Universidad Nacional de San Martín
Escuela de Ciencia y Tecnología

Trabajo Práctico

Redes Neuronales

Tema: Desarrollo de Redes Neuronales
Materia: Matemática III
Integrantes: Alvarez Juan Bautista, Maidana Juan Manuel
Profesores: Bompensieri Josefina, Prudente Tomás
2do Cuatrimestre 2024

Introducción

En este informe presentamos el desarrollo y análisis del trabajo práctico sobre redes neuronales. Veremos los criterios para la selección y preparación de una base de datos, junto con la implementación de una red neuronal para predicción.
Finalmente, también se incluye una reflexión sobre lo aprendido en el camino, y por qué nos puede servir más la construcción de una red neuronal desde cero (en el aspecto pedagógico) que simplemente copiar, pegar y crear una red utilizando alguna librería.

Entonces, no solamente crearemos una red, sino que también deberemos seleccionar una base de datos y tratarla si es necesario para que pueda ser usada de forma óptima en la red.

Finalmente, compararemos el rendimiento de nuestra red neuronal creada manualmente con una red generada mediante Scikit-Learn, y decidiremos si, efectivamente, la base elegida es adecuada para un desarrollo de este estilo.

A continuación cómo dividiremos los temas anteriormente mencionados:

1. **Análisis de la Base de Datos**
    - Describir y analizar columnas de datos.
    - Análisis de correlaciones.
    - Datos atípicos y limpieza de datos.
    - Transformaciones preliminares.

2. **Desarrollo de la Red Neuronal**
    - Arquitectura de la red.
    - Implementación en numpy.
    - Entrenamiento y evaluación.

3. **Comparación con Scikit-Learn**
    - Implementación de Scikit-Learn.
    - Comparación de rendimiento.

4. **Conclusiones finales**
    - Conclusiones y reflexiones sobre el trabajo realizado.

Durante el avance del informe se hará referencia a cálculos, pruebas y uso de gráficos. Todos estos fueron desarrollados y pueden ser vistos y probados en el archivo “Alvarez-Maidana.ipynb” que se adjunta como parte de la entrega de este TP.

Parte 1 - Análisis de la Base de Datos

Para trabajar con esta red neuronal, seleccionamos un dataset que contiene información acerca de estudiantes de secundaria, donde se detallan sus hábitos de estudio, su demografía, lo involucrados que están sus padres en su educación, actividades extracurriculares y performance académica.

El dataset en cuestión es el siguiente: *Kaggle - Students Performance Dataset*

Nuestro objetivo con este dataset es poder crear una red neuronal con la capacidad de predecir si un alumno es *bueno* o *malo.* Para ello, el criterio que usaremos será el siguiente:

- Estudiante “malo”: GPA < 2
- Estudiante “bueno”: GPA >= 2

siendo GPA “Grade Point Average”, o la suma de las notas de las materias dividido por los créditos del estudiante.

A continuación explicamos en profundidad todas las columnas de datos de las que disponemos, con esta información (y alguna más, como la correlación por ejemplo) podremos decidir cuáles columnas de datos nos sirven (porque influyen en el GPA) y cuáles no.

Columnas y tipos de Datos

En el dataset disponemos de un total de 15 columnas, las cuales podemos clasificar de la siguiente forma (lo haremos en inglés, ya que es de la manera que aparecen en el dataset):

- **Student Information**
  - *Student ID*
  - Demographics Details (*Age, Gender, Ethnicity, ParentalEducation*)
  - Study Habits (*StudyTimeWeekly, Absences, Tutoring*)

- **Parental Involvement**
  - *ParentalSupport*

- **Extracurricular Activities**
  - *Extracurricular*
  - *Sports*
  - *Music*
  - *Volunteering*

- **Academic Performance**
  - *GPA*
  - *GradeClass*

Ya teniendo una idea de las columnas que nos provee el dataset, a continuación pasamos a explicar qué representa cada una, de qué tipo de variable se trata y de qué podría servirnos.

StudentID: Simplemente es el número de identificación del estudiante. Es una variable de tipo Categórica. No nos sirve de nada prácticamente, es simplemente un identificador.

Age: La edad de los estudiantes (valores entre 15 y 18 años). Es una variable de tipo Discreta.

Gender: El género del estudiante (0 = masculino, 1 = femenino). Es una variable de tipo Categórica.

Ethnicity: Etnia (0 = Caucásico, 1 = Afroamericano, 2 = Asiático, 3 = otro). Es una variable de tipo Categórica.

ParentalEducation: Educación máxima alcanzada por los padres del alumno (0 = Ninguna, 1= Secundario Completo, 2 = Universidad, 3 = Bachiller, 4 = Más alto). Es una variable de tipo Categórica.

StudyTimeWeekly: Representa las horas semanales de estudio del alumno. Es una variable de tipo Continua.

Absences: La cantidad de inasistencias que tuvo el alumno durante un año. Es una variable de tipo Continua.

Tutoring: Indica si el alumno recibe tutoría especial (0 = No, 1 = Sí). Es una variable de tipo Categórica.

ParentalSupport: Indica el nivel de apoyo de los padres (0 = Ninguno, 1 = Bajo, 2 = Moderado, 3 = Alto, 4 = Muy alto). Es una variable de tipo Categórica.

Extracurricular, Sports, Music & Volunteering: Las cuatro columnas indican si el alumno realiza actividades extracurriculares, deportes, música o voluntariado respectivamente (0 = No, 1 = Sí). Son variables de tipo Categóricas.

GPA: Es el promedio de calificación del alumno, en un rango de 0 a 4. Está influenciada por las variables anteriormente mencionadas del estudiante. Es una variable de tipo Continua, y en nuestro caso, la columna objetivo.

GradeClass: Es la clasificación del alumno basado en el GPA:

- 0: “A” (GPA >= 3.5)
- 1: “B” (3.0 <= GPA < 3.5)
- 2: “C” (2.5 <= GPA < 3.0)
- 3: “D” (2.0 <= GPA < 2.0)
- 4: “F” (GPA < 2.0)

Ahora bien, ya teniendo en claro qué variables tenemos, qué representan cada una, de qué tipo son y nuestra columna objetivo, ¿cómo sabemos cuáles nos sirven y cuáles no para lo que queremos hacer?

Aparte del sentido común (en algunos casos como el StudentID) lo que podemos hacer en primer lugar es analizar las correlaciones entre dichas variables.

Para evaluar la **factibilidad** que podrá tener este dataset en el entrenamiento de nuestra red neuronal de clasificación, deberemos tener en cuenta tanto los datos como el objetivo del modelo.
La base de datos parece adecuada para crear una red neuronal de clasificación, ya que tenemos a nuestra disposición diferentes variables que podrían influir en el rendimiento académico de un alumno. Algunos ejemplos son Absences, StudyTimeWeekly o ParentalSupport, entre otras.
Estas variables, al ser categóricas y numéricas, podrán ser utilizadas para hacer la predicción binaria que se detalla más arriba.

Análisis de Correlaciones

La correlación mide la relación entre dos variables y nos indica cómo se comporta una en relación con la otra. En nuestro caso, que nuestra columna objetivo es la GPA, nos interesa muchísimo ver qué tipo de correlación tiene la misma con el resto de las columnas, pues deberemos poder saber cuáles influyen más y cuáles no.

En síntesis, los valores cercanos a 1 y a -1 indican una fuerte dependencia entre una variable y la otra (dependencia positiva o negativa respectivamente). En caso de que la correlación tienda a 0, es porque prácticamente no hay relación lineal entre las variables.

Además, una dependencia positiva indica que, a medida que una variable crece, la otra también lo hace. En contrapartida, la dependencia negativa indica lo contrario, mientras una crece la otra decrece.

Correlación con Pandas

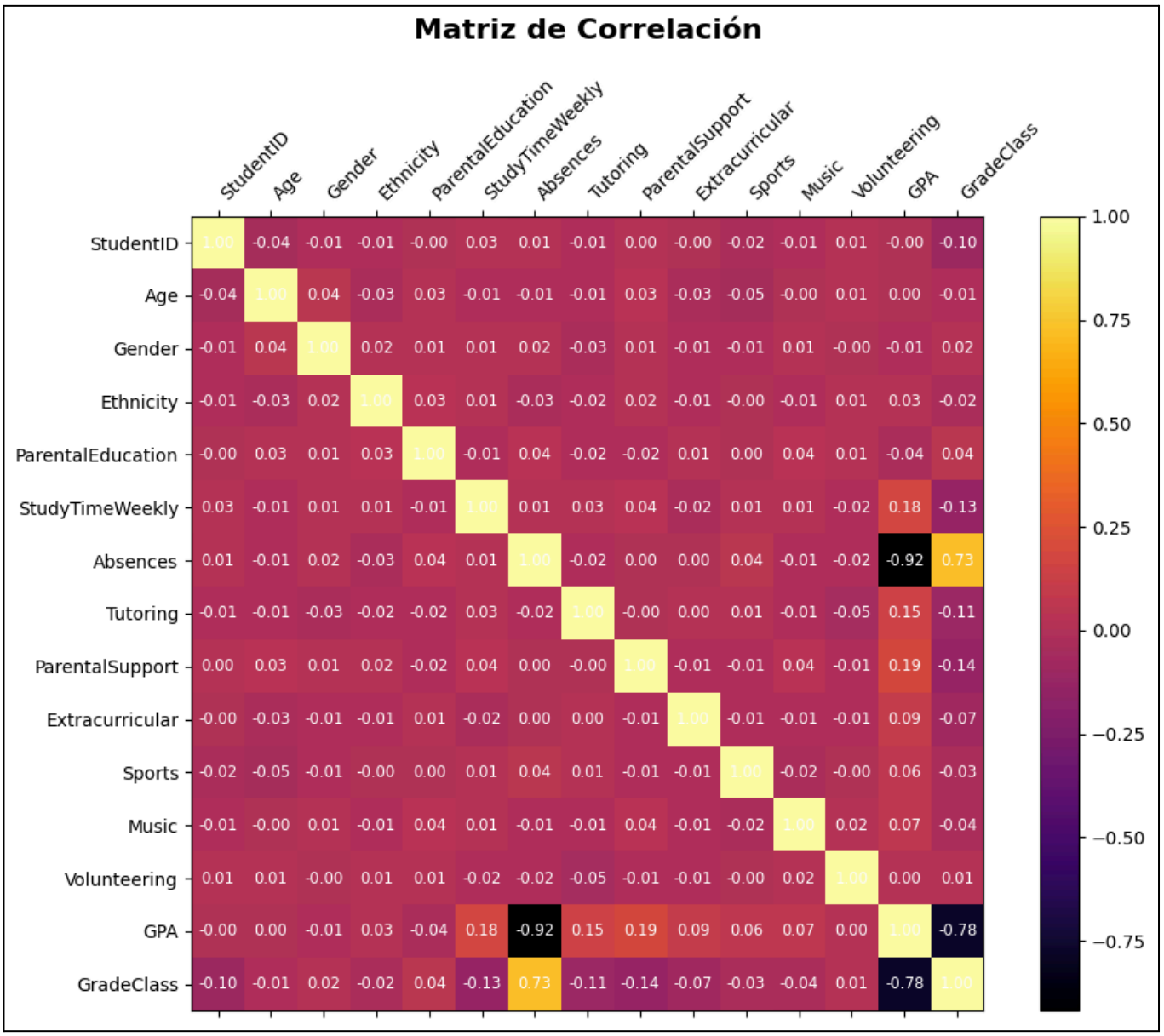
Ahora bien, podemos hacer uso de la librería Pandas para obtener la correlación de las variables con respecto al GPA de forma muy fácil y rápida. Nos arroja lo siguiente:

```
StudentID          -0.002697
Age                 0.000275
Gender             -0.013360
Ethnicity           0.027760
ParentalEducation  -0.035854
StudyTimeWeekly     0.179275
Absences           -0.919314
Tutoring            0.145119
ParentalSupport     0.190774
Extracurricular     0.094078
Sports              0.057859
Music               0.073318
Volunteering        0.003258
GPA                 1.000000
GradeClass         -0.782835
Name: GPA, dtype: float64
```

A simple vista ya podemos identificar algunos valores que tienen fuerte correlación, como las ausencias (mucha, por cierto) o el tiempo de estudio semanal.

