Universitat Oberta de Catalunya

uoc.edu

Reducción de complejidad computacional en simulaciones de dinámica de fluidos mediante Aprendizaje Profundo y Representación Simbólica.

Jose Antonio Guerrero Barberán
Grado en Ingeniería informática
Inteligencia Artificial

David Isern Alarcó
Susana Acedo Nadal

01/2024

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Reducción de complejidad computacional en simulaciones de problemas físicos mediante Aprendizaje Profundo y Representación Simbólica*
Nombre del autor:	*Jose Antonio Guerrero Barberán*
Nombre del consultor/a:	*David Isern Alarcó*
Nombre del PRA:	*Susana Acedo Nadal*
Fecha de entrega (mm/aaaa):	01/2024
Titulación:	*Grado en Ingeniería Informática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Español*
Palabras clave	*superresolution, machine learning, cfd*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Este trabajo es un estudio comparativo que explora la aplicación del Aprendizaje Profundo y la Representación Simbólica para reducir la complejidad computacional en simulaciones de dinámica de fluidos a través de un aumento de resolución en las dimensiones espaciales de las simulaciones. El trabajo se centra en abordar los desafíos en la dinámica de fluidos computacional (CFD) mediante el aprovechamiento de técnicas avanzadas de aprendizaje computacional. Se adentra en los fundamentos de las ecuaciones en diferenciales parciales, particularmente la ecuación de Navier-Stokes, y discute varias soluciones computacionales al problema planteado, incluyendo el uso de redes neuronales profundas, redes convolucionales y modelos de representación simbólica. Se ha generado un conjunto de datos de simulaciones de CFD que representan un flujo turbulento a partir de condiciones iniciales aleatorias utilizando métodos numéricos. Este trabajo ofrece un análisis comparativo que estudia diferentes metodologías, modelos y arquitecturas que pueden utilizarse en el proceso de superresolución, destacando su eficacia en la producción de simulaciones de alta resolución a partir de datos de menor resolución en comparación con métodos de interpolación clásicos. Se pone énfasis en comprender los compromisos entre el coste computacional, la precisión y el potencial de los métodos de IA en CFD. Esta investigación también incluye un análisis de la complejidad computacional involucrada y las capacidades predictivas de los modelos implementados, aportando una clara perspectivas sobre la integración de la IA en la dinámica de fluidos y un conjunto de diferentes arquitecturas que pueden aprovechar el potencial del aprendizaje computacional.

Abstract (in English, 250 words or less):

This work is a comparative study that explores the application of Deep Learning and Symbolic Regression to reduce computational complexity in fluid dynamics simulations through an increase in resolution task in the spatial dimensions of the simulations. The work primarily focuses on addressing the challenges in computational fluid dynamics (CFD) by leveraging advanced machine learning techniques. It delves into the fundamentals of partial differential equations, particularly the Navier-Stokes equation, and discusses various computational solutions, including the use of deep neural networks, convolutional networks, and symbolic regression models. A dataset of CFD simulations that represent a turbulent flow from random initial conditions is computed using direct numerical methods. This work offers a comparative analysis studies different methodologies, models and architectures that can be used in the superresolution process, highlighting their effectiveness in producing high-resolution simulations from lower-resolution data. Emphasis is placed on understanding the trade-offs between computational cost, accuracy, and the potential of AI-driven methods in CFD. This research also includes an analysis of the computational complexity involved and the predictive capabilities of the implemented models, contributing valuable insights into the integration of AI in fluid dynamics and a set of different architectures that can take advantage of the potential of machine learning.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

1.1.1 Ecuaciones en Derivadas Parciales

Las Ecuaciones en Derivadas Parciales (en inglés: *Partial Differential Equations*, PDE) son fundamentales para el modelado matemático de muchos fenómenos en ciencia e ingeniería. Son ecuaciones que relacionan ratios de cambios de variables continuas relacionadas con dichos fenómenos. En contraste con ecuaciones diferenciales ordinarias (en ingles: *Ordinary Differential Equations*, ODE), donde aparecen derivadas con respecto a una única variable independiente, las PDE relacionan multitud de variables y las variaciones entre dichas variables.

En el mundo de la física, la ingeniería, la economía y otras áreas de la ciencia, las PDE son cruciales para el modelado de fenómenos dinámicos que involucran a multitud de variables, lo que en la práctica engloba la mayoría de los fenómenos del mundo real, tales como la propagación de calor, de ondas mecánicas, el fluido de un líquido o las deformaciones de un sólido.

La complejidad de las PDE normalmente impide obtener soluciones analíticas en sus casos generales, con lo que en la mayoría de los casos tenemos que recurrir a métodos numéricos donde se aproximan los diferenciales de las variables con números muy pequeños que permiten aproximar el comportamiento del sistema.

Gracias a los grandes avances en computación de las últimas décadas y su aplicación a este tipo de problemas, se ha conseguido computar y predecir el comportamiento de sistemas altamente complejos en una gran cantidad de disciplinas. Desde la predicción meteorológica y climatológica, hasta el diseño de vehículos espaciales, pasando por una multitud de aplicaciones en los diversos campos de la ciencia, las PDE y su solución computacional se ha convertido en una herramienta esencial.

1.1.2 Soluciones computacionales a Ecuaciones en Derivadas Parciales

Las soluciones computacionales a las PDE han sido, por lo tanto, una revolución en la aproximación, con un alto grado de fidelidad, de las soluciones a estos sistemas que serían de otro modo intratables analíticamente. No solo permiten resolver las PDE en escenarios variados y complejos, sino que facilitan la exploración de situaciones que son imposibles de replicar en experimentos de la vida real.

La metodología general consiste en transformar el problema de variables continuas a variables discretas con un relativo alto grado de granularidad y la resolución de cada paso de la ecuación de forma iterativa a partir de un dominio definido, unas condiciones iniciales para todos los puntos y unas condiciones de contorno en las cotas del dominio. Con ello, puede intuirse la limitación principal de dichas soluciones: al aumentar la granularidad nos

acercamos cada vez más a la solución continua, pero con el coste de aumentar el número de pasos necesarios en cada iteración hasta el punto de que el problema escala en su complejidad computacional de forma exponencial con la granularidad aplicada.

1.1.3 Dinámica de fluidos computacional y la ecuación de Navier-Stokes

La Dinámica de Fluidos Computacional (en inglés: *Computational Fluid Dynamics*, CFD) es un área de especialización en las ciencias computacionales y la física que busca realizar simulaciones mediante métodos numéricos de la ecuación de Navier-Stokes, la cual rige el comportamiento de la materia en estado líquido y gaseoso a escala macroscópica. Las metodologías utilizadas en CFD utilizan algoritmos computacionales para aproximar una solución a la ecuación de Navier-Stokes y analizar con ello el comportamiento de fluidos, siendo una herramienta indispensable en la industria y la investigación de problemas que involucran estos sistemas físicos.

La ecuación de Navier-Stokes encapsula los principios físicos de conservación de la masa, el momento y la energía, o la variación de éstos ante fuentes externas, en un sistema de ecuaciones en derivadas parciales. A pesar de su importancia en muchos sectores de las ciencias, la complejidad de estas ecuaciones hace que las soluciones analíticas sean imprácticas en la mayoría de los escenarios aplicables al mundo real, siendo crucial el uso de métodos numéricos discretos.

La aproximación de una solución a estos sistemas dinámicos y complejos ha sido crucial para el avance de la ciencia y la tecnología en numerosos campos de estudio, ofreciendo una importante información y capacidad de predicción sobre este tipo de problemas. Para ello se divide un dominio físico en unidades o celdas discretas lo suficientemente pequeñas como para representar el funcionamiento general del problema. La exactitud de la simulación es inversamente proporcional al tamaño de las celdas, pero también aumenta el coste computacional. Por ello las limitaciones de granularidad en estas aproximaciones todavía son una gran fuente de disparidad entre el comportamiento simulado y el real, especialmente en sistemas altamente complejos y grandes como el modelado del clima y la meteorología u otros sistemas altamente caóticos.

1.1.4 Inteligencia artificial y aprendizaje computacional

Este trabajo busca aplicar nuevas herramientas en el campo de la inteligencia artificial y el aprendizaje computacional, las cuales han tenido un inmenso avance en los últimos años, a este tipo de problemas. Estos sistemas tienen la capacidad de ser entrenados y aprender en base a la optimización de sus parámetros a partir de un conjunto de datos, e inferir comportamientos con un alto grado de complejidad. Las capacidades emergentes de estos sistemas, en muchos casos inesperadas y con un alto grado de capacidad predictiva, están siendo aplicadas a un gran número de campos de estudio y es posiblemente la gran revolución del siglo XXI.

Es por ello por lo que el uso de estas técnicas está cada vez más presente en el campo de la física y, como caso de estudio de este trabajo, la Dinámica de Fluidos Computacional. Con ello, el uso de este tipo de modelos puede permitir, potencialmente, mejorar los resultados de las soluciones dadas por las CFD sin incurrir al coste computacional equivalente de realizar las simulaciones directas a una mayor resolución.

1.1.5 Redes neuronales

Las redes neuronales[1], en su forma más genérica, son el fundamento en el cual se basan los modelos de aprendizaje profundo que utilizamos en este trabajo. En su forma más básica, cada paso de una capa a la siguiente consiste en dado un vector de datos de entrada $\vec{x}$, aplicar una función tal que:

$$\{f: \vec{x} \to \vec{y} \mid \vec{y} = \sigma(W * \vec{x} + \vec{b})\}$$

Donde si n es el tamaño del vector $\vec{x}$ y m es el tamaño del vector $\vec{y}$, W es una matriz de tamaño $(\mathrm{m} * \mathrm{n})$ y $\vec{b}$ es un vector de tamaño m. La matriz W es denominada matriz de pesos y el vector $\vec{b}$ el vector de *bias* o sesgo. Cada elemento de los vectores se denomina, de forma análoga al funcionamiento de los sistemas nerviosos de los animales, *neurona*.

La función σ es una función de activación que se aplica al resultado de la multiplicación matricial y la suma del sesgo. Esta función de activación es una operación no lineal que permite al modelo capturar patrones no lineales. Son ejemplos de funciones de activación que utilizaremos en este trabajo:

$$ReLu(x) = \max(0, x)$$
$$PReLu(x) = \max(\alpha x, x)$$

Siendo α un parámetro entrenable del modelo.

Con esto se define la propagación de una capa hacia la capa siguiente (*forward propagation*). La consecuente aplicación de un conjunto de capas da el resultado final en un vector de salida del modelo.

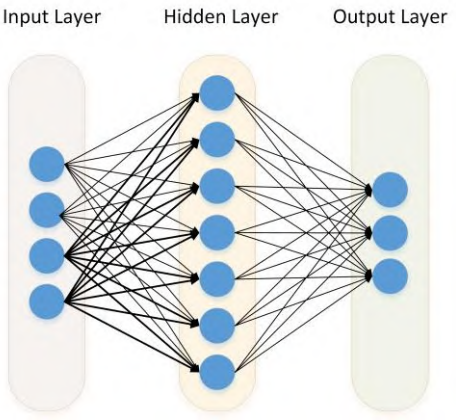


Figura 1: Red neuronal. Fuente: marktorr.com

La base del entrenamiento de las redes neuronales consiste en ajustar los parámetros (pesos) de la matriz de pesos W y el vector de sesgo $\vec{b}$ de forma iterativa para conseguir un conportamiento deseado en la capa final.

1.1.6 Modelos de aprendizaje profundo

Los modelos de aprendizaje profundo son un subconjunto de algoritmos de aprendizaje automático que utilizan múltiples capas de redes neuronales descritas en el apartado anterior. Este tipo de modelos han ganado una gran popularidad en los últimos años gracias a su habilidad de aprender comportamientos complejos.

La evolución de las redes neuronales profundas puede resumirse en dos períodos:

Inicios (1940s - 1980s): Los inicios de la redes neuronales datan a los años 40, y se consolidan con el desarrollo del perceptrón[2] por Frank Rosenblant en los años 50, sentando las bases del futuro desarrollo. A pesar de este importante avance conceptual, las limitaciones en la capacidad de cómputo el siglo XX mantuvieron limitados al potencial de esta metodología.

Renacimiento (2006-Presente): Las mejoras en la eficiencia de los algoritmos de optimización y el aumento de la capacidad de cómputo, especialmente con la paralelización de operaciones matriciales disponible en la arquitectura de las GPU [3] llevó a un resurgimiento del aprendizaje profundo. Especialmente en los últimos años, hemos visto una mejora sin precedente en la capacidad de estos modelos.

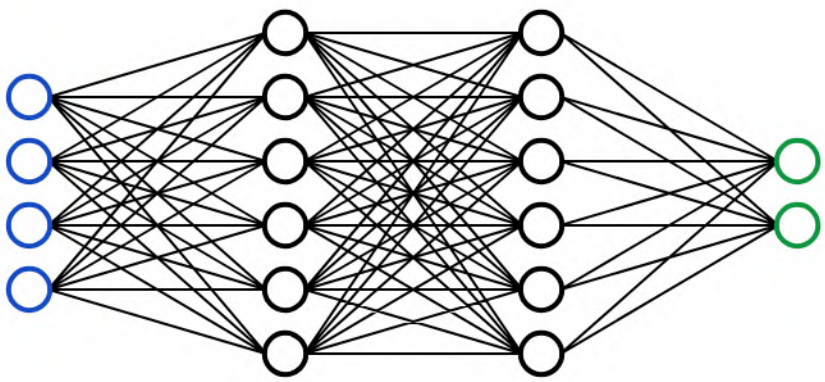


Figura 2: Red de dos capas profundas. Fuente: victorzhou.com

Al aumentar la profundidad del modelo (número de capas) y el número de neuronas, se ha observado como los modelos pueden aprender patrones en los datos altamente complejos y no lineales. Especialmente en el mundo actual, donde la gestión, almacenamiento y procesamiento de datos es muy importante, estos modelos han sido capaces de revolucionar la manera en que se realizan muchas tareas y han contribuido significativamente a los avances e innovación tecnológicos.

1.1.7 Optimización de modelos de redes neuronales

La optimización de modelos de aprendizaje profundo involucra a una tecnología matemática denominada el *descenso de gradiente*. Este proceso es fundamental para entrenar redes neuronales y consiste en el proceso iterativo de minimizar el resultado de aplicar una función de coste a la salida del modelo mediante la modificación de los parámetros de las matrices de pesos y vectores de sesgo de un modelo[4].

Así, se define en primera instancia una función de coste multivariable $F(\boldsymbol{x})$, que se aplica al output del modelo, la cual es diferenciable. Se calcula entonces el gradiente de dicha función $\nabla F(\boldsymbol{x})$, el cual, al ser la salida del modelo dependiente de todos sus parámetros, es un vector en el espacio de variables del modelo, que denominaremos $\boldsymbol{a}$. Como $\nabla F(\boldsymbol{a})$ apunta en la dirección de crecimiento de la función de coste en dicho espacio, podemos alterar el punto en el que el modelo se encuentra en el el espacio en la dirección opuesta:

$$\boldsymbol{a_{n+1}} = \boldsymbol{a_n} + \gamma \nabla F(\boldsymbol{a})$$

Donde γ es una constante denominada *learning rate* o ratio de aprendizaje.
Con lo que las variables del modelo se desplazan en la dirección en la que se minimiza la función de coste. Así, aplicando este proceso de forma iterativa, cada paso altera los parámetros del modelo optimizándolo de forma acorde a la minimización de la función de coste.

Podemos encontrar distintas estrategias en lo que respecta a bajo que criterios realizamos cada iteración del descenso de gradiente:

- Descenso de gradiente por *batch*: Utiliza todo el conjunto de datos de entrenamiento para calcular los gradientes antes de realizar un paso en la iteración. Así, la negativa del gradiente indica la dirección media de decrecimiento de la función de coste para todos los datos.

