsymbol{W}_{x,\tilde{C}}$ y $\boldsymbol{W}_{h,\tilde{C}}$ para los pesos de la potencial memoria a largo plazo.

En cada una, los subíndices indican de dónde hacia dónde van los pesos. Por ejemplo, $\boldsymbol{W}_{x,f}$ es la matriz de pesos que va desde la entrada x_t hacia la compuerta de olvido f_t, y $\boldsymbol{W}_{h,f}$ es la matriz de pesos que va desde el estado oculto h_{t-1} hacia la compuerta de olvido f_t. Con esto, veamos exactamente qué hace una neurona LSTM.

Para empezar, la celda toma tres entradas: el estado de celda del paso anterior C_{t-1}, el estado oculto del paso anterior h_{t-1}, y el vector de entrada del paso actual x_t. En cada paso, las tres compuertas actúan "internamente" para producir dos salidas: el nuevo estado de celda C_t y el nuevo estado oculto h_t.

La primera compuerta que actúa es la de olvido, para determinar que información ya presente en el estado de celda se "sigue recordando". Esta decisión se toma aplicando una función sigmoide sobre el estado oculto anterior h_{t-1}, y la entrada actual x_t, produciendo un número entre 0 y 1 para cada valor presente en C_{t-1}:

$$f_t = \sigma\left(\boldsymbol{W}_{x,f}x_t + \boldsymbol{W}_{x,h}h_{t-1}\right)$$

El próximo paso es decidir qué nueva información ingresa a C_t utilizando la compuerta de entrada. Esta etapa se puede ver como compuesta por dos partes:

- Por un lado, se calcula una potencial memoria a largo plazo $\tilde{C}_t$. Esto se hace combinando el estado oculto del paso anterior h_{t-1} y la entrada actual x_t y aplicando una tangente hiperbólica:

$$\tilde{C}_t = \tanh\left(\boldsymbol{W}_{x,\tilde{C}}x_t + \boldsymbol{W}_{h,\tilde{C}}h_{t-1}\right)$$

- Por otro lado, a través de la compuerta de entrada, se calcula cuáles van a ser los valores de la memoria a largo plazo que se van a actualizar.

$$i_t = \sigma\left(\boldsymbol{W}_{x,i}x_t + \boldsymbol{W}_{h,i}h_{t-1}\right)$$

Con estas tres cantidades, es momento de efectivamente actualizar el estado de celda:

1. En primer lugar, se multiplica elemento a elemento C_{t-1} por f_t. Esto provoca que un valor cercano a 0 en f_t haga que el valor correspondiene en C_{t-1} se "olvide", y que un valor cercano a 1 haga que se siga recordando.

2. Y luego, se le suma la nueva información, dada por la multiplicación elemento a elemento entre $\tilde{C}_t$ e i_t, que intuitivamente representa qué valores se van a actualizar y en qué medida.

De esta forma, la nueva memoria a largo plazo está dada por:

$$C_t = C_{t-1} \odot f_t + \tilde{C}_t \odot i_t$$

Por último, es necesario determinar cuál va a ser el nuevo estado interno de la celda. Este se construye como una versión "filtrada" del nuevo estado de celda, haciendo uso del valor de la compuerta de salida:

$$o_t = \sigma\left(\boldsymbol{W}_{x,o}x_t + \boldsymbol{W}_{h,o}h_{t-1}\right)$$

que, combinado con una aplicación de tanh sobre el nuevo estado de celda para situar a todos los valores en el rango $[-1, 1]$, da la nueva memoria a corto plazo:

$$h_t = o_t \odot \tanh(C_t)$$

Una aclaración es que cuando estas redes se utilizan para llevar a cabo clasificación de secuencias, lo que se hace es tomar el estado **oculto** del último paso y, similarmente a lo que se hace en las redes convolucionales, se agregan una o más capas totalmente conectadas que procesen este vector para hacer la clasificación final.

Capítulo 3

Presentación del problema

Como explicamos anteriormente, en cualquier evaluación de impacto, para poder calcular el efecto de un programa, se debe obtener una estimación apropiada del contrafactual: ¿qué hubiera pasado con la variable de objetivo de los beneficiarios del programa si el mismo no hubiera existido? Vimos también que para lograrlo, se construye un grupo de control, formado por individuos que no han sido tratados pero que idealmente, son estadísticamente similares a los que sí lo fueron.

