Proyecto Final de Métodos Numéricos

Jaime Francisco Aguayo González

December 9, 2017

1 Uso de Redes Neuronales para la resolución de Ecuaciones Diferenciales Ordinarias

En este proyecto me di a la tarea de investigar otros métodos para la resolción de ecuaciones diferenciales ordinarias que no hayamos visto en clase. Una de las que más llamó mi atención fue el uso de redes neuronales para su solución.

Este trabajo se divide en tres partes, en la primera abordaré algunos conceptos y definiciones sobre redes neuronales, abordaremos la base teórica que dicta el método de resolución, después veremos la implementación de la red neuronal y la resolución de dos problemas, comparando las aproximaciones con el método de diferencias finitas; finalmente abordaremos las conclusiones del trabajo, resaltando las diferencias encontradas entre ambos métodos, sus ventajas y desventajas.

1.1 Red neuronal

Una red neuronal artificial es un modelo computacional basado en el comportamiento observado en los axones de las neuronas en los cerebros biológicos. Es un conjunto de unidades neuronales conectadas entre si por enlaces que pueden incrementar o inhibir el estado de activación de sus neuronas vecinas. Cada unidad neuronal, de forma individual, opera empleando funciones de suma y producto.

Normalmente se identifican tres tipos de neuronas, las neuronas de entrada (Input layer), que colectan la información, las capas ocultas (hidden layers) que procesan la información y las neuronas de salida (output layer) que emiten un resultado.

Una red neuronal es un sistema que aprende y se forma a si misma, en lugar de ser programado de forma explícita, y sobresale en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional.

El método para resolver ecuaciones diferenciales ordinarias, toma su fundamento en el Teorema de Cybenko o Teorema de Aproximación Universal que expresa:
"Sea φ una función continua no constante, acotada y monótonamente creciente. Denotamos por I_m al cubo unitario de dimensión m, $[0,1]^m$ y al espacio de funciones continuas en I_m por $C(I_m)$. Entonces, dado $\epsilon > 0$ y $f \in C(I_m)$, existe un entero positivo N, constantes $v_i, b_i \in \mathbb{R}$ y $w_i \in \mathbb{R}_m$, $i = 1, 2, ..., N$, tales que:

$$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

cumple que

$$|F(x) - f(x)| < \epsilon, \forall x \in I_m$$

En otras palabras, $F(x)$ es denso en $C(I_m)$."

El resultado también es cierto si se sustituye I_m por un subconjunto compacto en $\mathbb{R}^m$

El teorema nos dice que cualquier función continua definida en un compacto, f, puede ser aproximada usando una red neuronal con una sola capa oculta. De aquí que se pueda aproximar una función que sea solución de una EDO.

1.1.1 Sigmoide

Una función que se suele usar mucho en redes neuronales es la función sigmoide definida por

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

Esta función es muy usada por las siguientes propiedades:

- $\frac{d}{dx}S(x) = \frac{e^x \cdot (1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2}$ por lo que $\frac{d}{dx}S(x) = S(x)(1 - S(x))$.
- $\lim_{x \to -\infty} S(x) = 0$
- $\lim_{x \to +\infty} S(x) = 1$

De lo anterior se observa que $S(x)$ es continua y acotada, además es fácil darse cuenta que la función es monótonamente creciente por lo tanto S satisface las condiciones del Teorema de Aproximación Universal.
Para los siguientes resultados usaremos las sigmoides como funciones de aproximación en cada red neuronal, por lo que la red neuronal a usar se puede expresar como:

$$N(x) = \sum_{i=1}^{n} v_i S(w_i x + u_i)$$

1.2 Descripción del método

Se pretende construir una solución basada en una red neuronal. Esto queda ilustrado en la ecuación:

$$G(x, \Psi(x), \Delta\Psi(x), \Delta^2\Psi(x)) = 0, x \in D$$

para el caso de una ecuación de segundo orden, sujeto a las condiciones de forontera o valores iniciales, donde $D \subset \mathbb{R}^n$ es el dominio y $\Psi(x)$ la solución a ser calculada.

