



# Aprendizaje automático: redes neuronales informadas por física aplicadas a sistemas con ciclos límite

**Jorge Faci Descartín**

Trabajo Fin de Grado en física

Director:
Ricardo López Ruiz

Septiembre 2024

# Índice

# 1. Introducción

El objetivo de este trabajo es mostrar una de las muchas parcelas de aplicabilidad a la física existentes en el aprendizaje automático, concretamente las llamadas Redes Neuronales Informadas por Física (o PINNs, por sus siglas en inglés Physics Informed Neural Networks).

Tras una breve introducción de carácter general al Machine Learning como un amplio campo de investigación se procede a una descripción más detallada del Aprendizaje Profundo (Deep Learning). A continuación se trata de convencer de la utilidad y del papel que prometen las PINNs, aún encontrándose lejos de su apogeo por su corto tiempo de desarrollo, en el *boom* de la inteligencia artificial; a la vez que de evidenciar las limitaciones actuales que se han encontrado durante la realización de esta tarea.

Para esta última parte se ha escogido el estudio de la efectividad de estas redes en sistemas con ciclos límite cuando aumentamos su no-linealidad. En el proceso se van aplicando distintas estrategias de aprendizaje y optimización, como el *time-marching* o la modificación de la arquitectura clásica del perceptrón multicapa, además de exponer la importancia de respetar la causalidad en las PINNs.

# 2. Aprendizaje automático

> *El aprendizaje automático es el campo de estudio que dota a los ordenadores de la capacidad de aprender sin estar explícitamente programados.*
>
> — *Arthur Samuel, 1959*

Así lo definió el informático Arthur Samuel [1], reconocido como uno de los responsables de la popularización del término. Su contemporáneo Frank Rosenblatt, psicólogo investigador, aunó su trabajo junto con modelos lógicos de células del cerebro para crear el hardware perceptrón Mark I, en esencia el primer ordenador que podía aprender en base a prueba y error, construido para reconocimiento de imágenes [2, 3].

En la década de los 60, unos prometedores avances en las redes neuronales artificiales (ANN) crearon altas expectativas que no fueron satisfechas en los años posteriores, lo que extendió el escepticismo entre los investigadores y su pronto abandono hasta entrados los años 90.

En este lapso de tiempo, la exploración de la inteligencia artificial dejó a un lado el aprendizaje automático para centrarse en resolver problemas prácticos, haciendo uso de herramientas teóricas y probabilísticas (como el método de k-vecinos más cercanos [4] o el algoritmo EM [5]). A principios de los años 80, a pesar de que algunas modernas arquitecturas reavivaron el interés, pronto el aprendizaje automático viró hacia técnicas que avanzaban más rápido y con mejores resultados (como los modelos Support Vector Machine (SVM) [6]), aislando al estudio de las redes neuronales a un nuevo campo llamado *conexionismo*.

El desarrollo paralelo del internet y de ordenadores más potentes permitió que el Machine Learning se consagrara como área de investigación independiente, y que con él emergieran campos como la minería de datos o la neurociencia computacional.

Actualmente estamos experimentando una nueva ola de interés en las ANNs. Sin embargo, hay algunos factores que predicen que el impacto en esta ocasión será más significativo [6]:

- Una cantidad de datos accesibles sin precedentes, que cuando se emplean para entrenar a las redes éstas superan a otras técnicas en problemas complejos.
- La potencia de los ordenadores actuales, estimulada por la industria de los videojuegos al fabricar potentes tarjetas gráficas, y la existencia de las plataformas en la nube.
- Optimización de los algoritmos de la década de los 90 con muy buenos resultados.
- El interés de los inversores que ayudan a sacar productos basados en ANNs, y que a su vez las publicitan y atraen nueva financiación.

# 3. Redes neuronales informadas por física (PINN)

En esta sección se trata de explicar los detalles técnicos más relevantes en el funcionamiento del perceptrón multicapa (MLP), base sobre la que se construyen las PINNs, para luego acotar las especialidades de este tipo de red [7].

## 3.1. Deep learning (DL)

En el aprendizaje profundo se busca replicar el funcionamiento elemental de las neuronas biológicas del tejido cerebral: las dendritas recogen una o varias señales (*input*), que son pesadas según su importancia y se activarán (o no) en función de si sobrepasan un umbral, emitiendo (o no) una nueva señal (*output*) a través del axón a nuevas neuronas.

Teniendo esto en cuenta, se puede definir una neurona matemática como una función que toma $n$ valores reales y los convierte en un número real a través de la composición de una transformación lineal con la acción de una función no lineal:

$$\begin{array}{ccccc} \mathbb{R}^n & \longrightarrow & \mathbb{R} & \longrightarrow & \mathbb{R} \\ \{x_n\} & \longrightarrow & z = \sum_{i=1}^{n} \omega_i x_i - b & \longrightarrow & a(z) \end{array}$$

En el trabajo del perceptrón original se usa la función de Heaviside como *a(z)*, llamada generalmente función de activación y que trata de imitar la respuesta desencadenante de las neuronas biológicas. A día de hoy, se ha comprobado que funciones suaves (como *tanh(z)* o *sinh(z)*) dan mejores resultados, a pesar de que no hay una solución general a la pregunta de cuál es la óptima para cada problema.

En la Figura 1 se observa la arquitectura básica del perceptrón multicapa. La información fluye de izquierda a derecha. La capa azul corresponde a la capa *input* $\vec{x}$ seguida de las llamadas capas *ocultas*, y terminando en la capa *output*. Esta última la forman los vectores $\vec{y}_{NN}(\vec{x}, \theta)$, donde $\theta = [W^k, b^k]$ son los pesos y sesgos de la red, con $1 \leq k \leq L$ siendo $L$ el número de capas ocultas. En los inicios de las redes neuronales, el uso de más de una capa oculta ya se consideraba profundo, no siendo algo extraño actualmente emplear varias decenas de ellas.

### 3.1.1. Función de pérdida

Existen redes que aprenden de manera no supervisada (por ejemplo en la búsqueda de estructuras o patrones en datos), semi-supervisada (con una cantidad determinada de datos etiquetados con la solución esperada) o incluso "por refuerzo" (a través del *feedback* que recibe, por prueba y error). Sin embargo, el uso más extendido es el aprendizaje supervisado, en el que se tiene acceso a los datos "reales" y para el entrenamiento se emplean

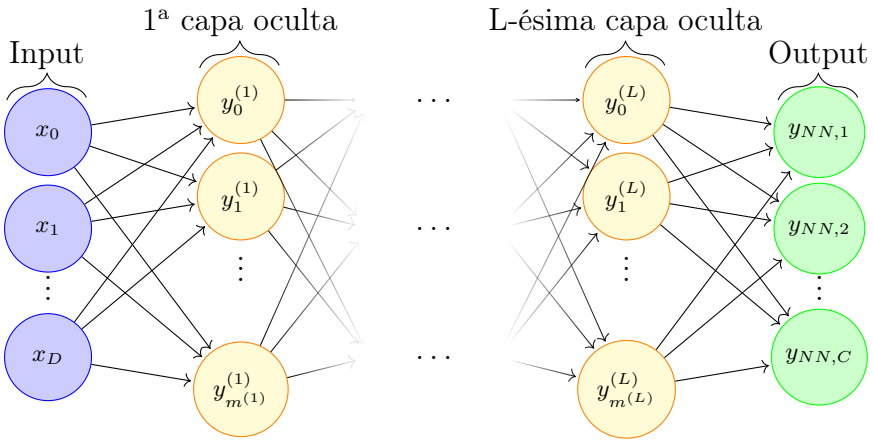



**FIG. 1:** Esquema de una red neuronal profunda (DNN)[8].

pares de datos de entrada y su solución ($\vec{x}$, $\vec{y}_{real}$).

En este último caso, se trata de minimizar la "distancia" entre entre las predicciones y los valores reales, idealmente cercana a cero, la cual se codifica en la llamada *función de pérdida*.
Hay una gama infinita de posibles funciones, pero la más común y sencilla es el error cuadrático medio (MSE) entre la predicción y los valores reales:

$$MSE = \frac{1}{n}\sum_i |\vec{y}_{real}(\vec{x}_i) - \vec{y}_{NN}(\vec{x}_i)|^2, \tag{3.1}$$

donde n es el número de puntos empleados.

### 3.1.2. Backpropagation y descenso de gradiente

El proceso de entrenamiento o aprendizaje se lleva a cabo habitualmente a través del método de *descenso de gradiente*, representado en la Figura 2. Tras inicializarse los pesos y sesgos de manera aleatoria, en cada paso éstos se actualizan conforme a las ecuaciones:

$$\omega_i^{l+1} = \omega_i^l - \eta\frac{\partial \mathcal{L}}{\partial \omega_i} \qquad\qquad b_i^{l+1} = b_i^l - \eta\frac{\partial \mathcal{L}}{\partial b_i}, \tag{3.2}$$

en las que $\mathcal{L}$ es la función de pérdida y $\eta$ es un hiperparámetro llamado *tasa de aprendizaje*.

El ejercicio mental más sencillo para imaginar este proceso es situarse en una montaña con niebla, en la que solo se nota la pendiente del suelo. La mejor estrategia para llegar al fondo del valle (los parámetros con la menor función de pérdida asociada) es seguir el camino con la pendiente descendente más pronunciada. Aquí es donde la tasa de aprendizaje $\eta$ juega un papel importante, ya que si es demasiado pequeña, el algoritmo tardará mucho en converger; y si es demasiado grande, puede saltarse el punto

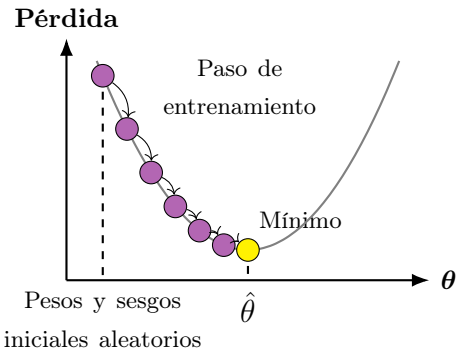



**FIG. 2:** Esquema general del descenso de gradiente [9].

deseado e incluso acabar en una peor situación que la inicial. Para facilitar la tarea, se emplean *optimizadores* que van adaptando $\eta$ en función de las magnitudes y pasos previos.

