Buscando simetrías en funciones mediante redes neuronales

Autores: Daniel Bocanegra: dflorezbo@unal.edu.co, Darío Penagos: dpenagosv@unal.edu.co

Introducción

En la física, un problema común es encontrar simetrías en datos correspondientes a algún fenómeno. Desafortunadamente, a menudo los datos disponibles no son suficientes para determinar si un fenómeno exhibe o no una simetría dada.

Una forma de resolver éste problema sería utilizando un ajuste de curvas: Dado un conjunto de datos del que queremos verificar la existencia de simetrías, podemos ajustar una función $\hat{f}$ a dichos datos, y verificar si $\hat{f}$ tiene o no las simetrías consideradas. Luego, si $\hat{f}$ exhibe una simetría dada, es probable que la simetría también esté presente en la función original.

En [1], S. Udrescu y M. Tegmark desarrollan un algoritmo de regresión simbólica que consiste de varios "módulos" donde uno de dichos módulos consiste en utilizar una red neuronal para aproximar los datos considerados e intentar encontrar una simetría en la red neuronal, la cual luego se utiliza restringir el rango de fórmulas que se pueden utilizar para aproximar los datos. En el presente texto, buscamos reproducir sus resultados. Es decir, entrenaremos redes neuronales con la misma arquitectura y datos para verificar si dichas redes neuronales exhiben alguna simetría que podría ser útil para obtener un mejor entendimiento del fenómeno físico considerado.

Metodología

Arquitectura

La arquitectura y método de entrenamiento se encuentran en el archivo `train.py`. Todas las redes neuronales utilizan 3 capas ocultas completamente conexas con dimensiones 128×128, 128×64, 64×64. Con función de activación tanh. Como optimizador, se utilizó `torch.optim.Adam`.

Datos

Los datos usados fueron generados sintéticamente de un conjunto de ecuaciones sacadas de [2]. Se determinaron 100.000 puntos arbitrarios en el dominio de la función considerada, y se computó el valor de la función para cada uno de éstos valores arbitrarios. Todas las funciónes son de la forma $\mathbb{R}^n \to \mathbb{R}$. Una descripción más detallada de éstas funciones se encuentra en el archivo `FeynmanEquations.csv`.

Algoritmo

El optimizador utilizado es `torch.optim.Adam` con un `batch_size` de 2048.

El valor de `lr` inicia en `1e-2`. Luego, se entrena una red neuronal por 1.000 épocas. Al final de éste proceso, el valor de `lr` se divide por 10. Dicho proceso se repite 4 veces. Si el algoritmo detecta que la red neuronal ha dejado de aprender con el valor de `lr` actual, símplemente interrumpe el proceso y sigue entrenando con el `lr` dividido por 10, como si hubiesen pasado las 1.000 épocas. En pseudocódigo:

```
for _ in range(4):
  check_loss = 10_000
  for epoch in range(1000):
    for (X,Y) in dataloader:
      optimizer.zero_grad()
      err = loss(model(X),Y)
      err.backward()
      optimizer.step()
    if epoch%20==0 and epoch>0:
      if check_loss<err:
        break
      else: check_loss = err
```

Reconocimiento simetrías

Comenzamos revisando manualmente las funciones y anotando cada una de las simetrías que cumplen en el archivo `Invariante_por_Funcion.csv`. En dicho archivo, se utilizan los siguientes strings para denotar las siguientes simetrías:

- `P1`: $f(-x) = -f(x)$
- `P0`: $f(-x) = f(x)$
- `H`: $f(tx) = t^k f(x)$
- `CCY1`: $f(x + a) = f(x) + f(a)$
- `CCY2`: $f(ax) = f(x)f(a)$
- `CCY3`: $f(x + a) = f(x)f(a)$
- `CCY4`: $f(ax) = f(x) + f(a)$
- `T`: $f(x + a) = f(x)$
- `I`: $f(x, y) = f(y, x)$
- `P`: $f(x^t) = f(x)$
- `ESC`: $f(cx) = cf(x)$

Por supuesto, la mayoría de éstas simetrías aplican a funciones de una sola variable, lo cual se contradice con el hecho de que la mayoría de las funciones que deseamos analizar tienen varias variables. En la práctica, símplemente consideramos si una función multivariada cumple éstas simetrías en uno de sus argumentos (dejando el resto de sus argumentos igual).