Matriz de Correlaciones

Para que sea más vistoso y fácil de interpretar podemos usar una matriz de correlaciones. Básicamente es una tabla con los coeficientes de correlación de las variables que componen el dataset, por lo que no solo podremos ver el del GPA sino que todas las correlaciones.



Ahora sí, es muchísimo más fácil de apreciar las correlaciones a simple vista: podemos comenzar a decidir con cuáles nos vamos a quedar y con cuáles no.

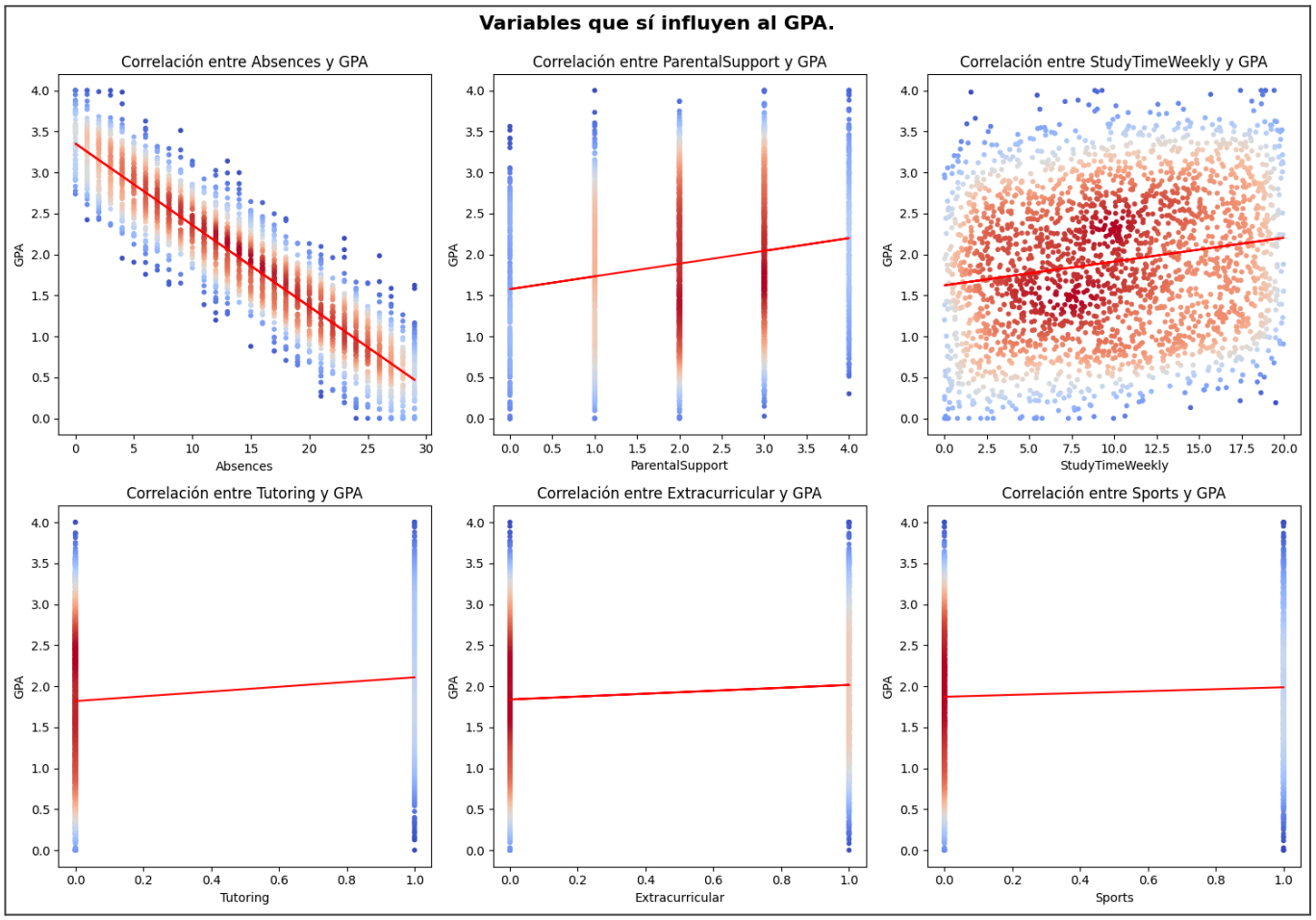
Tenemos que las columnas de StudentID, Age, Gender y Volunteering tienen una correlación prácticamente nula, por ende, nos desharemos de ellas para nuestra red neuronal.

Luego, las columnas de Absences, ParentalSupport, StudyTimeWeekly y Tutoring que sí tienen correlación. Importante destacar que todas tienen una correlación positiva, excepto Absences (que además tiene muchísima correlación, y es negativa).

Por último, tenemos otras columnas que también tienen correlación, aunque en mucha menor medida. Se trata de Extracurricular, Sports, Music, ParentalEducation, Ethnicity.

Gráficos de Correlación

Ya tenemos identificadas qué columnas nos interesan y cuáles no, pero tal vez no se aprecie del todo qué tan nula (o que tan fuerte) es la correlación entre alguna que otra variable y el GPA, por lo que procedemos a graficar las correlaciones entre dichas variables.



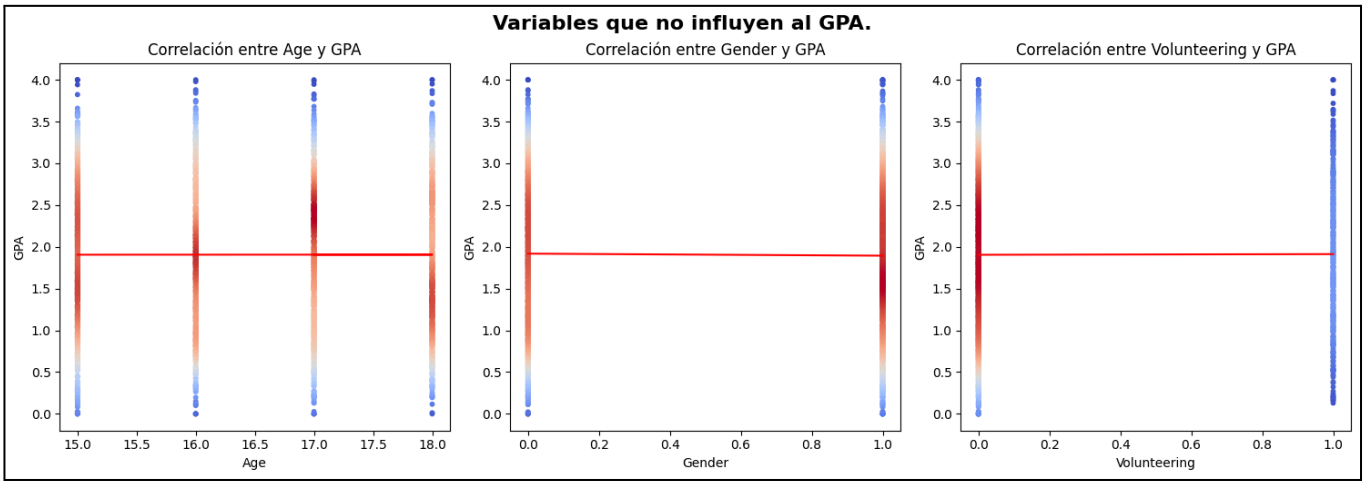
Comenzando por las variables que sí influyen (no todas, quedan por fuera un par cuya correlación es leve, simplemente no las agregamos para no saturar con gráficos), podemos denotar que claramente existen correlaciones en estos casos, pasamos a explicar qué sucede en cada caso.

Absences: Con las ausencias claramente tenemos una fuertísima correlación negativa. Esto quiere decir que, a medida que aumente la cantidad de ausencias, disminuye considerablemente el GPA.

ParentalSupport: En este caso la correlación es positiva, lo que indicaría que a mayor soporte parental mejor le va al alumno en general.

StudyTimeWeekly: Nuevamente tenemos una correlación positiva, indica que mientras más horas semanales le dedique un alumno al estudio, más chances tiene de tener un buen GPA.

Tutoring, Extracurricular, Sports: En este caso las pendientes ya no son tan pronunciadas pero sí existen. Se trata de una correlación positiva, por lo que esto indica que los alumnos que atraviesen estas actividades pueden tener una leve ventaja en lo que respecta a su desempeño académico. (En esta última categoría, de leve correlación pero existente, también entrarían **Music**, **ParentaEducation** y **Ethnicity**).



Con respecto a las variables que no influyen al GPA, no hay mucho que se pueda decir. Simplemente su correlación es nula, no aportan nada a la red y hasta podrían empeorar su desempeño (estaríamos haciendo cálculos que no aportarían nada, y hasta podrían confundir a la red).

Tal vez esté de más mencionarlo, pero claramente ni siquiera graficamos StudentID porque evidentemente no aportará nada, es simplemente un índice.

Síntesis

Entonces, como dijimos arriba, nos desharemos de las columnas con nula correlación, pero, ¿por qué tomamos esa decisión?

La respuesta es simple, es muy útil eliminar columnas que no aportan nada al propósito de nuestra red, porque *mejora el rendimiento* de la misma y *reduce el tiempo de cómputo* (recordemos que estamos realizando cálculos para poder predecir algo), *prevenimos el overfitting* (esas variables que no aportan nada podrían hacer que el modelo aprenda patrones aleatorios, lo que perjudica la precisión con nuevos datos) y también aporta a la *sencillez del modelo* (es decir, es más fácil interpretar una red con 8 neuronas que con 13, donde además la mitad no sirve para nada).

Entonces, dejemos en claro directamente con cuáles nos quedamos y con cuáles no:

Columnas que serán removidas:

- StudentID
- Age
- Gender
- Volunteering

Columnas que utilizaremos:

- Absences
- ParentalSupport
- StudyTimeWeekly
- Tutoring
- Extracurricular
- Sports
- Music
- ParentalEducation
- Ethnicity

Sin embargo, entre todas estas columnas, falta **GradeClass.** También nos desharemos de esta variable ya que es una clasificación directamente derivada de GPA, por lo que tenerla en el modelo podría sesgar o guiar de alguna manera la predicción.

Mantener ambas columnas podría ser una especie de trampa en el aprendizaje, por así decirlo. Simplemente se ajustaría la clasificación existente en vez de aprender a distinguir los patrones que generan las otras variables.