Se evidencia de esta manera que en las redes neuronales los hiperparámetros son de gran relevancia. En este grupo se incluyen todos los parámetros que pueden ser modificados manualmente: número de capas, número de neuronas por capa, número de épocas (de pasos en el descenso de gradiente), etc. Todos ellos deben ser elegidos en base a literatura previa o reglas heurísticas, al no haber unas fijas establecidas.

Queda explicar de qué manera se calculan los gradientes de la ecuación (3.2). Dado que realizar numéricamente todas las derivadas sería inabordable, una forma eficiente es mediante *retro-propagación* (*backpropagation* [6, 10]). Se trata de una técnica de descenso de gradiente en el que éstos se computan de manera automática: en dos pasos (uno hacia adelante y otro hacia atrás) es capaz de calcular el gradiente del error con respecto a cada parámetro del modelo, como muestra el esquema de la Figura 3. A cada ciclo compuesto por una ida y una vuelta se le denomina época. De esta forma, encuentra como los pesos y sesgos deben ser variados para reducir el error.

El algoritmo de *backpropagation* requiere de dos condiciones: que la función de pérdida se pueda expresar como el promedio de las pérdidas individuales de cada ejemplo de entrenamiento y como función de los outputs generados por la red. Ambas son satisfechas por la Ecuación (3.1).

## 3.2. Modelo informado por física

Las redes neuronales, como se comprueba en esta sección, son aproximadores universales de funciones. Una gran cantidad de procesos físicos se describen a través de ecuaciones diferenciales. Si a la función de pérdida se le incorpora tanto la ecuación diferencial como las condiciones iniciales/ de contorno, entonces se trata de una *red neuronal informada por física* (PINN) [11].

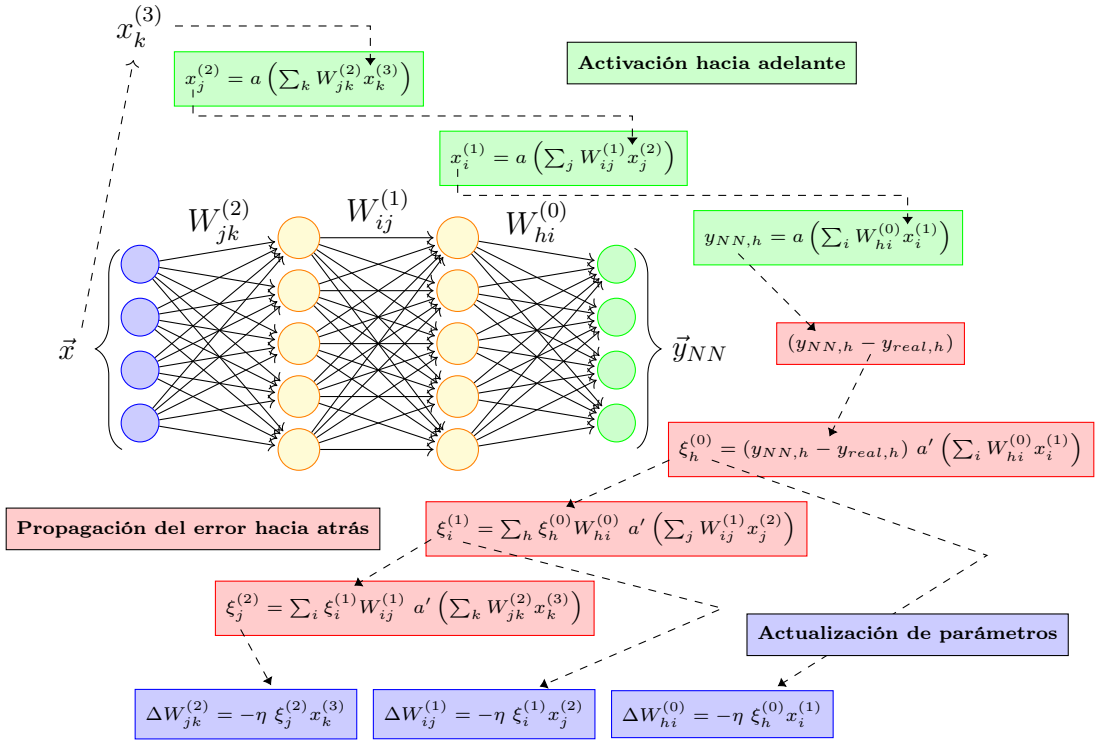



**FIG. 3:** Esquema de cómo aprende una red neuronal por retro-propagación del error, haciendo primero una predicción y calculando los errores de cada capa a través de la regla de la cadena, para posteriormente actualizar los pesos. Por simplicidad, en este ejemplo los sesgos $b^k$ están implícitos para todo k.

Una ecuación diferencial se puede expresar de forma general como:

$$y_t + \mathcal{N}[y] = 0, \quad t \in [0, T],\ x \in \Omega, \tag{3.3}$$

sujeta a condiciones iniciales y de contorno:

$$y(0, x) = g(x), \quad x \in \Omega, \tag{3.4}$$
$$\mathcal{B}[y] = 0, \quad t \in [0, T],\ x \in \partial\Omega, \tag{3.5}$$

donde $\mathcal{N}[\cdot]$ es un operador diferencial lineal o no lineal y $\mathcal{B}[\cdot]$ es un operador de contorno correspondiente a Dirichlet, Neumann, condiciones periódicas, etc. En el caso práctico que se abordará no habrá dependencia espacial luego este último no tendrá relevancia. La región $\Omega \subset \mathbb{R}^d$ contiene a los vectores $\vec{x} = \{x_1, \ldots, x_d\}$ donde $\partial\Omega$ es la frontera de $\Omega$.

Una red neuronal puede representar $\vec{x}$, $y(\vec{x})$ y las derivadas parciales de $y(\vec{x})$; y dependiendo de la capacidad que tenga para encontrar éstas últimas a través de la diferenciación automática hallará mejores o peores soluciones.

Siguiendo el trabajo original de Raissi *et.al* [11], la función de pérdida a minimizar constará de varios términos pesados:

$$\mathcal{L}(\theta) = \lambda_{ic}\mathcal{L}_{ic}(\theta) + \lambda_{bc}\mathcal{L}_{bc}(\theta) + \lambda_r\mathcal{L}_r(\theta), \tag{3.6}$$

donde:

$$\mathcal{L}_{ic}(\theta) = \frac{1}{N_{ic}}\sum_{i=1}^{N_{ic}} \left| y_{NN}(0, x_{ic}^i) - g(x_{ic}^i) \right|^2, \tag{3.7}$$

$$\mathcal{L}_{bc}(\theta) = \frac{1}{N_{bc}}\sum_{i=1}^{N_{bc}} \left| \mathcal{B}[y_{NN}](t_{bc}^i, x_{bc}^i) \right|^2, \tag{3.8}$$

$$\mathcal{L}_r(\theta) = \frac{1}{N_r}\sum_{i=1}^{N_r} \left| \frac{\partial y_{NN}}{\partial t}(t_r^i, x_r^i) + \mathcal{N}[y_{NN}](t_r^i, x_r^i) \right|^2. \tag{3.9}$$

Aquí $\{x_{ic}^i\}_{i=1}^{N_{ic}}$, $\{t_{bc}^i, x_{bc}^i\}_{i=1}^{N_{bc}}$ y $\{t_r^i, x_r^i\}_{i=1}^{N_r}$ pueden ser seleccionados como puntos de una malla fija o muestreados aleatoriamente en cada iteración de descenso de gradiente. Los hiperparámetros $\{\lambda_{ic}, \lambda_{bc}, \lambda_r\}$ permiten la flexibilidad de asignar diferente tasa de aprendizaje a cada término individual para balancear su interacción durante el entrenamiento.

Es importante resaltar que lo novedoso de estas redes parte de que en la función de pérdida no se introduce en ningún momento los valores $y_{real}$ para compararlos con $y_{NN}$, sino únicamente los puntos correspondientes a las condiciones iniciales (en este trabajo $y_{0,real}$ y la primera derivada $\dot{y}_{0,real}$). El resto de la información para ajustar los parámetros proviene meramente de la ecuación 3.9, es decir, del comportamiento físico del sistema. El error cuadrático medio entre la solución a la ecuación y la propuesta por la red se emplea por tanto como métrica para validar el entrenamiento y no para llevarlo a cabo.

# 4. Aplicación a sistemas de Liénard

Las *oscilaciones autosostenibles* son comunes en variedad de ramas de la ciencia (biología, química, dinámica de fluidos, etc.). Para lograr ese comportamiento, es necesaria la no-linealidad. Las oscilaciones se generan por un balance de amplificación y disipación incluso en ausencia de una fuerza periódica externa. Este estado dinámico puede ser modelado por *ciclos límite* estables encontrados en ecuaciones diferenciales no-lineales específicas.

Un ciclo límite es una trayectoria cerrada en el espacio de fase. Es estable si las soluciones vecinas tienden a él de forma asintótica. Un tipo de sistemas al que se dedicó mucha atención durante el siglo XIX es a los sistemas autónomos bidimensionales de la forma:

$$\begin{aligned} \dot{x} &= \mathcal{P}_n(x, y), \\ \dot{y} &= \mathcal{Q}_n(x, y), \end{aligned} \tag{4.1}$$

donde $\mathcal{P}_n$ y $\mathcal{Q}_n$ son polinomios de grado $n$ con coeficientes reales. El oscilador de Van der Pol $\ddot{x} + \mu(x^2 - 1)\dot{x} + x = 0$ es un ejemplo de sistema (4.1). En este caso, $\mathcal{P}_3(x, y) = y$ y $\mathcal{Q}_3(x, y) = -\mu(x^2 - 1)y - x$. Su comportamiento va de armónico ($\mu \to 0$) a oscilaciones de relajación cuando $\mu$ tiende a infinito ($\mu \to \infty$).