- Descenso de gradiente estocástico: Cada paso de la iteración se realiza utilizando una única muestra, lo cual aumenta el ruido, pero ayuda a escapar mínimos locales.

- Descenso de gradiente por *mini-batch*: Se computa cada iteración utilizando un subconjunto de todos los datos de entrenamiento. Presenta un compromiso entre ambos métodos antes mencionados y es normalmente la metodología más práctica, por lo que es la que se utiliza en este trabajo.

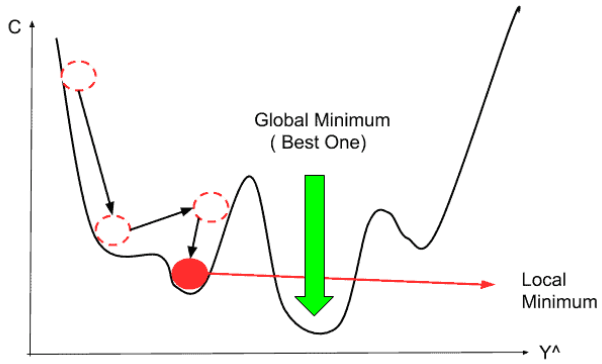


Figura 3: Proceso de descenso de gradiente hasta mínimo local. Fuente: mltut.com

El proceso de aplicar iterativamente el descenso de gradiente no está libre de potenciales problemas, ya que es necesario ajustar el ratio de aprendizaje para poder encontrar el mínimo absoluto de la función de coste sin quedar atrapados en un mínimo local, evitar problemas como la explosión de gradiente (*gradient explosion*) donde los gradientes calculados son tan grandes que no permiten al modelo alcanzar estabilidad, o la desaparición del gradiente (*vanishing gradient*) donde los gradientes son tan pequeños en la región que no permiten al modelo avanzar hasta mínimos, así como evitar sobreajustar los modelos a un conjunto de datos de entrenamiento que no le permitan encontrar patrones generales aplicables a datos con los que no se ha entrenado.

1.1.8 Redes neuronales convolucionales

Las redes neuronales convolucionales o *CNN* (*Convolutional Neural Networks*) [5] son un tipo de redes neuronales que funcionan en base a la aplicación y optimización de filtros o *kernels*. En lugar de aplicar una matriz de pesos a un vector de entrada unidimensional, estos modelos aplican un conjunto de operaciones consistente en la multiplicación de un grupo de filtros filtro por cada una de las potenciales posiciones en un tensor de entrada, que en este caso puede ser multidimensional. Con la aplicación de estos filtros, el modelo permite capturar patrones comunes e invariantes que aparezcan en las distintas posiciones de los datos de entrada y en la representación abstracta de las capas intermedias.

El mecanismo mediante el cual el filtro es aplicado a cada posición de los datos de entrada consiste en realizar una operación de producto interno entre el filtro y un subconjunto de los valores de entrada centrados en dicha posición. El escalar resultante es el valor del tensor de salida para la posición correspondiente. En la siguiente figura se representa una convolución de un filtro en una matriz de dos dimensiones (tensor de orden 2).

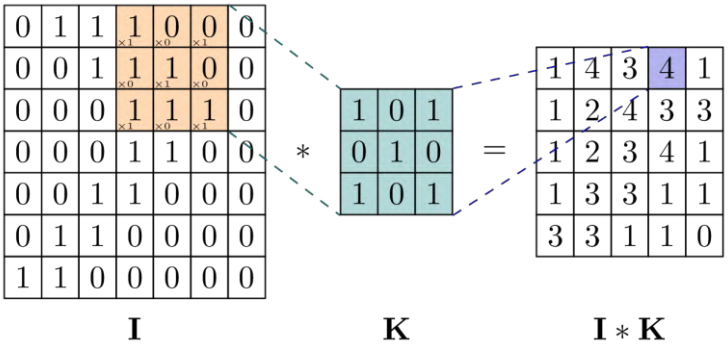


Figura 4: Aplicación de convolución de un filtro K sobre matriz I. Fuente: TikZ.net

De esta forma el tensor de salida representa de forma abstracta un valor correlacionado con la afinidad del filtro a cada posición del tensor de entrada, pudiendo detectar el patrón encapsulado en el filtro para cada posición de la imagen. Nótese que un filtro de tamaño *n* no puede aplicarse a los *floor(n)* elementos del borde del tensor, siendo *floor(x)* una función de redondeo al entero inferior, con lo que si se quiere mantener la misma dimensión en el tensor de salida es necesario utilizar un *padding*, donde se amplia el tamaño del tensor con ceros para poder aplicar el filtro a todas las posiciones de entrada. La aplicación del filtro sobre el tensor de entrada puede realizarse "desplazándolo" más de una unidad, lo que se denomina *stride*.

La operación de convolución realiza de forma natural un downsampling o reducción de la dimensión de los datos de entrada, y en contraste en este trabajo se hace uso de la operación de convolución traspuesta, también denominada deconvolución[6], que aplicada al upsampling consite en hacer un padding de ceros inicial intercalado a los datos de entrada, aumentando de forma efectiva su dimensión y aplicar un conjunto de filtros a este tensor de la misma forma que en la convolución. Ajustando el tamaño del *kernel* y el *stride* podemos conseguir aumentar el tamaño de los datos de entrada.

Una capa de convolución en el contexto del aprendizaje profundo consiste en aplicar un conjunto determinado de N filtros y obteniéndose N tensores de salida. De igual forma que a los pesos y sesgos de las capas neuronales *fully connected* descritas anteriormente, el algoritmo de descenso de gradiente permite optimizar los filtros utilizados de manera que consigan extraer información relevante de los datos de entrada.

Gracias a la capacidad de capturar patrones espacialmente invariantes, estos modelos son comúnmente aplicados al procesamiento de imágenes (aunque no exclusivamente), y de hecho han revolucionado este campo en los años recientes en sus aplicaciones como el reconocimiento de imágenes[7], la detección de objetos [8] o la generación de imágenes [9].

El uso de redes convolucionales es probablemente el más evidente en las tareas de upsampling y es la herramienta principal que el State-Of-The-Art utiliza en el tratamiento de imágenes. Por ello, se utilizan en este trabajo con la intención de extraer así los patrones espacialmente invariantes de los sistemas físicos [10] [11].

1.1.9 Modelos de representación simbólica

Los modelos de representación simbólica, también llamados de regresión simbólica, [12] presentan un mecanismo alternativo al aprendizaje profundo para encontrar patrones en un conjunto de datos. Estos modelos buscan encontrar una expresión matemática interpretable que se ajuste a dichos datos. Para ello se diseña una estructura de datos en forma de árbol donde los nodos representan operadores matemáticos u operandos, permitiendo así en la reconstrucción en base a unos operandos de entrada de un valor de salida computando dicho árbol desde sus hojas hasta la raíz.

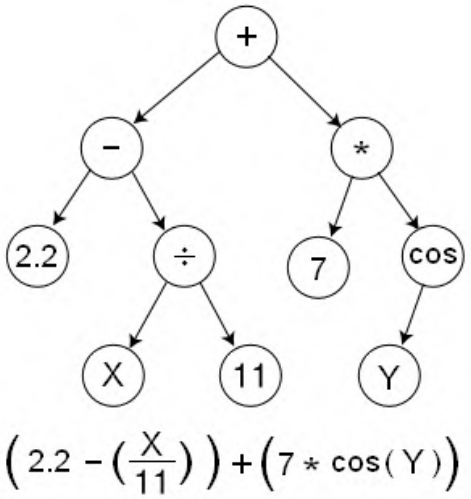


Figura 5: Ejemplo de árbol representando una ecuación. Fuente: Wikipedia.

Estos modelos son optimizados normalmente utilizando algún tipo de exploración del espacio del conjunto de ecuaciones posible mediante algoritmos genéticos, que seleccionan de forma iterativa, en base a una función de coste o *fitness*, un subconjunto de una población de árboles generados y mutados en un proceso estocástico. Tras un número de iteraciones determinado, se espera que este proceso permita encontrar el árbol que representa una relación existente en los datos de entrada y de salida que minimiza el coste.

Las ventajas de estos sistemas con respecto a los anteriores es la interpretabilidad que ofrece el resultado, que puede ser traducido fácilmente a una ecuación matemática clásica comprensible. Por otro lado, en contrate cn los métodos de aprendizaje profundo suelen tener un mayor coste computacional en el entrenamiento, al ser un proceso estocástico en lugar de una dirigido como nos ofrece el descenso de gradiente.
Estos modelos son de especial de interés en campos como la física donde se valora la capacidad de interpretar el funcionamiento de los modelos[13] [14] [15] [16], en contraste con el comportamiento de “caja negra” de las redes de aprendizaje profundo que son difícilmente interpretables.

1.2 Objetivos del Trabajo

El objetivo principal de este trabajo es la exploración y desarrollo de metodologías para el aumento de resolución del resultado de simulaciones mediante métodos numéricos directos a problemas físicos, en concreto a la

dinámica de fluidos regida por la ecuación de Navier-Stokes y el aumento de resolución inferido por modelos de aprendizaje computacional.

Este trabajo se centra realizar un análisis comparativo entre distintos modelos y aproximaciones que permitan obtener una visión general de qué metodologías son más satisfactorias para la tarea de producir simulaciones de alta resolución a partir de simulaciones de menor resolución. Se inspira, por un lado, en el proceso de upsampling de imágenes y pretende evaluar la capacidad de modelos utilizados en esta tarea en aplicaciones físicas, y por otro en el uso de modelos de representación simbólica que pueden ser capaces de encontrar relaciones análogas a las interpolaciones clásicas pero que puedan reflejar la interdependencia de las variables físicas.

Con ello, se busca encontrar modelos capaces de inferir comportamientos de sistemas físicos a partir de datos generados en simulaciones y entrenados de forma supervisada y se espera que dichos modelos sean capaces de reducir el coste computacional total de una simulación manteniendo su capacidad predictiva y minimizando el error inducido por el uso de estos modelos.

1.3 Enfoque y método seguido

La metodología general que se ha seguido consiste en, en primer lugar, la generación de datos referentes al problema de estudio. En la sección 2.1 se detallan los problemas físicos que se utilizarán como dominio de estudio y las condiciones simuladas. Se generarán un conjunto de datos, resultado de dichas simulaciones, que reflejen el resultado de un problema de CFD dados un conjunto variado de condiciones iniciales y de contorno tanto a alta como baja resolución espacial.

Una vez se generen los datos del dominio del problema, se buscará desarrollar un conjunto de modelos de aprendizaje computacional que premitan realizar la transformación:

$$X \rightarrow Y' \approx Y$$

Donde X es un *timestep* de una simulación de baja resolución, Y es el correspondiente *timestep* de alta resolución, e Y' es una aproximación resultante de aplicar el modelo a la simulación de baja resolución.

Se implementarán un conjunto de modelos de aprendizaje computacional los cuales serán entrenados de forma supervisada para optimizar la solución Y' buscada. Este entrenamiento se realizará mediante tres estrategias:

1. Se optimizará el modelo con una función de coste resultado de computar la diferencia o error entre Y e Y', siendo Y' la salida del modelo
2. Se optimizará el modelo con una función de coste resultado de computar la diferencia o error entre Y e Y', siendo Y' el resultado de sumar a la salida del modelo una interpolación bicúbica de X. Es decir la salida de los modelos son los residuos de la interpolación bicúbica con respecto al valor de alta resolución.

3. Optimización del modelo intentando reducir la desviación del resultado Y' de las condiciones físicas impuestas por las ecuaciones en derivadas parciales del problema. Este paradigma se ha denominado por otros autores como physics-driven o physics-informed[17] [18] [19].

En nuestro caso, el paradigma *physics-driven* obtiene sus condiciones de la propia ecuación de Navier-Stokes para un fluido incompresible.

A continuación, se realiza una evaluación del rendimiento de los modelos en la tarea planteada y se realizará un análisis comparativo de las distintas metodologías y modelos. Se considerará la viabilidad de los modelos y la estrategia desarrollada para la mejora del coste computacional y se estudiará la complejidad computacional en cada modelo utilizado, comparándose con métodos tradicionales de simulación directa.

1.4 Planificación del Trabajo

La planificación del trabajo se divide en las siguientes tareas:

1. Definición y planificación. Se definirán el alcance, objetivos y metodología a utilizar durante el desarrollo del trabajo.

2. Investigación del State-Of-The-Art: Se realizará una investigación profunda del State-Of- The-Art y los avances recientes en aquellos aspectos que sean de interés para el trabajo realizado, principalmente en la generación de los datos simulados, los modelos que podamos utilizar y las distintas metodologías aplicables a cada paso y en la comparación final entre modelos.

3. Obtención de datos: Utilizando simulaciones de los problemas de estudio se generarán un conjunto de datos que se utilizarán en el entrenamiento de los modelos.

4. Implementación de modelos y entrenamiento: Se implementarán los modelos a utilizar prestando atención a la casuística y particularidades de cada modelo y se entrenarán en base a los datos generados.

5. Análisis de resultados: Se hará un análisis de los resultados obtenidos en profundidad, se utilizarán métricas para evaluar cuantitativamente el rendimiento de cada uno de los modelos utilizados y se realizará un análisis de la complejidad computacional de cada metodología utilizada.

6. Elaboración de la memoria.

7. Preparación de la defensa.

8. Defensa del proyecto de fin de grado.

El siguiente diagrama de Gantt muestra las tareas planteadas y la planificación temporal de dichas tareas:

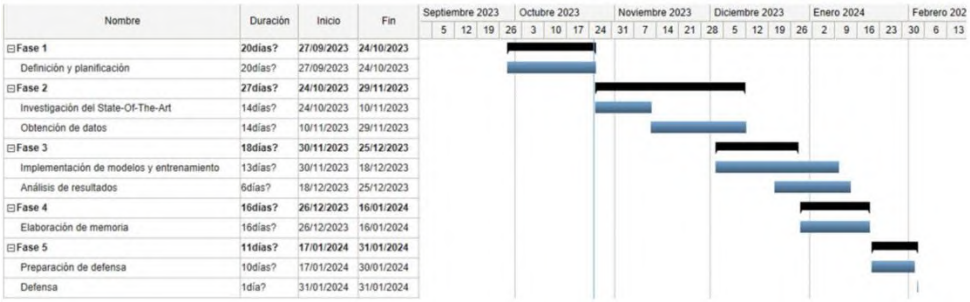


Figura 6: Diagrama de Gantt del desarrollo del trabajo

1.5 Breve sumario de productos obtenidos

1. Conjunto de datos correspondientes a 35 simulaciones de dinámica de fluidos en dos dimensiones a resolución espacial 255x255 y 320 timesteps, disponible en Kaggle [20].

2. Algoritmos de entrenamiento de modelos de aprendizaje profundo utilizando la librería pytorch.

3. Algoritmo de entrenamiento mediante algoritmos genéticos de árboles de representación simbólica.

4. Modelos entrenados en el proceso de upsampling de la dimensión espacial de simulaciones físicas.

5. Estudio comparativo de proceso de entrenamiento, rendimiento y coste computacional de los modelos implementados.

Los algoritmos, scripts y modelos entrenados utilizados se encuentran disponibles en el repositorio *github* del trabajo[21].

1.6 Breve descripción de los otros capítulos de la memoria

Capítulo *2.1 Definición del dominio del problema*: se concreta el dominio de las simulaciones generadas, y se definen las constantes, parámetros y fronteras del problema físico simulado.

Capítulo *2.2 Framework, software y hardware utilizado*: se detallan las herramientas utilizadas en el desarrollo del trabajo.

Capítulo *2.3 Datos*: Se explican los detalles sobre las condiciones de las simulaciones con las cuales se generan los datos que se utilizan en este trabajo.