En esto, aparece el problema del sesgo de autoselección, que consiste en no tomar en cuenta las diferencias preexistentes entre tratados y no tratados, que pueden haber afectado tanto a la decisión de participar en el programa como a los resultados potenciales posteriores.

En evaluaciones sobre programas con asignación aleatoria, el grupo de control se obtiene directamente tomando todos los individuos que no fueron tratados y el problema de autoselección se soluciona naturalmente. Sin embargo, en las evaluaciones cuasi-experimentales, hechas sobre tratamientos en donde la asignación al tratamiento no fue aleatoria y se desconocen los motivos que han llevado a los individuos a inscribirse o recibir el tratamiento, construir un grupo de control adecuado constituye un gran desafío. Este es uno de los focos centrales de este trabajo.

Una técnica estadística muy potente para abordar este problema es el PSM, discutida anteriormente y que es la que tomamos como referencia en el presente trabajo. Este método empareja individuos tratados con no tratados basándose en su probabilidad estimada de participación, calculada a partir de un conjunto de variables que se consideran influyentes en el programa bajo cuestión.

Estas variables son seleccionadas por los evaluadores y pueden representar tanto características observadas en un instante determinado como también compartamientos a lo largo del tiempo. Por ejemplo, si consideramos un tratamiento a empresas, se puede tomar como variable el número de empleados y los ingresos mensuales de solamente el mes anterior al inicio del programa, o se podría tomar esta cantidad observada durante diez meses previos al comienzo. De esta forma, es posible incluir series de tiempo en el

cálculo del puntaje de propensión, tomando como variables varios períodos de la misma característica. Esto permite capturar no solo un estado puntual de las unidades sino también como ha sido su evolución, lo cual puede contribuir a un mejor emparejamiento y consecuentemente, a una estimación más precisa del efecto del tratamiento.

Otra cuestión a tener en cuenta a la hora de identificar los individuos de control es la forma de implementación del programa con respecto al tiempo. Hay unos en los que el ingreso al tratamiento por parte de los individuos se realiza en un único instante de tiempo, pero en otros la entrada al programa ocurre de manera secuencial. En este último caso, se habla de **cohorte** para referirse al conjunto de individuos que ingresaron al programa en el mismo momento.

Habiendo presentado estas cuestiones, nuestro trabajo se enfoca en programas con entrada secuencial y en los que supondremos que los individuos tratados resultaron ser tratados (o decidieron inscribirse al programa) por la dinámica temporal observada en una característica en los períodos anteriores al inicio del programa. Bajo estas circunstancias, el PSM presenta ciertas limitaciones:

- Por un lado, cuando existen múltiples cohortes, la forma en la que se aplica la técnica es por cohorte. Es decir, lo que se hace es estimar la probabilidad por camada de ingreso al programa, y se buscan controles separadamente para cada una de estas camadas. Dicho de otra manera, se realizan tantos emparejamientos como cohortes haya.

- Por otro lado, cuando las variables tenidas en cuenta en realidad representan una misma característica pero en distintos períodos de tiempo, la regresión logística, que es la forma en la que se calcula el puntaje de propensión, no es capaz de capturar esta relación temporal entre ellas, sino que las ve como si fueran independientes.

Con estos problemas en mente, en esta tesina presentamos una alternativa para la selección de grupos de control bajo las hipótesis mencionadas previamente. Esta se basa en la utilización de diferentes tipos de redes neuronales, concretamente: redes totalmente conectadas, redes convolucionales y redes LSTM. Como explicamos en el capítulo anterior, las dos últimas incorporan naturalmente la relación temporal de los datos, permitiendo capturar patrones a lo largo del tiempo. En nuestro escenario, en donde suponemos que la dinámica previa a la intervención es un factor clave para entender la participación en el programa, estas características de estas redes resultan especialmente útiles.