Una generalización de éste es la ecuación de Liénard:

$$\ddot{x} + \mu f(x)\dot{x} + x = 0, \tag{4.2}$$

siendo $\mu$ un parámetro real y $f(x)$ cualquier función real.

Reuniendo las pautas y recomendaciones de varios estudios previos se lleva acabo el análisis de un caso concreto, al que se le modificará el parámetro fundamental con el fin de comprobar la efectividad y limitaciones de las PINNs en regímenes alejados de la linealidad, paralelo a la explicación de la arquitectura y optimizaciones que se han ido aplicando. El programa se ha construido sobre la base del presentado en Navarro *et al.* [7].

## 4.1. Oscilador de Van der Pol

Este sistema de Liénard en el que se centrará el estudio corresponde a un valor de $f(x) = (x^{2n} - 1)$ en la ecuación (4.2) con $n = 1$.

La biblioteca fundamental en este trabajo es *Tensorflow*, de código abierto desarrollada y lanzada por Google en 2015, siendo una de las más populares en el aprendizaje automático. Consiste en un conjunto de librerías para operar con tensores además de permitir la ejecución de grafos computacionales. A través de la API de *Keras*, se pueden construir arquitecturas complejas para redes neuronales.

La parte principal del algoritmo es una clase personalizada. Consta de dos funciones: una en la que se incluyen algunos hiperparámetros y las condiciones iniciales, *set_ODE_param*; y otra que contiene el bucle de aprendizaje, *train_step*.

De este último, lo fundamental es el conjunto de *tf.GradientTape()* anidados, puesto que es donde Tensorflow realiza la diferenciación automática explicada previamente. Una vez halladas las derivadas, se calcula la función de pérdida a partir de tres contribuciones de *self.compiled_loss*, en este caso el error cuadrático medio (MSE), que devuelve el error en las condiciones iniciales de posición, velocidad y de la propia ecuación.

A continuación se hallan los gradientes de la función de pérdida con respecto a los parámetros del modelo a través del *tf.GradientTape()* superior y se aplica el optimizador *Adam* para actualizar los pesos y sesgos. Finalmente se actualizan las métricas (que son la pérdida y el MSE de la ecuación (3.1)) y son devueltas por pantalla. Se ha de notar que el MSE de la solución no aparece en la función de pérdida sino que se emplea como set de validación.

```
class ODE_2nd(tf.keras.Model):

    def set_ODE_param(self,mu,y0_exact,x0,dy_dx0_exact,Nt):
        self.Nt = Nt
        self.mu=mu
        self.x0=x0
        self.y0_exact=y0_exact
        self.dy_dx0_exact=dy_dx0_exact

    def train_step(self, data):
        # Training points and the analytical
        # (exact) solution at this points
        x, y_exact = data
        print(f'x shape: {x.shape}')

        Nt = self.Nt  # Batch size

        #Change initial conditions for the PINN
        mu = self.mu
        x0=tf.constant([self.x0], dtype=tf.float32)
        y0_exact=tf.constant([self.y0_exact], dtype=tf.float32)
        dy_dx0_exact=tf.constant([self.dy_dx0_exact], dtype=tf.float32)

        # Calculate the gradients and update weights and bias
        with tf.GradientTape() as tape:
            tape.watch(x)
            tape.watch(y_exact)
            tape.watch(x0)
            tape.watch(y0_exact)
            tape.watch(dy_dx0_exact)
            # Calculate the gradients dy2/dx2, dy/dx
            with tf.GradientTape() as tape0:
                    tape0.watch(x0)
                    y0_NN = self(x0,training=True)
                    tape0.watch(y0_NN)
            dy_dx0_NN = tape0.gradient(y0_NN, x0)
            with tf.GradientTape() as tape1:
                tape1.watch(x)
                with tf.GradientTape() as tape2:
```

```
                    tape2.watch(x)
                    y_NN = self(x,training=True)
                    print(f'y_NN: {y_NN}')
                    tape2.watch(y_NN)
                dy_dx_NN = tape2.gradient(y_NN, x)
                tape1.watch(y_NN)
                tape1.watch(dy_dx_NN)
            d2y_dx2_NN = tape1.gradient(dy_dx_NN, x)
            tape.watch(y_NN)
            tape.watch(dy_dx_NN)
            tape.watch(d2y_dx2_NN)

            #Loss= ODE+ boundary/initial conditions
            y0_exact=tf.reshape(y0_exact,shape=y_NN[0].shape)
            dy_dx0_exact=tf.reshape(dy_dx0_exact,shape=dy_dx0_NN.shape)

            loss =  self.compiled_loss(y0_NN, y0_exact) \
                  + self.compiled_loss(dy_dx0_NN, dy_dx0_exact)
                  + self.compiled_loss(d2y_dx2_NN, -y_NN + mu * tf.multiply(1 - tf.square
    (y_NN),dy_dx_NN)) \


        gradients = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(gradients, self.trainable_weights))
        self.compiled_metrics.update_state(y_exact, y_NN)
        return {m.name: m.result() for m in self.metrics}
```

Una vez definido el entrenamiento de la PINN, solo queda hacerlo propio con su arquitectura y entrenarla. Para ello:

- Se indican el hiperparámetro de no-linealidad *mu* ($\mu$), el paso temporal $h$, el tamaño del *batch* o lote, los límites temporales y las condiciones iniciales *VDP_init*.
- Para comparar las soluciones y rastrearlas, se emplea el método de *Runge-Kutta de orden 4* (RK4) (código en el Anexo) ya que, aunque para $\mu = 0$ sí, en general no existe solución analítica para el oscilador de Van der Pol.
- Se detalla la arquitectura de la red: una neurona de input, tres capas ocultas de 50 unidades cada una y conectadas completamente (*Fully Connected Network o FCN*) y otra neurona de output. Como función de activación entre capas se emplea la tangente hiperbólica. Los pesos se inicializan aleatoriamente.
- Finalmente se declara el modelo, los hiperparámetros comentados y se compila.

```
# Parameters
mu = 0
t0 = 0
t_end = 8
h = 0.4   #Temporal steps
batch_size = 1

VDP_init = np.array([1.0, 0.0]) #Initial conditions: y0, dy0_dt
t_true, VDP_true = runge_kutta_4(van_der_pol, t0, VDP_init, t_end, h=0.0001, mu=mu)

#Selection of training points desired
t_train = t_true[::int(h/0.0001)]
VDP_train = VDP_true[::int(h/0.0001)]
```

```
# Input and output neurons (from the data)
input_neurons  = 1
output_neurons = 1

# Hiperparameters
epochs = 500

# Definition of the model
initializer = tf.keras.initializers.GlorotUniform(seed=5)
activation='tanh'
input=Input(shape=(input_neurons,))
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(input)
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(x)
x=Dense(50, activation=activation,
            kernel_initializer=initializer)(x)
output = Dense(output_neurons,
               activation=None,
               kernel_initializer=initializer)(x)
model=ODE_2nd(input,output)


# Initial conditions and parameter of non-linearity
model.set_ODE_param(mu=mu,y0_exact=VDP_init[0],x0=t0,dy_dx0_exact=VDP_init[1],Nt=
    batch_size)

# Definition of the metrics, optimizer and loss
loss= tf.keras.losses.MeanSquaredError()
metrics=tf.keras.metrics.MeanSquaredError()
optimizer= Adam(learning_rate=0.001)

model.compile(loss=loss,
          optimizer=optimizer,
          metrics=[metrics])
model.summary()

history=model.fit(t_train, VDP_train[:,0], batch_size=batch_size, epochs=epochs,
                  callbacks=None)
```

### 4.1.1. Estudio de $\mu = 0$

Tras entrenar con únicamente $N_t = 21$ puntos equiespaciados entre $0 \leq t \leq 8$, se grafican las métricas. A la izquierda de la figura 4 se comprueba cómo disminuye la función de pérdida paralelamente al MSE, hasta llegar a un régimen en el que da saltos. Estos son posiblemente debidos al optimizador Adam que puede modificar la tasa de aprendizaje, inicialmente $\eta = 10^{-3}$, si el error parece no seguir disminuyendo [7].

A la derecha de la misma figura se solapan 15 modelos entrenados independientemente, en el que se evidencia que existe cierta aleatoriedad que se arrastrará al resultado final y que no se mejorará añadiendo más épocas de entrenamiento. Con el fin de paliar este comportamiento se añade un *callback* (véase el código en el Anexo), un umbral o límite que detiene el código cuando se llega a cierto MSE prefijado. En este caso, se ha decidido que frene al superar MSE $= 2 \cdot 10^{-4}$ (*threshold = 0.0002*). La figura 5 y la tabla 1 reflejan como se ha logrado no solo reducir el error en más de un orden de magnitud,

si no también el tiempo de ejecución, con solo cuatro modelos sobrepasando las 400 épocas.

Se procede a continuación a predecir soluciones a la ecuación 4.2 con ambos modelos recién entrenados. Para ello, se selecciona en el mismo intervalo de tiempo 8000 puntos nuevamente equiespaciados y desconocidos hasta entonces para la red, y se le pide que prediga $\vec{y}_{NN}$ con *model.predict(t)*.