Capítulo *2.4 Aprendizaje profundo*: Se detallan los modelos implementados y se detalla el proceso de entrenamiento seguido. Se establecen métricas y comparativas que permitan evaluar los modelos tanto en su rendimiento como en su complejidad computacional. Se presentan los resultados obtenidos. Finalmente se valora la viabilidad de la metodología utilizada.

Capítulo *2.5 Representación simbólica*: Se explica el algoritmo utilizado para el modelo de representación o regresión simbólica y su optimización mediante algoritmo genético y se presentan los resultados obtenidos.

Capítulo *3 Conclusiones*: Se plantean las conclusiones extraídas de la realización de este trabajo.

2. Resto de capítulos

2.1 Definición del dominio del problema: Simulaciones de CFD mediante métodos numéricos directos.

La ecuación en derivadas parciales que rige el problema de estudio, que es la ecuación de Navier-Stokes para un fluido incompresible en dos dimensiones es que no esté sometido a fuerzas externas, es:

$$\nabla\vec{v} = 0$$
$$\frac{\partial\vec{v}}{\partial t} + (\nabla\vec{v})\vec{v} = -\frac{1}{\rho}\nabla \mathrm{p} + \nu\nabla^2\vec{v}$$

Donde el vector $\vec{v}$ es el vector de velocidad de dos componentes, p es la presión, ρ es la densidad del fluido (constante al ser incompresible), ν es la viscosidad del fluido. La primera ecuación se corresponde con la conservación de la masa cuando la densidad es constante, y la segunda es la conservación del momento.

El comportamiento general del fluido, y en concreto su tendencia a la turbulencia, está caracterizado por el llamado número de Reynolds el cual relaciona la viscosidad con la velocidad y las características físicas del problema.

$$Re = \frac{uL}{\nu}$$

Siendo L la longitud característica del problema, que es igual a la raíz cuadrada del área para el caso de dos dimensiones, y u es la velocidad del fluido.

El dominio de los datos generados que se utilizaran como entrada de los modelos es por lo tanto la resolución numérica por método de simulación directa de la ecuación de Navier-Stokes para un fluido incompresible en 2D en un área, constante a lo largo de todas las simulaciones, consistente en un cuadrado de lado 1 de 255x255 celdas ($\Delta x = \Delta y = 1/255$ m). La simulación se realiza durante 320 iteraciones o *timesteps*, en un total de 10s de simulación ($\Delta t = 1/32$ s).

Se opta por fijar la densidad del fluido a 1 y ajustar la viscosidad (ν) del fluido para que el número de Reynolds sea aproximadamente $10x10^4$, con el objetivo que las simulaciones reflejen así un flujo turbulento[22].

Dado que necesitamos un dominio homogéneo que se aplicable a todos los modelos, se opta por mantener la geometría del problema constante y alterar las condiciones iniciales para obtener comportamientos distintos.

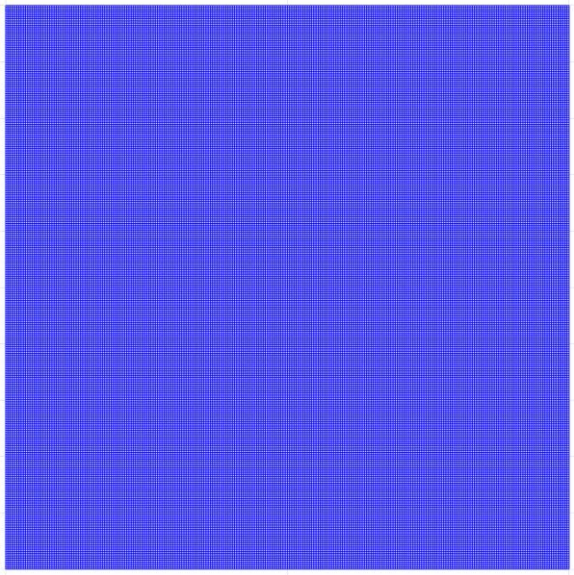


Figura 7: Visualización de malla utilizada

2.2 Framework, software y hardware utilizado

2.2.1 OpenFOAM

Para la realización de las simulaciones se ha utilizado el software OpenFOAM (del inglés *Open Field Operation and Manipulation*)[23]. OpenFOAM es un software gratuito y libre desarrollado por OpenCFD desde 2004. Este software permite simular modelos físicos descritos con derivadas parciales mediante discretización. Así, tiene un amplio campo de aplicaciones, desde simulaciones de dinámica de fluidos, en las cuales se centra, hasta reacciones químicas, ecuación de difusión de calor, electromagnetismo, o acústica.

OpenFOAM nos ofrece computación paralelizada y metodologías sofisticadas y eficientes. Con ello, y gracias también a su largo historial y su amplio uso en la industria, nos aseguramos de que el conjunto de datos utilizados refleja un sistema físico dentro de las garantías del State-Of-The-Art en el campo de las simulaciones.

El software se ha utilizado integrado en el software Matlab[24] y el toolbox FEATool Multiphysics, el cual nos permite definir el dominio de la simulación, el mallado, y las condiciones de contorno de una forma sencilla, visualizar los datos de la simulación y gestionar los datos para su exportación de forma eficiente.

2.2.2 Python

Los algoritmos implementados a lo largo del trabajo son realizados mediante el lenguaje de programación *python*[25]. *Python* es el lenguaje y framework más ampliamente usado en las aplicaciones de inteligencia artificial y aprendizaje computacional por los siguientes motivos:

- Facilidad de uso, simplicidad sintáctica y legibilidad, lo cual permite a los usuarios reducir el tiempo que se dedica a escribir o leer código y facilitando la implementación de cualquier proyecto.

- Amplia disponibilidad de librerías y frameworks orientados a la ciencia de datos y la inteligencia artificial, como Tensorflow, Pytorch, Scikit-learn, Pandas o Numpy, los cuales integran algoritmos escritos en C/C++ que reducen las limitaciones de eficiencia de cómputo de python.

- Estándar utilizado tanto en la comunidad online, el ámbito académico y de investigación como en el ámbito empresarial en aplicaciones de aprendizaje computacional, lo cual aumenta significativamente los recursos de consulta disponibles.

2.2.3 Pytorch

En la realización de este trabajo, se ha estudiado las diferencias, ventajas e inconvenientes entre las dos librerías utilizadas de facto en las aplicaciones de aprendizaje profundo, *Pytorch* [26], y Tensorflow/Keras [27]. Estas librerías ofrecen la capacidad de utilizar los módulos CUDA de las tarjetas gráficas Nvidia para paralelizar los cálculos tensoriales necesarios en el aprendizaje profundo, aumentando exponencialmente el rendimiento de los modelos en todas las fases del proyecto.

Tras analizar las ventajas e inconvenientes y realizar algunas aplicaciones de prueba, se opta por utilizar *Pytorch* dado su amplio uso en los años recientes y su mayor claridad y flexibilidad a la hora de implementar modelos, facilidad de *debugging* y mayor ajuste al funcionamiento imperativo de python. En contraste con Tensorflow, requiere de una mayor verbosidad y debemos definir de forma explícita los bucles de entrenamiento y el *cómputo hacia adelante* (*forward*), de un modelo desde sus datos de entrada hasta los datos de salida. Esta mayor verbosidad hace precisamente que el entorno sea más flexible, que se puedan definir estas acciones ajustándolo a las necesidades de la aplicación de forma explícita y se profundice de forma clara en el funcionamiento de los modelos y el proceso de optimización.

2.2.4 Hardware

El hardware utilizado es un procesador AMD Ryzen 5 5600X de 6 núcleos a 3.70GHz, 32GB de memoria RAM, y una tarjeta gráfica CUDA Nvidia GeForce 3080 de 10GB de VRAM con las que se realizan las operaciones tensoriales utilizando *pytorch*.

2.3 Datos

2.3.1 Simulaciones realizadas

Esta sección describe las condiciones de contorno y las condiciones iniciales de las simulaciones realizadas, las cuales definen junto con la ecuación de Navier-Stokes que rige su evolución, el comportamiento de las variables físicas.

Se opta por utilizar unas condiciones de contorno constantes en la frontera del dominio del problema, las cuales son la denominada *no-slip wall* [30], que implica que, siendo **u** la componente horizontal de la velocidad, **v** la componente vertical:

$$u(x = 0, x = 1) = 0$$
$$v(y = 0, y = 1) = 0$$

Estas condiciones de contorno se fuerzan en cada iteración de la simulación e implican de forma intuitiva que el fluido se encuentra encerrado en el dominio, es decir que no hay flujo normal de fluido a través de las paredes del dominio ni rozamiento con el mismo.

Así la variabilidad en las simulaciones viene dada por las condiciones iniciales del problema, las cuales son generadas de forma pseudoaleatoria y son las siguientes:

$$\begin{cases} u_0 0(x, y) = rand() * sin(rand() * 8 * (2 * \pi) * y + rand() * 2 * \pi) \\ v_0 0(x, y) = rand() * sin(rand() * 8 * (2 * \pi) * x + rand() * 2 * \pi) \\ p_0 = 0 \end{cases}$$

Siendo *rand()* una función que devuelve un valor pseudoaleatorio de distribución uniforme en el intervalo [0, 1). La componente pseudoaleatoria de las condiciones iniciales hace que el comportamiento del fluido sea caótico en cada simulación. Nótese que la condición de $p_0 = 0$ puede parecer a priori que no tiene sentido físico, pero recordamos que la ecuación de Navier-Stokes trata con la presión de forma diferencial, es decir con la variación de la presión, por lo tanto, un valor de presión 0 o negativo, como los encontramos en las simulaciones, solo indicaría, físicamente, la presión relativa a un valor de referencia, como podría ser, por ejemplo, la presión atmosférica en una aplicación práctica.

Las siguientes figuras muestran a modo de ejemplo una representación del estado inicial y final de una simulación:

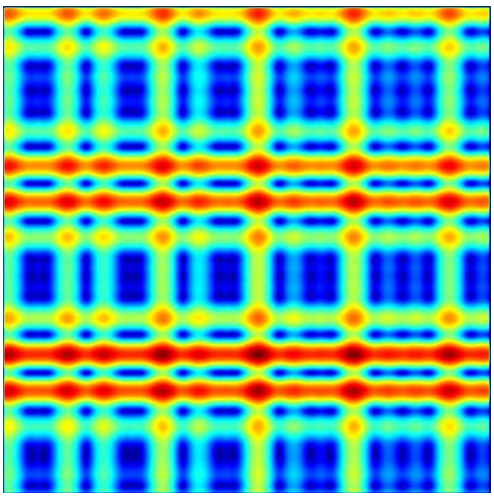


Figura 8: Módulo de la velocidad en condiciones inciales (t=0)

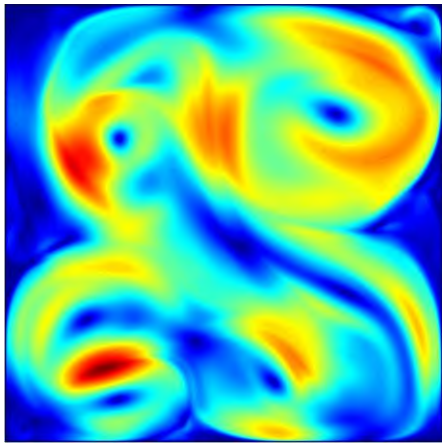


Figura 9: Módulo de la velocidad en el estado final de simulación (t=10)

Observamos como encontramos un comportamiento caótico y turbulento en la evolución del fluido. En el repositorio github del proyecto se incluyen también *gifs* animados que permiten visualizar más fácilmente la evolución de las simulaciones en el tiempo.

Las variables resultantes de la simulación tienen media 0, dado que las componentes de la velocidad oscilan entre valores positivos y negativos y la presión de referencia en las condiciones iniciales es 0. Mostramos como ejemplo un histograma de la distribución de las distintas variables en varias simulaciones.

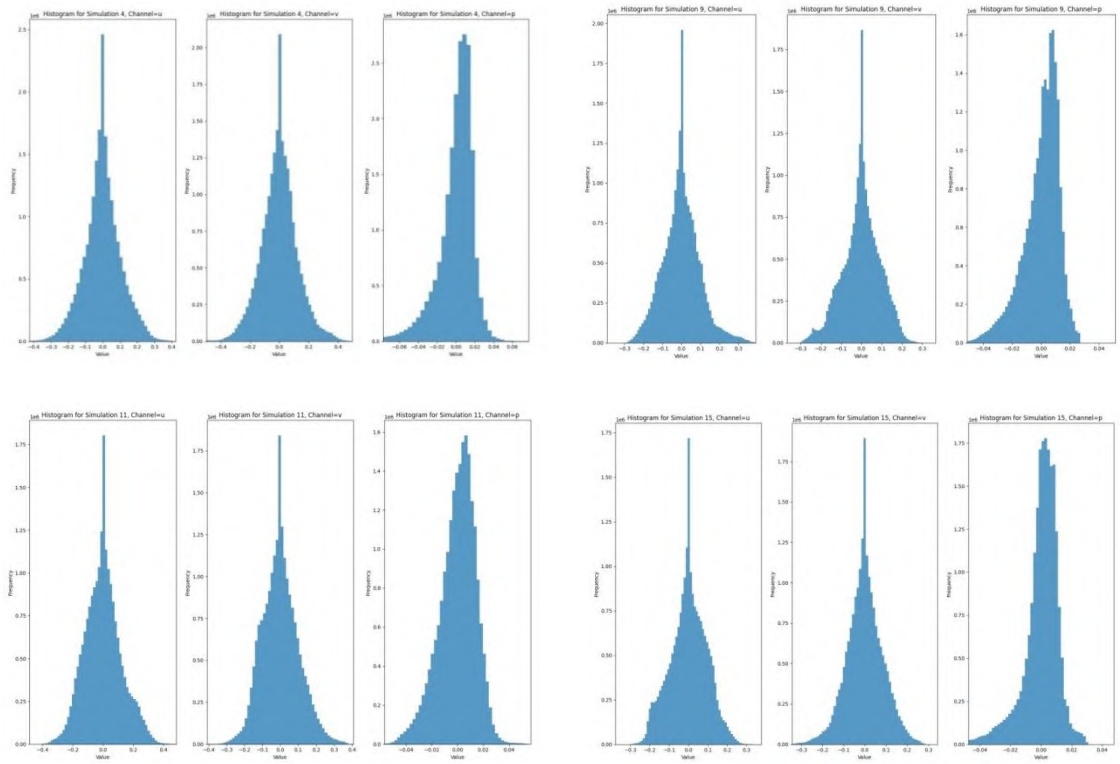


Figura 10: Distribución de variables en simulaciones 4, 9, 11 y 15

Se completan un total de 35 simulaciones, que nos da un dataset de aproximadamente 14 GB. Se ha publicado este dataset en Kaggle[20] para promover la reutilización y explotación de los datos generados. Los datos se encuentran codificados en un único tensor de numpy con la forma (N,T,W,H,C) donde:

- S: identifica a una simulación.
- T: identifica a un timestep.
- W: es el ancho de la malla.
- H: es el alto de la malla.
- C: es el canal que representa cada variable física, en nuestro caso:
    - Canal 0: Valor de la componente horizontal de la velocidad.
    - Canal 1: Valor de la componente vertical de la velocidad.
    - Canal 2: Valor de la presión.

En concreto, las dimensiones del tensor son (35, 320, 255, 255, 3).

Para el tratamiento y preparación de los datos de entrada de los algoritmos de aprendizaje profundo se realiza un escalado de los datos dividiendo cada variable de cada simulación por su desviación típica a lo largo de dicha simulación. Esta desviación debe ser almacenada con el objetivo de poder recuperar los datos originales de cara a calcular los residuos físicos.

