UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Red neuronal para resolver una ecuación diferencial aplicada a un modelo de canal abierto

Jose Ignacio Palacios García

Matemáticas

Trabajo de fin de carrera presentado como requisito

para la obtención del título de

Matemático

Quito, 3 de diciembre de 2023

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN

DE TRABAJO DE FIN DE CARRERA

Red neuronal para resolver una ecuación diferencial aplicada a un modelo de canal abierto

Jose Ignacio Palacios García

Julio César Ibarra Fiallo, Ph.D. (c) .....

Luis Servando Espín Torres, Ph.D. (c) .....

Antonio Di Teodoro, Ph.D. .....

Quito, 3 de diciembre de 2023

© DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en la Ley Orgánica de Educación Superior del Ecuador.

Nombres y apellidos:	Jose Ignacio Palacios García
Código:	00213682
C.I.:	1003474101
Fecha:	Quito, 3 de diciembre de 2023

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en `http://bit.ly/COPETheses`

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on `http://bit.ly/COPETheses`

RESUMEN

En este trabajo se explica e implementa un método que usa una red neuronal artificial para resolver ecuaciones diferenciales numéricamente y se la aplica a un modelo del flujo de agua en un canal abierto descrito por las ecuaciones de Saint-Venant, un sistema de ecuaciones diferenciales parciales. La implementación se realizó en Python usando las librerías Pytorch y Numpy para manejo de matrices y construcción de la red neuronal artificial. Se comparó el resultado con el de un método numérico común usando RK1 y se obtuvo un error relativo promedio de $4{,}05\,\%$. Los resultados obtenidos muestran que el método propuesto presenta un desempeño prometedor en la resolución de ecuaciones diferenciales parciales, sobre todo por la versatilidad que presenta para definir condiciones de contorno en geometrías complejas. El tiempo de ejecución, usando las optimizaciones computacionales desarrolladas para las redes neuronales artificiales, es similar al de métodos tradicionales. Se mencionan posibles mejoras para investigación a futuro.

***Palabras clave*:** Redes Neuronales, Ecuaciones Diferenciales Parciales, Gradiente Descendiente, Métodos Numéricos para Ecuaciones Diferenciales Parciales.

ABSTRACT

In this work we explain and implement a method that uses an artificial neural network to solve differential equations numerically. The method was applied to a model of the flow of water in an open channel described by the Saint-Venant Equations (SVE). These equations constitute a system of partial differential equations. The method was implemented in Python using the libraries Numpy and Pytorch to manage matrix operations and the construction of the artificial neural network. The results of the method were compared with a common numerical method using RK1, where an average relative error of $4{,}05\,\%$ was obtained. The results show that the proposed method has a promising performance in the resolution of partial differential equations, especially because of the versatility that it offers to define boundary conditions in complex geometries. The execution time was comparable to traditional methods, thanks to common performance enhancements developed for training artificial neural networks. Possible improvements for further research are mentioned.

***Keywords*:** Neural Networks, Partial Differential Equations, Gradient Descent, Numerical Methods for Partial Differential Equations.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

DEDICATORIA

Para mi tía Bertha y para mi madre.

AGRADECIMIENTO

Quiero agradecer a mis tutores Julio y Servando por su apoyo en este trabajo. Gracias a mi familia y amigos por aguantarme estos años y gracias a mi mascota, Rebeca, por ayudarme a entender cálculo vectorial.

CAPÍTULO 1

INTRODUCCIÓN

En la última década, se ha notado un aumento considerable en la popularidad de diferentes algoritmos de inteligencia artificial y redes neuronales. Entre estos algoritmos, el que ha tenido aplicaciones más notables es el de las redes neuronales artificiales; estas representan una abstracción del funcionamiento teórico de las neuronas en organismos vivos (Veelenturf, 1995). La exploración de estos modelos sigue siendo un punto de interés para la investigación, tanto en el campo de la optimización desde la teoría (Kingma and Ba, 2017) como en aplicaciones para predicción y reconocimiento de patrones.

Por otro lado, los métodos de solución de sistemas de ecuaciones diferenciales complejos son un punto clave de investigación en la mayoría de ciencias e ingenierías. Sin embargo, los métodos analíticos de resolución de ecuaciones diferenciales son relativamente limitados y complicados; y en el caso de las ecuaciones diferenciales parciales, suele ser más eficiente usar métodos numéricos que buscar una solución exacta. Es por eso que la investigación en métodos de resolución numérica de estos sistemas es de gran importancia.

En este trabajo se va a describir e implementar un método para resolver ecuaciones diferenciales de distintos tipos que aprovecha los avances y la arquitectura diseñados para la implementación de redes neuronales artificiales. Este método fue descrito en detalle en (Neha Yadav, 2015) y en (Chen et al., 2020). Para demostrar el funcionamiento, se lo aplicará a un sistema de ecuaciones diferenciales que describen el flujo de agua en un canal abierto, que han sido resueltas numéricamente (Fauzi and Wiryanto, 2018). Finalmente, se comparará los resultados con un método numérico simple similar al usado en (Sukron et al., 2021) y se discutirá las ventajas y limitaciones del método.

CAPÍTULO 2

DEFINICIONES Y MÉTODOS

2.1. Algoritmo del gradiente descendiente

El gradiente descendiente es un método numérico diseñado para resolver el problema de optimización de encontrar el mínimo de una función diferenciable. El algoritmo está basado en la idea conocida de que se puede obtener la dirección en la que una función diferenciable aumenta más rápidamente si se toma el vector ordenado de todas las derivadas parciales con respecto a cada una de las bases del espacio vectorial (usualmente $\mathbb{R}^n$). Dada esa observación, el algoritmo es casi trivial, pero se le atribuye a (Cauchy, 1847), que lo describió formalmente en 1847. Claramente, el método no fue usable hasta que las computadoras se volvieron populares, pero la convergencia del método ya había sido estudiada por (Curry, 1944).

El algoritmo, como fue descrito por (Curry, 1944), puede ser resumido de la siguiente forma:

Algorithm 1 Gradiente Descendiente

Input: punto inicial x_0, función $f : \mathbb{R}^n \to \mathbb{R}$, gradiente ∇f, tasa de aprendizaje γ, número de iteraciones K.

Output: Secuencia $(x_k)_{k\in\{1,\dots,K\}}$ donde x_K es un valor aproximado al mínimo local de la función f

$i \leftarrow 1$
while $i \leq K$ **do**
    $x_i \leftarrow x_{i-1} - \gamma \nabla f(x_{i-1})$
end while
return (x_k)

Dos cosas importantes se pueden notar en este algoritmo: la primera es que no se necesita una expresión exacta para la gradiente de f pues se puede aproximar usando una derivada numérica, pero en caso de existir la expresión para la gradiente, se puede usar. La segunda es que este método se queda *estancado* en puntos que tienen gradiente cero, idealmente estos puntos serían el mínimo local que se estaría buscando, pero también es posible que sea un máximo local o

un punto de silla de montar. Es por eso que en algunas aplicaciones se puede agregar un vector aleatorio muy pequeño si el gradiente es cero o usar un sistema de momentos. Si el punto es un mínimo local, el algoritmo va a regresar a ese punto en las siguientes iteraciones, pero si es un máximo o un punto de silla de montar, la pequeña aleatoriedad sería suficiente para obligar al algoritmo a encontrar otro punto crítico.