Este método tambien puede ser usado para ecuaciones diferenciales de grados superiores pero en este trabajo no consideramos problemas mayores que segundo grado.

Para obtener una solución a la ecuación diferencial anterior, asumimos que es posible hacer una discretización del dominio D y su frontera S en un conjunto de puntos $\hat{D}$ y $\hat{S}$ respectivamente. Luego, el problema se tranforma en el sistema de ecuaciones:

$$G(x_i, \Psi(x_i), \Delta\Psi(x_i), \Delta^2\Psi(x_i)) = 0, \forall x_i \in \hat{D}$$

nuevamente sujetas a las condiciones de frontera.

Denotemos a nuestra aproximación de la solución por $\Psi_t(x, \overrightarrow{p})$ donde $\overrightarrow{p}$ son los parámetros ajustables de la arquitectura neuronal (pesos y *biasis*). Notemos que ahora nuestra solución depende únicamente del parámetro $\overrightarrow{p}$, por lo que podemos transformar nuestro problema anterior en encontrar una solución en el sentido de mínimos cuadrados de la expresión, es decir:

$$\min_{\overrightarrow{p}} \sum_{x_i \in \hat{D}} [G(x_i, \Psi_t(x_i, \overrightarrow{p}), \Delta\Psi_t(x_i, \overrightarrow{p}), \Delta^2\Psi_t(x_i, \overrightarrow{p}))]^2$$

(1)

Queremos definir Ψ_t como una función que dependa de la red neuronalbde propagación hacia adelante (feedfordward) y que además satisfaga las condiciones iniciales del problema, donde, como ya se ha mencionado, $\overrightarrow{p}$ representa los pesos y biasis de nuestra red, así que construimos a $\Psi_t(x)$ como la suma de dos términos:

$$\Psi_t(x) = A(x) + F(x, N(x, \overrightarrow{p}))$$

donde $A(x)$ satisface las condiciones iniciales del problema y, cuando eso sucede, F no contribuye a las condiciones iniciales del problema.

En este sentido, entrenar a nuestra red neuronal corresponde a minimizar la ecuación (1), y para minimizar la expresión podemos usar el método del descenso del gradiente por lo que necesitamos obtener el gradiente de la función con respecto a $\overrightarrow{p}$.

1.2.1 Solución de una EDO de primer orden

Consideremos una ecuación diferencial ordinaria de primer orden:

$$\frac{d\Psi(x)}{dx} = f(x, \Psi)$$

con $x \in [0, 1]$ y el valor inicial $\Psi(0) = A$.

Se puede escribir la solución como

$$\Psi_t(x) = A + xN(x, \overrightarrow{p})$$

De esta forma se satisface trivialmente la condición inicial del problema. La función error a minimizar es:

$$E[\overrightarrow{p}] = \sum_i [\frac{d\Psi(x_i)}{dx} - f(x_i, \Psi(x_i)]^2$$

donde $x_i \in [0, 1]$. Notemos que $\frac{d\Psi(x)}{dx} = N(x, \overrightarrow{p}) + x\frac{dN(x, \overrightarrow{p})}{dx}$ lo cual nos gusta pues derivar la red neuronal con respecto de x es una tarea muy sencilla.

Esto se puede generalizar a un sistema de ecuaciones, donde a cada $y_i(x)$ se le asocia una solución $\Psi_{ti}(x)$.

1.2.2 Solución a una EDO de segundo orden

En este caso consideramos el problema:

$$\frac{d^2\Psi(x)}{dx^2} = f(x, \Psi, \frac{d\Psi(x)}{dx})$$

con $x \in [0, 1]$.