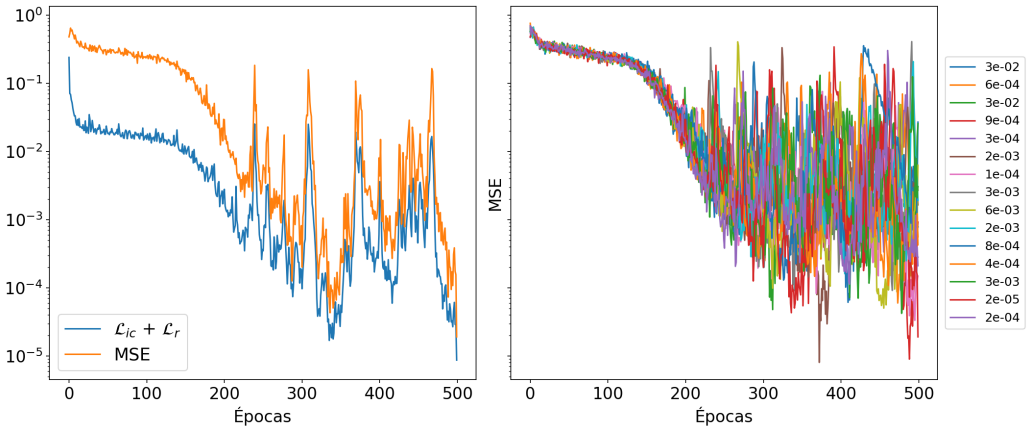



**FIG. 4: Izquierda**: Evolución de la función de pérdida y del error cuadrático medio para un modelo entrenado 500 épocas. **Derecha**: Evolución del error cuadrático medio para 15 modelos independientes, acompañados del valor final asociado a cada uno de ellos.

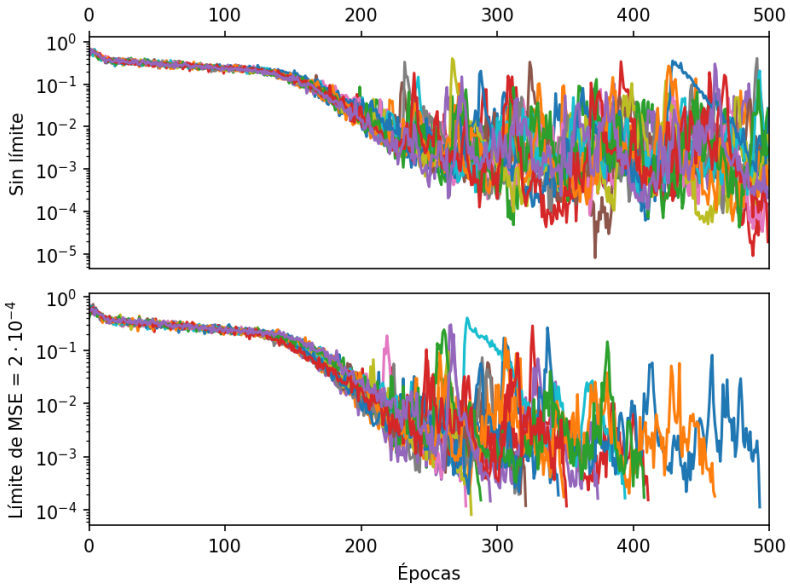



**FIG. 5: Arriba**: Evolución del error cuadrático medio para 15 modelos independientes sin umbral de detención. **Abajo**: Evolución del error cuadrático medio para 15 modelos independientes todos ellos con un umbral de detención de MSE $= 2 \cdot 10^{-4}$.

| $\mu = 0$ | Tiempo | Pérdida | MSE | Nº Épocas |
|---|---|---|---|---|
| **Sin umbral** | 1 min 20 s | $9{,}18e-04$ | $4{,}75e-03$ | 500 |
| **Con umbral** | 0 min 59 s | $7{,}22e-05$ | $1{,}51e-04$ | 358.53 |

**TAB. 1:** Tabla con los promedios de tiempo de entrenamiento, valor de pérdida final, error cuadrático medio final y número de épocas necesitado para $\mu = 0$ sin umbral de error y con.

```
VDP_init = np.array([1.0, 0.0]) #Initial conditions: y0, dy0_dt
t, VDP_exact = runge_kutta_4(van_der_pol, t0, VDP_init, t_end = 8, h = 0.0001, mu = mu)
y_exact=VDP_exact[:,0]
dy_dt_exact=VDP_exact[:,1]

y_NN=model.predict(t)

# The gradients (y'(x) and y''(x)) from the model
t_tf = tf.convert_to_tensor(t, dtype=tf.float32)
with tf.GradientTape(persistent=True) as g:
  g.watch(t_tf)
  with tf.GradientTape(persistent=True) as g2:
        g2.watch(t_tf)
        y = model(t_tf,training=True)
  dy_dt_NN = g2.gradient(y, t_tf)
d2y_dt2_NN = g.gradient(dy_dt_NN, t_tf)
```

La figura 6 muestra tanto la solución $\vec{y}_{NN}$ sobre los valores calculados por RK4 como el espacio de fases con sendas representaciones para dos ejecuciones arbitrarias con umbral de error y sin él. Se observa que como adelantaban las métricas, la aproximación mejora si se emplea el *callback*. La red es por tanto capaz de interpolar con una precisión muy llamativa a partir de tan solo $N_t = 21$, lo que supone una novedad y gran ventaja frente a otros métodos y resalta su gran eficiencia trabajando con sistemas lineales.

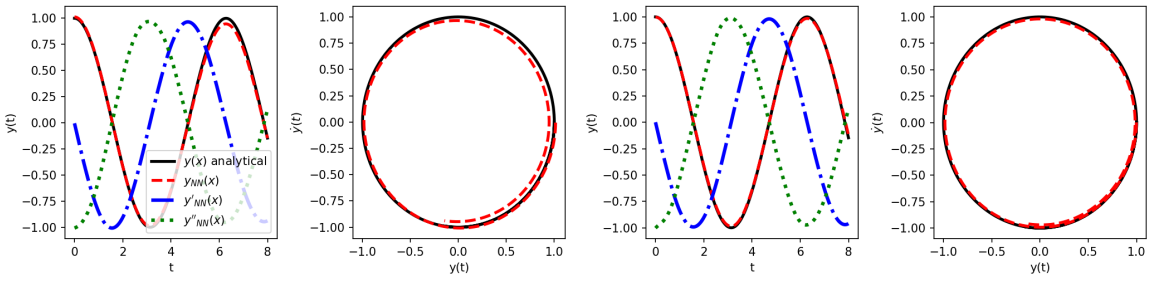



**FIG. 6: Izquierda**: predicción sin umbral de error de la solución, primera y segunda derivada para el oscilador de Van der Pol junto a la solución por RK4. **Centro-izquierda**: Representación en el espacio de fases de la predicción junto a la calculada por RK4. **Centro-derecha**: Ídem que en la figura de la izquierda, con umbral de error. **Derecha**: Ídem que en la figura del centro-izquierda, con umbral de error.

### 4.1.2. Estudio de $\mu = 0{,}75$

A partir de este punto, se abandona el régimen lineal y se estudiará el oscilador para $\mu > 0$. Dado que ahora la red se enfrenta a un problema más exigente, se aumenta la tasa de aprendizaje a $\eta = 3 \cdot 10^{-4}$ y se proponen 2000 épocas, manteniendo los mismos puntos de aprendizaje y mismo umbral. Se adaptan también las condiciones iniciales para saltarse la fase transitoria.

En la fila de arriba de la figura 7 se aprecia que efectivamente hay un largo periodo donde la pérdida no disminuye, para a continuación seguir un proceso de saltos parecido al caso lineal. El tiempo que tarda en converger ha aumentado por esta razón de forma considerable, con una media de 6 minutos por entrenamiento.

Para acelerar el proceso, un primera idea puede ser aumentar el tamaño de *batch*, es decir, entrenar paralelamente varios puntos de la red de forma simultánea. Sin embargo, al observar la fila de abajo de la misma figura 7, se ve que la red no aprende nada más que ligeramente la trayectoria aproximada de los primeros puntos de la misma.

La respuesta a este comportamiento es que cuando se trata de entrenar con un tamaño

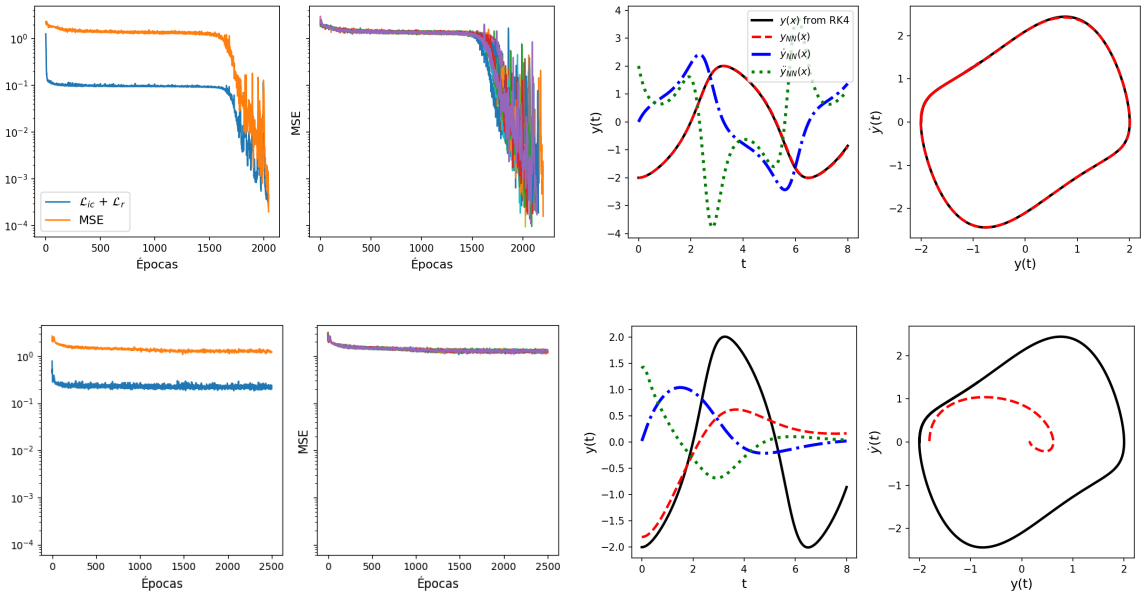



**FIG. 7: Izquierda**: Evolución de la función de pérdida y del error cuadrático medio para un modelo arbitrario bajo las condiciones descritas al inicio del apartado 4.1.2, arriba con *batchsize* = 1 y abajo con *batchsize* = 21. **Centro-izquierda**: Evolución del error cuadrático medio para 15 modelos independientes bajo las mismas condiciones. **Centro-derecha**: Predicción de la solución, primera y segunda derivada de un modelo arbitrario bajo las mismas condiciones. **Derecha**: Representación en el espacio de fases de la predicción junto a la calculada por RK4.

de lote temporal superior a la unidad, la función de pérdida no respeta la causalidad, como se explica en Wang *et al.* [12]. La demostración parte de la función de pérdida de la ecuación (3.9). Dado que en el caso del oscilador el número de puntos que entran a la red es $N_r = N_t$, la igualdad se puede reescribir de la forma

$$\mathcal{L}_r(\theta) = \frac{1}{N_t}\sum_{i=1}^{N_t}\mathcal{L}_r(t_i,\theta), \tag{4.3}$$

y si ahora se discretiza $\frac{\partial y_{NN}}{\partial t}$ con el método de Euler explícito, obviando los términos espaciales, se tiene que

$$\mathcal{L}_r(t_i,\theta) \approx \left|\frac{y_{NN}(t_i) - y_{NN}(t_{i-1})}{\Delta t} + \mathcal{N}[y_{NN}](t_i)\right|^2. \tag{4.4}$$

Se concluye por tanto que la minimización de $\mathcal{L}(t_i,\theta)$ depende de la correcta predicción de $y_{NN}(t_i)$ y $y_{NN}(t_{i-1})$, mientras que la formulación original trata de minimizar todos los $\mathcal{L}(t_i)$ simultáneamente, incluso cuando las predicciones en $t_i$ y tiempos anteriores no son precisas, violando la causalidad temporal.