2.1.1. Notas sobre algoritmos similares

Existe un algoritmo parecido llamado coordenadas descendientes, que sigue la misma idea pero hace la descendencia de coordenada en coordenada. Este algoritmo es útil en aplicaciones donde la memoria es limitada y se tiene demasiadas dimensiones en el espacio de salida. Se menciona porque una implicación de la convergencia de este algoritmo es que se puede aplicar el gradiente descendiente como se describió en el Algoritmo 2, pero usando solo una porción de las dimensiones para computar el gradiente. Esto puede resultar útil en problemas en los que hay limitación de memoria. Más información sobre este algoritmo se puede encontrar en (Wright, 2015).

Varias modificaciones se han propuesto en los últimos años como *Averaging Gradient Descent*, que se puede encontrar en (Polyak and Juditsky, 1992) y métodos que incluyen momento. Sin embargo, el más usado en los recientemente ha sido el algoritmo **Adam**, publicado en 2015, que significa *Adaptive Moment Estimation*. Es un método altamente eficiente que recuerda la última dirección de movimiento y actualiza los pesos de los momentos de forma apropiada para mejorar la velocidad de convergencia. Es más útil en problemas que tienen una convergencia predecible y por eso se lo va a usar en este trabajo. La descripción del algoritmo en el artículo original, (Kingma and Ba, 2017), es la siguiente:

Algorithm 2 Adam

Input: α: Tamaño de paso, $\beta_1, \beta_2 \in [0, 1)$: Tasas de decaimiento, $f(\theta)$: función objetivo, θ_0: Vector de parámetros iniciales

Output: θ: Valor aproximado del vector de parámetros que minimiza la función objetivo.

$m_0 \leftarrow 0$

$v_0 \leftarrow 0$ (Son los vectores de momento)

$t \leftarrow 0$

while $i \leq K$ **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla f_t(\theta_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\hat{m}_t \leftarrow m_t / (1 - \beta_2^t)$

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$

end while

return (θ_t)

La explicación del algoritmo, así como un análisis de su convergencia, se puede encontrar en (Kingma and Ba, 2017). Como se puede ver, es un algoritmo mucho más elaborado que el de gradiente descendiente y el paper recomienda ciertos valores de β_1, β_2 así como el tamaño de paso y ε. En este trabajo se va a usar la implementación del paquete `pytorch`, que cita el paper mencionado y usa las constantes recomendadas.

2.2. Estructura básica de una red neuronal

Como ya se mencionó, el objetivo del modelo de las redes neuronales artificiales es emular el modelo teórico de como las neuronas funcionan en organismos vivos. En este modelo, las neuronas son objetos que deciden si van a estar activadas o no en base a los impulsos que han recibido, ver Figura 2.1. Si la neurona ya *sabe* lo que tiene que hacer, eso significa que

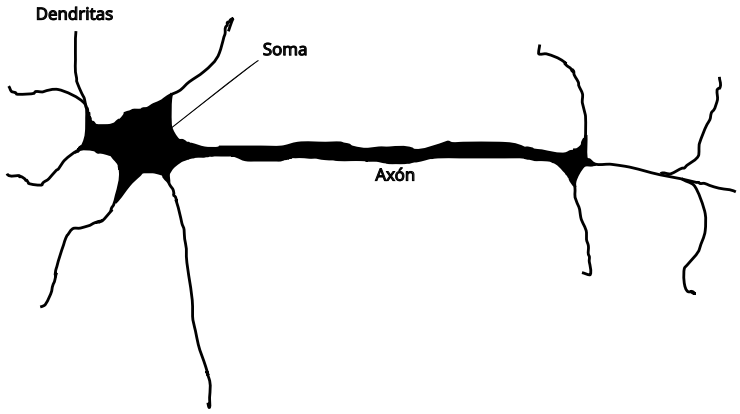

Figura 2.1: Modelo clásico de una neurona

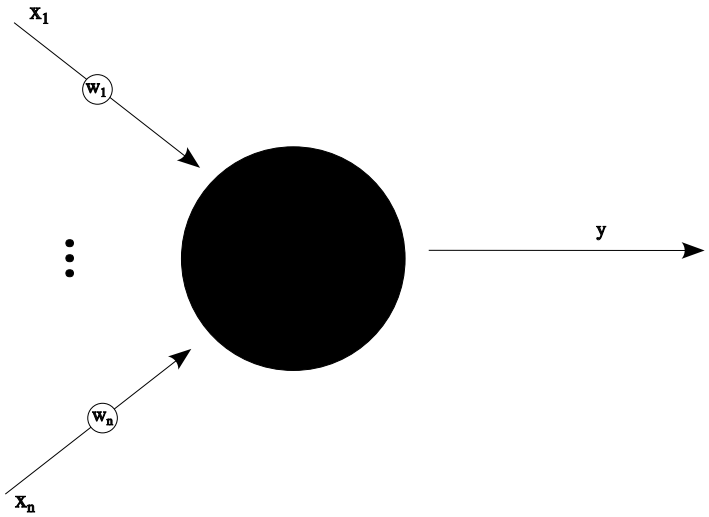

Figura 2.2: Diagrama de un perceptrón.

las conexiones que hay entre esta y otras neuronas cercanas tiene la fuerza apropiada en cada conexión como para que la neurona solamente se active cuanto tiene que activarse, desde este punto de vista, la neurona no es tan importante en el proceso de aprendizaje como las conexiones con otras neuronas (Veelenturf, 1995).

La abstracción de este modelo se llama perceptrón (ver Figura 2.2). Un perceptrón es la unidad básica de una red neuronal artificial y busca simular el modelo de la neurona. Este objeto tiene un numero fijo de conexiones con otras neuronas, en cada una de estas conexiones, la neurona anterior envía un número real (la salida respectiva) al perceptrón. Cuando se envía el

número real, el perceptrón lo multiplica por un peso (w_i en la figura 2.2), que representa la fuerza de la conexión en el modelo real, y luego suma el resultado de cada una de las multiplicaciones. Usando esta suma, el perceptrón decide el valor que va a enviar al siguiente perceptrón por su salida.

Usualmente estas neuronas artificiales se organizan por capas, de forma que las neuronas de una capa se relaciones sólamente con las neuronas de la siguiente capa.

Esto se puede abstraer aún más si se crea un vector w de pesos:

$$w = (w_0, ..., w_n)$$

y un vector X de impulsos de la capa anterior:

$$X = (x_0, ..., x_n)$$

entonces, la operación que se describió se puede representar como

$$f(w \cdot X)$$

donde $\cdot$ es el producto interno usual en $\mathbb{R}^n$ y f es la función que decide la respuesta del perceptrón a esa suma.

Para una capa entera, si se tiene k perceptrones, se puede reducir este grupo de operaciones a una multiplicación matricial:

$$f(XW)$$

donde W es la matriz formada por los vectores de peso de cada uno de los perceptrones de la capa. Hay que notar que al aplicar f en este caso, se está haciendo la operación para cada elemento de la matriz y no para el vector resultante.