Si tenemos un problema de valore iniciales $\Psi(0) = A$, $\Psi'(0) = A'$, podemos construir nuestra solución como:

$$\Psi_t(x) = A + xA' + x^2 N(x, \overrightarrow{p})$$

En cambio, si nos dan las condiciones de frontera $\Psi(0) = A, \Psi(1) = B$, podemos construir la solución como:

$$\Psi_t(x) = A(1 - x) + Bx + x(1 - x)N(x, \overrightarrow{p})$$

De esta forma no es necesario transformar el problema a una ecuación con dos incógnitas, sin embargo se necesita obtener más aproximaciones a derivadas.

1.3 Construcción de la red neuronal

Antes de pasar a resolver algunos problemas, vamos a implimentar nuestra red. Para esto usaremos python y las librerias numpy y autograd. La última de estas, contiene una función que nos será útil para obtener el gradiente de la red neuronal con respecto de $\overrightarrow{p}$.

```
In [1]: import autograd.numpy as np
        from autograd import grad
        import autograd.numpy.random as npr
        from autograd.core import primitive
        from matplotlib import pyplot as plt
        # Configura tamaõ de las gr\'aficas de salida
        %matplotlib inline
        plt.rcParams["figure.figsize"] = (10, 6)

In [2]: # Funci\'on sigmoide
        def sigmoid(x):
            return 1 / (1 + np.exp(-x))


        # Derivada del sigmoide con respecto x
        def sigmoid_grad(x):
            return sigmoid(x) * (1 - sigmoid(x))


        # Red neuronal con tres neuronas de pesos W
        def neuralNetwork(W, x):
            a1 = sigmoid(np.dot(x, W[0])+W[2])
            return np.dot(a1, W[1])
```

```
# Derivada de la red neuronal con respecto de x con pesos W
def d_neuralNetwork_dx(W, x, k=1):
    return np.dot(sigmoid_grad(np.dot(x, W[0])+W[2]), W[1]*W[0].T)


# Funci\'on a minimizar con pesos W, xi's en x
def lossFunction(W, x):
    loss_sum = 0.
    for xi in x:
        net_out = neuralNetwork(W, xi)[0][0]        # Valor de la red neuronal en xi
        psy_t = 1. + xi * net_out                    # Psi_t(x) OJO: Unica parte que depe
        d_net_out = d_neuralNetwork_dx(W, xi)[0][0]# derivada de la red neuronal evaluad
        d_psy_t = net_out + xi * d_net_out           # derivada de trial solution
        func = F(xi, psy_t)                          # sumando izquierdo de la cosa a min
        err_sqr = (d_psy_t - func)**2                # Saca cuadrado

        loss_sum += err_sqr                          # Suma de los cuadrados
    return loss_sum                                  # Regresa valor de E[x, W]
```

1.4 Ejemplo 1:

Consideremos el sisguiente problema de valores iniciales:

$$\frac{d\Psi(x)}{dx} + (x + \frac{1+3x^2}{1+x+x^3})\Psi(x) = x^3 + 2x + x^2\frac{1+3x^2}{1+x+x^3},$$

$\Psi(0) = 1, x \in [0,1]$

Hacemos una partición de 10 puntos en $[0,1]$ y definimos la función $F(x, \Psi(x))$

```
In [3]: nx = 10         # Numero de puntos
        dx = 1. / nx    # Espacio entre puntos
        x_space = np.linspace(0, 1, nx)    # Creamos 10 puntos equiespaciados en [0, 1]

        # f
        def F(x, psy):
            return B(x) - psy * A(x)


        # Parte dependiente de psi
        def A(x):
            return x + (1. + 3.*x**2) / (1. + x + x**3)


        # Parte independiente de psi
        def B(x):
            return x**3 + 2.*x + x**2 * ((1. + 3.*x**2) / (1. + x + x**3))
```