Para evaluar la estrategia presentada, desarrollamos diferentes conjuntos de datos sintéticos, ya que en los casos reales algunos comportamientos no pueden verificarse directamente. Aquí, algunos de los parámetros a variar son la cantidad de períodos observados, la cantidad de períodos de dependencia temporal, y la dinámica temporal observada. Teniendo esto en cuenta, los objetivos de nuestro trabajo se pueden resumir en los siguientes puntos:

- Evaluar la capacidad de las redes neuronales de reconocer individuos de control ante diferentes dinámicas temporales de la variable observada.
- Superar las limitaciones observadas del PSM.
- Obtener resultados que permitan comparar el desempeño de las redes con el del PSM en diferentes escenarios.

Nuestras hipótesis son las siguientes:

- Cuando no hay una dependencia temporal fuerte, es decir no hay un comportamiento determinado en la evolución de la variable observada, el PSM y las redes tienen aproximadamente el mismo desempeño.
- Por el contario, cuando existe una alta dependencia temporal, o sea cuando en los datos forzamos un determinado comportamiento temporal, las redes funcionan mejor que el PSM.
- Las redes convolucionales y las LSTM funcionan mejor que las densas.
- Ante menor cantidad de períodos de observación, las redes cometen cada vez más errores.

A continuación, presentamos el marco sobre el cual llevamos a cabo los experimentos, detallando el alcance de nuestro trabajo, la forma en que construimos las diferentes simulaciones, las arquitecturas de redes, las métricas para evaluar los modelos, y las herramientas que nos ayudaron en todo este proceso.

Capítulo 4

Marco Experimental

El alcance del trabajo desarrollado está dado por las siguientes etapas, de las cuales hablaremos en este capítulo:

1. Generación de datos sintéticos.
2. Transformación de datos.
3. Diseño de modelos.
4. Búsqueda de hiperparámetros.
5. Entrenamiento del modelo.
6. Evaluación del modelo con diferentes métricas.
7. Comparación con los resultados obtenidos con el PSM.

También nombraremos las diferentes herramientas que nos ayudaron a llevar a cabo cada uno de estos pasos.

4.1. Conjuntos de Datos

El primer paso para poder llevar a cabo los experimentos es contar con datos. Nuestro entorno de simulación está diseñado para generar **datos sintéticos** que imiten escenarios reales en donde la asignación al tratamiento es no aleatoria, secuencial y potencialmente dependiente de resultados pasados.

El proceso de generación de datos involucra la creación de una serie de tiempo univariada para cada individuo, que incorpora efectos fijos y componentes autoregresivos. Distinguimos entre tres tipos de individuos:

- **Individuos tratados**: son aquellos que han recibido el tratamiento en algún período. Es importante notar que en una situación real, son datos con los que contamos.
- **Individuos de control**: son aquellos que en los datos sintéticos, sabemos que forman parte del grupo de control. Es importante notar que en un escenario real, no sabemos quiénes son estos individuos sino que son los que tratamos de identificar.
- **Individuos "NiNi"**: son aquellas unidades que no han sido tratadas y que en los datos sintéticos, sabemos que no forman parte del grupo de control.

Al tratarse de Aprendizaje Supervisado, también incluimos la etiqueta que le corresponde a cada individuo: 1 para tratados y controles, y 0 para los NiNi.

Los parámetros para la generación de los datos son los siguientes:

- `n_sample`: cantidad de individuos en la simulación.
- `treated_pct`: porcentaje de individuos tratados.
- `control_pct`: porcentaje de individuos de control.
- `T`: cantidad de períodos observados de cada individuo.
- `first_tr_period`: primer período de tratamiento (o único, dependiendo de la cantidad de cohortes).
- `n_cohorts`: cantidad de cohortes.
- `phiT`: persistencia auto-regresiva para los tratados.
- `phiC`: persistencia auto-regresiva para los controles.
- `n_dep_periods`: cantidad de períodos de dependencia para la participación en el tratamiento.

Concretamente, la fórmula con la que se generaron los valores de las series de tiempo fue la siguiente: **TODO**

El resultado de la simulación es un conjunto de datos de panel compuesto por series de tiempo univariadas de longitud `T` para los distintos tipos de individuos:

En el dataset, cada individuo tiene un identificador y su tipo; y aquellos de tipo 1 y 2 tienen como feature extra el período (desde `first_tr_period` hasta `first_tr_period + n_cohorts`) en el que ingresaron al programa. A continuación, se muestran algunos ejemplos, tomando `T = 6`, `first_tr_period = 3` y `n_cohorts = 2`:

En este punto cabe recordar cuál es nuestra meta: a partir de información sobre los individuos que fueron tratados, queremos identificar de entre los no tratados, quiénes son los que podrían formar parte del grupo de control.

