Programación 2: Proyecto 1

1st Carlos Zamorano
Escuela Ing. Eléctrica, PUCV
Valparaiso, Chile
carlos.zamorano.c@mail.pucv.cl

2nd Benjamin Aguirre
Escuela Ing. Eléctrica, PUCV
Valparaiso, Chile
benjamin.aguirre.s@mail.pucv.cl

***Abstract*—El presente informe tiene como objetivo describir el enfoque empleado para desarrollar el primer proyecto del curso Programación 2, utilizando el lenguaje de programación Python. En este proyecto, se trabajó en el procesamiento y análisis de señales ruidosas provenientes del espacio exterior, aplicando algoritmos de Machine Learning y diferenciación automática. Para ello, se emplearon las librerías PyTorch y NumPy, ampliamente utilizada para operaciones numéricas eficientes.**

***Index Terms*—Python, PyTorch, Numpy, Loss Function, Diferenciación automática, Machine Learning, Convergencia.**

I. Introducción y Teoria Aplicada

Para comenzar con el desarrollo del proyecto se ha importado la librería PyTorch la cual es una herramienta de aprendizaje automatico, Pytorch gracias a su enfoque basado en tensores y su capacidad de automatizar la diferencias mediante el motor Autograd, permitirá construir, entrenar y optimizar modelos, entre los factores mas destacables de esta librería es su compatibilidad para acelerar cálculos a través de los usos de la GPU.

Adicionalmente se implementan las librerías ya antes utilizadas como Numpy y Matplotlib para visualizar datos y realizar operaciones matemáticas así realizar calculo complejos de manera eficientes.

Sumado a las librerías, se debe entender que las funciones de perdidas son métodos para evaluar que tan bien el algoritmo modela su conjunto de datos, a continuación esto será fundamental para conocer el comportamiento de los datos a analizar y que tan eficiente está siendo la con la convergencia provocada por nuestro método de optimización.

Otra característica fundamental a explicar es que se utiliza el método de Newthon-Raphson ya que este algoritmo ayuda a encontrar aproximaciones de los ceros o raíces de una función real. El problema que tiene el método de Newton-Raphson es muy sensible a las condiciones iniciales, es decir al parámetro inicial con el cual se ejecutan las iteraciones.

Este metodo se implemento para la segunda actividad requerida tanto del problema 1 como del problema 2, este metodo se utilizo particularmente eligiendo un numero random que será un Xn+1 se le resta a la división de ese numero evaluada en la función, dividido en ese numero evaluado en la derivada de la respectiva función. El resultado obtenido es el nuevo numero “random” el cual se usara para comenzar las iteraciones. Con esta nueva cifra se vuelve a realizar el paso comentado anteriormente resultando así que cada ves mas , dicha cifra se aproxima logra llegar al cruce por cero.

Finalmente se volvera a crear un codigo cumpliendo con todo lo solicitado anteriormente, esta vez utilizando solo la libreria numpy asi realizar un analisis de la performance entre ambas librerias

II. Desarrollo de Contenidos

A. Ajuste del modelo con diferenciación automática

Para la primera pregunta se obtienen datos de un archivo txt llamado ’signal_N’ con N el numero de muestras de una señal la cual contiene ruido proveniente del espacio exterior, cuyo modelo físico esta dado por:

$$s(t) = a_1 + a_2 \arctan(x-1) + a_3(x+1)^3 \qquad (1)$$

Para resolver el misterio del origen de la señal, se necesita tener certeza del instante de tiempo en que su amplitud fue nula, su primera derivada y el valor de su integral en el intervalo [0,5].