Valores Atípicos y Limpieza de Datos

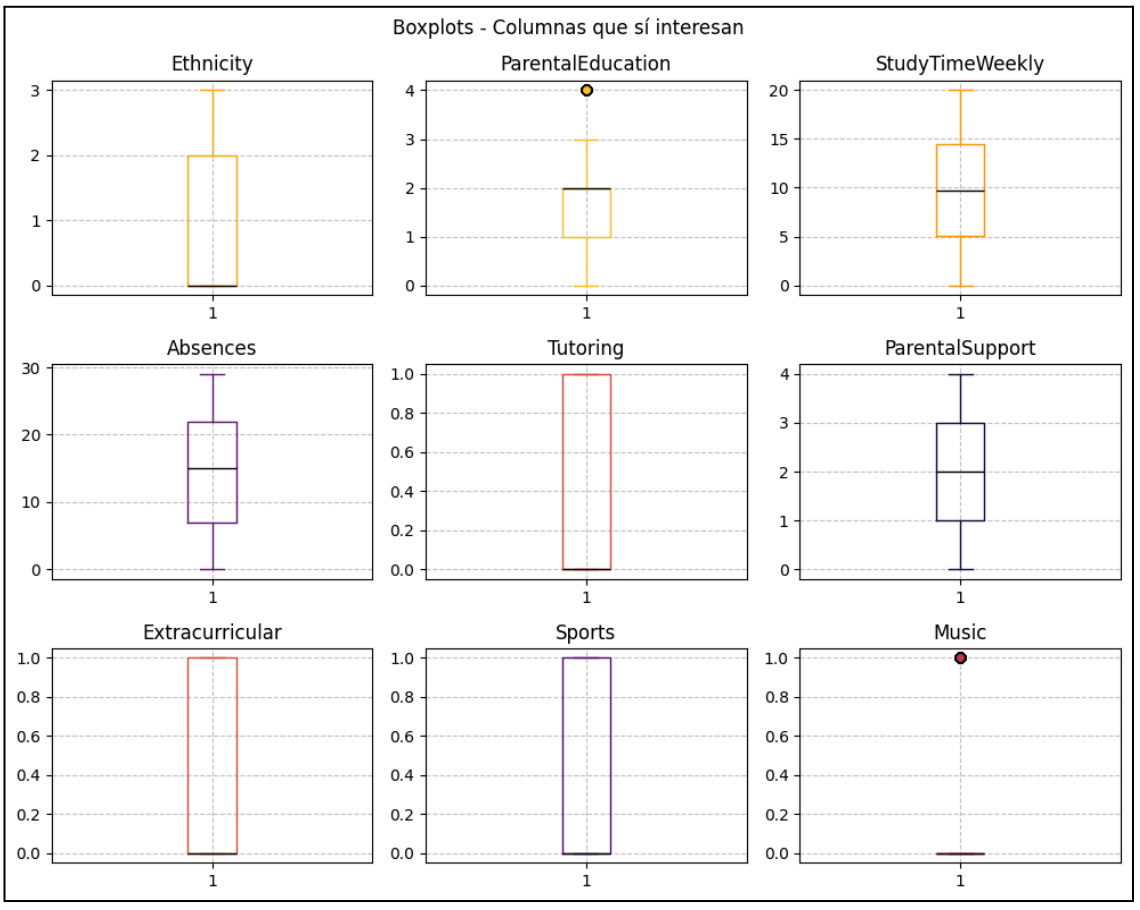
Cuando hablamos de **valores atípicos** en nuestra base de datos, hablamos de aquellos valores que pueden llegar a distorsionar completamente el análisis al conjunto de datos.
Por ejemplo, en una determinada columna, tener un valor extremadamente atípico (ya sea alto o bajo) podría alterar claramente el valor de la media (por esto mismo a veces, además de una limpieza de datos, es recomendable usar la mediana en ciertos casos, ya que es mucho menos sensible a valores atípicos) o inclusive llegar a alterar nuestra interpretación de los datos.

Pueden surgir debido a una mala medición en los datos o algún fenómeno inusual en la población estudiada, lo que es seguro es que generalmente no los queremos tener cerca.

Pero en síntesis, debemos tratar este tema porque, como mencionamos, pueden distorsionar los análisis estadísticos o a la misma red neuronal, pueden causar que interpretemos incorrectamente los datos e inclusive perjudicar al rendimiento del modelo (al ingresar datos nuevos que no sean de entrenamiento, no sabrá qué hacer porque probablemente se ajustó demasiado a esos puntos extremos). En caso de tenerlos, claramente deberemos tomar algún tipo de acción con respecto a ellos (eliminarlos, modificarlos, o ignorarlos).

Para ver si tenemos valores atípicos en nuestra base de datos podemos ayudarnos con boxplots, básicamente utilizamos la mediana y los cuartiles para ver si tenemos o no valores atípicos. Este método nos permite encontrar rápidamente valores atípicos en nuestro dataset.

Boxplots



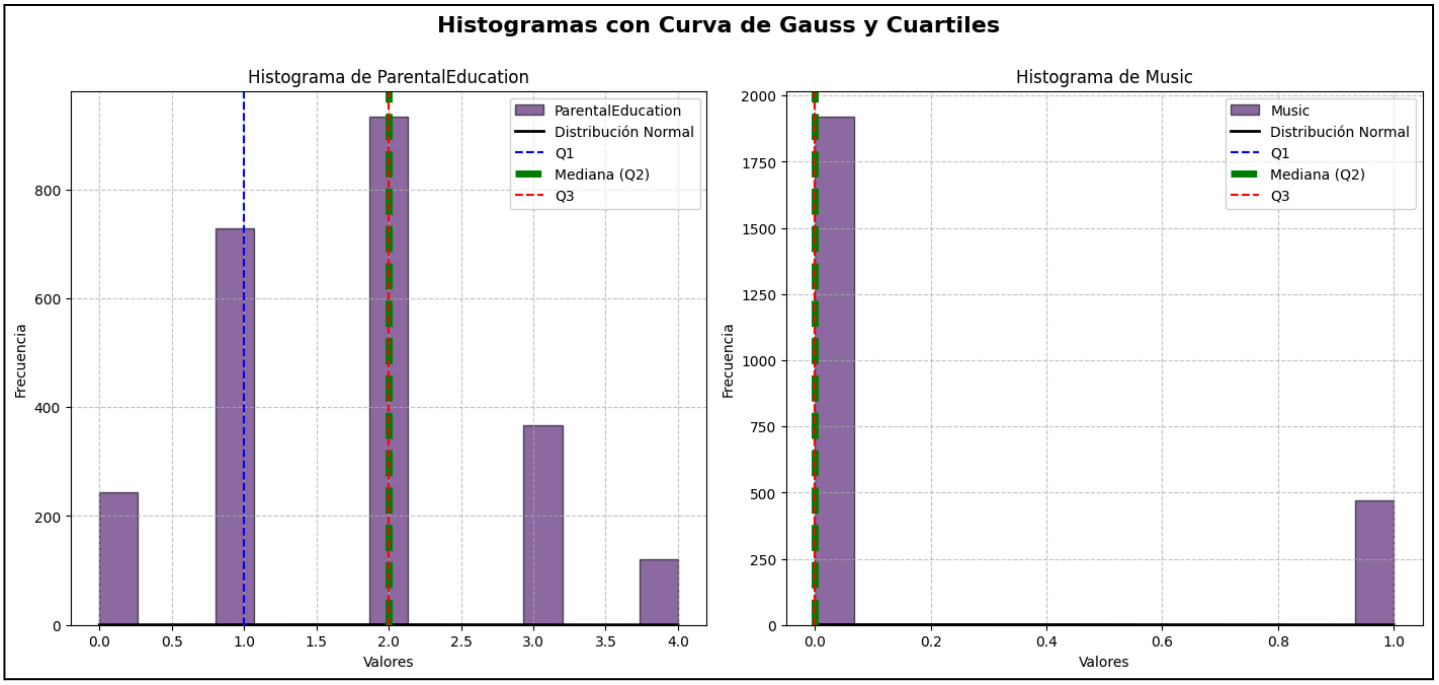
Claramente, nos va a interesar analizar valores atípicos en las columnas que se utilizarán en la red. Aquellas que descartamos anteriormente no nos interesan.

En el anterior gráfico podemos apreciar los boxplots de cada columna, podemos observar valores atípicos en las columnas de ParentalEducation y de Music.

Sin embargo, esas columnas son algo particulares, a continuación veremos por qué.

Histogramas

Muy probablemente no haga falta el uso de los siguientes histogramas, pero la idea es dejar bien en claro por qué dejaremos los valores atípicos en la base de datos.



Evidentemente, no es tan útil usar histogramas en este caso. Sin embargo, sirve para justificar el porqué de dejar los valores atípicos.

Si recordamos, la variable Music es una variable binaria de tipo categórica (0 o 1) que nos indica si el estudiante participa de actividades musicales o no. Claramente no aporta mucho intentar aplicar un análisis o interpretación con histogramas, pero algo queda en claro: la mayoría de estudiantes no participan en música, por ende este valor “1” queda como poco frecuente frente a “0”, y pasa a percibirse como un valor atípico en nuestros boxplot. Entonces, sí, desde una perspectiva estadística, es un “valor atípico”, pero la realidad es que no es nada raro, simplemente es menos común.

Por otro lado, la variable ParentalEducation posee 5 valores ordenados (0,1,2,3,4) para clasificar la educación de los padres del estudiante. Al ser una variable de tipo categórica, los valores si tienen un orden lógico (0-4) pero no necesariamente una distribución continua. Esto tiene todo el sentido del mundo, porque estadísticamente pocas personas alcanzan un nivel de educación superior tan alto (4). Esto hace que nuevamente en los boxplots los alumnos con padres que alcanzaron una educación superior muy alta resalten como valores atípicos, pero tan solo es un reflejo de la realidad de la muestra.

Por estas razones es que consideramos dejar estas variables como están, con sus valores atípicos.

Transformaciones Preliminares

En este caso, afortunadamente, no es necesario realizar algún tipo de transformación a los datos, ya que están convertidos a valores numéricos aptos para el desarrollo de una red neuronal.
Esto quiere decir que pueden utilizarse sin problemas para realizar operaciones matriciales de forma optimizada, y están determinados siguiendo una determinada lógica (por ejemplo, un valor más bajo de clase implica un mejor rendimiento, y no es un número asignado arbitrariamente).

La única transformación lineal que aplicaremos será a la columna GPA, que deberá pasarse a valores de 0 (bueno) o 1 (malo) para hacer que nuestro modelo sea justamente el de una red neuronal de clasificación binaria.

Además, por las razones explicadas, se eliminan las columnas de las variables que no serán tenidas en cuenta para la creación de la red neuronal.
A continuación se muestra una comparación de una parte de la base de datos original con la transformada:

	StudentID	Age	Gender	Ethnicity	ParentalEducation	StudyTimeWeekly	Absences	Tutoring	ParentalSupport	Extracurricular	Sports	Music	Volunteering	GPA	GradeClass
0	1001	17	1	0	2	19.833723	7	1	2	0	0	1	0	2.929196	2.0
1	1002	18	0	0	1	15.408756	0	0	1	0	0	0	0	3.042915	1.0
2	1003	15	0	2	3	4.210570	26	0	2	0	0	0	0	0.112602	4.0
3	1004	17	1	0	3	10.028829	14	0	3	1	0	0	0	2.054218	3.0
4	1005	17	1	0	2	4.672495	17	1	3	0	0	0	0	1.288061	4.0
5	1006	18	0	0	1	8.191219	0	0	1	1	0	0	0	3.084184	1.0
6	1007	15	0	1	1	15.601680	10	0	3	0	1	0	0	2.748237	2.0
7	1008	15	1	1	4	15.424496	22	1	1	1	0	0	0	1.360143	4.0
8	1009	17	0	0	0	4.562008	1	0	2	0	1	0	1	2.896819	2.0
9	1010	16	1	0	1	18.444466	0	0	3	1	0	0	0	3.573474	0.0

Primeros 10 valores de base de datos original

	Ethnicity	ParentalEducation	StudyTimeWeekly	Absences	Tutoring	ParentalSupport	Extracurricular	Sports	Music	GPA
0	0	2	19.833723	7	1	2	0	0	1	0.0
1	0	1	15.408756	0	0	1	0	0	0	0.0
2	2	3	4.210570	26	0	2	0	0	0	1.0
3	0	3	10.028829	14	0	3	1	0	0	0.0
4	0	2	4.672495	17	1	3	0	0	0	1.0
5	0	1	8.191219	0	0	1	1	0	0	0.0
6	1	1	15.601680	10	0	3	0	1	0	0.0
7	1	4	15.424496	22	1	1	1	0	0	1.0
8	0	0	4.562008	1	0	2	0	1	0	0.0
9	0	1	18.444466	0	0	3	1	0	0	0.0

Primeros 10 valores de base de datos transformada

Parte 2 - Desarrollo de la Red Neuronal

En esta segunda parte del informe, nuestro objetivo es documentar y explicar paso a paso cómo implementamos nuestra red neuronal.

Cada paso tiene un porqué y nada está ahí simplemente por estar, por lo que ésta práctica es excelente para comprender qué es lo que pasa dentro de una red neuronal, cómo está compuesta la misma y cómo es que es capaz de “predecir” algo.

Dividiremos el desarrollo de la red en las siguientes partes:

- **Arquitectura de la Red**
    - Esquema de la Red Neuronal
    - Capas y neuronas de la Red
    - Funciones de Activación

- **Implementación en Numpy**
    - Inicialización de los pesos
    - Cálculo de las activaciones en cada capa
    - Función de Costo
    - Forwardpropagation y Backpropagation
    - Ajuste de pesos

- **Entrenamiento, Evaluación y Overfitting**
    - Entrenando la Red
    - Curvas de precisión y función de pérdida
    - Análisis de overfitting
    - Prevenir overfitting

Apoyándonos en los puntos anteriormente mencionados cubriremos todo el desarrollo de nuestra red neuronal.