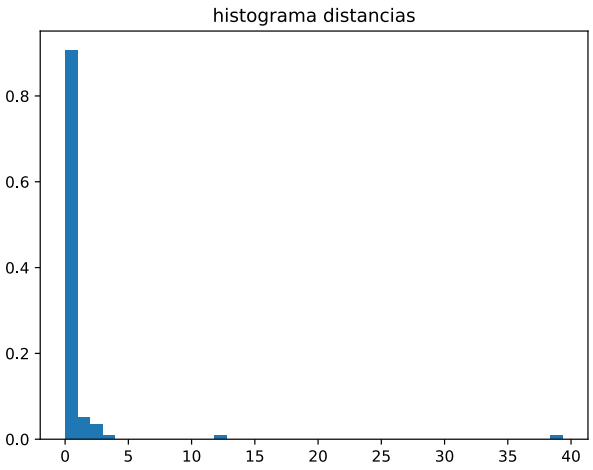
Ahora bien. Sea f una función que cumple una de éstas simetrías en alguna de sus componentes. Ésto significa que, para algún $k \in \mathbb{N}$ existe una transformación $\mathcal{L} : C^k \to C^k$ tal que $f = \mathcal{L}(f)$. Ahora bien, sea $\hat{f}$ la red neuronal que aproxima a f. Si $\hat{f}$ es una buena aproximación de f, es razonable esperar que $\hat{f} \approx \mathcal{L}\left(\hat{f}\right)$.

Por todo lo anterior, nuestra forma de revisar si una función f cumple una simetría dada, es computar $d = \left\| \hat{f} - \mathcal{L}\left(\hat{f}\right) \right\|_2$ y verificar si $d < \varepsilon$ para algún ε suficientemente pequeño.

En el archivo `reconocimiento_simetrías.py` computamos los valores $\left\| \hat{f} - \mathcal{L}\left(\hat{f}\right) \right\|_2$ para todas las simetrías que la función f (la función que $\hat{f}$ aproxima) cumple. Los resultados de éstos cómputos se encuentran en la carpeta `results`.

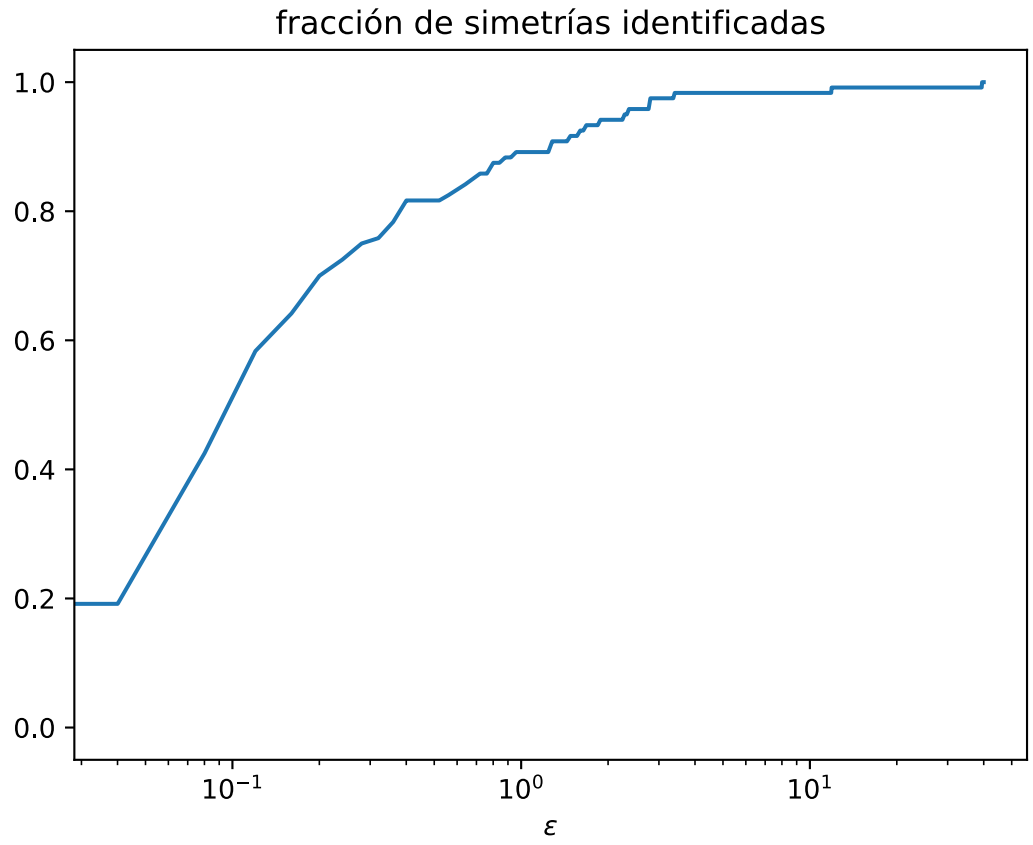
Análisis

El histograma de las distancias obtenidas se muestra a continuación:

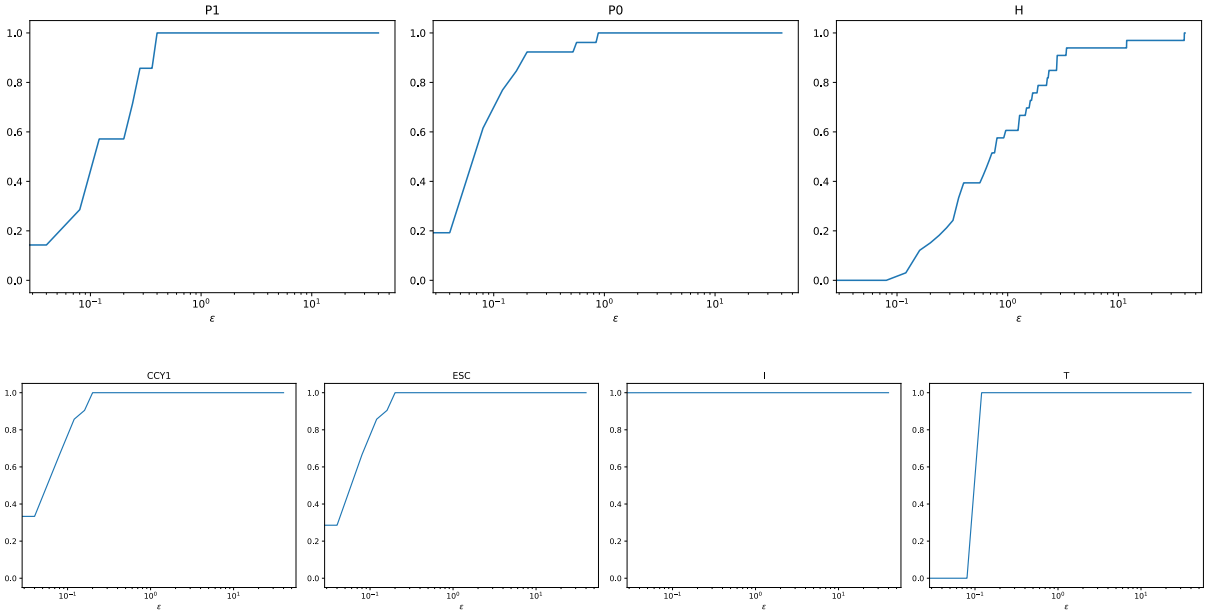


Aquí podemos ver que, en la mayoría de los casos (más del 80%), la distancia es bastante pequeña, un resultado positivo para nuestro método.

Por otro lado, podemos contar la fracción de simetrías identificadas por nuestro método en función de ε:



Si visualizamos la misma cantidad para cada simetría individualmente obtenemos los gráficos:



En las anteriores gráficas podemos ver que la simetría más difícil de detectar con éste método parece ser `H`, mientras que la más fácil es `I`, la cual puede ser detectada usando cási cualquier ε distinto de cero.

Conclusiones

Parece ser que, en el caso de las funciones y simetrías escogidas para éste ejercicio, el método propuesto es relativamente eficiente para encontrar la mayoría de simetrías, y de hecho, es capaz de encontrar el 89.166% de las simetrías utilizadas con $\varepsilon = 1$, un valor de épsilon relativamente pequeño.

Por supuesto, a medida que ε se hace más grande, se vuelve más y más fácil detectar simetrías (la taza de falsos negativos disminulle), pero también aumentará la tasa de falsos positivos. Debido a que, en éste texto, únicamente testeamos las tasas de falsos negativos, queda abierta la pregunta de si $\varepsilon = 1$ es realmente un valor suficientemente bajo para mantener razonablemente baja la tasa de falsos positivos.

Bibliography

[1] S. Udrescu and M. Tegmark, “AI Feynman: A physics-inspired method for symbolic regression.”

[2] R. Feynman and R. Leighton, “The Feynman Lectures on Physics: The New Millennium Edition.”