UNIVERSIDAD CATÓLICA DEL MAULE
Facultad de Ciencias de la Ingeniería
Escuela de Ingeniería Civil Informática

Profesor Guía
Dr. Sergio Hernández Álvarez

USO Y APLICACIÓN DE DIFERENCIACIÓN AUTOMÁTICA A UN ALGORITMO DE APRENDIZAJE SUPERVISADO EN PYTHON

CARLOS FELIPE ARZOLA PALMA

Tesis para optar al
Título Profesional de Ingeniero Civil Informático

TALCA, DICIEMBRE 2018

UNIVERSIDAD CATÓLICA DEL MAULE
FACULTAD DE CIENCIAS DE LA INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL INFORMÁTICA

TESIS PARA OPTAR AL
TÍTULO PROFESIONAL DE INGENIERO CIVIL INFORMÁTICO

USO Y APLICACIÓN DE DIFERENCIACIÓN AUTOMÁTICA A UN ALGORITMO DE APRENDIZAJE SUPERVISADO EN PYTHON

COMISIÓN EXAMINADORA	**FIRMA**
PROFESOR GUÍA Dr. Sergio Hernández Álvarez Universidad Católica del Maule	_____
PROFESOR COMISIÓN Dr. Paulo González Gutiérrez Universidad Católica del Maule	_____
PROFESOR COMISIÓN Mg. Rodrigo Cofré Loyola Universidad Católica del Maule	_____
NOTA FINAL EXAMEN DE TÍTULO	_____

TALCA, DICIEMBRE 2018

AGRADECIMIENTOS

RESUMEN

La era tecnológica que se vive actualmente está dando lugar a una gran cantidad de información que debe ser analizada de una forma rápida y eficiente. El machine learning puede ayudar a mitigar esta sobrecarga de información, encontrando patrones y regularidades en los datos de un problema en específico. Los cuales pueden ser relevantes para realizar o resolver una determinada tarea o proceso.

Junto con esto, existe una amplia gama de problemas computacionales tratados con machine learning, pero muchos de ellos requieren el conocimiento del cálculo de derivadas, gradientes o jacobianos de una función para ser resueltos. En casos donde se conocen las funciones de las que se necesita su diferenciación, pueden ser simplemente calculadas de forma manual. En cambio, para los casos donde el manejo matemático comienza a ser un inconveniente a partir de expresiones de mayor complejidad, existen métodos para aproximar la diferenciación que se requiera. Sin embargo, estas técnicas traen consigo ciertas desventajas que se desean evitar. Por un lado, el costo de realizar la diferenciación, a medida que las funciones aumenten su dificultad. Pero principalmente, el problema de introducir errores significativos a este cálculo.

Esta tesis explora una opción diferente, el uso de la Diferenciación Automática. La cual es una técnica que realiza el cálculo de las derivadas de una función, sin introducir ninguna complejidad adicional al código que lo usa. Además de mitigar la problemática de los errores al diferenciar. En esta investigación se abordará la teoría e implementación de este método específicamente con la herramienta Autograd. Junto con esto, se experimentará su desempeño, principalmente sobre un algoritmo de aprendizaje supervisado, versus diferentes técnicas de diferenciación y métricas propias del problema anteriormente señalado.

INDICE GENERAL

LISTA DE FIGURAS

LISTA DE TABLAS

CAPÍTULO 1. INTRODUCCIÓN

Actualmente, en la era del Big Data, se dispone de cantidades masivas de información que requiere ser procesada de manera eficiente. Se conoce como machine learning al conjunto de técnicas existentes para la detección de patrones en los datos, para luego utilizarlos en tareas como la predicción, toma de decisiones, clasificación o en la generación de nuevos datos.

Una gran parte de los algoritmos enfocados en machine learning, optimizan una función objetivo. Para ello se construye un modelo. Una representación compacta de los datos, que permitirá realizar predicciones sobre un determinado conjunto de valores. Dichas predicciones permitirán optimizar una serie de parámetros, con el objetivo de minimizar o maximizar una función [1].

El gradiente. La generalización multivariada de la derivada, son el pilar sobre el que se sostienen este tipo de optimizaciones. El cuál nos indica la dirección de máxima pendiente de la función en un punto. Utilizándose de forma iterativa, para desplazarse por el espacio de la función y encontrar mínimos y máximos relativos.

$$\nabla f(r) = \left(\frac{\partial f(r)}{\partial x_1}, \cdots, \frac{\partial f(r)}{\partial x_n} \right) \tag{1.1}$$

Dada la importancia de esta operación matemática se han desarrollado diferentes técnicas de diferenciación con el objetivo de calcular estos gradientes de la forma más eficiente posible, y acorde a las necesidades de cada problemática. Entre las técnicas que existen se pueden mencionar: la diferenciación analítica, numérica , simbólica y automática.

De forma análoga junto a estas técnicas nacen paquetes software para el cálculo de gradientes como: Tensorflow [2], Theano [3] o Autograd [4]. Estas librerías utilizan alguna de las formas de diferenciación comentadas anteriormente, para permitir a los desarrolladores optimizar sus modelos tanto en eficacia como eficiencia o acorde a sus propias necesidades.

1.1 Planteamiento del problema

En el campo del machine learning se distingue una forma de aprendizaje de modelos: el aprendizaje supervisado. El cual es utilizado cuando el objetivo es predecir el valor de una variable. Este tipo de aprendizaje requiere de un conjunto de datos de entrenamiento y otro pruebas. Cada ejemplo del conjunto de entrenamiento viene acompañado del resultado de la variable a predecir, de forma que el objetivo de este tipo de aprendizaje pasa por entrenar un modelo, que dadas las características de las muestras (features), obtenga la salida deseada de la variable a predecir (target). Después se utiliza el conjunto de prueba para verificar lo bien que generaliza el modelo entrenado con la intención de evitar que haya memorizado el conjunto de entrenamiento (overfitting), en vez de haber capturado las características esenciales del conjunto de datos [5].

Algunos algoritmos de este tipo de aprendizaje son la regresión lineal y logística [6], las redes neuronales [7] o los árboles de decisión [8]. Cabe destacar que, al momento de llevar a cabo la construcción de un modelo a partir de los algoritmos anteriormente mencionados, en su fase de entrenamiento implicará el cálculo de gradientes respecto a su función objetivo. Es necesario tener en cuenta de que, si se cambian las especificaciones del modelo, también cambiará la complejidad del cálculo del gradiente.

Es en este punto donde se debe manejar un equilibrio entre la eficacia y la eficiencia respecto a los cálculos y cómputos a realizar. Es en esta situación donde se explora el uso de la diferenciación automática como técnica de optimización para este tipo de algoritmos, partiendo por su desempeño a partir desde su tiempo de ejecución, así como el impacto de su integración en este tipo de problemáticas.

1.2 Motivación

Los métodos de optimización juegan un papel fundamental en el diseño de algoritmos en diferentes ramas científicas o ingeniería . Por lo general, los métodos de optimización iterativos requieren el cálculo repetitivo del gradiente, elemento crítico en lo que a eficiencia se refiere.

Dado que la mayoría de las funciones en problemas de optimización requieren la realización algún tipo de diferenciación (ya sea una primitiva, gradiente, jacobiano o hessiano), se vuelve muy difícil para los usuarios proporcionar este cálculo para cada función, en especial si se habla de una confección del tipo manual al momento de programar. Método que es ampliamente utilizado, pero con muchas desventajas, sobre todo, si se manejan escenarios de naturaleza no lineal de los datos utilizados [9].

Existen diferentes maneras de obtener este valor. Por un lado, la diferenciación analítica, requiere de una gran inversión de tiempo y esfuerzo. En cambio, la diferenciación numérica, aunque su implementación es simple, es método propenso a errores de truncamiento.

Una alternativa que soluciona los inconvenientes de los métodos anteriores es la diferenciación automática, este último, puede usarse para calcular las derivadas de cualquier función diferenciable, de forma automática. En este contexto "automática" significa que el usuario solo necesita escribir el código fuente de la función en un lenguaje determinado, y la herramienta puede calcular las derivadas solicitadas sin incurrir en errores de truncamiento [10]. Sin embargo, a pesar de que los inicios de este método corresponden a décadas atrás, aún no ha sido explotado en su totalidad principalmente si se habla de su eficacia. La cual está sujeta en gran parte a su enfoque de resolución o el lenguaje de programación utilizado para su construcción-

Estos cuestionamientos valen la pena ser investigados, puesto que las conclusiones que se pueden obtener sobre la diferenciación automática podrían desempeñar un papel importante y un avance significativo a la mejora de procesos que actualmente necesiten optimización y estén bajo el alcance de este método.

.

1.3 Justificación

La optimización aparece como una necesidad prácticamente en todo lo que nos rodea. Problemas típicos de diseño, modelado, identificación y control de procesos incluyen alguna etapa de optimización, bien sea estática, dinámica, fuera de línea o en tiempo real. El uso de la diferenciación automática para el cálculo del gradiente para un modelo implementado en algoritmos de aprendizaje supervisado pretende optimizar el costo de generar la diferenciación de manera manual respecto la exactitud y los tiempos de ejecución.

1.4 Objetivos

A continuación, se presentan los objetivos generales y específicos relacionados con este trabajo de tesis.

1.4.1 Objetivos generales

- Evaluar la aplicación de la diferenciación automática en un algoritmo de aprendizaje supervisado en lenguaje Python.

1.4.2 Objetivos específicos

- Implementar diferenciación automática para un modelo base.
- Analizar la exactitud de la diferenciación automática.
- Medir los tiempos de ejecución de la diferenciación automática versus un gradiente cerrado respecto al modelo base implementado.

CAPÍTULO 2. MARCO TEÓRICO

2.1 Introducción

La derivada de una función en un punto mide el coeficiente de variación o cambio de dicha función. Es una herramienta de cálculo fundamental en distintas disciplinas, tales como Electricidad, Electrónica, Termodinámica, Mecánica, Economía y Biología en las cuales resulta de vital importancia no sólo saber que determinada magnitud o cantidad varía respecto a otra, sino conocer la rapidez en que se produce dicha variación [11]. Por ejemplo, en el campo de la ingeniería, muchas leyes y otras generalidades se basan en predecibles en las cuales el cambio mismo se manifiesta en el mundo físico, como por ejemplo la segunda ley de Newton, que no está dirigida en términos de la posición de un objeto, sino en su cambio de posición con respecto al tiempo [12].

2.1.1 Tipos de diferenciación

Hoy en día existen diferentes tipos de diferenciación, las cuales se ajustan a los requerimientos de cada problema, pero con un alcance limitado, ya sea por su construcción o los resultados entregados al realizar la diferenciación.

2.1.1.1 Diferenciación Analítica

En el contexto de las Matemáticas, la derivada representa la razón de cambio de una variable dependiente con respecto a una independiente. La definición matemática de la derivada empieza con una aproximación por diferencias:

$$\frac{\Delta y}{\Delta x} = \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \tag{2.1}$$

Donde y y $f(x)$ son representaciones alternativas de la variable dependiente y x es la variable independiente. Si se permite que Δx se aproxime a cero, la diferencia es ahora una derivada:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \tag{2.2}$$

Donde $\frac{dy}{dx}$ es la primera derivada de y con respecto a x evaluada en x_i (también denominada como y' o $f'(x_i)$) [13].

2.1.1.2 Diferenciación Numérica

La derivación o diferenciación numérica consiste en evaluar derivadas de una función usando únicamente los valores que toma la función en una serie de puntos. La técnica de aproximar las derivadas por diferencias tiene muchas aplicaciones, en particular a la resolución numérica de ecuaciones diferenciales y ecuaciones en derivadas parciales.

Si se recuerda la definición de la derivada de una función $f(x)$ en un punto x:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.3}$$

Se tendrá que una primera aproximación al valor de $f'(x)$ se consigue con la expresión:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{2.4}$$

De cara a analizar el error de la aproximación, se puede suponer que $f(x)$ es diferenciable dos veces entorno al punto x. Aplicando la Formula de Taylor a $f(x+h)$ en x:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(\varepsilon)}{2}h^2 \tag{2.5}$$

Para algún $\varepsilon \in (x, x+h)$. Despejando se tendrá:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(\varepsilon)}{2}h^2 \tag{2.6}$$

De manera que la aproximación lleva asociado un error proporcional a h y a la derivada segunda de la función en un punto indeterminado. Denominando M_2 al máximo que alcance $f''(\varepsilon)$ en $(x, x+h)$. Se tendrá:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \varepsilon \leq \left|\frac{h}{2}M_2\right| \tag{2.7}$$

Una aproximación similar se obtiene desarrollando la función $f(x-h)$:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}, \varepsilon \leq \left|\frac{h}{2}M_2\right| \tag{2.8}$$

Es posible, sin embargo, mejorar la precisión de la siguiente manera: Se pueden considerar los polinomios de Taylor de las funciones $f(x+h)$ y $f(x-h)$, suponiendo que la función es al menos tres veces derivable:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(\varepsilon_1)}{6}h^3$$

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(\varepsilon_1)}{6}h^3 \qquad (2.9)$$

Si se restan ambas expresiones y despejamos tendremos:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{12}\left(f'''(\varepsilon_1) + f'''(\varepsilon_2)\right) \qquad (2.10)$$

De manera que la aproximación (a veces denominada aproximación central) tendría asociado un error proporcional a h^2:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \varepsilon \leq \left|\frac{h}{12}M_3\right| \qquad (2.11)$$

Siendo M_3 el máximo de la derivada tercera en $(x-h\ ,\ x+h)$.

Es interesante comentar que con las fórmulas anteriores pueden aparecer graves errores de redondeo, sobre todo si los datos de la función no se conocen con demasiada precisión y además h es muy pequeña, debido a las sustracciones que es necesario realizar (y los errores de redondeo que suelen llevar aparejados). [14]

2.1.1.3 Diferenciación Simbólica

La derivación simbólica consiste en tomar la función de entrada y generar una nueva función que calcule su derivada, para ello se aplica la regla de la cadena junto a las fórmulas conocidas de las derivadas básicas. En general, este método arroja fórmulas muy grandes, con muchas evaluaciones de sub-expresiones, lo que sugiere un crecimiento de la complejidad algorítmica según se calculen derivadas de mayor grado [15].

2.1.1.4 Diferenciación Automática

La diferenciación automática, permite obtener la derivada de una función definida en un programa, por compleja que sea, sin pérdida de precisión y en un tiempo de computación razonable. Se basa en el hecho de que cualquier programa de computación que implemente una función vectorial $y = F(x)$ se puede descomponer en una secuencia de asignaciones elementales, siendo cada una trivialmente diferenciable. Usando diferenciación automática un programa que calcule una función matemática es transformado en otro programa que calcule la función y su derivada, esto se logra aplicando la regla de la cadena en combinación con los principios de derivación de las operaciones matemáticas elementales, funciones trigonométricas y trascendentales, cuyas derivadas son conocidas [16].

$$l_1 = x$$
$$l_{n+1} = 4l_n(1 - l_n)$$
$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

$$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)^2 + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$$

Diferenciación de forma manual

Código

Código

Diferenciación Simbólica

```
f(x):
 v=x
 for i = 1 to 3
       v = 4*(1-v)
 return v
ó
f(x):
return 64*x(1-x)*((1-2*x)^2)
```

```
F'(x):
Return 128*x*(1-x)*(-8+16*x)*((1-2*x)^2)*
       (1-8*x+8*x*x)+64*(1-x)*((1-2*x)^2)*
       ((1-8*x+8*x*x)^2)-(64*x*(1-2*x)^2)*
       (1-8*x+8*x+8*x*x)^2-256*x*(1-x)*
       (1-2*x)* (1-8*x+8*x*x)^2
```

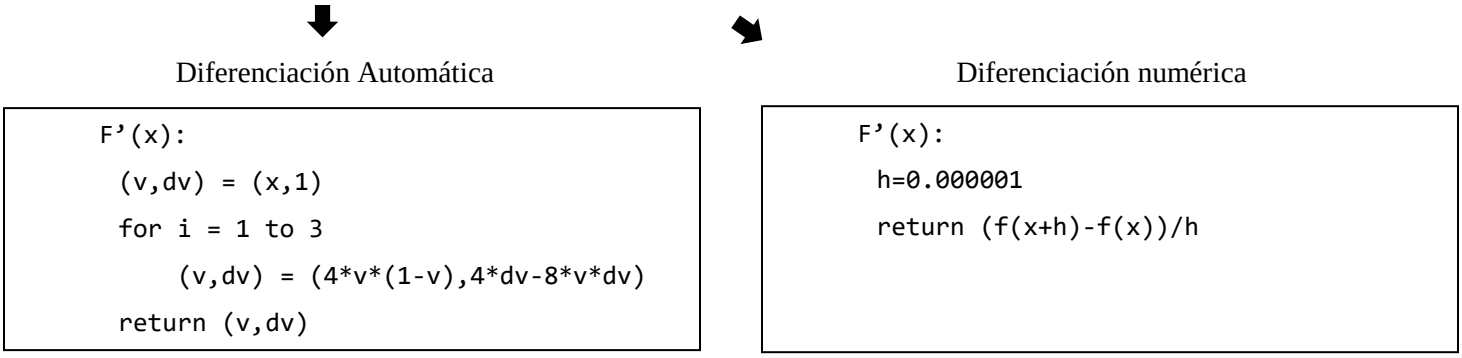

Figura 2.1: Diferentes enfoques de diferenciación

Si se aprecia la figura 2.1, se puede notar que la diferenciación simbólica (centro derecha) entrega resultados exactos, pero requiere entrada de forma cerrada y sufre un aumento de la expresión; la diferenciación numérica (abajo a la derecha) tiene problemas de precisión debido a errores de redondeo y truncamiento; la diferenciación automática (abajo a la izquierda) es tan precisa como la diferenciación simbólica. Sin embargo, esta técnica aun contiene un factor considerado incierto, su rapidez.

A pesar de que existen diferentes maneras de resolver la operación de diferenciar una expresión, será el problema a resolver y las métricas de interés, las que determinen que técnica se ajusta más a utilizar. Sin embargo, para este trabajo de tesis, predominará el uso de la diferenciación automática, debido a que no es una técnica utilizada de forma masiva, y pesar que sus primeros indicios se remontan la década de los 80. Aún se desconocen detalles de su implementación en aplicaciones de renombre, transformándose prácticamente en una caja negra para investigadores y desarrolladores. Además de su performance en problemas de mediana y gran dimensionalidad.

CAPÍTULO 3. ESTADO DEL ARTE

3.1 Introducción

Dentro de las ciencias de la computación Existe una amplia variedad de problemas matemáticos y científicos en los que es necesario obtener evaluaciones de derivadas precisas, como cuando se trata de la solución iterativa de problemas no lineales, o análisis de sensibilidad en la optimización de diseño [17]. A menudo, el cálculo de las derivadas de una función debe llevarse a cabo por código de computadora. Por ejemplo, supongamos que se tiene un código que calcula una función $f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ donde x es una variable independiente, e y es una variable dependiente, y n es el número de variables independientes, entonces generalmente es el caso en que se necesita otro código que calcule el gradiente $f' = \frac{\partial y}{\partial x}$. Por ahora es fácil calcular las derivadas de una función simple con sólo un lápiz y papel y las reglas del cálculo básico. Sin embargo, el proceso se vuelve más desafiante cuando se trata de funciones complicadas. En este caso, un enfoque alternativo es usar software para la diferenciación simbólica, como Maple o Mathematica [18] , que, sin embargo, produce fórmulas para derivadas de una función dada usando poco más que las reglas del cálculo básico. El método es preciso, pero limitado por la complejidad de la función o la expresión. Otra forma de resolver el problema de diferenciar funciones complejas y variables múltiples es aproximar las derivadas utilizando métodos numéricos, como el método de diferencias finitas . Ahora bien, este enfoque, aunque es simple de aplicar, es casi seguro que tenga errores de algún tipo en su salida, debido al hecho de que se aproxima al valor esperado, especialmente si la función es ruidosa.

Estos métodos de aproximación también son ineficientes desde el punto de vista del número de cálculos requeridos para encontrar la derivada. Por ejemplo, tal es el caso para la diferenciación unilateral que para una función de n variables necesita $\{n + 1\}$ evaluaciones de funciones para calcular $\{n\}$ derivadas.

Un enfoque alternativo al problema, que se ha desarrollado durante los últimos treinta años, es utilizar una clase de método para la diferenciación numérica precisa, conocida como

diferenciación automática (DA) [19]. En este capítulo se explicará el proceso de la DA (también conocido como diferenciación computacional, diferenciación algorítmica o diferenciación de algoritmos) y se explicará cómo funciona el método y el costo computacional del método. También se presentan las 2 formas de aplicación de esta técnica, que son la diferenciación automática hacia adelante o directa [20] y la diferenciación automática hacia atrás o inversa [21]. Para cada caso se mostrarán ejemplos para ilustrar el proceso. Cabe señalar que la AD es una poderosa herramienta para obtener no solo las primeras derivadas de manera exacta, sino también para calcular las derivadas de orden superior.

3.2 Principios de la diferenciación automática

La diferenciación automática es un conjunto de técnicas destinadas a diferenciar un código programado que calcula un valor de una función. El método difiere de otros dado que calcula derivados exactamente, y algunas veces en tiempo óptimo.

La exactitud de la DA se debe al hecho de que usa exactamente las mismas reglas de diferenciación que las del cálculo elemental y las aplica a una especificación algorítmica de una función en términos de un programa de computadora. La ejecución de cualquier programa, sin importar su complejidad, se reduce a una combinación de operadores elementales tales como operadores aritméticos o funciones elementales. Esto implica que, en cualquier función se puede dividir en operadores atómicos que involucrarán no más de dos variables independientes o dependientes a la vez. En la DA, los derivados se calculan aplicando repetidamente un conjunto de reglas de cadena a cada operador o función elemental [22].

Aunque los principios subyacentes de la AD podrían haber sido utilizados mucho antes de la invención de la computadora digital, el comienzo de la DA moderna se remonta a principios de los años sesenta. Sin embargo, a principios de los años 80, el desarrollo de nuevos conceptos de programación como la sobrecarga de operadores simplificó la tarea de implementar DA. Esto, en combinación con la invención del modo inverso DA, sentó las bases para un renacimiento del campo. El trabajo realizado por Andreas Griewank y otros en el Laboratorio Nacional de Argonne desempeñó un papel importante en muchos de los avances de este período. El interés en DA no ha disminuido desde entonces, como lo demuestra el creciente

número de aplicaciones [23]. Como se mencionó anteriormente En esencia, DA se basa en la regla de la cadena del cálculo elemental,

$$\frac{dy}{dt} = \frac{dy}{dx}\frac{dy}{dt}$$

que se utiliza para propagar derivadas en paralelo con los cálculos ordinarios. Esto es posible debido al hecho de que todos los cálculos matemáticos expresados en un lenguaje de programación se implementan utilizando un conjunto de funciones intrínsecas u atómicas (por ejemplo, $\sin, \exp,$ etc.) y operadores (+, *, etc.) con derivados bien conocidos. En el modo directo AD, las derivadas se propagan desde los parámetros de entrada a través de las diversas variables intermedias hasta los resultados finales. Una alternativa a este enfoque es operar en modo inverso, en el cual, como sugiere su nombre, la cadena de cómputo se recorre hacia atrás desde el resultado final hasta las variables de entrada. En cualquier caso, se cree que DA tiene aproximadamente la misma precisión que los cálculos del valor original no derivado [24].

3.2.1 Generar código para la diferenciación automática

Es muy importante comprender el concepto de secuencia de caracterización de una función para poder escribir software para DA. Existen dos enfoques algorítmicos diferentes para diferenciar un código de programa: "sobrecarga del operador" [25] y "transformación de código fuente" [26].

3.2.1.1 Sobrecarga del operador

En este enfoque, las operaciones aritméticas básicas y las funciones intrínsecas se asignan a rutinas que calculan las derivadas de las salidas de los operadores además del cálculo del valor de la función. El código fuente de la función se diferencia progresivamente llamando a estas rutinas al mismo tiempo que cada operación realizada en la evaluación del programa. La sobrecarga del operador solo está permitida por un número limitado de lenguajes de programación como Fortran90 [27], ADA [28], C ++ [29] o Python [30].

3.2.1.2 Transformación de código fuente

Este método define un nuevo código fuente para calcular los derivados explícitamente del código fuente del programa que calcula el valor de la función. Si el gradiente se calcula más de una vez, este método es mucho más rápido que el método de sobrecarga del operador por la simple razón de que un nuevo código fuente para generar el degradado está disponible en la primera evaluación de degradado y no cambia para las evaluaciones de pendiente restantes. Sin embargo, este tipo de método requiere una gran cantidad de esfuerzo de programación para implementarlo.

En el trabajo descrito en esta tesis, se utiliza principalmente el enfoque de sobrecarga del operador ya que es sencillo de implementar y fácil de desarrollar, sin embargo, también se considerará al menos una herramienta de transformación de código fuente. El software está escrito en el lenguaje de programación Python que permite la sobrecarga del operador y la declaración de tipo de diferenciación; dos características importantes que serán discutidas más a fondo en las siguientes en el siguiente capítulo.