Arquitectura de la Red

En este punto nuestro objetivo es ilustrar qué tipo de esquema o estructura tendrá nuestra red. Claramente esto no es algo aleatorio, tiene una razón de ser.

A grandes rasgos, podemos pensar una red neuronal como algo a lo que le podemos arrojar un conjunto de variables de entrada, realiza algunos cálculos (que parecen algún tipo de magia cuando no se comprende qué pasa ahí) y en base a ellos arroja una predicción.

En nuestro caso, daremos variables de entrada que corresponden a un estudiante (por ejemplo ausencias, apoyo parental, etc) y la red tomará esos valores, los procesará y devolverá una predicción (bueno o malo).

Entonces ahí surge el concepto de capas. Una capa es la de entrada, otra la oculta y finalmente tenemos la de salida.

Ahora, cada capa está compuesta por neuronas, gracias a ellas es que podemos interconectar toda la red.

En la **primera capa** es lógico pensar que tendremos la misma cantidad de neuronas que la cantidad de variables de entrada.
Ya para las **capas ocultas** puede empezar a confundir cuántas capas o cuántas neuronas por capa debemos tener, la verdad es que es algo más de prueba y error que otra cosa: algunas configuraciones funcionarán mejor y otras peor con la red, es simplemente cuestión de ir probando o tener experiencia en el tema.
Por último está la **capa de salida** donde claramente estará la predicción de nuestro modelo.

Esquema de la Red Neuronal

En nuestro caso estamos realizando una red neuronal básica de clasificación binaria, por lo que dispondremos de la siguiente distribución de capas y neuronas

Como ya mencionamos en la parte 1, específicamente en la parte del análisis de correlaciones, tendremos columnas que dejaremos de lado, columnas que sí nos interesan y por último la columna objetivo.

Repasando, a continuación las columnas que sí nos interesan, y que por ende estarán como entrada en nuestra red neuronal:

- Absences
- ParentalSupport
- StudyTimeWeekly
- Tutoring
- Extracurricular
- Sports
- Music
- ParentalEducation
- Ethnicity

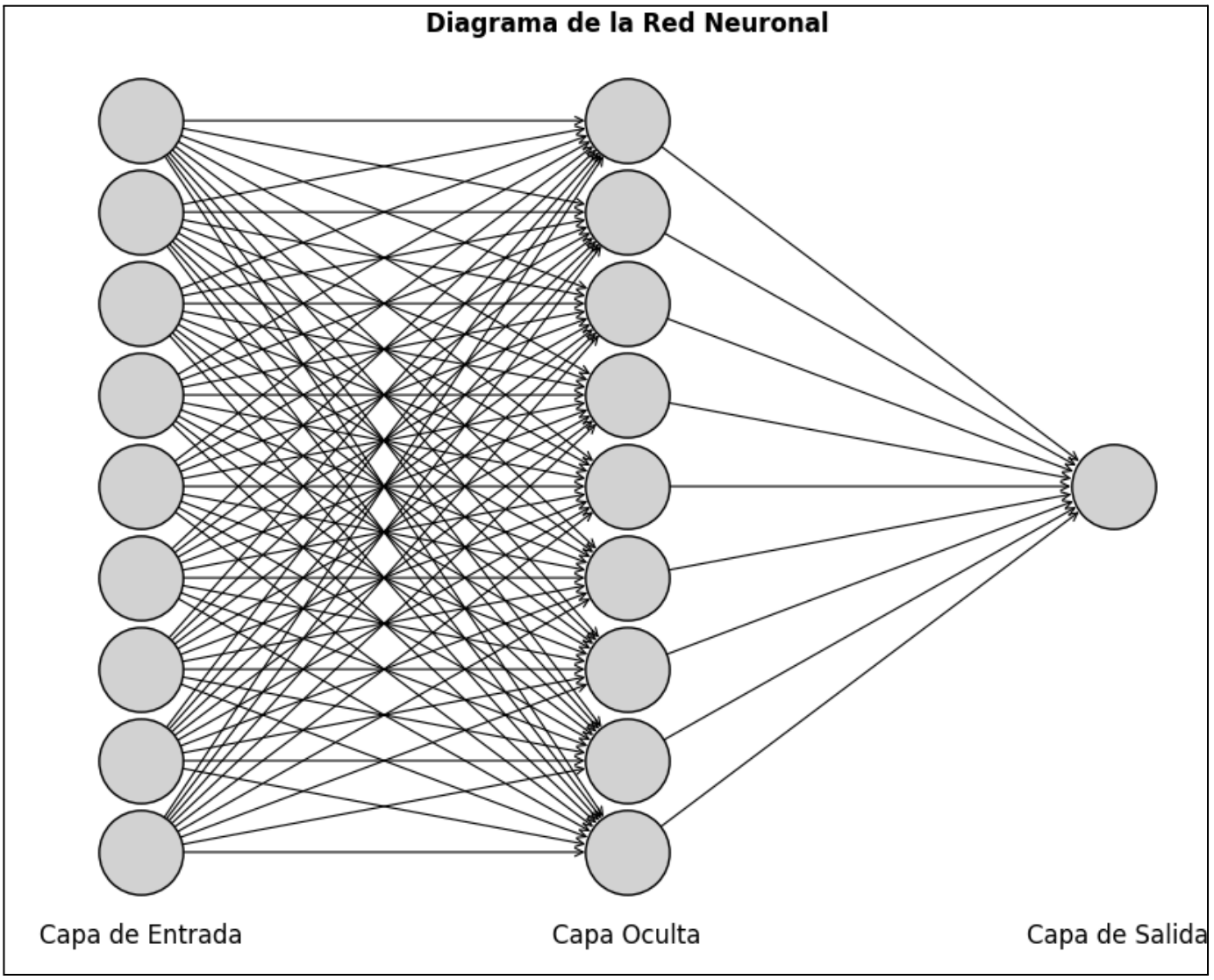
Evidentemente contamos con nueve variables de entrada, por lo que nuestra capa de entrada estará conformada por nueve neuronas. Con respecto a la capa oculta, por una simple decisión de diseño, tendrá nueve neuronas de igual manera, y por último, la capa de salida dispondrá de una sola neurona.

En resumen tenemos lo siguiente:

- **Capa de Entrada:** Es la que se encarga de recibir las variables numéricas que representan los datos de entrada para el modelo. Dispone de nueve neuronas.

- **Capa Oculta:** Es donde se toman esas entradas y se procesa la información realizando cálculos internos del modelo (lo que llamábamos la *mystery math*), básicamente combinaciones y transformaciones de datos. Igual que la anterior capa, dispone de nueve neuronas.

- **Capa de Salida:** Es la última capa, dispone de una sola neurona y su objetivo es el de dar la predicción final.

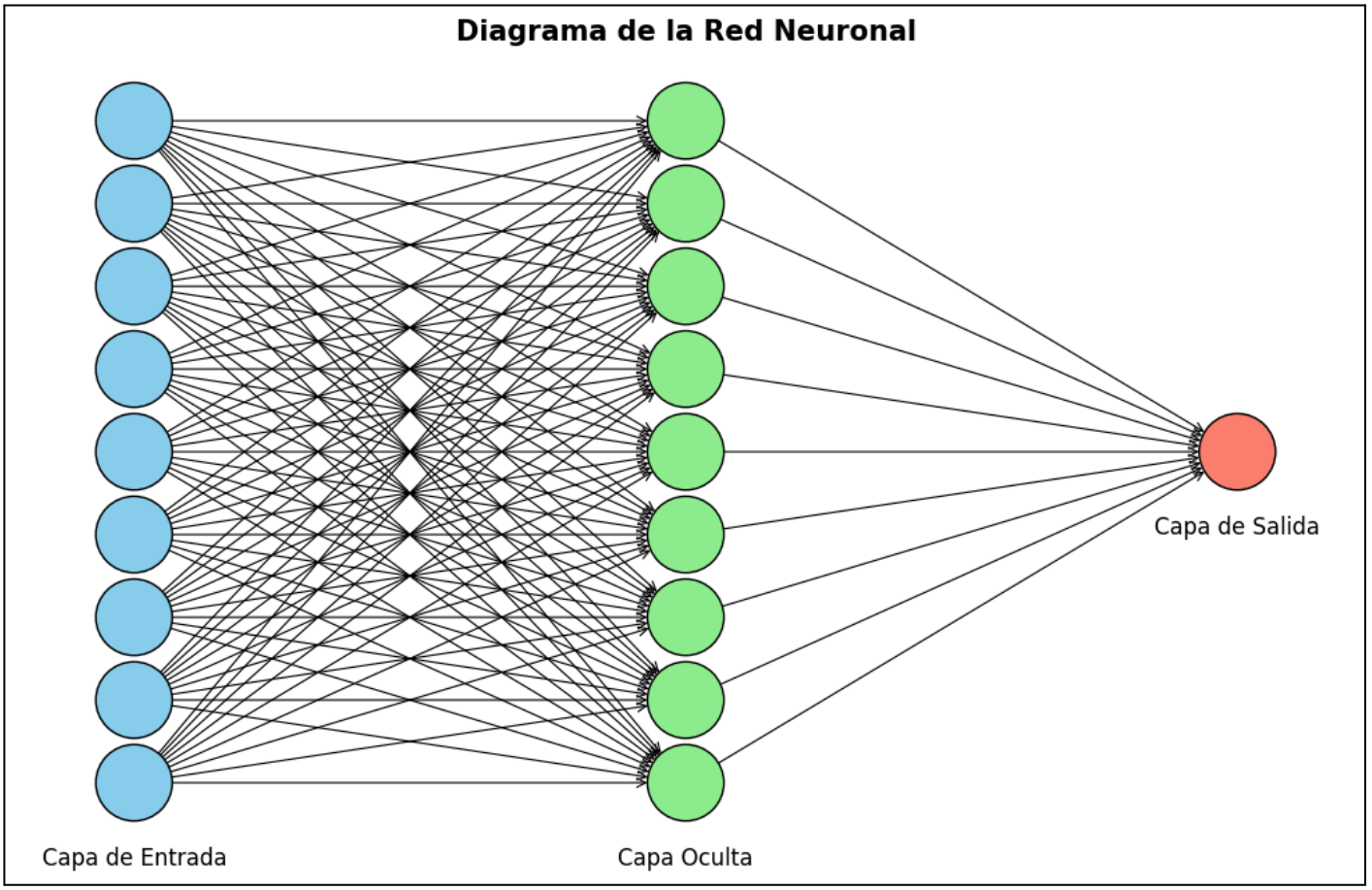


Para que se pueda interpretar mejor, tendremos el siguiente esquema:

Ahora teniendo todo lo anteriormente mencionado en mente, podemos seguir explicando el resto de la arquitectura de la red.

Capas y Neuronas de la Red

Si bien anteriormente expusimos un esquema de la red, a continuación lo volveremos a hacer pero con colores para poder distinguir dónde estamos parados (en una explicación posterior).