También resulta muy importante tener en cuenta que en los datos generados sintéticamente, sabemos quiénes son los controles, pero en la realidad esto es justamente lo que queremos identificar.

4.1.1. Conjuntos de Entrenamiento y de Test

Ahora bien, para entrenar y evaluar a nuestros modelos, tomamos en cuenta lo que ocurriría en un escenario real, en el que sabríamos solamente quiénes son los tratados y quiénes los no tratados. Por lo tanto, construimos el conjunto de entrenamiento con la totalidad de los individuos de tipo 1 con etiqueta 1 y algunos de tipo 3 con etiqueta 0, y en el conjunto de test colocamos a todos los de tipo 2, queriendo predecir en ellos un 1, y al resto de tipo 3, queriendo predecir en ellos un 0.

Más aún, como queremos identificar a los grupos de control de cada cohorte,

4.2. Arquitecturas de Redes Neuronales

A continuación, describimos las diferentes arquitecturas de redes neuronales que utilizamos durante los experimentos y los valores particulares que fueron incluidos en la búsqueda de hiperparámetros. Un detalle que vale la pena aclarar es que el tamaño de la salida es 1 ya que

4.2.1. Arquitectura 1: Red Totalmente Conectada

Capa	Entrada	Salida
Entrada	n	n_1
Oculta 1	n_1	n_2
Oculta 2	n_2	1
Salida	1	1

Tabla 4.1: Arquitectura de Autoencoder 1: 2 capas convolucionales en el encoder, ambas con kernels de 3×3. Se pasa de tener la imagen de entrada a 8 feature maps de 13×13, a 16 de 5×5.

4.2.2. Arquitectura 2: Red LSTM

4.2.3. Arquitectura 3: Red Convolucional

4.2.4. Arquitectura 4: Red Convolucional + LSTM

4.3. Herramientas

A continuación, nombramos las herramientas que nos ayudaron a llevar a cabo los experimentos, desde la generación de datos sintéticos hasta el diseño, entrenamiento y

evaluación de los modelos, junto con la búsqueda de hiperparámetros óptimos. En la sección de bilbiografía, se incluye el enlace a su documentación.

4.3.1. Hardware

Para el entrenamiento de los modelos y la búsqueda de hiperparámetros, utilizamos recursos computacionales de UNC Supercómputo (CCAD) de la Universidad Nacional de Córdoba (https://supercomputo.unc.edu.ar), que forman parte del Sistema Nacional de Computación de Alto Desempeño (SNCAD) de la República Argentina. En particular, hicimos uso de la Computadora Nabucodonosor, destinada especialmente a potenciar el uso del ML. Cuenta con 2 procesadores Xeon E5-2680v2 de 10 núcleos cada uno, 64 GiB de RAM, y 3 GPU NVIDIA GTX 1080Ti.

4.3.2. Jupyter

Jupyter es una plataforma que ofrece un entorno de desarrollo interactivo a través de documentos llamados "*notebooks*", organizados en unidades denominadas "celdas". La potencia de estas notebooks radica en que pueden contener celdas de código, texto en lenguaje natural escrito en Markdown, imágenes, y diversas otras representaciones, como tablas y diagramas. Estos documentos proporcionan una forma rápida y práctica de prototipar y explicar código, explorar y visualizar datos, y realizar chequeos intermedios observando la salida de cada celda, lo cual contribuye a prevenir errores. Jupyter soporta diferentes lenguajes de programación y permite la utilización de entornos virtuales.

En este proyecto, implementamos un *pipeline* mediante notebooks de Jupyter que abarca desde la carga, visualización y transformación de los datos, hasta la selección, entrenamiento y evaluación de los modelos.

