Universitat de les Illes Balears

Facultat de Ciències

Memòria del Treball de Fi de Grau

Estudi sobre la millora dels algoritmes pel càlcul d'interaccions hidrodinàmiques mitjançant tècniques d'aprenentatge automàtic (Machine Learning).

Leonardo Prieto Yagi

Grau de Física

Any acadèmic 2018-19

Treball tutelat per Joan Josep Cerdà Pino
Departament de Física

S'autoritza la Universitat a incloure aquest treball en el Repositori Institucional per a la seva consulta en accés obert i difusió en línia, amb finalitats exclusivament acadèmiques i d'investigació	Autor		Tutor	
	Sí	No	Sí	No
		X		X

Paraules clau del treball:
Hidrodinàmica computacional, smoothed particle hydrodynamics, deep learning.

Índice

Resumen

En este trabajo se explora la aplicación de las técnicas modernas de aprendizaje automático (*Machine Learning)* en el ámbito de las simulaciones físicas de fluidos. Se escoge como modelo de simulación el denominado *Smoothed-particle hydrodynamics* (SPH), método Lagrangiano aplicado con éxito en varios campos de la física. Se busca disminuir el tiempo de cálculo total usando redes neuronales artificiales, mediante reemplazo de las partes que imponen más carga, vigilando al mismo tiempo el error que se introduce. Se concluye que los errores numéricos son aceptables pero su uso no es capaz de acelerar los cálculos.

Introducción

Gracias a los activos avances en computación y debido a la necesidad de analizar y experimentar con la creciente cantidad de datos y teorías, actualmente el uso de esta herramienta en física, así como en gran parte del ámbito científico, es ineludible. Desde las primeras simulaciones físicas realizadas a mediados del siglo XX, la capacidad y la precisión de cálculo de las máquinas han aumentado enormemente, facilitando el análisis de datos así como la predicción de fenómenos complejos y encaminando a la mayoría de campos de la ciencia hacia nuevos conocimientos. Con la gran explosión evolutiva de la ciencia computacional, se invierten muchos recursos en mejorar la eficiencia de procesamiento con el objetivo de obtener resultados más complejos y precisos en un tiempo menor.

Muchas veces la mejora necesaria para conseguir los resultados deseados es de carácter físico (*hardware*) y hay que recurrir al desarrollo de nuevos equipos que soporten la elevada carga computacional requerida. No obstante, el estudio de nuevas técnicas y algoritmos capaces de procesar datos de forma más eficiente es también necesario. Es notable en este campo de estudio la aparición de las denominadas redes neuronales artificiales. Desde los albores de la computación, se ha mostrado interés por modelar las interacciones de las neuronas en el cerebro animal [1]. Este interés se ha ido desarrollando en períodos de mayor actividad que otros, siendo las dos últimas décadas una etapa de gran avance, lo que ha hecho que el conjunto de herramientas conocidas actualmente como *machine learning* sean de uso común en análisis y predicción de datos.

Paralelamente, la física de fluidos resultó ser también uno de los primeros campos donde se aplicaron métodos numéricos [2]. Debido a la naturaleza compleja de los problemas de dinámica, fue necesario desarrollar modelos de resolución compatibles con el cálculo computacional. Hoy en día hay varios modelos que se usan con éxito en diversos campos como astrofísica, meteorología, oceanografía e ingenierías varias. Sigue siendo actualmente un tema importante de estudio, primero porque la resolución analítica a las ecuaciones de Navier-Stokes no ha sido hallada debido a la no linealidad del problema y segundo porque su resolución numérica implica la interacción de una vasta cantidad de partículas. En el segundo panorama, además, errores a pequeñas escalas pueden afectar fatalmente a mayores escalas, dificultando aún más el tratamiento numérico, el cual induce inevitablemente errores. Es necesario por tanto una elevada potencia de cálculo si se desea obtener resultados precisos.

Es justamente sobre este tipo de obstáculos en el tratamiento numérico que puede resultar de interés la aplicación de las nuevas técnicas de *machine learning* para reducir la carga computacional y mejorar la eficiencia de los cálculos. Se pueden abordar diversos enfoques en cuanto al uso de las redes neuronales artificiales, pero en este trabajo se explora la posibilidad de que "la máquina aprenda" a partir de los datos que se pueden generar a partir del propio modelo.

1. Dinámica de fluidos computacional y formulación de SPH

En el estudio de la dinámica de los fluidos, la descripción de un fluido viscoso en movimiento viene dada por el conjunto de ecuaciones diferenciales parciales conocidas como ecuaciones de Navier-Stokes. Éstas tienen como base la conservación de masa y momento, introduciendo un término de incompresibilidad y uno de disipación (viscosidad). Las ecuaciones, juntamente con la descripción del material o medio, las condiciones de contorno y las condiciones iniciales determinan completamente el comportamiento del sistema. Sin embargo, la obtención de la solución analítica resulta muy compleja exceptuando algunos casos simplificados. Es por ello que la resolución numérica es muy frecuente en la física de fluidos, donde se discretizan las integrales y derivadas, tomando formas más simples que la computadora es capaz de calcular.

La evolución temporal de un fluido se puede enmarcar en dos maneras diferentes de describir el mismo flujo. Las ecuaciones en cada marco toman una forma distinta, sin embargo están relacionadas entre sí y describen el mismo fenómeno. Tales especificaciones se ven reflejadas en los modelos, que basan sus algoritmos de cálculo en las expresiones de uno u otro marco.

Una de ellas es la descripción euleriana del flujo, que sigue la evolución de éste en el tiempo en determinados puntos fijos del espacio. Computacionalmente, el espacio se discretiza en forma de malla fija, en los nodos de la cual se realizan los cálculos y se obtienen las cantidades deseadas. Esto permite obtener la evolución del sistema usando diferencias finitas en los puntos fijos. Con el paso del tiempo, las propiedades del flujo varían en un nodo a medida que un elemento de fluido lo atraviesa. De esta manera, las propiedades se representan en función de la posición y el tiempo. La velocidad de flujo, por ejemplo, será $\mathbf{u}(\mathbf{r}, t)$.

La otra es la descripción lagrangiana, donde se *rastrean* pequeñas parcelas del fluido. Las propiedades del flujo son transportadas a través del espacio por medio de las parcelas, que se consideran pequeñas partículas del fluido con sus propias características. A la hora de implementar métodos lagrangianos computacionalmente, no es necesario discretizar el espacio reticularmente, sino que cada partícula posee una posición continua en el espacio, así como las otras cantidades de interés. Esto hace que la evolución se convierta en un problema de interacciones entre partículas.

La relación entre ambos marcos se recoge en la definición de derivada material (o total), compuesta por la suma de la derivada local y la derivada convectiva:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla \tag{1.1}$$

La expresión indica que la variación de las propiedades siguiendo al flujo (derivada total) puede ser debida a la propia fluctuación de las propiedades en el punto fijo en que se encuentre (derivada local) o a que el elemento de fluido se esté desplazando hacia otro punto donde las propiedades son distintas (derivada convectiva).

Los métodos eulerianos están bien consolidados y establecidos en el ámbito de las simulaciones de fluidos, sin embargo están limitados al refinamiento de la malla estática y no son adecuados para tratar problemas con superficies libres, geometrías irregulares o seguir cantidades en puntos concretos del material o medio simulado. Los modelos lagrangianos sin malla son más apropiados para paliar con este tipo de planteamientos más generales. Un modelo de este último tipo usado en varios ámbitos es el de *Smoothed-particle hydrodynamics* o simplemente SPH. Éste es el modelo que se usa en el trabajo, siendo sencillo de implementar y potente.

El modelo SPH fue concebido inicialmente por Gingold, Monaghan y Lucy el año 1977 para resolver problemas de astrofísica en espacios abiertos tridimensionales [3], cuyo movimiento conjunto de las partículas es similar al movimiento de un líquido o gas y se puede modelar por las ecuaciones de hidrodinámica clásica. En la actualidad, el método SPH se usa en áreas de astrofísica como pueden ser simulaciones de colisiones estelares y de estrellas binarias, supernovas, colapso y formación de galaxias, entre otros. También se aplica a una vasta extensión de problemas de mecánica de sólidos y fluidos por la habilidad del modelo de incorporar fenómenos físicos complicados en su formulación. En dinámica de fluidos, por listar algunos ejemplos dentro de una gran variedad de problemas, se ha aplicado con éxito en flujos elásticos, magnetohidrodinámica, flujos multifásicos o flujos incompresibles. Todas estas aplicaciones han sido posibles gracias a numerosas contribuciones que han permitido extender el modelo en cuanto a adaptabilidad y precisión.

En SPH, el dominio del fluido se representa por un conjunto arbitrariamente distribuido de partículas que representan pequeñas parcelas del fluido total. La partículas no poseen una interconexión predefinida, como pudiera ser en el caso de métodos que usan mallas. Cada una posee propiedades materiales y las transporta a lo largo del flujo: el movimiento del fluido es el movimiento de las partículas. La interacción entre las partículas viene dada mediante la integración de una función *kernel* dependiente de la distancia, la cual queda discretizada mediante la *aproximacón de partícula* en forma de sumatorio e interpola las propiedades entre las partículas vecinas. A cada paso temporal se realizan las interpolaciones y se integran las propiedades para avanzar al siguiente paso.

1.1. Representación integral

El concepto básico del modelo es la *representación integral*, denominada también *aproximación por kernel* de funciones. La idea parte de la simple identidad:

$$f(\mathbf{r}) = \int_{\Omega} f(\mathbf{r'})\delta(\mathbf{r} - \mathbf{r'})d\vec{\mathbf{r}} \tag{1.2}$$

donde f es una función escalar que depende de la posición y $\delta(\mathbf{r} - \mathbf{r'})$ es la delta de Dirac. El volumen sobre el que está definida la función y donde se encuentra $\mathbf{r}$ se define como Ω. Siempre y cuando $f(\mathbf{r})$ esté bien definida y sea continua en Ω, la representación integral de la ecuación (1.2) es exacta. Se puede reemplazar la delta de Dirac por una función

de suavizado (*smoothing function*) denominada *kernel* $W(\mathbf{r}-\mathbf{r'},h)$, con una distancia característica h, que cumpla las siguientes propiedades

$$\lim_{h\to 0} W(\mathbf{r}-\mathbf{r'},h) = \delta(\mathbf{r}-\mathbf{r'}) \tag{1.3}$$

$$\int_\Omega W(\mathbf{r}-\mathbf{r'},h)d\mathbf{r'} = 1 \tag{1.4}$$

es decir, que cuando la distancia característica tienda a cero el kernel se vuelva una delta de Dirac y que esté normalizada, respectivamente. De esta manera, la representación integral de la función se vuelve

$$f(\mathbf{r}) := \int_\Omega f(\mathbf{r'})W(\mathbf{r}-\mathbf{r'},h)d\mathbf{r'} \tag{1.5}$$

Desde luego, mientras el kernel no sea la delta de Dirac, la representación mediante la función de suavizado sólo puede ser una aproximación.