A continuación se utiliza el archivo signal_1000.txt para resolver las siguientes interrogantes: al utilizar Pytorch junto con sus herramientas de diferenciación automática se logra ajustar en el modelo físico a la seña obteniendo como resultado el siguiente gráfico:

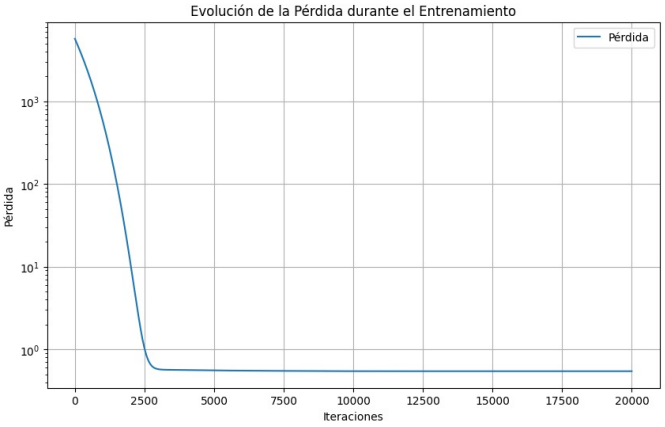

Fig. 1. Gráfico Perdida vs Iteraciones

A través del tiempo el modelo se ajusta cada vez más con mayor precisión logrando un 0.544% de perdidas, cifra mas que aceptable para lo requerido. Esto se logró utilizando la función de perdida y el recurso de optimización que incluye la librería Pytorch utilizando la cantidad de 20000 épocas.

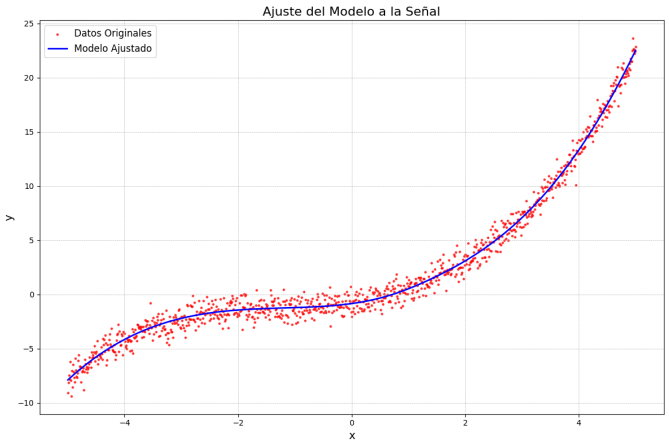

Fig. 2. Modelo ajustado a las muestras

La ecuación final que nos queda ajustada para nuestro modelo utilizando Pytorch es la siguiente:

$$-0.283118+0.847379\cdot\arctan(x-1)+0.100341\cdot(x+1)^3 \quad (2)$$

Cabe destacar que esta ecuación puede generar distintos valores de ajuste, ya que depende del tamaño de las muestras que se le proporcionen. A medida que se introducen más datos, la ecuación se ajusta de manera similar, aunque los resultados pueden variar en algunos decimales, sin embargo, seguirán una tendencia similar de ajuste.

B. Metodo Newton-Raphson

Se utiliza el método de Newton-Raphson para calcular el cruce por cero de la señal. En esta operación, se implementa la diferenciación automática proporcionada por la librería PyTorch.

$$g(x_n) = x_n - \frac{f(x_n)}{f'(x_n)} \quad (3)$$

Para realizar el cálculo con este método numérico, utilizamos la ecuación Eq. (3)

Primero, se define un valor inicial para comenzar las iteraciones, que será nuestro x_n. Este valor se evalúa para obtener $f(x_n)$, y la derivada correspondiente, $f'(x_n)$, se calcula mediante diferenciación automática utilizando el método 'backward' de PyTorch.

A partir de estos valores, se actualiza x_n en cada iteración. El proceso se repite hasta que la diferencia entre el valor actual y el anterior sea suficientemente pequeña. Una vez alcanzada la convergencia, habremos encontrado la raíz, es decir, el cruce por cero del modelo Eq. (2). Para el archivo de 1000 muestras nuestros resultados fueron los siguientes:

TABLE I
RESULTADOS MÉTODO NEWTON-RAPHSON

Iteraciones	Valor Inicial (x_n)	Valor Final
8	-0.8055 [-]	0.722364 [-]
4	0.1966 [-]	0.722364 [-]
6	-0.0682 [-]	0.722364 [-]

Decidimos realizar el experimento tres veces y observamos que en cada caso llegamos al mismo resultado, muy cercano al valor real del cruce por cero de nuestra ecuación. Sin embargo, también notamos que la cantidad de iteraciones puede variar, a veces siendo elevada, debido a que el valor inicial es aleatorio. Cuanto mayor sea este valor inicial, más tiempo tarda en converger al resultado final. Esto se debe a que el método es muy sensible a las condiciones iniciales (x_n).