La solución propuesta por Wang *et al.* [12] es una reformulación de la función de pérdida introduciendo pesos:

$$\mathcal{L}_r(\theta) = \frac{1}{N_t}\sum_{i=1}^{N_t} w_i\mathcal{L}_r(t_i,\theta), \tag{4.5}$$

Los pesos $w_i$ deben ser grandes si y solo si los residuos $\{\mathcal{L}_r(t_k,\theta)\}_{k=1}^{i}$ antes de $t_i$ son minimizados previamente. Esto se logra con

$$w_i = exp\left(-\epsilon\sum_{k=1}^{i-1}\mathcal{L}_r(t_k,\theta)\right),\ para\ i = 2,3\ldots,N_t, \tag{4.6}$$

donde $\epsilon$ se denomina *parámetro de causalidad*. Éste sirve para controlar la fuerza de los $w_i$, por lo que uno muy pequeño puede resultar contraproducente, pero uno grande puede suponer problemas de optimización. Una forma de soslayar esta cuestión es ir variándolo progresivamente como estrategia de *annealing*, usando una secuencia creciente $\{\epsilon_i\}_{i=1}^{k} = [10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}]$.

Con esto, la nueva expresión para la ecuación (3.9) pasa a ser

$$\mathcal{L}_r(\theta) = \frac{1}{N_t}\sum_{i=1}^{N_t} exp\left(-\epsilon\sum_{k=1}^{i-1}\mathcal{L}_r(t_k,\theta)\right)\mathcal{L}_r(t_i,\theta). \tag{4.7}$$

Se procede por tanto a hacer una modificación del código incluyendo esta discusión, que se

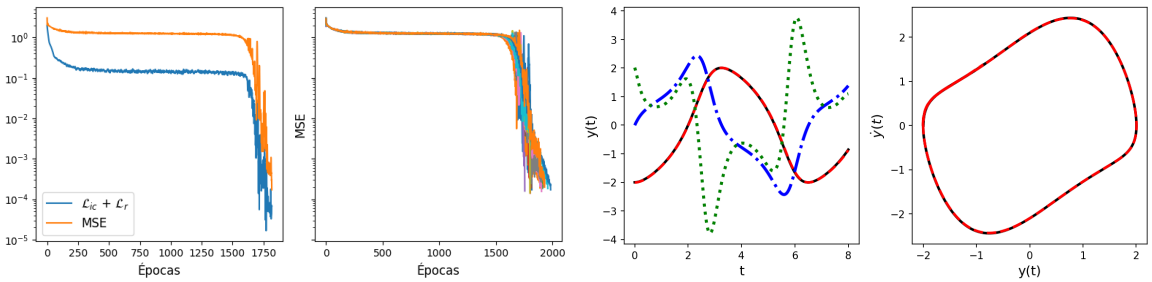



**FIG. 8:** Ídem que en la figura 7 respetando la causalidad, con $\eta = 0.001$ y un *batchsize* $= 7$.

puede leer en el Anexo. Los resultados que presenta la figura 8 demuestran la efectividad del método, logrando en un tiempo medio seis veces menor la misma precisión (véase la tabla 2). Nótese que el umbral aplicado frena al sistema ligeramente antes de alcanzar la fase de saltos habitual, indicando que se puede (y se debe) aumentar el valor de este para lograr una mayor exactitud. El entrenamiento no ha requerido tampoco de una tasa de aprendizaje mayor que en el caso lineal, empleándose por tanto $\eta = 10^{-1}$.

| $\mu = 0{,}75$ | Tiempo | Pérdida | MSE | Nº Épocas |
|---|---|---|---|---|
| **Original** | 6 min 4 s | $3{,}08e-04$ | $1{,}61e-04$ | 2082.93 |
| **Causal** | 1 min 1s | $4{,}38e-05$ | $1{,}75e-04$ | 1874.92 |

**TAB. 2:** Tabla con los promedios de tiempo de entrenamiento, valor de pérdida final, error cuadrático medio final y número de épocas necesitado con $\mu = 0.75$ para 15 redes entrenadas con *batchsize* $= 1$ y la formulación original y otras tantas con *batchsize* $= 7$ y respetando la causalidad.

### 4.1.3. Estudio de $\mu = 2{,}5$

Dada la eficacia de la red con residuos pesados para las oscilaciones cuasi-armónicas, se prueba a continuación con una $\mu$ mas elevada. Antes de comenzar con el análisis, es conveniente comparar la distribución de puntos a medida que se incrementa la no-linealidad.

En la figura 9 se muestra dicho reparto de posiciones que la red emplea para aplicar las restricciones físicas de la ecuación (4.7). Se puede ver que para un $\mu$ grande hay aceleraciones bruscas y por tanto derivadas fuertes, lo que genera regiones de la gráfica muy poco muestreadas en comparación a los casos lineales y cuasi-lineales. Por este motivo es necesario optimizar el aprendizaje y aumentar la frecuencia de puntos etiquetados.

Para ello se duplicará el número de puntos en $0 < t < 8$, pasando de $N_t = 21$ a $N_t = 42$ y un dato etiquetado cada $h = 0.2$. Además se emplea una nueva estrategia propuesta en Wight *et al.* [13], llamada *time marching*, y una nueva arquitectura de la red presentada en Wang *et al.* [14], denominada *perceptrón multicapa modificado* (*modified* MLP o mMLP).

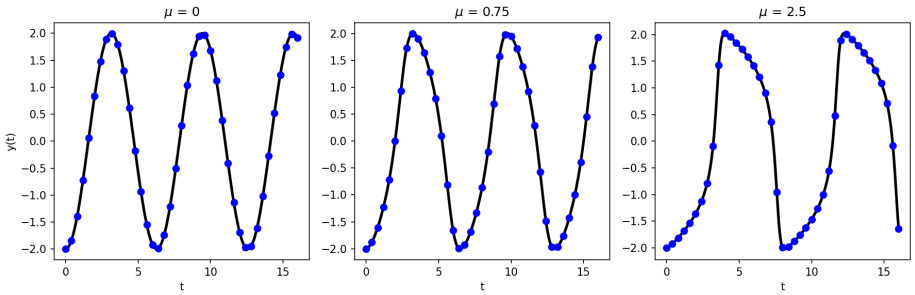



**FIG. 9:** Frecuencia de puntos $h = 0.4$ empleada en los casos $\mu = 0$ y $\mu = 0{,}75$ sobre la gráfica del oscilador de Van der Pol obtenido por el método RK4.

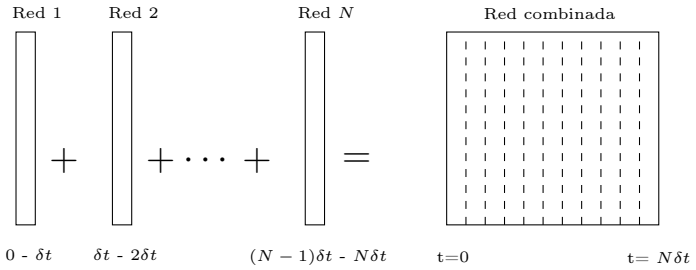



**FIG. 10:** Esquema del método de *time marching*

El método de *time marching*, esquematizado en la figura 10, consiste en segmentar el dominio de entrenamiento en intervalos más pequeños, empleando una red distinta para entrenar cada uno de ellos. Primero se entrena una red en $[0, \delta t]$, tras lo cual se predice la solución en $t = \delta t$ obteniendo así los valores que se emplearán como condiciones iniciales de la segunda red en $[\delta t, 2\delta t]$. Se repite el mismo proceso hasta que se ha cubierto el área deseada, para posteriormente predecir empleando cada modelo en la zona en la que se ha entrenado. En cuanto al perceptrón multicapa modificado, se diferencia de un MLP clásico en la introducción de dos transformaciones extra que proyectan las variables de entrada en un espacio de alta dimensionalidad, para a continuación actualizar a través de una multiplicación punto a punto las capas ocultas (véase el Anexo) [12, 14].

Se procede por tanto a implementar ambas optimizaciones para tratar el oscilador de Van der Pol con $\mu = 2.5$ en intervalos de $\delta t = 2$. Como las condiciones iniciales se predicen unas sobre otras, es importante que el error en éstas sea mínimo y no se arrastre exponencialmente. Por ello, se pone un peso adicional de $\lambda_{ic} = 10^3$ a $\mathcal{L}_{ic}$ (ecuación (3.7)) y se reduce el umbral de error del MSE a $threshold = 2e - 06$.