La función kernel también debe cumplir una condición de compacidad:

$$W(\mathbf{r}-\mathbf{r'},h) = 0 \quad \text{cuando} \quad |\mathbf{r}-\mathbf{r'}| > \kappa h \tag{1.6}$$

donde κ es una constante que define el área de efectividad (no nula) del kernel en torno a un punto. Este área efectiva se denomina el dominio de soporte para el kernel en el punto $\mathbf{r}$. De esta manera, la integral sobre todo el dominio es realmente una integración sobre el dominio de soporte Ω.

La función kernel se suele tomar par. Se puede demostrar que, de ser así, el error cometido por la aproximación en la ecuación (1.5) es de orden h^2 [4] realizando un desarrollo de $f(\mathbf{r'})$ en torno a $\mathbf{r}$ y teniendo en cuenta que $|\mathbf{r}-\mathbf{r'}|$ es como máximo κh.

Un aspecto muy interesante que se obtiene de esta formulación es la representación de la derivada de la función f. Se trata de aplicar $\nabla \equiv \frac{\partial}{\partial \mathbf{r}}$ a la ecuación (1.5)

$$\nabla f(\mathbf{r}) = \frac{\partial}{\partial \mathbf{r}} \int_\Omega f(\mathbf{r'})W(\mathbf{r}-\mathbf{r'},h)d\mathbf{r'} \tag{1.7}$$

donde el gradiente es respecto a las coordenadas no primadas. Dado que el único término que depende de $\mathbf{r}$ es la función kernel, el gradiente de $f(\mathbf{r})$ es simplemente

$$\nabla f(\mathbf{r}) = \int_\Omega f(\mathbf{r'})\nabla W(\mathbf{r}-\mathbf{r'},h) \tag{1.8}$$

En la ecuación (1.8) se observa cómo la derivada en la función pasa a ser una derivada en el kernel. De esta manera, se puede obtener la derivada a partir del valor de la propia función y de la derivada del kernel, en vez de derivar la función en sí misma. Las derivadas, por tanto, se realizan analíticamente y no numéricamente. Además, para el caso de funciones kernel radialmente simétricas, con $r = |\mathbf{r}-\mathbf{r'}|$, el gradiente se puede expresar como

$$\nabla W(r,h) = \frac{\partial W(r,h)}{\partial r}\frac{\mathbf{r}-\mathbf{r'}}{r} \tag{1.9}$$

Para una función vectorial $\mathbf{F}(\mathbf{r})$ se pueden obtener resultados de forma similar, quedando

$$\mathbf{F}(\mathbf{r}) = -\int_{\Omega} \mathbf{F}(\mathbf{r'})W(\mathbf{r}-\mathbf{r'},h) \tag{1.10}$$

$$\nabla\cdot\mathbf{F}(\mathbf{r}) = -\int_{\Omega} \mathbf{F}(\mathbf{r'})\cdot\nabla W(\mathbf{r}-\mathbf{r'},h) \tag{1.11}$$

1.2. Aproximación de partícula

Como ya se ha mencionado, en el método SPH, el sistema entero se representa mediante un número finito de partículas con su propia masa y que ocupan un espacio concreto.

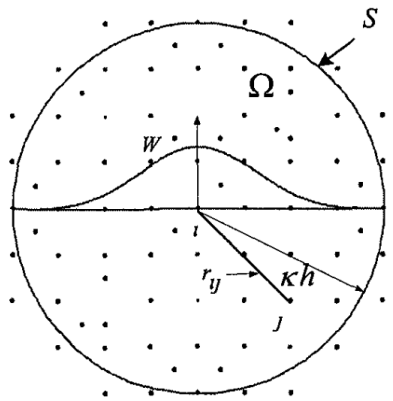


Figura 1: Aproximaciones de partícula usando partículas dentro del dominio de soporte de la función kernel W para la partícula i. El dominio de soporte es circular con radio κh. Crédito: [4].

Esto se consigue mediante la *aproximación de partícula*, la cual permite pasar de las integrales continuas, vistas en la sección anterior, a sumatorios discretos sobre todas las partículas en el dominio de soporte. Si reemplazamos el volumen infinitesimal $d\mathbf{r}'$ en la posición de una partícula j por un volumen finito de la partícula ΔV_j, relacionado con su propia masa

$$\Delta V_j = \frac{m_j}{\rho_j} \tag{1.12}$$

donde ρ_j es la densidad de la partícula j, y donde j puede tomar valores $j = 1, 2, 3, ..., N$; siendo N el número de partículas comprendidas en el dominio de soporte Ω de la partícula i. Usando, por tanto, la descripción de aproximación de partícula, se obtiene

$$\int_{\Omega} f(\mathbf{r'})W(\mathbf{r}-\mathbf{r'},h)d\mathbf{r'} \approx \sum_{j=1}^{N} f(\mathbf{r}_j)W(\mathbf{r}-\mathbf{r}_j,h)\Delta V_j = \sum_{j=1}^{N} f_j W(\mathbf{r}-\mathbf{r}_j,h)\frac{m_j}{\rho_j}$$

$$f_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} f_j W_{ij} \tag{1.13}$$

Según la ecuación (1.13), el valor de la función en la partícula i se aproxima usando la media pesada de la función sobre todas las partículas dentro del dominio de soporte de la partícula i, siendo el peso la función kernel. Obtener la densidad de esta manera es muy simple, quedando una media pesada sobre las masas:

$$f_i = \sum_{j=1}^{N} m_j W_{ij} \tag{1.14}$$

De la misma manera, se puede discretizar la integral para las derivadas espaciales:

$$\nabla f_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} f_j \nabla W_{ij} \tag{1.15}$$

$$\nabla \cdot \mathbf{F}_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} \mathbf{F}_j \cdot \nabla W_{ij} \tag{1.16}$$

1.3. Errores

Las aproximaciones realizadas engloban el error $O(h^2)$ debido al uso de la función kernel así como los errores inherentes de la discretización. El primer tipo de error se puede reducir usando una distancia característica menor, con un consecuente incremento del número de partículas para que haya una cantidad suficiente de éstas en el dominio de soporte. Al aumentar el número de partículas, también se consigue reducir el error debido la discretización, teniendo en cuenta que el límite imaginario de infinitas partículas representaría el continuo. Como es costumbre, una reducción del error implica un aumento de la carga computacional. En este caso, un aumento de partículas representa una carga importante ya que para cada una de ellas hay que determinar las interacciones con sus partículas cercanas, de manera que el número de operaciones escala al menos como $O(NN_j)$, donde N es el número de partículas en la simulación y N_j es el número de partículas que interaccionan con cada una de ellas.

Estos errores quedan ilustrados considerando las aproximaciones a la función constante $f(\mathbf{r}) = 1$. Directamente de la formulación básica de SPH:

$$1 \approx \sum_{j=1}^{N} \frac{m_j}{\rho_j} W_{ij}$$

$$0 \approx \sum_{j=1}^{N} \frac{m_j}{\rho_j} \nabla W_{ij}$$

Como en ambos casos no se cumple la identidad exacta, se deduce que el modelo posee errores inherentes para aproximar incluso funciones constantes. Sin embargo, tomando un número suficiente de partículas se pueden obtener resultados, si más no, aceptables. Se han desarrollado además técnicas de normalización [5] con el objetivo de que se cumplan las igualdades, las cuales no se han aplicado en este trabajo.

1.4. Mejora de derivadas espaciales

A simple vista se puede ver que las derivadas (1.15) y (1.16) de funciones constantes no serán nulas y que los términos en los sumatorios no son antisimétricos al intercambiar las partículas[1]. Una forma general de obtener expresiones que corrijan estos errores es

[1]El gradiente de un kernel par sí lo es. Se puede ver en la ecuación (1.9).

considerar la siguiente igualdad de cálculo vectorial:

$$\nabla(f\rho^n) = nf\rho^{n-1}\nabla\rho + \rho^n\nabla f \tag{1.17}$$

la cual se puede aislar para ∇f y seguidamente aplicar el formalismo de SPH para el gradiente de funciones escalares. Para el caso $n = 1$ (1.18) se corrige el primer error, pero no el segundo. En este trabajo de usa el caso $n = -1$ (1.19), que sólo corrige la propiedad de antisimetría:

$$\nabla f_i = \frac{1}{\rho_i}\sum_j m_j\left(f_i - f_j\right)\nabla W_{ij} \tag{1.18}$$

$$\nabla f_i = \rho_i\sum_j m_j\left(\frac{f_i}{\rho_i^2} + \frac{f_j}{\rho_j^2}\right)\nabla W_{ij} \tag{1.19}$$

Esta expresión es comúnmente usada para calcular el gradiente de presión $\nabla p/\rho$, ya que al ser antisimétrico a pares garantiza la conservación del momento lineal. Además, se puede obtener de forma natural a partir de la formulación lagrangiana de las ecuaciones de fluidos [6]. De manera prácticamente idéntica se pueden obtener dos expresiones para la divergencia de una función vectorial. En este caso se aplicará el caso $n = 1$ (1.20) para la divergencia de la velocidad[2] $\nabla\cdot\mathbf{u}$.

$$\nabla\cdot\mathbf{F}_i = \frac{1}{\rho_i}\sum_j m_j\left(\mathbf{F}_i - \mathbf{F}_j\right)\nabla W_{ij} \tag{1.20}$$

$$\nabla\cdot\mathbf{F}_i = \rho_i\sum_j m_j\left(\frac{\mathbf{F}_i}{\rho_i^2} + \frac{\mathbf{F}_j}{\rho_j^2}\right)\cdot\nabla W_{ij} \tag{1.21}$$

1.5. Ecuaciones de fluidos

El problema a abordar ahora consiste en traducir al formalismo de SPH las ecuaciones usadas para describir la dinámica de fluidos. Se discuten a continuación las ecuaciones de Euler para un fluido compresible, que vienen a ser un caso sin viscosidad ni transporte térmico de las ecuaciones de Navier Stokes. Se considerará la introducción de término viscoso en su momento. Se busca usar las expresiones que describen las propiedades en las partículas, es decir, con la notación de *derivada material*:

$$\frac{D\rho}{Dt} = -\rho\nabla\cdot\mathbf{u} \tag{1.22}$$

$$\frac{D\mathbf{u}}{Dt} = -\frac{\nabla p}{\rho} + \mathbf{g} \tag{1.23}$$

Éstas corresponden a las ecuaciones de continuidad[3] y conservación de momento respectivamente. A partir de ellas se pueden obtener las variaciones de densidad y de velocidad (aceleración) en cada punto, cada instante de tiempo. Para ello hay que discretizarlas primero usando el método de SPH ya descrito. Como ya se ha comentado, éstas se pueden obtener a partir de derivación directa de la densidad para el caso de continuidad o

[2]Éste es el resultado que surge naturalmente derivando totalmente respecto al tiempo la ecuación (1.14) de densidad de SPH.