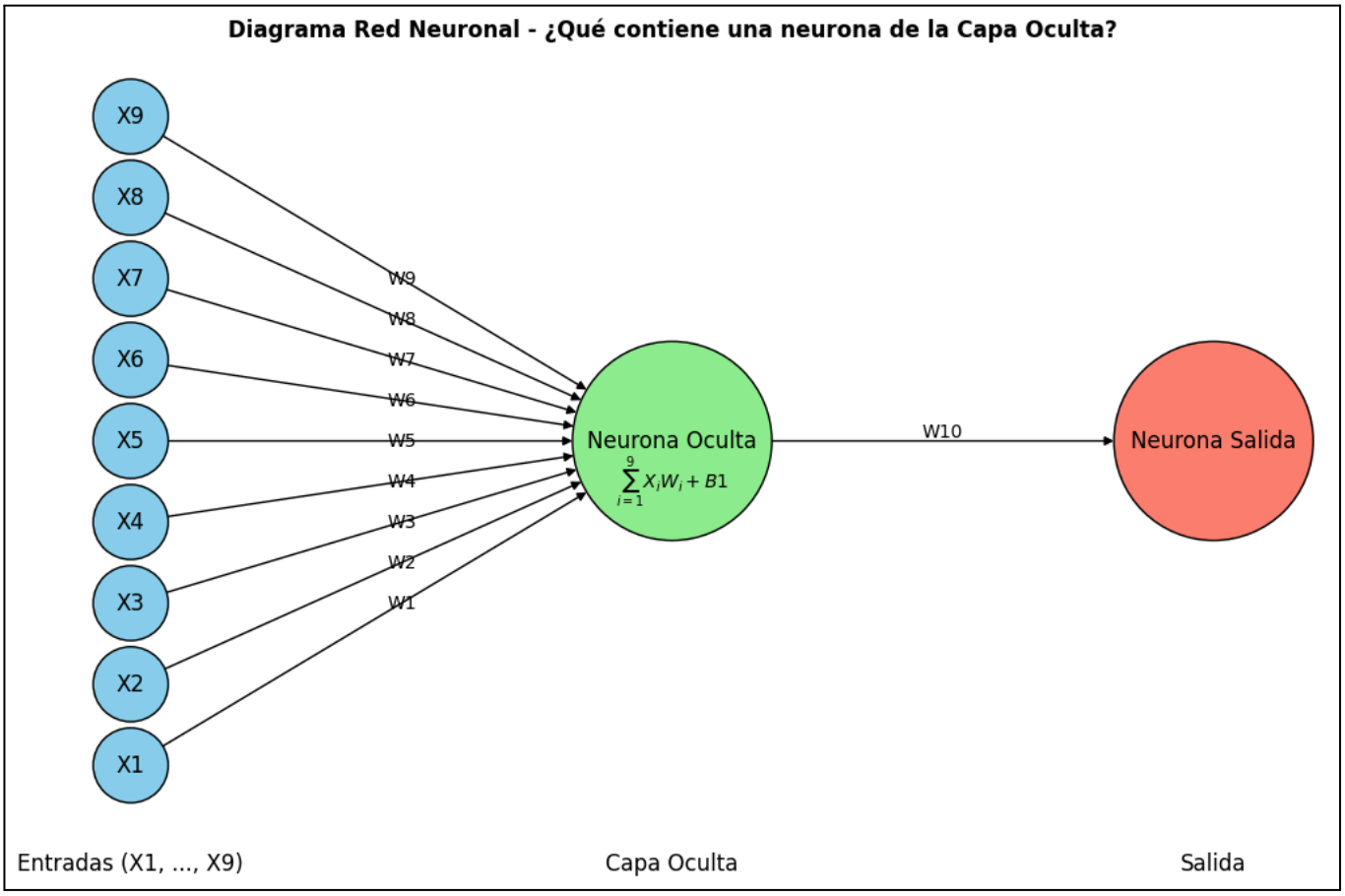


Entonces, vamos de alguna forma a hacer el recorrido que tendría que suceder para que obtengamos una predicción (algo así como el forwarding).

Primero que nada, tenemos la capa de entrada que es la que recibe las variables de entrada, está compuesta por nueve neuronas. Una vez que las tenemos, pasamos a conectar exactamente todas las neuronas de la capa de entrada con cada neurona de la capa oculta (todas a cada una).
Debemos recordar, como se ve en la imagen, que nuestra capa oculta está compuesta por nueve neuronas de igual forma.
Por último, estas nueve neuronas van a terminar saliendo hacia la capa de salida, la cual está compuesta por una única neurona, pues allí se encontrará nuestra predicción (bueno/malo).

El uso de una sola neurona en la salida no es casualidad. Si tuviésemos una red neuronal no de clasificación binaria sino de multi-clasificación (por ejemplo, clasificar al estudiante en muy malo, malo, normal, bueno, muy bueno, etc) deberíamos tener más neuronas de salida, y a su vez cambiaría la *función de activación.* Sobre esto último volveremos más adelante igualmente.

Para ilustrar mejor el punto de que todas las neuronas de entrada están conectadas a cada neurona de la capa oculta podemos observar el siguiente gráfico:



Acá podemos ver claramente que todas las neuronas de la capa de entrada se conectan a una neurona de la capa oculta (hacen exactamente lo mismo pero con todas, no lo graficamos porque sería graficar el esquema de la red prácticamente, donde puede que no se entienda del todo de primeras).

Entonces, todas esas entradas entran a cada neurona, esa neurona maneja cálculos matemáticos (sumas y productos) y vuelve a tener una salida hacia la neurona de salida.

Precisamente, lo que tenemos dentro de esa neurona, es lo siguiente:

$$X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 + X_5W_5 + X_6W_6 + X_7W_7 + X_8W_8 + X_9W_9 + B1$$

En la neurona de salida, como sí se puede apreciar más claramente en el gráfico de esquema de la red neuronal, se recibe una entrada de cada neurona de la capa oculta (no hay tanta confusión).

Sin embargo, vemos algunas cosas más en los gráficos: pesos **W** y sesgos **B**. Estos son elementos claves que nos permiten ajustar el modelo y permitir que “aprendan” patrones de entrada (de esta forma pueden “predecir” la salida).
Por ahora tenemos que tener en claro que los **pesos** son valores numéricos que asignamos a cada conexión entre neuronas en diferentes capas (como se aprecia claramente en el gráfico), y los **sesgos** son un valor que sumamos al cálculo de la neurona después de haber hecho la suma de los productos entre cada peso y cada entrada de la red (pueden evitar que una neurona quede completamente apagada en caso de que los cálculos con los pesos den cercanos a 0).

Estos pesos y sesgos tendrán valores pseudo-aleatorios en un principio, y se irán ajustando iterativamente a medida que la red aprende, para que el modelo se ajuste mejor a los datos y pueda tener más capacidad de predicción.

Explicaremos con más detalle estos puntos más adelante.

Funciones de Activación

Las funciones de activaciones en modelos de redes neuronales son extremadamente importantes. Si no hiciéramos uso de ellas, básicamente nuestro modelo de red neuronal sería una especie de regresión lineal gigante: no podría llegar a aprender patrones de relaciones más complejas (no tan lineales) en los datos.

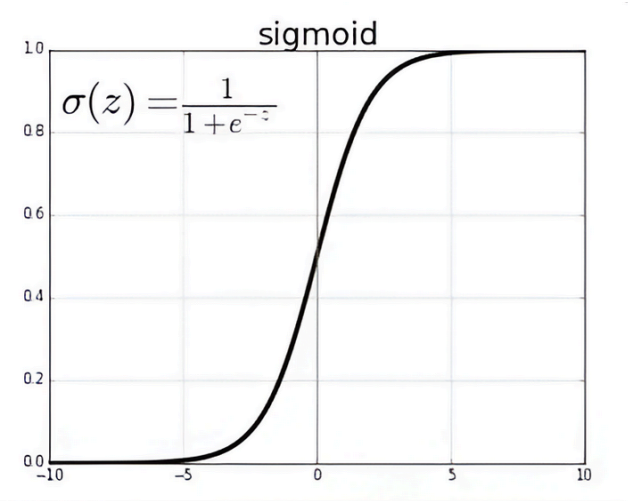
Y no sólo eso, sino que ayudan con la problemática de la propagación de valores (es decir, evitar que manejemos valores extremadamente gigantes debidos a los cálculos matemáticos que se llevan a cabo) y por ende terminan mejorando también la eficiencia en los cálculos.

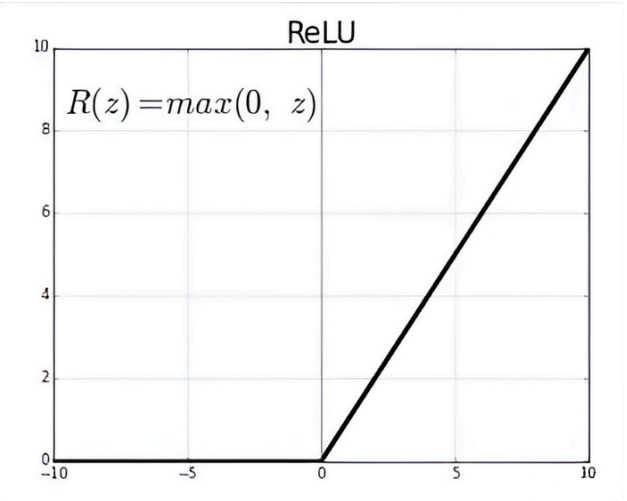
Por último, en algunos casos, ayudan a que una neurona se “active” sólo cuando es relevante. Esto ahorra procesamiento a la red y suma a la eficiencia. Además, este apagado en las neuronas le sirve a la misma red para que identifique más fácilmente patrones y características en los datos (en lugar de aprender todos los detalles y terminar teniendo overfitting).

Ahora, volviendo a nuestra red, manejaremos dos tipos de funciones de activación, y cada una tendrá su propósito.

En la **capa oculta** tendremos una función de activación de tipo **ReLU.** ReLU o Rectified Linear Unit es una función de activación que convierte todas las entradas negativas a cero, por ende sólo mantiene los valores positivos. Es simple y eficiente.

En la **capa de salida** tendremos una función de activación de tipo **Logística.** Funciona transformando cualquier valor de entrada en un número entre 0 y 1. Esto es extremadamente útil en nuestro caso, clasificación binaria, nos permite interpretar el resultado sin ningún tipo de problema.





Una función de activación puede fortalecer, dejar como está o debilitar una señal.

Una vez que se pasó el valor ponderado, sesgado y sumado de un nodo a través de alguna de las funciones de activación que mencionamos (o cualquier otra) pasamos a decir que es una **salida activada.** Básicamente ahora esta salida fue filtrada a través de la función de activación, y esta señal está lista para ser tomada por la siguiente capa.

Implementación en Numpy

Construiremos una red neuronal básica ayudándonos solamente de la librería Numpy, por lo que vamos a tener que desarrollar cada uno de los componentes fundamentales a la hora de crear un modelo de red neuronal de este tipo.
Estos componentes están conformados por:

- Inicialización de los pesos
- Cálculo de las activaciones de cada capa
- Función de costo
- Backpropagation

Conceptos previos

A continuación hablaremos de pesos, sesgos, matrices, tamaño de las mismas, backpropagation y forwardpropagation. Sin embargo, debemos no solo tener en cuenta la arquitectura que tendrá el modelo sino también algunos conceptos sobre las matrices que utilizaremos para realizar los cálculos.

Como dijimos, vamos a utilizar Numpy, específicamente arrays multidimensionales para toda la parte de cálculo en la red neuronal. Esta matemática será la que nos permitirá hacer un backpropagation, forwardpropagation, entrenar la red, usar la red y demás.

Como ya sabemos, la capa oculta tendrá nueve entradas, será una matriz de 9x1 (también se puede pensar como un vector de 9 elementos).

Ahora, la capa oculta deberá recibir esos valores, por lo que tendrá una matriz de pesos 9x9 asociada, además de los sesgos que serán de tamaño 9x1. Acá es importante volver a recordar que eso sería sin activar, con la salida activada tendríamos exactamente lo mismo pero metido dentro de la función de activación (en esta capa, ReLU).

En la capa de salida nuevamente volvemos a recibir valores (esta vez de la capa oculta, ya están activados), dispondremos de una matriz de pesos 1x9 y de sesgos una 1x1. Tomamos todo esto y lo volvemos a pasar por una función de activación (en esta capa, una logística). En este caso, con la salida activada, ya tendremos nuestra predicción.

Usaremos la siguiente nomenclatura para que sea todo más claro:

$$W_n = \text{Pesos}$$

$$B_n = \text{Sesgos}$$

$$Z_n = \text{Salida sin activar}$$

$$A_n = \text{Salida activada}$$

Teniendo esto en cuenta, pasamos a definir cada una:

$$Z_1 = \left[W_1\right]_{9x9} \left[X\right]_{9x1} + \left[B_1\right]_{9x1}$$

$$A_1 = ReLU(\left[Z_1\right]_{9x1})$$

$$Z_2 = \left[W_2\right]_{1x9} \left[A_1\right]_{9x1} + \left[B_2\right]_{1x1}$$

$$A_1 = Sigmoid(\left[Z_2\right]_{1x1})$$

Entonces, todo lo que escribimos anteriormente queda de forma clara y concisa en las ecuaciones anteriores. Cabe destacar que aclaramos el tamaño de cada una primero porque es importante, y en segundo lugar para no poner matrices enteras.