La figura 11 muestra un modelo de las características descritas. La evolución del MSE manifiesta con claridad los saltos en los que se cambia de red, dada la aleatoriedad inicial de los parámetros en cada una de ellas. También se aprecia que el tiempo de aprendizaje es mucho mayor en las dos regiones donde la derivada es pronunciada (azul), mientras

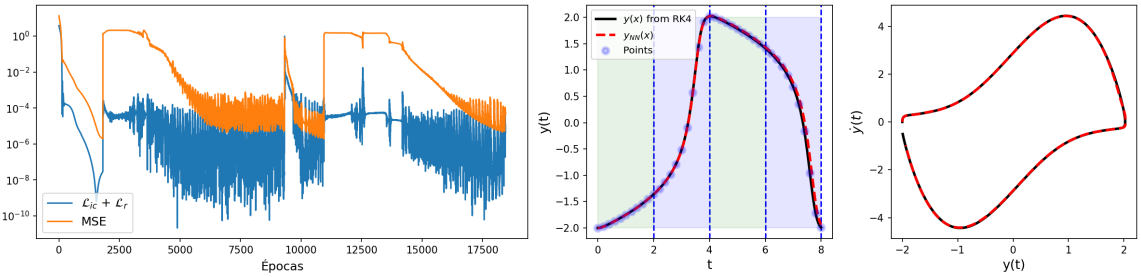



**FIG. 11: Izquierda**: Evolución de la función de pérdida y del error cuadrático medio para un modelo arbitrario empleando *time marching* y un mMLP. **Centro**: Predicción de la solución del mismo modelo. **Derecha**: Representación en el espacio de fases de la predicción junto a la calculada por RK4.

| $\mu = 2{,}5$ | Tiempo | Pérdida | MSE | Nº Épocas |
|---|---|---|---|---|
| **0 - 2** | 0 min 53s | $2{,}94e-08$ | $1{,}99e-06$ | 1817 |
| **2 - 4** | 3 min 42s | $3{,}22e-07$ | $7{,}47e-05$ | 7500 |
| **4 - 6** | 0 min 48s | $2{,}89e-05$ | $1{,}94e-06$ | 1626 |
| **6 - 8** | 3 min 40s | $4{,}38e-06$ | $5{,}50e-05$ | 7500 |

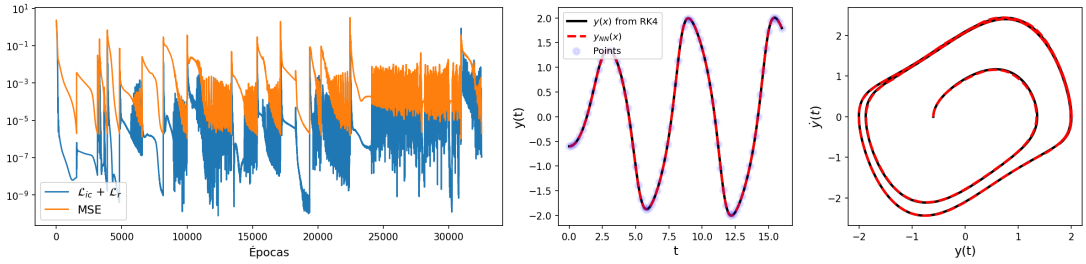
**TAB. 3:** Tabla con el tiempo de entrenamiento, valor de pérdida final, error cuadrático medio final y número de épocas necesitado para cada subintervalo temporal con *batchsize* = 11 en cada uno de ellos.

que alcanza rápidamente el umbral en las zonas con más densidad de puntos (verde). El código en el que se incluyen ambas optimizaciones se puede ver en el Anexo.

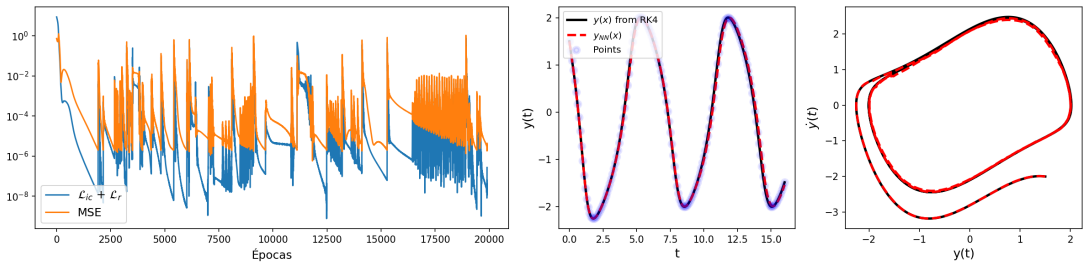
### 4.1.4. Estudio de la fase transitoria

Por último se realiza un breve análisis de cómo responde una PINN cuando las condiciones iniciales no son coordenadas del ciclo límite, si no que el sistema avanza hacia él. Se estudiará para los dos casos con $\mu > 0$ vistos, ya que para $\mu = 0$ las soluciones son oscilaciones periódicas de amplitud constante, y no existe un proceso de transición hacia un estado estable diferente.

En las figuras 12 y 13 y en la tabla 4 se aprecia como con la misma técnica que en el apartado anterior se puede conseguir hallar una solución con error bajo, aunque es necesario un número mucho más elevado de puntos y un mayor número de redes para captar las variaciones del sistema correctamente.

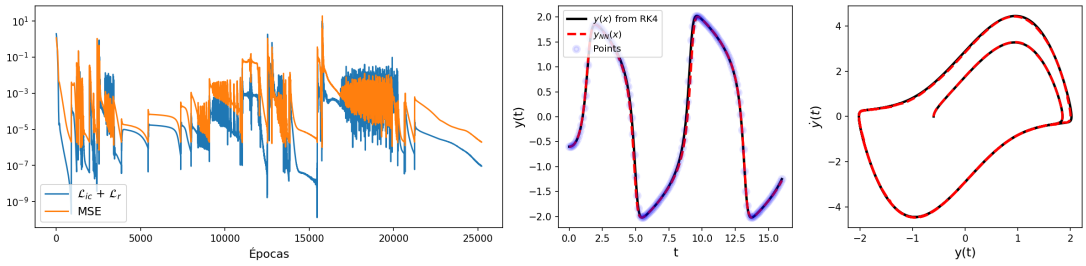



**(a)** $\mu = 0.75$ y condiciones iniciales $y_0$ = -0.6, $\dot{y}_0 = 0.0$

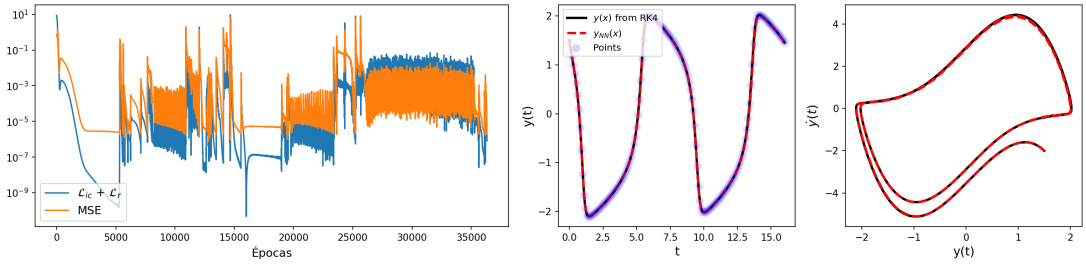



**(b)** $\mu = 0.75$ con condiciones iniciales $y_0 = 1.5$, $\dot{y}_0$ = -2.0

**FIG. 12: Izquierda**: Evolución de la función de pérdida y del error cuadrático medio para un modelo arbitrario empleando *time marching* y un mMLP. **Centro**: Predicción de la solución del mismo modelo. **Derecha**: Representación en el espacio de fases de la predicción junto a la calculada por RK4.




**(a)** $\mu = 2.5$ con condiciones iniciales $y_0$ = -0.6, $\dot{y}_0 = 0.0$




**(b)** $\mu = 2.5$ con condiciones iniciales $y_0 = 1.5$, $\dot{y}_0$ = -2.0

**FIG. 13: Izquierda**: Evolución de la función de pérdida y del error cuadrático medio para un modelo arbitrario empleando *time marching* y un mMLP. **Centro**: Predicción de la solución del mismo modelo. **Derecha**: Representación en el espacio de fases de la predicción junto a la calculada por RK4.

| | **Cond. Iniciales** | **Tiempo** | $\boldsymbol{\delta t}$ | **Nº de puntos** | **Tamaño de batch** |
|---|---|---|---|---|---|
| $\boldsymbol{\mu = 0{,}75}$ | $y_0 = -0{,}6$, $\dot{y}_0 = 0{,}0$ | 14 min 50s | 1 | 81 | 6 |
| $\boldsymbol{\mu = 0{,}75}$ | $y_0 = 1{,}5$, $\dot{y}_0 = -2{,}0$ | 13 min 32s | 0.5 | 161 | 6 |
| $\boldsymbol{\mu = 2{,}5}$ | $y_0 = -0{,}6$, $\dot{y}_0 = 0{,}0$ | 16 min 30s | 0.5 | 161 | 6 |
| $\boldsymbol{\mu = 2{,}5}$ | $y_0 = 1{,}5$, $\dot{y}_0 = -2{,}0$ | 19 min 49s | 0.5 | 161 | 6 |

**TAB. 4:** Tabla con los parámetros de los modelos representados en las figuras 12 y 13, en el mismo orden.

# 5. Conclusión

Durante la elaboración de este trabajo se ha podido probar una de las vías de utilidad del aprendizaje automático, la cual a pesar de contar con apenas unos años de investigación consigue resultados muy sobresalientes y promete jugar un papel importante en la modelización de sistemas físicos, sin pasar por alto que todavía trabaja en tiempos relativamente altos comparado con los algoritmos convencionales. Su gran ventaja frente a ellos es el reducido número de puntos requeridos para reproducir de una manera fidedigna las soluciones y que no necesita conocer las soluciones etiquetadas dentro del dominio, si no las ecuaciones que describen el sistema.