[3]Equivalente a conservación de masa.

a partir del funcional lagrangiano para el caso de conservación de momento [6]. Equivalentemente, se pueden aplicar directamente las expresiones vistas en el apartado anterior. Los resultados siguiendo cualesquiera de los métodos son los siguientes:

$$\frac{d\rho_i}{dt} = -\sum_j m_j(\mathbf{u}_i - \mathbf{u}_j)\cdot\nabla_i W_{ij} \tag{1.24}$$

$$\frac{d\mathbf{u}_i}{dt} = -\sum_j m_j\left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla_i W_{ij} \tag{1.25}$$

donde $\nabla_i W_{ij} = -\nabla_j W_{ij}$. En la ecuación de continuidad se ve explícitamente cómo en el caso en que las velocidades de dos partículas son idénticas, sus densidades no varían (ya que la situación entre ellas será la misma en el siguiente paso de tiempo). De forma similar, para la ecuación de momento se puede comprobar que la fuerza será de igual magnitud y sentido opuesto entre dos partículas, conservándose el momento lineal.

Finalmente, hay que recordar que las cuatro ecuaciones que se consideran[4] contienen cinco incógnitas (densidad, tres velocidades y presión). Se necesita una restricción para la restante, la presión. Se recurre a una ecuación de estado que depende de la densidad, en este trabajo la *ecuación de Tait*:

$$p = B\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right) \tag{1.26}$$

donde $\gamma = 7$ es un valor típicamente usado y ρ_0 es la densidad de referencia, normalmente la propia del fluido. La constante B se puede relacionar a la velocidad del sonido c_s mediante

$$c_s^2 = \left.\frac{\partial p}{\partial \rho}\right|_{\rho=\rho_0} = \frac{\gamma B}{\rho_0} \tag{1.27}$$

La velocidad del sonido se toma como parámetro, eligiéndose de manera que la variación de densidad entre partículas sea pequeña [5] (normalmente alrededor del 1 %). Ya que

$$\frac{|\delta\rho|}{\rho} = \frac{u^2}{c_s^2} \tag{1.28}$$

para mantener una variación menor al 1 % hay que tomar $c_s^2 \geq 100u_m^2$, siendo u_m la velocidad máxima estimada. Usar una velocidad del sonido elevada deberá compensarse con un paso temporal de integración pequeño, ya que c_s^2 determina la velocidad a la que se transmite la información en el flujo.

1.6. Efectos disipativos: viscosidad

Con el desarrollo realizado hasta ahora ya es posible producir una simulación. Sin embargo, se busca comprobar la validez de ésta. Un caso resoluble analíticamente para someter a comparación, es el que se conoce como *flujo de Poiseuille*. Se trata de un flujo viscoso a través de un canal con la condición de no deslizamiento en los bordes ($\mathbf{u} = 0 \quad en \quad y = 0, H$), dando como solución[5], en el régimen estacionario, un perfil de velocidad en el eje x

[4] La ecuación de conservación del momento tiene tres componentes

[5] Consultar [7] para la demostración.

parabólico $u = \frac{G}{2\nu}(Hy - y^2)$, siendo G una aceleración constante en el eje x debido a un gradiente de presión, ν la viscosidad y H el ancho del canal. Claramente resulta necesaria la introducción de un término viscoso para llevar a cabo tal comprobación.

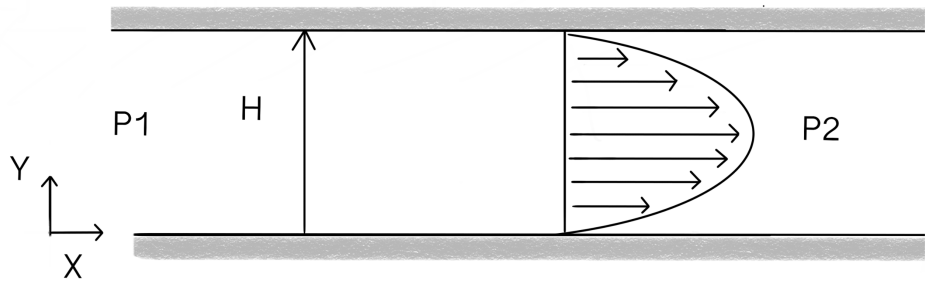


Figura 2: Esquema del flujo de Poiseuille . Flujo a través de un canal de ancho H debido a una diferencia de presión ($p_1 > p_2$).

La disipación en SPH fue introducida por Monaghan y Gingold [8] posteriormente a su desarrollo para modelar ondas de choque, un tipo de perturbación que se propaga más rápido que la velocidad del sonido en el medio y donde la energía cinética se disipa en forma de energía térmica. El término de viscosidad artificial en SPH, ampliamente usado, es el siguiente:

$$\Pi_{ij} = \begin{cases} \frac{-\alpha c_s \mu_{ij}}{\bar{\rho}_{ij}} & \mathbf{u}_{ij} \cdot \mathbf{r}_{ij} < 0 \\ 0 & \mathbf{u}_{ij} \cdot \mathbf{r}_{ij} \geq 0 \end{cases} \tag{1.29}$$

$$\mu_{ij} = \frac{h\mathbf{u}_{ij} \cdot \mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^2 + 0.01h^2} \tag{1.30}$$

$$\bar{\rho_{ij}} = \frac{1}{2}(\rho_i + \rho_j) \tag{1.31}$$

$$\mathbf{u}_{ij} = \mathbf{u}_i - \mathbf{u}_j, \quad \mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j \tag{1.32}$$

La constante α se usa para regular el efecto y suele tomar un valor cercano a 1. El factor h^2 en μ_{ij} previene divergencias a medida que dos partículas se acercan. El término viscoso solamente es no nulo en eventos de compresión ($\mathbf{u}_{ij} \cdot \mathbf{r}_{ij} < 0$). Además no debe ser negativo, por lo que se le añade el signo menos delante. El término se incluye en la ecuación de conservación del momento lineal:

$$\frac{d\mathbf{u}_i}{dt} = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} + \Pi_{ij} \right) \nabla_i W_{ij} \tag{1.33}$$

Dado que $\mathbf{r}_{ij}$, $\mathbf{u}_{ij}$ son antisimétricos bajo intercambio en ij, el término viscoso es simétrico y por tanto la ecuación seguirá siendo antisimétrica a pares, de forma que se conserva el momento lineal.
Es posible relacionar los parámetros con la viscosidad del fluido, lo que permitirá comparar la simulación con la solución analítica.

$$\nu \approx \frac{\alpha c_s h}{2(2 + dim)} \tag{1.34}$$

Esta expresión no tiene en cuenta que en las simulaciones sólo se aplica el término viscoso

cuando las partículas se acercan ($\mathbf{u}_{ij} \cdot \mathbf{r}_{ij} < 0$), siendo por tanto su valor aproximadamente la mitad en estos casos [9].

1.7. Integración temporal

Como ya se ha comentado, uno de los puntos fuertes de la formulación SPH es que permite ser deducida directamente del Lagrangiano del sistema [10], de manera que se cumple la conservación del momento lineal[6]. Es por ello que se suele hacer uso de *integradores simplécticos* [11], que se derivan de transformaciones canónicas y por tanto preservan el lagrangiano. Además son reversibles en el tiempo para sistemas no disipativos, lo que permite comprobar la precisión de la simulación (revirtiendo las velocidades y verificando su trayectoria, por ejemplo). Uno de ellos es el esquema *Leap-frog* de segundo orden, que calcula la posición a medio paso de tiempo posterior y la usa para calcular la velocidad y posición en el paso completo. Se le añade al esquema, que sólo integra posiciones y velocidades, el cálculo de densidad [5], quedando de la siguiente manera:

$$\begin{aligned}
\mathbf{r}_{1/2} &= \mathbf{r}_0 + \frac{dt}{2}\mathbf{u}_0 \\
\mathbf{u}_{1/2} &= \mathbf{u}_0 + \frac{dt}{2}\mathbf{F}(\mathbf{r}_{-1/2}, \mathbf{u}_{-1/2}, \rho_{-1/2}) \\
\rho_{1/2} &= \rho_0 + \frac{dt}{2}D(\mathbf{r}_0, \mathbf{u}_0) \\
\mathbf{u}_1 &= \mathbf{u}_0 + dt\mathbf{F}(\mathbf{r}_{1/2}, \mathbf{u}_{1/2}, \rho_{1/2}) \\
\mathbf{r}_1 &= \mathbf{r}_{1/2} + \frac{dt}{2}\mathbf{u}_1 \\
\rho_1 &= \rho_{1/2} + \frac{dt}{2}D(\mathbf{r}_1, \mathbf{u}_1)
\end{aligned}$$

Siendo $D = \frac{d\rho}{dt}$ y $\mathbf{F} = \frac{d\mathbf{u}}{dt}$ las variaciones en densidad y velocidad respectivamente. El paso de tiempo dt queda acotado por las condiciones de Courant respecto a c_s [3] y ν [12]

$$dt = \text{mín}\left(0.8\frac{h}{c_s},\, 0.125\frac{h^2}{\nu}\right) \tag{1.35}$$

2. Machine learning

Machine learning, literalmente aprendizaje de máquinas y traducido como aprendizaje automático, es el nombre dado a una serie de técnicas que permiten que un programa sea capaz de *aprender* a partir de un conjunto de datos, generalmente grande (o muy grande), sin necesidad de programar explícitamente las reglas. Se dice que un programa es capaz de aprender sobre una tarea a partir de experiencias si su rendimiento sobre la tarea mejora con las experiencias (Tom Mitchell, 1997 [13]).