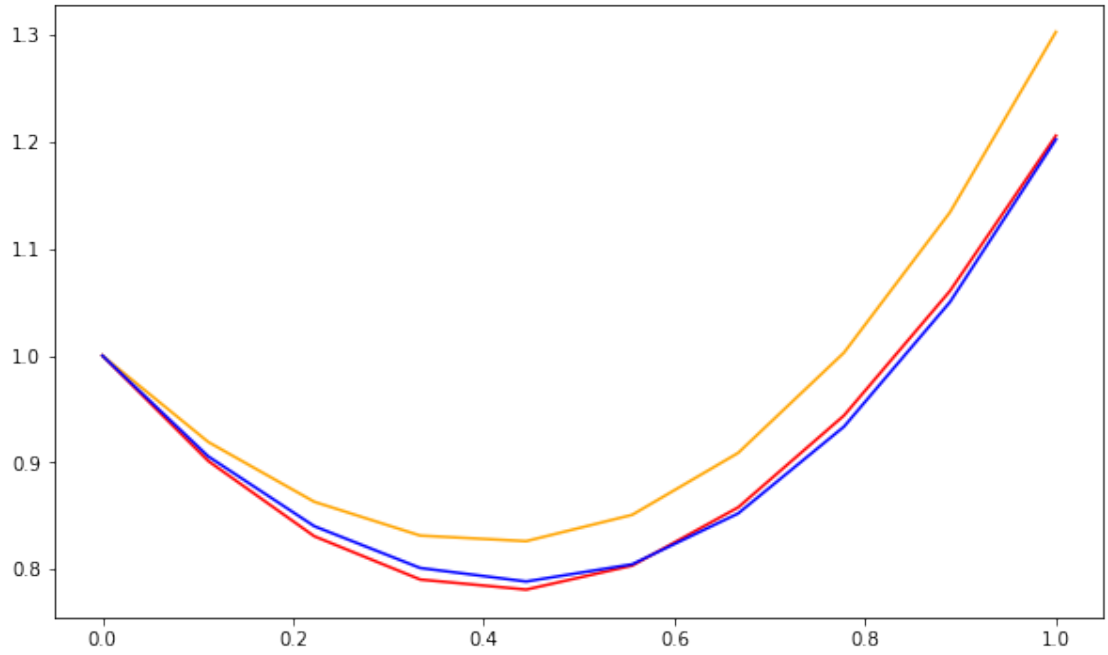
Ahora entrenamos nuestra red neuronal, es decir, encontramos los pesos p tales que la expresión (1) es mínima. En este caso, usaremos una red neuronal con 10 neuronas. En el ejemplo 2 veremos que se pueden obtener resultados igual de buenos con menos neuronas.

```
In [4]: # Pesos en la red neuronal, dos vectores de 1x10 y 10x1 (De input a hidden layer, de hid
        # N\'umero de neuronas: 10
        W = [npr.randn(1, 10), npr.randn(10, 1), npr.randn(1, 10)]
        lmb1 = 0.004
        lmb2 = 0.0003

        ## Entrenando la red neuronal

        for i in range(1000):
            loss_grad =  grad(lossFunction)(W, x_space)    # Arreglo n-dimensional

            # Restandole el gradiente
            W[0] = W[0] - lmb1 * loss_grad[0]
            W[1] = W[1] - lmb1 * loss_grad[1]
            W[2] = W[2] - lmb2 * loss_grad[2]

            #print(lossFunction(W, x_space))
            if lossFunction(W, x_space) < 0.025:
                print ("lossFunction = ",lossFunction(W, x_space), ", Iter: ", i)
                break
        print ("lossFunction = ",lossFunction(W, x_space))

lossFunction =  0.0249814304011 , Iter:  549
lossFunction =  0.0249814304011
```