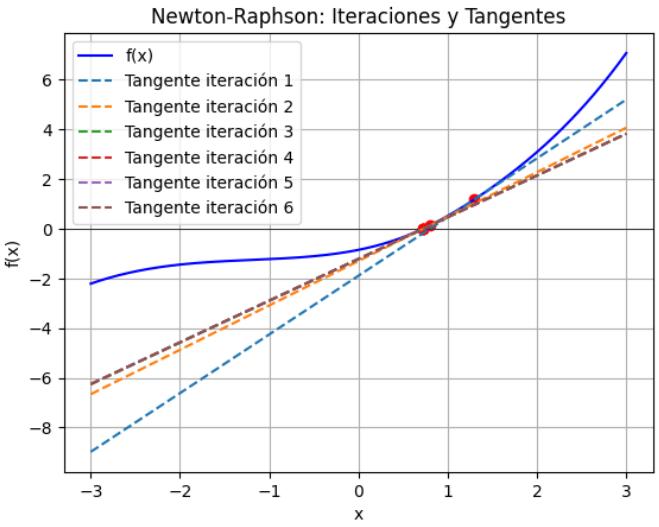

Fig. 3. Gráfico de tangentes cortando en cada punto de nuestro modelo

En la Imagen 3 se es de observar las iteraciones realizadas por el modelo. Cada tangente corresponde a un punto iterado, mostrando el proceso hasta que finalmente converge en la intersección con el eje en el punto cero.

C. Diferencias finitas hacia adelante

El método numérico llamado 'Diferencias finitas hacia adelante' sirve para aproximar la derivada de una función $f(x)$ en un punto dado.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (4)$$

Empezamos definiendo un tensor de tamaño $1e-6$ el cual será nuestra h a este le damos un valor pequeño para aproximar de manera más precisa la derivada de la función. Después definimos un x el cual será el punto en el que se va a evaluar la derivada, en este caso de valor 0, es decir en el cruce por 0 de nuestra derivada. Ya que tenemos los valores anteriores, iteramos en la Eq. (4) y obtenemos los siguientes resultados:

TABLE II
RESULTADOS DIFERENCIAS FINITAS HACIA ADELANTE

Iteraciones	Error relativo	Valor Final
5	6.922%	0.774860 [-]

Como podemos visualizar en la Tabla II, nos da un valor de error de casi el 7% pero aunque el error no es insignificante,

el resultado sigue siendo bastante cercano al valor teórico, el cual es de 0.7247[-].

D. Método de integración trapezoidal

Como ultima actividad a realizar se requiere estimar el valor de la integral en el rango dado [0,5] utilizando el método de integración trapezoidal implementado en PyTorch.

Cabe mencionar que el método de integración trapezoidal es una técnica que en lugar de integrar una función de manera exacta, se aproxima la curva por segmentos lineales, así sumar el area de cada uno de estos trapezoides para obtener el valor del área bajo la curva de manera aproximada. La ecuación de dicho método es la siguente:

$$I \approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right) \tag{5}$$

- h es el ancho de cada subintervalo $\left(h = \frac{b-a}{n}\right)$,
- $f(a)$ y $f(b)$ son los valores de la función en los extremos,
- $f(x_i)$ es el valor de la función en los puntos intermedios.

Como primer paso en el método que utilizamos, definimos una variable utilizando la función ‘torch.linspace‘, que genera un tensor de números equidistantes dentro de un intervalo. En este caso, el intervalo que establecimos es de 0 a 5, tal como se indica en el enunciado. Decidimos generar 100 puntos equidistantes en este rango, ya que mientras más puntos utilicemos, mejor será la aproximación que obtendremos para el valor de la integral dentro del intervalo que nos piden. Finalmente Pytorch, ya trae un método el cual calcula con este método el valor de la integral de la función llamado ’torch.trapz’.

$$\int_0^5 (f(x))dx = 33.99[-] \tag{6}$$

El valor de la integral calculada de forma directa, se visualiza en la Eq. (6) mientras que utilizando el método de integración trapezoidal, obtuvimos un valor de 33.9937[-], lo cual es prácticamente igual al valor teórico. Esto indica que la aproximación trapezoidal es muy precisa, pero esta depende de cuantos puntos intermedios le demos para aumentar la precisión.