Debido a la capacidad de este tipo de algoritmos de procesar una gran cantidad de datos para luego realizar tareas acordes a sus *conocimientos*, los programas desarrollados mediante *machine learning* son capaces de obtener resultados mucho mejores que los

[6]También se conserva la energía, aunque en este trabajo no se trata con esta variable.

convencionales en problemas complejos. Un ejemplo de ello es su rendimiento superior en reconocimiento de patrones en datos, como pueden ser imágenes o sonido, palabras y frases, así como cualquier tipo de señal temporal. Una de sus primeras aplicaciones a gran escala fue la implementación de un filtro de correo *spam*, en torno al año 1990. Programar explícitamente un algoritmo capaz de clasificar eficazmente mensajes a partir de su contenido resulta altamente desalentador. Basta tan sólo con imaginar la cantidad de condiciones y la creación de un diccionario para detectar palabras y frases maliciosas. Si además se tiene en cuenta la evolución del lenguaje y la sociedad, cada poco tiempo sería necesario mejorar el programa para incluir la nuevas expresiones, así como nuevas formas de publicitar y/o engañar, contribuyendo constantemente a enmarañar el ya complicado programa. Con la aparición del *machine learning*, sin embargo, se crea la posibilidad de generar un programa que sea capaz de clasificar el correo mostrándole ejemplos sobre qué es deseable y qué no. Además, generar actualizaciones resulta menos tedioso, ya que sólo hacen falta nuevas muestras que enseñarle. Éste es el paradigma de un aprendizaje automatizado donde los datos están previamente etiquetados, lo que se conoce como *aprendizaje supervisado*. Desde luego, existe el *aprendizaje no supervisado*, donde se suele buscar que el algoritmo agrupe los datos según su similaridad, reduzca la dimensionalidad de algún problema o deduzca relaciones entre ciertas variables de los datos introducidos. Esta clase de problemas no requiere de etiquetado previo y se llevan a cabo por la propia estructura del algoritmo.

Como ya se ha comentado, desde los inicios de la computación se ha mostrado gran interés en el aprendizaje automático. El artículo publicado en el 1943 por W. McCulloch y W. Pitts [1], donde presentaban un modelo computacional simple sobre cómo el funcionamiento de neuronas en animales podían producir cálculos complejos, se considera la primera arquitectura de red neuronal artificial. La idea de poder reproducir inteligencia artificialmente generó una gran cantidad de investigaciones con una perspectiva muy positiva, hasta llegar a los años 1960, cuando se entró en un periodo de pesimismo debido al incumplimiento de las expectativas generadas. Alrededor de 1980, nuevas investigaciones y arquitecturas reviven el entusiasmo por el campo. En la siguiente década se desarrollan nuevos algoritmos desde un enfoque más cercano al tratamiento de datos y destaca el uso de las *Support Vector Machine*, dejando en segundo plano las redes neuronales artificiales. Sin embargo, Geoffrey Hinton et al. publican un artículo [14] en 2006 mostrando la capacidad de una red neuronal artificial de reconocer dígitos escritos a mano con gran precisión. La novedad consistió en el uso de redes neuronales *profundas*, de más de una capa, que hasta ese momento no se habían conseguido *entrenar* con éxito.
Poco más de una década más tarde, en la actualidad, cuando se habla de *machine learning* se hace referencia al uso de redes neuronales artificiales profundas en la gran mayoría de casos, siendo la investigación en este campo muy activa.

2.1. Redes neuronales artificiales

La estructura de una red neuronal artificial se inspira en las interacciones del cerebro animal. Ésta consiste en un número arbitrario de unidades denominadas neuronas que se agrupan en una sucesión ordenada de capas. De forma general, cada neurona se conecta a todas las neuronas de las capas inmediatamente vecinas (la capa previa y la próxima), siéndole asignado a cada conexión un peso específico. En la forma más simple, una neurona recoge la señal de cada conexión (x_i) multiplicada por su peso (w_i), para seguidamente

sumar todas las contribuciones. Finalmente, se añade un término independiente (en inglés *bias*, b). Se suele aplicar además una *función de activación* al resultado que determina la señal que emite la propia neurona:

$$y = f_{act}\left(\sum_i (x_i \cdot w_i) + b\right) \tag{2.1}$$

Normalmente la primera capa, la de entrada, simplemente emite como señal los datos introducidos. Los resultados se obtienen de las señales emitidas por las neuronas de la última capa, la de salida. La función de activación se elige teniendo en cuenta factores como el rango de valores, la arquitectura de la red o el tipo de problema a resolver. En capas intermedias se usan funciones como la tangente hiperbólica o la ReLU ($ReLU(z) \equiv max(0, z)$), mientras que en las de salida se usan sigmoides o lineales.

El entrenamiento de una red parte de la idea de las interacciones en el cerebro bajo la *regla de Hebb* [15]: cuando una neurona activa a otra neurona, su conexión se refuerza. Es decir, los pesos de conexión entre las neuronas que contribuyen a un mismo resultado se ven aumentados. El proceso se lleva a cabo mediante una variante de esta idea que tiene en cuenta el error cometido por la red respecto al resultado deseado: para cada instancia del entrenamiento se realiza una predicción, es decir, los datos introducidos atraviesan la red de entrada a salida y producen unos resultados. Seguidamente se calcula el error respecto al resultado esperado y se atraviesa la red en sentido contrario, de salida a entrada, midiendo de cada conexión la contribución en el error, para finalmente modificar ligeramente los pesos con el objetivo de reducir el error. Este último viene determinado por una función a elegir que se denomina *coste* (o *loss*). El entrenamiento consiste, por tanto, en encontrar el conjunto de pesos que minimizan la función coste.

Para la tarea de minimizar existen diversos métodos, pero todos parten de la idea del *gradient descent*: derivar la función coste respecto a los parámetros ajustables (pesos) para obtener la pendiente de la función y desplazarse hacia la zona de mayor disminución, es decir, calcular el gradiente y desplazarse en sentido opuesto. Iterando el proceso se puede llegar al mínimo a una velocidad que depende del tamaño de los pasos realizados. La siguiente ecuación define un paso del algoritmo, donde $\boldsymbol{\theta}$ es el vector de pesos, η controla el tamaño de los pasos y se denomina tasa de aprendizaje (*learning rate*) y J la función coste:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.2}$$

La resolución de derivadas se puede abordar de varias maneras. Las métodos más directos serían diferenciación manual, simbólica o numérica. Sin embargo, ninguna de ellas se ajusta a la cantidad ni a la precisión necesaria en el problema de minimizar el coste. Un método que sí cumple los requisitos, especialmente cuando se precisa de muchas entradas y pocas salidas, es el método denominado *reverse-mode autodiff*. Consiste en una sucesión jerárquica de nodos, cuyos valores se obtienen en un paso *forward* y son resultados de operaciones usando los valores de entrada. Una vez obtenidos éstos, se realiza un paso *reverse* donde se obtienen las derivadas parciales mediante regla de la cadena, sobre las operaciones que determinan cada nodo[7]. La librería de Tensorflow, la cual se usa en este trabajo, implementa este algoritmo para diferenciar la función coste.

[7]Para una explicación más detallada, consultar [16].

Una vez resuelto el problema de la diferenciación, cabe destacar que el algoritmo de *gradient descent* también plantea problemas: con una tasa de aprendizaje constante, la convergencia hacia un mínimo puede ser muy lenta debido a que el gradiente irá disminuyendo progresivamente. Además, dependiendo de la función coste usada, se puede acabar en un mínimo local, quedando mermada la eficiencia de la red. Uno de los métodos más usados en la actualidad es el método *adam* (*adaptative moment estimation*) [17], que combina las ideas de otros algoritmos avanzados y es capaz de corregir ambos problemas mencionados. Éste implementa una tasa de aprendizaje adaptativa que es menor o mayor dependiendo de si el gradiente es grande o pequeño, respectivamente, y además acumula los gradientes anteriores, dotándolo de una cierta inercia para rehuir mínimos locales.

2.1.1. Conceptos

Es de interés distinguir una serie términos usados a la hora de entrenar la redes neuronales:

- *Instancia*: cada conjunto de datos de entradas y salidas que se usan para entrenar la red neuronal.
- *Época*: consiste en una iteración completa sobre todos los datos proporcionados y donde los parámetros han sido ajustados para reducir la función coste. Generalmente, una red se entrena durante varias épocas hasta alcanzar un estado más o menos estacionario.
- *Batch*: hace referencia a los datos que se suministran al sistema de una vez, en paquete, durante el entrenamiento. El algoritmo de minimización actúa sobre el conjunto del *batch*. El tamaño de éste puede consistir en una sola muestra, en varias o en el conjunto completo. Puede ser de interés para cargar datos secuencialmente cuando el tamaño total es demasiado grande como para caber en la memoria. Además, un tamaño menor puede ayudar a escapar de mínimos locales, ya que el resultado de la optimización será menos suave que al usar un *batch* grande.
- *Overfitting*: fenómeno que ocurre cuando la red neuronal no tiene datos suficientemente variados respecto al número de parámetros a optimizar. En vez de inferir la forma real de la función objetivo, la red se ajusta a los datos usados en el entrenamiento, mostrando un rendimiento pobre sobre instancias nuevas, es decir, no generaliza bien.
- *Conjunto de validación*: instancias sobre las cuales, transcurrido un número determinado de pasos de entrenamiento (generalmente al final de una época), se realiza una predicción y se calcula su error respecto al resultado esperado. Este conjunto no se usa para entrenar la red y sirve para identificar posibles errores de *overfitting*.
- *Conjunto de prueba*: instancias sobre las cuales se realiza una predicción y se calcula su error tras entrenar por completo la red. Este conjunto no se usa para entrenar la red y sirve como indicador del rendimiento general.