La función f suele llamarse función de activación. Diferentes funciones de activación tienen ventajas en distintas aplicaciones, para nuestra aplicación, sólamente se necesita que esta función sea continuamente diferenciable y con derivada distinta de cero en todas partes.

Para decidir si la salida de un perceptrón es correcta, se usa una función de error. Se busca una función que de como resultado cero si la respuesta de la red es perfecta y que de como resultado un número positivo de lo contrario.

Dado esto, el problema de entrenar una red neuronal, se reduce a un problema de optimización en el que se quiere encontrar la matriz de parámetros W tal que el error sea mínimo. Esto se puede hacer con el algoritmo de gradiente descendiente, pero usualmente se escoge un algoritmo de optimización dependiendo de la aplicación.

CAPÍTULO 3

MÉTODO PROPUESTO

3.1. Descripción

El método que se va a usar tiene como objetivo explotar los mecanismos optimizados de entrenamiento de redes neuronales para encontrar una solución aproximada a una ecuación diferencial.

El primer paso es discretizar el dominio sobre el que se quiere resolver las ecuaciones diferenciales. El número de puntos en esta discretización tendrá un impacto importante sobre el tiempo de ejecución de los siguientes pasos. Dada la discretización, se crea una red neuronal de la siguiente forma:

1. Existe una sola capa de perceptrones.
2. Existe el mismo número de perceptrones como puntos en el dominio discretizado.
3. La función de activación de cada uno de los perceptrones no es acotada o las cotas son más amplias que las de la ecuación diferencial.
4. La red neuronal acepta una sola entrada.
5. El valor de salida de cada uno de los perceptrones será el valor de salida de la red neuronal.

Una vez creada la red neuronal, se debe crear una función de error que sea tal que una vez entrenada la red, lo que la misma devuelva sea una solución para la ecuación diferencial que se quiere resolver.

Dada una ecuación diferencial de grado g de la forma

$$F\left(x, \frac{dy}{dx}, ..., \frac{d^g y}{dx^g}\right) = 0 \tag{3.1}$$

y que con la discretización que se creó, el dominio es $X = \{x_0, ..., x_n\}$. Una función de error positiva que, de dar como resultado cero, garantizaría el cumplimiento de la ecuación diferencial sería:

$$E(x_0, ..., x_n) = \sum_{i=0}^{n} F\left(x_i, \frac{dy}{dx}(x_i), ..., \frac{d^g y}{dx^g}(x_i)\right)^2$$

Usar E como función de error en esta red neuronal donde X es la salida de la red, asegura que el resultado, de llegar a ser optimizado, va a cumplir con la ecuación diferencial. Sin embargo, para aplicaciones, no basta con encontrar *una* solución a la ecuación diferencial, sino que es necesario encontrar una solución que al mismo tiempo cumpla con ciertas condiciones iniciales o de frontera.

Ese problema se puede resolver usando de forma apropiada la salida de la red neuronal como en (Neha Yadav, 2015). Sea $N_0, ..., N_n$ los valores de salida de la red neuronal en cada uno e los perceptrones. Si A es una función C^g definida en el dominio de la ecuación diferencial que cumple con las condiciones de frontera del problema, pero no necesariamente cumple con la ecuación diferencial. También se asume que f es una C^g con el mismo dominio que es cero en la frontera, pero que es distinta de cero en el resto del dominio. Se define entonces

$$y_T(x_i) = A(x_i) + f(x_i, N_i) \tag{3.2}$$

esta será la función con la que se va a calcular la función de error. De esa forma, cuando el algoritmo encuentre el mínimo, se sabrá que se cumple la ecuación diferencial y también las condiciones de frontera porque la red neuronal es incapaz de alterar los valores en la frontera por la forma en la que se construyó f.

Hay que notar que este método se podría extender a más dimensiones simplemente exten-

diendo la discretización y la red neuronal. También es importante recordar que al ser un método numérico, se tendría que reemplazar las derivadas por derivadas numéricas.

Este es un método en desarrollo y, aunque ha sido descrito por varias fuentes, es importante experimentar para obtener los resultados deseados.

Actualmente está en desarrollo un paquete de Python en código abierto que se puede usar para resolver ecuaciones diferenciales ordinarias de primer orden con valor inicial y está descrito en (Chen et al., 2020). Sin embargo, para ecuaciones con condiciones de contorno más complicadas se tiene que implementar el método completo como está descrito aquí.

3.2. Un ejemplo simple

Para probar la eficacia del método, se tomará la ecuación diferencial

$$\frac{\mathrm{d}}{dx}y(x) = y(x) \tag{3.3}$$

con valor inicial $y(0) = 1$. Como se sabe, la solución exacta de esta ecuación diferencial es una exponencial simple: $y(x) = \exp(x)$ así que ese sería el resultado que se espera obtener. Se encontrará la solución aproximada sobre el dominio $[0, 1]$

Para la discretización, se usó una lista de 100 puntos $x_1, ..., x_{100}$ donde $x_1 = 0$, $x_{100} = 1$. Se define $N = w = (n_1, ..., n_{100})$, es la salida de la red neuronal con matriz de pesos w y función de activación identidad, la arquitectura se puede ver en la figura 3.1 . se va a a tomar $A(x) = \frac{1}{x+1}$, esta función cumple con $A(0) = 1$ y es diferenciable en todo el intervalo. como función f, se usa una función lineal

$$f(x_i, n_i) = x_i n_i$$

evidentemente esta función es diferenciable y es cero en la frontera ($f(0, n_0) = 0$). Entonces, se

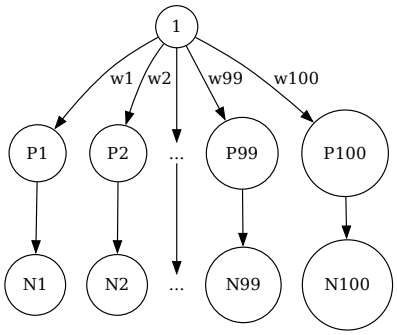

Figura 3.1: Arquitectura de la red neuronal

tiene que la función de prueba es:

$$y_T(x_i, n_i) = \frac{1}{x_i + 1} + x_i n_i$$

. Con eso, se puede definir la función de error de la siguiente manera:

$$E(N) = \frac{1}{100} \sum_{i=1}^{100} \left(\frac{\mathrm{d}}{\mathrm{d}x} y_T(x_i, n_i) - y_T(x_i, n_i) \right)^2$$