4.3.3. Python

Python es un lenguaje de programación interpretado, orientado a objetos y con tipado dinámico. Es multipropósito, por lo que se puede usar para una diversiad de tareas. Otros de sus aspectos más relevantes son su sintaxis simple e intuitiva con énfasis en la legibilidad, el alto grado de abstracción que provee, la gran cantidad de librerías que existen, y el extenso soporte de la comunidad. Además, es gratis, de código abierto y multiplataforma, por lo que puede ejecutarse en diferentes sistemas operativos.

Durante los últimos años, ha sido el lenguaje más utilizado en las áreas de Ciencia de Datos y Aprendizaje Automático, principalmente por el desarrollo de librerías como pandas, numpy, PyTorch, TensorFlow, y Scikit-learn, que facilitan el manejo de grandes volúmenes de datos y la utilización de algoritmos de ML.

En nuestro caso, utilizamos Python para todo la implementación de experimentos.

4.3.4. Pandas

Pandas es una librería de Python que provee estructuras de datos rápidas, flexibles y expresivas que permiten trabajar de manera fácil e intuitiva con datos tabulares. Las dos principales estructuras de datos de pandas son las `Series` (unidimensionales) y los `DataFrames` (bidimensionales). En cada una, se pueden utilizar diferentes tipos de datos, como enteros, decimales, palabras, fechas, entre otros. Algunas de sus capacidades destacadas incluyen la lectura y escritura de datos en diferentes formatos (CSV, Microsoft Excel, Stata, etcétera), el manejo de datos faltantes, y la simple y eficiente transofrmación de datos.

Utilizamos `DataFrames` para almacenar los datos sintéticos en archivos y para los conjuntos de entrenamiento y test.

4.3.5. NumPy

NumPy es una librería de Python que permite el cómputo numérico. Ofrece arreglos N-dimensionales, funciones matemáticas variadas, generadores de números aleatorios, rutinas de álgebra lineal, y más. El núcleo de NumPy es código C optimizado, por lo que combina la flexiblidad de Python con la rapidez propia del código compilado. Estas características hacen que sea la base de muchas otras librerías utilizadas en ámbitos muy diversos.

Empleamos numpy principalmente para crear las series de tiempo de los diferentes individuos en matrices, que luego pasamos a `DataFrames`.

4.3.6. PyTorch

PyTorch es una librería de Python que facilita la utilización de redes neuronales, tanto en su diseño como en su entrenamiento. Provee cómputo acelerado mediante el uso de placas gráficas (*Graphic Processing Unit*, GPU), e implementa internamente las optimizaciones numéricas necesarias para entrenar un modelo, siendo la más importante el la del algoritmo de backpropagation. La mayoría de PyTorch está escrito en C++ y CUDA.

Aquí, utilizamos PyTorch justamente para diseñar las diferentes arquitecturas de redes neuronales, que se implementan como clases en Python, y para llevar a cabo el ciclo de entrenamiento de los distintos modelos.

4.3.7. Optuna

Optuna es una librería de Python que permite realizar la optimización automática de hiperparámetros de una manera eficiente. Resulta intuitiva de utilizar y su incorporación en un pipeline es bastante directa. Un aspecto muy útil es que incluye un *dashboard*

interactivo que posibilita la visualización en tiempo real de los valores de hiperparámetros evaluados en los diferentes ensayos, como así también de gráficos que reflejan la relación entre estso valores y el resultado obtenido con ellos (medido a través de una métrica específica, determinada por el usuario). Otra característica relevante es que Optuna permite la paralelización de las evaluaciones, lo que contribuye a acelerar el proceso de optimización.

4.3.8. MLflow

MLflow es una librería de Python que permite llevar un registro fino de las diferentes etapas y los diferentes experimentos llevados a cabo en un proyecto de ML. Provee una interfaz con la cual se pueden crear diferentes proyectos, y dentro de cada uno diferentes experimentos. En cada uno de ellos, permite cargar los parámetros y valores que el usuario crea adecuados (como pueden ser los hiperparámetros empleados para entrenar el modelo), los datasets utilizados, y diferentes archivos que pueden incluir gráficos, métricas, y hasta incluso los pesos del modelo entrenado.