Los datos se separan en datos de entrenamiento, evaluación y test de manera que tenemos 18 simulaciones utilizadas en los datos de entrenamiento, 9 en los datos de evaluación y 8 en los datos de *test*. Además se guardan estos datos, con respecto al dataset en crudo publicado en Kaggle, en forma (N,T,C,W,H) al ser una práctica común y por defecto en las librerías utilizadas tener la dimensión de los canales antes que la dimensión espacial.

2.4 Aprendizaje profundo

2.4.1 Estrategias de optimización

En este trabajo se analiza la capacidad de los modelos aplicados al problema planteado en base a dos grupos de estrategias de optimización, las cuales se describen en este apartado.

2.4.1.1 Predicción directa de variables a alta resolución con respecto a predicción de los residuos de interpolación trivial.

Inspirado en trabajo previos consultados [28] [29], se han considerado dos mecanismos de predicción del modelo de la simulación a alta resolución a partir de la simulación de baja resolución. Por un lado, se pretende optimizar el modelo de forma supervisada para que el *output* sea, directamente, la matriz de tres canales (velocidad horizontal, velocidad vertical y presión) calculada por el modelo a partir de la representación en baja resolución de esta matriz. Es decir, en este paradigma la función *f* del modelo es optimizada para modelar de forma directa:

$$f: X \rightarrow Y' \approx Y$$

Por otro lado, en lugar de hacer que el modelo calcule directamente el valor final, se plantea la estrategia de hacer que el modelo calcule los residuos resultantes de una interpolación tradicional, es decir, en nuestro caso, la diferencia, entre la simulación de baja resolución interpolada a alta resolución mediante una interpolación bicúbica y la simulación de alta resolución correspondiente:

$$\{f: X \rightarrow Res \mid Res + Interp(X) \approx Y\}$$

Esto tiene la ventaja de que los outputs del modelo van a ser valores centrados en 0 con una distribución mucho más homogénea, lo cual se espera que facilite el entrenamiento. Una vez calculados los residuos, se suman a una interpolación bicúbica de los datos de entrada obteniendo con ello el valor análogo a la estrategia anterior, que pasamos de forma idéntica a la función de coste.

2.4.1.2 Minimización de diferencia en residuos de ecuaciones físicas

Dado que disponemos de información sobre el mecanismo físico que gobierna el funcionamiento de las variables en el dominio, podemos introducir un mecanismo de control que haga que la salida del modelo es compatible con dicho funcionamiento. Como la diferenciación de un conjunto de datos ya simulados, al contrario que la integración que implica resolver dichas ecuaciones es una tarea computacionalmente simple, podemos elaborar una métrica que mida la reducción de la disparidad del resultado con las ecuaciones en derivadas parciales.

En el caso de la ecuación de Navier-Stokes para un fluido incompresible, se busca que se cumplan las relaciones:

$$\nabla \vec{v} = 0$$
$$\frac{\partial \vec{v}}{\partial t} + (\nabla \vec{v})\vec{v} + \frac{1}{\rho}\nabla \mathrm{p} - \nu \nabla^2 \vec{v} = 0$$

Es decir, que se minimice mediante optimización los residuos de estas ecuaciones en derivadas parciales para todos los diferenciales espaciales y temporales. Para ello, se implementa una función de coste específica a las condiciones del problema que calcule estos residuos y los compare con los residuos de la representación de alta resolución. Con ello se espera conseguir no solo minimizar la desviación media del resultado esperado, sino también asegurar que el resultado mantiene coherencia con el mecanismo físico que lo rige. La función de coste que calcula estos residuos incorpora también un cálculo de las derivadas con respecto al tiempo, con lo que es posible que sea información relevante que pueda ser capturada por el modelo.

2.4.2 Modelos utilizados

Se acude a la bibligrafía disponible sobre el proceso de aumento de resolución de imágenes y videos y se implementan algunos modelos importantes del desarrollo del estado del arte en esta tarea, siguiendo el orden cronológico de su evolución. Se busca por lo tanto aplicar estos modelos al problema planteado, cuyos datos tienen una naturaleza implícitamente distinta a la del aumento de resolución de imágenes, tanto en la distribución de los datos como en la relación entre los canales, que en nuestro caso son variables físicas en lugar de canales de color. Por ello, no puede asumirse que el rendimiento de los modelos y las comparativas realizadas previamente en la tarea del aumento de resolución de imágenes se correlacione necesariamente con el rendimiento en el problema de este trabajo. En base a estos modelos, se expande la comparación con un modelo (en dos variantes) propuesto en este trabajo que hace uso de las capas y estructuras ya planteadas por dichos modelos y su bibliografía.

Todos los modelos se encuentran implementados en el archivo **model.py** del repositorio del trabajo [21].

2.4.2.1 Convolución transpuesta *naïve*

Se ha implementado un modelo sencillo para ejemplificar el funcionamiento de la convolución transpuesta, denominado en este trabajo NaiveTransConv consistente en dos capas de convolución transpuesta en dos dimensiones. Con un *kernel* de 2x2 y un *stride* de 2 posiciones, consiguen cada una un factor de superresolución de 2x, acumulando las dos un total de 4x en la salida del modelo. Cada capa utiliza el módulo de pytorch ConvTransp2D. Se aplican ambas capas sin ninguna función de activación, para ejemplificar así el funcionamiento base de un modelo estrictamente lineal.

Vemos su implementación en el siguiente código:

```
class NaiveTransConv(nn.Module):
    def __init__(self, numInputChannels, deep_channels, kernel=6):
        super(NaiveTransConv, self).__init__()
        self.convTrans1 = nn.ConvTranspose2d(in_channels=numInputChannels, out_channels=deep_channels,
                                    kernel_size=kernel, stride=2, padding = 2)
        self.convTrans2 = nn.ConvTranspose2d(in_channels=deep_channels, out_channels=numInputChannels,
                                    kernel_size=kernel, stride=2, padding = 2)
    def forward(self,x):
        x = self.convTrans1(x)
        x = self.convTrans2(x)
        return x
```

2.4.2.2 SRCNN

Se implementa el modelo propuesto en el trabajo Image Super-Resolution Using Deep Convolutional Networks [31], denominado SRCNN, el cual consiste en la siguiente arquitectura general, de acuerdo con la configuración propuesta en el *paper* original:

1. Operación de interpolación bicúbica: Primero la matriz de entrada es aumentada de resolución con un algoritmo de interpolación bicúbica en base a un factor de escala determinado, en nuestro caso 4x.

2. Capa de extracción de características (Feature extraction layer): En esta capa se extrae la información relevante de la imagen de baja resolution interpolada utilizando filtros en una capa de convolución en dos dimensiones de **d** filtros y un tamaño de kernel 9. Un *padding* de 4 mantiene la dimensión de la imagen constante. A continuación, se utiliza una función de activación ReLu.

3. Capa de mapeo no lineal (None-linear Mapping): En esta capa el modelo mapea la salida no lineal de la primera capa con una representación a alta resolución. Se utiliza otra capa de convolución de tamaño de kernel 5, **s** filtros y padding de 2 para mantener la dimensión de la matriz. Se utiliza de nuevo una función de activación ReLu.

4. Reconstrucción (Reconstruction): En esta última capa de convolución de kernel 5, un número de filtros equivalente al número de canales de entrada y padding 2, se genera la representación final de la matriz a alta resolución.

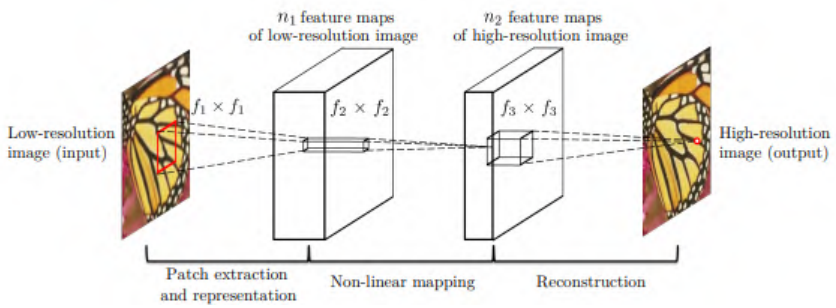


Figura 11: Arquitectura SRCNN[32]

En este trabajo se utiliza el modelo con los parámetros utilizados en el artículo original, con d = 64, s = 32 para un escalado de 4x y 3 canales. Se utiliza una inicialización de los parámetros del modelo como se hace en el trabajo original. El siguiente código muestra la implementación del modelo.

```
class SRCNN(nn.Module):
    #https://arxiv.org/abs/1501.00092v3
    def __init__(self, numInputChannels, d, s):
        super(SRCNN, self).__init__()
        self.upsample = nn.Upsample(scale_factor=4,mode='bicubic')
        # Feature extraction layer.
        self.feat_extraction = nn.Conv2d(in_channels=numInputChannels, out_channels=d, kernel_size=9,stride=1,padding=4)
        # Non-linear mapping layer.
        self.mapping = nn.Conv2d(in_channels=d, out_channels=s, kernel_size=5,stride=1,padding=2)
        # Rebuild the layer.
        self.reconstruction = nn.Conv2d(in_channels=s, out_channels=numInputChannels, kernel_size=5,stride=1,padding=2)
        # Initialize model weights.
        self._initialize_weights()
    def forward(self, x):
        x = self.upsample(x)
        x = F.relu(self.feat_extraction(x))
        x = F.relu(self.mapping(x))
        x = self.reconstruction(x)
        return x
    def _initialize_weights(self) -> None:
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.normal_(module.weight.data, 0.0, math.sqrt(2 / (module.out_channels * module.weight.data[0][0].numel())))
                nn.init.zeros_(module.bias.data)
        nn.init.normal_(self.reconstruction.weight.data, 0.0, 0.001)
        nn.init.zeros_(self.reconstruction.bias.data)
```

2.4.2.3 FSRCNN

En el trabajo Accelerating the Super-Resolution Convolutional Neural Network [28] se propone el modelo *Fast Super-Resolution Convolutional Neural Network* (FSRCNN), el cual es una mejora del modelo SRCNN planteado en la sección anterior que busca no solo mejorar el rendimiento en cuando a la capacidad de super resolución de SRCNN sino que mejora su coste computacional, permitiendo incluso una aplicación en tiempo real a un conjunto de imágenes por segundo. Consiste en la siguiente estructura:

1. Feature Extraction: De forma análoga al modelo SRCNN, se utiliza una capa convolucional de d filtros para extraer las características de la imagen de entrada.

2. Shrinking: Se reduce la dimensionalidad de las características extraídas utilizando una capa de convolución de kernel 1. Con ello se mapea el espacio de las características a de un espacio de d filtros a otro de s filtros, con s << d. Se busca con ello representar de forma compacta la información de entrada y conseguir una mayor eficiencia computacional en las siguientes capas.

3. Non-Linear Mapping: Se aplica una secuencia de m capas convolucionales de kernel 3 para extraer información espacial de la representación anterior, manteniendo s filtros.

4. Expanding: Se Vuelve a mapear el espacio de características o filtros utilizando una capa convolucional de d filtros, aumentando así la dimensionalidad de la representación.

5. Deconvolution: Finalmente, se utiliza una convolución traspuesta para aumentar el tamaño final de la matriz y converger los datos a 3 canales. El resultado es la imagen escalada.

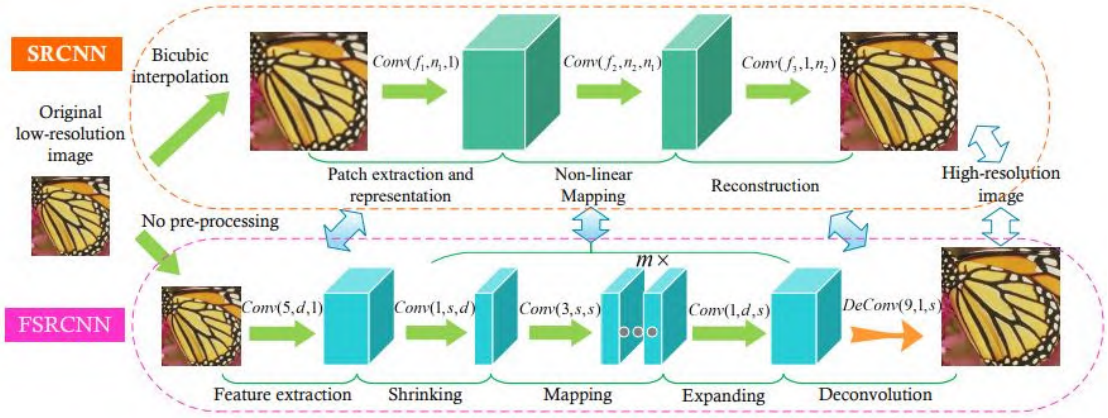


Figura 12:Arquitectura de modelo FSRCNN[33]

Se han ajustado los parámetros del modelo de acuerdo con el modelo propuesto en el *paper* original, con s = 56, d = 12 y m = 10.

Se utiliza también una inicialización de los parámetros del modelo como se hace en el trabajo original. El siguiente código muestra la implementación del modelo.

```
class FSRCNN(nn.Module):
    def __init__(self, numInputChannels, d, s, m, scale_factor):
        # call the parent constructor
        super(FSRCNN, self).__init__()
        #Feature extraction (as in paper) Conv(filter,d,channels)
        self.feat_extraction = nn.Sequential(nn.Conv2d(in_channels=numInputChannels, out_channels=d, kernel_size=5, padding=5//2),
                                             nn.PReLU(d))
        #Schrinking extraction (as in paper)
        self.schrinking = nn.Sequential(nn.Conv2d(in_channels=d, out_channels=s, kernel_size=1), nn.PReLU(s))
        #Mapping (as in paper)
        self.mapping = nn.ModuleList()
        for _ in range(m):
            self.mapping.append(nn.Conv2d(in_channels=s, out_channels=s, kernel_size=3, padding=3//2))
            self.mapping.append(nn.PReLU(s))
        self.mapping = nn.Sequential(*self.mapping)
        #Expanding (as in paper)
        self.expanding = nn.Sequential(nn.Conv2d(in_channels=s, out_channels=d, kernel_size=1),
                                       nn.PReLU(d))
        #Deconvoltion
        self.deconv = nn.ConvTranspose2d(in_channels=d, out_channels=numInputChannels, kernel_size=9,
                                         stride=scale_factor,padding=9//2, output_padding=scale_factor-1)
        # Initialize model weights.
        self._initialize_weights()
    def forward(self,x):
        x = self.feat_extraction(x)
        x = self.schrinking(x)
        x = self.mapping(x)
        x = self.expanding(x)
        x = self.deconv(x)
        return x
    def _initialize_weights(self) -> None:
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.normal_(module.weight.data, 0.0, math.sqrt(2 / (module.out_channels * module.weight.data[0][0].numel())))
                nn.init.zeros_(module.bias.data)
        nn.init.normal_(self.deconv.weight.data, 0.0, 0.001)
        nn.init.zeros_(self.deconv.bias.data)
```

2.4.2.4 VDSR

En el trabajo Accurate Image Super-Resolution Using Very Deep Convolutional Networks[32] se propone el uso de una red de alta profundidad en el proceso de *upsampling*. En el *paper* original de VDSR se propone la técnica de que la salida del modelo sean los residuos en lugar de los valores de forma directa, la cual tomamos prestada en este trabajo y la aplicamos a todos los modelos. Se demuestra que esta técnica ayuda a los problemas de *vanishing/exploding gradients* y favorece a la convergencia en un menor número de epochs para la tarea de upsampling de imágenes. La arquitectura del modelo consiste en:

1. Una operación de interpolación bicúbica a partir de la matriz de baja resolución de entrada.

2. Un mapeo inicial con una capa convolucional en dos dimensiones de d filtros, kernel de 3x3 y padding de 1 para mantener la misma dimensión de la matriz de entrada.

3. Una serie de k capas, de d filtros, kernel de 3x3 y padding de 1 consecutivas.

4. Un mapeo final con una capa de convolución, manteniendo la misma dimensión, a los 3 canales de salida de la matriz final.