3.3 Modo hacia adelante

Con el fin de dilucidar el concepto de AD, a modo de ejemplo puede considerarse la siguiente expresión :

$$z = e^{(xy)} + y\sin(x) \tag{3.1}$$

Como se mencionó anteriormente la DA permite dividir la expresión en varias operaciones atómicas, almacenadas en una serie de variables intermedias ($w_1, w_2, w_3, \ldots w_n$) de la siguiente manera:

$$\begin{aligned} w_1 &= x \\ w_2 &= y \\ w_3 &= w_1 w_2 \\ w_4 &= e^{(w_3)} \\ w_5 &= \sin(w_1) \\ w_6 &= w_2 w_5 \\ w_7 &= w_4 + w_6 \\ z &= w_7 \end{aligned} \tag{3.2}$$

Si se quisiera implementar la expresión anterior en un lenguaje de programación de alto nivel, un programador, en la mayoría de los caso, optaría simplemente por escribir directamente la expresión (3.1) en un código en lugar de la construcción elaborada de (3.2). Sin embargo, este último es más general, ya que este marco puede describir cálculos más complicados en los que participan varias subrutinas, incluidos los cálculos en los que el resultado final no se puede expresar en las variables de entrada como una fórmula de forma cerrada. Ahora bien, respecto a la administración de z con respecto a la variable t, deberíamos simplemente diferenciar cada paso utilizando la regla de la cadena y las reglas fundamentales para diferenciar funciones elementales y operadores.

$$
\begin{aligned}
\dot{w}_1 &= \dot{x} \\
\dot{w}_2 &= \dot{y} \\
\dot{w}_3 &= \dot{w}_1 w_2 + w_1 \dot{w}_2 \\
\dot{w}_4 &= \dot{w}_3 e^{(w_3)} \\
\dot{w}_5 &= \dot{w}_1 \sin(w_1) \\
\dot{w}_6 &= \dot{w}_2 w_5 + w_2 \dot{w}_5 \\
\dot{w}_7 &= \dot{w}_4 + \dot{w}_6 \\
\dot{z} &= \dot{w}_7
\end{aligned}
\tag{3.3}
$$

Para este ejemplo se ha utilizado la notación de Newton ($\dot{z} = \frac{dz}{dt}, \ddot{z} = \frac{d^2z}{dt^2}$, etc.) por simplicidad. Puesto que x e y podrían ser funciones de otras variables, y en ese caso se necesitarían más variables intermedias para describir toda la cadena de cómputos implicados en el proceso. Alternativamente, se podría pensar en x e y como si estuvieran suministrados por una rutina de caja negra que devuelve tanto los valores de x e y como sus derivados, $\dot{x}$ e $\dot{y}$. Mientras se tengan valores y derivados para alimentar en la parte inferior de la cadena, se puede recuperar el valor y el derivado del resultado final. Si x e y son las variables de entrada, se desearía recuperar el valor de z así como los valores de las derivadas parciales z_x y z_y dados ciertos valores de x e y. El acto de asignar valores a $\dot{x}$ e $\dot{y}$. para obtener los derivados correctos al final de la cadena se conoce como siembra. Por ejemplo, si quisiéramos derivar z con respecto a x, esta asumirá el rol de la variable t mencionada anteriormente, y los valores de siembra correctos serán $\dot{x} = \frac{dx}{dt} = \frac{dx}{dx} = 1$ y $\dot{y} = \frac{dy}{dt} = \frac{dy}{dx} = 0$. Si se quisiera llegar a z_y en su lugar, la siembra correcta sería $\dot{x} = 0$ y $\dot{y} = 1$. Es fácil verificar que estos dos conjuntos de valores de siembra son realmente válidos insertándolos en:

$$\dot{z} = (\dot{x}y + x\dot{y})e^{(xy)} + \dot{y}\sin(x) + y\dot{x}\cos(x),$$

Lo que resulta en las derivadas parciales correctas

$$z_x = ye^{(xy)} + y\cos(x)$$
$$z_y = xe^{(xy)} + \sin(x)$$

Finalmente, podría ser apropiado enfatizar una vez más que no se necesita la fórmula (3.1) para llegar a los derivados: si queremos calcular z_x , por ejemplo.$x = 2$ e $y = 7$, simplemente se ingresarían estos valores, así como los valores de siembra ($\dot{x} = 1$, $\dot{y} = 0$) en la ecuación. (3.2) y (3.3) y calcular todas las variables intermedias y sus derivadas hasta que llegar a $\dot{z} = \dot{w}_7$

3.4 Modo hacia atrás

Como se mencionó anteriormente puede operar de manera inversa en el modo descrito en la sección anterior. Además, también existen métodos que combinan los dos enfoques denominándose híbridos .

Para aplicar DA modo inverso a la ecuación (3.1), las operaciones atómicas en la ecuación (5.2) debe ser nuevamente diferenciado, pero esta vez en el orden opuesto:

$$z = w_7$$

es primero diferenciado con respecto a w_7, dando lugar a la adjunta

$$\overline{w_7} = \frac{\partial z}{\partial w_7} = 1$$

Los adjuntos de las variables intermedias w_4 y w_6 inmediatamente anteriores a $w_7 = w_4 + w_6$ se dan a través de la regla de la cadena como

$$\overline{w_6} = \frac{\partial z}{\partial w_6} = \frac{\partial z}{\partial w_7}\frac{\partial w_7}{\partial w_6} = \overline{w_7}.1 = \overline{w_7}$$

$$\overline{w_4} = \cdots = \overline{w_7},$$

y los restantes adjuntos se pueden mostrar para igualar

$$\overline{w_5} = \overline{w_6} w_2$$
$$\overline{w_3} = \overline{w_4} e^{(w_3)}$$
$$\overline{w_2} = \overline{w_6} w_5 + \overline{w_3} w_1$$
$$\overline{w_1} = \overline{w_5} \cos(w_1) + \overline{w_3} w_2$$

Se debe tener en cuenta que los dos últimos adjuntos, $\overline{w_1} = \frac{\partial z}{\partial w_1}\frac{\partial z}{\partial x}$ y $\overline{w_2} = \frac{\partial z}{\partial w_2}\frac{\partial z}{\partial y}$, son las dos derivadas parciales a resolver, que se han evaluado con un solo barrido inverso a través de la cadena de cálculo. Esto se puede comparar con DA a futuro, donde se habría necesitado un barrido por cada derivado parcial. Como consecuencia, la DA modo inverso es generalmente más rápido para funciones

$$f: \mathbb{R}^m \to \mathbb{R}^n$$

con $m \gg n$ (es decir, muchas variables independientes y pocas variables independientes), mientras que el modo alternativo es una desventaja para funciones del tipo $m \ll n$. Una desventaja del modo inverso es el mayor uso de memoria, ya que los adjuntos calculados deben almacenarse además de las otras variables.

3.5 Precisión y eficiencia

Tras haber explicado cada método , los cuales tienen un desarrollo matemático que distinto en su ejecución, se consigue el mismo resultado. Sin embargo, al finalizar el modo hacia atrás, se dispone además de las derivadas parciales de las variables dependientes respecto a las variables intermedias. Esto es de gran ayuda cuando se tienen varias variables independientes y

se desean hallar las derivadas parciales, ya que sólo se tiene que ejecutar el método una vez. Pero algo a considerar es el modo hacia atrás necesita mucha más memoria, ya que ha de ir almacenado todos los resultados parciales de su árbol computacional. Lo cual puede generar inconvenientes en dispositivos de bajas prestaciones.

En cuanto a la precisión, dado que la diferenciación automática está basada en simples operaciones aritméticas, su precisión básica es igual a la precisión de la máquina y del algoritmo considerado, y los métodos de DA no introducen error por sí mismos. El coste computacional del modo hacia adelante es, a priori, del mismo orden de magnitud que con diferencias finitas: evaluar una función y su derivada es por lo general entre dos y dos veces y media más costoso que evaluar la función únicamente.

$$COSTO(f, f') \leq [2, 2.5] COSTO(f)$$

En el modo hacia atrás , sin embargo, evaluar la función y su derivada tiene un coste computacional entre tres y cinco veces mayor que evaluar la función únicamente.

$$COSTO(f, f') \leq [3, 5] COSTO(f)$$

En general, las características del modo hacia adelante lo hacen preferible si en la aplicación hay muchas más variables dependientes que independientes. También hay que tener en cuenta que los requerimientos de memoria en el modo hacia adelante son aproximadamente lineales con el número de variables independientes, mientras que en el modo hacia atrás los requerimientos de memoria son mucho mayores [31].

Junto con esto, también existen casos particulares a tener en cuenta en cuanto al uso de cada modo. Si lo que causa preocupación es la cantidad de memoria de la computadora necesaria, entonces para problemas grandes, el modo hacia adelante es poco efectivo ya que toda la información sobre la función debe almacenarse. En cambio, para el método hacia atrás ocurre de forma distinta. Esto es una consecuencia del hecho de que este modo requiere estos

detalles para el barrido de forma inversa, mientras que el modo hacia adelante construye sus derivadas a medida que avanza en el programa. Sin embargo, en el tiempo, el método hacia atrás funciona mejor cuando el número de variables independientes es grande y la función es simple. El método hacia adelante es mucho más lento cuando hay una gran cantidad de variables independientes ya que se trata de una gran cantidad de aritmética vectorial.

CAPÍTULO 4. DIFERENCIACIÓN AUTOMÁTICA EN PYTHON

4.1 Introducción

Hoy en día existen diferentes tipos de aplicaciones, donde se resuelven problemas de computación numérica, en los cuales se tiene que lidiar con la tarea de cálculo de una derivada o un gradiente de forma exacta. Esto también incluye el cálculo de jacobianos y hessianos que se utilizan para resolver ecuaciones diferenciales ordinarias y parciales, así como para encontrar soluciones a diferentes problemas de optimización. Teniendo en cuenta lo anterior, una de las posibilidades para resolver este inconveniente es aplicar la técnica de diferenciación algorítmica o automática. Cabe señalar que para aplicar diferenciación automática existen múltiples herramientas en diversos lenguajes de programación [32].

En este capítulo se proporcionará información sobre las diferentes herramientas o alternativas de diferenciación automáticas en Python, un lenguaje de programación ideal para trabajar con grandes volúmenes de datos ya que favorece su extracción y procesamiento, siendo el elegido por grandes empresas para el trabajo de Big Data. A nivel científico, posee una amplia biblioteca de recursos con especial énfasis en las matemáticas para aspirantes a programadores en áreas especializadas como el Deep y el machine learning [33]. Además de contar con una gran comunidad de usuarios activos que comparten constantemente sus conocimientos y recursos en todo tipo de medios.

4.2 Diferenciación automática versus diferenciación numérica y simbólica

En palabras simples los diferenciadores automáticos, son calculadoras de gradiente que computan y proporcionan una función basada en un programa de cómputo para evaluar la (s) derivada (s) de una función. Sin embargo, existen otros tipos de calculadoras de gradiente, incluidos los diferenciadores numéricos y los diferenciadores simbólicos. Los primeros se construyen utilizando la definición de límite fundamental de la derivada (no las reglas derivadas en sí mismas). Si bien son fáciles de implementar, tienen problemas de estabilidad que los hacen menos útiles como una calculadora de gradiente universal. Los diferenciadores simbólicos, que calculan una fórmula algebraica para derivadas en lugar de una función de programa de cómputo, son mucho menos útiles para nuestras aplicaciones, ya que principalmente necesitamos funciones de gradiente programáticas (para uso en esquemas de optimización locales). Además, las funciones derivadas algebraicamente pueden tardar un tiempo mayor y requieren grandes cantidades de memoria para ser representadas

Con el fin de considerar todos los métodos de diferenciación en una comparativa general, se utilizó la función `derivative` de la biblioteca scipy.misc[34], para calcular derivadas por el método de diferencias finitas, y SymPy[35] como herramienta de cálculo simbólico de derivadas. Esto se abordará con más detalle en el capítulo 6.

4.3 Herramientas de diferenciación automática en Python

En esta sección, proporcionará información sobre las principales herramientas disponibles a la fecha para este lenguaje, haciendo una gran hincapié en Autograd, una herramienta que dada su performance respecto a su velocidad y exactitud será utilizada en la confección de un algoritmo de aprendizaje supervisado en el próximo capitulo.

Lo primero que se debe mencionar al momento de usar y trabajar con herramientas de diferenciación automática en Python es la diferencia en la sintaxis y la inicialización de los datos para el procedimiento, así como la especificación propia de la herramienta a utilizar. Estos detalles son fundamentales para trabajar de manera óptima con la opción que se escoja. Todas las herramientas que se mencionan a continuación pueden usar diferenciación automática hacia

adelante o hacia atrás, aplicando la técnica de sobrecarga del operador para implementar la diferenciación automática, excepto CasADI, que se basa en la transformación del código fuente.

4.4 Autograd

Autograd puede diferenciar automáticamente los códigos nativos de Python y Numpy. Puede manejar un gran subconjunto de funciones de Python, incluidos bucles, ifs, recursión y cierres, e incluso puede tomar derivados de derivados de derivados. Admite la diferenciación en modo inverso (backpropagation), lo que significa que puede tomar de manera eficiente gradientes de funciones de valor escalar con respecto a los argumentos de valor de matriz, así como la diferenciación en modo directo, y los dos pueden componerse de manera arbitraria. La principal aplicación prevista de Autograd es la optimización basada en gradientes [36].

4.4.1 Cómo usar Autograd

La función `grad` de Autograd toma una función y entrega la función que calcula su derivada. Su función debe tener una salida de valor escalar (es decir, flotante). Esto cubre el caso común cuando quieres usar el cálculo del gradiente para optimizar algún problema en específico.

Autograd trabaja en el código nativo de Python y Numpy que contiene todas las estructuras de control habituales, incluidos los ciclos condicionales `while` e `if` además de las declaraciones y cierres. El siguiente ejemplo muestra una manera simple de usar Autograd :

```
from autograd import elementwise_grad as egrad
import matplotlib.pyplot as plt

x = np.linspace(-7, 7, 200)
plt.plot(x, tanh(x),
...          x, egrad(tanh)(x),
...          x, egrad(egrad(tanh))(x),
...          x, egrad(egrad(egrad(tanh)))(x),
...          x, egrad(egrad(egrad(egrad(tanh))))(x),
```

```
...           x, egrad(egrad(egrad(egrad(egrad(tanh)))))(x),
...           x, egrad(egrad(egrad(egrad(egrad(egrad(tanh))))))(x))
plt.show()
```

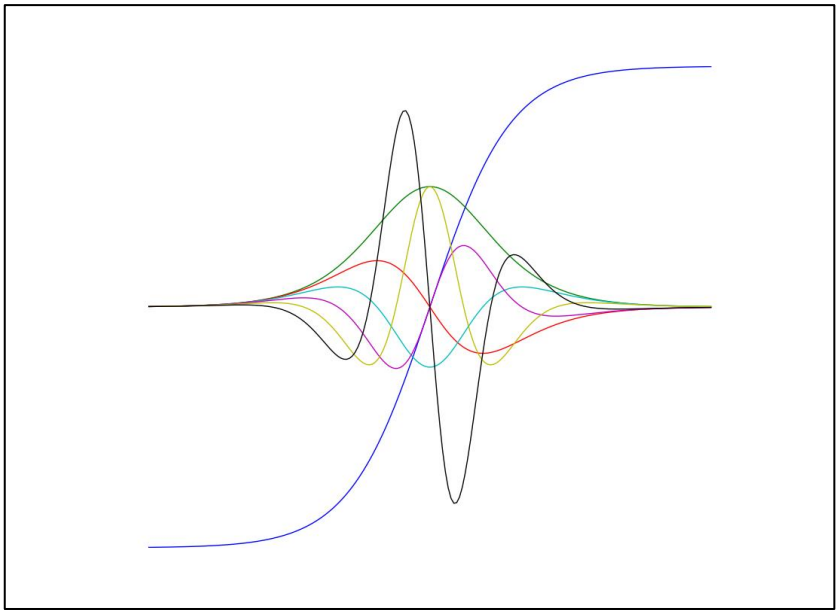

Figura 4.1: Gradientes de la función tangente utilizando Autograd

4.4.2 ¿Qué sucede en un nivel más profundo?

Para calcular el gradiente, Autograd primero tiene que registrar cada transformación que se aplicó a la entrada, ya que se convirtió en la salida de su función. Para hacer esto, Autograd ajusta las funciones (usando la función `primitive`) para que cuando sea llamada, se agreguen a una lista de operaciones realizadas. El núcleo de Autograd tiene una tabla que mapea estas primitivas envueltas a sus correspondientes funciones de gradiente (o, más precisamente, sus funciones de producto vector-Jacobiano). Para marcar las variables con las que se está calculando el gradiente, se envuelven usando la clase `Box`.

Después de evaluar la función, Autograd tiene un grafo que especifica todas las operaciones que se realizaron en las entradas con respecto a las cuales queremos diferenciar. Este es el grafo de cálculo que realiza la evaluación de la función. Para calcular la derivada, simplemente aplicamos las reglas de diferenciación a cada nodo en el grafo [37].

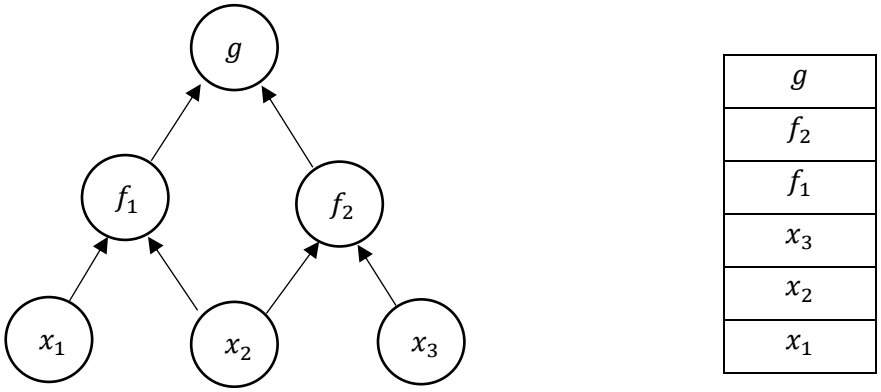

Figura 4.2: Representación gráfica del enfoque de la diferenciación automática

4.4.3 Diferenciación de modo inverso en Autograd

Dada una función formada por varias llamadas a función anidadas, hay varias formas de calcular su derivada.

Por ejemplo, dado $L(x) = F\left(G\left(H(x)\right)\right)$, la regla de la cadena dice que su gradiente es $\frac{\partial L}{\partial x} = \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x}$. Si evaluamos este producto de derecha a izquierda:$\left(\frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x}\right)$, se realizó el mismo orden que los cálculos mismos, esto se denomina diferenciación de modo directo. Si evaluamos este producto de izquierda a derecha: $\left(\frac{\partial F}{\partial G} * \frac{\partial G}{\partial H}\right) * \frac{\partial H}{\partial x}$, esto se denomina diferenciación en modo inverso.

En comparación con las diferencias finitas o el modo hacia adelante, la diferenciación en modo inverso es, con mucho, el método más práctico para diferenciar funciones que toman un vector grande y dan como resultado un solo número. En la comunidad de aprendizaje automático, la diferenciación de modo inverso se conoce como 'backpropagatión', ya que los gradientes se propagan hacia atrás a través de la función. Este método es particularmente amigable ya que no necesita crear instancias de las matrices jacobianas intermedias de manera explícitamente, y en su lugar solo se enfoca en aplicar una secuencia de funciones de producto jacobianas vectoriales sin matriz (VJP).

Debido a que Autograd también es compatible con derivados más elevados, los productos vector-Hessiano (una forma de segunda derivada) también están disponibles y son eficientes de calcular mediante este método.

Algunos paquetes de diferenciación automática (como TensorFlow) funcionan de al especificar un grafo de cálculo que realiza su función, incluido todo el flujo de control (como `if` y ciclos `for`), y luego convertir ese grafo en otro que calcula los gradientes. Esto tiene algunos beneficios (como permitir optimizaciones en tiempo de compilación y paralelismo), pero requiere que se exprese el flujo de control en un mini lenguaje limitado que solo esos paquetes saben cómo manejar. (Por ejemplo, las operaciones `tf.while` y `tf.cond` de TensorFlow).

Por el contrario, Autograd no tiene que saber sobre if's, bifurcaciones, ciclos o recursiones que se utilizaron para decidir qué operaciones se invocaron. Para calcular el gradiente de una entrada particular, solo se necesita saber qué transformaciones se aplicaron de manea continua

a esa entrada en particular, y no qué otras transformaciones se pudieron haber aplicado. Dado que Autograd realiza un seguimiento de las operaciones relevantes en cada llamada de función por separado, no es un problema que todas las operaciones de flujo de control de Python sean invisibles para Autograd. De hecho, simplifica enormemente la implementación [37].

4.4.4 ¿Qué puede diferenciar Autograd?

La principal limitación es que cualquier función que opera en una `Box` está marcada como `primitive`, y tiene su gradiente implementado. Esto se soluciona para la mayoría de las funciones en la biblioteca Numpy, y es fácil escribir sus propios gradientes. La entrada puede ser un número escalar, complejo, vector, tupla, una tupla de vectores, una tupla de tuplas, etc. Al usar la función `grad`, la salida debe ser escalar, pero las funciones `elementwise_grad` y `jacobian` dan la posibilidad de permitir gradientes de vectores.

Operadores	$+, -, *, /, (-), **, \%, <, \leq, ==, !=, \geq, >$
Funciones matematicas básicas	`exp,log,square,sqrt,sin,cos,tan,sinh,cosh, tanh,sinc,abs,fabs,logaddexp,logaddexp2, absolute,reciprocal,exp2,expm1,log2,log10, log1p,arcsin,arcos,arctan,arcsinh,arccosh, arctanh,rad2deg,degrees,deg2rad,radians`
Numeros complejos	`real,imag,conj,angle,fft,fftshift,ifftshift, real_if_close`
Reducciones de array	`sum,mean,prod,var,std,max,min,amax,amin`
Reorganización de array	`Reshape,ravel,squeeze,diag,roll,array_split, Split,vsplit,hsplit,dsplit,expand_dims,fliplr, rot90,swapaxes,rollaxis,transpose,atleast_2d, atleast_3d`
Álgebra lineal	`dot,tensordot,einsum,cross,trace,outer,det, slogdet,inv,norm,eigh,cholesky,sqrtm,`

	`solve_traingular`
Otras operaciones de array	`Cumsum,clip,maximum,minimun,sort,msort,partition, Concatenate,diagonal,truncate_pad,tile,full,triu, Tril,where,diff,nan_to_num,vstack,hstack`
Funciones de probabilidad	`t.pdf,t.cdf,tlogpdf,tlogcdf, multivariate_normal.logpdf multivariate_normal.pdf multivariate_normal.entropy, norm.pdf,norm.cdf, norm.logpdf,norm.logcdf,dirichlet.logpdf, dirichlet.pdf`
Funciones especiales	`polygamma,psi,digamma,gamma,gammaln, rgamma,multigammaln,j0,y0,j1,y1,jn,yn,erf,erfc`

Tabla 4.1: Primitivas con gradientes implementados en Autograd.

4.4.5 Trabajo futuro

Muchas han sido las peticiones por parte de la comunidad para agregar integrar cómputo de GPU [38] hacia Autograd. De hecho, si se está apuntando a cálculos en el régimen limitado de BLAS[39], sería una buena opción el aprovechar las GPU para operaciones de álgebra lineal altamente paralelas. Una de las alternativas es que la construcción de una biblioteca de álgebra lineal para GPU, o incluso la creación de enlaces de Python a una biblioteca existente. Sin embargo, esto está por el momento fuera del alcance de Autograd dada su construcción. Por lo que se espera que Numpy (u otro proyecto de Python) construya una biblioteca así de estas características, y que pueda integrarse con Autograd. Dado esto, existe una versión de Autograd escrita por Lua/Torch, un equipo de Twitter (directamente inspirado por Python Autograd) que tiene compatibilidad nativa con GPU [40] y es ampliamente utilizada por desarrolladores y expertos dedicados al machine learning.

4.5 AlgoPy

El propósito de AlgoPy es la evaluación de las derivadas de orden superior en el modo directo e inverso de la diferenciación automática de las funciones que se implementan como programas Python. El enfoque particular son expresiones que contienen funciones de álgebra lineal numérica. El uso previsto de AlgoPy es para crear prototipos fácilmente a velocidades de ejecución razonables [41].

4.6 CasADI

CasADI es una herramienta para la diferenciación automática y la optimización numérica. Esta contiene una amplia funcionalidad que se centra en el control óptimo. Mientras que los otros paquetes utilizan la técnica de sobrecarga del operador para implementar la diferenciación automática, CasADI, en su forma actual, utiliza la transformación de código fuente. Así mismo, CasADI explota el mismo enfoque para proporcionar su funcionalidad, así como la eficiencia de la implementación de C ++ a Python, pero utiliza SWIG en lugar de Boost. Esta herramienta usa una sintaxis especial para marcar variables activas y crear objetos de función. Los últimos pueden ser utilizados para fines de diferenciación automática [42].

4.7 Numdifftools

Numdifftools es un conjunto de herramientas escritas en Python para resolver problemas de diferenciación numérica automática en una o más variables. permitiendo al usuario especificar si se usan diferencias de pasos complejos, centrales, hacia adelante o hacia atrás. Pudiendo calcular los derivados de orden 1 a 10 en cualquier función escalar y un vector gradiente de una función escalar de una o más variables [43].

4.8 AD

El paquete AD permite realizar de forma fácil y transparente la diferenciación automática de primer y segundo orden. Las matemáticas avanzadas que incluyen funciones trigonométricas, logarítmicas, hiperbólicas, etc. también pueden evaluarse directamente utilizando el submódulo admath.

Se admiten todos los tipos numéricos básicos ($int, float, complex$, etc.). Este paquete está diseñado para que los tipos numéricos subyacentes interactúen entre sí como lo hacen normalmente al realizar cualquier cálculo. AD explota el hecho de que cada programa de computadora, sin importar cuán complicado sea, ejecuta una secuencia de operaciones aritméticas elementales (suma, resta, multiplicación, división, etc.) y funciones elementales ($\exp, \log, \sin, \cos$, etc.). Al aplicar la regla de la cadena repetidamente a estas operaciones, los derivados de orden arbitrario se pueden calcular automáticamente y de forma totalmente precisa [44].

Herramienta	**Lenguaje**	**Modo**	**Codificación**
Autograd	Python	D-I	SO
AlgoPy	Python	D-I	SO
CasADI	Python, C++, Matlab	D-I	TF
Numdifftools	Python	D-I	SO
AD	Python , C++	D-I	SO

Tabla 4.2: Comparativa de herramientas de diferenciación automática

D: Directo

I: Inverso

SO: Sobrecarga de operador

TF: Transformación de código fuente

El propósito de esta comparativa es llamar la atención sobre las diferentes alternativas que existen diferenciación automática en Python, para brindar información sobre las principales características de estas, y resaltar las ventajas de utilizar cada una de ellas, dado que dependerá de las necesidades y problemáticas a resolver. En el siguiente capítulo se proporcionará información sobre el problema que pretende resolver con el uso de Autograd, mientras especulamos sobre sus características. Posterior a esto las herramientas descritas en este capítulo serán sometidas a pruebas de performance tanto de velocidad como exactitud en diferentes tipos de problemas.