III. Pregunta 2

Para este caso se solicita replicar lo realizado en la pregunta N°1 pero solo utilizando la librería Numpy, para así comparar la performance entre ambas implementaciones, se requiere correr ambos experimentos al menos 10 veces para cada archivo de ”signal_N.txt” a ellos se les calculo el valor promedio del tiempo de ejecución y su desviación estándar.

A. Implementación de experimentos con Numpy

En esta sección, se solicita repetir los mismos experimentos que hicimos antes, pero ahora usando la librería Numpy. como ya se desarrolla de manera mas consistente en la primera parte del documento, para esta ocacion lo desarrollado se explicara de manera más breve que para el caso de Pytorch, porque las ecuaciones y conceptos son prácticamente los mismos. Lo único que cambia es la forma en que se implementan, ya que los métodos de Numpy son distintos. se estima que no es necesario repetir lo mismo otra vez, solo mostrar las diferencias clave. Primero que todo el ajuste mediante Numpy, este no tener una función como Pytorch que calcula los gradientes de manera eficiente, debemos realizarla ’Manual’ con una función de costo, la cual esta dada por la siguiente ecuación:

$$\sum (\hat{y} - y_{\text{datos}})^2 \tag{7}$$

Donde $\hat{y}$ son las predicciones, y y_{datos} los datos reales. Para después calcular los gradientes para ajustar los tres parámetros (a_1, a_2 y a_3) utilizando descenso de gradiente. Finalmente, los parámetros se actualizan restando el producto de la tasa de aprendizaje y sus gradientes respectivos para reducir el error, y se logran ajustar adecuadamente.

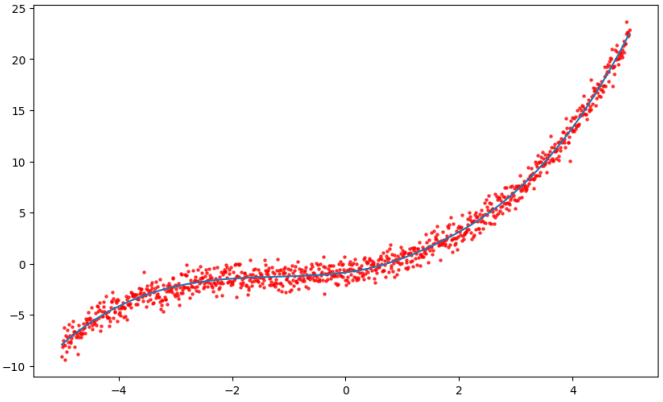

Fig. 4. Modelo ajustado con Numpy utilizando 1000 muestras

Por consiguiente, se realizaron los otros tres experimentos previamente ejecutados en Pytorch, utilizando las mismas funciones. Tal como se mencionó, únicamente cambiaron algunos métodos. Los resultados obtenidos fueron muy similares. Sin embargo, al usar Numpy, la velocidad de ejecución varió considerablemente en función del número de muestras, incrementando el tiempo conforme aumentaba el tamaño de los datos.

En este segmento del documento no se replicaran las gráficas anteriores aunque se haya modificado gran parte del código por el cambios de librerías utilizadas, esto es debido a que lo único que varia entre Numpy y Pytorch es el tiempo que se demora en ejecutarse cada código.