Para que nos demos una idea, la matriz W1 (es decir, solamente los pesos de la capa oculta) al ser una matriz 9x9 tendría la siguiente pinta:

$$W_1 = \begin{bmatrix} w_1 & w_2 & \dots & w_9 \\ w_{10} & w_{11} & \dots & w_{18} \\ \vdots & \vdots & \ddots & \vdots \\ w_{73} & w_{74} & \dots & w_{81} \end{bmatrix}$$

Inicialización de los pesos

¿Qué significa inicializar los pesos? Si recordamos lo que mencionamos brevemente en la parte de arquitectura de la red, sabemos que determinan la influencia que tiene una neurona de una capa sobre las neuronas de la capa siguiente (porque cada una se conecta a todas las que siguen). Siempre cada conexión entre neuronas tiene un peso asociado.

Es importante inicializarlos correctamente porque, por ejemplo, si todos los pesos se inicializan en el mismo valor (supongamos cero) la red no va a poder aprender de forma adecuada: todas las neuronas recibirán la misma señal y actualizarán los pesos de la misma manera en cada iteración. Nuevamente, no sería algo mucho mejor que una regresión lineal gigante.

Pero entonces, ¿de qué forma los inicializamos? Una práctica común y que resulta muy buena es inicializarlos de forma aleatoria, de esta manera rompemos la simetría y permitimos que cada neurona pueda entrenar/aprender patrones únicos.

Sin embargo, estos valores deben ser limitados a un rango de entre -1 y 1, ya que, si fueran demasiado grandes, dificultarían el proceso de hacer todas las multiplicaciones necesarias, haciendo que dure mucho más.

Con la siguiente porción de código lo que hacemos es inicializar los pesos y sesgos de la red neuronal (tanto los de la capa oculta como los de la salida) de forma aleatoria, con valores de entre -1 y 1:

```
w_hidden = np.random.rand(9, 9) * 2 - 1
w_output = np.random.rand(1, 9) * 2 - 1

b_hidden = np.random.rand(9, 1) * 2 - 1
b_output = np.random.rand(1, 1) * 2 - 1
```

Cálculo de las Activaciones

Como ya se mencionó anteriormente, las activaciones son el resultado de aplicar una función de activación a una combinación lineal de entradas y pesos en una neurona (es decir, meter una salida sin activar en una función de activación, Z1 por ejemplo).

Sin embargo, antes de pasar por la activación justamente debemos realizar el cálculo previo. El mismo ya se detalló anteriormente en la parte de **conceptos previos,** a continuación cómo implementamos lo mismo pero en código, haciendo uso de Numpy.

Primero definimos las funciones, luego podemos ver cómo se utilizan a la hora de activar las salidas.

```
relu = lambda x: np.maximum(x, 0)
logistic = lambda x: 1 / (1 + np.exp(-x))
```

```
Z1 = w_hidden @ X + b_hidden
A1 = relu(Z1)
Z2 = w_output @ A1 + b_output
A2 = logistic(Z2)
```

Anteriormente ya explicamos brevemente por qué las utilizamos, pero a continuación un breve resumen.

Sin estas funciones de activaciones, la red sería equivalente a una regresión lineal gigante, por así decirlo. No sería capaz de capturar relaciones complejas entre las variables. Nos ayudan a evitar mucha linealidad, lo que permite que la red se adapte a valores más complejos en los datos.

En la capa oculta usamos la *función de activación ReLU* en específico porque funciona perfectamente en ese escenario, directamente devuelve 0 para valores negativos y deja el valor original para los positivos. Es rápida, puede ayudar a mitigar problemas de gradientes pequeños (con gradientes tan bajos, los ajustes se vuelven mínimos y el aprendizaje a la hora del entrenamiento se vuelve muy lento, inclusive llegando a detenerse) y también el hecho de devolver cero ante un valor negativo suma mucho porque se “apaga” esa neurona para ese patrón en particular.

Estos beneficios permiten que la red aprenda valores específicos, prestando atención a neuronas relevantes y dejando de lado por así decirlo a las que no son necesarias en un determinado patrón.

Por otro lado, la *función de activación logística* es justo lo que necesitamos en la capa de salida. Transforma cualquier valor de entrada en un número entre 0 y 1. Asigna valores altos a valores de entrada positivos y valores cercanos a cero para valores negativos.
Nos es extremadamente útil en este caso, es decir, un modelo de clasificación binaria.

Función de Costo

La función de costo o función de pérdida nos ayuda a medir que tan bien se está desempeñando nuestra red neuronal en su tarea de predicción.
Es la medida de la diferencia entre las predicciones de la red y los valores reales (valores de entrenamiento que estamos seguros son correctos) del conjunto de datos.

Por esta misma razón es la que dividimos una porción de nuestra base de datos, para la parte del entrenamiento. Ahí es donde utilizamos la función de Costo y split train/test, es decir, entrenamos con ciertos datos y la probamos con otros (que la red no conoce).

En síntesis el objetivo del Costo es proporcionar un valor que se minimice durante el proceso de entrenamiento.

La función de Costo que usamos en nuestro caso es la del error cuadrático medio, o MSE, la cual se utiliza en problemas de regresión. Se calcula como la media de las diferencias al cuadrado entre las predicciones y los valores reales.
Sin embargo, dada la complejidad de nuestro modelo, se ve reducida a simplemente la siguiente ecuación:

$$C = (A_2 - Y)^2$$

Donde C representa el Costo, A_2 la salida predicha por la red y Y el valor real.

Algoritmos de Forwardpropagation y Backpropagation

Antes de meternos con el Backpropagation y el descenso de gradiente estocástico, vamos a hablar sobre el Forwardpropagation.

El **Forwardpropagation** es literalmente ir hacia adelante en nuestra red, hasta llegar al output. Es decir, es el primer paso en el ciclo del entrenamiento de una red neuronal.
En este proceso se pasan los datos de entrada a través de la red, para calcular la salida (la cual tendrá una determinada precisión, no es algo que nos interese en este momento ya que eventualmente la entrenaremos y ésta mejorará).

Entonces, tomaremos las variables de entrada de la red y luego aplicamos el cálculo de las activaciones correspondientes.
Todo esto lo podemos ilustrar con las ecuaciones que detallamos arriba, en *conceptos previos***.**
Entonces, cada neurona de la capa oculta de la red recibe las entradas, aplica los pesos correspondientes, suma el sesgo se toma la salida para activarla y pasarla a la capa oculta (la cual hace exactamente lo mismo, pero con distinta función de activación).

Esta implementación en código para nuestra red neuronal se verá de la siguiente forma:

```
def forward_prop(X):
    Z1 = w_hidden @ X + b_hidden
    A1 = relu(Z1)
    Z2 = w_output @ A1 + b_output
    A2 = logistic(Z2)
    return Z1, A1, Z2, A2
```

Este proceso, el de propagar datos, se repite capa por capa (no importa las que tengamos) hasta llegar a la capa de salida, donde ya habrá recorrido todas las capas y nodos de la red. Finalmente se produce la predicción de la red.

Acá es donde podemos comparar nuestro resultado para ver qué tan precisa es la red, utilizando la función de Costo. Si este valor está muy mal, podemos proceder a entrenar a la red.

Ahora bien, ¿qué hacemos si no tenemos una precisión muy buena? ¿Y si quiero mejorarla? Como dijimos anteriormente, la precisión de la red, aparte de la arquitectura de la misma, dependerá fuertemente de qué tan bien esté entrenada, por ende depende de qué tan bien estén ajustados todos y cada uno de los pesos y sesgos del modelo.

Justamente la función del **Backpropagation** es la de ir ajustando los pesos y sesgos de la red en función de cómo la red se desempeñó en la última iteración (porque vamos a hacerlo reiteradas veces, cuantas sean necesarias y con un L óptimo): utilizaremos el **descenso de gradiente estocástico**.

El problema es el siguiente, ¿cómo hacemos para ajustar eso? Bueno, justamente por eso es necesario entender lo que detallamos arriba en **conceptos previos.** Si recordamos, podemos representar absolutamente todo el Forwardpropagation con esas ecuaciones, entonces también partiendo desde la predicción final podemos hacer uso de las derivadas sucesivas y regla de la cadena para llegar hasta el comienzo de la red, y una vez allí (y en cada paso también) ir ajustando todos los pesos y sesgos que necesitemos.

Nos podemos dar cuenta que vamos en el sentido contrario al Forward Propagation: en este último avanzamos desde las entradas hasta la capa final, en el Back Propagation vamos desde el final hasta el comienzo.

Es decir, antes de poder aplicar un descenso de gradiente estocástico vamos a necesitar tener todas las derivadas parciales. Para comprender por qué esto es tan eficiente y conveniente, podemos ilustrarlo de la siguiente manera:

$$C = (A_2 - Y)^2$$
$$\downarrow$$
$$A_2 = sigmoid(Z_2)$$
$$\downarrow$$
$$Z_2 = W_2 A_1 + B_2$$
$$\downarrow$$
$$A_1 = ReLU(Z_1)$$
$$\downarrow$$
$$Z_1 = W_1 X + B_1$$

Por esta razón tenemos la analogía de tener un anidamiento similar al de una cebolla, tenemos cosas que están dentro de otras, que a su vez dependen de otras, y así sucesivamente.

Finalmente, las derivadas parciales serán las siguientes:

$$\frac{dC}{dW_2} = \frac{dZ_2}{dW_2}\frac{dA_2}{dZ_2}\frac{dC}{dA_2} = (A_1)\left(\frac{e^{-Z_2}}{1+e^{-Z_2}}\right)(2A_2 - 2y)$$

$$\frac{dC}{dB_2} = \frac{dZ_2}{dB_2}\frac{dA_2}{dZ_2}\frac{dC}{dA_2} = (1)\left(\frac{e^{-Z_2}}{1+e^{-Z_2}}\right)(2A_2 - 2y)$$

$$\frac{dC}{dW_1} = \frac{dC}{dA_2}\frac{dA_2}{dZ_2}\frac{dZ_2}{dA_1}\frac{dA_1}{dZ_1}\frac{dZ_1}{dW_1} = (2A_2 - 2y)\left(\frac{e^{-Z_2}}{1+e^{-Z_2}}\right)(W_2)(Z_1 > 0)(X)$$

$$\frac{dC}{dB_1} = \frac{dC}{dA_2}\frac{dA_2}{dZ_2}\frac{dZ_2}{dA_1}\frac{dA_1}{dZ_1}\frac{dZ_1}{dB_1} = (2A_2 - 2y)\left(\frac{e^{-Z_2}}{1+e^{-Z_2}}\right)(W_2)(Z_1 > 0)(1)$$

Ajuste de pesos

Una vez que tengamos todo lo anterior resuelto (es decir, estamos en condiciones de poder realizar un backpropagation), podemos comenzar a aplicar el descenso de gradiente estocástico en nuestra red.

Es clave destacar que el descenso de gradiente no es uno solo, tenemos por lotes, mini lotes y estocástico. Nosotros utilizaremos este último. Su función básicamente es la de encontrar los valores óptimos para los pesos y sesgos en una red neuronal, con el objetivo de que minimicemos la *función de Costo.*

En el descenso de gradiente estocástico vamos a tener un proceso iterativo donde utilizamos una sola muestra de un conjunto de datos (ese conjunto será el de entrenamiento, como habíamos detallado anteriormente).
Luego, se calcula el gradiente de la función usando este ejemplo (es decir, la muestra tomada) y se ajustan los parámetros del modelo en dirección opuesta al gradiente, pues buscamos minimizar el error o costo.

La implementación del código de backpropagation se ve representada de la siguiente forma:

```python
def backward_prop(Z1, A1, Z2, A2, X, Y):
    dC_dA2 = 2 * A2 - 2 * Y
    dA2_dZ2 = d_logistic(Z2)
    dZ2_dA1 = w_output
    dZ2_dW2 = A1
    dZ2_dB2 = 1
    dA1_dZ1 = d_relu(Z1)
    dZ1_dW1 = X
    dZ1_dB1 = 1

    dC_dW2 = dC_dA2 @ dA2_dZ2 @ dZ2_dW2.T

    dC_dB2 = dC_dA2 @ dA2_dZ2 * dZ2_dB2

    dC_dA1 = dC_dA2 @ dA2_dZ2 @ dZ2_dA1

    dC_dW1 = dC_dA1 @ dA1_dZ1 @ dZ1_dW1.T

    dC_dB1 = dC_dA1 @ dA1_dZ1 * dZ1_dB1

    return dC_dW1, dC_dB1, dC_dW2, dC_dB2
```

Entonces, en resumen, vamos a tener una determinada cantidad de iteraciones, con un determinado coeficiente de aprendizaje L (es el que determina cuánto nos movemos en dirección opuesta al gradiente) donde seleccionamos una muestra de un conjunto de datos (el de entrenamiento) para ir entrenando a la red (que es ir viendo cuánto se equivoca y corregir ajustando pesos y sesgos).