CAPÍTULO 5. IMPLEMENTACIÓN DE DIFFERENCIACIÓN AUTOMÁTICA EN UN MODELO DE APRENDIZAJE SUPERVISADO

5.1 Introducción

Para muchos problemas de machine learning se pretende probar un nuevo modelo de aprendizaje automático para un determinado set de datos de estudio. Esto generalmente significa tener alguna función de pérdida asociada que permita capturar qué tan bien este modelo, se ajusta a los datos y buscar optimizar esa pérdida con respecto a los parámetros del modelo. Si se consideran una cantidad significativa de parámetros para el modelo seleccionado (por ejemplo, las redes neuronales pueden tener millones), entonces se necesitará el cálculo de gradientes como técnica de optimización. En esta situación se dispone de dos opciones: derivarlos y codificarlos de manera manual, o implementar su modelo usando las restricciones sintácticas y semánticas de herramientas como Theano o Tensor Flow.

Pero en este escenario se pretende proporcionar una tercera forma: simplemente escribir la función de pérdida utilizando una biblioteca numérica estándar como Numpy, y usar Autograd para el cálculo efectivo del gradiente. Para llevar a cabo la implementación de esta forma, se usará un de aprendizaje supervisado ampliamente utilizado en el área de machine learning . La regresión logística.

5.2 Regresión logística

La regresión logística es un algoritmo de clasificación utilizado para realizar observaciones a un conjunto discreto de clases. A diferencia de la regresión lineal que genera valores numéricos continuos, la regresión logística transforma su resultado utilizando la función sigmoide o función logística para entregar un valor de probabilidad que luego se puede asignar a dos o más clases discretas.

La regresión logística hace uso de lo que se conoce como un clasificador binario. Donde utiliza la función sigmoide para predecir una probabilidad de que la respuesta a alguna pregunta

sea: 1 o 0, sí o no, verdadero o falso, bueno o malo, etc. Es esta función la que impulsará el algoritmo y también es interesante en que se puede usar como una " función de activación " para redes neuronales [45].

5.2.1 Clasificación con una regresión logística

La Regresión logística es un algoritmo que es relativamente simple y poderoso para decidir en su forma más atómica entre dos clases, es decir, es un clasificador binario. Básicamente entrega una función que es un límite entre dos clases diferentes. Se puede ampliar este concepto para manejar más de dos clases mediante un método denominado "uno contra todos" (regresión logística multinomial) que es realmente una colección de clasificadores binarios que simplemente selecciona la clase más probable al observar cada opción individualmente aislándola sobre todo lo demás para luego elegir la clase con la probabilidad más alta.

Probablemente la idea más simple de definir este modelo es un tipo simple de clasificador de sí o no. La regresión logística puede hacer uso de un gran número de características, incluidas variables continuas y discretas, además características no lineales. Se puede usar para muchos tipos de problemas como el filtrado de spam, selección de imágenes cancerígenas, escaneo de piezas en una línea defectuosa, etc.

5.3 Implementación de una regresión logística en Python utilizando Autograd

La implementación de una regresión logística en Python tiene como fin llevar a cabo un experimento el cual contiene una serie de pruebas respecto al comportamiento de la herramienta Autograd, con el fin de medir su desempeño. Se escogió este algoritmo y este lenguaje dado que es ampliamente utilizado en distintas áreas de investigación. Donde la optimización de resultados, así como la reducción del costo de realizar determinadas tareas, siempre es un camino para seguir. En esta sección se mostrarán los principales pasos para la construcción de

una regresión logística simple la cual servirá para realizar las pruebas del próximo capítulo, así su respectiva integración con Autograd.

5.3.1 Carga y normalización de un set de datos con scikit-learn

Para entrenar un algoritmo de aprendizaje supervisado, se necesitan datos de entrenamiento, es decir, datos previamente seleccionados en base a un problema o situación de análisis. Una característica importante que deben cumplir los datos seleccionados es que deben estar normalizados y estandarizados, esto con el fin de lograr un mejor aprendizaje y realizar posteriormente una predicción más certera. Para garantizar este objetivo, se utilizarán datos de entrenamiento de la API scikit-learn [46], la cual posee una gran variedad datasets para algoritmos de aprendizaje supervisado además de funciones de normalización.

5.3.1.1 sklearn datasets

El paquete sklearn.datasets entrega datos de entrenamiento para problemas asociados al machine learning. Con el fin, de evaluar el impacto de la escala del conjunto de datos (n muestras y n características) mientras se controlan las propiedades estadísticas de los datos (por lo general, la correlación e información de las características), donde también es posible generar datos de manera simulada para problemas de una mayor dimensionalidad [47].

Las funciones de generación de datasets de este paquete comparten una interfaz simplista, que devuelve una tupla (X, y) . Donde X es una matriz que consta de n muestras por n características e y , que es un arreglo de longitud n muestras que contienen los objetivos y.

La mayoría de los conjuntos de datos con los que trabaja este paquete son obtenidos desde mldata.org. Un sitio que almacena una extensa base de datos de aprendizaje para problemas asociados al machine learning [48]

Para la implementación de la regresión logística, se utilizará la siguiente carga y asignación de datos:

```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()

X = data['data']
y = data['target']
```

5.3.1.2 Normalización de características de entrada

5.3.1.2.1 Preprocesado de datos

El paquete sklearn.preprocessing proporciona varias funciones comunes de utilidad y clases de transformación para cambiar los vectores de características en "bruto" a una representación que sea más adecuada al problema a resolver [47]. En general, los algoritmos de aprendizaje se benefician de la estandarización del conjunto de datos. Dado que se puede mitigar la presencia de valores atípicos.

5.3.1.2.2 Funciones de escalamiento

Una Función de escalamiento permite "acomodar" las características para que se encuentren entre un valor mínimo y máximo dado. a menudo entre cero y uno, o para que el valor absoluto máximo de cada característica se escale al tamaño de la unidad. Esto se puede lograr usando `MinMaxScaler` o `MaxAbsScaler`, respectivamente.

La motivación para usar este escalado incluye la solidez a desviaciones estándar muy pequeñas de características y la preservación de cero entradas con datos dispersos. Volviendo a la regresión logística, el escalamiento se realizaría de la siguiente manera:

```
From sklearn.preprocessing import MinMaxScaler X =\
MinMaxScaler(feature_range=(-1,1)).fit_transform(X)
```

También con este paquete se puede particionar un dataset según se requiera. Formando los conjuntos de datos de entrenamiento y de prueba. Esto se realiza con el fin de lograr una predicción más certera y acorde a los resultados esperados por el algoritmo. Para el caso de la regresión logística, el conjunto de datos destinados al entrenamiento es de una mayor magnitud (75% del total de los datos) respecto a los asignados para prueba (25% del total de los datos). Con esto se pretende que la clasificación sea más precisa, y que los resultados sean convergentes a las cifras esperadas.

```
From sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
```

5.3.2 Modelo de regresión logística

5.3.2.1 Representación de un modelo

Para establecer una notación, en esta sección se utilizará $x^{(i)}$ para denotar las variables de "entrada", también llamadas características de entrada, y $y^{(i)}$para denotar la "salida" o variable objetivo que se intenta predecir. Un par $(x^{(i)}, y^{(i)})$ se denomina ejemplo de entrenamiento; y el conjunto de datos que utilizará para el aprendizaje: una lista de m ejemplos de entrenamiento $(x^{(i)}, y^{(i)})$; $i = 1 \ldots m$, se llama conjunto de entrenamiento. También se usará X para indicar el espacio de los valores de entrada, y Y para indicar el espacio de los valores de salida. Donde, $X = Y = \mathbb{R}$

Para describir el problema de aprendizaje supervisado de manera formal, el objetivo es, dado un conjunto de entrada, se realice un entrenamiento a una función $h: X \to Y$ para que $h(x)$

sea un "buen" predictor para el valor correspondiente de y. Por razones históricas, esta función h se llama hipótesis. Visto de manera gráfica, el proceso es así:

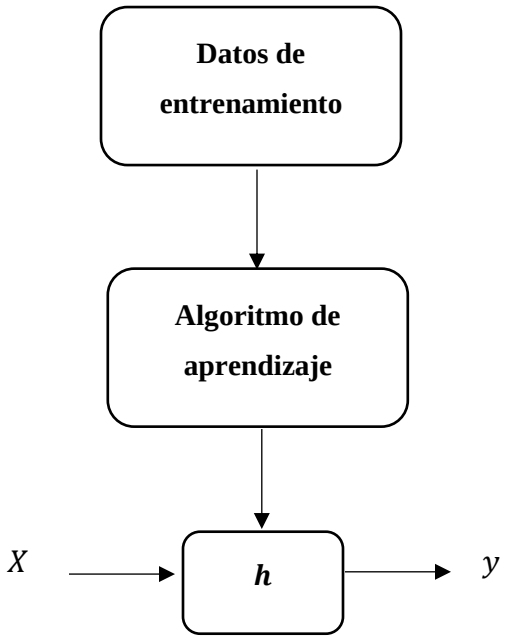

Figura 5.1: Esquema del proceso de aprendizaje supervisado

Ahora bien, tras haber seleccionado el conjunto de datos de estudio, además de normalizarlo, escalarlo y distribuirlo. Lo primero que debe entender al aplicar un modelo de regresión logística. es que se entiende por "aprendizaje supervisado". En este escenario los datos de entrenamiento serán etiquetados como datos y efectivamente tienen solo 2 valores (en el caso de la regresión logística binaria), es decir:

$$y \in \{0,1\}$$
$$y = \begin{cases} 0\ El\ caso\ es\ negativo \\ 1\ El\ caso\ es\ positivo \end{cases} \tag{5.1}$$

5.3.2.2 Límite de decisión

Para obtener una clasificación 0 o 1 discreta, podemos traducir la salida de la función de hipótesis de la siguiente manera:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$h_\theta(x) < 0.5 \rightarrow y = 0 \tag{5.2}$$

La forma en que se comporta la función logística g es que cuando su entrada es mayor o igual a cero, su salida es mayor o igual a 0.5:

$$g(z) \geq 0.5$$

Donde:

$$z \geq 0 \tag{5.3}$$

Cabe recordar que:

$$z = 0, e^0 = 1 \rightarrow g(z) = \frac{1}{2}$$
$$z \rightarrow \infty, e^{-\infty} \rightarrow 0 \rightarrow g(z) = 1$$
$$z \rightarrow -\infty, e^{\infty} \rightarrow \infty \rightarrow g(z) = 0 \tag{5.4}$$

Entonces, si la entrada a g es $\theta^T X$ por lo que significa:

$$h_\theta(x) = g(\theta^T x) \geq 0.5$$

Entonces:

$$\theta^T x \geq 0 \tag{5.5}$$

De estas afirmaciones ahora se puede decir:

$$\theta^T x \geq 0 \rightarrow y = 1$$

$$\theta^T x < 0 \rightarrow y = 0 \tag{5.6}$$

El límite de decisión es la línea que separa el área donde y = 0 y donde y = 1. Es creada por nuestra función de hipótesis.

Donde 0 y 1 son las etiquetas que están asignadas a los "objetos o preguntas" que se están observando. Por ejemplo, si estamos mirando correo no deseado, entonces un mensaje que es spam se etiqueta como 1. Se requiere un modelo que produzca valores entre 0 y 1 e interprete el valor del modelo como una probabilidad de que el caso de prueba sea positivo o negativo.

$$0 \leq h_\theta(x) \leq 1$$
$$h_\theta(x) = P(y = 1 \,|x; \theta) \tag{5.7}$$

La expresión anterior se lee como La probabilidad de que $y = 1$ dados los valores en el vector de características x parametrizado por a. Además, dado que h se interpreta como una probabilidad, la probabilidad de que $y = 0$ está dada por $P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$ ya que las probabilidades tienen que sumar 1.

La función h del modelo (hipótesis) es similar a:

$$h_\theta(x) = g(\theta^T x)$$
$$z = \theta^T x$$
$$h_\theta(x) = g(z) = \frac{1}{1+e^{-z}} \tag{5.8}$$

Cuando se vectoriza el modelo para generar algoritmos se usará X, la matriz aumentada de variables de características con una columna de unidades, al igual que en la regresión lineal. Se debe tener en cuenta que cuando se visualiza un solo vector de entrada x, el primer elemento

de x se establece en 1, es decir, $x_0 = 1$. Esto multiplica el término constante $\theta_0 h_\theta(x)$ y $h_\theta(X)$ en el caso en que tenemos n características similares a:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n)$$

$$h_\theta(x) = g\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_n^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & x_n^{(m)} \end{bmatrix}\begin{bmatrix} \theta_0 \\ \vdots \\ \theta_n \end{bmatrix} \tag{5.9}$$

m es la cantidad de elementos en el conjunto de prueba.

5.3.2.3 Representación de hipótesis

Para este caso se puede abordar el problema de clasificación ignorando el hecho de que y tiene un valor discreto, y usar nuestro antiguo algoritmo de regresión lineal para tratar de predecir y dado x. Sin embargo, es fácil construir ejemplos en los que este método funciona muy mal. Intuitivamente, tampoco tiene sentido que $h_\theta(x)$ tome valores mayores que 1 o menores que 0 cuando sabemos que $y \in \{0,1\}$. Para solucionar esto, se debe cambiar la forma de las hipótesis $h_\theta(x)$ para satisfacer $0 \leq h_\theta(x) \leq 1$. Esto se logra conectando $\theta^T x$ en la función logística [50].

Nuestra nueva forma utiliza la "función sigmoidea", también llamada "función logística":

5.3.2.4 Creación de la función sigmoide

De acuerdo con lo definido anteriormente implementación de la función sigmoide sería de la siguiente forma:

```
def sigmoid(z):

return 1.0/(1.0 + np.exp(-z))
```

Para garantizar que la forma de la función sigmoide es la correcta se graficará a una escala:$-5 \leq x \leq 5$, $0 \leq y \leq 1$

```
import matplotlib.pyplot as plt

x = np.arange(-5, 5, 0.1)
plt.plot(x, sigmoid(x))
plot.show()
```

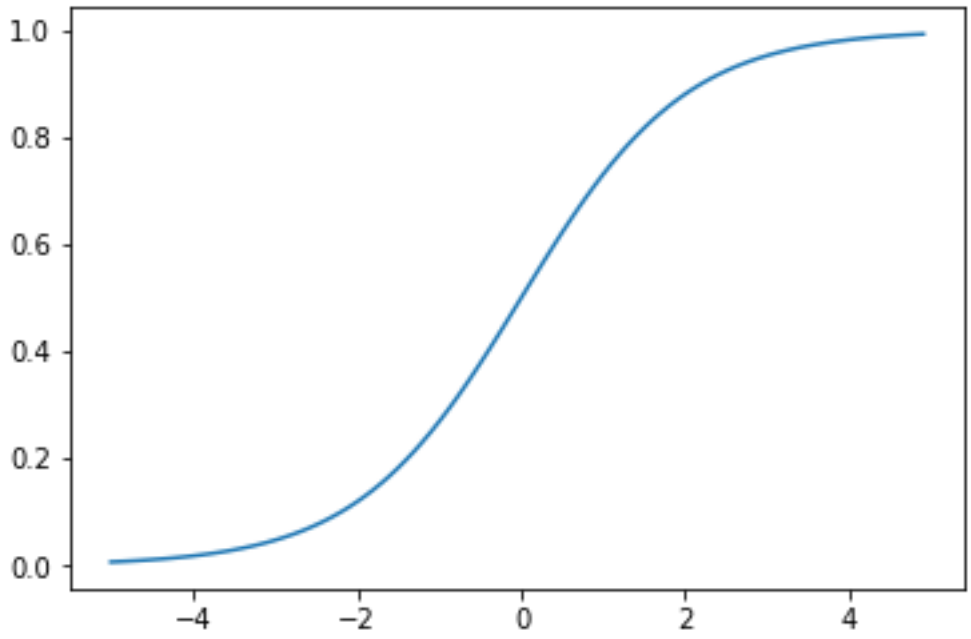

Figura 5.2: Función Sigmoide

5.3.2.5 Creación del modelo de regresión logística

Para esta implementación de regresión logística se tiene el siguiente modelo:

```
def logisticRegression(weights, inp):
```

```
                    prediction = np.dot(inp,weights)
    return sigmoid(prediction)
```

5.3.3 Función de costo una regresión logística

El objetivo principal de una función de pérdida en una regresión logística es penalizar las malas elecciones para optimizar los parámetros y recompensar los aciertos probabilísticos. Dicho de otra forma, lo que se pretende es ajustar la función para que coincida estrechamente con los datos de entrenamiento. Si las etiquetas son cero y uno, $y \in \{0,1\}$, podríamos minimizar la pérdida al cuadrado:

$$\sum_{i=1}^{N} \left(y^{(i)} - h_\theta\left(x^{(i)}\theta\right)\right)^2 \tag{5.10}$$

Sin embargo, el modelo de regresión logística utiliza la interpretación de la función como una probabilidad, $h_\theta(x) = P(y = 1|x, \theta)$, más directamente. La adaptación de máxima verosimilitud de este modelo minimiza la probabilidad logarítmica negativa de los objetivos, que podría escribirse como:

Como antes, cualquiera de las funciones de costo puede tener un regularizador agregado para desalentar las ponderaciones extremas. La estimación de máxima verosimilitud tiene algunas buenas propiedades estadísticas. En particular, asintóticamente (para muchos datos) es el estimador más eficiente. Aunque la pérdida puede ser extrema cuando se hacen predicciones erróneas confiadas, lo que podría significar que los valores atípicos causan más problemas que el enfoque de pérdida cuadrada [51].

Para este caso no es posible usar la misma función de costo que se usa para una regresión lineal porque la función logística hará que la salida sea ondulada, causando muchos óptimos locales. En otras palabras, no será una función convexa.

En cambio, esta función de costo para la regresión logística se ve así:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right) \tag{5.11}$$

Donde:

$$\begin{aligned} Cost(h_\theta(x), y) &= -\log\left(h_\theta(x)\right) \quad \text{si } y = 1 \\ Cost(h_\theta(x), y) &= -\log\left(1 - h_\theta(x)\right) \quad \text{si } y = 0 \end{aligned} \tag{5.12}$$

Es posible comprimir los dos casos condicionales de la función de costo en un caso:

$$Cost(h_\theta(x), y) = -y\log\left(h_\theta(x)\right) - (1-y)\log\left(1 - h_\theta(x)\right) \tag{5.13}$$

Se puede observar que cuando y es igual a 1, entonces el segundo término $(1-y)\log\left(1-h_\theta(x)\right)$ será cero y no afectar el resultado. Si y es igual a 0, entonces el primer término $-\log\left(h_\theta(x)\right)$ será cero y no afectará el resultado[52].

De forma completa la función de costo de puede escribirse de la siguiente manera:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] \tag{5.14}$$

Una implementación vectorizada es:

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m}\cdot\left(-y^T\log(h) - (1-y)^T\log(1-h)\right) \tag{5.15}$$

5.3.4 Implementación de función de pérdida

Tras la teoría expuesta anteriormente, la interpretación de la función de perdida para esta la regresión logística, con diferenciación automática sería la siguiente:

```
def cross_entropy(weights):
      pred = logisticRegression(weights, X_train)
return - np.mean(y_train*np.log(pred) + (1.0 - y_train)*np.log(1.0 - pred))
```

5.3.5 Función de gradiente

La pieza final requerida es el vector de gradiente. Un optimizador basado en gradiente puede encontrar los pesos que minimizan el costo.

La derivada de la función sigmoide es la siguiente:

$$\frac{\partial}{\partial\theta}(J(\theta) = (h_\theta(x) - y)x \tag{5.16}$$

La cuál tiende a cero en las asíntotas $x \to \pm\infty$

Si bien a simple vista se puede inferir que este proceso en la práctica no es trivial. Puesto que requiere de un conocimiento mínimo para poder entenderlo y poder replicarlo de forma correcta. Sin embargo, Autograd disminuye esta brecha, optimizando la obtención del gradiente a partir de la función de perdida construida para este problema.

5.3.5.1 Algoritmo del descenso del gradiente

Como técnica de optimización se utilizará algoritmo llamado descenso de gradiente, dado que es rápido, efectivo y fácil de entender. El descenso de gradiente requiere que se tome el gradiente de la función de pérdida y se comience a iterar respecto a su formulación. En casos simples se puede diferenciar fácilmente con lápiz y papel o si se conoce el gradiente de la

función, codificarlo a medida. Sin embargo, Con funciones de pérdida más complejas, a menudo no se puede obtener de manera trivial. Para este caso Autograd explota la regla de la cadena de cálculo para diferenciar función de costo[53]

Cabe recordar que la forma general de descenso del gradiente es:

$$Repetir\{$$

$$\theta_j \coloneqq \theta_j - \alpha \frac{\partial}{\partial \theta_j}\big(J(\theta)\big) \tag{5.17}$$

$$\}$$

Donde α se denomina learning rate [54]

De manera general la expresión quedaría de la siguiente forma:

$$Repetir\{$$

$$\theta_j \coloneqq \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}\big(h_\theta\big(x^{(i)}\big) - y^{(i)}\big)\, x_j^{(i)} \tag{5.18}$$

$$\}$$

Una implementación vectorizada sería:

$$\theta \coloneqq \theta - \frac{\alpha}{m} X^T (g(X\theta) - \bar{y}) \tag{5.19}$$

5.3.5.2 Importación de librería de diferenciación automática

Para comenzar la integración de la diferenciación automática con el algoritmo de la regresión logística. Se importa la librería Autograd. Para instalarla si se utiliza anaconda basta con ejecutar el comando `sudo conda install Autograd` [55]

```
import autograd.numpy as np
from autograd import grad
```

Con Autograd, obtener el gradiente de la función de pérdida personalizada bastante sencillo, como definir `gradiente = grad(Función de costo)`[56]. Por lo que el enfoque principal debe estar en la función de costo, la cual debe estar definida correctamente. Ahora bien, para esta implementación sería:

```
grad_function = grad(cross_entropy)
```

Siguiendo la lógica de las ecuaciones 5.17 y 5.18, los detalles de implementación del descenso del gradiente con esta técnica serían los siguientes:

```
for i in range(iterations):
    loss[i] = cross_entropy(weights)
    acc[i] = accuracy(y_train, logisticregression(weights, X_train))
    weights = weights - alpha*grad_fcn(weights)
```

5.3.1 Función de precisión

Esta función ayuda a determinar la exactitud de las predicciones de la regresión logística implementada.

```
def accuracy(y_true, y_pred):
return np.mean(y_true == np.round(y_pred))
```

5.3.2 Parametrización inicial

Se inicializan los parámetros funcionales de la regresión logística binaria, como los son el número de iteraciones, la tasa de aprendizaje, los pesos y los vectores que almacenan tanto la precisión como la perdida.

```
iterations = 1000
alpha = 0.1
loss = np.zeros(iterations)
acc =  np.zeros(iterations)
weights = np.random.randn(X_train.shape[1])*0.1
```

5.3.3 Resultados

Para esta implementación a pesar de que el conjunto de datos no era de una alta dimensionalidad, se puede apreciar de manera significativa el comportamiento del algoritmo del descenso del gradiente respecto a las funciones de perdida y precisión

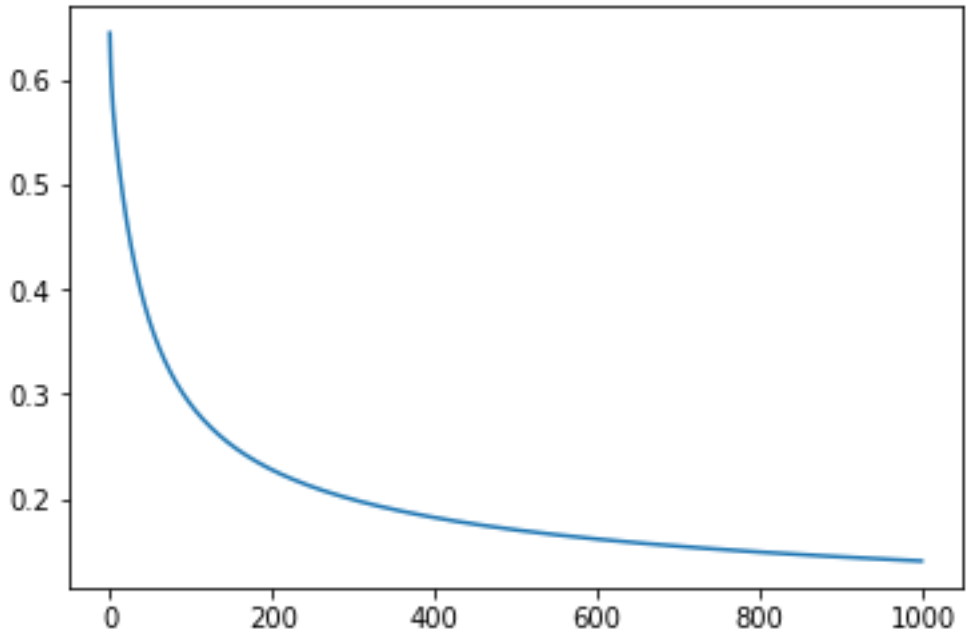

Figura 5.3: Función de perdida transcurridas 1000 iteraciones

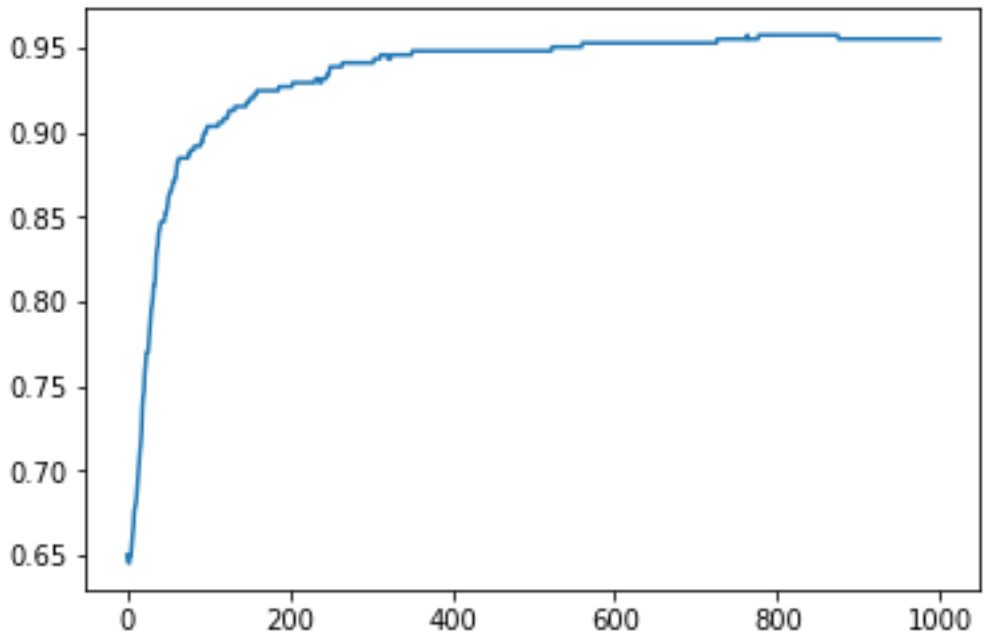

Figura 5.4: Función de precisión transcurridas 1000 iteraciones.