Como esta es una ecuación diferencial simple y que no tiene demasiados puntos, se puede hacer la implementación usando `numpy`, la librería de Python[1] para manejo de vectores y matrices.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 XN = 100
5
6 XL = np.linspace(0, 1, XN)
7 dx = XL[1] - XL[0]
8 NF = np.ones_like(XL)
9 AF = 1/(XL + 1)
10
11 FT = AF + XL * NF
12 DF = np.gradient(FT)
13
14
15 def error():
16     return np.sum((DF - FT) ** 2)
```

[1] Se usará Python3

En este ejemplo, `NF` es la matriz de pesos de la red neuronal, `AF` es la función A que se describió en (3.2). La función f estaría representada por la matriz `FT`, que se tendrá que recalcular en cada paso.

También se define la función de error como la suma de cuadrados de la derivada, que se obtiene usando la función `np.gradient`, menos los valores individuales de la función[2].

Es importante recordar que como este es en un espacio discreto, la derivada $\frac{\mathrm{d}}{\mathrm{d}x} y_T(x_i, n_i)$ se hace numéricamente de la siguiente forma según la documentación de numpy:

$$y'_T(x_i, n_i) = \begin{cases} \frac{y_T(x_{i+1}, n_{i+1}) - y_T(x_i, n_i)}{h} & \text{si } x = 1 \\ \frac{y_T(x_i, n_i) - y_T(x_{i-1}, n_{i-1})}{h} & \text{si } x = 100 \\ \frac{y_T(x_{i+1}, n_{i+1}) - y_T(x_{i-1}, n_{i-1})}{2h} & \text{en otro caso} \end{cases} \tag{3.4}$$

esto solo significa que se hace la aproximación de segundo orden si el punto está en el interior del dominio y la de primer orden si está en el borde.

Con eso, se aplica el algoritmo de gradiente descendiente tal y como fue descrito en 2:

```
1 iters = 1000000
2 lr = 0.00001
3 deltay = 0.000001
4 print(error())
5 for i in range(iters):
6     GradE = np.zeros_like(XL)
7     for i in range(XL.size):
8         FT = AF + XL * NF
9         DF = np.gradient(FT, dx)
10        e0 = error()
11        NF[i] = NF[i] + deltay
12        FT = AF + XL * NF
13        DF = np.gradient(FT, dx)
14        ef = error()
15        NF[i] = NF[i] - deltay
16        GradE[i] = (ef - e0)/deltay
17    NF = NF - lr * GradE
18 save()
```

[2]En numpy, cuando se eleva una matriz al cuadrado usando el operador $**$, en realidad se obtiene el cuadrado por elemento, no el cuadrado usando multiplicación matricial.

El algoritmo de gradiente descendiente está escrito explícitamente en este ejemplo. Primero se crea una matriz de ceros y luego se itera en cada dimensión calculando la derivada parcial numérica y agregándola a la matriz. Al final se da el paso del gradiente descendiente (`NF = NF - lr * GradE`) y se vuelve a empezar.

En este ejemplo se inicializa la red neuronal con todos los pesos en 1. Obviamente, qué tan lejos esté de la función solución va a cambiar el tiempo de convergencia en el entrenamiento. Luego de un millón de iteraciones del gradiente descendiente con una tasa de aprendizaje de $0{,}00001$, se obtuvo un resultado a una distancia de $0{,}002766817$ de la función exponencial (ver figura 3.2) (usando la métrica usual entre funciones para medir la distancia[3])[4].

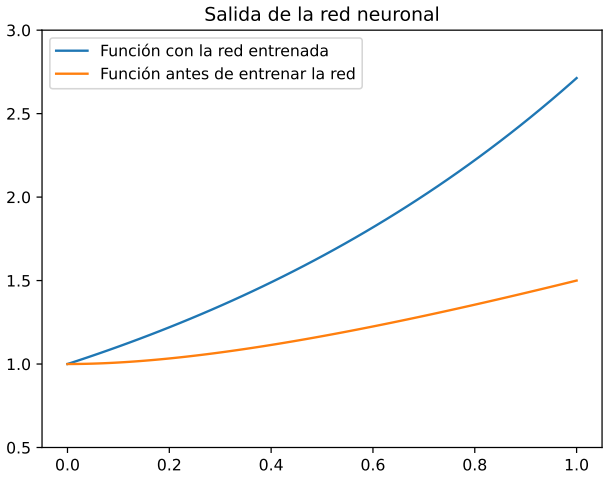

Figura 3.2: Solución obtenida con la red neuronal.

[3]Usando la misma discretización del dominio, se puede definir el vector $\hat{E} = (\exp(x_1), ..., \exp(x_{100}))$. Para medir la distancia se usa la fórmula

$$d(\hat{E}, y_Y(X)) = \frac{1}{100} \sum_{i=1}^{100} |E_i - y_Y(x_i)|$$

. También se puede pensar como la distancia promedio entre los puntos de las dos funciones

[4]Para dos funciones continuas, si se hace una discretización de k puntos, la métrica mencionada aumenta linealmente con el número de puntos en la discretización. Es por eso que una métrica más justa para calcular la precisión es d/k, donde d es la misma métrica mencionada antes.

Como ya se mencionó, esta implementación es relativamente lenta porque se hace el gradiente descendiente en Python, que es un lenguaje de programación interpretado. Es mucho más eficiente usar un paquete diseñado específicamente para redes neuronales. En el resto de este trabajo se va a usar el paquete `pytorch`, que tiene implementaciones mucho más avanzadas para computar el gradiente y hacer operaciones con vectores y matrices.

CAPÍTULO 4

PROBLEMA DE APLICACIÓN

4.1. Descripción del problema: Ecuaciones de Saint-Venant