En este análisis no se ha tenido en cuenta la aplicación de las PINNs a sistemas con dependencia espacial $y(x,t)$, pero existen publicaciones con resultados igualmente satisfactorios empleando los mismos métodos o similares en áreas diversas como mecánica cuántica [15], termodinámica [16] o mecánica de fluidos [12, 17].

Para optimizar el desempeño en las zonas donde la no linealidad es extremadamente pronunciada, existen estudios que combinan estas redes con métodos clásicos, como en Lv *et al.* [18], donde emplean PINNs en las regiones suaves y métodos numéricos para captar las discontinuidades en las regiones que las contienen, en un algoritmo llamado *hybrid*-PINNs.

El abanico de posibilidades es realmente muy amplio y aún nos encontramos en el estadio inicial de la revolución, donde la heurística todavía impera, quedando en este aspecto lejos de métodos tradicionales los cuales a pesar de demandar más cantidad de información son a día de hoy más rápidos y directos, sin necesidad de hacer una manipulación previa más o menos extensiva de hiperparámetros. Pero como en todo aprendizaje en seres con redes neuronales biológicas, ya se trate de andar, nadar u agarrar objetos; los primeros pasos son siempre a base de prueba y error, por lo que la modelización de redes neuronales, quizá por ser inherente a ellas, debe pasar también por este proceso.

# Referencias


[1] Wikipedia contributors. *Arthur Samuel (computer scientist) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Arthur_Samuel_(computer_scientist)&oldid=1228604453. [Online; accessed 21-August-2024]. 2024.

[2] Keith D. Foote. *A Brief History of Machine Learning*. DATAVERSITY. [Online; accessed 21-August-2024]. 2021. URL: https://www.dataversity.net/a-brief-history-of-machine-learning/.

[3] Wikipedia. *Frank Rosenblatt — Wikipedia, La enciclopedia libre*. [Internet; descargado 20-septiembre-2023]. 2023. URL: \url{https://es.wikipedia.org/w/index.php?title=Frank_Rosenblatt&oldid=153874585}.

[4] Wikipedia. *K vecinos más próximos — Wikipedia, La enciclopedia libre*. [Internet; descargado 11-septiembre-2023]. 2023. URL: \url{https://es.wikipedia.org/w/index.php?title=K_vecinos_m%C3%A1s_pr%C3%B3ximos&oldid=153688741}.

[5] Wikipedia. *Algoritmo esperanza-maximización — Wikipedia, La enciclopedia libre*. [Internet; descargado 16-febrero-2024]. 2024. URL: \url{https://es.wikipedia.org/w/index.php?title=Algoritmo_esperanza-maximizaci%C3%B3n&oldid=158232439}.

[6] Aurèlien Gèron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2019.

[7] Luis Medrano Navarro, Luis Martín Moreno y Sergio G Rodrigo. *Solving differential equations with Deep Learning: a beginner's guide*. 2023. arXiv: 2307.11237 [physics.comp-ph]. URL: https://arxiv.org/abs/2307.11237.

[8] David Stutz. *Collection of LaTeX resources and examples*. https://github.com/davidstutz/latex-resources. Accessed on 24.08.2024.

[9] Laurent. *Replicating a plot using TikZ*. Accessed: 2024-08-26. 2020. URL: https://tex.stackexchange.com/questions/561921/replicating-a-plot-using-tikz.

[10] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. *Learning Internal Representations by Error Propagation*. Technical Report ADA164453. Defense Technical Information Center, 1985.

[11] Maziar Raissi, Paris Perdikaris y George E. Karniadakis. "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations". En: *CoRR* abs/1711.10561 (2017). arXiv: 1711.10561. URL: http://arxiv.org/abs/1711.10561.

[12] Sifan Wang, Shyam Sankaran y Paris Perdikaris. *Respecting causality is all you need for training physics-informed neural networks*. 2022. arXiv: 2203.07404 [cs.LG]. URL: https://arxiv.org/abs/2203.07404.

[13] Colby L. Wight y Jia Zhao. *Solving Allen-Cahn and Cahn-Hilliard Equations using the Adaptive Physics Informed Neural Networks*. 2020. arXiv: 2007.04542 [math.NA]. URL: https://arxiv.org/abs/2007.04542.

[14] Sifan Wang, Yujun Teng y Paris Perdikaris. *Understanding and mitigating gradient pathologies in physics-informed neural networks*. 2020. arXiv: 2001.04536 [cs.LG]. URL: https://arxiv.org/abs/2001.04536.

[15] Karan Shah et al. *Physics-Informed Neural Networks as Solvers for the Time-Dependent Schrödinger Equation*. 2022. arXiv: 2210.12522 [quant-ph]. URL: https://arxiv.org/abs/2210.12522.

[16] Brett Bowman et al. "Physics-Informed Neural Networks for the Heat Equation with Source Term under Various Boundary Conditions". En: *Algorithms* 16.9 (2023). ISSN: 1999-4893. DOI: 10.3390/a16090428. URL: https://www.mdpi.com/1999-4893/16/9/428.

[17] Hubert Baty y Léo Baty. "Solving differential equations using physics informed deep learning: a hand-on tutorial with benchmark tests". working paper or preprint. Abr. de 2023. URL: https://hal.science/hal-04002928.

[18] Chunyue Lv, Lei Wang y Chenming Xie. *A hybrid physics-informed neural network for nonlinear partial differential equation*. 2021. arXiv: 2112.01696 [math.NA]. URL: https://arxiv.org/abs/2112.01696.

# Anexos

# A. Optimizaciones del algoritmo

```
# Tensorflow Keras and rest of the packages
import tensorflow as tf
from tensorflow.keras.layers import Layer,Input,Dense
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
import json
import time

config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.compat.v1.Session(config=config)
```

**Listing 1:** Librerias empleadas.

```
def van_der_pol(t, Y, mu):
    y, dy_dt = Y
    d2y_dt2 = mu * (1 - y**2) * dy_dt - y
    return np.array([dy_dt, d2y_dt2])

def runge_kutta_4(f, t0, Y0, t_end, h, mu):
    t_values = np.arange(t0, t_end+0.001, h)
    Y_values = np.zeros((len(t_values), len(Y0)))
    Y_values[0] = Y0

    for i in range(1, len(t_values)):
        t = t_values[i-1]
        Y = Y_values[i-1]

        k1 = h*f(t, Y, mu)
        k2 = h*f(t + 0.5 * h, Y + 0.5 * k1, mu)
        k3 = h*f(t + 0.5 * h, Y + 0.5 * k2, mu)
        k4 = h*f(t + h, Y + k3, mu)

        Y_values[i] = Y + (k1 + 2*k2 + 2*k3 + k4)/ 6

    return t_values, Y_values
```

**Listing 2:** Método tradicional para hallar la solución del ocilador de Van der Pol (Runge Kutta de Orden 4).

```
class threshold_callback(tf.keras.callbacks.Callback):
    def __init__(self, threshold):
        super(threshold_callback, self).__init__()
        self.threshold = threshold

    def on_epoch_end(self, epoch, logs=None):
        if logs is None:
            logs = {}
        mse = logs.get('mean_squared_error', None)
        if mse is not None:
            if mse <= self.threshold:
                print(f'\nEpoch {epoch+1}: MSE reached threshold {self.threshold}.
    Stopping training.')
```

```
                self.model.stop_training = True
```

**Listing 3:** Callback para frenar el entrenamiento cuando se supera cierto umbral o *threshold*.

```
class ODE_2nd(tf.keras.Model):

    def set_ODE_param(self,mu,y0_exact,x0,dy_dx0_exact,Nt):
        self.Nt = Nt
        self.mu=mu
        self.x0=x0
        self.y0_exact=y0_exact
        self.dy_dx0_exact=dy_dx0_exact
        self.w = tf.Variable(tf.ones([Nt], dtype=tf.float32),trainable = False)  #
    Initialize w here

    def update_epsilon(self, eps):
        self.eps = eps

    def causal_loss(self, y_exact, y_net, dy_net, d2y_net, y0_net, y0_true, dy0_net,
    dy0_true):
        y_net = tf.squeeze(y_net)

        res_loss = tf.square(d2y_net + y_net - self.mu*tf.math.multiply(1-tf.math.square(
    y_net),dy_net))

        for i in range(2,self.Nt):
            self.w[i].assign(tf.exp((-self.eps*(tf.reduce_sum(res_loss[1:(i-1)],0)))))

        return 1000/self.Nt*self.compiled_loss((tf.multiply(tf.sqrt(self.w[0]),y0_net)),(
    tf.multiply(tf.sqrt(self.w[0]),y0_true))) + \
               1000/self.Nt*self.compiled_loss((tf.multiply(tf.sqrt(self.w[0]),dy0_net))
    ,(tf.multiply(tf.sqrt(self.w[0]),dy0_true))) + \
               1/self.Nt * tf.reduce_sum(tf.multiply(self.w,res_loss))

    def train_step(self, data):
        # Training points and the analytical
        # (exact) solution at this points
        x, y_exact = data
        print(f'x shape: {x.shape}')

        #Change initial conditions for the PINN
        x0=tf.constant([self.x0], dtype=tf.float32)
        y0_exact=tf.constant([self.y0_exact], dtype=tf.float32)
        dy_dx0_exact=tf.constant([self.dy_dx0_exact], dtype=tf.float32)

        # Calculate the gradients and update weights and bias
        with tf.GradientTape() as tape:
            tape.watch(x)
            tape.watch(y_exact)
            tape.watch(x0)
            tape.watch(y0_exact)
            tape.watch(dy_dx0_exact)
            # Calculate the gradients dy2/dx2, dy/dx
            with tf.GradientTape() as tape0:
                    tape0.watch(x0)
                    y0_NN = self(x0)
                    tape0.watch(y0_NN)
            dy_dx0_NN = tape0.gradient(y0_NN, x0)
            with tf.GradientTape() as tape1:
                tape1.watch(x)
                with tf.GradientTape() as tape2:
```

```
                    tape2.watch(x)
                    y_NN = self(x)
                    print(f'y_NN: {y_NN}')
                    tape2.watch(y_NN)
                dy_dx_NN = tape2.gradient(y_NN, x)
                print(f'dy_dx_NN: {dy_dx_NN}')
                tape1.watch(y_NN)
                tape1.watch(dy_dx_NN)
            d2y_dx2_NN = tape1.gradient(dy_dx_NN, x)
            print(f'd2y_dx2_NN: {d2y_dx2_NN}')
            tape.watch(y_NN)
            tape.watch(dy_dx_NN)
            tape.watch(d2y_dx2_NN)

            #Loss= ODE+ boundary/initial conditions
            y0_exact=tf.reshape(y0_exact,shape=y_NN[0].shape)
            dy_dx0_exact=tf.reshape(dy_dx0_exact,shape=dy_dx0_NN.shape)

            loss = self.causal_loss(y_exact,y_NN,dy_dx_NN,d2y_dx2_NN,y0_NN,y0_exact,
dy_dx0_NN,dy_dx0_exact)

        gradients = tape.gradient(loss, self.trainable_weights)

        # print("Gradients:")
        # for grad, var in zip(gradients, self.trainable_weights):
        #     print(f"Variable: {var.name}, Gradient: {grad}")

        self.optimizer.apply_gradients(zip(gradients, self.trainable_weights))

        # print("Trainable variables:")
        # for var in self.trainable_weights:
        #     print(var.name)

        self.compiled_metrics.update_state(y_exact, y_NN)
        return {m.name: m.result() for m in self.metrics}
```