3. Simulación SPH

Previamente a desarrollar cualquier red neuronal, se implementa una simulación según el modelo de SPH ya explicado. Se pretende validar el programa y los métodos usados comparando con el resultado analítico del flujo de Poiseuille, para posteriormente usar los bloques de cálculo en la elaboración y análisis de las redes neuronales.

Se usa *Python 3* para la totalidad del programa. Se recurre a la librería *Numpy* para almacenar y operar sobre los datos debido a su gran eficiencia, resultante de poseer su código base en *C*. Resulta de interés que la ejecución de la simulación sea de una velocidad aceptable para posteriormente comparar con el rendimiento de la red neuronal. Las representaciones gráficas se realizan mediante *Matplotlib*.

3.1. Planteamiento

Acorde a las características del problema, es suficiente con que la simulación sea en dos dimensiones, lo cual reduce la carga computacional. El dominio escogido para el problema, donde se halla el fluido, es de $x \in [0,\, 0.1]$, $y \in [0,\, 0.4]$. El dominio en x es menor debido a que se usan condiciones de contorno periódicas en este eje. Con esta misma idea, se genera una matriz de 10x40 partículas repartidas equidistantemente en todo el dominio del problema, con velocidad y aceleración inicial nulas. La masa se escoge de manera que al aplicar inicialmente el sumatorio para interpolar las densidades, el valor de éstas coincida con el de la densidad de referencia ρ_0. Seguidamente se podrá calcular la presión mediante la ecuación de estado.

Es necesario generar *paredes* en los límites del dominio en y que contengan el fluido y al mismo tiempo impongan la condición de no deslizamiento mediante viscosidad. A lo largo de los años se han desarrollado diversos métodos con varios órdenes de precisión y complejidad. En este trabajo se opta por un enfoque sencillo, pero que explota las capacidades de interacción de partículas en SPH: se genera una sucesión de partículas en el eje x a la altura de cada límite en y, con el mismo espacio entre partículas que el usado en el fluido. Las partículas que conforman la pared poseen velocidad nula en todo momento de la simulación. A efectos de las interacciones, forman parte en los cálculos de la ecuación de continuidad y momento de la misma manera que las partículas del fluido. También poseen la misma masa, de forma que inicialmente la densidad es uniforme en todo el dominio. Adicionalmente, se añaden tantas capas exteriores como sean necesarias con el fin de que una partícula que se aproxime a

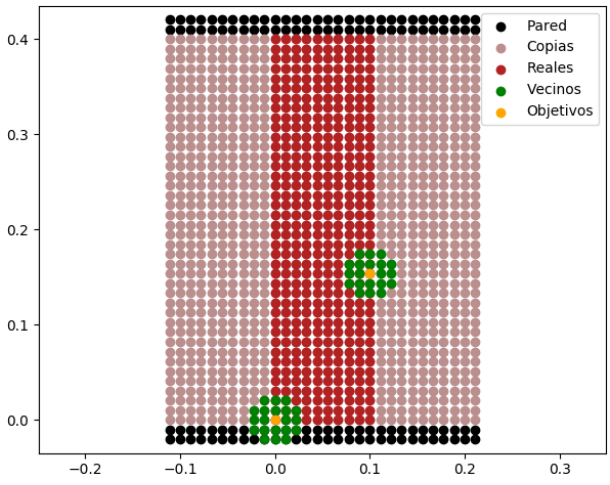


Figura 3: Representación de las diferentes partículas por colores en el estado inicial. Se muestran los vecinos próximos de dos partículas, fijadas como objetivos, en los límites del dominio. La muestra se genera con el propio programa de simulación.

la pared no note discontinuidad alguna.

Para implementar la condición de contorno periódico, se copia y traslada el sistema en el eje x: $x_{i\pm} = x_i \pm x_{sp} \pm (x_{max} - x_{min})$, siendo $x_\pm$ la copia trasladada a derecha o izquierda del dominio, x_{sp} el interespaciado de las partículas en x y x_{max}, x_{min} los límites en el mismo eje. Los cálculos se realizan sólo sobre las partículas dentro del dominio, pero teniendo en cuenta las copias.

La función kernel que se usa es, si bien no la que ofrece mejores resultados, la más intuitiva y clásica: la gaussiana (isótropa)

$$W(r,h) = \frac{1}{(\pi h^2)^{dim/2}} e^{-(r/h)^2} \tag{3.1}$$

$$\nabla W(r,h) = \frac{-2r}{h^2} W(r,h)\hat{\mathbf{r}} \tag{3.2}$$

Un problema que supone la gaussiana es que no es de soporte compacto, sino que se extiende al infinito. En la práctica, generalmente se trunca a una distancia de $3h$ [18]. De esta manera, la longitud característica h se toma de tal forma que dentro del dominio de soporte de cada partícula se incluyan inicialmente 20 otras partículas, procedimiento habitual al usar longitudes características constantes.

El último aspecto técnico a tratar este apartado es la búsqueda de próximos vecinos. Se ha usado el algoritmo de *KDTree* implementado en la librería *Scipy* según *Maneewongvatana* [19], que ofrece un gran rendimiento. Un *kdtree* es una estructura de datos para almacenar un conjunto finito de puntos de un espacio k-dimensional. El algoritmo particiona el espacio es diversas zonas de menor tamaño. Todo el espacio queda representado en un árbol jerárquico donde cada partición genera un nuevo nodo. Cada subpartición se representa como un nodo hijo del anterior, siendo dos el máximo de descendientes inmediatos para cada nodo[8]. De esta manera dos puntos cercanos en el espacio también lo son en el árbol, volviéndose más ágil la búsqueda de próximos vecinos[9].

3.2. Parámetros

Los valores usados a lo largo de la simulación son los que se muestran en la tabla (1), donde c_s se escoge acorde a la condición de variaciones pequeñas en densidad y dt acorde a la condición de Courant. En $\rho_0 = 1000 kg/m^3$ se ha usado la densidad del agua.

G	masa	ρ_0	c_s	γ	α	h	dt
0.05	0.11365	1000	30	7	1	$0.9x_{sp}$	10^{-4}

Tabla 1: Tabla con los parámetros usados en la simulación del flujo.

La aceleración constante G se añade directamente a la aceleración calculada por la ecuación de momento, de forma análoga a como se hace para la inclusión de la aceleración de la gravedad.

[8]Esta estructura jerárquica se la conoce por el nombre de *árbol binario*, debido al número máximo de hijos directos de un nodo.

[9]Para una explicación más detallada, consultar la referencia [19].

3.3. Resultados

La simulación se lleva a cabo monitorizando la velocidad máxima en el eje x y se detiene cuando ésta llega a un estado estacionario. Tras unos 6×10^4 pasos (6 unidades de tiempo) la velocidad deja de aumentar tras alcanzar el valor máximo de $u_{max} = 0.0789\, m/s$. A partir de la ecuación analítica del flujo de Poiseuille, con $\nu = 0.0128\, m^2/s$ según la expresión vista anteriormente, la velocidad máxima debería ser $u_{max} = 0.0779\, m/s$. Queda representada en la figura (4) la comparación entre los perfiles analítico y de simulación del flujo en el estado estacionario, representando la posición en el eje de las ordenadas y la velocidad en las abscisas. Según el resultado de esta prueba, los cálculos del programa desarrollado son fiables, al menos para flujos que no presentan discontinuidades ni cambios bruscos.

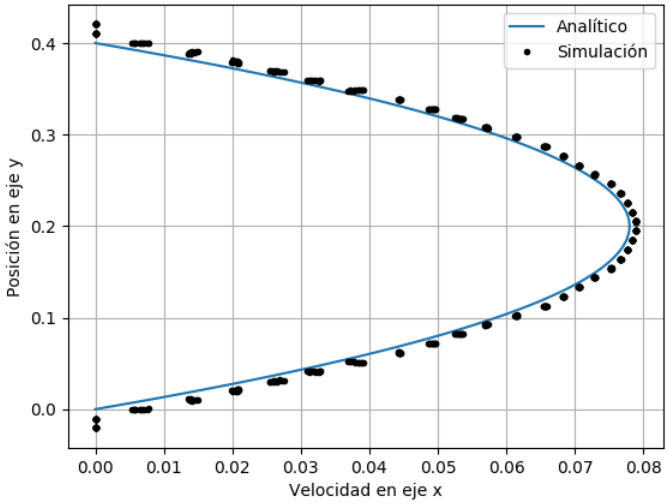


Figura 4: Comparación del flujo estacionario para el cálculo teórico y el resultado de la simulación. El perfil hiperbólico queda bien ajustado.

4. Implementación de redes neuronales

Tras comprobar la validez del programa y sus métodos, el problema a afrontar es el de estudiar la posible substitución de cálculos por redes neuronales artificiales que realicen la misma función. En la actualidad se dispone de una gran variedad de herramientas que permiten crear y entrenar redes neuronales artificiales de distintos tipos. Cada vez más lenguajes de programación y plataformas de análisis de datos incluyen en su repertorio librerías orientadas a estas tareas. En el caso de Python, una de las más conocidas es *Tensorflow*, desarrollada por Google [20] y de código abierto, implementada en C++ y extendida a Python. Es la base del libro de Aurélien Géron [16] usado como referencia en el trabajo. Su flexibilidad y libertad son un gran punto a favor en el estudio e investigación de redes neuronales. Sin embargo, para facilitar el desarrollo y la prueba de arquitecturas, Google también ha concebido una API de alto nivel[10], usando como fondo Tensorflow, llamada *Keras*, que a pesar de no ser tan detallada como la librería original, dispone de

[10] *High-level API* hace referencia a interfaces que son más sencillas e intuitivas de usar, en contraposición a las *low-level API* que están más orientadas al control detallado. Keras y Tensorflow son ejemplos de cada tipo respectivamente.)

una gran variedad de capas ya preparadas con la posibilidad de ajustar sus parámetros, así como funciones de activación y de coste, entre muchas otras herramientas. Además, implementa la *functional API*, donde las capas se pueden llamar entre ellas como funciones, pasando como argumentos otras capas, lo cual permite el desarrollo de estructuras complejas de forma relativamente simple.

