Solución de Sistemas de Ecuaciones Diferenciales Ordinarias Paramétricos mediante Redes Neuronales Artificiales

Jordan Pla González
Grupo C311

Dianelys Cruz Mengana
Grupo C311

Tutor: MsC. Fernando Raúl Rodríguez Flores

Resumen

Las Redes Neuronales Artificiales (RNA) constituyen un modelo computacional de Aprendizaje de Máquinas, con numerosas aplicaciones en la actualidad. Una de las aplicaciones, que puede ser aprovechada en significativas esferas del desarrollo y la investigación, es la resolución de Ecuaciones Diferenciales Ordinarias. El objetivo fundamental de este trabajo es proponer un método para resolver Sistemas de Ecuaciones Diferenciales Ordinarias Paramétricas a través del uso de Redes Neuronales Artificiales. En concreto, se implementa el método para la resolución del sistema SIR, aunque las ideas que se exponen son generales y fácilmente extensibles.

Abstract

Artificial Neural Networks (ANN) constitute a computational model of Machine Learning, with numerous applications nowadays. One of the applications, which can be used in significant areas of development and research, is the resolution of Ordinary Differential Equations. The main objective of this work is to propose a method to solve Parametric Ordinary Differential Equation Systems through the use of Artificial Neural Networks. Specifically, the method for solving the SIR system is implemented, although the ideas that are exposed are general and easily extensible.

Palabras Clave: Red Neuronal Artificial, Sistema de Ecuaciones Diferenciales Ordinarias, Pytorch.

Tema: Solución de Sistemas de Ecuaciones Diferenciales Ordinarias Paramétricas mediante Redes Neuronales Artificiales.

1. Introducción

Un Sistema de Ecuaciones Diferenciales Ordinarias Paramétricas, puede definirse como un conjunto de Ecuaciones Diferenciales Paramétricas, condiciones iniciales e intervalos para la variable independiente y las variables paramétricas:

$$\begin{cases} \frac{dy_1}{dx} = f_1\left(x, \alpha_1, \dots, \alpha_n, y_1, \dots, y_m\right) & y_1(x_1) = y_{10} \\ \frac{dy_2}{dx} = f_2\left(x, \alpha_1, \dots, \alpha_n, y_1, \dots, y_m\right) & y_2(x_1) = y_{20} \\ \dots & \\ \frac{dy_m}{dx} = f_m\left(x, \alpha_1, \dots, \alpha_n, y_1, \dots, y_m\right) & y_m(x_1) = y_{m0} \end{cases}$$

Partiendo de la definición anterior y de otros aspectos que se exponen a continuación, llevamos a cabo el entrenamiento de una red neuronal que da solución a un sistema con este formato. Hemos utilizado el *framework* para trabajar con redes neuronales, **Pytorch**.

2. Solución Implementada

Dado que una red neuronal puede aproximar cualquier función, es posible crear una red neuronal que tenga como entrada un valor, correspondiente a la variable independiente del sistema, y como salida, la evaluación de este valor en cada una de las funciones del sistema. Por ende, la red neuronal empleada tiene un valor de entrada y tantos valores de salida como funciones haya en el sistema [1]. Para el ejemplo resuelto se utilizaron 100 neuronas en 1 capa oculta.

2.1. Solución Paramétrica

Se crea una solución paramétrica para cada una de las funciones, que garantiza que se cumplan las condiciones iniciales, de la siguiente forma:

$$\boldsymbol{y_i(x, \alpha_1, \dots, \alpha_n, w) = y_{i(0)} + (x - x_0)N_i(x, \alpha_1, \dots, \alpha_n, w)}$$

Donde:

- $\boldsymbol{y_i}$: función i-ésima del sistema.
- $\boldsymbol{y_i(x_0) = y_{i(0)}}$: condición inicial correspondiente a la ecuación i-ésima.
- $\boldsymbol{N_i(x, \alpha_1, \dots, \alpha_n w)}$**:** salida correspondiente a la función i-ésima en la red neuronal.
- $\boldsymbol{w}$ **:** vector de pesos de la red.

Notar que siempre que se cumpla $\boldsymbol{x = x_0}$ la evaluación en la función paramétrica es precisamente $\boldsymbol{y_{i(0)}}$, por lo cual se garantiza que se cumplan las condiciones iniciales.