Tras la implementación de la regresión logística utilizando Autograd y su correcta ejecución. El siguiente objetivo es formular un experimento de pruebas, considerando se pueda la velocidad como métrica principal de los resultados obtenidos versus la dimensionalidad de los datos seleccionados. La razón principal de tomar este camino surge por la escasa información que existe de la velocidad de las herramientas de diferenciación automática en este tipo de problemas. Esto se abordará con total detalle en el siguiente capítulo de pruebas y resultados.

CAPÍTULO 6. PRUEBAS Y RESULTADOS

6.1 Introducción

En este capítulo se mostrarán una serie de pruebas realizadas que muestran el desempeño de la diferenciación automática en líneas generales, así como en situaciones específicas y de alto desempeño. Este capítulo contará de cuatro secciones. La primera sección tratará las pruebas realizadas para la resolución de derivadas comunes, con el fin de medir la exactitud y velocidad de las distintas técnicas de diferenciación que se mostraron en el capítulo 2. La segunda sección repetirá el experimento anterior, solo que en este caso serán sometidas a pruebas las herramientas de diferenciación automática mostradas en el capítulo 4. Una tercera sección medirá las herramientas de DA más efectivas de la segunda sección, con funciones más complejas respecto a su gradiente. Finalmente, la cuarta sección abordará las pruebas de la implementación de la regresión logística que se ha desarrollado en el capítulo anterior. Para ello se utilizarán set de datos tanto de baja como alta dimensionalidad, con el fin de medir el tiempo de ejecución del cálculo del gradiente a diferentes escalas, y de dos perspectivas distintas. La obtención del gradiente usando diferenciación automática con la herramienta Autograd versus un gradiente cerrado o confeccionado de forma manual.

Para el desarrollo de estas pruebas se darán a conocer las distintas métricas a utilizar y la definición de los datasets escogidos para estas pruebas, los cuales han sido seleccionados desde sklearn.datasets con el fin de mantener un estándar y uniformidad de acuerdo las pruebas van avanzado. Las pruebas se desarrollarán tanto en una CPU convencional, como en un servidor de mejores prestaciones respecto a esta CPU, esto con el fin de poder experimentar el cambio de velocidad respecto al hardware.

6.1.1 Características de CPU

Características CPU	
SO	Ubuntu 16.04 LTS
Procesador	AMD A08-5550M APU 2.10 GHZ
Memoria RAM	8 GB

Tabla 6.1: Características de la CPU utilizada en las pruebas

6.1.2 Características Servidor

Características Servidor TUI	
SO	CentOS 7.2
Procesador	Intel Xeon
Memoria RAM	24 GB

Tabla 6.2: Características del servidor utilizado en las pruebas

6.2 Pruebas de exactitud y tiempo de técnicas de diferenciación

La primera parte de este experimento medirá el desempeño tanto desde la exactitud como del tiempo de ejecución de las diferentes técnicas de diferenciación expuestas en el capítulo 2. Esto con el fin de visualizar el comportamiento de cada técnica respecto a una derivada en particular. El set de derivadas sometido a esta prueba cuenta con expresiones de complejidad aleatoria, donde se muestra la función junto con diferenciación respectiva.

N	$f(x)$	$f'(x)$
1	$\frac{(5x^2+7x+2)^2}{(x^2+6)}$	$\frac{2(5x^2+7x+2)(5x^3+58x+42)}{(x^2+6)^2}$
2	$e^{\frac{\sin(x)}{\cos(x)}}$	$e^{\sin(x)}1+\tan(x)\frac{1}{\cos(x)}$
3	$\frac{(7+2x)^3}{(x^3+4x^2+1)}$	$\frac{(7+2x)^2(-13x^2-56x+6)}{(x^3+4x^2+1)^2}$
4	$\ln\left(\frac{1}{(x^3-4x+1)^2}\right)$	$-\frac{2(3x^2-4)}{(x^3-4x+1)}$
5	$\frac{\sin(e^x)}{x}$	$\frac{e^{(x)}x\cos(e^{(x)})\sin(e^{(x)})}{x^2}$
6	$(x^3+4x^2)(5x^4+4x^2)^3$	$x^4(5+4x)^2(36x^2+158x+100)$
7	$(4x^3+3x^2)e^{(x^2+7)}$	$2xe^{(x^2+7)}(4x^3+3x^2+6x+3)$
8	$\tan\left(4x^4-2x^2+\frac{7}{2}x^{-3}+5\right)^{-2}$	$\frac{-2(16x^3-4x-\left(\frac{21}{2}\right)x^{-4}}{\left(4x^4-2x^2+\left(\frac{7}{2}\right)x^{-3}+5\right)^3\cos(4x^4-2x^2+(\frac{7}{2})x^{-3}+5)^3)^{-2}}$
9	$\frac{(x^2+3x+6)^4}{(x+1)}$	$\frac{(x^2+3x+6)^3(7x^2+17x+6)}{(x+1)^2}$
10	$\frac{e^{\sin(x)}}{e^{\cos(x)}}$	$e^{\sin(x)}1+\tan(x)\frac{1}{\cos(x)}$
11	$(4x^6+5x+3)e^{(x^2+5x+1)}$	$(-8x^7+20x^6+24x^5-10x^2+19x+20)e^{(-x^2+5x+1)}$
12	$\frac{\cos(e^{(x)})}{x}$	$-\left(\frac{xe^{(x)}\,sin\left(e^{(x)}\right)+cos\left(e^{(x)}\right)}{x^2}\right)$

13	$x^2 e^{x^5}$	$e^{(x^5)} x(5x^5 + 2)$
14	$(5x^4 - 3x^2 + 2x)e^{(-3x^2+x-2)}$	$(-30x^5 + 5x^4 + 38x^3 - 15x^2 - 4x + 2)e^{(-3x^2+x-2)}$
15	$(6x^2 + x)^2(x^5 + x^6)^4$	$2(6x^2 + x)(x^5 + x^6)^3(84x^7 + 85x^6 + 11x^5)$

Tabla 6.3: Set de funciones y sus derivadas a medir

Tras realizar la prueba, se puede visualizar que los resultados obtenidos son prácticamente idénticos para todas las técnicas, solo con excepción de la solución simbólica que su fin es entregar una expresión algebraica. También hay que destacar que la solución obtenida por las diferencias finitas contiene un error respecto a la solución analítica dado su tamaño de paso h sin embargo esto para casos particulares de diferenciación no es totalmente notable, la única forma que esto pueda ser realmente evidente será el caso en que se ejecute este método en un algoritmo de características iterativas donde el error se acumule hasta entregar un determinado resultado.

N°	Solución analítica	Solución DF	solución Autograd	solución Simbólica
1	142,56	142,56	142,56	Simbólica
2	15,51771538	15,51771538	15,51771538	Simbólica
3	-30,5888	-30,58879999	-30,5888	Simbólica
4	-16	-16	-16	Simbólica
5	1,433000973	1,433000973	1,433000973	Simbólica
6	1514240	1514240	1514240	Simbólica
7	14130297,44	14130297,44	14130297,44	Simbólica
8	-0,001033803	-0,001033228	-0,001033228	Simbólica
9	30947,55556	30947,55556	30947,55556	Simbólica
10	15,51771538	15,51771538	15,51771538	Simbólica
11	1142691,751	1142691,751	1142691,751	Simbólica
12	-3,414461264	-3,414461264	-3,414461264	Simbólica
13	2,5584E+16	2,5584E+16	2,5584E+16	Simbólica
14	-0,003944584	-0,003944584	-0,003944584	Simbólica
15	7,61128E+11	7,61128E+11	7,61128E+11	Simbólica

Tabla 6.4: Soluciones numéricas de técnicas de diferenciación

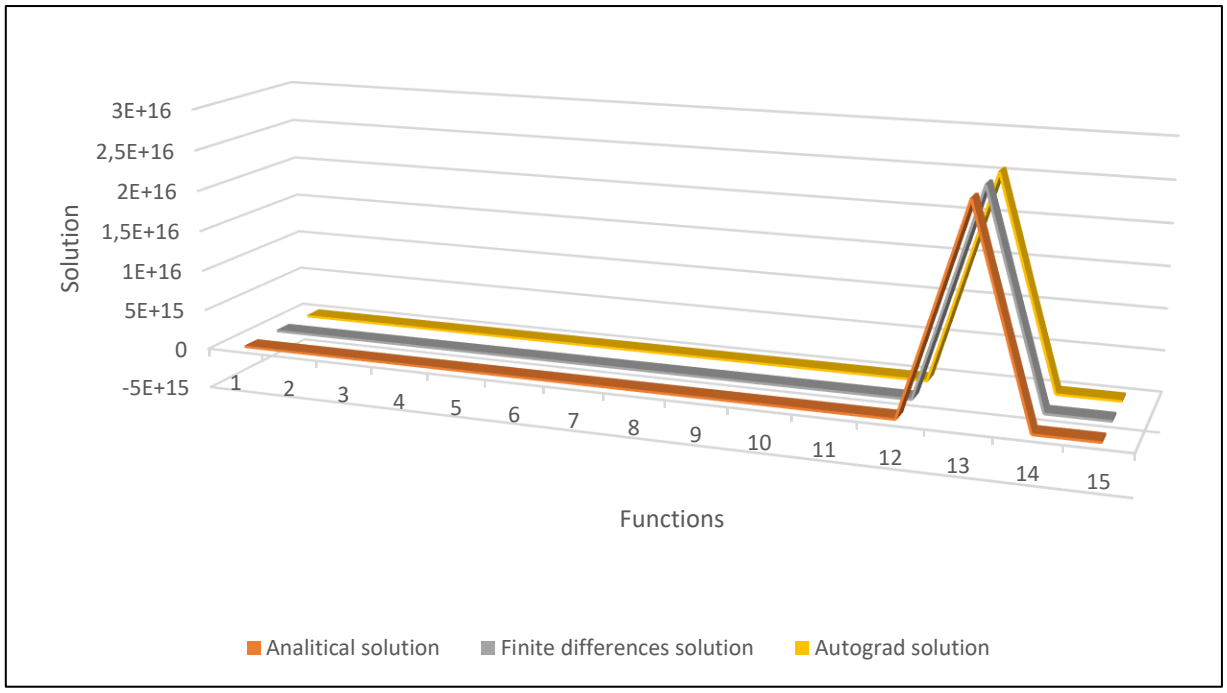

Figura 6.1: Gráfico solución numérica de técnicas de diferenciación

Ahora bien, su visualizamos el tiempo de ejecución de cada una de las técnicas, se pueden ver ya diferencias notorias sobre todo si se considera como solución una expresión simbólica, lo cual es totalmente evidente, dado que el resultado que debe entregar comprende características y cómputos diferentes a las otras alternativas. Es importante mencionar que en este caso solo se realizó la diferenciación simbólica y no se procedió a evaluar la expresión con el fin de no aumentar de manera considerable su tiempo de ejecución. Respecto a las demás técnicas, todas se mantienen de forma constante en un determinado margen, sin embargo, a pesar de que este margen sea mínimo, las diferencias pueden visualizarse en la figura 6.2. Por un lado, el método de diferencias finitas consigue un tiempo menor al de Autograd respecto a la solución analítica, pero no se debe olvidar que la primera arrastra un error que, si bien es mínimo, está ahí presente. en cambio, Autograd ofrece un resultado análogo al analítico, pero

con un mayor tiempo de ejecución, y de menor holgura respecto al caso de la diferenciación simbólica.

N°	Tiempo analítica	Tiempo DF	Tiempo Autograd	Tiempo Simbólica
1	5,00679E-06	4,1962E-05	0,000551939	0,023996115
2	4,1008E-05	4,8161E-05	0,000192881	0,004359961
3	4,05312E-06	3,2902E-05	0,000435114	0,01503706
4	3,09944E-06	3,6001E-05	0,000432968	0,00434494
5	2,69413E-05	3,8862E-05	0,000144005	0,002562046
6	2,14577E-06	2,6941E-05	0,000370026	0,011934042
7	2,40803E-05	3,8862E-05	0,000396967	0,004311085
8	1,5974E-05	3,7193E-05	0,000392914	0,01482892
9	4,05312E-06	3,0041E-05	0,000349998	0,00919795
10	3,40939E-05	4,7922E-05	0,000205994	0,000922203
11	1,19209E-05	3,6001E-05	0,000412941	0,004893064
12	2,31266E-05	3,9816E-05	0,00014782	0,001822233
13	1,09673E-05	3,4094E-05	0,00019598	0,001712799
14	1,19209E-05	3,7193E-05	0,000446796	0,005488873
15	4,05312E-06	3,0994E-05	0,000516891	0,012034178

Tabla 6.5: Tiempo de técnicas de diferenciación para obtener su solución numérica

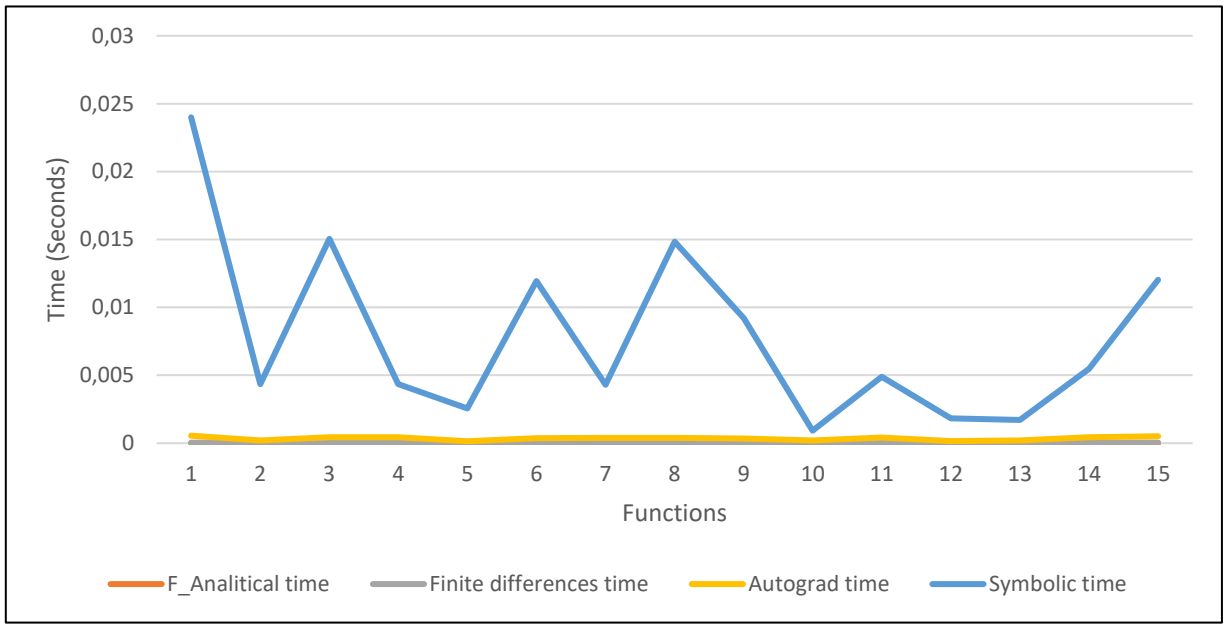

Figura 6.2: Gráfico tiempo de técnicas de diferenciación y solución numérica

6.3 Pruebas de exactitud y tiempo de herramientas de diferenciación automática

En esta sección , se considerará la precisión respecto a la rapidez, sometiendo a pruebas de desempeño a las herramientas de diferenciación automática mostradas en el capítulo 4. Para esta prueba se consideró el mismo set de derivadas utilizado en la primera sección a excepción de la ecuación N° 8 donde la función tangente, generó inconvenientes en AlgoPy. Ahora bien, respecto a la prueba de exactitud no encontramos diferencias de solución numérica comparado con la función analítica.

N	Solución AlgoPy	Solución Autograd	Solución casADI	Solución Numdifftools	Solución AD
1	142,56	142,56	142,56	142,56	142,56
2	15,51771538	15,51771538	15,5177	15,51771538	15,517715
3	-30,5888	-30,5888	-30,5888	-30,5888	-30,5888
4	-16	-16	-16	-16	-16
5	1,433000973	1,433000973	1,433	1,43300097	1,433001
6	1514240	1514240	1,51E+06	1514240	1514240

7	14130297,44	14130297,44	1,41E+12	14130297,44	14130297
8	30947,55556	30947,55556	30947,6	30947,55556	30947,556
9	15,51771538	15,51771538	15,51771538	15,51771538	15,517715
10	1142691,751	1142691,751	1142691,751	1142691,751	1142691,8
11	-3,414461264	-3,414461264	-3,41446	-3,414461264	-3,4144613
12	2,5584E+16	2,5584E+16	2,56E+16	2,56E+16	2,558E+16
13	-0,003944584	-0,003944584	-0,00394458	-0.00394458	-0,0039446
14	7,61128E+11	7,61128E+11	7,61E+11	7,61E+11	7,611E+11

Tabla 6.6: Soluciones numéricas de técnicas de diferenciación automática

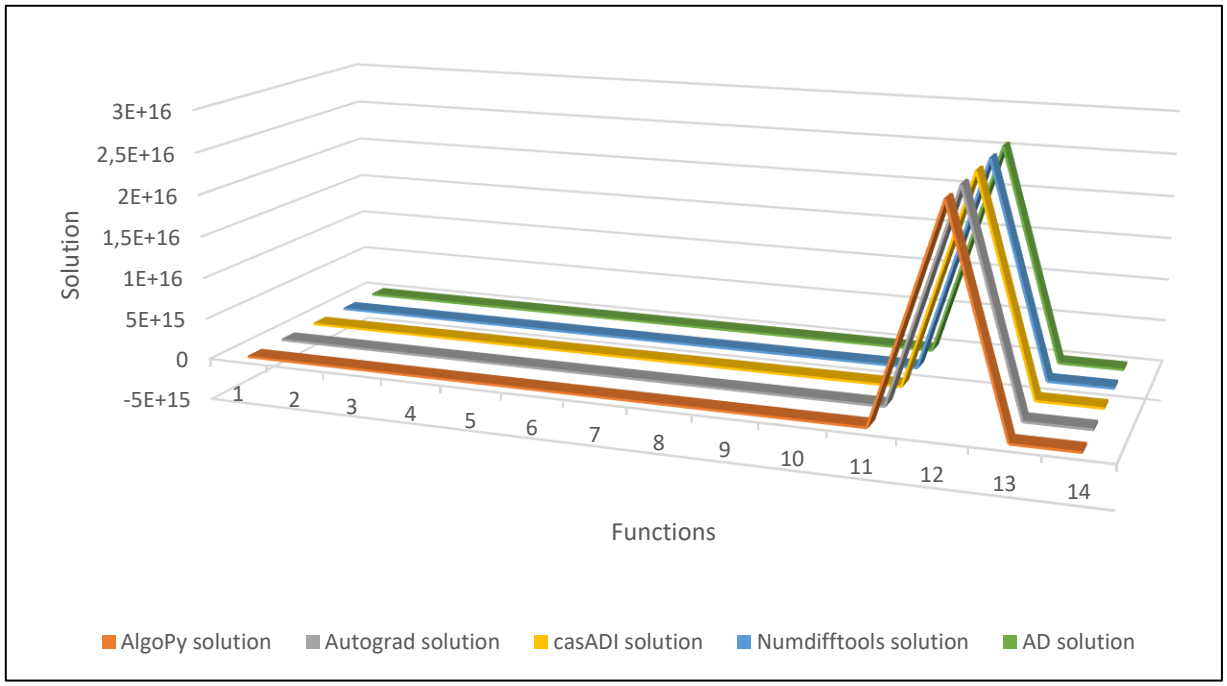

Figura 6.3: Gráfico solución numérica de técnicas de diferenciación automática

Ahora, si se considera el tiempo de ejecución a pesar de que el tiempo es mínimo, se puede ver que Autograd y AD se destacan respecto a las demás herramientas, con el menor tiempo de

ejecución forma constante. Un detalle no menor a considerar es que CasADI a pesar de tener un enfoque de construcción más complejo que las otras herramientas, logra conseguir un tiempo inferior al de AlgoPy y Numdifftools. Volviendo al análisis de Autograd y AD, como visualmente las diferencias de estas últimas no es gráficamente notable, someteremos a estas herramientas por ser las más rápidas , a una prueba más de características más complejas.

N	Tiempo AlgoPy	Tiempo Autograd	Tiempo casADI	Tiempo Numdifftools	Tiempo AD
1	0,00142694	0,000549078	0,000584126	0,02302599	0,000257969
2	0,00085783	0,000226021	0,00036788	0,00228405	8,2016E-05
3	0,00133514	0,00040102	0,000527143	0,002501011	0,000150919
4	0,00108314	0,000324011	0,000446081	0,00247097	0,000166178
5	0,00059795	0,000182867	0,000270128	0,002232075	0,000100851
6	0,00144506	0,000485897	0,000536919	0,002486944	0,000159025
7	0,00125003	0,00039196	0,000510931	0,002434015	0,000142813
8	0,00109816	0,000346899	0,000432014	0,002433062	0,00011301
9	0,00061703	0,000167131	0,000266075	0,001986027	6,50883E-05
10	0,00140691	0,000408173	0,000511885	0,002151012	0,000185013
11	0,00051379	0,000154972	0,000231981	0,00189805	4,81606E-05
12	0,00061989	0,000205994	0,000258923	0,002002001	6,79493E-05
13	0,00149393	0,000449181	0,000554085	0,002176046	0,000209093
14	0,00122404	0,000421047	0,000428915	0,002110958	0,000143051

Tabla 6.7: Tiempo de técnicas de DA para obtener su solución numérica

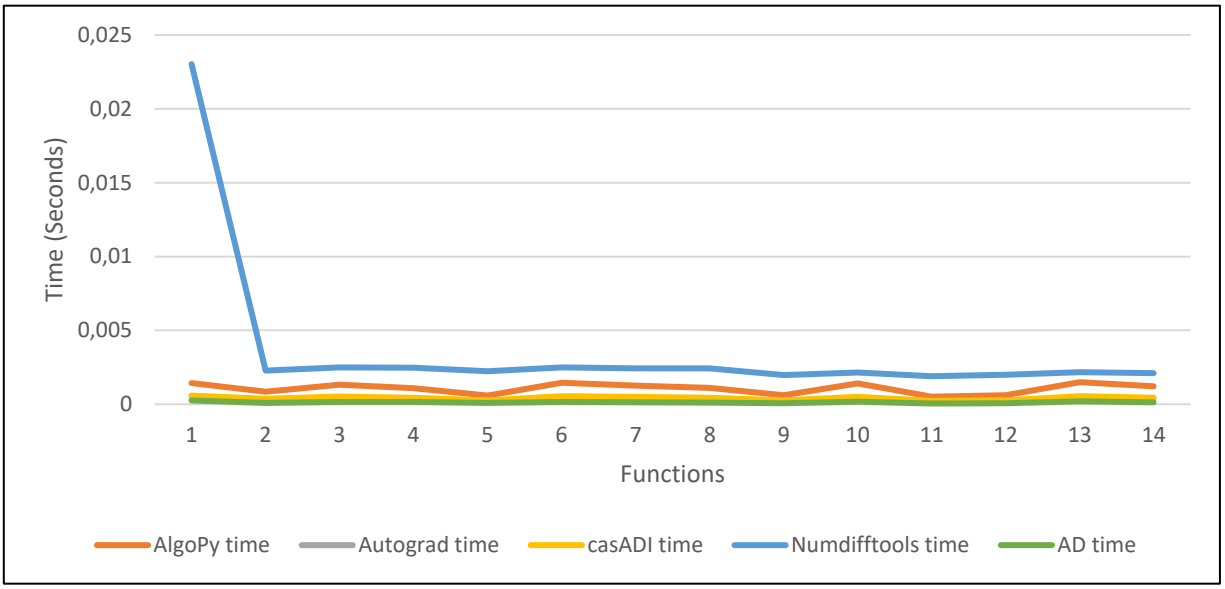

Figura 6.4: Gráfico tiempo de técnicas de DA en obtener su solución numérica

6.4 Pruebas de exactitud y tiempo entre Autograd y AD

Como se mencionó en el punto anterior se medirá el desempeño de las herramientas Autograd y AD, en un escenario más complejo. En este caso lo que se busca es obtener un gradiente de funciones en un grado más avanzado de diferenciar, lo que determinará un tiempo de ejecución mayor. El set de funciones a probar corresponde en su mayoría a problemáticas de optimización o química.