4.1. Procedimiento

Las instancias de entrenamiento se generan en Python, recurriendo a Numpy, y usando las funciones implementadas en la simulación para el cálculo de los resultados. Esto permite generar una cantidad arbitraria de datos con la forma deseada, lo que debería de contribuir a la robustez del modelo entrenado. El análisis consiste en primero identificar el cálculo a substituir, para el cual se generan datos y entrenan varias redes neuronales de diferentes características. Seguidamente se evalúan los errores que producen tras los entrenamientos y se escoge un caso aceptable para comparar la velocidad de procesamiento de la red neuronal con la del cálculo original. Se usa primero Keras para probar con distintas configuraciones de redes neuronales, debido a la facilidad con que permite ajustar los diferentes parámetros y estadios del entrenamiento y evaluación. Sin embargo, se observa que la velocidad de predicción de las redes desarrolladas en Keras resultan algo más lentas que sus contrapartidas en Tensorflow. Es por ello que se recurre al *estimador* DNNRegressor de esta última API para entrenar y probar las redes que se someten a la comparación final.

Durante el entrenamiento se usa el método de optimización *adam* sobre la función coste de error cuadrado medio (MSE, *mean squared error*). Como métrica adicional, se usa el error absoluto porcentual medio (MAPE, *mean absolute percentage error*). Tras el entrenamiento, se cuantifica el error final mediante RMSE, que es simplemente tomar la raíz cuadrada del MSE. Sea un conjunto de N valores auténticos $\{X_i\}$ y predicciones $\{x_i\}$, se definen:

$$RMSE(x, X) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - X_i)^2} \tag{4.1}$$

$$MAPE(x, X) = \frac{100}{N}\sum_{i=1}^{N}\left|\frac{x_i - X_i}{X_i}\right| \tag{4.2}$$

Las partes del método SPH donde se busca mejorar el cálculo es en la obtención de la función kernel y su gradiente, así como el término de interacción en la ecuación de continuidad. Para generalizar el cálculo, se busca escalar tanto las posiciones como las velocidades, de forma que la predicción sea adimensional. Para las posiciones se usa la distancia de suavizado h del kernel. Para las velocidades se usa c_s, que como se ha visto viene determinada por la velocidad máxima (estimada) del fluido.

$$q \equiv \frac{r}{h} \in [0, 3]; \qquad \omega \equiv \frac{10u}{c_s} \in [0, 1] \tag{4.3}$$

4.2. Resultados

4.2.1. Kernel

Se inicia el estudio con la predicción del kernel. La función en sí es sencilla, así que parece un buen punto de partida sobre el que evaluar el método. Reescribiendo el kernel en las variables adimensionales:

$$W(q) = \frac{e^{-q^2}}{\pi} \times \left(\frac{1}{h^2}\right) \tag{4.4}$$

Se busca que la red neuronal aprenda a predecir la función en unidades de $h = 1$, siendo posible reescalar el valor según la longitud característica del problema. Al ser isotrópica la función kernel considerada, sólo depende del módulo de la distancia y por tanto basta facilitar un valor de entrada. El resultado también es un sólo valor, por lo que la red resulta de lo más simple posible.

Se generan 10^6 distancias $q \in [0, 3]$ de forma aleatoria y se calcula el kernel con $h = 1$. De las instancias, un 1 % se usa como validación. Ya que todos los valores son mayores a cero, en las capas intermedias se puede usar la función de activación ReLU sin temor a que desaparezca información. La capa de entrada no lleva ninguna activación y la de salida es lineal, la cual no tiene gradientes y simplemente muestra el valor recibido. Los datos de entrenamiento generados son distintos para cada red y se barajan al inicio de cada época.

Inicialmente se prueba con una red de una sola capa intermedia, con diferente número de neuronas. En cada caso se entrena a lo largo de cinco épocas y para cada una se guarda el estado de la red en el final de la época con menor error en el conjunto de validación. Seguidamente, se evalúan las redes sobre un grupo de prueba de 10^5 datos generados aleatoriamente con las mismas características que las instancias originales, midiendo el MSE y MAPE. Este proceso se repite cinco veces para cada red, conjunto del cual se saca la media de cada error y su desviación estándar. Los entrenamientos, aunque son distintos entre ellos por la aleatoriedad de los datos, deberían converger a unos resultados muy similares siempre y cuando el número de iteraciones (épocas) sea elevado. Al usar sólo cinco épocas en esta fase, es conveniente realizar varias medidas para paliar con la dispersión provocada. Se repite el proceso para todos los números de neuronas, siendo el conjunto de prueba el mismo para todas ellas. El proceso se realiza además para redes neuronales con dos y tres capas intermedias. Se presentan a continuación los resultados.

1 capa	10	20	50	100	200	500
RMSE	0.014(17)	0.0028(14)	0.0017(5)	0.0005(1)	0.00025(5)	0.00024(9)
MAPE	2300(3500)	380(250)	200(100)	60(40)	37(25)	30(10)

2 capas	10	20	50	100	150	250
RMSE	0.0024(19)	0.0009(3)	0.00033(15)	0.0002(1)	0.00014(5)	0.000007(2)
MAPE	360(380)	83(35)	40(20)	25(29)	17(13)	6(5)

3 capas	10	20	50	100	150	250
RMSE	0.00068(25)	0.00044(25)	0.00014(6)	0.00008(3)	0.00007(2)	0.00005(1)
MAPE	71(19)	70(57)	17(14)	10(4)	7(2)	5(2)

La desviación queda representada entre paréntesis, modificando la últimas cifras correspondientes. Valores de esta medida mayores que la propia media, hacen que esta última deje de tener sentido, siendo muy difícil determinar un número concreto debido a grandes variaciones.

En las tablas se observa cómo el MAPE puede ser una medida útil en este problema de regresión. Un RMSE bajo no implica necesariamente que la red esté rindiendo bien, ya que esta medida depende del rango en que se realizan las predicciones. En este caso $W(q) \in [4 \times 10^{-5}, 0.318]$, donde se puede ver que la proporción de un error de 0.014 es completamente diferente en ambos extremos. El MAPE, en este rango de valores, revela cómo de bien ajustados están los resultados más cercanos a cero. En cualquier caso, hay que tener en cuenta que el RMSE proporciona una medida de la magnitud de los errores, mientras que el MAPE mide la proporción de los errores.

Los resultados muestran, a número de capas intermedias constante, cómo ambos errores tienden a disminuir al incrementar el número de neuronas. Para una sola capa se obtienen resultados bastante aceptables, sobretodo a partir de las 100 neuronas. Sin embargo, incluso con 500 neuronas, resulta complicado obtener un buen rendimiento general: el RMSE es del orden de 10^{-4}, lo que es más que aceptable para los valores mayores de $W(q)$, sin embargo es elevado para los más pequeños, hecho que se refleja en el MAPE. Este resultado es natural al estar minimizando el MSE, ya que de esta manera todos los valores presentarán una cantidad de error similar. Esta observación hecha a partir de la tabla queda representada en la gráfica de la figura (5), donde se representa el error relativo respecto al valor de la función para el número mayor de neuronas en cada capa. El rendimiento es también mayor al aumentar la cantidad de capas. Esto se puede ver como que cada capa se especializa en detectar y tratar con ciertos patrones, siendo la resolución de estos patrones más pequeña y precisa a medida que se adentra en la red[11]. En la práctica no tiene por qué ser estrictamente así, pero esta interpretación ayuda a comprender la utilidad de la adición de capas.

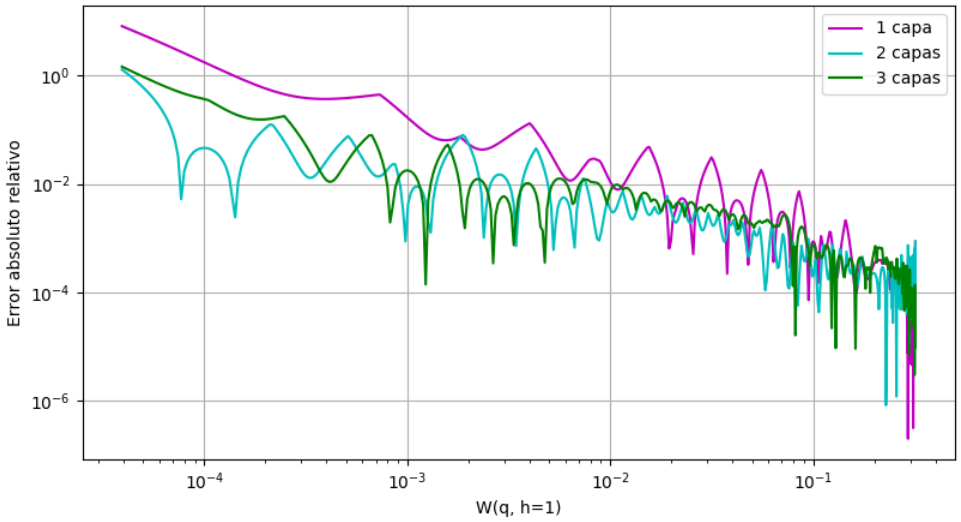


Figura 5: Error absoluto relativo (proporción entre error absoluto y valor real) en función de $W(q)$, en escala logarítmica. Se aprecia cómo los errores de más peso se producen en los valores más pequeños.

[11]Esta estructura jerárquica se ha observado en el reconocimiento de imágenes mediante redes neuronales convolucionales. Para más información, consultar el libro de referencia [16]

Teniendo en cuenta que cada neurona se conecta a todas las de las capas adyacentes, resulta fácil calcular el número de conexiones en cada red[12]. Cuantas más conexiones, la optimización se vuelve más larga y necesita una cantidad mayor de datos, así como mayor es la *profundidad* que alcanza la red. Esto hace que se prefiera el uso de la red de tres capas con 100 neuronas (20200 conexiones) a la de dos capas con 250 neuronas (62505 conexiones). La primera es la que se escoge para entrenar y cualificar.