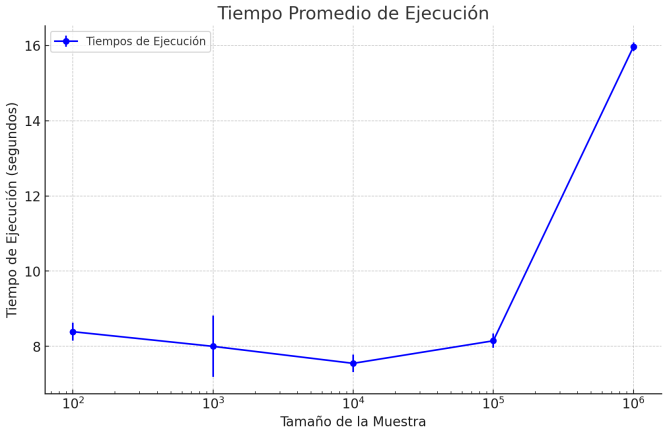

Fig. 5. Tiempo promedio de ejecución con Pytorch

Como se puede observar en la Imagen 5, PyTorch mostró tiempos de ejecución muy bajos y eficientes. Sin embargo, como era de esperarse, al aumentar el tamaño de las muestras, el tiempo de cálculo se incrementó ligeramente, aunque se mantuvo dentro de un rango razonable y eficiente.

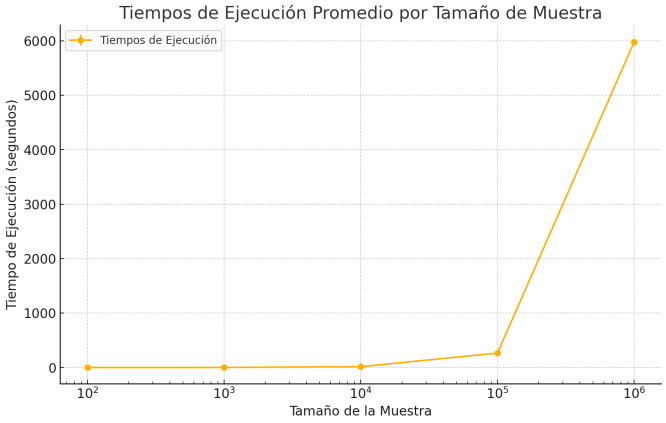

Fig. 6. Tiempo promedio de ejecución con Numpy

Al contrario, en la Imagen 6, donde se utilizaron Numpy/CPU, los tiempos de ejecución fueron considerablemente mayores, llegando a ser desproporcionados en comparación con PyTorch. No obstante, Numpy fue más eficiente al trabajar con señales de tamaño pequeño, resolviéndolas en un tiempo muchísimo menor que Pytorch. Sin embargo, a medida que el tamaño de las muestras aumentaba, los tiempos de ejecución de Numpy crecieron significativamente, volviéndose cada vez más lentos en realizar los cálculos para señales más grandes.

Cabe mencionar y esto, es muy importante para entender el por que de la demora de estos, en comparacion uno con otro. Observamos que las variables que influyen en la velocidad de los resultados obtenidos son principalmente tres: el número de épocas, el ratio de aprendizaje (learning rate) y la cantidad de muestras. Estas variables juegan un papel clave en el tiempo de ejecución y en la precisión del modelo.

Es importante destacar, y esto resulta fundamental para entender las diferencias en la demora entre distintos enfoques, que los ajustes del modelo no dependen únicamente de un parámetro. El número de muestras y el ratio de aprendizaje son cruciales, siendo este último especialmente relevante. Si se busca un modelo que entregue resultados rápidamente, aunque con menor precisión, se recomienda utilizar un ratio de aprendizaje bajo. En cambio, si se requiere un ajuste más preciso, se sugiere aumentar el ratio de aprendizaje, aunque esto inevitablemente incrementará el tiempo de ejecución.

Asimismo, los gráficos de la Imagen 5 y Imagen 6 reflejan estas variaciones, siendo claramente afectados por el ratio de aprendizaje. Por último, cabe mencionar que el uso exclusivo de Numpy en las iteraciones será siempre considerablemente más lento en comparación con Pytorch.

REFERENCES

[1] Hernan Mella, “GPU e intro a Pytorch”, Septiembre 2024.
[2] Método de Newton,” *Wikipedia, la enciclopedia libre*. Disponible en: https://es.wikipedia.org/wiki/Metodo_de_Newton.