Nombre	Función
Ackley [56]	$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{(0.5(\cos 2\pi x + y))} + e + 20$
Cross in Tray [57]	$f(x,y) = -0.0001\left[\left\lvert \sin x \sin y\, e^{\left(\left\lvert 100 - \frac{\sqrt{x^2+y^2}}{\pi}\right\rvert\right)}\right\rvert + 1\right]^{0.1}$
Shaffer N2 [58]	$f(x,y) = 0.5\frac{\sin^2(x^2-y^2) - 0.5}{(1+0.001(x^2+y^2))^2}$
Shaffer N4 [58]	$f(x,y) = 0.5\frac{\cos^2(\sin(\lvert x^2-y^2\rvert)) - 0.5}{(1+0.001(x^2+y^2))^2}$
Gibbs-Duhem [59]	$0 = x_1\frac{d\ln\gamma_1}{dx_1} + (1-x_1)\frac{d\ln\gamma_2}{dx_1}$ $\frac{G^{ex}}{RT} = nx_1(1-x_1)(A_{21}x_1 + A_{12}(1-x_1))$ $Donde\ n = n_1 + n_2\ , y\ x_1 = \frac{n_1}{n}\ y\ x_2 = \frac{n_2}{n}$ $\ln\gamma_1 = \frac{\frac{\partial G_{ex}}{RT}}{\partial n_1}\ \ln\gamma_2 = \frac{\frac{\partial G_{ex}}{RT}}{\partial n_2}$ $0 = x_1\frac{d\ln\gamma_1}{dn_1} + x_2\frac{d\ln\gamma_2}{dn_1}$

Murnaghan [60]	$P = -\frac{dE}{dV}\ ,\quad E = E_0 + \frac{B_0 V}{B'_0}\left[\frac{\left(\frac{V_0}{V}\right)^{B'_0}}{B'_0 - 1} + 1\right] - \frac{V_0 B_0}{B'_0 - 1}$

Tabla 6.8: Funciones a diferenciar con Autograd y AD

Como ya es tendencia los resultados obtenidos de forma numérica, son prácticamente análogos en ambas herramientas, ya tanto en su derivada parcial $\frac{dF}{dx}$ como en $\frac{dF}{dy}$

Función	**Solución $\frac{dF}{dx}$ Autograd**	**Solución $\frac{dF}{dy}$ Autograd**	**Solución $\frac{dF}{dx}$ AD**	**Solución $\frac{dF}{dy}$ AD**
Ackley	1,637461506	-1,637461506	1,637461506	-1,637461506
Cross-in-tray	-0,084830628	0,084830628	-0,084830628	0,084830628
Gibbs-Duhem	0,760325926	0,244959259	0,760325926	0,244959259
Murnaghan	4,44694E-15	0,808167685	0	0,808167685
ShafferN2	-1,251698845	12,54589449	-1,251698845	12,54589449
shafferN4	1,233666799	-12,30776195	1,233666799	-12,30776195

Tabla 6.9: Solución numérica obtenida diferenciando con Autograd y AD

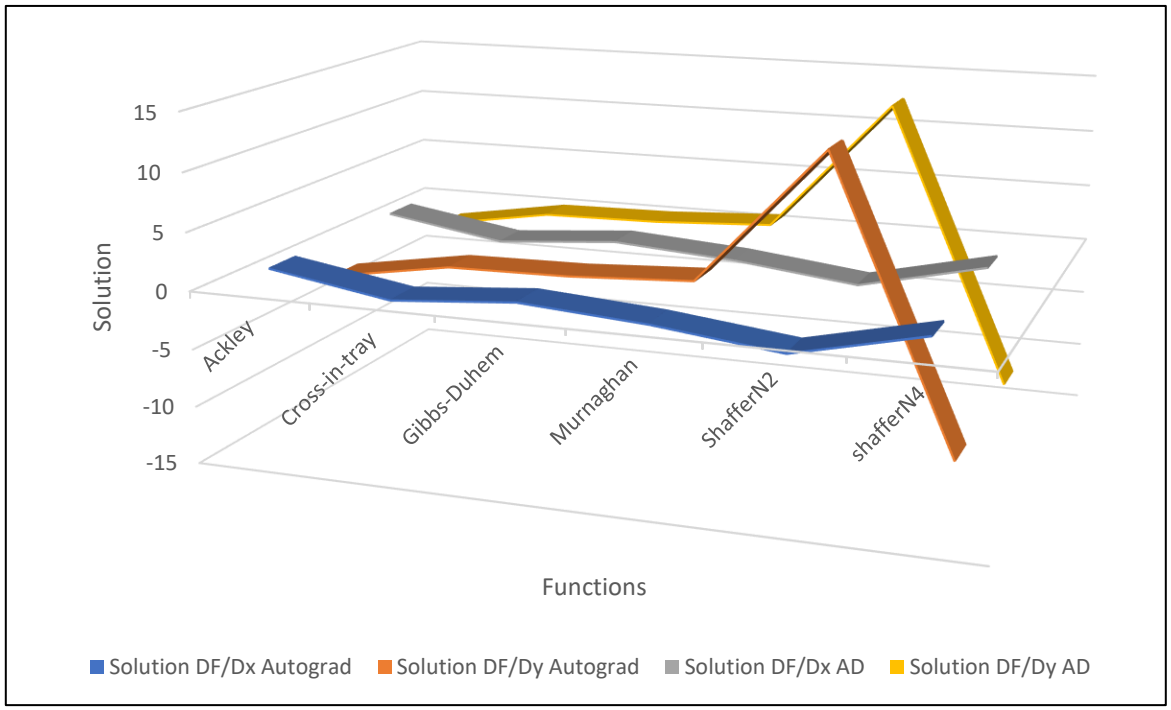

Figura 6.5: Gráfico solución numérica obtenida diferenciando con Autograd y AD

Ya de forma aislada se puede ver diferencias notorias de una herramienta u otra, y donde Autograd se consolida como la herramienta de diferenciación automática más rápida probada en este experimento, sobrepasando a AD en todas las pruebas. Una observación interesante a mencionar es que Autograd específicamente en la derivada parcial $\frac{dF}{dy}$, si se revisan los resultados, respecto a $\frac{dF}{dx}$ siempre menor, en cambio con AD el caso es inverso.

Función	**Tiempo $\frac{dF}{dx}$ Autograd**	**Tiempo $\frac{dF}{dy}$ Autograd**	**Tiempo $\frac{dF}{dx}$ AD**	**Tiempo $\frac{dF}{dy}$ AD**
Ackley	0,000423111	0,000277728	0,000860444	0,001555752
Cross-in-tray	0,000380839	0,000277728	0,000823308	0,00141432
Gibbs-Duhem	0,000346864	0,000252839	0,000384	0,000350815
Murnaghan	0,000279703	0,000187654	0,000330271	0,000209383
ShafferN2	0,000402173	0,000248099	0,000818962	0,00154548
shafferN4	0,000404148	0,000286815	0,000902715	0,001816492

Tabla 6.10: Tiempo de Autograd y AD para obtener su solución numérica

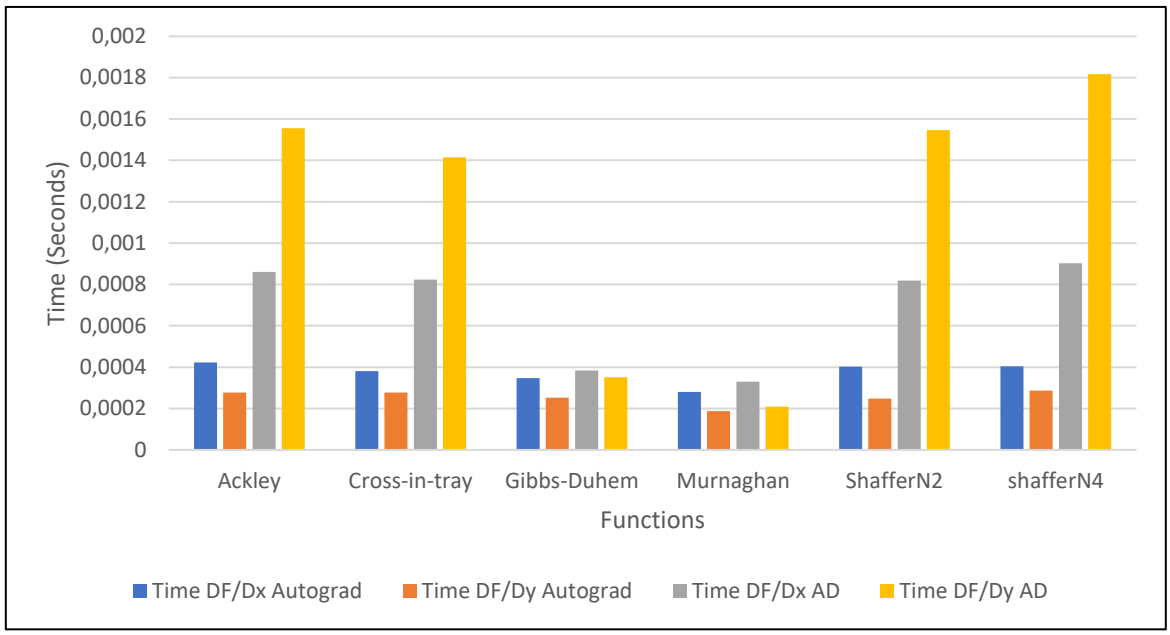

Figura 6.6: Gráfico tiempo de Autograd y AD para obtener su solución numérica

Tras estos resultados como respaldo, Autograd será la herramienta de diferenciación automática que será medida con la confección de un gradiente cerrado (o elaborado de manera manual), en el algoritmo de regresión logística implementado en el capítulo anterior.

6.5 Pruebas tiempo de velocidad de regresión logística con distintos tipos de datasets

Como se mencionó en la introducción de este capítulo, para las pruebas efectuadas en el algoritmo de regresión logística se trabajó con distintos dataset de distintos tipos de dimensionalidad, esto con el fin de visualizar resultados a diferentes escalas y así poder realizar un análisis más certero. Como un detalle adicional respecto a las pruebas realizadas en esta parte del experimento, es que para cada caso el algoritmo se ejecutó un total de 20 veces, y de esas

20 veces se obtuvieron el tiempo medio y la varianza, los cuales son los resultados visualizados en esta sección.

También hay que mencionar que, dada la robustez de este segmento del experimento, se consideró someter las pruebas de Autograd en un servidor con mejores prestaciones que la CPU de las secciones anteriores. Esto con el fin de equilibrar los resultados finales obtenidos.

6.5.1 Datasets de dimensionalidad baja

Los dataset de baja dimensionalidad se caracterizan por ser considerados “datos de juguete”, dado que estos poseen una cantidad mínima de datos, usados generalmente en problemas de ejemplo dada su maniobrabilidad y tamaño. En este caso se encogieron 3 datasets de este tipo los cuales se enumeran a continuación:

Dataset	**N° Total de Muestras**	**N° Clases**	**N° muestras por clase**	**Dimensionalidad**
Iris [61]	150	3	-	4
Wine [62]	178	3	59-71-48	13
Breast_Cancer [63]	569	2	212-357	30

Tabla 6.11:Dataset de dimensionalidad baja

Al tener sus parámetros de manera fija, el parámetro que fue variando fueron las solamente las iteraciones.

6.5.1.1 Pruebas dataset Iris

Dataset	Iteraciones	Tiempo Manual CPU	Tiempo Autograd CPU	Tiempo Autograd TUI
Iris	1000	0,380357165	1,463742703	0,515876067
Iris	2000	0,763299169	2,771230811	0,990848827
Iris	3000	1,208933831	4,161807393	1,464397347
Iris	4000	1,479698332	5,451016117	1,972035873
Iris	5000	1,954432291	6,67294682	2,453182793
Iris	6000	2,432841098	8,675158906	2,949151099
Iris	7000	3,395552565	9,522063534	3,493963933
Iris	8000	3,303002362	9,732206825	4,189566755
Iris	9000	3,776395988	11,22845365	4,409406507
Iris	10000	4,020561615	13,15236475	4,897174907

Tabla 6.12: Tiempo medio dataset iris

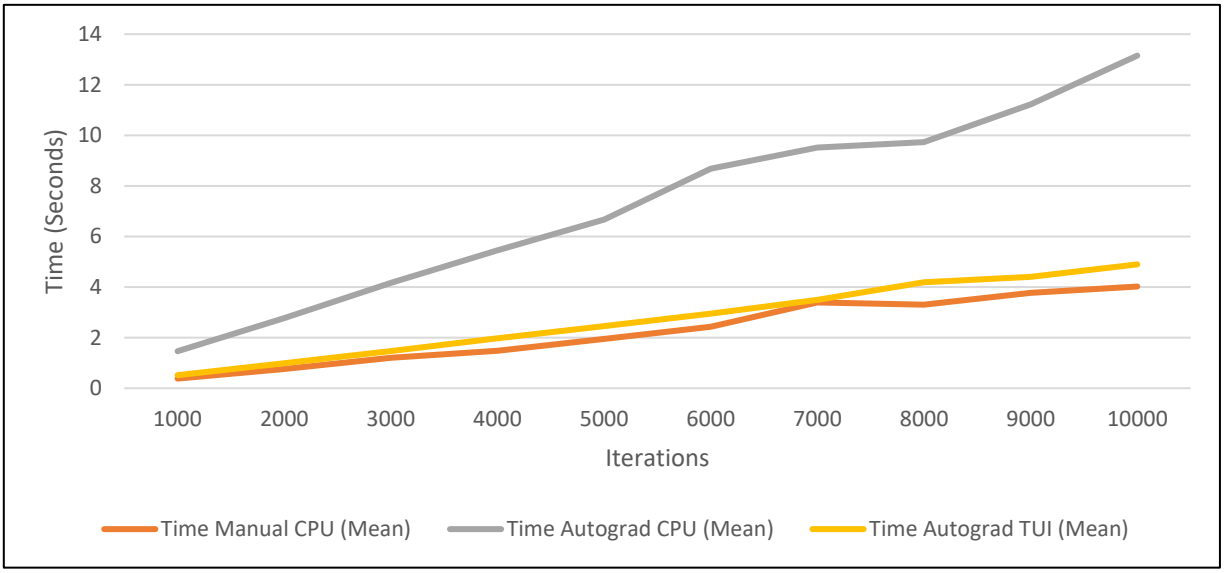

Figura 6.7: Gráfico tiempo medio dataset iris

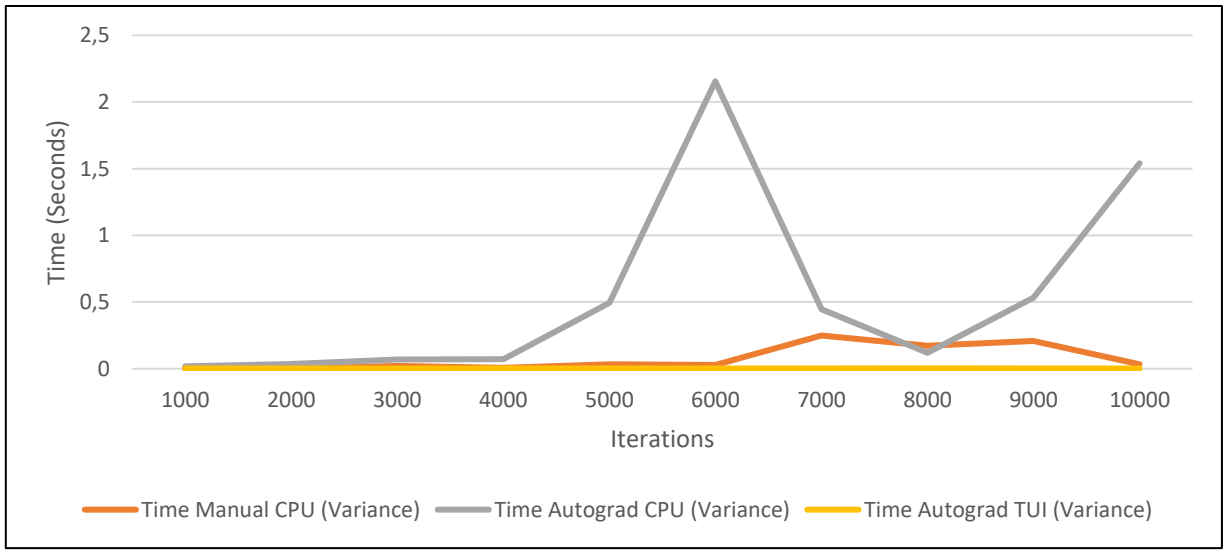

Figura 6.8: Gráfico varianza de tiempo dataset iris

6.5.1.2 Pruebas dataset Breast Cáncer

Dataset	Iteraciones	Tiempo Manual CPU	Tiempo Autograd CPU	Tiempo Autograd TUI
Breast_Cancer	1000	0,51236598	1,382479781	0,669923702
Breast_Cancer	2000	1,024805205	3,131435767	1,375069096
Breast_Cancer	3000	1,544909584	4,255380089	2,013908936
Breast_Cancer	4000	2,062154938	5,942138542	2,718626361
Breast_Cancer	5000	2,609982254	7,272098287	3,344952759
Breast_Cancer	6000	3,173113311	8,612757217	4,040159759
Breast_Cancer	7000	3,651400574	10,19437289	4,70788542
Breast_Cancer	8000	4,157842202	11,59322105	5,378557486
Breast_Cancer	9000	4,749483101	12,97314978	6,045158941
Breast_Cancer	10000	5,237222938	14,60043973	6,735985227

Tabla 6.13: Tiempo medio dataset breast cáncer

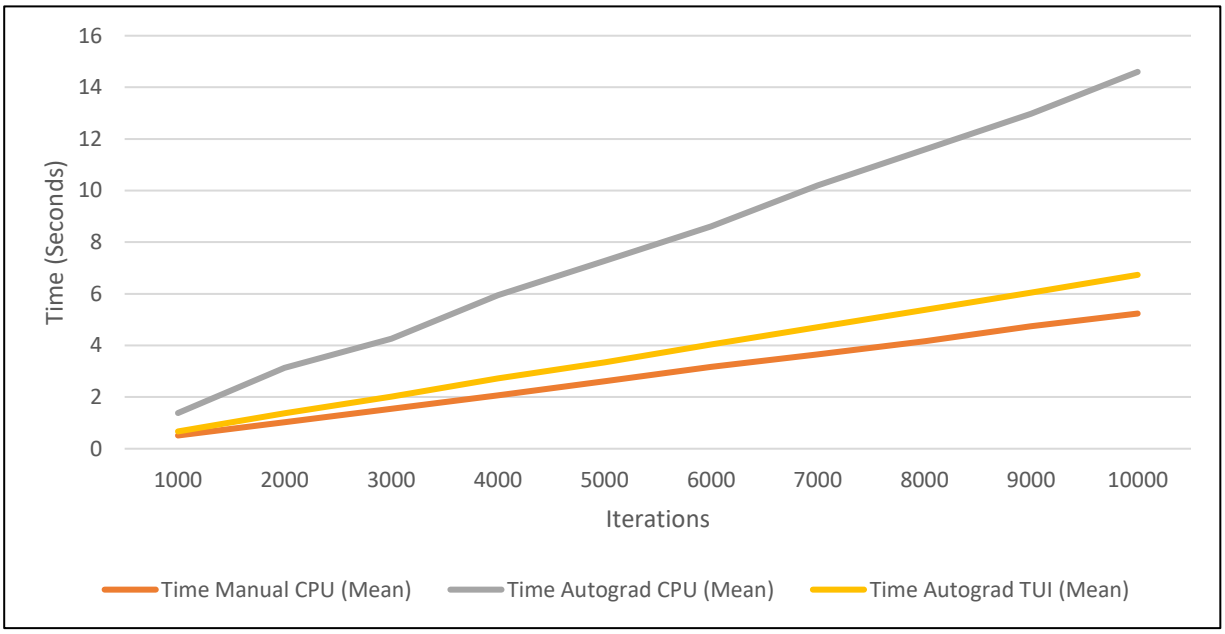

Figura 6.9: Gráfico tiempo medio dataset breast cáncer

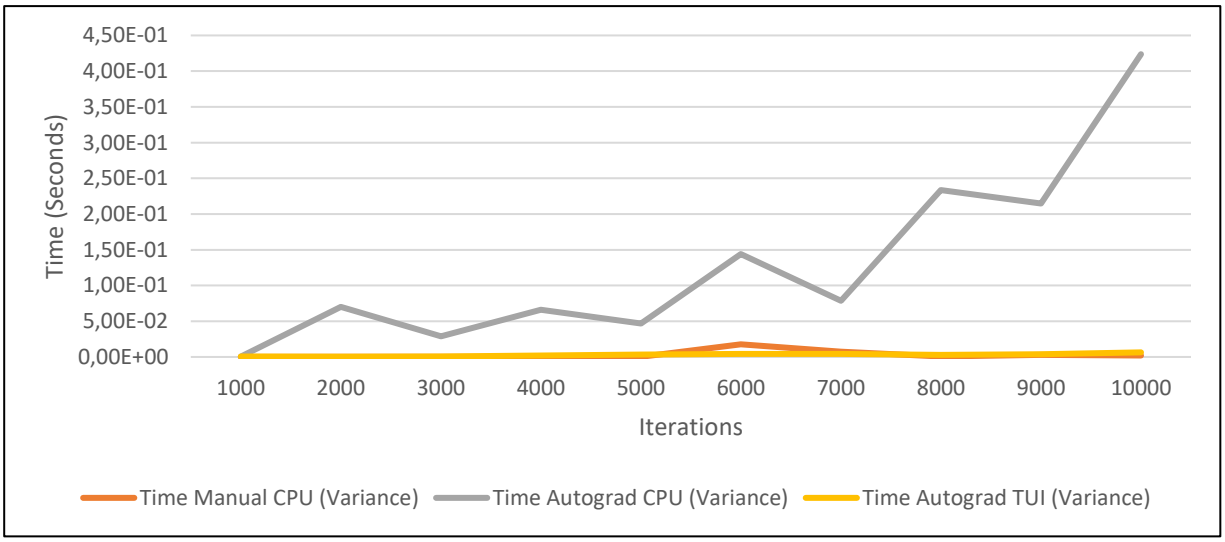

Figura 6.10: Gráfico varianza de tiempo dataset breast cancer

6.5.1.3 Pruebas dataset Wine

Dataset	Iteraciones	Tiempo Manual CPU	Tiempo Autograd CPU	Tiempo Autograd TUI
Wine	1000	0,428844918	1,413523038	0,552086258
Wine	2000	0,901859274	2,603735353	1,092769969
Wine	3000	1,341661517	4,054243816	1,634834647
Wine	4000	1,762975346	4,997608374	2,182029951
Wine	5000	2,273532955	6,670911134	2,628380501

Tabla 6.14: Tiempo medio dataset wine

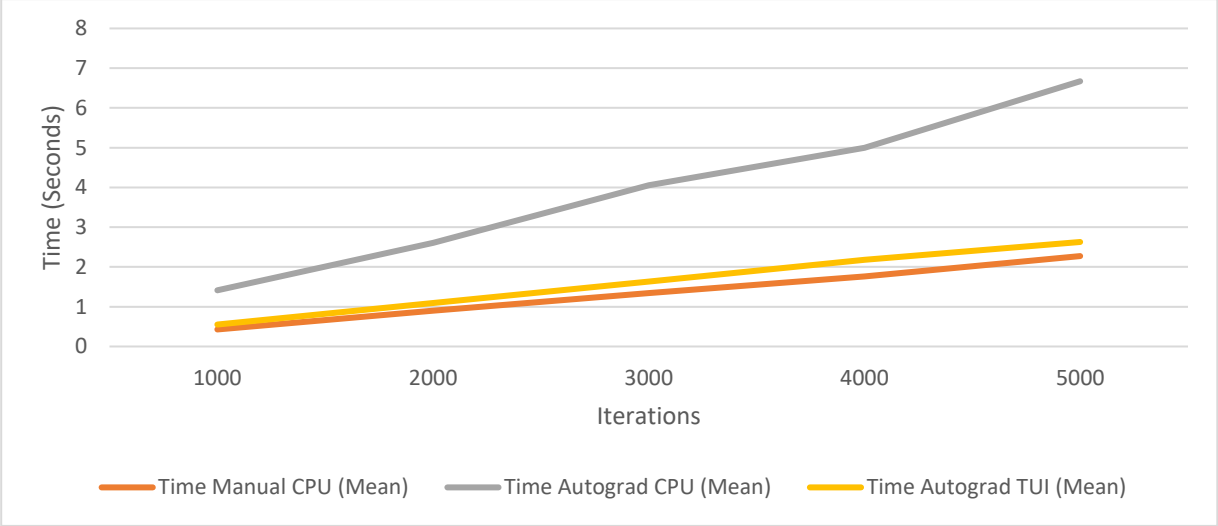

Figura 6.11: Gráfico tiempo medio dataset wine

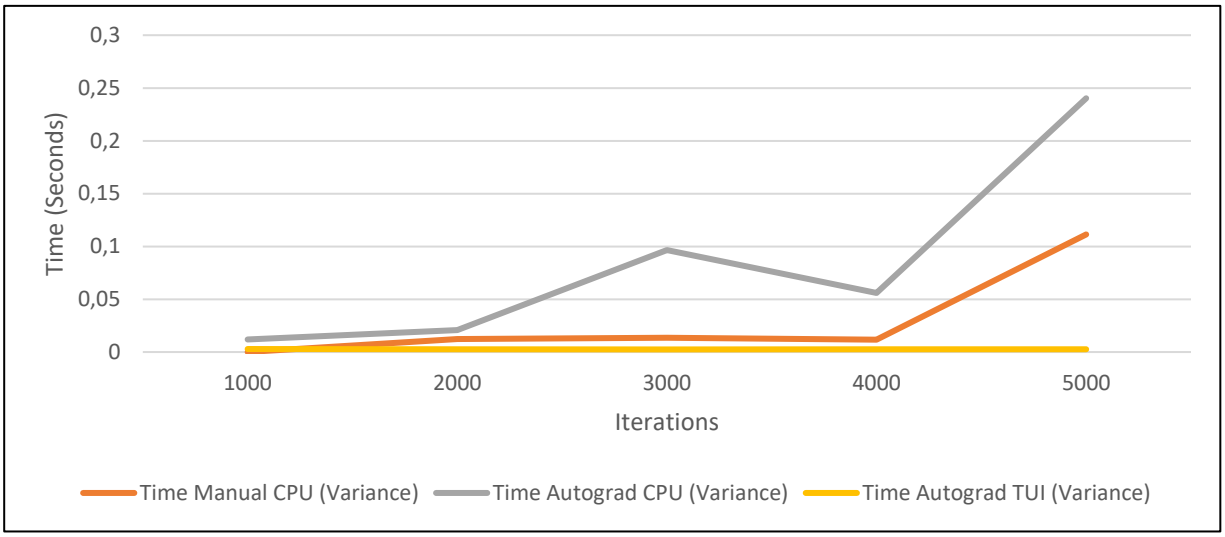

Figura 6.12: Gráfico varianza de tiempo dataset wine

De acuerdo con los resultados obtenidos de las pruebas realizadas con dataset de baja dimensionalidad se puede apreciar claramente en las figuras 6.7, 6.9 y 6.11, que el tiempo medio de ejecución del cálculo del gradiente aumenta de una forma directamente proporcional a la variación iteraciones. Sin embargo, el detalle más notorio, es la amplia brecha que existe entre el la obtención del gradiente de forma automática versus una forma manual, si se compara CPU versus CPU. Ahora bien, si se considera el tiempo del servidor respecto al gradiente manual esta brecha se estrecha. Esto obviamente sucede por las características del hardware del servidor utilizado.

En el caso de los resultados obtenidos de la varianza del tiempo de ejecución, la tendencia se repite. Sin embargo, si se observan las figuras 6.8, 6.10 y 6.12, se puede visualizar una inestabilidad por parte del tiempo de ejecución de Autograd en CPU, principalmente a medida que las iteraciones iban en aumento. Lo que claramente generó los altos tiempos medios de ejecución.