Para evaluar el tiempo se usa el módulo *timeit* de Python, el cual está desarrollado para determinar el tiempo de ejecución de porciones concretas de código, ofreciendo la posibilidad añadida de poder iterarlo un número concreto de veces y que devuelva el tiempo mínimo. Éste se mide sobre un *DNNRegressor* de ReLUs, entrenado con *adam* sobre el MSE durante 15 épocas, cada época consistiendo en 1000000 pasos. Las características de las instancias son idénticas a las del entrenamiento con Keras. Tras entrenarla, se evalúan tanto la red neuronal como el algoritmo original (basado en arrays de numpy) en conjuntos de tamaño 10^n con $n = 0, 1..., 8$, midiendo la velocidad de ambas redes. Finalmente, se compara la función exacta con la predicha por la red neuronal, mostrando los errores de la segunda.

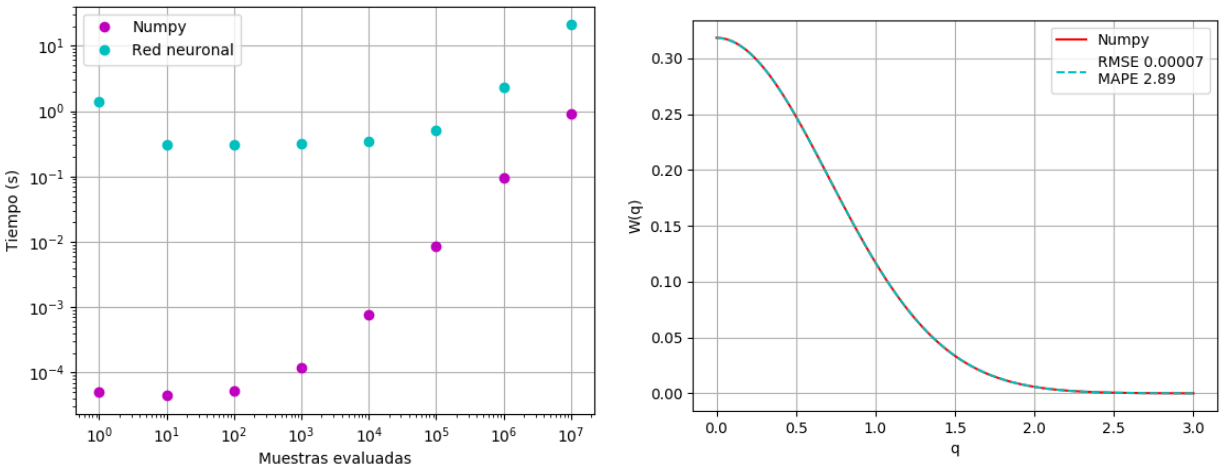


Figura 6: Resultados para la predicción del kernel. A la izquierda la comparativa de tiempos de ejecución. A la derecha la función por cálculo exacto y por red neuronal superpuestas. $RMSE = 7 \times 10^{-5}$, $MAPE = 2.89$.

Numpy consigue proporcionar una eficiencia mucho mayor que la red neuronal. Se puede apreciar que además, para muestras muy pequeñas la red neuronal rinde peor que para muestras de mayor magnitud. Esto se debe a que la librería usa funciones de multiprocesado y necesita un tiempo (*overhead*) para distribuir los datos a través de distintos núcleos, tiempo que para muestras pequeñas es un desperdicio.

4.2.2. Divergencia del kernel

La expresión de la divergencia en función de q y h es:

$$\nabla W(q) = -2qW\frac{\hat{\mathbf{r}}}{h} = \frac{2qe^{-q^2}}{\pi} \times \left(\frac{-\hat{\mathbf{r}}}{h^3}\right) \tag{4.5}$$

[12]En el caso de dos capas, por ejemplo, siendo n el número de neuronas, el número de conexiones será $2n + n \times n$.

La divergencia de la gaussiana es también isótropa: aunque el resultado es un vector, éste sólo depende del módulo de la distancia, obteniéndose la forma vectorial al multiplicar por el vector unitario entre las dos partículas interaccionantes. Esto permite que, de nuevo, tanto la entrada como la salida consistan en una sola neurona. Al usar ReLU como función de activación, es preciso evitar los valores negativos, por lo que el signo menos se aplica posteriormente junto a la longitud característica y el vector unitario. El tratamiento seguido es, por tanto, el mismo que para el caso del kernel, por lo que se muestran directamente los resultados.

1 capa	10	20	50	100	200	500
RMSE	0.089(12)	0.095(1)	0.093(1)	0.093(1)	0.0926(1)	0.0925(2)
MAPE	5700(2800)	5800(2800)	2000(3000)	2000(3000)	360(70)	370(40)

2 capas	10	20	50	100	150	250
RMSE	0.0028(14)	0.0014(4)	0.0005(1)	0.0004(2)	0.00024(5)	0.00020(3)
MAPE	70(50)	35(10)	10(4)	8(4)	6(3)	3(1)

3 capas	10	20	50	100	150	250
RMSE	0.00122(3)	0.0006(3)	0.00022(4)	0.00015(2)	0.00014(6)	0.00008(1)
MAPE	24(15)	12(7)	5(2)	4(2)	3(2)	1.6(4)

La casuística para el gradiente es similar a la del kernel. Esta vez, la red no parece ser capaz de aproximarla con éxito con el uso de una sola capa. En efecto, la función es algo más complicada que la anterior y parece razonable este resultado. De nuevo los valores que más contribuyen al MAPE son los cercanos a cero, lo cual queda representado en la figura (7). Con el aumento de capas, el error general disminuye rápida y considerablemente. Se opta por estudiar la misma estructura de red que en el caso anterior: tres capas de 100 neuronas.

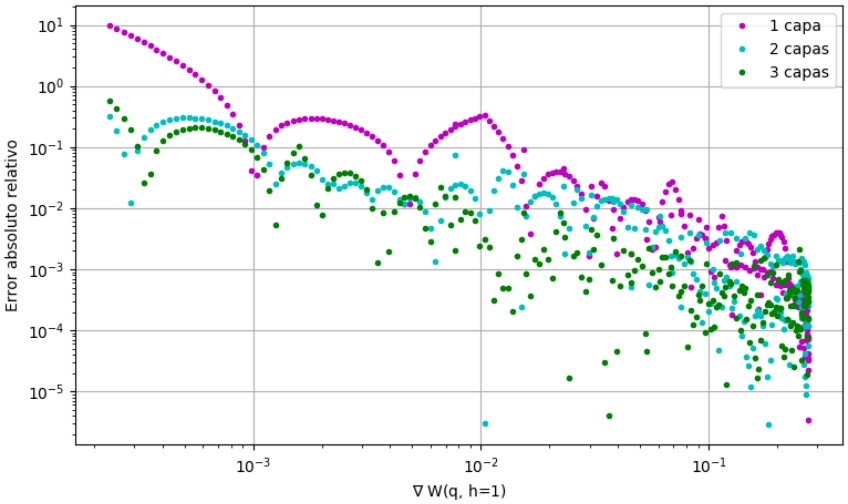


Figura 7: Error absoluto relativo (proporción entre error absoluto y valor real) en función de $\nabla W(q)$, en escala logarítmica. Se aprecia cómo los errores de más peso se producen en los valores más pequeños. Se usan puntos para evitar confusión ya que $\nabla W(q)$ no es biyectiva.

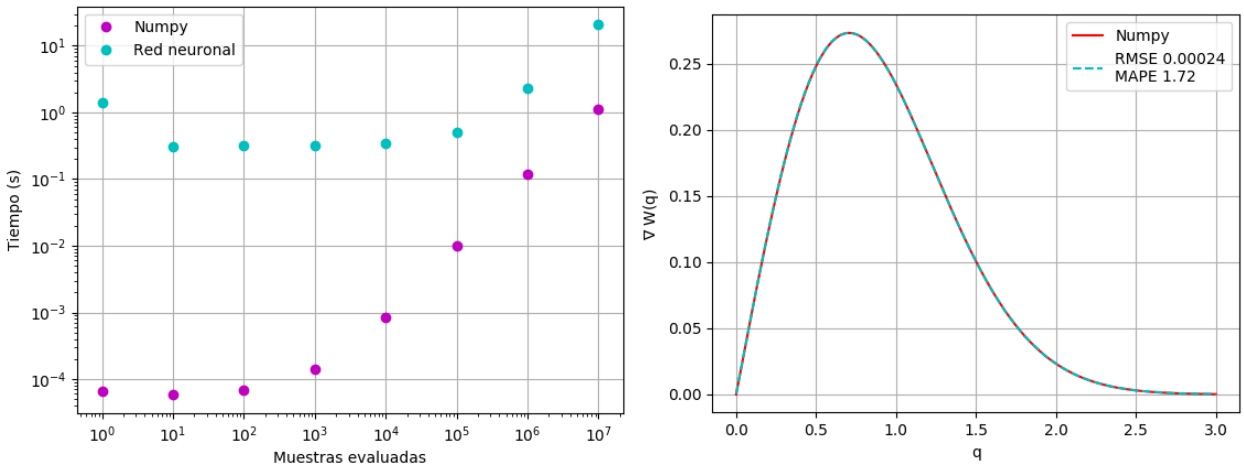


Figura 8: Resultados para la predicción de la divergencia del kernel. A la izquierda la comparativa de tiempos de ejecución. A la derecha la función por cálculo exacto y por red neuronal, superpuestas. $RMSE = 2.4 \times 10^{-4}$, $MAPE = 1.72$.

El perspectiva no mejora respecto al apartado anterior: el uso de la red neuronal en el cálculo de la divergencia no ofrece ventajas.