Se utiliza la función de activación ReLu tras cada capa de convolución.

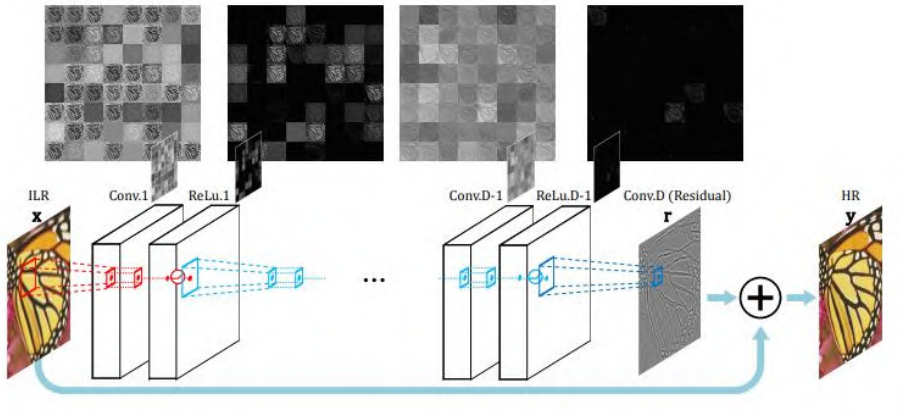


Figura 13: Arquitectura de modelo VDSR[33]

Debido a utilizar un número grande de capas profundas, con k = 20 en el trabajo original, el modelo desarrolla su capacidad de encontrar patrones no lineales gracias a su profundidad, mejorando su rendimiento de forma significativa en las tareas de *upsampling* de imágenes. En nuestro caso, dadas las limitaciones en los recursos computacionales, no limitamos a un modelo con 10 capas.

```
class VDSR(nn.Module):
    def __init__(self, numInputChannels, deep_channels, num_layers, scale_factor):
        # call the parent constructor
        super(VDSR, self).__init__()
        self.upsample = nn.Upsample(scale_factor=scale_factor,mode='bicubic')
        self.conv_first = nn.Conv2d(in_channels=numInputChannels, out_channels=deep_channels, kernel_size=3, padding=1)

        self.conv_middle = nn.ModuleList()
        for _ in range(num_layers-1):
            self.conv_middle.append(nn.Conv2d(in_channels=deep_channels, out_channels=deep_channels, kernel_size=3, padding=1))
        self.conv_middle = nn.Sequential(*self.conv_middle)

        self.conv_last = nn.Conv2d(in_channels=deep_channels, out_channels=numInputChannels, kernel_size=3, padding=1)

    def forward(self,x):
        x = self.upsample(x)
        x = F.relu(self.conv_first(x))
        for layer in self.conv_middle:
            x = F.relu(layer(x))
        x = self.conv_last(x)
        return x
```

2.4.2.5 ESPCN

Se ha implementado el modelo descrito en el trabajo *Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network*[33] utilizando pytroch. Este modelo consiste en la siguiente arquitectura:

1. Un mapeo inicial con dos capas convolucionales en dos dimensiones de kernels 5 y 3 respectivamente que proyectan el espacio de características a 64 y 32 canales respectivamente, utilizando una función de activación tanh(x) en ambos casos.

2. Una capa de convolución que incrementa el número de canales a $C*r^2$ siendo C el número de canales de salida y r el factor de upsampling. Posteriormente se realiza una reorganización de las entradas del tensor movíendolos a las dimensiones espaciales, consiguiendo así el *upsampling*.

La técnica utilizada en la reorganización de los pixeles se denomina sub-pixel convolution, y se propone en el *paper* original como método de conseguir un upsampling con capacidad de representar un mayor número de detalles sin incurrir en un mayor coste computacional. Consiste en la combinación de varios canales en la misma posición de una imagen para formar una imagen de menos canales y mayor resolución.

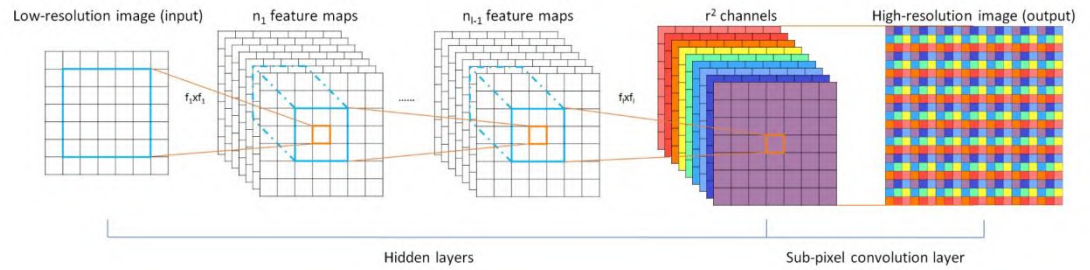


Figura 14: Arquitectura de modelo ESPCN[36]

Se utiliza también una inicialización de los parámetros de acuerdo con lo indicado en el *paper* original. El siguiente código muestra la implementación del modelo:

```
class ESPCN(nn.Module):
    #https://arxiv.org/pdf/1609.05158.pdf
    def __init__(self,in_channels: int,out_channels: int,
                 channels: int,upscale_factor: int):
        super(ESPCN, self).__init__()
        self.hidden_channels = channels // 2
        out_channels = int(out_channels * (upscale_factor ** 2))

        # Feature mapping
        self.feature_maps = nn.Sequential(
            nn.Conv2d(in_channels, channels, 5, 1, 2),
            nn.Tanh(),
            nn.Conv2d(channels, self.hidden_channels, 3, 1, 1),
            nn.Tanh(),
        )
        # Sub-pixel convolution layer
        self.sub_pixel_conv = nn.Sequential(
            nn.Conv2d(self.hidden_channels, out_channels, 3, 1, 1),
            nn.PixelShuffle(upscale_factor))
        self._initialize_weights()
    def forward(self, x):
        x = self.feature_maps(x)
        x = self.sub_pixel_conv(x)
        return x
    def _initialize_weights(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                if module.in_channels == self.hidden_channels:
                    nn.init.normal_(module.weight.data,0.0,0.001)
                    nn.init.zeros_(module.bias.data)
                else:
                    nn.init.normal_(module.weight.data,0.0,math.sqrt(2 / (module.out_channels * module.weight.data[0][0].numel())))
                    nn.init.zeros_(module.bias.data)
```

2.4.2.6 Encoder-Decoder

En contraste los modelos ya presentados, los cuales mantienen la anchura de los modelos al profundizar en las capas o la reducen para encapsular la información en un menor tamaño, se propone un modelo que aumente la anchura utilizando un aumento en el número de filtros utilizados en las capas convolucionales, actuando a modo de codificador que pueda así representar la relación entre los canales, que al estar relacionadas de forma física es mucho mas compleja que la existente en los canales de una imagen genérica. Posteriormente se vuelve a transformar en una imagen a alta resolución en una arquitectura inspirada en la metodología de codificador-decodificador [34] [35](Encoder-Decoder). El siguiente código muestra la implementación del modelo:

```python
class ED_TransConv(nn.Module):
    def __init__(self,channels):
        super(ED_TransConv, self).__init__()
        # Input image size is (N, 3, H, W)
        #Encoder
        self.conv1 = nn.Conv2d(in_channels=channels, out_channels=128, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)

        # Upsample by a factor of 2
        #Decoder
        self.upsample1 = nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=2, stride=2)
        # Another convolutional layer
        self.conv3 = nn.Conv2d(in_channels=128, out_channels=64, kernel_size=3, padding=1)
        # Final upsample by a factor of 2 to achieve total 4x upsampling
        self.upsample2 = nn.ConvTranspose2d(in_channels=64, out_channels=channels, kernel_size=2, stride=2)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.upsample1(x)
        x = self.conv3(x)
        x = self.upsample2(x)
        return x
```

Se utiliza para el codificador dos capas convolucionales que aumenten el número de canales a 128 y 256, respectivamente. Se mantiene la dimensión de la matriz utilizando un tamaño de kernel de 3 y un padding de 1 para facilitar el proceso de upsampling posterior. Tras cada capa se aplica una función de activación ReLu. A partir de esta representación de alta densidad, la sección del modelo que actúa como decodificador la transforma en una imagen de mayor resolución utilizando una capa de deconvolución de tamaño de kernel 2 y stride 2 para conseguir el duplicar la resolución mientras se reduce el número de canales a 128. Se vuelve a aplicar una capa de convolución para reducir el número de canales de forma progresiva y finalmente se aplica una capa final de deconvolución que duplica la resolución y se reduce a 3 canales de salida, consiguiendo un upsampling efectivo de 4 veces el de la entrada del modelo.

Se implementa también otro modelo equivalente al anterior que utiliza la técnica del *pixel shuffle* ya implementada en ESPCN para realizar el proceso de aumento de resolución.

```python
class ED_PixelShuffle(nn.Module):
    def __init__(self,channels,upscale_factor):
        super(ED_PixelShuffle, self).__init__()
        # Input image size is (N, 3, H, W)
        ps_channels = int(channels * (upscale_factor ** 2))
        #Enconder
        self.conv1 = nn.Conv2d(in_channels=channels, out_channels=128, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)

        #Decoder and Sub-pixel convolution layer
        self.conv4 = nn.Conv2d(in_channels=256, out_channels=128, kernel_size=3, padding=1)
        self.sub_pixel_conv = nn.Sequential(
            nn.Conv2d(128, ps_channels, 3, 1, 1),
            nn.PixelShuffle(upscale_factor))
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.conv4(x)
        x = self.sub_pixel_conv(x)
        return x
```

2.4.3 Funciones de coste

Las funciones de coste utilizadas en el proceso de optimización de los modelos de aprendizaje profundo incluyen el error medio cuadrado y una función específica que calcula los residuos de la ecuación de Navier-Stokes, la cual rige el comportamiento de las simulaciones realizadas. Estas se encuentran

implementadas en el archivo **losses.py** del repositorio del trabajo [21]. El error medio cuadrado se define como:

$$MSE(Y, Y') = \frac{1}{N}\sum_{i=1}^{N}(Y_i - Y'_i)^2$$

Donde Y e Y' son el valor de salida del modelo y el ground-truth y N es el número de elementos. Pytorch ya dispone de una implementación del error medio cuadrado, con lo que hacemos uso de ella para definir la función de coste utilizada:

```
from torch.nn.functional import mse_loss
import torch
class MSE_loss(nn.Module):
    def __init__(self):
        super(PolymorphicMSE_loss, self).__init__()
    def forward(self, x, y, _):
        return mse_loss(x,y)
```

Esta función de coste es ampliamente utilizada en la optimización supervisada de modelos. Alternativamente podrían haberse utilizado otras funciones de coste como la pérdida de Hubber o el error medio absoluto, las cuales son menos sensible a valores extremos del error. Sin embargo, se ha considerado que se prefiere precisamente que el valor de coste o pérdida sea sensible ante estos valores extremos, con el fin de conseguir minimizarlos y encontrar un resultado robusto en todo el dominio que evite la presencia de *outliers*. Además, el utilizar el cuadrado para variables físicas como la velocidad permite que el error medio cuadrado represente un valor que se correlacione con la diferencia de energía cinética del sistema, la cual es proporcional al cuadrado de la velocidad.

Para la función de coste que calcula los residuos planteamos las ecuaciones de Navier Stokes para un fluido incompresible, separando las componentes de la velocidad. Con ello buscamos minimizar los residuos resultantes de separar las variables en la ecuación de continuidad y momento.

$$Eq\ de\ continuidad: R_0 = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

$$Eq\ de\ momento: \begin{cases} R_1 = \rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) + \frac{\partial p}{\partial x} - \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ R_2 = \rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) + \frac{\partial p}{\partial y} - \mu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \end{cases}$$

Siendo u y v las componentes horizontal y vertical de la velocidad, respectivamente, y p la presión. Para hallar una aproximación a las derivadas parciales, la función de coste hace uso de la diferencia finita. Se utiliza la diferencia central, definida como:

$$\frac{\partial f}{\partial x} \approx \frac{f(t + \Delta x) - f(t - \Delta x)}{2\Delta x}$$

Finalmente, se utiliza el diferencial finito de segundo orden para hallar una aproximación a la segunda derivada parcial con respecto a una variable:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(t+\Delta x) - 2f(t) + f(t-\Delta x)}{\Delta x^2}$$

Como se conocen las constantes y el tamaño de los diferenciales al haber sido fijados en las simulaciones, dado un *batch* de *timesteps* contiguos de datos de una simulación podemos aproximar un valor de la diferencia del residuo en cada punto el cual puede ser minimizado. En base a esto se implementa la función de coste:

```
class Navier_Stokes_informed_loss(nn.Module):
    def __init__(self, rho=1, mu=0.00001, dt=1/32,dx=1/255,dy=1/255):
        super(Navier_Stokes_informed_loss, self).__init__()
        self.rho = rho
        self.mu = mu
        self.dt = dt
        self.dx = dx
        self.dy = dy
    def calculate_res(self,x,std):
        #De-normalize to recover true physical values
        x *= std
        u = x[:,0,:,:]
        v = x[:,1,:,:]
        p = x[:,2,:,:]

        #Central first order difference
        dudt = (torch.roll(u,-1,dims=0)-torch.roll(u,1,dims=0))/self.dt
        dvdt = (torch.roll(v,-1,dims=0)-torch.roll(v,1,dims=0))/self.dt
        dudx = (torch.roll(u,-1,dims=1)-torch.roll(u,1,dims=1))/(2*self.dx)
        dudy = (torch.roll(u,-1,dims=2)-torch.roll(u,1,dims=2))/(2*self.dy)
        dvdx = (torch.roll(v,-1,dims=1)-torch.roll(v,1,dims=1))/(2*self.dx)
        dvdy = (torch.roll(v,-1,dims=2)-torch.roll(v,1,dims=2))/(2*self.dy)
        dpdx = (torch.roll(p,-1,dims=1)-torch.roll(p,1,dims=1))/(2*self.dx)
        dpdy = (torch.roll(p,-1,dims=2)-torch.roll(p,1,dims=2))/(2*self.dy)

        #Central second order difference
        d2udx2 = (torch.roll(u,-1,dims=1)-2*u+torch.roll(u,1,dims=1))/(self.dx**2)
        d2udy2 = (torch.roll(u,-1,dims=2)-2*u+torch.roll(u,1,dims=2))/(self.dy**2)
        d2vdx2 = (torch.roll(v,-1,dims=1)-2*v+torch.roll(v,1,dims=1))/(self.dx**2)
        d2vdy2 = (torch.roll(v,-1,dims=2)-2*v+torch.roll(v,1,dims=2))/(self.dy**2)

        #Borders are not computed, it's finite differences makes no sense with torch.roll method
        u = u[1:-1,1:-1,1:-1]
        v = v[1:-1,1:-1,1:-1]
        p = p[1:-1,1:-1,1:-1]
        dudt = dudt[1:-1,1:-1,1:-1]
        dudx = dudx[1:-1,1:-1,1:-1]
        dudy = dudy[1:-1,1:-1,1:-1]
        dvdt = dvdt[1:-1,1:-1,1:-1]
        dvdx = dvdx[1:-1,1:-1,1:-1]
        dvdy = dvdy[1:-1,1:-1,1:-1]
        dpdx = dpdx[1:-1,1:-1,1:-1]
        dpdy = dpdy[1:-1,1:-1,1:-1]
        d2udx2 = d2udx2[1:-1,1:-1,1:-1]
        d2udy2 = d2udy2[1:-1,1:-1,1:-1]
        d2vdx2 = d2vdx2[1:-1,1:-1,1:-1]
        d2vdy2 = d2vdy2[1:-1,1:-1,1:-1]

        #Calculate residuals
        #Continuity eq
        r0 = torch.abs(dudx+dvdy)
        #Momentum
        r1 = torch.abs(self.rho*(dudt+u*dudx+v*dudy) + dpdx - self.mu*(d2udx2+d2udy2))
        r2 = torch.abs(self.rho*(dvdt+u*dvdx+v*dvdy) + dpdy - self.mu*(d2vdx2+d2vdy2))

        return r0, r1, r2
    def forward(self, x, y, std):
        r0_y, r1_y, r2_y = self.calculate_res(y,std)
```

```
        r0_x, r1_x, r2_x = self.calculate_res(x,std)

        dif_res_0 = torch.mean((r0_y-r0_x)**2)
        dif_res_1 = torch.mean((r1_y-r1_x)**2)
        dif_res_2 = torch.mean((r2_y-r2_x)**2)

        return 0.001*(dif_res_0+dif_res_1+dif_res_2)
```

El resultado se multiplica por una constante determinada de forma empírica utilizada para escalar el coste y estandarizarlo con el orden de magnitud del error medio cuadrado, para poder utilizar ambas funciones de coste de forma simultanea y que den como resultado gradientes similares.