2.2. Función de Pérdida

Para asegurar que las salidas de nuestro perceptrón multicapas se corresponden con la evaluación de las funciones del sistema en el intervalo especificado se debe solucionar el siguiente problema de optimización:

$$\min_w \sum_{j=0}^{p} \sum_{i=0}^{m} \left(\frac{dy_i(x_j, \alpha_1, \dots, \alpha_n, w)}{dx} - f_i(x, \alpha_1, \dots, \alpha_n, y_1, \dots, y_m) \right)$$

Donde:

- $x_0, \dots, x_p$ es un conjunto de valores en el intervalo de la variable independiente especificado.

Por tanto, nuestra red deberá ser entrenada para obtener valores de sus pesos que minimicen la sumatoria anterior. Esta sería, entonces la función de pérdida.

2.3. Flujo de Entrenamiento

El entrenamiento se realiza por épocas. En cada una de las épocas se tratan de optimizar los valores de los pesos en la red, de la siguiente forma:

Se genera un conjunto de valores $\boldsymbol{x_0, \dots, x_n}$ en el intervalo de la variable independiente. La idea de generar un nuevo conjunto en cada iteración contribuye a "escapar" de los óptimos locales [2]. Luego, la función de pérdida es evaluada en dichos valores y se calculan las derivadas de dicha función con respecto a los parámetros de la red (usando la función *backward* disponible en **Pytorch**).

Para la actualización de los pesos, **Pytorch** nos proporciona varios algoritmos de optimización, entre ellos **RProp** (*Resilient Propagation*). Este es un popular algoritmo de descenso de gradiente que solo usa los signos de gradientes para calcular actualizaciones.

En cada iteración de **RProp**, se calculan los gradientes y el tamaño de paso es actualizado.

Esto se hace comparando el signo del gradiente de la iteración actual y la anterior. La idea es la siguiente [3]:

- Cuando los signos son los mismos, vamos en la misma dirección que en la iteración anterior. Dado que esta parece ser una buena dirección, el tamaño del paso debe aumentarse para ir al óptimo más rápidamente.
- Si el signo cambió, la nueva actualización se está moviendo en una dirección diferente. Esto significa que acabamos de saltar sobre un óptimo. El tamaño del paso debe reducirse para evitar volver a saltar sobre el valor óptimo.

3. Conclusiones

A pesar de que los resultados obtenidos pueden variar mucho en el tiempo necesario para la ejecución, así como en la precisión, el método utilizado constituye una importante aplicación del Aprendizaje de Máquina al trabajo con ecuaciones diferenciales.

4. Recomendaciones

La implementación de estas ideas solo se realizó para la resolución del sistema SIR y sería conveniente obtener una generalización del código para cualquier sistema. A continuación, se mencionan los cambios fundamentales que se deben realizar para solucionar otro sistema:

- Cantidad de valores entrada y de salida de la red.
  Se deben modificar las variables **n_in** y **n_out** que representan la cantidad de parámetros y funciones del sistema, respectivamente.

- Intervalo de la variable independiente.
  Modificación de las variables **t_min** y **t_max** que corresponden al valor mínimo y máximo de la variable independiente, respectivamente.

- Condiciones iniciales.
  Se especifica el valor inicial de la variable independiente, así como la evaluación de cada una de las funciones en dicho valor.

- Intervalos y número de valores de los parámetros del sistema.
  Se deben añadir variables para los valores máximos y mínimos de cada una de las variables paramétricas. Así como la cantidad de valores a generar de las mismas en cada iteración.

- Funciones de las ecuaciones del sistema.
  Definición de $f_1, \dots, f_n$

- Creación de un conjunto de valores (y a partir de estos tensores) para cada una de las entradas de la red, en el método **closure_s**.

- Modificación de la función de pérdida:
  Se debe añadir el cálculo del gradiente por cada una de las funciones, así como el sumando correspondiente en la función de pérdida.

5. Referencias

[1] Hornik, K. (1991) Approximation capabilities of multilayer feedforward networks Neural Networks, 4, 251-257

[2] Ruder, Sebastian. An overview of gradien descent optimization algorithms.

[3] https://florian.github.io/rprop/