4.2.3. Ecuación de continuidad

La ecuación de continuidad se simplifica de manera similar a cómo se ha venido haciendo en las secciones anteriores. Ahora la función, sin embargo, depende de la distancia y de la diferencia de velocidades. Esta última es un vector (al igual que la distancia inicialmente) que se puede descomponer en magnitud y vector unitario.

$$D_{ij}(q_{ij}, \omega ij) = -m_j(\mathbf{u}_i - \mathbf{u}_j) \cdot \nabla W_{ij} = \omega_{ij} \frac{2q_{ij}e^{-q_{ij}^2}}{\pi} \times \left(\frac{c_s m}{10h^2}\hat{\mathbf{u}}_{ij} \cdot \hat{\mathbf{r}}_{ij}\right)$$

En este caso, la entrada a la red consiste en dos nodos, mientras que la salida sigue siendo de uno sólo. Se puede volver a usar ReLU como función de activación. Los resultados de los diferentes entrenamientos son los siguientes:

1 capa	10	20	50	100	200	500
RMSE	0.022(15)	0.004(2)	0.0015(2)	0.00068(8)	0.00079(15)	0.00045(7)
MAPE	14000(2000)	2700(1200)	1750(250)	600(200)	800(500)	310(40)

2 capas	10	20	50	100	150	250
RMSE	0.0026(1)	0.0011(2)	0.0005(1)	0.00038(6)	0.00024(3)	0.00021(5)
MAPE	2600(150)	2100(1200)	250(130)	350(250)	220(150)	170(50)

3 capas	10	20	50	100	150	250
RMSE	0.0017(7)	0.00064(3)	0.000362(8)	0.00028(15)	0.00028(5)	0.00016(4)
MAPE	2100(900)	600(300)	380(320)	280(120)	190(180)	100(70)

Esta vez, al depender de dos variables, la red neuronal no consigue tan buenos resultados. En cualquier caso, para un número suficiente de entrenamientos, debería ser posible

rebajarse el error a un margen aceptable. Se escoge la red de tres capas con 250 neuronas, que presenta el mejor resultado. Como se puede ver a continuación, se consigue reducir el MAPE, es decir, los valores cercanos a cero se ajustan mejor que antes, mientras que el RMSE no mejora. Se representan las dos dependencias de la función en dos gráficas distintas, Los valores de la predicción se encuentran mayoritariamente superpuestos a los exactos, por lo que estos últimos apenas se pueden apreciar. Finalmente, obtenemos otra vez que los arrays de Numpy hacen un buen trabajo optimizando los cálculos, siendo mucho más eficientes que la red neuronal.

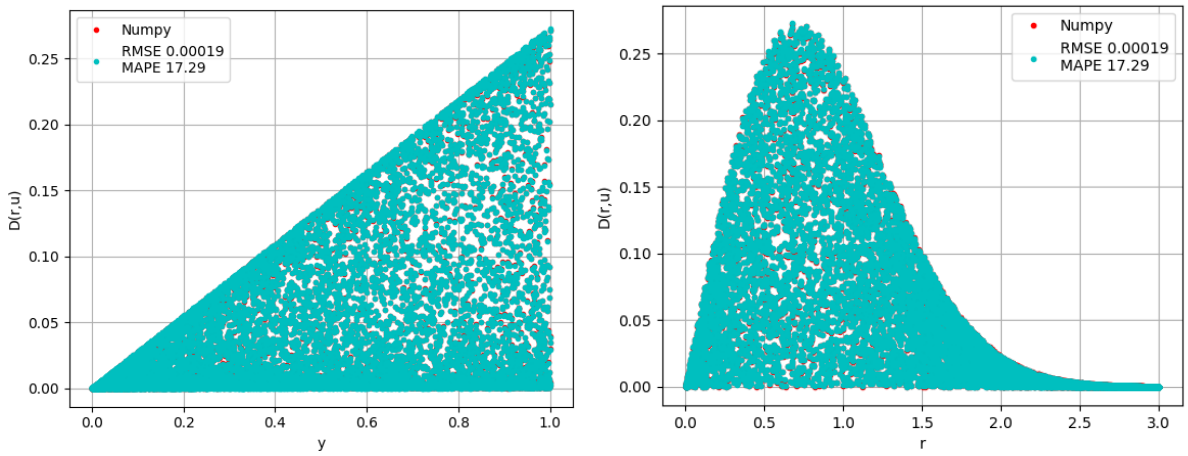
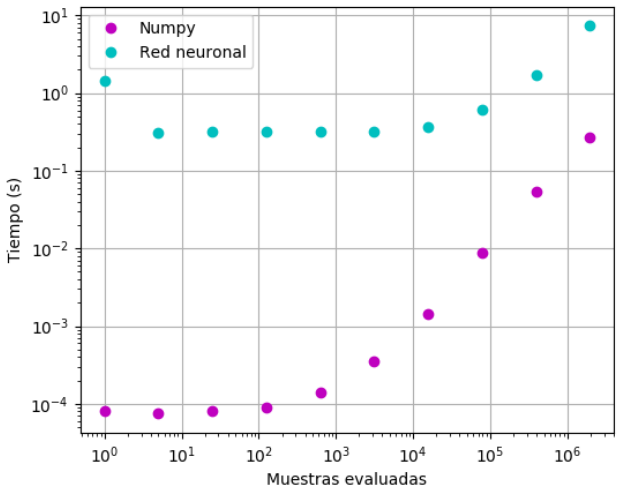


Figura 9: Resultados para la predicción de la ecuación de continuidad. A la izquierda se representa en función a la velocidad y a la derecha en función a la distancia. $RMSE = 1.9 \times 10^{-4}$, $MAPE = 17.29$.



5. Conclusión

Este trabajo se ha enfocado en explorar la viabilidad de la substitución directa de algunos de los cálculos que más tiempo llevan en interacciones hidrodinámicas. En concreto, se ha estudiado el caso de *smoothed particle hydrodynamics*, un modelo muy interesante debido a su potencial para adaptarse y aplicarse en distintas áreas de ciencia e ingeniería. Se ha analizado el cálculo del kernel gaussiano así como de su gradiente, para finalmente acercarse al caso algo más general de calcular el término de interacción en la ecuación de continuidad, resultando en todos los casos más eficiente el cálculo directo. Se ha mostrado, sin embargo, la capacidad de emular el comportamiento de distintas funciones y de converger a un resultado mejor a medida que crece la complejidad de la red y su entrenamiento.

De momento aparenta ardua la tarea de resolver la evolución del fluido de una forma más general, por ejemplo, prediciendo las propiedades de las partículas en el siguiente paso de tiempo en función de las de un instante dado. Supongamos el caso de SPH, donde las partículas interaccionan con sus vecinos más cercanos. Podría intentar predecirse las propiedades de una partícula en el instante siguiente dadas las de las partículas vecinas. Un problema en este caso podría radicar en el desconocimiento del número de partículas que entrarían en su dominio de soporte. Como se ha visto, una red neuronal queda encasillada en un número de entradas invariable desde el momento en que se empieza a entrenar la red. No obstante, una cualidad que demuestran estos artefactos es la flexibilidad: una sola red neuronal permite realizar regresiones o clasificaciones, pero éstas se pueden ir acumulando secuencialmente de forma que pequeñas redes con tareas muy concretas formen una mayor con grandes capacidades. Por tanto, tal vez se podría disponer de distintas redes especializadas al trato de diferente número de partículas u otras situaciones, orquestadas todas ellas por una o varias redes que clasifiquen el estado del fluido.

A parte de las redes neuronales más simples como las que se han revisado en este trabajo, actualmente se usan mucho las llamadas *redes convolucionales (CNN)*, así como las *long-short-term-memory (LSTM)*. Las primeras se inspiran en el córtex visual para procesar, comprender y detectar patrones en imágenes. Usadas conjuntamente con otros tipos de redes, son capaces de identificar relaciones espaciales, clasificar objetos e incluso predecir movimientos. En el estudio de fluidos, ser capaces de mejorar las simulaciones en base a redes neuronales entrenadas con experimentos reales podría ser interesante. La mejora de los modelos eulerianos mediante CNNs ya es un tema de estudio, donde la malla se puede usar como una "imagen" que procesar. Finalmente, las LSTM son redes con estados internos que permiten guardar información que ha pasado con anterioridad por la red, siendo muy eficientes en la predicción de series temporales y en el procesamiento de lenguaje, donde las conexiones entre palabras pueden prevalecer a gran distancia. Estas últimas, sin embargo, parecen estar más orientadas a secuencias aisladas, no siendo por ello descartables su aplicación en un sistema de partículas interaccionantes.

En definitiva potencial de las diferentes arquitecturas de redes neuronales artificiales para la física es muy grande, encontrando sitio donde se pueda disponer de una gran cantidad de datos que permitan mejorar la eficiencia de problemas tediosos. También pueden ayudar a identificar patrones pasados por alto o a comprender mejor ciertas situaciones mediante la inspección de la propia estructura de la red. No cabe duda, en cualquier caso, de que esta *inteligencia* sigue siendo muy artificial.

Referencias

[1] G. Palm. Warren mcculloch and walter pitts: A logical calculus of the ideas immanent in nervous activity. *Brain Theory*, page 229–230, 1986.

[2] J. Vonneumann and R. D. Richtmyer. A method for the numerical calculation of hydrodynamic shocks. *Journal of Applied Physics*, 21(3):232–237, 1950.

[3] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, Jan 1977.

[4] G. R. Liu and M. B. Liu. *Smoothed particle hydrodynamics: a meshfree particle method*. World Scientific, 2003.

[5] Martin Jeremy Robinson, J. J. Monaghan, and Paul Cleary. *Turbulence and Viscous Mixing using Smoothed Particle Hydrodynamics*. PhD thesis.

[6] Peter J. Smoothed particle hydrodynamics, Oct 2010.

[7] David J. Tritton. *Physical fluid dynamics*. Clarendon Press, 2009.

[8] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992.

[9] Daniel J. Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *Journal of Computational Physics*, 231(3):759–794, 2012.

[10] J. J. Monaghan. Sph compressible turbulence. *Monthly Notices of the Royal Astronomical Society*, 335(3):843–852, 2002.

[11] Ernst Hairer. Global modified hamiltonian for constrained symplectic integrators. *Numerische Mathematik*, 95(2):325–336, Jan 2003.

[12] J.p. Morris and J.j. Monaghan. A switch to reduce sph viscosity. *Journal of Computational Physics*, 136(1):41–50, 1997.

[13] Tom M. Mitchell. *Machine learning*. MacGraw-Hill, 1997.

[14] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[15] D.o. Hebb. The organization of behavior. Nov 2005.

[16] Geron Aurelien. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. OReilly Media, 2017.

[17] Kingma, Diederik P., Jimmy, and Ba. Adam: A method for stochastic optimization, Jan 2017.

[18] Pysph documentation.

[19] David M. Analysis of approximate nearest neighbor searching with clustered point sets, Jan 1999.

[20] Google Brain. Tensorflow: A system for large-scale machine learning, 2016.