Como ejemplo más concreto de el uso de este algoritmo, se simulará el flujo gradualmente variado en un canal abierto. Un canal abierto es un conducto por el que puede pasar un fluido, pero, a diferencia de los canales cerrados, se permite que el agua aumente en volumen. El ejemplo más fácil de comprender es el flujo de agua en un rio. Si el flujo aumenta considerablemente, la altura del agua sube y por lo tanto el volumen que ocupa también. Por otro lado, si el flujo es de canal cerrado, ese aumento de volúmen no está permitido y se tiene que transformar en un aumento de presión o de velocidad del fluido.

El modelo más apropiado para representar este fenómeno son las ecuaciones de *Saint-Venant* o ecuaciones de aguas poco profundas. Para este trabajo solamente se va a considerar el caso en una dimensión que fue propuesto originalmente por *Saint-Venant* en en paper "Théorie du mouvement non permanent des eaux, avec application aux crues des rivières et a l'introduction de marées dans leurs lits.[en] 1871. Una versión más moderna se puede encontrar en .ºpen-Channel Hydraulics"de Ven Te Chow. Las ecuaciónes de Saint-Venant (18.2 y 18.13 en (Chow, 1959, pp. 525-528)) se pueden escribir así:

$$\frac{\partial A}{\partial t} + \frac{\partial (Q)}{\partial x} = 0 \tag{4.1}$$

$$g\frac{\partial y}{\partial x} + u\frac{\partial u}{\partial x} + \frac{\partial u}{\partial t} + g(S_f - S_0) = 0 \tag{4.2}$$

donde $Q = Au$ es el flujo, u es la velocidad, A es el área del corte del canal en el punto x, x es la distancia, y es la altura del agua, t es el tiempo y g es la gravedad. S_0 y S_f son la pendiente de la

superficie y de la fricción respectivamente[1]. En este caso, por simplificación, se va a considerar S_f una constante, que se puede calcular usando la ecuación (7.5) de (Chow, 1959, p. 168):

$$S_f = \frac{\tau}{\rho g R} \tag{4.3}$$

donde R es A/P, y P es el perímetro del área que tiene contacto con la superficie del canal.

Se considera un canal rectangular donde la altura del agua es y y el ancho del canal es w. Entonces $A = wy$ y $P = w + 2y$. $R = w/(w + 2y)$. Entonces con $\tau = 1$ $S_f = \frac{w+2y}{y\rho g w}$. Si además se toma un canal plano, es decir que a lo largo del canal no haya cambios en la altura de la superficie, entonces $S_0 = 0$.

Una ilustración de este caso particular se puede encontrar en la figura 4.1. Como se puede

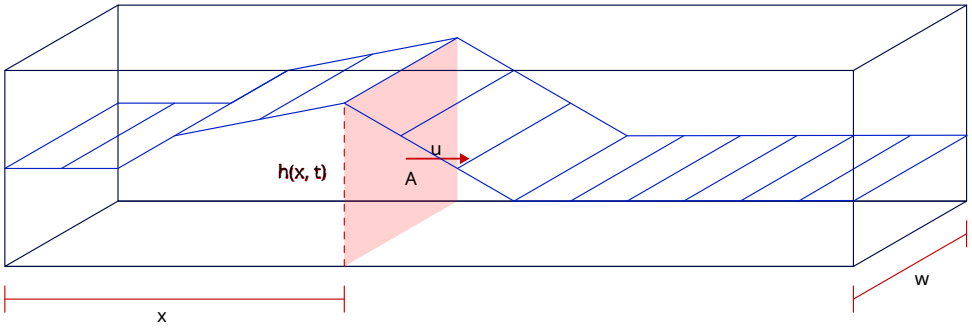

Figura 4.1: Ilustración de un canal abierto

ver, las perturbaciones en el agua del canal vienen de la acumulación de masa en la condición inicial.

El objetivo del resto de este trabajo será encontrar una solución aproximada a este sistema usando el método propuesto.

4.2. Método de resolución

Se seguirá el método descrito en la sección 3.1. Recordando que en la ecuación diferencial, ambas funciones (u y y) tienen dominios en espacios de dos dimensiones, x y t. Es decir que,

[1]En este modelo, la fricción actúa exactamente que una pendiente en el lado opuesto del movimiento.

para discretizar el dominio con K puntos por dimensión, se va a necesitar dos vectores de K^2 puntos, que se van a llamar U_T y H_T. Enumerando ambos vectores de 0 a K, se puede definir la función de error de la siguiente manera:

$$E(x_0, ..., x_n) = \sum_{i=0}^{n} \left[\left(\frac{\partial(y)}{\partial t} + \frac{\partial(yu)}{\partial x} \right)^2 + \left(g\frac{\partial y}{\partial x} + u\frac{\partial u}{\partial x} + \frac{\partial u}{\partial t} + g\frac{w+2y}{\rho g w y} \right)^2 \right] \tag{4.4}$$

Para el dominio se usará puntos $(t, x) \in [0, 4] \times [0, 12]$ definidos sobre el rectángulo. Como condiciones iniciales se tiene $u(0, x) = 0$ y

$$y(0, x) = \begin{cases} 5 & \text{si } x \in [0, 4] \cup [8, 12] \\ -(x-4)(x-8) + 5 & \text{en otro caso} \end{cases} \tag{4.5}$$

. Esta función es simplemente un ejemplo de condición inicial para este método. Estrictamente no es diferenciable en 4 y 8, pero como el mallado es relativamente grueso, las derivadas se pueden aproximar en esos puntos.

Entonces, una función válida para $A_u(t, x)$, como en la ecuación (3.2) sería $A_u(t, x) = 0$ en todo el dominio y para la función $A_y(t, x)$, se podría tomar $A_y(t, x) = y(0, x)$, pero por el momento se usará una función que decrezca en el tiempo para que la red neuronal tenga que ajustarse más y se note el funcionamiento del método. Se tomará entonces

$$A_y(t, x) = \begin{cases} 5 & \text{si } x \in [0, 4] \cup [8, 12] \\ -(x-4)(x-8) * \exp(-0{,}7t) + 5 & \text{en otro caso} \end{cases} \tag{4.6}$$

Como funciones f_u y f_y, se define $f_u(t, x) = tN_u(t, x)$ y $f_y(t, x) = tN_y(t, x)$ donde N_u y N_y son las salidas de la red neuronal en ese punto. De esa forma se cumple con todas las condiciones que pide el método. La programación de estas funciones usando `pytorch` sería la siguiente:

```
1  def A_h(i, j):
2      # j es a x, i es a t
3      x = X[i, j]
4      t = T[i, j]
5      if (0 <= x) and (x <= 4):
6          return 5
7      if (4 <= x) and (x <= 8):
8          return (- (x - 4) * (x - 8)) * (np.exp(-(0.7) * t)) + 5
9      if (8 <= x) and (x <= 12):
10         return 5
11     print("Out of bounds")
12 # los parmetros de la red neuronal son cada uno de los puntos de las
       matrices H y U
13 def f_h(i, j):
14     t = T[i, j]
15     return Nh[i, j] * t
16
17 def A_u(i, j):
18     return 0
19
20 def f_u(i, j):
21     x = X[i, j]
22     t = T[i, j]
23     return t * Nu[i, j]
24
25 AH = torch.Tensor([
26     [A_h(i, j) for j in range(Num)] for i in range(Num)
27 ])
28
29 AU = torch.Tensor([
30     [A_u(i, j) for j in range(Num)] for i in range(Num)
31 ])
```

Como las funciones A_h y A_u no cambian con el proceso de entrenamiento, se pueden calcular una sola vez al principio (líneas 25 y 29). También se puede definir la red neuronal como una clase:

```
1  class NeuralNetwork(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.flatten = nn.Flatten()
5          self.Layer10 = nn.Linear(1, Num * Num)
6          self.Layer11 = nn.Linear(1, Num * Num)
7          nn.init.zeros_(self.Layer10.weight)
8          nn.init.zeros_(self.Layer11.weight)
9
10     def forward(self, x):
11         x = self.flatten(x)
12         output1 = self.Layer10(x).reshape(Num, Num) / 100
13         output2 = self.Layer11(x).reshape(Num, Num) / 100
14         return output1, output2
```

En la primera parte de la clase, se define los objetos. En esta red neuronal se hizo una pequeña modificación al método original. en las líneas 5 y 6 se definen las dos capas de la red neuronal, sin embargo, estas dos capas se aplican paralelamente, es decir que no son secuenciales. Eso significa que, la red neuronal actua como si solo tuviera una capa con $2N^2$ puntos en vez de dos capas con N^2. También hay que notar que se inicializó todos los pesos de la red neuronal con ceros. Esto no es una práctica común, lo que se hace usualmente es iniciar con valores aleatorios, pero en este caso, los valores aleatorios crean derivadas grandes y obligan a disminuir demasiado la tasa de aprendizaje, que hace más lenta la convergencia del método.

El método `forward` es la función de evaluación de la red neuronal. Como se puede ver que toma un solo número `x`, que en este caso siempre es 1 y lo multiplica por ambas capas individualmente. Luego deuelve la salida de esas dos capas.

Se define la función de error justo como en la ecuación 4.4. Para aprovechar la paralelización del paquete `torch`, se debe usar funciones del paquete en la medida de lo posible. Afortunadamente, el paquete incluye funciones para computar gradientes y derivadas parciales:

```
1  def criterion(Nh, Nu):
2      H = AH + T * Nh
3      U = AU + T * Nu
4      HU = H * U
5      Dht, Dhx = torch.gradient(H)
6      Dht = Dht * 1/dt
7      Dhx = Dhx * 1/dx
8      Dut, Dux = torch.gradient(U)
9      Dut = Dut * 1/dt
10     Dux = Dux * 1/dx
11     _, Dhux = torch.gradient(HU)
12     Dhux = Dhux / dx
13     E = (Dht + Dhux) ** 2 + (Dut + U * Dux + g * Dhx + g * \
14                              (1 * (r * g * H) ** (-1) + \
15                               (2 / (r * g * w)))) ** 2
16     E = E / (Num * Num)
17     return torch.sum(E)
```

En las líneas 2 y 3, se recalculan los valores de `H` y `U` como se describió antes. También se calcula el producto `HU` que representa el flujo Q en la ecuación original. Luego se calcula el

gradiente el gradiente y se escala de forma apropiada usando los valores de delta que se usaron para el intervalo y finalmente se calcula el error.

Con todo eso terminado, solo hace falta entrenar la red neuronal usando el gradiente descendiente. El código es el siguiente:

```
1  Xi = torch.Tensor([[0]])
2
3  model = NeuralNetwork()
4
5  trials = int(5 * 10**7)
6
7  optimizer = torch.optim.SGD(model.parameters(), lr=0.00000001)
8  losses = []
9  for trial in range(trials):
10     optimizer.zero_grad()
11     Nh, Nu = model(Xi)
12     loss = criterion(Nh, Nu)
13     losses.append(loss.item())
14     loss.backward()
15     optimizer.step()
16     if trial % 10000 == 0:
17         print(round(loss.item(), 3), trial, end = '            \r')
18     if trial % 1000000 == 0:
19         save(trial)
```

en la línea 3 se instancia la red neuronal. `torch.optim.SGD` es la implementación optimizada de torch para el gradiente descendiente estocástico, que en este caso, como el input siempre va a ser el mismo, es equivalente al gradiente descendiente ordinario. En la línea 11 se evalúa el modelo y en la 12 se calcula la pérdida. El comando `loss.backward()` se aplica al tensor `loss` que se calculó con la función de error y calcula la gradiente para todos los parámetros. El comando `optimizer.step()` da el paso del gradiente descendiente $W_i = W_{i-1} - \gamma \nabla E$. La función `save` simplemente guarda el modelo para poder cargarlo luego.

El código anterior usa el algoritmo básico de gradiente descendiente. Este sí nos lleva en la dirección de convergencia, pero es muy lento porque conforme el modelo se acerca a la solución definitiva, los gradientes se hacen cada vez más pequeños y hacen los pasos más pequeños. Se puede mejorar ese problema calculando el decrecimiento promedio y aumentando la tasa de

aprendizaje si decrece muy lentamente, pero se tiene que limitar la tasa de aprendizaje para que no exista divergencia cuando el método esté muy cerca de la solución final. Esto no es óptimo porque cada vez que la tasa de aprendizaje sea baja, se tendrá que volver a definir el optimizador y es imposible saber cuándo la velocidad de convergencia es apropiada. La mejor solución es usar el optimizador *Adam*, que se explicó antes. El código modificado sería el siguiente:

```
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
losses = []
import statistics as st
for trial in range(trials):
    optimizer.zero_grad()
    Nh, Nu = model(Xi)
    loss = criterion(Nh, Nu)
    losses.append(loss.item())
    loss.backward()
    optimizer.step()
    if trial % 10000 == 0:
        print(round(loss.item(), 3), trial, end = '            \r')
        if losses[-1] < 0.0015:
            print("Loss low enough. Exiting training loop.")
            break
save(trials)
```

CAPÍTULO 5

RESULTADOS

5.1. Resultados del método propuesto

La función con la que se empezó, es decir, la salida de la red antes del entrenamiento, se puede ver en la figura 5.1 como se puede ver, a esta escala, el decrecimiento de la función

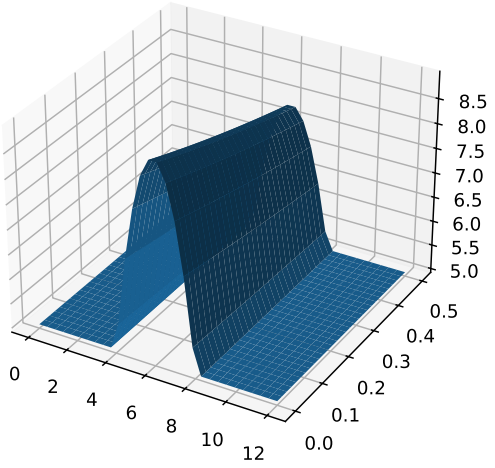

Figura 5.1: Función A_h

exponencial casi no tiene efecto, así que se podría haber tenido resultados similares sin ese decrecimiento.

Para entrenar redes neuronales tan grandes, es importante poner atención al progreso de la función de error con el algoritmo. Luego de correr el código expuesto anteriormente, se obtuvo los resultados mostrados en la figura 5.2 , en ese punto, la razón de cambio en cada paso del entrenamiento es muy baja porque la red ya está cerca de la solución final y las gradientes son pequeñas. Haciendo la modificación mencionada en la sección anterior para ajustar la razón

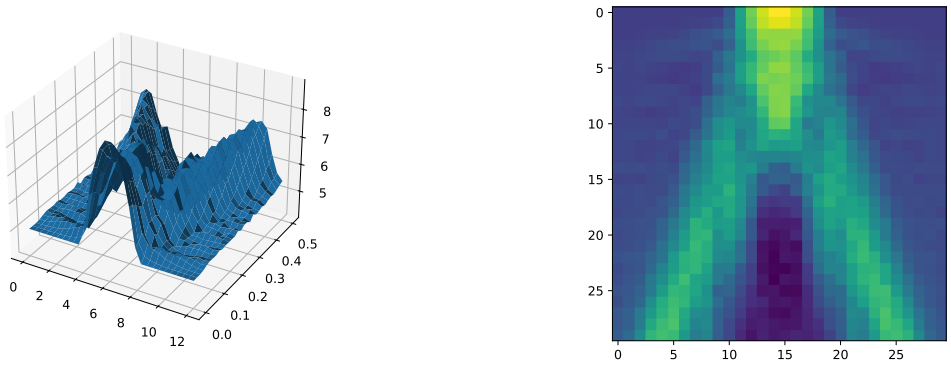

Figura 5.2: Salida de la red entrenada por primera vez

de cambio apropiadamente, y limitando la tasa de aprendizaje a 200[1], se tiene un error de menos de 0.0015 en aproximadamente 4×10^6 iteraciones, que corre en aproximadamente tres horas usando un nodo de 4 CPUs y 8gb de RAM. Los resultados luego de este proceso son los mostrados en la figura 5.3. En la parte superior se encuentran las soluciones para la altura del agua y en la inferior las de la velocidad. A la izquierda se muestran los gráficos en 3D, donde el tiempo va de 0 a 0.5 y x va de 0 a 12. A la derecha están los mismos gráficos pero como imagen de calor. El gráfico de error se puede encontrar en 5.4. Se puede notar alrededor de $1{,}5 \times 10^6$ iteraciones, que se empieza a activar el ajuste de tasa de aprendizaje.

Como ya se mencionó, el algoritmo *Adam* es una forma más eficiente de mejorar la convergencia. Usando el código modificado presentado en la sección anterior, se llega a un error de menos de 0.0015 en menos de 10^4 iteraciones como se puede ver en la figura 5.5. Este termina de converger en menos de un minuto.

5.2. Comparación con un método tradicional

El método más común para resolver ecuaciones diferenciales parciales es el de diferencias finitas. Este es muy similar al método de Euler para ecuaciones diferenciales ordinarias. En este caso, se va a implementar una solución similar a la propuesta por (Sukron et al., 2021), pero se usará Runge Kutta de orden 1 (Forward Euler) en lugar de orden 4, como recomienda el paper, para avanzar sobre el tiempo. La implementación es la siguiente:

[1]La tasa de aprendizaje es relativamente alta en comparación a otros modelos por la división por el número de puntos que se hace en la función de error.

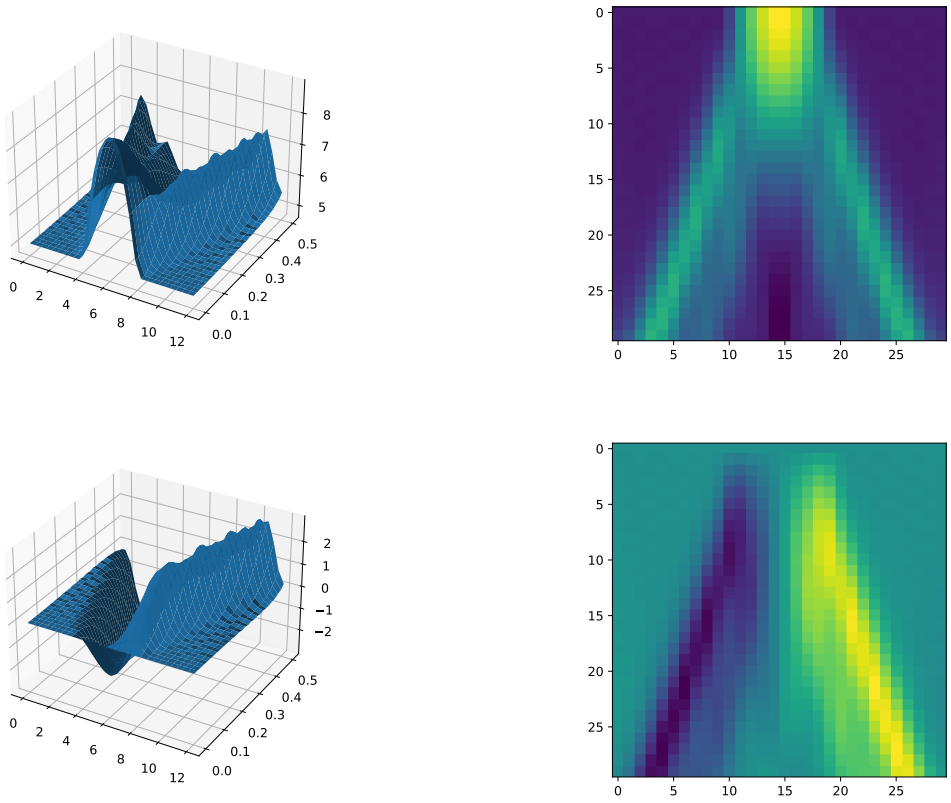

Figura 5.3: Red al final del entrenamiento.