6.5.2 Datasets de dimensionalidad Media

Estos dataset pueden ser considerados de baja dimensionalidad, sin embargo, la ventaja que poseen respecto al grupo anterior es la parametrización inicial que se les puede dar. Donde se puede manipular la cantidad de datos, las muestras por clase o el ruido de estas muestras. por lo que mejoran las características del experimento de acuerdo se varía la parametrización

Los parámetros que se variaron fueron las iteraciones en los valores 1000, 5000, y 10000. Además de las muestras de cada dataset desde 500 a 300000 muestras de carácter aleatorio y por último el ruido, que se modificó a nivel de dataset siendo 0.1 para make moons y 0.5 para make circles.

Dataset	N° Total Muestras	N° Clases	N° muestras por clase	Dimensionalidad	Ruido
Make_Moons [64]	500-300000	2	Aleatorio	2	0.1
Make_Circles [65]	500-300000	2	Aleatorio	2	0.5

Tabla 6.15: Dataset dimensionalidad media

6.5.2.1 Pruebas dataset Make Moons

Dataset	Datos	Tiempo Manual CPU 1000 IT	Tiempo Autograd CPU 1000 IT	Tiempo Autograd TUI 1000 IT
Make_Moons	500	0,547890866	0,641893721	0,619951713
Make_Moons	2500	0,83159395	0,965099867	1,240884209
Make_Moons	5000	1,356699135	1,426611118	1,794962573
Make_Moons	15000	1,1040957	3,305707784	1,577531815
Make_Moons	20000	1,468281605	4,244723406	1,821769762
Make_Moons	25000	2,174953877	4,991836477	2,433511138
Make_Moons	50000	4,271871884	8,985445732	3,439704204
Make_Moons	75000	4,495709435	13,00433645	4,615091372

Make_Moons	150000	7,718493536	25,75982592	7,972357559
Make_Moons	300000	15,61162504	77,11150074	17,59260734

Tabla 6.16: Tiempo medio dataset make_moons 1000 iteraciones

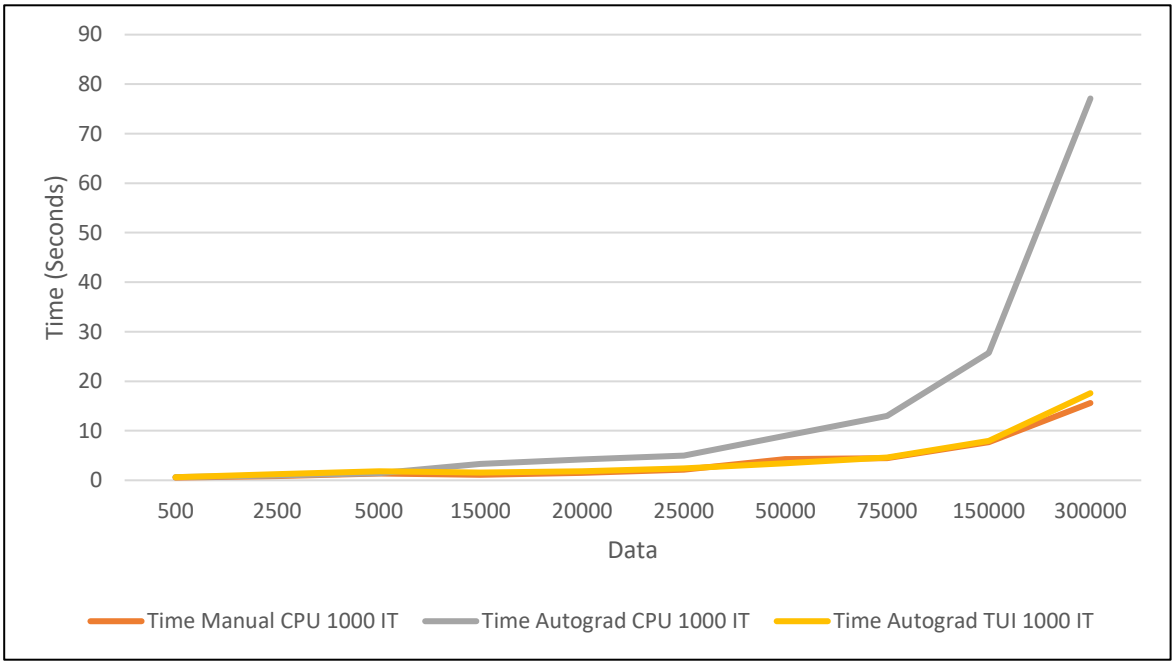

Figura 6.13: Gráfico Tiempo medio dataset make_moons 1000 iteraciones

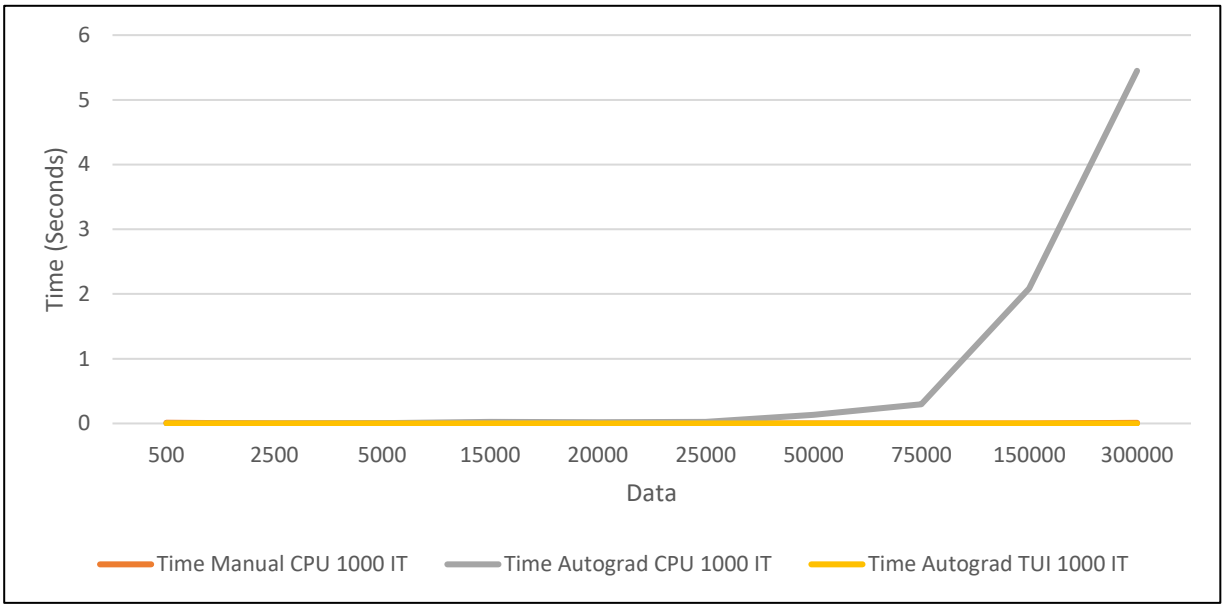

Figura 6.14: Gráfico varianza de tiempo dataset make_moons 1000 iteraciones

Dataset	Datos	Tiempo Manual CPU 5000 IT	Tiempo Autograd CPU 5000 IT	Tiempo Autograd TUI 5000 IT
Make_Moons	500	2,558564714	3,247032103	3,071724832
Make_Moons	2500	4,178098407	4,926547257	5,962758732
Make_Moons	5000	6,865354714	7,171956627	8,797259331
Make_Moons	15000	5,499110164	15,86046645	7,708282876
Make_Moons	20000	7,277856709	20,89312248	9,089201427
Make_Moons	25000	11,35825376	24,89298502	12,84386024
Make_Moons	50000	21,43126662	46,73674675	19,24043896
Make_Moons	75000	28,11941976	65,81453177	30,28267522
Make_Moons	150000	39,71849828	131,1148542	40,7722208
Make_Moons	300000	80,76289548	395,1578194	90,94859171

Tabla 6.17: Tiempo medio dataset make_moons 5000 iteraciones

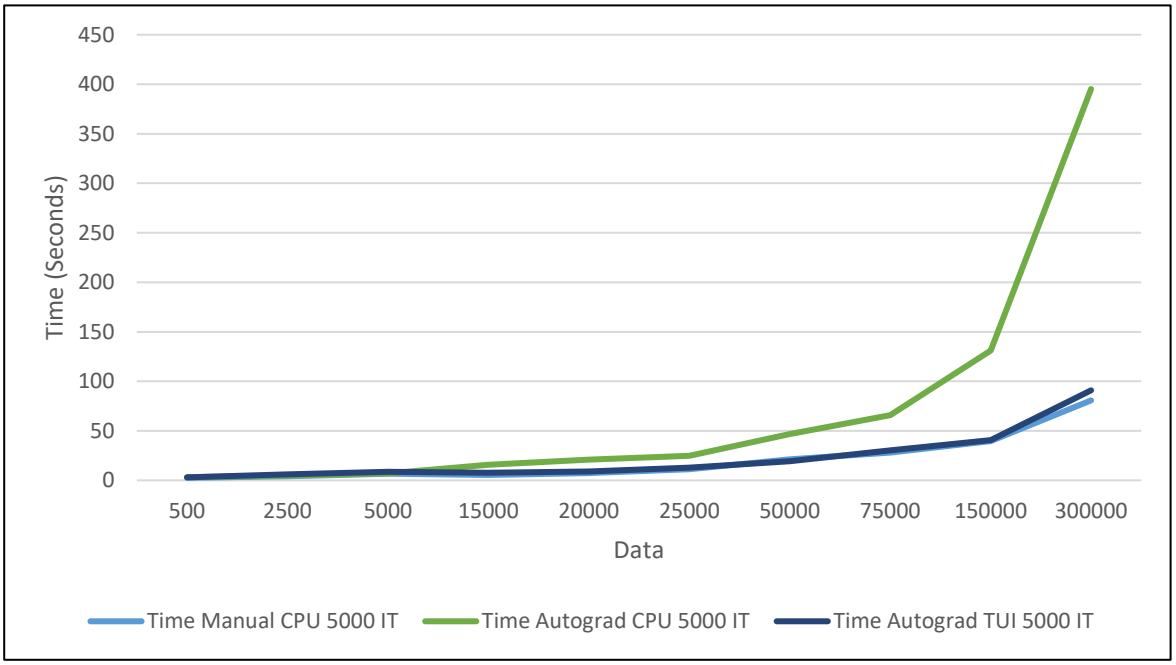

Figura 6.15: Gráfico tiempo medio dataset make_moons 5000 iteraciones

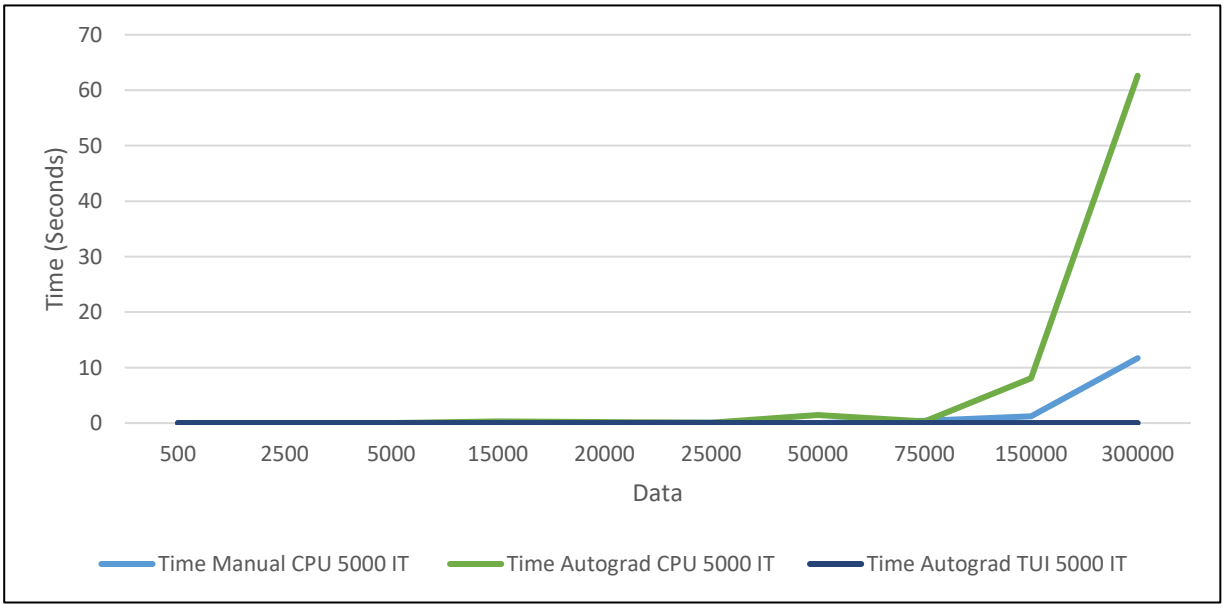

Figura 6.16: Gráfico varianza de tiempo dataset make_moons 5000 iteraciones

Dataset	Datos	Tiempo Manual CPU 10000 IT	Tiempo Autograd CPU 10000 IT	Tiempo Autograd TUI 10000 IT
Make_Moons	500	5,10178712	6,453814374	6,106623673
Make_Moons	2500	8,412839279	9,949419477	12,29696152
Make_Moons	5000	13,80917702	14,41105399	17,40962782
Make_Moons	15000	11,00862202	31,47730986	15,60946927
Make_Moons	20000	14,53936678	41,60589376	17,66040201
Make_Moons	25000	21,68335897	50,17016236	26,10168469
Make_Moons	50000	36,10554097	95,37147874	37,82827532
Make_Moons	75000	55,69532346	130,3645673	60,89602098
Make_Moons	150000	79,75821842	258,883683	77,96038723
Make_Moons	300000	157,1374401	789,400102	177,4419723

Tabla 6.18: Tiempo medio dataset make_moons 10000 iteraciones

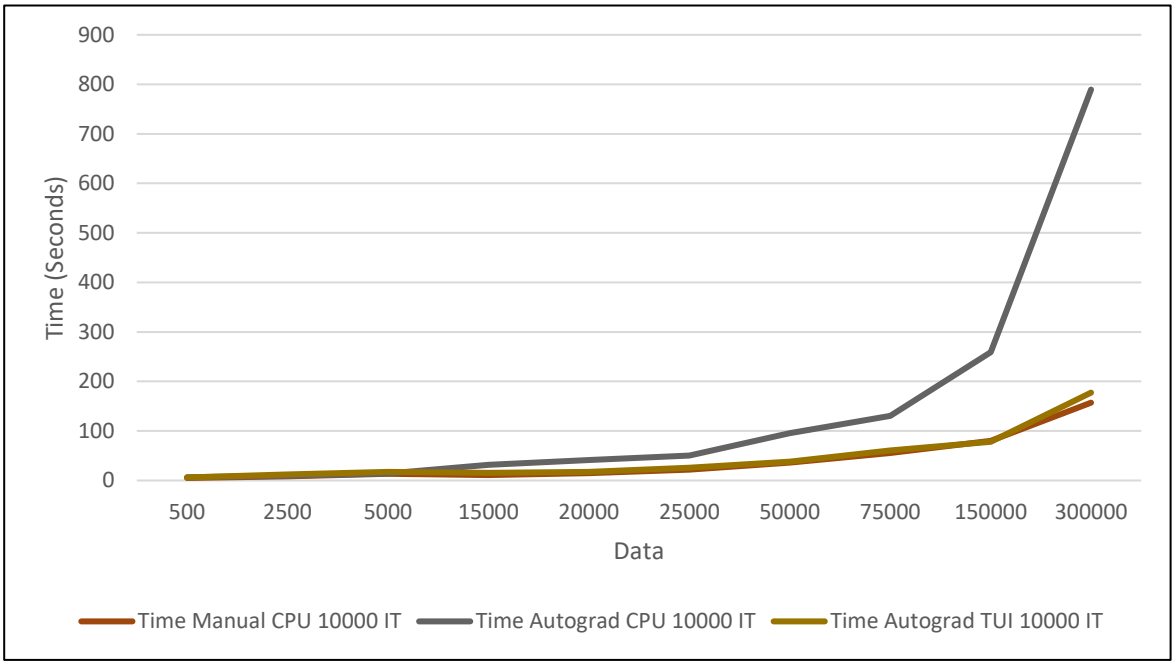

Figura 6.17: Gráfico tiempo medio dataset make_moons 10000 iteraciones

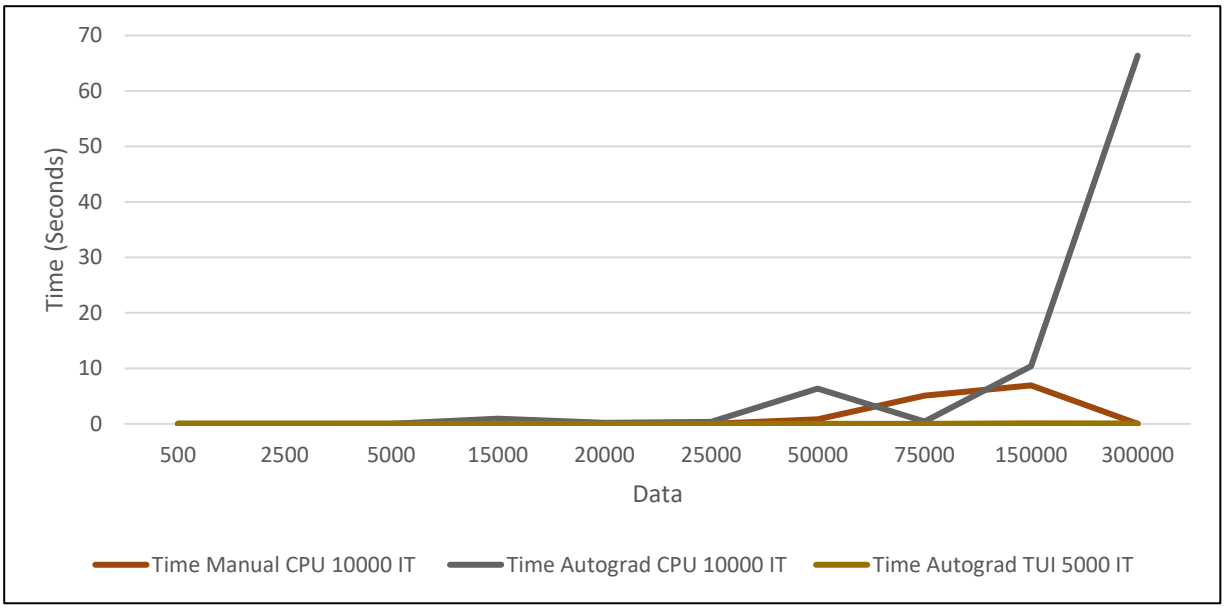

Figura 6.18: Gráfico varianza de tiempo dataset make_moons 10000 iteraciones

6.5.2.2 Pruebas dataset Make Circles

Dataset	Datos	Tiempo Manual CPU 1000 IT	Tiempo Autograd CPU 1000 IT	Tiempo Autograd TUI 1000 IT
Make_Circles	500	0,42556954	1,481258674	0,593308973
Make_Circles	2500	1,168836351	2,059816386	1,178917766
Make_Circles	5000	1,642934781	3,143024639	1,808943081
Make_Circles	15000	1,108468161	6,822573188	1,592665815
Make_Circles	20000	1,462116455	8,899922568	1,804415178
Make_Circles	25000	1,828993399	11,08941398	2,495630836
Make_Circles	50000	3,390279	20,91604525	3,473963785
Make_Circles	75000	5,048722134	29,24018225	5,26293354
Make_Circles	150000	7,387588158	Memory Error	7,7144382
Make_Circles	300000	14,91641433	Memory Error	15,40210614

Tabla 6.19: Tiempo medio dataset make_circles 1000 iteraciones

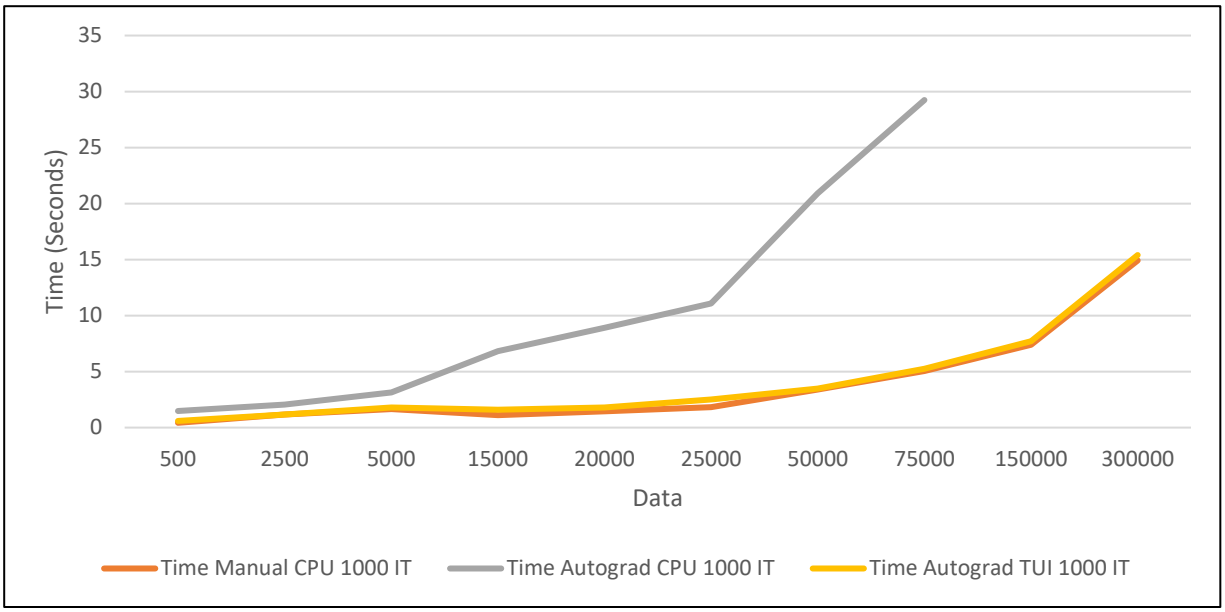

Figura 6.19: Gráfico tiempo medio dataset make_circles 1000 iteraciones

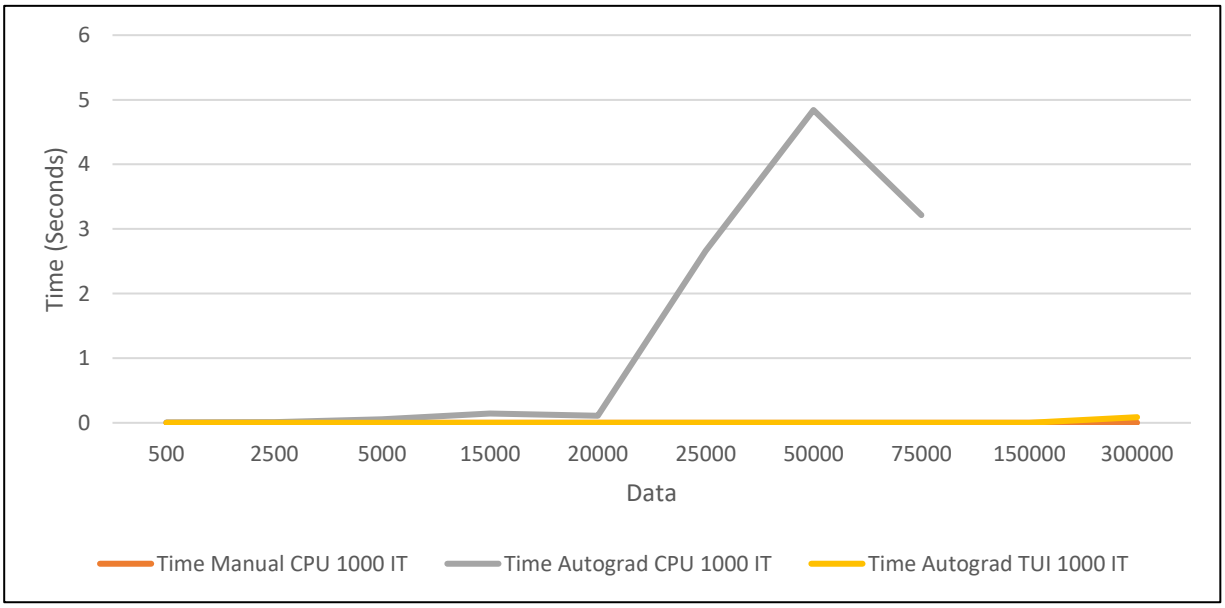

Figura 6.20: Gráfico varianza de tiempo dataset make_circles 1000 iteraciones

Dataset	Datos	Tiempo Manual CPU 5000 IT	Tiempo Autograd CPU 5000 IT	Tiempo Autograd TUI 5000 IT
Make_Circles	500	2,125713224	7,729425806	3,038521576
Make_Circles	2500	5,916531049	10,20602837	6,066298914
Make_Circles	5000	8,099898647	16,33115396	9,01990037
Make_Circles	15000	5,502933805	33,49629525	7,721518993
Make_Circles	20000	7,309258877	44,06110112	9,070097733
Make_Circles	25000	9,108961711	51,90065148	12,23499799
Make_Circles	50000	16,85879124	104,1548493	17,01107078
Make_Circles	75000	25,16224509	Memory Error	26,11388597
Make_Circles	150000	36,69374796	Memory Error	39,41911621
Make_Circles	300000	74,46614541	Memory Error	88,18439908