2.4.4 Entrenamiento de los modelos

En esta sección se describe paso a paso el proceso de entrenamiento de los modelos de aprendizaje profundo implementados utilizando las funcionalidades de la librería pytorch de computación paralela utilizando módulos CUDA de GPU. El código presentado se encuentra disponible en el archivo **train.py** del repositorio del trabajo[21].

Se importan las librerías necesarias utilizadas en el trabajo: pytorch, numpy y tqdm. Además, se importan las funciones y clases implementadas en los módulos **losses.py** y **model.py**, que implementan las funciones de coste y modelos utilizados respectivamente:

```
#Exteral libraries imports
import numpy as np
import torch
import torch.nn as nn
import os
from tqdm import tqdm
import matplotlib.pyplot as plt
#Custom imports
from model import *
from losses import *
from enum import Enum
```

Se definen las constantes e hiperparámetros utilizados en el entrenamiento del modelo. Aquí se definen el número de veces que se va a entrenar el modelo con todo el conjunto de datos de entrenamiento (N_EPOCHS), el factor de aumento de resolución (UPSAMPLE_FACTOR), el tamaño de los *minibatchs* que define la cantidad de *timesteps* que se utilizarán para calcular los gradientes antes de cada optimización de los parámetros del modelo (BATCH_SIZE), el número de canales de los datos de entrada (INPUT_CHANNELS). Finalmente se utiliza una variable para controlar el modo de entrenamiento:

- DIRECT_MSE: Minimiza el error medio cuando la salida del modelo es directamente la matriz de alta resolución.
- RESIDUAL_MSE: Minimiza el error medio cuando la salida del modelo es la diferencia entre la matriz de alta resolución y una interpolación bicúbica.
- RESIDUAL_PHYSICS: Minimiza la diferencia entre los residuos físicos y el error medio de forma simultánea cuando la salida del modelo es la diferencia entre la matriz de alta resolución y una interpolación bicúbica.

```
Training_Modes = Enum('Mode', ['DIRECT_MSE', 'RESIDUAL_MSE', 'RESIDUAL_MSE_PHYSICS'])
#Training parameters
N_EPOCHS = 100               #Number of training epochs
UPSAMPLE_FACTOR = 4          #Upsample factor
BATCH_SIZE = 16              #Training Mini-Batch seize
INPUT_CHANNELS = 3           #Number of channels, representing physical variables
MODE = Training_Modes.DIRECT_MSE
```

Se realiza la carga de los datos a partir de archivos guardados en el disco local. Los datos se encuentran guardados en un tensor de forma (N, T, C, W, H), donde:

- N identifica a una simulación.
- T identifica el *timestep* o tiempo de la simulación.
- C es el canal (dos componentes de la velocidad y presión)
- W es la coordenada horizontal de la malla.
- H es la coordenada vertical de la malla.

```
data_train_y = np.load('.\\data\\trainData.npy')
data_eval_y = np.load('.\\data\\evalData.npy')
print(data_train_y.shape)
print(data_eval_y.shape)
```

Se crea la instancia del modelo a entrenar y se crea también el optimizador y las funciones de coste utilizadas.

```
#Model instance declaration
model = ED_TransConv(INPUT_CHANNELS).to(device)

#Optimizer parameters
weight_decay = 1e-7
learning_rate = 0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

#Loss functions
error_loss = PolymorphicMSE_loss()
physics_loss = Navier_Stokes_informed_loss(rho=1, mu=0.00001, dt=1/32,dx=1/255,dy=1/255)
```

Se define una función utilizada para el entrenamiento de cada *epoch*:

```
def epoch_trainer(model,data,loss_function,loss_function_aux=None,upscale_factor=4,
                  predict_res=True,batch_size=16):
    #Creation of instances of upsampler-downsampler
    bicubic = torch.nn.Upsample(scale_factor=upscale_factor, mode='bicubic')
    downsampler = torch.nn.AvgPool2d(upscale_factor)
    #Variable to store the average loss of all the minibatches for this epoch
    avg_loss = 0
    n = 0
    #Set the model for training
    model.train()
    optimize_aux_loss = loss_function_aux != None
    #Loop through data and train
    for sim_index in range(data.shape[0]):
        simulationData = data[sim_index]
        #Normalize
        std = np.std(data[sim_index],axis=(0,2,3),keepdims=True)
        simulationData = simulationData/std
        std = torch.tensor(std).float().to(device)

        for batch_index in range(0,data.shape[1],batch_size):
            y = torch.tensor(simulationData[batch_index:batch_index+batch_size,:,:,:]).float()
            x = downsampler(y)
            #Move to device, for efficient use of VRAM
            x = x.to(device)
            y = y.to(device)

            #Randomly rotate matrix on spatial dimentions
```

```
            k = np.random.randint(low=0,high=4)
            x = torch.rot90(x,k,dims=(-1,-2))
            y = torch.rot90(y,k,dims=(-1,-2))

            output = model(x) # forwards pass
            if predict_res: #Reconstruct when modelling for PREDICT_RESIDUAL
                output = bicubic(x)+output
            loss_error = loss_function(output,y,std)
            optimizer.zero_grad() # set gradients to zero
            loss_error.backward(retain_graph=optimize_aux_loss) # backwards pass
            if optimize_aux_loss:
                #Extra loss optimization optimizacion
                loss_aux = loss_function_aux(output,y,std)
                loss_aux.backward() # backwards pass
            optimizer.step() # update model parameters
            avg_loss += loss_error.item()
            n += 1
            del x, y, output
    if n != 0: avg_loss /= n
    return avg_loss
```

Esta función realiza las siguientes acciones:

1. Se crean instancias de un interpolador bicúbico y un downsampler para simular el proceso de recuperar datos de alta resolución a partir de datos de baja resolución.
2. Se configura el modelo para entrenamiento haciendo la llamada al método train() del modelo, el cual es heredado de la librería pytorch y permite que los parámetros del modelo puedan ser alterados.
3. Se itera por cada una de las simulaciones, estandarizando los valores de cada uno de los canales a lo largo de toda la simulación.
4. Se itera por cada minibatch de la simulación. En cada *mini-batch*:
    a. Se hace un downsample de los datos de la simulación para simular los datos de entrada de baja resolución y se vuelcan los datos sobre la VRAM de la GPU.
    b. Se rotan la matriz de los datos de la simulación de manera aleatoria para aumentar la variabilidad de los datos y reducir el sobreajuste.
    c. Se realiza la propagación hacia delante de los datos de entrada (forward pass)
    d. En el caso de que se quieran optimizar los residuos de la interpolación en lugar de inferir directamente el valor de salida, se recupera el resultado final a partir de la suma de los residuos con una interpolación bicúbica con la línea.
    e. Se aplica la función de coste que calcula el error medio al resultado. A continuación, se ajustan los gradientes iniciales a 0 para posteriormente realizar la retropropagación o *backpropagation* y acumular los gradientes. En el caso de optimizar también los residuos físicos, se realiza la misma acción aplicando la función de coste de los residuos y acumulando también los gradientes correspondientes.
    f. Finalmente el optimizador altera los parámetros del modelo en base al gradiente resultante obtenido utilizando el método *step()*.
    g. Se acumula el valor de la función de coste del error calculado para su monitorización.

h. Finalmente se limpian de la memoria de la GPU los tensores.

5. Finalmente, la función devuelve la media de los valores de coste para todo el *epoch*.

Se define también una función análoga de evaluación de datos, la cual realiza un recorrido por un conjunto de datos dado, calculando el valor de la función de coste, pero sin realizar ninguna optimización de los parámetros del modelo. Para ello se hace uso del método test(), heredado por el modelo, y la función torch.no_grad() para no permitir alterar los parámetros del modelo y también poder prescindir de guardar una representación de los gradientes, respectivamente. Con ello esta función comprueba el rendimiento del modelo en el conjunto de datos que tenga de entrada sin alterarlo.

```
def epoch_tester(model,data,loss_function,predict_res=True,upscale_factor=4,batch_size=16):
    #Creation of instances of upsampler-downsampler
    bicubic = torch.nn.Upsample(scale_factor= upscale_factor, mode='bicubic')
    downsampler = torch.nn.AvgPool2d(upscale_factor)
    #Variable to store the average loss of all the minibatches for this epoch
    avg_loss = 0
    n = 0
    #Set the model for evaluation
    model.eval()
    #Loop through data and train
    for sim_index in range(data.shape[0]):
        simulationData = data[sim_index]
        #Normalize
        std = np.std(data[sim_index],axis=(0,2,3),keepdims=True)
        simulationData = simulationData/std
        std = torch.tensor(std).float().to(device)
        for batch_index in range(0,data.shape[1],batch_size):
            y = torch.tensor(simulationData[batch_index:batch_index+batch_size,:,:,:]).float()
            x = downsampler(y)
            #Move to device, for efficient use of VRAM
            x = x.to(device)
            y = y.to(device)
            with torch.no_grad():
                output = model(x) # forwards pass
                if predict_res: #Reconstruct when modelling for PREDICT_RESIDUAL
                    output = bicubic(x)+output
                loss_mse = loss_function(output,y,std)
            avg_loss += loss_mse.item()
            n += 1
            del x, y, output
    if n != 0: avg_loss /= n
    return avg_loss
```

Finalmente, el script incorpora un *loop* de entrenamiento que gestiona la iteración por cada *epoch* y los datos resultantes de cada llamada a las funciones de entrenamiento definidas anteriormente. En base al tipo de modelo se configura de forma dinámica el etiquetado y guardado del modelo y los parámetros de configuración de la función *epoch_trainer*.

```
#Train loop
modelName = model.__class__.__name__
main_loss = loss_func_error
aux_loss = None
residual_pred = False
if MODE == Training_Modes.RESIDUAL_MSE:
    modelName += '_Res'
    main_loss = loss_func_error
    aux_loss = None
    residual_pred = True
if MODE == Training_Modes.RESIDUAL_MSE_PHYSICS:
    modelName += '_Res_Physics'
    main_loss = loss_func_error
```

```
    aux_loss = loss_func_physics
    residual_pred = True
print(modelName)
modelPath = "./savedModels/"+modelName+"/"
train_losses = []
eval_losses = []
try:
    os.mkdir(modelPath)
except: pass

for epoch in tqdm(range(1, N_EPOCHS + 1), desc="Training...", ascii=False, ncols=50):
    #Training batch loop
    bestEvalLoss = float('inf')
    train_losses.append(epoch_trainer(model,data_train_y,main_loss,loss_function_aux=aux_loss,
                                      upscale_factor=UPSAMPLE_FACTOR,predict_res=residual_pred,
                                      batch_size=BATCH_SIZE))
    eval_losses.append(epoch_tester(model,data_eval_y,main_loss,predict_res=residual_pred,
                                      upscale_factor=UPSAMPLE_FACTOR,batch_size=BATCH_SIZE))
    print(f' Epoch {epoch}, TrainL: {train_losses[-1]:.2e}, EvalL: {eval_losses[-1]:.2e}')
    # Plotting the losses
    if (epoch >= 10 and epoch % 10 == 0):
        np.save(modelPath+"train_loss",np.array(train_losses))
        np.save(modelPath+"eval_loss",np.array(eval_losses))
    #Save best eval model after third epoch
    if epoch > 3 and eval_losses[-1] < bestEvalLoss:
        bestEvalLoss = eval_losses[-1]
        torch.save(model, modelPath+modelName+".pth")
```

Se guardan los datos resultantes de evaluar la función de coste cada 10 *epochs* para su posterior análisis. El algoritmo realiza el entrenamiento de cada *epoch* y selecciona y guarda el modelo identificado por su clase y metodología de entrenamiento cada vez que el rendimiento sobre los datos de evaluación mejore. Con ello se busca seleccionar el modelo con mayor capacidad de generalización sobre datos sobre los cuales no se ha entrenado, que se evalúan en un análisis final posterior sobre datos de prueba (*test*) no computados en ningún paso del proceso de entrenamiento.

2.4.5 Resultados obtenidos

2.4.5.1 Estudio comparativo del proceso de entrenamiento de los modelos

Se ha realizado un entrenamiento de todos los modelos y todas las estrategias de optimización implementados con las mismas condiciones para poder así ofrecer una visión comparativa sobre el proceso y tiempos de convergencia. Esto se implementa en los archivos **compare_models.py** y **compare_strategies.py** del repositorio del trabajo[21]. Así, se ha dispuesto un *learning rate* relativamente bajo de 0.0001 con el optimizador *Adam*[36] durante 100 epochs. Como se describe también anteriormente, se utiliza un tamaño de mini-batch de 16 timesteps de simulación.

El análisis comparativo de esta sección tiene como objetivo, por un lado, comparar el proceso de convergencia entre modelos ante una misma estrategia de estrategia de optimización, y por otro, comparar las estrategias de optimización para cada modelo. La siguiente figura representa, discriminando por cada tipo de optimización, el error medio cuadrado de la salida del modelo con respecto al ground-truth en el set de datos de evaluación en el proceso de entrenamiento preliminar durante 100 epochs. La columna de la izquierda representa el error medio cuadrado, mientras que en la columna de la derecha se muestra también el logaritmo del mismo error medio cuadrado para facilitar su visualización dada la variabilidad de los valores.

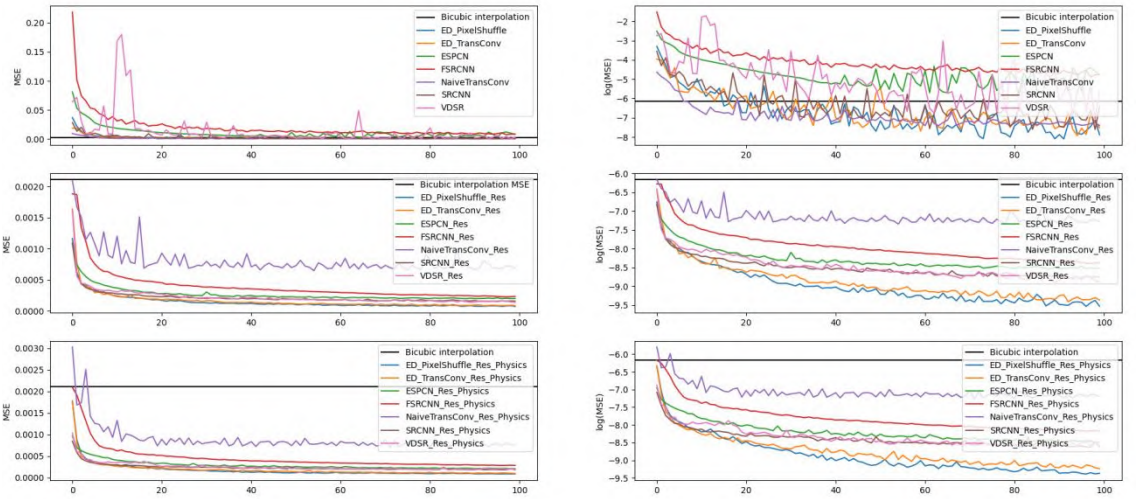


Figura 15: Comparativa de los entrenamientos de cada modelo agrupados por estrategia de optimización.