**Listing 4:** Modelo que incluye la reformulación de la función de pérdida de la ecuación (4.7).

```
class ModifiedMLP(Layer):

def __init__(self,units,n_layers,activation='tanh', **kwargs):
    super(ModifiedMLP, self).__init__(**kwargs)
    self.L = n_layers
    self.units = units
    self.activation = tf.keras.activations.get(activation)
    self.hidden = [Dense(units,activation='tanh',kernel_initializer=tf.keras.
initializers.GlorotUniform(seed=5))
                   for _ in range(n_layers - 2)]
    self.residual1=Dense(units,activation='tanh',kernel_initializer=tf.keras.
initializers.GlorotUniform(seed=5),name = 'U_residual')
    self.residual2=Dense(units,activation='tanh',kernel_initializer=tf.keras.
initializers.GlorotUniform(seed=5),name = 'V_residual')
    self.initialH=Dense(units,activation='tanh',kernel_initializer=tf.keras.
initializers.GlorotUniform(seed=5), name = 'H_initial')
    self.initialZ=Dense(units,activation='tanh',kernel_initializer=tf.keras.
initializers.GlorotUniform(seed=5), name = 'Z_initial')

def call(self,X):
    U = self.residual1(X)
    V = self.residual2(X)
    H = self.initialH(X)
    Z = self.initialZ(H)
    H = tf.multiply(tf.ones_like(Z) - Z,U) + tf.multiply(Z,V)
```

```
        for l in self.hidden:
            Z = l(H)
            H = tf.multiply(tf.ones_like(Z) - Z,U) + tf.multiply(Z,V)

        return H

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])
```

**Listing 5:** Perceptrón multicapa modificado.

```
# Input and output neurons

input_neurons  = 1
output_neurons = 1

for _ in range(1):   #Number of examples we want to run
    start_time = time.time()
    initializer = tf.keras.initializers.GlorotUniform(seed=5)
    units = 50
    L = 5   #Number of hidden layers

    input=Input(shape=(input_neurons,))
    H = ModifiedMLP(units = units, n_layers = L)(input)
    output = Dense(output_neurons,
                   activation=None,
                   kernel_initializer=initializer)(H)
    model=ODE_2nd(input,output)

    # Definition of the metrics, optimizer and loss
    loss= tf.keras.losses.MeanSquaredError()
    metrics=tf.keras.metrics.MeanSquaredError()
    optimizer= Adam(learning_rate=0.001)

    callback = threshold_callback(2e-6)

    VDP_init = np.array([1.5, -2.0]) #Initial conditions: y0, dy_dt0
    epsilon = np.array([0.01, 0.1, 1, 10, 100])
    epochs = 2000

    mu = 0.75

    batch_size = 6
    h = 0.1 #Temporal steps
    dt = 0.5

    t0 = 0
    t_end = 16
    tmax = t0 + dt

    start_time = time.time()

    histories_inner = []

    while tmax <= t_end:
        t_true, VDP_true = runge_kutta_4(van_der_pol, t0, VDP_init, tmax, h=0.0001, mu=mu
    )

        model.set_ODE_param(mu=mu,y0_exact=VDP_init[0],x0=t0,dy_dx0_exact=VDP_init[1],Nt=
    batch_size)

        model.compile(loss=loss,      #We create a model per subdomain
```

```
                optimizer=optimizer,
                metrics=[metrics])
        model.summary()

        t_train = t_true[::int(h/0.0001)]   # Points desired in h size time windows
        VDP_train = VDP_true[::int(h/0.0001)]

        print(t_train)
        print(VDP_train)

        #print(t_train)
        #print(VDP_train)

        for eps in epsilon:
            print(f'eps={eps}')
            time.sleep(1)

            model.update_epsilon(eps=eps)
            history=model.fit(t_train, VDP_train[:,0], batch_size=batch_size, epochs=
    epochs,
                              callbacks=[callback])
            histories_inner.append(history.history)
            if model.stop_training:
                print(f'Training stopped because of callback at eps = {eps}')
                break

        model.save(f'C:/Users/facib/Desktop/TFG/VDP/callback/mu{mu}/{model.name}{tmax}',
    save_format='tf') #Save the model for later combination
        time.sleep(2)

        t, VDP_exact = runge_kutta_4(van_der_pol, t0, VDP_init, tmax, h=0.0001, mu=mu)
        y_exact=VDP_exact[:,0]
        dy_dt_exact=VDP_exact[:,1]

        y_step=model.predict(t)

        t_tf = tf.convert_to_tensor(t, dtype=tf.float32)

        with tf.GradientTape(persistent=True) as gt:
            gt.watch(t_tf)
            y = model(t_tf,training=True)
        dy_dt_step = gt.gradient(y, t_tf)
        dy_dt_step = dy_dt_step.numpy()

        t0 = tmax
        tmax += dt

        VDP_init = np.array([y_step[-1,0], dy_dt_step[-1]])
        histories.append(histories_inner)
    end_time = time.time()
    times.append(end_time - start_time)
```

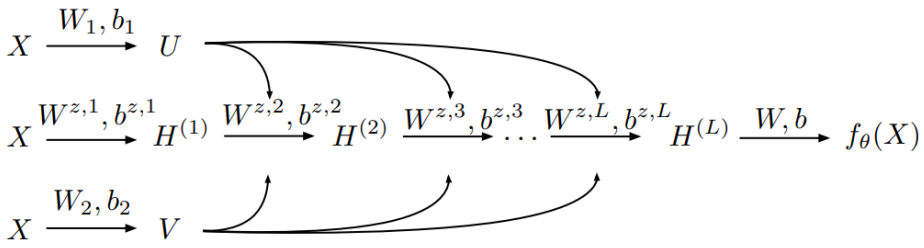
**Listing 6:** Entrenamiento empleando time marching y mMLP.

```
n_models = 32  #Number of subdomains
h = 0.0001
mu = 0.75
t0= 0.0
t_end = 16.0
all_models = []
tmax= 0.5
for _ in range(n_models):
```

```
    model = tf.keras.models.load_model(f'C:/Users/facib/Desktop/TFG/VDP/callback/mu{mu}/{
    model.name}{tmax}')
    all_models.append(model)
    tmax+= 0.5

VDP_init = np.array([1.5, -2.0]) #Initial conditions: y0, dy0_dt

t, VDP_exact = runge_kutta_4(van_der_pol, t0, VDP_init, t_end, h, mu)
y_exact=VDP_exact[:,0]
dy_dt_exact=VDP_exact[:,1]

y_NN = []
dy_dt_NN = []
d2y_dt2_NN = []

tmax = 0.5

for model in all_models:
   t = np.arange(t0,tmax,h)
   y_NN.append(model.predict(t))

   t_tf = tf.convert_to_tensor(t, dtype=tf.float32)
   with tf.GradientTape(persistent=True) as g:
       g.watch(t_tf)
       with tf.GradientTape(persistent=True) as g2:
          g2.watch(t_tf)
          y = model(t_tf,training=False)
       dy_dt_NN.append(g2.gradient(y, t_tf))
   d2y_dt2_NN.append(g.gradient(dy_dt_NN, t_tf))

   t0 = tmax
   tmax+= 0.5

t=np.arange(0,t_end,h)
y_NN = np.array(y_NN)
y_NN = y_NN.reshape(-1)

dy_dt_NN = np.array(dy_dt_NN)
dy_dt_NN = dy_dt_NN.reshape(-1)

d2y_dt2_NN = np.array(d2y_dt2_NN)
d2y_dt2_NN = d2y_dt2_NN.reshape(-1)
```

**Listing 7:** Predicción empleando time marching.

# B. Estructura del perceptrón multicapa modificado




$$U = \phi(XW^1 + b^1), \quad V = \phi(XW^2 + b^2)$$
$$H^{(1)} = \phi(XW^{z,1} + b^{z,1})$$
$$Z^{(k)} = \phi(H^{(k)}W^{z,k} + b^{z,k}), \quad k = 1, \ldots, L$$
$$H^{(k+1)} = (1 - Z^{(k)}) \odot U + Z^{(k)} \odot V, \quad k = 1, \ldots, L$$
$$f_\theta(x) = H^{(L+1)}W + b$$

**FIG. 14:** Estructura del perceptrón multicapa modificado, incluyendo dos transformaciones extras que aumentan la dimensionalidad del espacio de características [14].

# C. Software y hardware

## C.1. Software

Todos las simulaciones y gráficos han sido escritos y generados con las librerías mencionadas de Python.

## C.2. Hardware

Todo el trabajo ha sido llevado a cabo con un procesador AMD Ryzen 5 5000 series y una tarjeta gráfica de NVIDIA GEFORCE GTX de dos núcleos con 8 GB de memoria RAM cada uno.