Finalmente, el descenso de gradiente queda implementado de la siguiente manera:

```python
# Función que realiza forward y backward, para entrenar la red
def aplicarRed(L, rango):
    global w_hidden, w_output, b_hidden, b_output
    for i in tqdm(range(rango)):
        # Selecciona aleatoriamente uno de los datos de entrenamiento
        idx = np.random.choice(n, 1, replace=False)
        X_sample = X_train[idx].transpose()
        Y_sample = Y_train[idx]

        # Pasa datos seleccionados aleatoriamente a través de la red neuronal
        Z1, A1, Z2, A2 = forward_prop(X_sample)

        # Distribuir error a través de la retropropagación
        # y devolver pendientes para pesos y sesgos
        dW1, dB1, dW2, dB2 = backward_prop(Z1, A1, Z2, A2, X_sample, Y_sample)

        # Actualiza pesos y sesgos
        w_hidden -= L * dW1
        b_hidden -= L * dB1
        w_output -= L * dW2
        b_output -= L * dB2
```

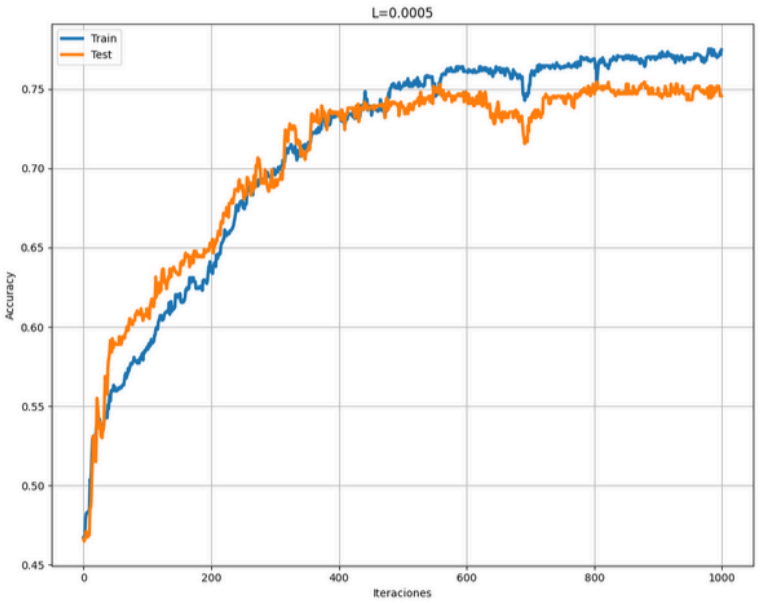
Sin entrar en tantos detalles de código, procedemos a explicar nuestra función anterior:

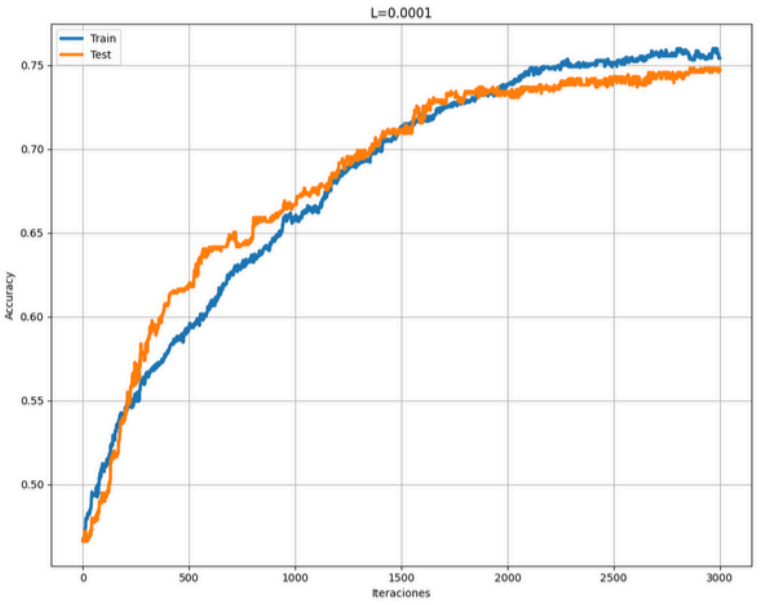
- Como mencionamos anteriormente, es un proceso básico de entrenamiento de una red neuronal, haciendo uso del descenso de gradiente estocástico puro.
- A la función debemos pasarle como parámetros un L (tasa de aprendizaje) y rango (iteraciones).
- La función se encarga de tomar un sample (una sola muestra) del conjunto de entrenamiento para hacer un Forward Prop con la misma (ahí se actualizan las salidas activadas y no activadas). Lo que recibe Forward Propagation es una muestra de entrada (X, todas las variables de entrada necesarias, en nuestro caso 9) y la predicción correcta que corresponde en esta muestra (Y).
- Con Backward Propagation (y después de que Forward haya actualizado Z1,A1,Z2 y A2) calculamos las derivadas parciales del error con respecto a cada peso y sesgo de la red. Acá usamos la regla de la cadena. Distribuimos el error en cada peso y sesgo.
- Luego de que se ejecutó Backward (ya tenemos los gradientes o derivadas parciales del error) procedemos a actualizar los valores de los pesos y sesgos utilizando nuestra tasa de aprendizaje seleccionada. Acá podemos recordar el ejemplo de la curva y cuánto avanzamos hacia el cero (pues queremos minimizar el error).
- Todo este proceso se realiza tantas veces como iteraciones seleccionadas tengamos.

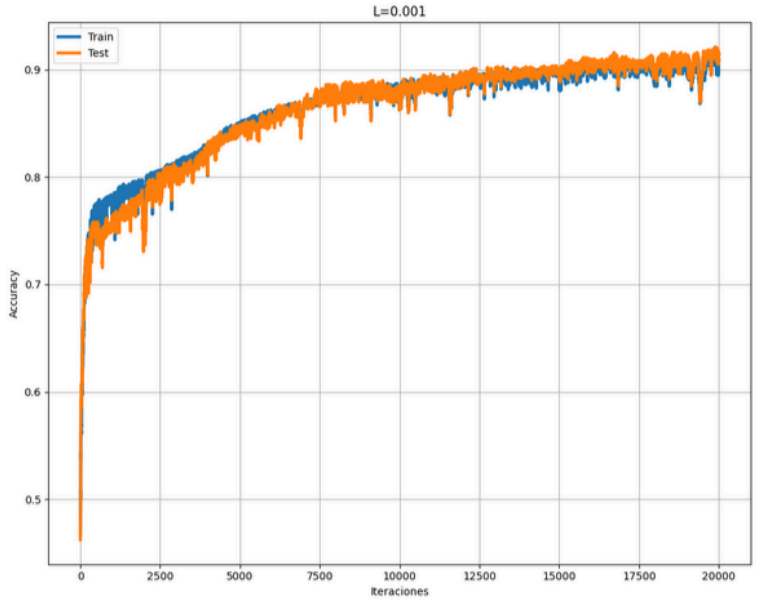
Sin embargo, debemos recordar tener precaución de no caer en un **overfitting**, que básicamente se da cuando ya entrenamos demasiado nuestra red, al punto de que se memorizó los datos y, ante un eventual valor nuevo para la red, prediga cualquier cosa. También es importante que L no sea muy pequeño, para poder llegar al menor valor de costo posible, ni que sea demasiado grande, porque daría pasos demasiado grandes y nunca encontraría este valor.

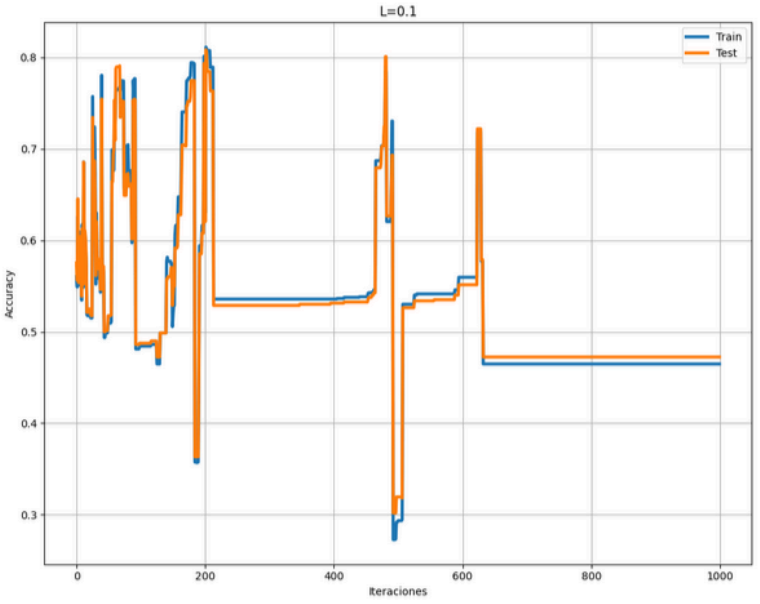
Para poder encontrar los valores más efectivos de L y cantidad de iteraciones, se puede correr la función de entrenamiento de la red, probando distintas combinaciones de estos valores, hasta encontrar una que consiga una buena precisión, y cuya curva de aprendizaje tenga una tendencia a crecer. Para hacer esto, los valores aleatorios de pesos, sesgos y conjuntos de entrenamiento y testeo son fijados de antemano, para conseguir una representación fidedigna de la utilidad de los coeficientes que se quiere encontrar.

Para visualizarlo mejor, por cada iteración de este proceso se grafica los valores de L e iteraciones probadas, y la curva de aprendizaje asociada a ellos. A continuación, algunos ejemplos de combinaciones probadas:









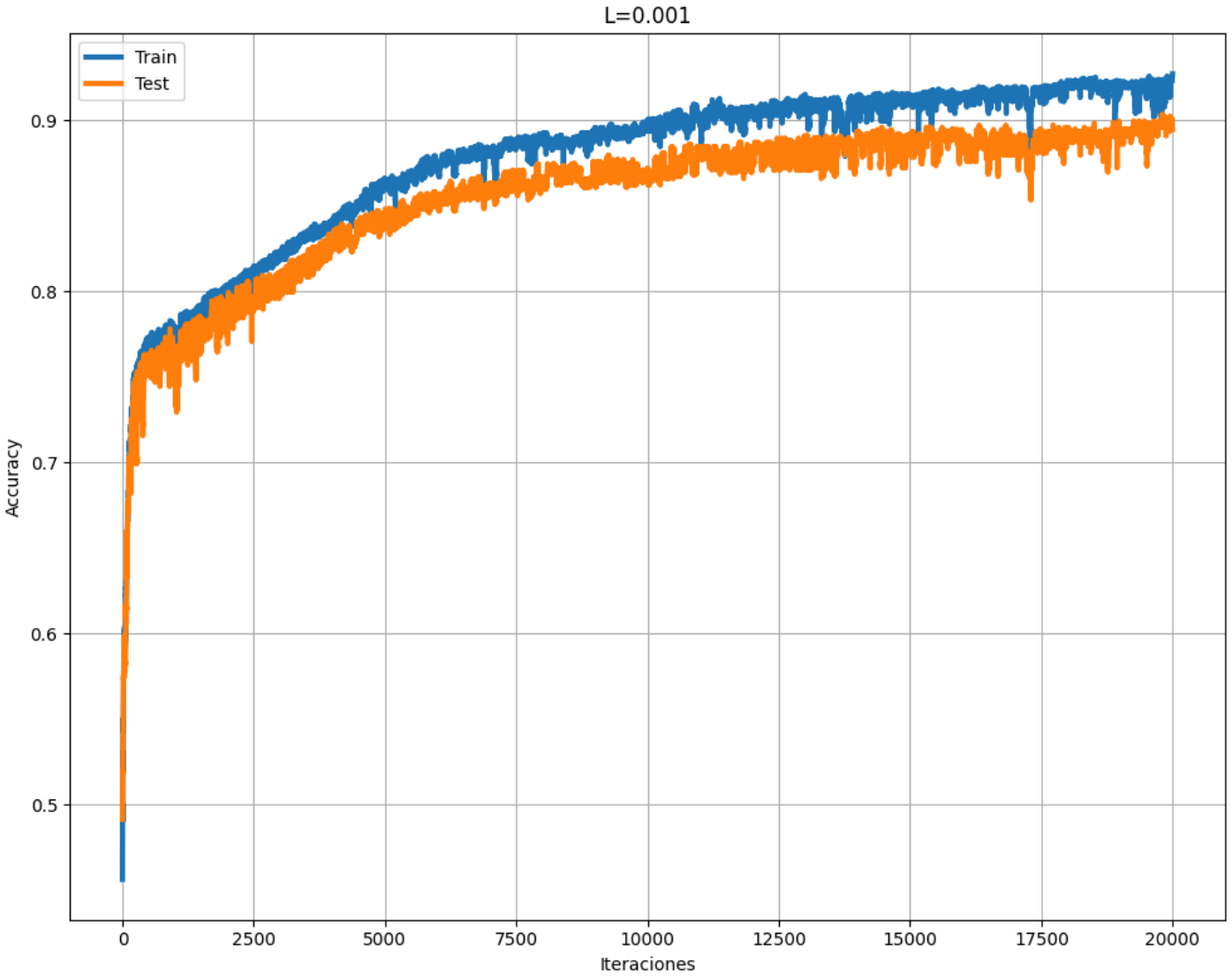
Como se puede ver, en algunos casos la curva de aprendizaje es constante, y en otros prácticamente no se aprecia una mejoría o una lógica a medida que se avanza en el entrenamiento. Analizar estos gráficos nos ayuda, además, a determinar si estamos en presencia de overfitting (si es que la curva de aprendizaje del entrenamiento queda prácticamente en 1, pero la de testeo mucho más abajo, es porque aprendió de memoria el primer conjunto) o underfitting (puede que la red alcance un valor mejor de precisión si aumentamos el número de iteraciones).

Entrenamiento y evaluación

Luego de explicar la teoría y la arquitectura de nuestra red neuronal, además de las funciones necesarias, podemos proceder a poner manos a la obra: es hora de **entrenar** la red.
Para ello, tras probar con muchas configuraciones distintas, llegamos a que la óptima es con los siguientes valores:

- **L (Tasa de aprendizaje)** = 0.001
- **Range (Cantidad de iteraciones)** = 20.000

Podemos hacer uso del gráfico correspondiente a la configuración (donde trazamos las curvas de precisión tanto para el set de entrenamiento como para el de validación) para terminar de explicar la decisión:



Se puede observar un aumento gradual en la precisión, tanto para el train como para el test, lo que indica que la red estaría aprendiendo de forma efectiva, los pesos se están actualizando y se optimiza el rendimiento de la misma.

Si es verdad que hay una leve separación entre test y train, lo que de primeras podría indicar algo de overfitting. La verdad es que las curvas siguen trayectorias muy similares, por lo que de haber overfitting, no sería extremadamente significativo. Caso contrario sería el de que estén muy separadas.

También, llegando al final, podemos ver que las curvas dejan de fluctuar tanto y se estabilizan: esto podría indicar que estamos en un punto óptimo de L, con la cantidad de iteraciones seleccionadas.

Por estas razones, **la consideramos una configuración adecuada** para nuestra red.

Evaluando un poco más podemos ver, por ejemplo, que con esta configuración podemos llegar a obtener una precisión de:

- Train: 95%
- Test 90%

Lo que pasa probablemente en un escenario real es que, además de estar usando nuevos datos, la precisión debería rondar el 90% (basando en la curva de prueba) y no tanto en el 95% (que representa el train).
Esto es algo común, y refleja que el modelo pudo generalizar bien, sin overfitting.

Ahora, si llegáramos a considerar que nuestra red neuronal tiene una precisión bastante mala, ¿Qué podríamos hacer?
Primero que nada, dependiendo del tipo de red que tengamos, y el tipo de modelo que estemos haciendo, vamos a tener distinta cantidad de precisión. Sin embargo, no tenemos con qué compararlo directamente (es algo que sí haremos a continuación con Scikit-Learn).
En el caso de que efectivamente sea mala la precisión, o de que no alcance valores que uno esperaría, algunas razones podrían ser:

- Pocas variables de entrada: Tal vez el rendimiento de un alumno depende de variables que no hayamos tenido en cuenta.
- Cantidad de muestras baja: Depende directamente de las muestras del dataset.
- Datos atípicos, supresión de variables.

Teniendo en cuenta todo lo que hablamos en el informe hasta el momento, podemos dar por descartada totalmente la primera opción (se hizo un análisis exhaustivo y se dejó sólo lo que interesa).
En segundo lugar, nuestro dataset no es necesariamente gigante, pero es suficiente para lo que queremos hacer (no es algo extremadamente complejo, es una red básica de clasificación binaria).
Por último, tal vez podría deberse (en muy pequeña parte) a algún valor atípico, después de todo sí decidimos dejarlos (nuevamente justificado en el informe).

Para estar completamente seguros de que no hicimos muy mal las cosas, podemos utilizar la librería Scikit-Learn para realizar una red neuronal similar, y comparar los resultados obtenidos.

Parte 3 - Comparación con Scikit-Learn

En esta parte del informe se utilizará Scikit-Learn como camino alternativo para desarrollar la red neuronal, con el objetivo de comparar los resultados así obtenidos con el rendimiento de la desarrollada en la parte 2.

Implementación de Scikit-Learn

Scikit-Learn es una librería de Python de uso didáctico para crear redes neuronales, de manera simplificada, con el objeto de sacar resultados rápida y fácilmente.

Claramente nos viene como anillo al dedo en esta situación donde queremos corroborar qué tan bien funciona nuestra red.

La red utilizará los mismos principios explicados anteriormente, pero sus definiciones en código para el usuario final son más sencillas, y toda la lógica del código queda oculta, ya que es llevada a cabo por las funciones propias de la librería. Más de esto vuelve en la conclusión final.

Una vez importados los datos, de forma similar a como se hizo con anterioridad, se definen los valores necesarios para el modelo de aprendizaje, como el número de neuronas, la función de activación, la cantidad de iteraciones y el coeficiente de aprendizaje:

```
# Cargar y preparar datos
dataframe_ = pd.read_csv("students.csv")
dataframe_.drop(columns=["StudentID", "Age", "Gender", "Volunteering", "GradeClass"], inplace=True)
df = dataframe_
np.random.seed(0)
df['GoodStudent'] = (df.iloc[:, -1] >= 2).astype(int)
X = df.values[:, :-1]
Y = df['GoodStudent'].values
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=1/3, random_state=42)
```

Podemos apreciar en el código anterior varias similitudes con lo que hicimos en nuestra red neuronal: se carga el csv, se dropean las columnas que no utilizaremos, podemos la semilla del random en cero, transformamos la columna para definir un buen estudiante, usamos un split train-test, etc.

```
red_neuronal = MLPClassifier(
        solver='sgd', hidden_layer_sizes=(9,), activation='relu',
        max_iter=rango, learning_rate_init=L)
```

En la última imagen también definimos cosas claves: la capa oculta contará con nueve neuronas, con una función de activación tipo ReLU, y si bien no especificamos una sigmoide para la salida, la verdad es que MLPClassifier utiliza la función de activación sigmoide en la capa de salida por defecto para problemas de clasificación binaria.

Cuando se define lo mencionado, ya estamos en condiciones de entrenar la red. Para esto, haremos lo mismo que en la parte del desarrollo de la red neuronal: buscaremos los valores de L y rango, entrenaremos con un conjunto de entrenamiento, y calcularemos la precisión con el conjunto de entrenamiento.
Luego, compararemos los resultados obtenidos en cada una de las formas de proceder.

Para el número de iteraciones y el coeficiente de aprendizaje, claramente deberemos usar los mismos que en el modelo anterior, **L = 0,001** e **Iteraciones = 20.000**, debido a que de esta manera podemos comparar el rendimiento de ambos métodos correctamente.

Luego, se hace el entrenamiento propiamente dicho, usando el método .fit(). En esta parte se realiza el Forward Propagation, el Backward Propagation y la actualización de los pesos y los sesgos, aunque en un lapso de tiempo inferior al de nuestra estructura, debido al uso de funciones optimizadas y específicas para el problema.

Comparación de rendimiento

Por último, al entrenar la red con Scikit-Learn podemos apreciar la siguiente precisión:

```
ACCURACY:  0.5162907268170367
100%|██████████| 20000/20000 [00:01<00:00, 11986.80it/s]
ACCURACY:  0.9235588972430928
Precisión del conjunto de entrenamiento= 0.9422835633626098
Precisión del conjunto de testeo= 0.9461152882205514
```

En el output podemos observar (respectivamente):

- Precisión pre-entrenamiento
- Precisión post-entrenamiento
- Precisión del conjunto train
- Precisión del conjunto test

Esto nos confirma que la red creada manualmente funciona apropiadamente y llega a una buena precisión, sin llegar a tal porcentaje, posiblemente debido a que, aún siendo la base de la red la misma en cuanto a cantidad de neuronas, funciones de activación, etc., los métodos que tiene para calcular son diferentes a los nuestros.

Parte 4 - Conclusión Final

En esta sección se incluirán reflexiones acerca del proceso de creación de una red neuronal, la importancia de aprender cómo funcionan y experimentar su desarrollo, su uso, sus fortalezas y sus limitaciones.

En este caso, al tener como objetivo la clasificación de estudiantes en dos categorías, quizás podría parecer fácil predecir cuándo a un alumno le irá bien en sus estudios: por ejemplo, definiríamos que un buen estudiante dedica una cantidad determinada de horas semanales al estudio, mientras que uno malo no complementa el tiempo que pasa en la escuela con actividades de repaso o estudio independiente. Sin embargo, la realidad es mucho más compleja, y hay muchos factores que entran en juego. Entonces, ¿cómo darse cuenta de cuáles son estos factores que determinan la realidad?

Como es difícil para un humano tener en cuenta tantos patrones que se relacionan entre sí, una de las soluciones a las que se llegó es por medio de las redes neuronales: nos ocuparemos de recopilar la mayor variedad, cantidad y calidad de información posible, y procesamos esa información por medio de un algoritmo de aprendizaje automático.

Sin embargo, sigue sin ser suficiente: aún hay que analizar los resultados obtenidos, y la red neuronal no puede hacerlo. Lo único que hace es ajustarse a un modelo, y según ella, todo lo que aprende es la verdadera realidad.

Por lo tanto, las redes neuronales son útiles para simplificar el proceso del análisis, viendo patrones donde sería difícil para nosotros verlos.

Esta tecnología es usada constantemente, para comprender mejor el tema que se está estudiando. Es por esto que es importante que hayamos aprendido a construirla nosotros mismos, porque nos permite conocer un poco más de los procesos que se llevan a cabo a diario, y entender cómo se desarrollan y funcionan internamente nos lleva a una mejor comprensión de los datos que se pueda estudiar y los resultados que se pueda extraer.

De forma similar, el uso de librerías como Scikit-Learn en este caso sirve para verificar que hayamos adquirido y aplicado bien los conocimientos. Además, en un modelo de aprendizaje aplicado a la práctica, es decir, usada profesionalmente, estas librerías son fundamentales para hacer todo el proceso sin tener que preocuparse por realizar todo desde cero. Es análogo al uso de una calculadora: para poder hacer un correcto uso de sus funciones, es necesario saber qué es lo que hace cada una, para identificar si estamos teniendo algún error de cálculo.

En resumen, esta experiencia sirvió para adquirir conocimientos acerca del análisis de datos, de la teoría que hay detrás de ella, y de los procedimientos que lleva consigo, para poder aplicarlo en un futuro, ya sea en el ámbito cotidiano o profesional.