Se observa como la inferencia directa de los valores de la representación de los datos a alta resolución, que se observa en el gráfico superior, tiene un comportamiento significativamente más caótico que no consigue tan buenos resultados como predecir la diferencia entre la interpolación bicúbica y el *ground-truth* como se hace en los dos gráficos inferiores que muestran el entrenamiento con esta metodología sin y con el uso de la función de coste de los residuos de la Ecuación de Navier Stokes, respectivamente. El uso de esta función de coste se aplica únicamente a la estrategia de predecir la diferencia

entre la interpolación bicúbica y el *ground-truth* dado que el resultado es significativamente mejor en los entrenamientos preliminares.

A su vez, la siguiente figura representa de nuevo los mismos datos agrupados por cada arquitectura de modelo de aprendizaje profundo utilizada.

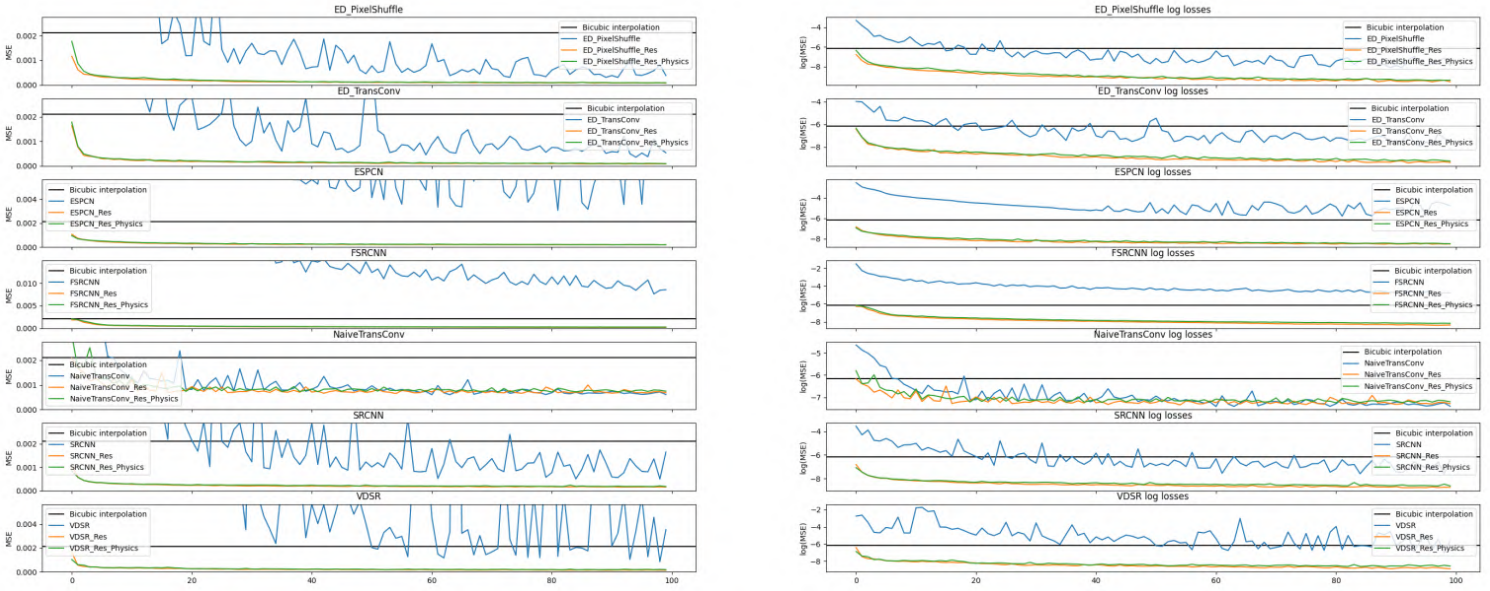
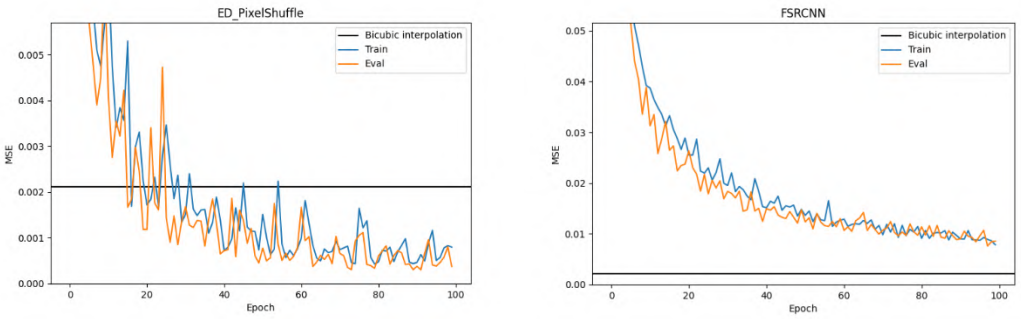


Figura 16: Comparativa de evolución de los entrenamientos agrupados por arquitecturas.

Podemos observar como por un lado todos los modelos convergen más rápido cuando se utiliza la técnica de optimizar el modelo para que su salida sean los residuos de una interpolación bicúbica en lugar de inferir directamente los valores de la representación de alta resolución. Los modelos que están diseñados precisamente con esta metodología en la forma propuesta pos sus autores tienen un rendimiento significativamente peor. A su vez, no encontramos una diferencia significativa en la evolución del entrenamiento de utilizando o no la función de coste que optimiza las diferencias entre los residuos de la ecuación de Navier Stokes.

Si observamos además la evolución del valor de la función de coste sobre los datos de los datos de entrenamiento y evaluación, observamos que en ningún momento encontramos direcciones divergentes lo cual nos indica que los modelos aplicados al problema no tienen tendencia al sobreajuste, y una mejora en el rendimiento sobre los datos de entrenamiento implica también una mejora en los datos de evaluación. Las siguientes figuras ejemplifican este hecho para algunos entrenamientos de ejemplo:



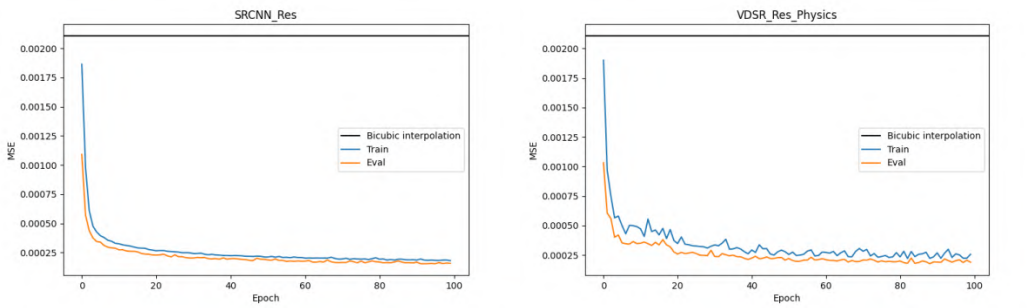


Figura 17: Muestras de evolución del entrenamiento de algunos modelos. Se observa que los modelos no tienen tendencia al sobreajuste de los datos de entrenamiento.

2.4.5.2 Estudio comparativo del rendimiento de los modelos

Cada modelo, ya pre-entrenado de forma preliminar, se sigue entrenando hasta conseguir una convergencia clara y se adapta el *learning rate* en función de la evolución observada en el entrenamiento preliminar. En concreto, se entrenan los modelos que implementan la arquitectura FSRCN, las dos arquitecturas Encoder-Decoder propuestas, ya que como vemos en la representación de la evolución de la función de coste en la Figura 15, son los únicos modelos que tienen pendiente negativa no nula tras los 100 epochs de pre-entrenamiento, con lo que se entrenan en las mismas condiciones durante 100 *epochs* más, tras los cuales se ha producido la convergencia y no existe mejoría del resultado del set de datos de evaluación.

Se evalúa el rendimiento de los modelos en el set de datos de *test*, el cual no se ha utilizado en el proceso de entrenamiento. Esto se implementa en el archivo **test.py** del repositorio del trabajo[21]. La siguiente tabla representa el valor del error medio cuadrado de cada uno de los modelos y estrategia de optimización, así como el error medio máximo y la diferencia de los residuos de la ecuación física.

Modelo	MSE	MaxMSE	PhysicsLoss
Bicubic	2.42E-03	2.99E-01	3.15E-03
ED_PixelShuffle	3.97E-04	7.26E-03	2.68e-04
ED_PixelShuffle _Res	**7.79-05**	4.93E-03	1.24e-04
ED_PixelShuffle _Res_Physics	8.09E-04	**4.67E-03**	**1.19E-04**
ED_TransConv	5.39E-04	7.50E-03	3.08E-04
ED_ TransConv _Res	8.48 E-04	4.94E-03	1.30E-04
ED_ TransConv _Res_Physics	1.10E-04	5.56E-03	1.57E-04
ESPCN	7.05E-03	1.45E-02	1.31E-03
ESPCN_Res	2.45E-04	7.68E-03	3.52E-04
ESPCN_Res_Physics	2.58E-04	8.00E-03	3.59E-04
FSRCNN	6.31E-03	1.97E-02	1.79E-03
FSRCNN_Res	2.12E-04	6.76E-03	2.80E-04
FSRCNN_Res_Physics	2.82E-04	7.52E-03	3.10E-04
NaiveTransConv	6.80E-04	1.01E-02	6.63E-04
NaiveTransConv_Res	7.54E-04	1.13E-02	9.44E-04
NaiveTransConv_Res_Physics	8.32E-04	1.12E-02	7.37E-04
SRCNN	1.63E-03	8.80E-03	4.36E-04
SRCNN_Res	1.89E-04	6.87E-03	2.79E-04
SRCNN_Res_Physics	2.20E-04	7.40E-03	2.90E-04
VDSR	3.59E-03	1.12E-02	5.32E-04
VDSR_Res	1.66E-04	6.57E-03	2.30E-04
VDSR_Res_Physics	2.31E-04	7.13E-03	2.81E-04

Tabla 1: Comparación de rendimiento de los modelos tras entrenamiento sobre set de datos *test*.

Podemos ver que, claramente, el uso de la mayoría de los modelos utilizados de redes profundas convolucionales presenta una mejoría sustancial con respecto a una simple interpolación, especialmente utilizando la optimización de la diferencia entre una interpolación bicúbica y la representación de alta resolución. Entre ellos, los modelos propuestos (ED_TrasnConv y ED_PixelShuffle) tienen el error medio cuadrado, máximo y diferencias en los residuos de las ecuaciones físicas más bajo.

Por otra parte, no parece existir una diferencia significativa en utilizar la función de coste que minimiza la diferencia a través de los residuos de las ecuaciones físicas, y las diferencias marginales no nos permiten concluir que el modelo es capaz de utilizar esta información para mejorar la solución, que, aunque sea el caso en el caso del mejor modelo (ED_PixelShuffle) para el error máximo y la función de coste física, no es consistente a nivel general en la aplicación a los diferentes modelos y la diferencia no es concluyente.

2.4.5.1 Visualización del funcionamiento de los modelos sobre el set de test.

En esta sección se presenta una visualización, agrupada por cada metodología de entrenamiento, de la aplicación de los modelos a algunas muestras de las simulaciones del conjunto de datos test. Esto permite visualizar el funcionamiento de los modelos y sus diferencias. El script que realiza esta tarea se implementa en el archivo **test_visual.py**[21]. Se muestran los datos correspondientes a la presión.

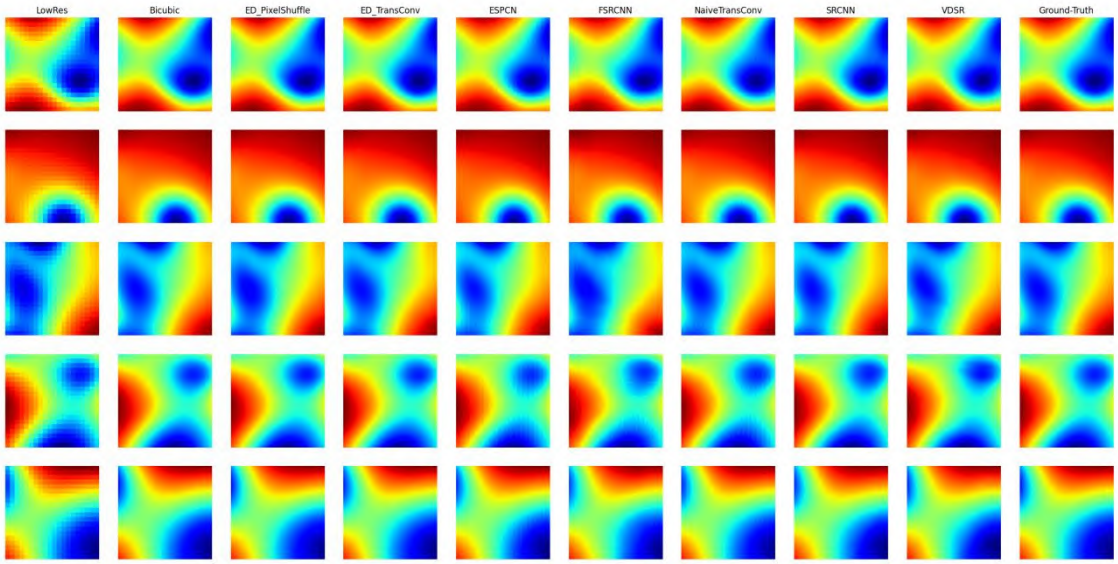


Figura 18: Comparativa visual de reconstrucción de representación de alta resolución para los modelos que utilizan inferencia directa de valores de alta resolución.

Puede observarse una clara pixelación y artefactos de malla en la mayoría de los modelos, destacando de forma positiva SRCNN, lo cual es en parte esperado ya que es un modelo diseñado originalmente para inferir

directamente el resultado final en lugar de los residuos de una interpolación bicúbica.

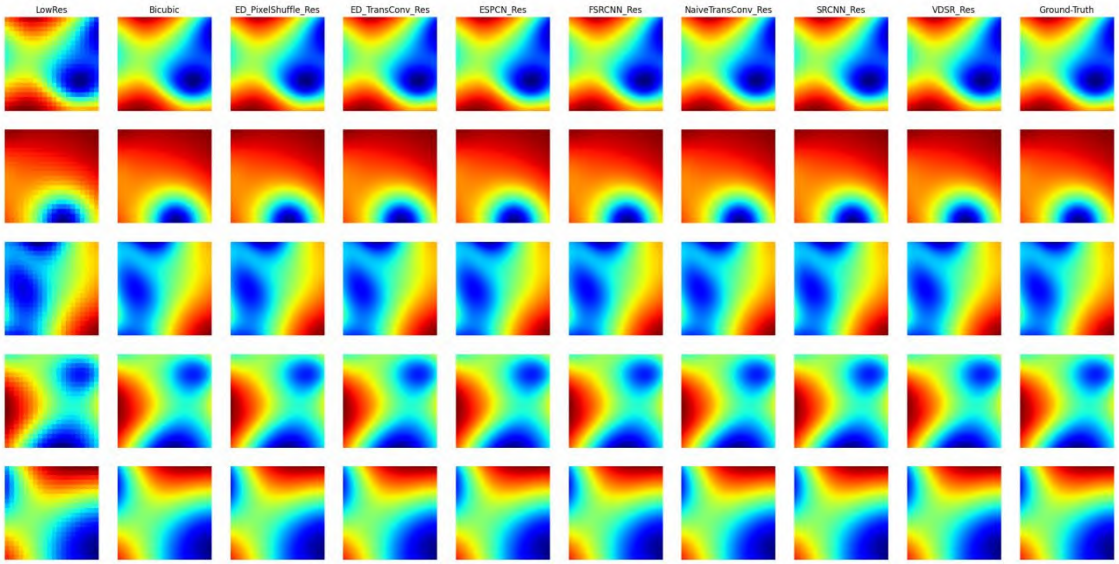


Figura 19: Comparativa visual de reconstrucción de representación de alta resolución para los modelos que se optimizan para predecir la diferencia (residuos) entre interpolación bicúbica y valores de alta resolución.