Tabla 6.20: Tiempo medio dataset make_circles 5000 iteraciones

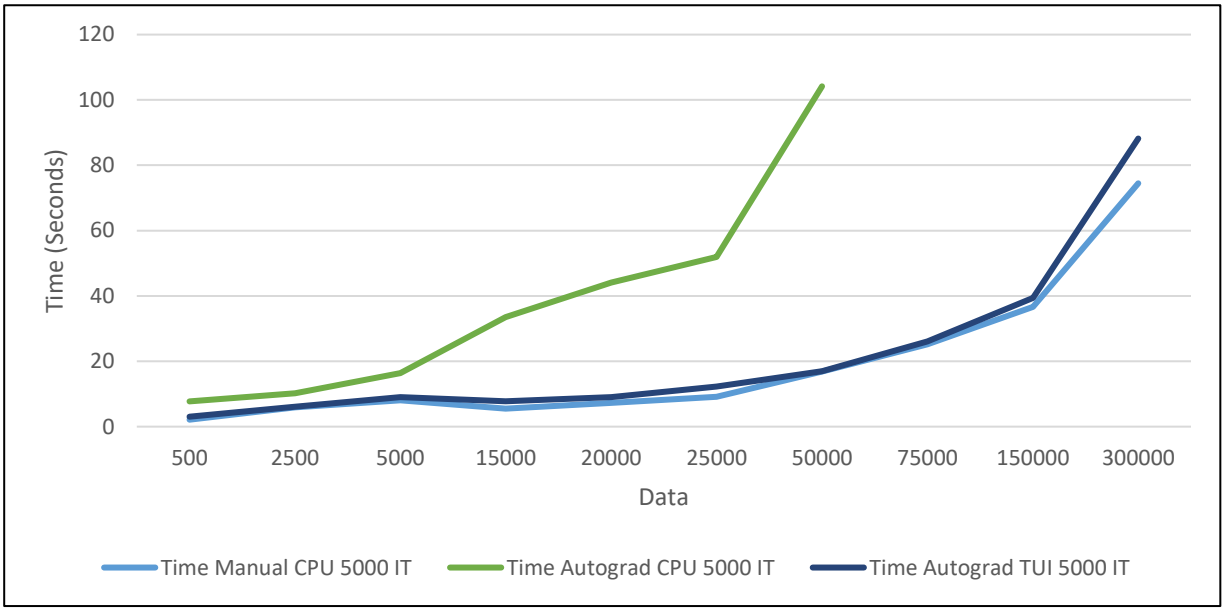

Figura 6.21: Gráfico tiempo medio dataset make_circles 5000 iteraciones

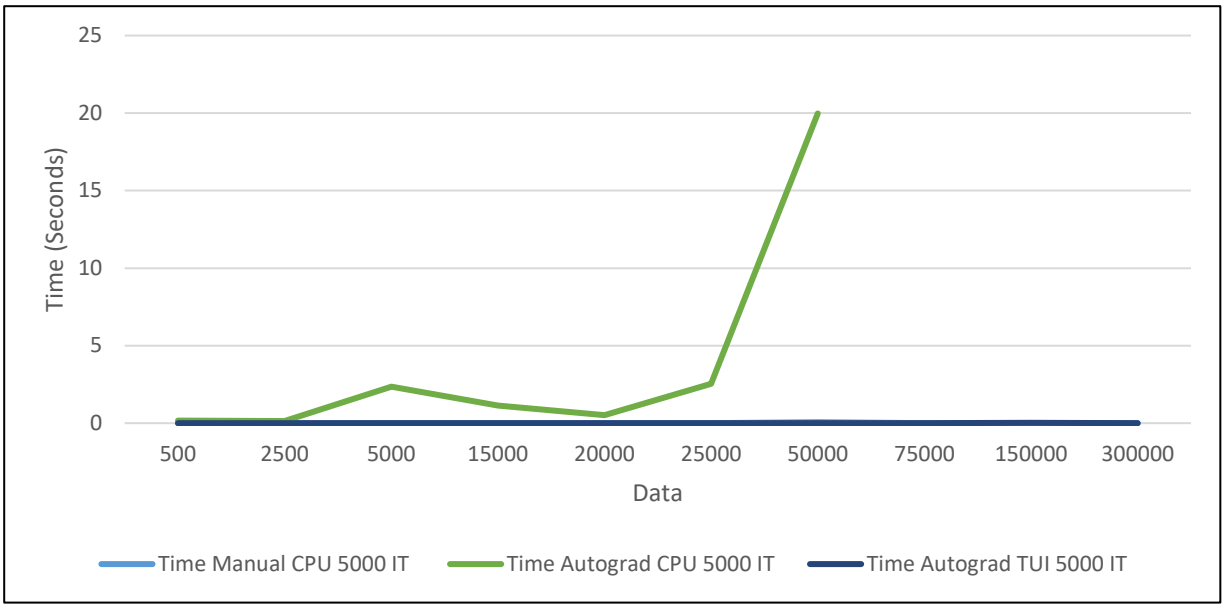

Figura 6.22: Gráfico Varianza de tiempo dataset make_circles 5000 iteraciones

Dataset	Datos	Tiempo Manual CPU 10000 IT	Tiempo Autograd CPU 10000 IT	Tiempo Autograd TUI 10000 IT
Make_Circles	500	4,293797674	13,88879734	5,752625394
Make_Circles	2500	11,813015	20,80335735	11,64681327
Make_Circles	5000	16,20739559	32,22808799	17,71709898
Make_Circles	15000	11,00329633	66,62769208	15,60717063
Make_Circles	20000	14,69194626	81,07378706	18,31857233
Make_Circles	25000	18,27270752	105,3800821	25,02235823
Make_Circles	50000	33,55079263	208,9679449	34,36521258
Make_Circles	75000	50,2893589	Memory Error	52,31530824
Make_Circles	150000	73,42248453	Memory Error	78,53207932
Make_Circles	300000	147,7413975	Memory Error	151,6324435

Tabla 6.21: Tiempo medio dataset make_circles 10000 iteraciones

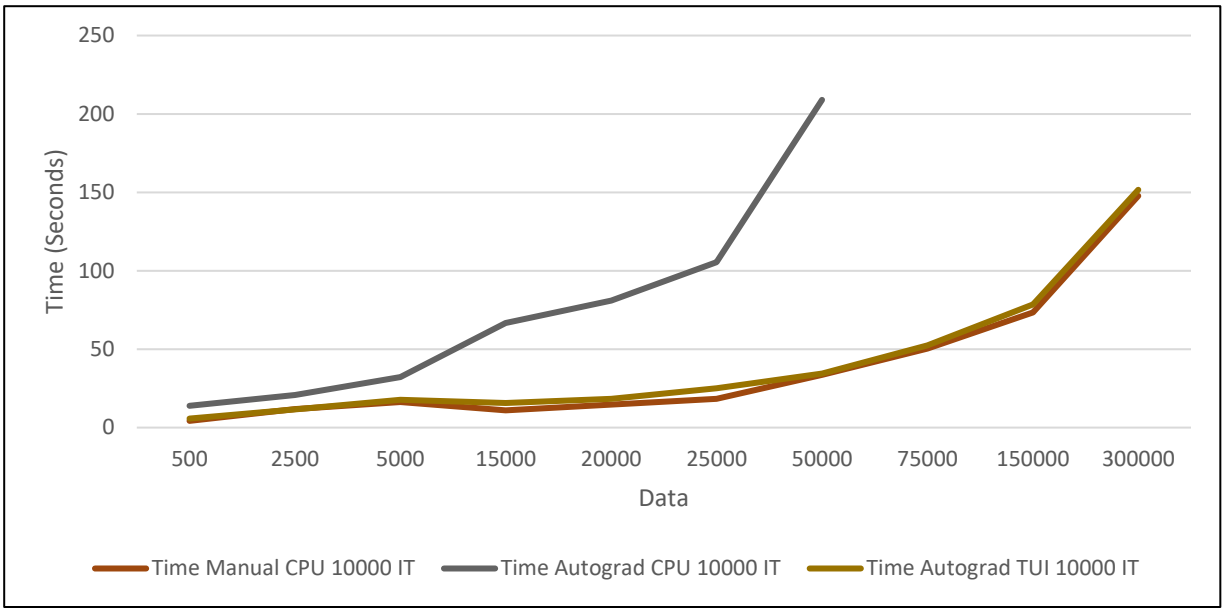

Figura 6.23: Gráfico tiempo medio dataset make_circles 10000 iteraciones

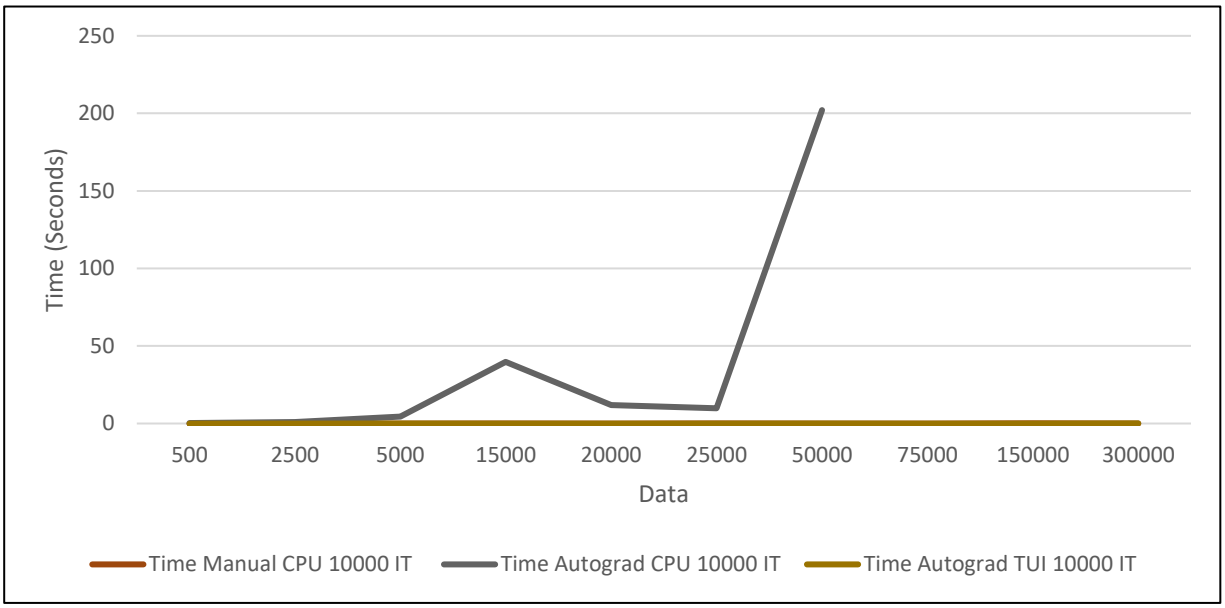

Figura 6.24: Gráfico varianza de tiempo dataset make_circles 10000 iteraciones

Dados los resultados obtenidos de las pruebas realizadas con dataset de dimensionalidad media a diferencia de los baja dimensionalidad, se puede apreciar que la parametrización de los datos es un tema a considerar si lo que se requiere conseguir es estabilidad. Dado que si se observan las figuras 6.13. 6.15 y 6.17 de los tiempos medios del dataset make moons, si bien siguen la tendencia de los datasets anteriores, los resultados son más estables y la brecha, aunque existe es menor. Lo mismo ocurre con la varianza para este caso, haciéndose más considerable cuando las iteraciones y las muestras alcanzan sus niveles más altos.

En el caso del dataset make circles, se puede apreciar en las figuras 6.19, 6.21, 6.23 que el comportamiento es similar. Pero a diferencia del dataset anterior se consideró un ruido más alto, siendo este 0.5. lo que significó que los tiempos medios de ejecución fueran más altos desde iteraciones y cantidad de muestras menores al igual que su varianza.

Si se habla de la obtención del gradiente de forma automática versus una forma manual, si se compara CPU versus CPU la situación se repite la brecha existe, aunque a otra escala. Pero si se considera el tiempo del servidor respecto al gradiente manual esta brecha se estrecha pudiendo igualarse en algunos segmentos del experimento. Un detalle importante a tener en cuenta de esta sección es que Autograd, en su desempeño en CPU, cuando alcanzo los niveles más altos tanto de iteraciones como de cantidad de muestras arrojó un error de memoria. Sobre esto se hablará al finalizar el capítulo.

6.5.3 Datasets de dimensionalidad Alta

Este grupo de dataset se caracteriza por ser altamente parametrizable, sobre todo respecto a las clases y dimensiones que eran fijas en los datasets anteriores, haciendo el experimento más complejo y denso respecto al procesamiento de datos. Para el dataset Make blobs se variaron las muestras y para el dataset aleatorio se variaron las muestras las clases y las dimensiones

Dataset	N° Total de Muestras	N ° Clases	N° muestras por clase	Dimensionalidad
Make_Blobs [66]	500-300000	10	Aleatorio	20
Aleatorio	1000-500000	3-800	Aleatorio	10-4000

Tabla 6.22: Datasets dimensionalidad alta

6.5.3.1 Pruebas dataset Make Blobs

Dataset	Datos	Tiempo Manual CPU 1000 IT	Tiempo Autograd CPU 1000 IT	Tiempo Autograd TUI 1000 IT
Make_Blobs	500	0,503865636	1,614744144	0,585806417
Make_Blobs	2500	1,084387943	2,208880801	1,218010616
Make_Blobs	5000	1,707866384	3,190167737	1,775124931
Make_Blobs	15000	1,430077073	8,227601291	1,53311286
Make_Blobs	20000	1,808789326	9,606343569	1,836143589
Make_Blobs	25000	2,163829955	11,64423468	2,541429996
Make_Blobs	50000	3,217392796	20,52452272	3,830737448
Make_Blobs	75000	4,830785141	30,83774211	5,211181927
Make_Blobs	150000	7,361005779	89,47885957	7,669105864
Make_Blobs	300000	14,84815696	Memory Error	17,57813716

Tabla 6.23: Tiempo medio dataset make_blobs 1000 iteraciones

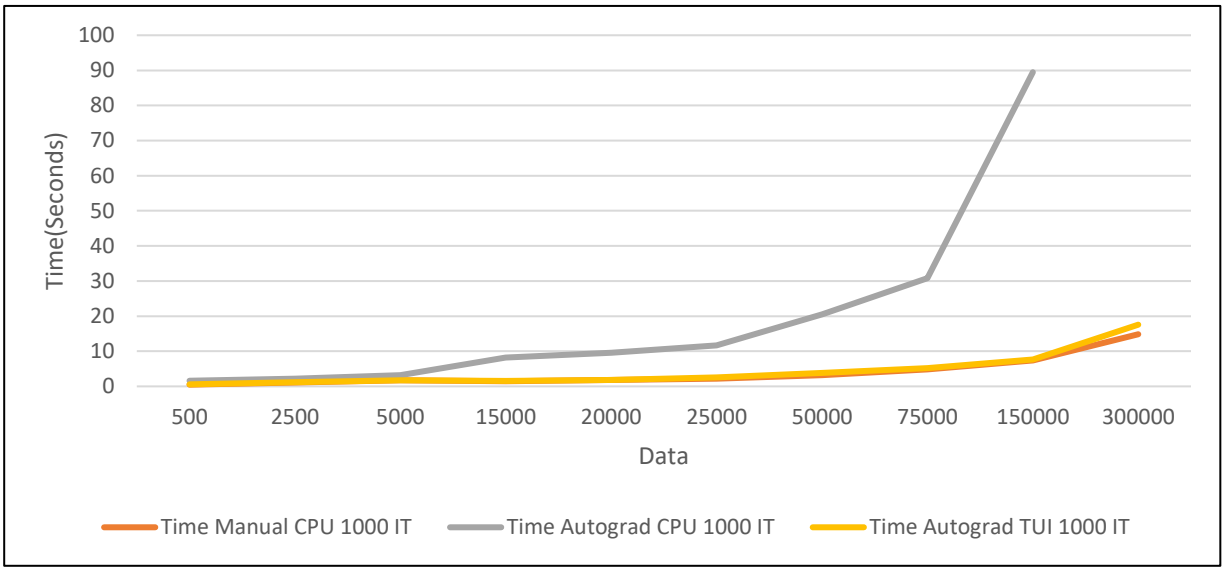

Figura 6.25: Gráfico tiempo medio dataset make_blobs 1000 iteraciones

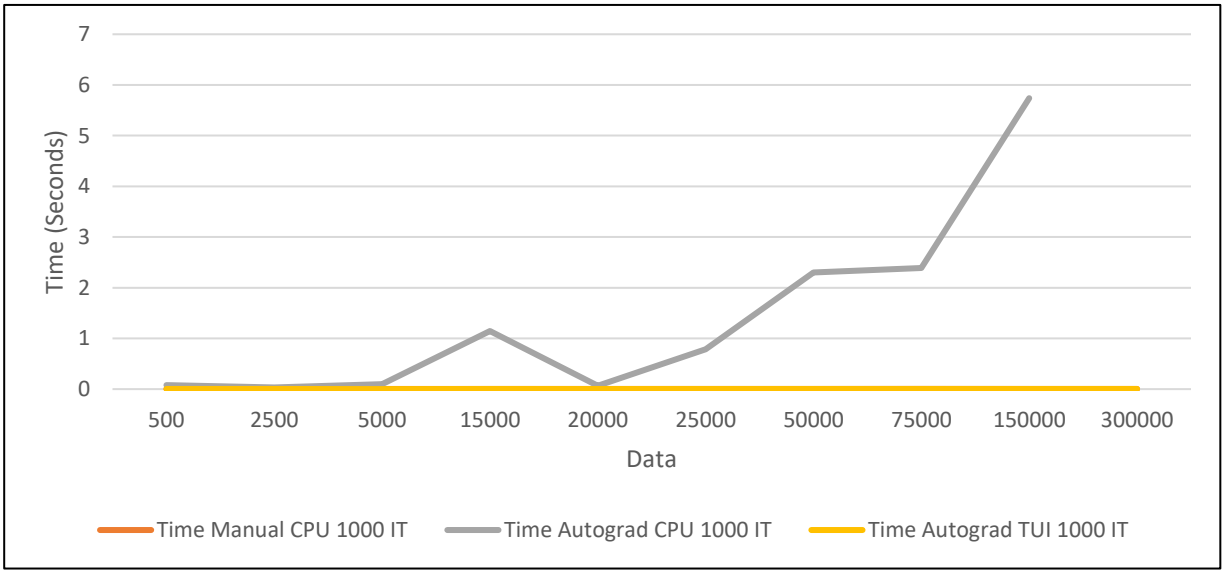

Figura 6.26: Gráfico varianza de tiempo dataset make_blobs 1000 iteraciones

Dataset	Datos	Tiempo Manual CPU 5000 IT	Tiempo Autograd CPU 5000 IT	Tiempo Autograd TUI 5000 IT
Make_Blobs	500	2,526318182	6,89290131	3,040982199
Make_Blobs	2500	5,532749307	10,97181159	5,929470825
Make_Blobs	5000	8,634279775	16,45398204	8,7416996
Make_Blobs	15000	7,251649796	38,7218529	7,667331219
Make_Blobs	20000	8,757336113	44,80990027	11,29655757
Make_Blobs	25000	10,89146966	57,82240997	12,89024377
Make_Blobs	50000	16,19829833	108,6093582	19,09761972
Make_Blobs	75000	24,22511876	148,300731	30,44064102
Make_Blobs	150000	36,67773714	Memory Error	38,62079082
Make_Blobs	300000	73,98898454	Memory Error	88,4153286

Tabla 6.24: Tiempo medio dataset make_blobs 5000 iteraciones

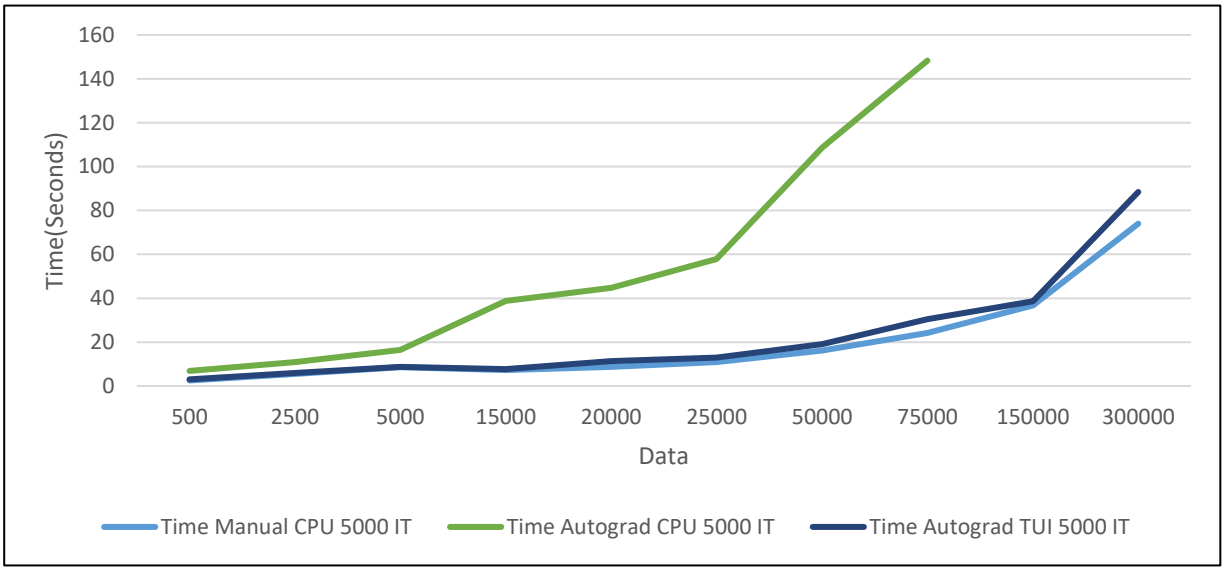

Figura 6.27: Gráfico tiempo medio dataset make_blobs 5000 iteraciones

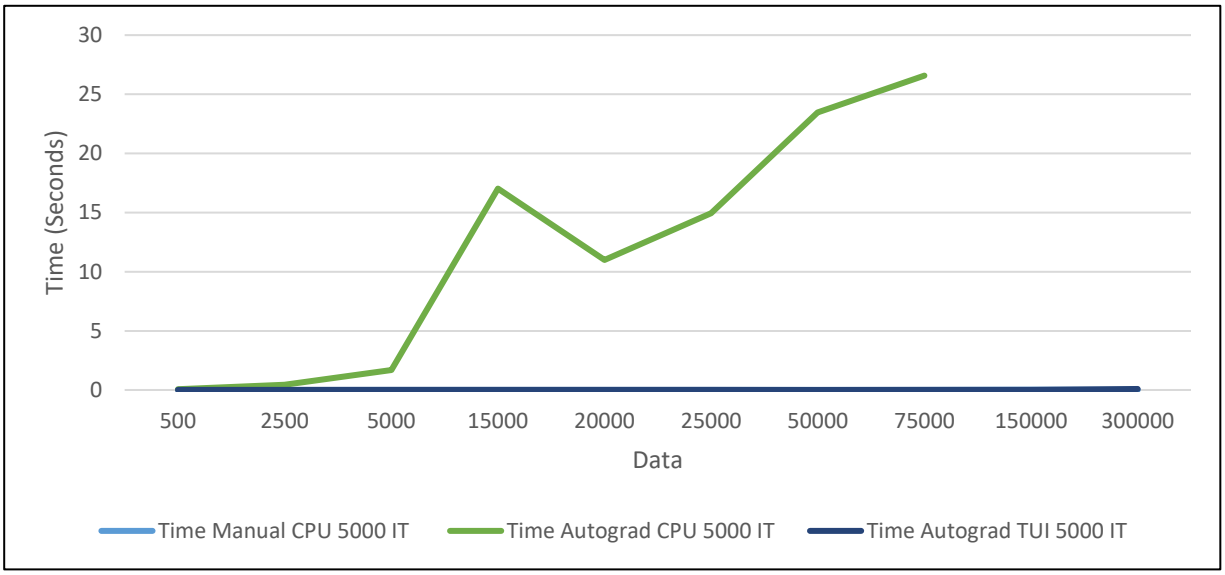

Figura 6.28: Gráfico varianza de tiempo dataset make_blobs 5000 iteraciones

Dataset	Datos	Tiempo Manual CPU 10000 IT	Tiempo Autograd CPU 10000 IT	Tiempo Autograd TUI 10000 IT
Make_Blobs	500	5,037155526	15,85907096	5,801024008
Make_Blobs	2500	11,13559776	21,62531936	11,76086764
Make_Blobs	5000	17,32056194	30,7897349	17,22579818
Make_Blobs	15000	14,55253997	67,06138972	15,29066515
Make_Blobs	20000	17,5824426	87,80035252	17,99232321
Make_Blobs	25000	21,73241752	104,3874757	24,4910182
Make_Blobs	50000	32,48101629	213,1270946	34,4163734
Make_Blobs	75000	48,53622227	306,8870617	61,23033824
Make_Blobs	150000	73,04813333	Memory Error	76,72604342
Make_Blobs	300000	147,5124632	Memory Error	177,6894119