4.4. Métricas

En problemas de clasificación binaria, existen diferentes métricas para evaluar el desempeño del modelo. Las que más relevancia tendrán depende del problema. Para describirlas, utilizaremos la siguiente notación, asumiendo que la etiqueta 1 coresponde a la clase "positiva" y la 0 a la "negativa":

- VP: Verdaderos Positivos. Es la cantidad de predicciones correctas para la clase positiva. Es decir, aquellos datos de entrada cuya etiqueta verdadera es 1 y que el modelo también clasificó como 1. En nuestro caso, los VP son los indiviudos de control que el modelo identificó correctamente como de control.

- VN: Verdaderos Negativos. Es la cantidad de predicciones correctas para la clase negativa. Es decir, aquellos datos de entrada cuya etiqueta verdadera es 0 y que el modelo también clasificó como 0. En nuestro caso, los VN son los indiviudos NiNi que el modelo identificó correctamente como NiNi.

- FP: Falsos Positivos. Es la cantidad de predicciones incorrectas para la clase negativa. Es decir, aquellos datos de entrada cuya etiqueta verdadera es 0 pero que el modelo clasificó como 1. En nuestro caso, los FP son los indiviudos NiNi que el modelo identificó como incorrectamente de control.

- FN: Falsos Negativos. Es la cantidad de predicciones incorrectas para la clase positiva. Es decir, aquellos datos de entrada cuya etiqueta verdadera es 1 pero que el

modelo clasificó como 0. En nuestro caso, los FN son los indiviudos de control que el modelo identificó incorrectamente como NiNi.

Todas estas cantidades se suelen presentar de forma conjunta en lo que se denomina **matriz de confusión**:

Con estos conceptos en mente, las distintas métricas que hay son las siguientes:

4.4.1. Exactitud

Es la proporción de entradas para las cuales el modelo predijo la salida correcta. Su fórmula está dada por:

$$Exactitud = \frac{VP + VN}{VP + VN + FP + FN}$$

Utilizar esta medida no es recomendable en datasets desbalanceados, es decir en aquellos en donde hay un alto porcentaje de una clase pero bajo de otra. Veamos por qué con un ejemplo.

Supongamos que tenemos un dataset de 100 muestras en donde el 90 % de los datos tienen etiqueta 0 y el restante 10 % tiene etiqueta 1. Se puede ver que si el modelo predice un valor de 0 para cualquier entrada, entonces tendría una exactitud del 90 %, que es un porcentaje bastante alto, pero se habrá equivocado en todas las instancias negativas.

Por esto, y particularmente en datasets balanceados como son con los que trabajamos nosotros, es necesario prestar atención a otros valores.

4.4.2. Precisión

Es la proporción de verdaderos positivos sobre todos los positivos detectados por el modelo. Una precisión de 1 indica que cada elemento predicho como 1 efectivamente era un 1. Su fórmula está dada por:

$$Precisión = \frac{VP}{VP + FP}$$

Nuevamente, utilizar solamente la precisión es engañoso. Una forma de tener una precisión perfecta es crear un clasificador que siempre prediga un valor de 0, excepto en única instancia de la que esté más seguro de predecir 1 [3]. Si esta predicción es correcta, entonces tendríamos $VP = 1$, $FP = 0$, dando de esta forma una precisión de 1. Por esto, se la suele combinar con otra proporción llamada sensibilidad.

4.4.3. Sensbilidad o Ratio de Verdaderos Positivos

Es la proporción de instancias positivas predichas corrctamente por el modelo. Su fórmula está dada por:

$$Sensibilidad = \frac{VP}{VP + FN}$$

Prestar atención solamente a la sensibilidad también es riesgoso. Si volvemos al dataset de 100 muestras donde el 90 % del dataset tiene etiqueta 0 y el restante 10 % tiene etiqueta 1, podemos obtener una sensibilidad de 1 simplemente construyendo un modelo que siempre prediga 1, ya que tendríamos $VP = 10$ y $FN = 0$. Sin embargo, tendríamos una precisión de 0.1, ya que $FP = 90$.

Dicho esto, existe una medida que combina a estas dos métricas.