```
for j in range(NL-1): #estamos avanzando sobre t
    HU = H * U
    #se calculan las gradientes
    DxH = np.gradient(H, deltax, edge_order = 2)[1]
    DxHU = np.gradient(HU, deltax, edge_order = 2)[1]
    DxU = np.gradient(U, deltax, edge_order = 2)[1]
    #se generan las derivadas para estimar el siguiente paso
    DtH = -DxHU
    DtU = g * (1 * (r * g * H) ** (-1) + (2 / (r * g * w))) - U * DxU - g *
     DxH
    # se calcula y se escribe el siguiente paso
    H[j + 1, :] = H[j, :] + deltat * DtH[j, :]
    U[j + 1, :] = U[j, :] + deltat * DtU[j, :]
```

La salida del método tradicional se puede encontrar en la figura 5.6. Como se puede ver, el resultado es similar, pero no es igual. El error relativo con respecto al método tradicional ($|H_{\text{Tradicional}} - H_{\text{Red}}/H_{\text{Tradicional}}|$) en promedio es de $0{,}0405$ o $4{,}05\,\%$. Esta discrepancia puede aparecer por la inestabilidad numérica del método de Euler tradicional. La inestabilidad es conocida para estas ecuaciones diferenciales y usualmente se usa un modelo de predictor-

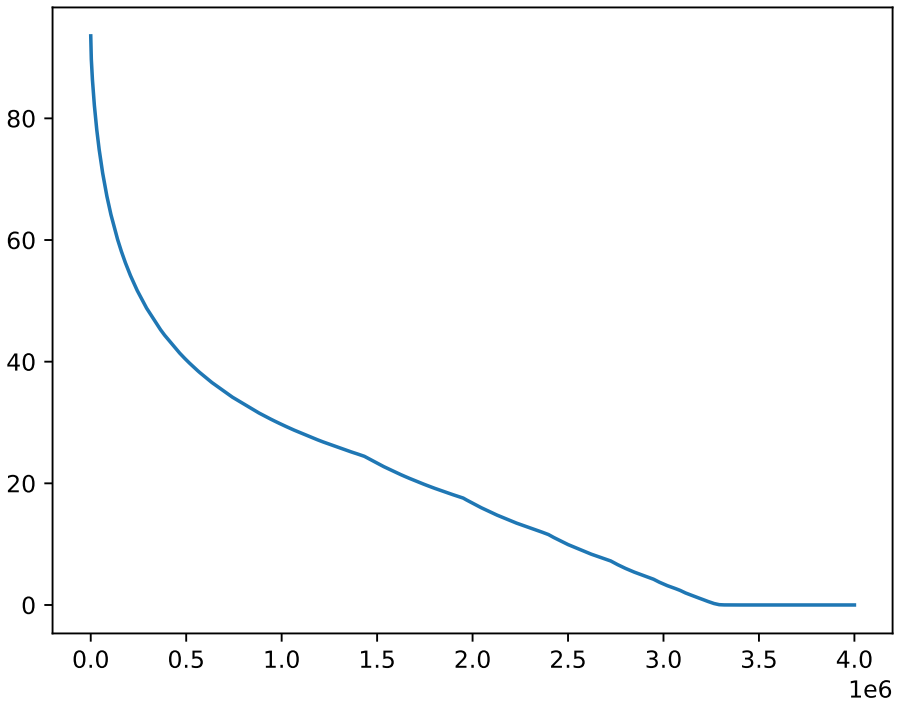

Figura 5.4: Gráfica de errores con tasa de aprendizaje ajustada

corrector como en (Fauzi and Wiryanto, 2018).

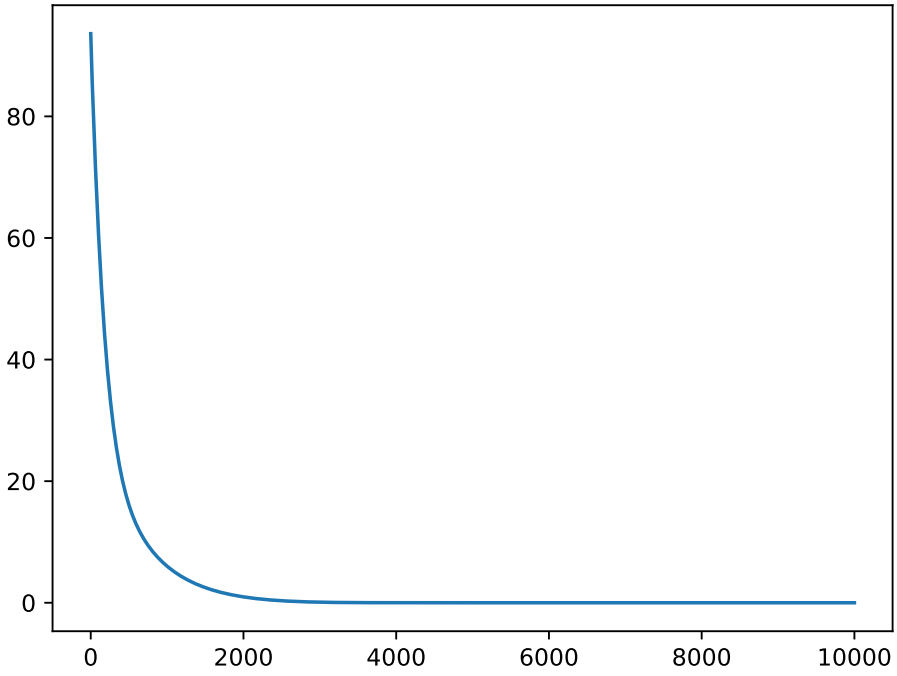

Figura 5.5: Errores usando *Adam* (Kingma and Ba, 2017)

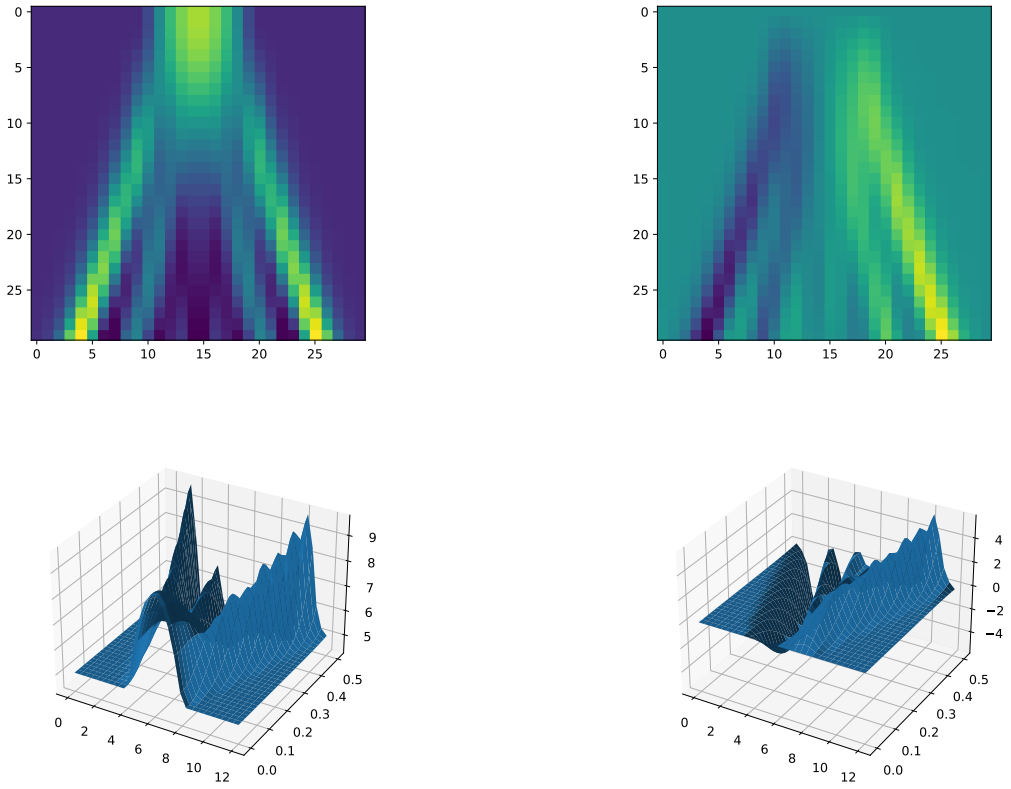

Figura 5.6: Resultado del método de tradicional con los mismos parámetros. A la izquierda la matriz H y a la derecha la velocidad U

CAPÍTULO 6

CONCLUSIONES Y TRABAJO FUTURO

Como se ha expuesto, el método propuesto es capaz de resolver ecuaciones diferenciales con precisión aceptable con respecto al método numérico tradicional. Se mencionó que la eficiencia del método depende del algoritmo de optimización que se use y se expuso dos algoritmos de optimización relacionados para resolver el mismo problema.

Teóricamente, el método expuesto es más lento que otros métodos para resolver ecuaciones diferenciales parciales como Runge-Kutta de diferentes órdenes, sin embargo, gracias a la capacidad de paralelización que ofrecen las redes neuronales, se pueden obtener tiempo de ejecución y precisión comparables a los de otros métodos.

Una ventaja considerable de este método es la facilidad que presenta para definir condiciones de contorno. Además, gracias a la naturaleza iterativa del método, el orden de aproximación de los valores de la función en el dominio pierde importancia y gracias a que se calcula la función de error en cada paso, es trivial averiguar si la función cumple con las condiciones de frontera y la ecuación diferencial.

Es importante recordar que este método es relativamente nuevo y requiere más investigación en las funciones posibles para A, los algoritmos de optimización y arquitecturas de redes neuronales. Además, el método se puede hacer aún más eficiente haciendo uso de paralelización o algoritmos de aceleración de divergencia como la extrapolación de Richardson.

BIBLIOGRAFÍA

Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538.

Chen, F., Sondak, D., Protopapas, P., Mattheakis, M., Liu, S., Agarwal, D., and Di Giovanni, M. (2020). Neurodiffeq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, 5(46):1931.

Chow, V. T. (1959). *OPEN-CHANNEL HYDRAULICS*. McGraw-Hill Book Company, Inc.

Curry, H. B. (1944). The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261.

Fauzi, R. and Wiryanto, L. H. (2018). Predictor-corrector scheme for simulating wave propagation on shallow water region. In *IOP Conference Series: Earth and Environmental Science*, volume 162, page 012047. IOP Publishing.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

Neha Yadav, Anupam Yadav, M. K. (2015). *An Introduction to Neural Network Methods for Differential Equations*. Springer.

Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855.

Sukron, M., Habibah, U., and Hidayat, N. (2021). Numerical solution of saint-venant equation using runge-kutta fourth-order method. In *Journal of Physics: Conference Series*, volume 1872, page 012036. IOP Publishing.

Veelenturf, L. (1995). *Analysis and Applications of Artificial Neural Networks*. Prentice Hall International.

Wright, S. (2015). Coordinate descent algorithms. *Mathematical Programming*, 151(1847):3–34.