Tabla 6.25: Tiempo medio dataset make_blobs 10000 iteraciones

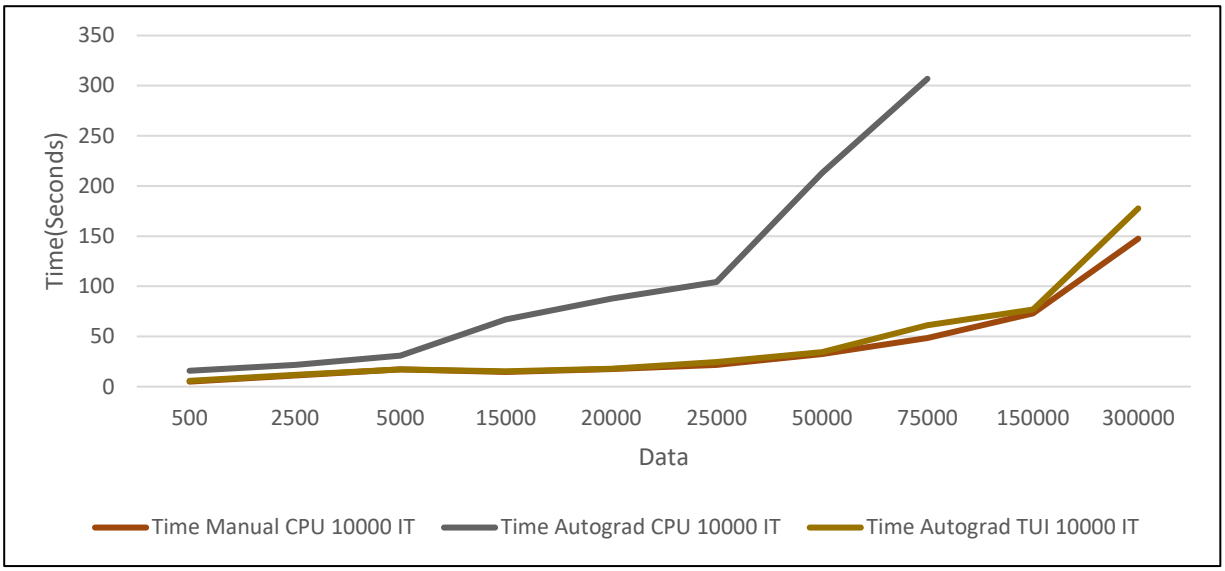

Figura 6.29: Gráfico tiempo medio dataset make_blobs 10000 iteraciones

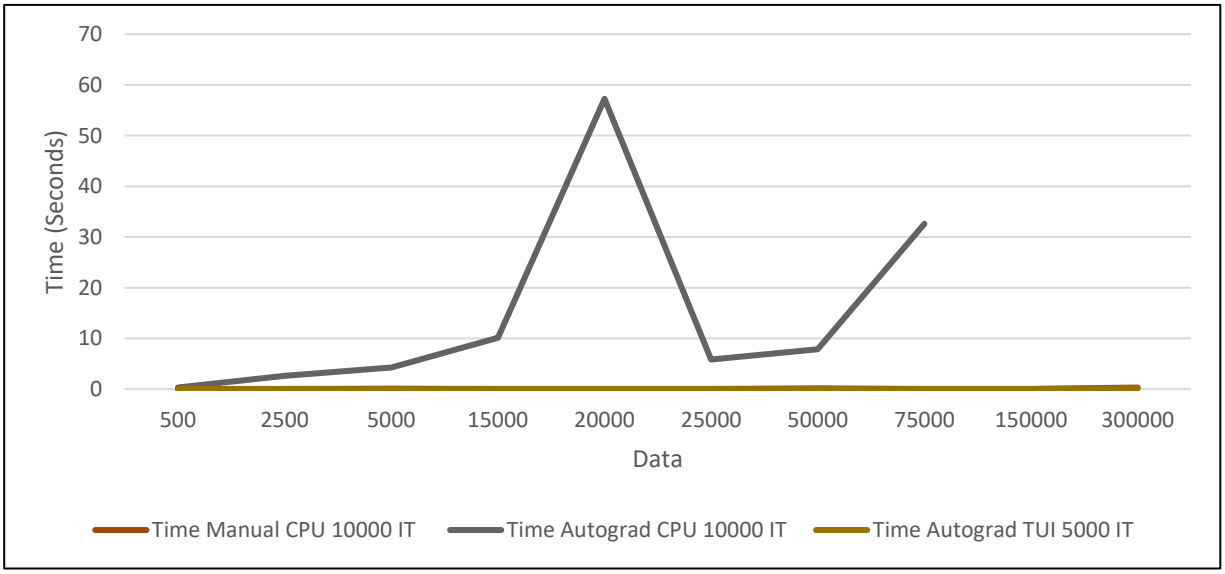

Figura 6.30: Gráfico varianza de tiempo dataset make_blobs 10000 iteraciones

6.5.3.2 Pruebas dataset Aleatorio

Dataset	N° Datos	Dimensiones	Clases	Tempo Manual CPU	Tiempo Autograd CPU	Tiempo Autograd TUI
Aleatorio	1000	10	3	0,002042864	0,003	0,001641167
Aleatorio	1500	15	4	0,001869513	0,0225	0,002022067
Aleatorio	3000	30	9	0,005005557	0,0545	0,004523739
Aleatorio	3500	40	12	0,007019426	0,0425	0,006029854
Aleatorio	5000	50	15	0,001751943	0,0455	0,009532317
Aleatorio	8000	80	18	0,036630239	0,0955	0,02007631
Aleatorio	10000	100	30	0,011367875	0,168	0,03863327
Aleatorio	12000	115	40	0,017464376	0,215	0,061037169
Aleatorio	15000	130	50	0,02776906	0,3005	0,095812104
Aleatorio	30000	150	90	0,095845901	0,9665	0,340513134
Aleatorio	35000	300	120	0,163452579	1,519	0,523768008
Aleatorio	50000	500	150	0,604538751	2,911	0,979048331
Aleatorio	80000	800	225	1,619912675	7,1975	3,260621168
Aleatorio	100000	1000	300	2,572261557	11,5625	6,227763806
Aleatorio	120000	1250	400	10,09214778	25,87	29,55636443
Aleatorio	150000	1500	450	35,89702278	49,29179373	38,15677896
Aleatorio	300000	3000	600	53,48888864	Memory Error	85,7125
Aleatorio	350000	4000	750	Memory Error	Memory Error	150,1165
Aleatorio	400000	4000	775	Memory Error	Memory Error	211,638
Aleatorio	500000	4000	800	Memory Error	Memory Error	454,0266667

Tabla 6.26: Tiempo medio dataset random

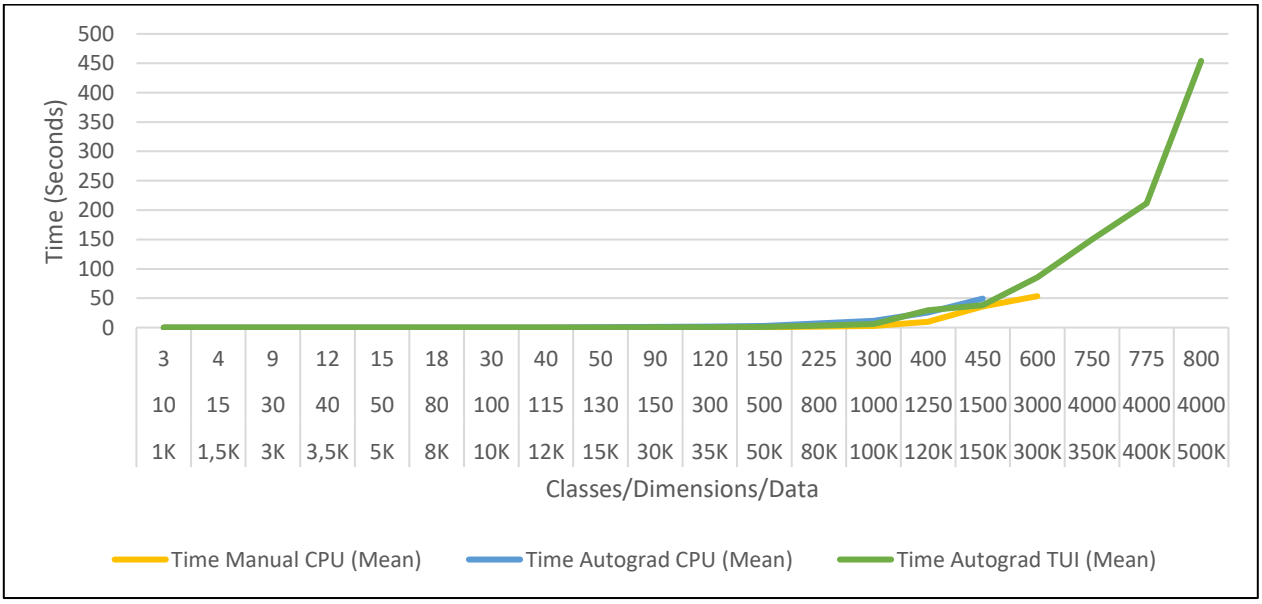

Figura 6.31: Gráfico tiempo medio dataset aleatorio

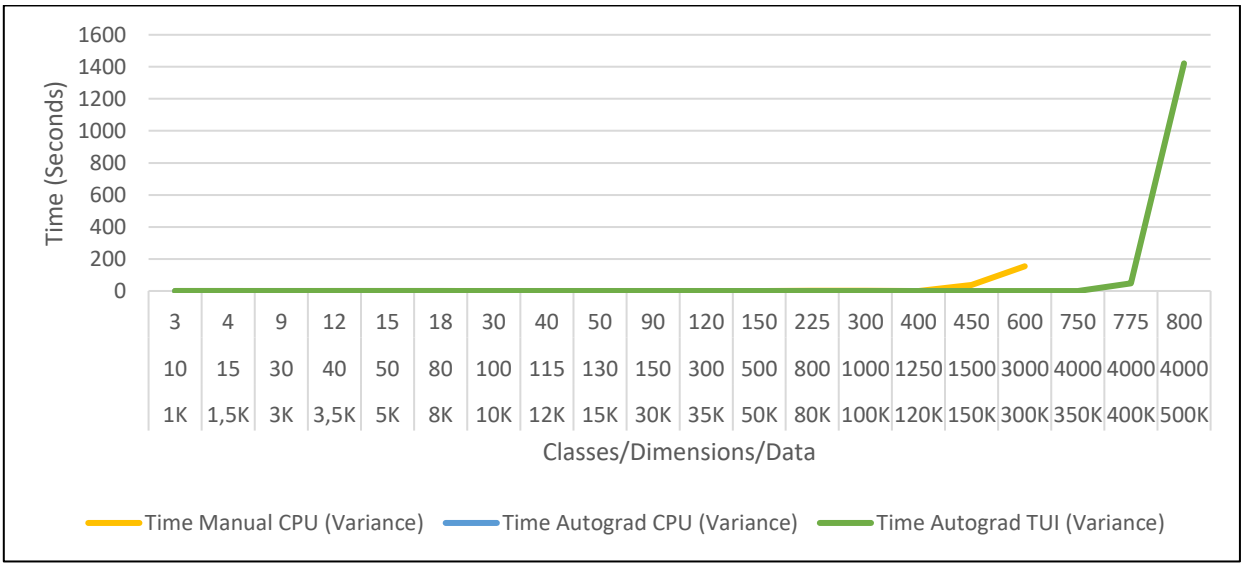

Figura 6.32: Gráfico varianza de tiempo dataset aleatorio

En esta sección del experimento, se pueden apreciar los resultados de los datasets de alta dimensionalidad, los cuales respecto a la tendencia de los resultados anteriores no hay cambios significativos. Esto claramente sostenido por la brecha que existe en el cálculo manual del gradientes respecto a la diferenciación automática en CPU. Si se visualizan las figuras 6.25 6.27 y 6.29 del dataset make blobs se puede observar que la inestabilidad vuelve, dado que la parametrización no se realizó de una manera uniforme, quedando de forma fija tanto las clases como la dimensionalidad. En cambio, si se visualiza los resultados del dataset aleatorio, en la figura 6.31 se puede ver claramente que los mejores resultados se obtuvieron para este caso. Donde se utilizó una parametrización gradual y uniforme.

Dadas las características a nivel de servidor se pudieron obtener resultados a un nivel mayor de muestras llegando hasta 500.000 datos, sin embargo, en CPU, Autograd volvió a arrojar errores de memoria a en los niveles más altos del experimento.

Tras los resultados obtenidos de las pruebas del experimentales, podemos obtener el siguiente análisis:

- Al comparar las diferentes técnicas de diferenciación, así como las herramientas de diferenciación automática de forma aislada. Independiente la derivada o función respecto a la exactitud no se encontraron diferencias significativas. Casos aislados son los métodos de diferencias finitas, donde el error solo será significativo en el caso de que este sea acumulado. Por ejemplo, en un algoritmo de características iterativas. En cambio, de la diferenciación simbólica toma un objetivo completamente diferente al de las técnicas expuestas. Si consideramos el tiempo de ejecución como medida, se pueden encontrar diferencias mas significativas donde el método de diferencias finitas tiene un mejor desempeño que una técnica de diferenciación automática, por lo que se deberá escoger cual método se acerca de mejor forma a lo que se necesita. Donde se tendrá que escoger entre la velocidad o exactitud.

- Si se compara el desempeño de Autograd versus un gradiente implementado de forma manual, claramente la brecha entre estos resultados es importante. Sin embargo, se deben considerar detalles a partir del gradiente manual, dado que su construcción se basa en su mayoría a una solución de carácter más analítico. Lo que conlleva en la práctica a una evaluación más directa reduciendo el consumo excesivo de memoria. En cambio, el trabajo que realiza Autograd es más elaborado, dado que realiza la diferenciación a partir de una función definida de manera estándar para este tipo de problemas. Dicho procesamiento tiene un tiempo asociado, lo cual genera esta diferencia importante a medida que aumentan las iteraciones, la cantidad de muestras, las clases o las dimensiones. Otro detalle importante para considerar es el lenguaje Python propiamente tal, dado que Python es un lenguaje del tipo interpretado, donde la principal desventaja de estos lenguajes es el tiempo que necesitan para ser interpretados. Al tener que ser

traducido a lenguaje máquina con cada ejecución, este proceso es más lento que en los lenguajes compilados, realizando un gran consumo de memoria[67].

- Si se considera esta última idea como el principal inconveniente de Autograd, esto puede mitigarse tras el uso de un hardware de mejores prestaciones respecto a CPU y memoria. Lo cual quedo comprobado tras el uso del servidor donde se pudo acercar e incluso igualar los resultados del gradiente manual durante algunos segmentos de la experimentación. Además, con el uso del servidor se consiguió el procesamiento de una mayor cantidad de muestras, y eliminar el inconveniente de error de memoria que se obtenía en CPU.

- La parametrización es un tema importante para tener en cuenta si se requieren obtener mejores resultados. Un claro ejemplo de esto se muestra en los dataset de dimensionalidad media, donde la brecha entre el gradiente implementado de forma manual y Autograd fue de una menor magnitud. En cambio, si se consideran los dataset tanto de baja dimensionalidad como el dataset make blobs en el caso de alta dimensionalidad, al tener una gran cantidad de parámetros fijos, afecta el tiempo efectivo de ejecución.

- Si bien Autograd, es una herramienta poderosa para el cálculo de derivadas o gradientes, dados estos resultados, carece en cierta forma de una optimización en el alto desempeño. Esto posiblemente también está asociado en parte a la customización de librerías nativas de Python como es el caso de Numpy, lo que hace que su desempeño sea más lento. Pero esto como se mencionó anteriormente puede ser mitigado por el uso de un hardware de mejores prestaciones o una optimización en GPU propiamente tal. Esto último puede establecerse abierto como un trabajo a futuro.

CAPÍTULO 7. CONCLUSIONES

Hoy en día, el cálculo de derivadas, gradientes, jacobianos o hessianos son un componente crucial en muchas áreas de la ciencia y la ingeniería, y su evaluación precisa a menudo se requiere en diversas aplicaciones ya sea de carácter científico o ingeniería. Sin embargo, con demasiada frecuencia en el área de la programación el cálculo del gradiente se realiza a partir de una implementación de forma manual. Lo que conlleva a que las expresiones del gradiente que se construyen de esta manera sean propensas a errores, lo que dificulta principalmente la portabilidad del código sí las especificaciones del problema son modificadas.

Es por esto, que a diferencia de un procedimiento manual, la transmisión de la información siempre debe apuntar a ser de alguna manera automatizada, si es que se buscan mejores resultados. Una idea a esto sería que, dada una función codificada y expresada como un programa de computadora, pudiera tener acceso a su gradiente sin requerir un gran esfuerzo de manera adicional. Basados en esta idea actualmente existen herramientas para realizar este trabajo denominándose diferenciación automática [68].

Uno de los principales fundamentos de este proyecto de tesis, ha sido implementar y medir la técnica de la diferenciación automática en problemas asociados al área del machine learning. Mediante la diferenciación automática es posible aplicar la regla de la cadena a secuencias de operaciones elementales, tales como sumas, multiplicaciones, funciones trigonométricas, entre otras, representadas en un programa escrito en un lenguaje determinado, siendo en este caso Python. Para de esta manera obtener valores precisos de la forma de diferenciación que se necesite.

A medida que se fueron tocando los tópicos de las diferentes técnicas de diferenciación, se pudo evidenciar las tendencias de cada método. Si bien conseguían el objetivo de entregar un resultado ya fuera simbólico o numérico, la velocidad y precisión eran medidas de carácter divergente. Por lo que se debe escoger una técnica respecto a las necesidades según la problemática a resolver.

Para los experimentos de esta tesis se escogió Autograd, como herramienta de diferenciación automática dado que, para su implementación, sólo bastaba incorporar algunas

líneas de código, teniendo así una nueva clase a partir de la cual con la simple invocación de un método se tendrá el gradiente de la función dada. Esto resulta de gran importancia, puesto que este es un dato indispensable para la ejecución de la mayoría de los métodos de optimización. Acercando así nuevos desafíos de carácter más avanzado. Un detalle importante a considerar es que Autograd aclara muchas ideas en el aprendizaje el uso de las herramientas de diferenciación automática y su implementación.

Ahora bien, respecto a las pruebas realizadas si bien Autograd no logró numéricamente los mejores resultados, siendo la dimensionalidad, la parametrización de los datos o inclusive el propio lenguaje Python como sus principales inconvenientes. Es un buen comienzo, ya que, al ser sometidos a pruebas en un hardware de una mejor arquitectura, se lograron resultados aceptables y comparables a una implementación del gradiente de forma manual. Y como se mencionó anteriormente el horizonte de estas técnicas es la optimización. Siguiendo con esta idea existen hoy en día implementaciones de diferenciación automática fuertemente inspiradas en Autograd, siendo Pytorch la más conocida. La cual se centra principalmente en su uso en GPU. Un detalle importante a considerar es el gran apoyo que ofrece autodiff.org [69] a la comunidad difundiendo una gran cantidad de herramientas de DA en diferentes tipos de lenguajes. Además de compartir material como papers, artículos o libros relacionados con el tema.

En líneas generales, el uso de la diferenciación automática como técnica de optimización representa un aporte significativo dentro del área de machine learning y abre un campo de investigación acerca de los diferentes mecanismos que pudieran implementarse para aprovechar al máximo las ventajas proporcionadas por este método de diferenciación.

REFERENCIAS

[1] Kotsiantis, S. B., Zaharakis, I., & Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. Emerging artificial intelligence applications in computer engineering, 160, 3-24.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016, November). Tensorflow: a system for large-scale machine learning. In OSDI (Vol. 16, pp. 265-283).

[3] Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., ... & Bengio, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. arXiv preprint.

[4] Maclaurin, D., Duvenaud, D., & Adams, R. P. (2015). Autograd: Effortless gradients in numpy. In ICML 2015 AutoML Workshop.

[5] Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.). (2013). Machine learning: An artificial intelligence approach. Springer Science & Business Media.

[6] Nasrabadi, N. M. (2007). Pattern recognition and machine learning. Journal of electronic imaging, 16(4), 049901.

[7] Funahashi, K. I. (1989). On the approximate realization of continuous mappings by neural networks. Neural networks, 2(3), 183-192.

[8] Dietterich, T. G. (2000, June). Ensemble methods in machine learning. In International workshop on multiple classifier systems(pp. 1-15). Springer, Berlin, Heidelberg.

[9] Malaspina, U. (2007). Intuición, rigor y resolución de problemas de optimización. Revista latinoamericana de investigación en matemática educativa, 10(3), 365-399.

[10] Griewank, A. (1989). On automatic differentiation. Mathematical Programming: recent developments and applications, 6(6), 83-107.

[11] Piskunov, N., & Medkov, K. P. (1983). Cálculo diferencial e integral (Vol. 1). Mir.

[12] Chapra S. y Canale R. (2008). "Métodos Numéricos para ingenieros". Tercera edición. Me Graw Hill

[13] Nolan, J. F. (1953). Analytical differentiation on a digital computer (Doctoral dissertation, Massachusetts Institute of Technology).

[14] Milne, W. E., & Milne, W. E. (1953). Numerical solution of differential equations (Vol. 19, No. 3). New York: Wiley.

[15] Alexandrom and M. Hussaini Edts, Multidisciplinary Design Optimization,SIAM, Philadelphia, 1997.

[16] Nikhil Ketkar ,Bangalore, Karnataka, Deep Learning with Python: A Hands-on Introduction, 2017

[17] N. Alexandrom and M. Hussaini Edts, Multidisciplinary Design Optimization,SIAM, Philadelphia, 1997.

[18] Chonacky, N., & Winch, D. (2005). 3Ms for instruction: Reviews of Maple, Mathematica, and Matlab. Computing in Science & Engineering, 7(3), 7-13.

[19] A. Griewank, On Automatic Differentiation,Technical Report, Argonne National Laboratory, Illinois, 1988.

[20] Khan, K. A., & Barton, P. I. (2015). A vector forward mode of automatic differentiation for generalized derivative evaluation. Optimization Methods and Software, 30(6), 1185-1212.

[21] Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in C++. arXiv preprint arXiv:1509.07164.

[22] Rall, L. B., & Corliss, G. F. (1996). An introduction to automatic differentiation. Computational Differentiation: Techniques, Applications, and Tools, 89.

[23] C. Faure, ”An introduction to automatic adjoint code generation, In C. Faure Edts,Automatic Differentiation for adjoint code generation, Rapport de recherche, No. 3555, INRIA, 1998

[24] L. Rall, Perspectives on automatic dierentiation: Past, present, ¿and future?, in: M. Bücker, G. Corliss, U. Naumann, P. Hovland, B. Norris (Eds.), Automatic Dierentiation: Applications, Theory, and Implementations, Vol. 50 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2006, pp. 1–14.

[25] Corliss, G. F., & Griewank, A. (1993). Operator overloading as an enabling technology for automatic differentiation (No. ANL/MCS/CP--79481). Argonne National Lab.

[26] Jiménez, D. G. Aplicaciones de la Diferenciación Automática en Ingeniería Mecánica: Simulación. Libros de David Gómez.

[27] Benkner, S., Chapman, B. M., & Zima, H. P. (1992, April). Vienna fortran 90. In 1992 Proceedings Scalable High Performance Computing Conference (pp. 51-59). IEEE.

[28] Wolfe, M. I., Babich, W., Simpson, R., Thall, R., & Weissman, L. (1981). The Ada Language System. Computer, (6), 37-45.

[29] Eckel, B. (2000). Thinking in C++. Prentice-Hall.

[30] Ascher, D., & Lutz, M. (1999). Learning Python. O'Reilly.

[31] Callejo Goena, A., García de Jalón de la Fuente, F. J., & Hidalgo, A. F. (2010). Diferenciación automática de fuerzas en la integración implícita de sistemas multicuerpo.

[32] Bücker, H. M., Corliss, G., Hovland, P., Naumann, U., & Norris, B. (Eds.). (2006). Automatic differentiation: applications, theory, and implementations (Vol. 50). Springer Science & Business Media.

[33] McKinney, W. (2012). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. " O'Reilly Media, Inc.".

[34] Scipy.org (2018) scipy.misc. Recuperado de https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html

[35] Meurer, A., Smith, C. P., Paprocki, M., Certík, O., Kirpichev, S. B., Rocklin, M., ... & Rathnayake, T. (2017). SymPy: symbolic computing in Python. PeerJ Computer Science, 3, e103.

[36] HIPS/Autograd (2017). Recuperdo de https://github.com/HIPS/autograd

[37] Maclaurin, D. (2016). Modeling, inference and optimization with composable differentiable procedures (Doctoral dissertation).

[38] Nguyen, V. Optimized R and Python: standard BLAS vs. ATLAS vs. OpenBLAS vs. MKL. Super Nerdy Cool.

[39] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. Proceedings of the IEEE, 96(5), 879-899.

[40] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... & Lerer, A. (2017). Automatic differentiation in pytorch.

[41] Walter, S. F., & Lehmann, L. (2013). Algorithmic differentiation in Python with AlgoPy. Journal of Computational Science, 4(5), 334-344.

[42] Andersson, J., Ákesson, J., & Diehl, M. (2012). CasADi: A symbolic package for automatic differentiation and optimal control. In Recent advances in algorithmic differentiation (pp. 297-307). Springer, Berlin, Heidelberg.

[43] Brodtkorb, P. A., & D'Errico, J. (2018). numdifftools Documentation.

[44] ad (2018): a Python package for first- and second-order automatic differentation, Recuperado de http://pythonhosted.org/ad/

[45] Raschka, S. (2015). Python machine learning. Packt Publishing Ltd.

[46] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., ... & Layton, R. (2013). API design for machine learning software: experiences from the scikit-learn project. arXiv preprint arXiv:1309.0238.

[47] Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. The Journal of Machine Learning Research, 18(1), 559-563.

[48] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository. Recuperado de http://archive.ics.uci.edu/ml/index.php

[49] Hosmer, D. W., Hosmer, T., Le Cessie, S., & Lemeshow, S. (1997). A comparison of goodness-of-fit tests for the logistic regression model. Statistics in medicine, 16(9), 965-980.

[50] Dreiseitl, S., & Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: a methodology review. Journal of biomedical informatics, 35(5-6), 352-359.

[51] Niculescu-Mizil, A., & Caruana, R. (2005, August). Predicting good probabilities with supervised learning. In Proceedings of the 22nd international conference on Machine learning (pp. 625-632). ACM.

[52] Jordan, M. I. (1995). Why the logistic function? A tutorial discussion on probabilities and neural networks.

[53] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., ... & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Advances in Neural Information Processing Systems (pp. 3981-3989).

[54] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[55] Anaconda (2018). Anaconda distribution. Recuperado de https://www.anaconda.com/download/

[56] Narayanan, S. H. K., Hovland, P., Kulshreshtha, K., Nagarkar, D., MacIntyre, K., Wagner, R., & Fu, D. (2017). Comparison of two gradient computation methods in Python

[57] Mishra, S. K. (2006). Some new test functions for global optimization and performance of repulsive particle swarm method.

[58] Schaffer, J. David (1984). Some experiments in machine learning using vector evaluated genetic algorithms (artificial intelligence, optimization, adaptation, pattern recognition)(PhD). Vanderbilt University. OCLC 20004572

[59] Darken, L. S. (1950). Application of the Gibbs-Duhem equation to ternary and multicomponent systems. Journal of the American Chemical Society, 72(7), 2909-2914.

[60] Murnaghan, F. D. (1944). The compressibility of media under extreme pressures. Proceedings of the National Academy of Sciences, 30(9), 244-247.

[61] scikit-learn.org (2018) The Iris dataset. Recuperado de https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

[62] scikit-learn.org (2018) sklearn.datasets.load_breast_cancer. Recuperado de https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html

[63] scikit-learn.org (2018) sklearn.datasets.load_wine. Recuperado de https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html

[64] scikit-learn.org (2018) sklearn.datasets.make_moons. Recuperado de https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html

[65] scikit-learn.org (2018) sklearn.datasets.make_circles. Recuperado de https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html

[66] scikit-learn.org (2018) sklearn.datasets.make_blobs. Recuperado de https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

[67] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2017). Automatic differentiation in machine learning: a survey. Journal of Machine Learning Research, 18(153), 1-153.

[68] Deep Learning with Python: A Hands-on Introduction Nikhil Ketkar Bangalore, Karnataka (2017), India

[69] Autodiff.org (2018). Recuperado de http://www.autodiff.org