4.4.4. Puntaje F Beta

Capítulo 5

Resultados

Capítulo 6

Conclusiones y Trabajos Futuros

Bibliografía

[1] Raquel Bernal y Ximena Peña. *Guía práctica para la evaluación de impacto*. 1.ª ed. Universidad de los Andes, Colombia, 2011. ISBN: bernal2011. URL: http://www.jstor.org/stable/10.7440/j.ctt1b3t82z (visitado 10-02-2025).

[2] Centro de Computación de Alto Desempeño - Universidad Nacional de Córdoba. *Computadora Nabucodonosor*. URL: https://supercomputo.unc.edu.ar/equipamiento/computadora-nabucodonosor/.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. 1st. O'Reilly Media, 2017. ISBN: 9781491962299.

[4] Paul Gertler et al. *La evaluación de impacto en la práctica: Segunda edición*. Banco Internacional para la Reconstrucción y el Desarrollo/Banco Mundial, 2016. DOI: https://doi.org/10.18235/0006529. URL: https://hdl.handle.net/10986/25030.

[5] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[6] Sepp Hochreiter y Jürgen Schmidhuber. «Long Short-Term Memory». En: *Neural Computation* 9 (nov. de 1997), págs. 1735-1780. DOI: 10.1162/neco.1997.9.8.1735.

[7] *Jupyter*. URL: https://docs.jupyter.org/en/latest/.

[8] Diederik P. Kingma y Jimmy Ba. «Adam: A Method for Stochastic Optimization». En: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6980.

[9] Sebastian Martinez, Sophie Naudeau y Vitor Azevedo Pereira Pontual. «The promise of preschool in Africa : a randomized impact evaluation of early childhood development in rural Mozambique (English)». En: (2012). URL: http://documents.worldbank.org/curated/en/819111468191961257.

[10] Warren McCulloch y Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». En: *The bulletin of mathematical biophysics* (19430). URL: https://doi.org/10.1007/BF02478259.

[11] Tom M. Mitchell. *Machine Learning*. McGraw-hill, 1997. ISBN: 0070428077.

[12] *MLFlow*. URL: https://mlflow.org/docs/latest/index.html.

[13] Izaak Neutelings. *Neural Networks with TikZ*. URL: https://tikz.net/neural_networks/.

[14] *NumPy*. URL: https://numpy.org/doc/stable/.

[15] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[16] *Optuna*. URL: https://optuna.readthedocs.io/en/stable/index.html.

[17] *pandas*. URL: https://pandas.pydata.org/docs/.

[18] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2024. URL: http://udlbook.com.

[19] *Programa Jornada Extendida y su impacto en el clima escolar*. Informe de evaluación de impacto. Publicado en la Biblioteca Digital de la UEICEE. Unidad de Evaluación Integral de la Calidad y Equidad Educativa (UEICEE), Ministerio de Educación, Gobierno de la Ciudad de Buenos Aires, 2023. URL: https://bde-ueicee.bue.edu.ar/documentos/708-programa-jornada-extendida-y-su-impacto-en-el-clima-escolar.

[20] *Python*. URL: https://docs.python.org/3/.

[21] *PyTorch*. URL: https://pytorch.org/docs/stable/index.html.

[22] PAUL R. ROSENBAUM y DONALD B. RUBIN. «The central role of the propensity score in observational studies for causal effects». En: *Biometrika* 70.1 (1983), págs. 41-55. ISSN: 0006-3444. DOI: 10.1093/biomet/70.1.41. URL: https://doi.org/10.1093/biomet/70.1.41.

[23] Frank Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain». En: *Psychological Review* 65.6 (1958), págs. 386-408. DOI: 10.1037/h0042519.

[24] Donald Rubin. «ESTIMATING CAUSAL EFFECTS OF TREATMENTS IN EXPERIMENTAL AND OBSERVATIONAL STUDIES». En: *Journal of Education Psychology* 66.5 (1974), págs. 688-701. DOI: https://doi.org/10.1002/j.2333-8504.1972.tb00631.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2333-8504.1972.tb00631.x.

[25] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Learning representations by back-propagating errors». En: *Nature* (1986). DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[26] Stuart J. Russell y Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4, Global Edition. Pearson Education, 2020. ISBN: 9781292401133.

[27] Howard White. «Theory-based impact evaluation: principles and practice». En: *Journal of development effectiveness* 1.3 (2009), págs. 271-284.