Para comparar, vamos a resolver la misma ecuación usando diferencias finitas y a graficar ambas soluciones.

```
In [5]: # Resolviendo con diferencias finitas
        psy_fd = np.zeros_like(x_space)
        psy_fd[0] = 1. # IC
        for i in range(1, len(x_space)):
            psy_fd[i] = psy_fd[i-1] + B(x_space[i]) * dx - psy_fd[i-1] * A(x_space[i]) * dx

        # Graficando puntos usando la red neuronal
        Psi_NN = [1 + xi * neuralNetwork(W, xi)[0][0] for xi in np.linspace(0, 1, 10)] # Guarda

        # Graficando soluci\'on anal'itica
        def psy_analytic(x):
            return (np.exp((-x**2)/2.)) / (1. + x + x**3) + x**2

        y_space = psy_analytic(x_space)

        # Imprime
        plt.figure()
        plt.plot(np.linspace(0, 1, 10), Psi_NN, "red")
        plt.plot(x_space, psy_fd, "orange")
```

```
        plt.plot(x_space, y_space, "blue")
        plt.show()
        print("En azul, se ve la soluci\'on anal\'itica")
        print("En rojo se tiene la soluci\'on usando la red neuronal")
        print("En naranja tenemos la soluci\'on usando diferencias finitas")
```




```
En azul, se ve la solución analítica
En rojo se tiene la solución usando la red neuronal
En naranja tenemos la solución usando diferencias finitas
```

```
In [6]: suma1 = 0.
        suma2 = 0.
        for i in range(nx):
            suma1 += (y_space[i] - psy_fd[i])**2
            suma2 += (y_space[i] - Psi_NN[i])**2
        print("Error en diferencias finitas: ", np.sqrt(suma1))
        print("Error con red neuronal: ", np.sqrt(suma2))
```

```
Error en diferencias finitas:  0.174168061407
Error con red neuronal:  0.0233072296299
```

Por lo que podemos observar que el error, con respecto a diferencias finitas es significativamente menor con el mismo número de puntos.

Podemos obtener un error similar usando diferencias finitas si aumentamos el tamaño de la partición.

Para este ejemplo se obtuvieron los siguientes resultados:

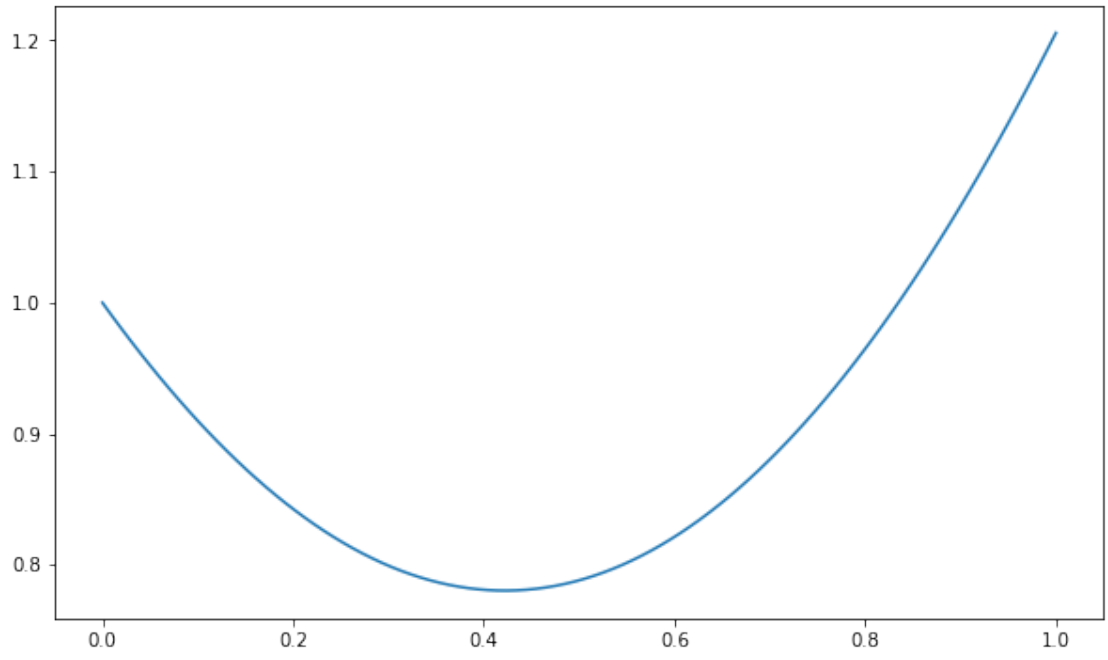
Tamaño de la partición	Error absoluto
10	0.17416
15	0.13623
20	0.07088
25	0.04569
30	0.03282
50	0.01354

Por lo que para obtener un resultado similar se tendría que hacer una partición de 50 puntos.

Otra de las ventajas que presenta la solución usando redes neuronales es que si ahora queremos obtener el valor de la solución en otros puntos, basta con evaluar $\Psi_t(x) = A + xN(x, \overrightarrow{p})$ en dichos puntos, mientras que con el método de diferencias finitas, se tiene que usar algun método de interpolación, o en el peor de los casos, usar diferencias finitas con una nueva partición.

Por ejemplo, ahora podemos aumentar el número de puntos a 100 y graficarlos.

```
In [7]: Psi_NN = [1 + xi * neuralNetwork(W, xi)[0][0] for xi in np.linspace(0, 1, 100)] # Guarda

        # Imprime
        plt.plot(np.linspace(0, 1, 100), Psi_NN)
        plt.show()
```



1.5 Ejemplo 2

Vamos a considerar el siguiente problema:

$$\frac{d\Psi(x)}{dx} = e^{-x/5}\cos(x) - \frac{1}{5}\Psi(x)$$

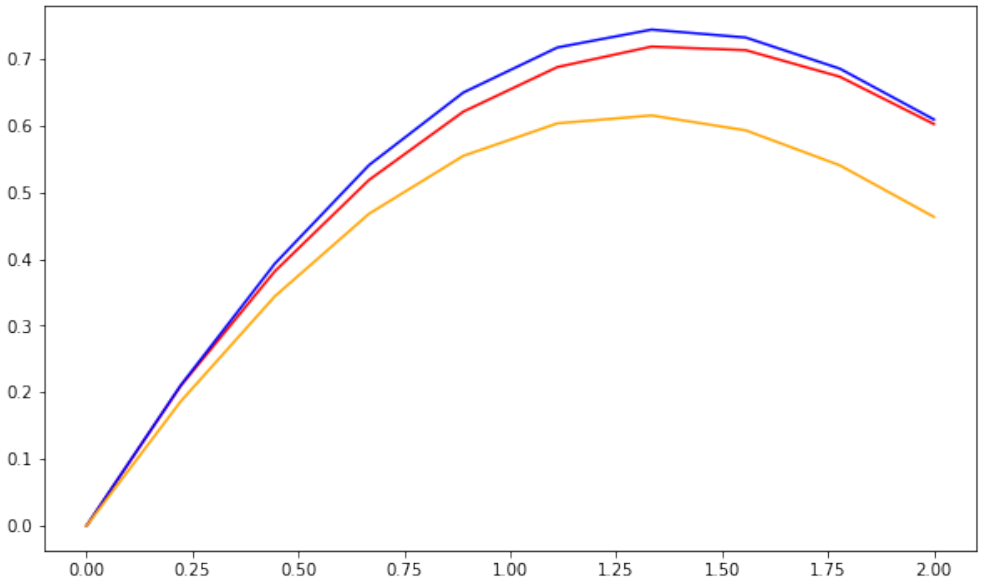
con $\Psi(0) = 0$, $x \in [0, 2]$

La solución analítica de esta ecuación es $\Psi(x) = e^{-\frac{x}{5}}\sin(x)$.

Usando como solución $\Psi_t(x) = xN(x, \overrightarrow{p})$ donde $N(x, \overrightarrow{p})$ es una red neuronal con una solo capa oculta. En este ejemplo nos interesa ver qué tanto afecta el número de neuronas en la solución.

Vamos a considerar una red neuronal con un *hidden layer* de 2, 3, 5 y 10 neuronas con una partición fija de 10 puntos y una tolerancia de 0.01 para la minimización. Se obtuvieron los resultados mostrados en la tabla 01.

Notemos que se obtuvieron muy buenos resultados incluso con 2 neuronas. El mejor resultado se obtuvo un 3 neuronas y el peor con 4, aunque con 10 neuronas tambien se obtuvo un buen resultado. Para compara, con el mismo tamaño de partición, usando diferencias finitas se obtuvo un error de 0.329. A continuación grafico ambas soluciones usando 2 neuronas, nuevamente, la roja es la solución usando la red neuronal, la azul la solución analítica y la naranja usando diferencias finitas:



En este ejemplo, si aumentamos la partición a 30, con diferencias finitas se obtiene un error de 0.04137, similar al obtenido usando dos neuronas y una partición de 10 puntos.

Neuronas	Error absoluto
2	0.0593209
3	0.0207175
4	0.0564565
5	0.0306570
8	0.0298798
10	0.0290086

Tabla 1

Tam. Partición	Error absoluto
3	0.1723
4	0.1006
5	0.0840
10	0.04978
20	0.03282

Tabla 2

Ahora vamos a dejar fijas 3 neuronas y aumentamos el tamaño de la partición a 3, 4, 5, 10 y 20. En cada caso calculo el error generado al comparar 10 puntos equiespaciados. Los resultados se encuentra en la tabla 2.

Además, si aumentamos el tamaño de la partición y al mismo tiempo aumentamos el número de neuronas, se obtiene un mejor resultado.

Finalmente en cuanto a tiempos de ejecución, usando 3 neuronas y una partición de 10 obtenemos 2.192s en su ejecución, mientras que usando una partición de tamaño 50 con diferencias finitas tenemos un tiempo de ejecución de 0.405s. En el primero obtenemos un error de 0.099, mientras que en el otro un error de 0.12. Por lo que podemos observar que diferencias finitas es más rápido.

1.6 Conclusiones:

Después de analizar el método, llegamos a varias conclusiones que podemos dividir en ventajas que presenta sobre el método de diferencias finitas y desventajas que presenta el método descrito.

1.6.1 Ventajas del método:

- La solución obtenida con redes neuronales es una forma analítica cerrada, continua y diferenciable que puede ser usada en cualquier otro cálculo.
- Usando la misma partición, en los tres ejemplos presentados se puede observar que da una mejor aproximación que la proporcionada por diferencias finitas
- No requiere de propiedades adicionales de la ecuación
- Puede ser fácilmente generalizable para resolver ecuaciones parciales
- El método puede ser fácilmente adaptable para trabajar en paralelo
- Puede usarse métodos más eficientes para calcular el gradiente

- Pueden usarse otros métodos adaptativos para que el aprendizaje sea más rápido

1.6.2 Desventajas del método:

- La implementación es tardada, ya que requiere programar varias cosas. De hecho, la solución por diferencias finitas se puede implementar en unas 4 líneas de código
- Es considerablemente más tardado que diferencias finitas. Incluso es un poco más tardado que usar diferencias finitas con más puntos de forma tal que el error sea el mismo
- Tienes que jugar un poco más con los parámetros

2 Bibliografía

Para el trabajo me basé en dos documentos:

- Artificial Neural Networks Approach for Solving Stokes Problem, Modjtaba Baymani, Asghar Kerayechian, Sohrab Effati, 2010
- Artificial Neural Networks for Solving Ordinary and Partial Differential Equations, I. E. Lagaris, A. Likas and D. I. Fotiadis, 1997

Además de obtener algunas ideas del seminario de Deep Learning en el CIMAT.