Puede observarse como los modelos propuestos, en especial la implementación que utiliza el *Pixel Shuffle* como metodología de aumento de resolución, reducen considerablemente algunos artefactos de malla que aparecen en la interpolación bicúbica y el resultado de otros modelos. El resultado de los modelos que se han entrenado utilizado también la función de coste del residuo físico no se representan al ser indistinguibles de la figura anterior a simple vista.

2.4.5.3 Estudio comparativo de la complejidad computacional de los modelos

Dado que el objetivo del trabajo es ofrecer una evaluación del rendimiento de modelos que buscan sustituir a procesos de simulaciones de, en principio, mayor complejidad computacional, se ofrece un análisis comparativo de la complejidad computacional de los modelos. Esta comparativa se realiza utilizando el script disponible en **compare_complexity.py**[21]. Dos métricas comúnmente utilizadas para este propósito to Multiply-Accumulate Operations (MACs), que cuantifica el número de operaciones de multiplicación y suma, y Float Point Operations (FLOAT), que cuantifica el número de operaciones totales. Aunque puede depender del modelo, normalmente encontramos que MAC=2FLOAT.

Utilizamos la librería pytorch-OpCounter[37] para contabilizar el número de operaciones FLOATs.

	NaiveTransConv	ED_PixelShuffle	ED_TransConv	ESPCN	FSRCNN	SRCNN	VDSR
FLOPs(G)	287.65	168.69	293.88	9.64	61.51	291.02	1398.33

Tabla 2: Número de operaciones (en millones) de cada arquitectura

En la siguiente figura se muestra una representación gráfica del error cuadrado medio del modelo (menos es mejor) con respecto al número de operaciones para el caso donde se optimizan los modelos para inferir la diferencia entre una interpolación bicúbica y el valor ground-truth de alta resolución, la cual como vimos es la metodología de optimización con un significativo mejor rendimiento general de los modelos.

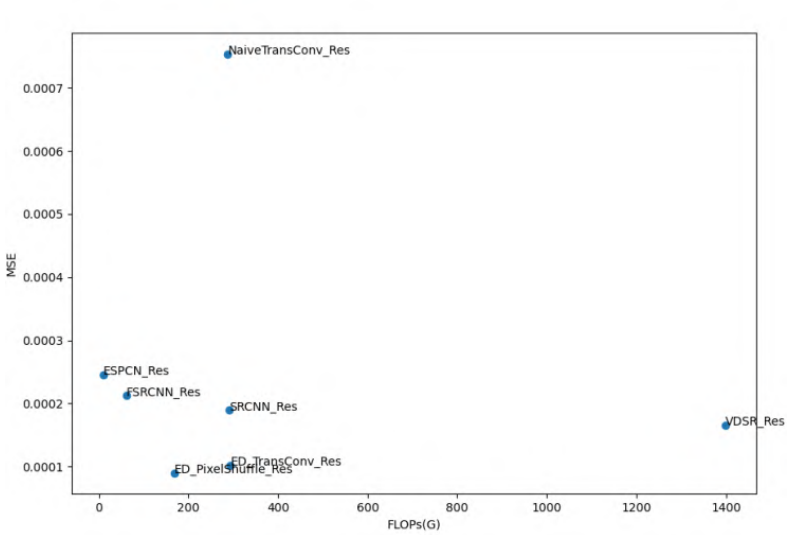


Figura 20: Error cuadrado medio con respecto al número de operaciones.

Para estudiar la complejidad computacional y su dependencia con el número de elementos de las matrices de entrada de cada *timestep*, en nuestro caso 64x64, debemos considerar la complejidad computacional de las operaciones de convolución, deconvolución y *pixel shuffle*, ya que en base a ellas se construyen todos los modelos que hemos implementado.

Para una capa de convolución, el número de operaciones es proporcional al número de elementos de salida de la capa de convolución, ya que se aplica a cada una de las posiciones. El tamaño de salida de una capa de convolución aplicada sobre una matriz de tamaño nxm, un padding p, un stride s y kernel k es:

$$\left(\frac{n+2p-k}{s}+1\right)\left(\frac{m+2p-k}{s}+1\right)$$

A su vez, en cada elemento se realizan k^2 multiplicaciones y k^2 - 1 sumas. Esto tiene que aplicarse a cada uno de los C canales de entrada. Si la capa tiene F filtros, tenemos un total de operaciones de:

$$\left(\frac{n+2p-k}{s}+1\right)\left(\frac{m+2p-k}{s}+1\right)(2k^2-1)*C*F$$

Lo cual, al considerar la complejidad computacional temporal en función del tamaño de la matriz de datos de entrada nos da $O(NM)$, en el caso de una matriz cuadrada de lado N, $O(N^2)$.

Para una capa de convolución traspuesta o deconvolución[6], el tamaño de salida de una capa de convolución traspuesta aplicada sobre una matriz de tamaño nxm, un padding p, un stride s y kernel k es:

$$\big((m-1)*s*k*2p\big)\big((n-1)*s*k*2p\big)$$

De nuevo tendremos por cada elemento k^2 multiplicaciones y k^2 - 1 sumas aplicados a cada uno de los C canales de entrada para un total de F filtros. Tendremos un total de operaciones:

$$\big((m-1)*s*k*2p\big)\big((n-1)*s*k*2p\big)(2k^2-1)*C*F$$

Por lo tanto, tenemos la misma complejidad computacional que para el caso de la convolución, $O(N^2)$ para una matriz cuadrada de lado N.

Finalmente, el *pixel shuffle* es una reordenación de los elementos de un tensor, con lo que tiene una complejidad computacional proporcional al número de elementos reordenados, igualmente a las funciones de activación ReLu y Tanh, en nuestro caso para una matriz cuadrada de lado N es $O(N^2)$.

Al estar todos los modelos construidos con capas de complejidad $O(N^2)$ o $O(N)$, siendo N el lado de la matriz espacial cuadrada, es esta la complejidad temporal que tienen todos los modelos con respecto al número de datos de entrada.

Para comparar la complejidad espacial, al estar todos los parámetros de los modelos representados por números de coma flotante con el mismo número de bits, nos es suficiente con analizar la cantidad de parámetros utilizados.

	NaiveTransConv	ED_PixelShuffle	ED_TransConv	ESPCN	FSRCNN	SRCNN	VDSR
Params(K)	111.107	649.136	504.515	37.2	32.603	69.251	335.875

Tabla 3: Número de parámetros (en miles) de cada arquitectura

En la siguiente figura se muestra una representación gráfica del error cuadrado medio del modelo (menos es mejor) con respecto al número de parámetros utilizados:

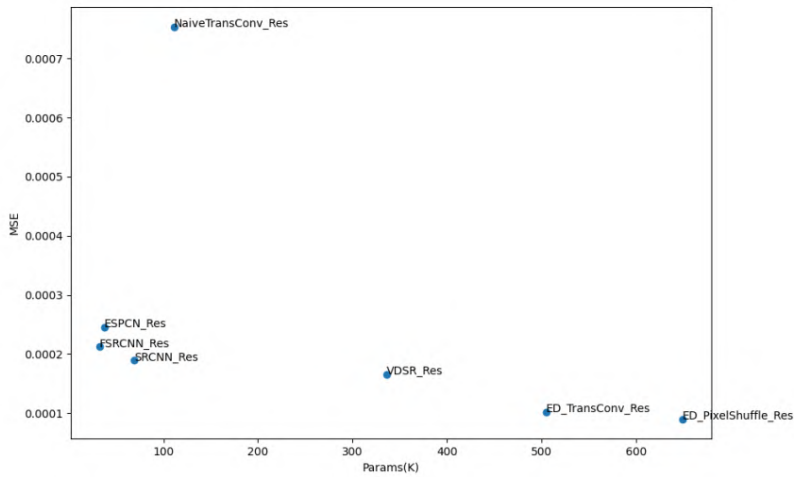


Figura 21: Error cuadrado medio con respecto al número de parámetros.

Vemos como, tomando como referencia una misma estrategia de optimización, se produce una mejora del rendimiento de los modelos a medida en que aumenta el número de parámetros utilizados. A su vez, comparando los modelos con un mejor rendimiento y mayor número de parámetros, vemos que los dos modelos de arquitectura codificador-decodificador propuestos que se expanden en anchura en lugar de profundidad consiguen un buen resultado sin aumentar de forma excesiva el número de operaciones realizadas, como vimos en la Tabla 2, en contraste con VDSR que tiene un entrenamiento considerablemente más lento y costoso. En cualquier caso, la complejidad espacial de los modelos utilizados es, en general, muy reducida para un computador actual, siendo el máximo aproximadamente de 650.000 parámetros los cuales puede almacenarse en unos pocos MB.

En la práctica, la complejidad temporal de realizar el proceso de aumento de resolución de todos los modelos es muy pequeña en comparación con el proceso de generación de las simulaciones, y todos pueden completarse en una CPU corriente en varios minutos. La siguiente tabla muestra los tiempos de ejecución de una simulación completa (320 *timesteps* representados con una matriz de entrada 64x64 cada uno, con un aumento de resolución de 4x) en la CPU del equipo donde se ha realizado el trabajo (AMD Ryzen 5 5600X).

	NaiveTransConv	ED_PixelShuffle	ED_TransConv	ESPCN	FSRCNN	SRCNN	VDSR
Tiempo (s)	41.5	6.2	10.6	1.6	2.7	13.9	64.6

Tabla 4: Muestra comparativa de tiempos de ejecución en CPU.

Alternativamente, utilizando la paralelización que permiten las tarjetas gráficas al hacer operaciones matriciales, se reducen los tiempos de ejecución a pocos segundos como máximo. En contraste, la realización de una simulación a alta resolución (255x255) nos ocupa alrededor de dos horas y media, mientras que una simulación a cuatro veces menos resolución nos toma alrededor de media hora. Vemos que, en comparación, la ejecución de los modelos es despreciable en tiempo de cómputo.

2.4.6 Viabilidad de la metodología utilizada

Podemos detectar las siguientes limitaciones en la metodología seguida en el proceso de entrenamiento de los modelos de aprendizaje profundo descritos, las cuales implicarían un estudio futuro más profundo en el dominio del problema para poder así evaluar la viabilidad de la metodología.

1. Los modelos no tienen capacidad de utilizar la información temporal presente en el conjunto de datos, ya que todos los modelos implementados realizan el proceso de *supersampling* en la dimensión espacial sin tener como datos de entrada el resto de *timesteps*. Esto da pie a futuras investigaciones donde se utilicen modelos capaces de incorporar estos datos, como modelos con capas de convolución en tres dimensiones o modelos recurrentes que sean capaces de almacenar características de una secuencia en su estado interno.
2. Al haberse realizado una simulación de la generación datos de baja resolución a partir de un *downsample* de los datos *ground-truth* de alta resolución, para mantener así la equivalencia en la evolución de la simulación, la validez física de la metodología queda siempre supeditada a la capacidad de una simulación de baja resolución a capturar los patrones relevantes a la escala de interés.
3. El proceso de aumento de resolución se realiza únicamente en la dimensión espacial de la simulación. No se ha explorado el proceso de interpolación temporal.
4. La mayoría de los modelos consultados e implementados han sido desarrollados con el objetivo del aumento de resolución de imágenes, mientras que los datos a los que se aplican, aunque estén estandarizados, representan variables físicas de una naturaleza muy distinta que la relación existente entre los canales (colores) de una imagen. Por ello este trabajo sirve para evaluar el rendimiento de estas arquitecturas en este campo de aplicación, pero quedan pendientes evaluar modelos con arquitecturas distintas.

Aun así, vemos que los modelos demuestran una alta capacidad de mejorar una interpolación trivial que muestran que potencialmente son capaces de capturar y aprovechar la relación contenida en la relación entre las variables físicas. El uso de redes convolucionales es especialmente eficiente para el problema dada la invarianza espacial de los fenómenos físicos. Con todo, el proceso de realizar una simulación a baja resolución y aplicar posteriormente un modelo como los estudiados se realiza a un coste computacional significativamente más bajo que realizar una simulación directa a alta resolución, y el error inducido es significativamente menor que una interpolación bicúbica.

2.5 Representación simbólica

2.5.1 Modelo y algoritmo implementado

El código implementado en esta sección se encuentra íntegramente incluido en el archivo **symbolic_representation.py**[21]**.** El modelo de representación simbólica implementado tiene en su base una estructura de árbol binario, dado que los operadores tienen como máximo dos operandos. Se ha implementado la estructura de este árbol mediante una clase que representa un nodo, cada uno con dos atributos que apuntan a su hijo izquierdo y derecho, y otro atributo que representa el valor del nodo, lo que nos permite construir los árboles completos de forma recursiva a partir del nodo raíz.

```
class Node:
    def __init__(self, value, left=None, right=None):
        self.value = value  # Can be an operation or an operand
        self.childs = 0
        self.left = left
        self.right = right

def tree_depth(node):
    if node is None:
        return 0
    else:
        left_depth = tree_depth(node.left)
        right_depth = tree_depth(node.right)
        return max(left_depth, right_depth) + 1
```

Los valores de los nodos pueden contener el siguiente conjunto definido de operaciones:

```
operations = {
    '+': operator.add,
    '-': operator.sub,
    '*': operator.mul,
    '/': operator.truediv,
    '^2': power2,
    '^3': power3,
    'sin': np.sin,
    'exp': np.exp,
    'log': np.log,
}
```

Los nodos pueden representar un número real constante o alguno de los datos de entrada:

```
variables = ['x','y',
             'U00','U10','U01','U11',
             'V00','V10','V01','V11',
             'P00','P10','P01','P11',
             'x0','y0','x1','y1']
```

Siendo (x, y) la posición en la malla (a alta resolución), x_0, y_0, x_1, y_1 los valores de posición contiguos correspondientes inferiores (subíndice 0) y superiores (subíndice 1) en la malla de baja resolución, Y P_{ij}, U_{ij}, V_{ij} los valores de la presión, componente horizontal de la velocidad y componente vertical de la velocidad, en cada uno de los puntos de baja resolución.

La salida del modelo es un único valor real, y se utiliza esta arquitectura para intentar encontrar relaciones entre una variable de la simulación en un punto de la simulación a alta resolución en función de los valores más cercanos equivalentes a baja resolución. Matemáticamente se intenta encontrar un conjunto de relaciones:

$$P(x,y) = F_p(P_{00}, P_{01}, P_{10}, P_{11}, U_{00}, U_{01}, U_{10}, U_{11}, V_{00}, V_{01}, V_{10}, V_{11}, x_0, y_0, x_1, y_1, x, y)$$
$$U(x,y) = F_u(P_{00}, P_{01}, P_{10}, P_{11}, U_{00}, U_{01}, U_{10}, U_{11}, V_{00}, V_{01}, V_{10}, V_{11}, , x_0, y_0, x_1, y_1, x, y)$$
$$V(x,y) = F_v(P_{00}, P_{01}, P_{10}, P_{11}, U_{00}, U_{01}, U_{10}, U_{11}, V_{00}, V_{01}, V_{10}, V_{11}, , x_0, y_0, x_1, y_1, x, y)$$

Se espera que dichas relaciones puedan encontrar un patrón de interpolación utilizando estos valores vecinos, análogo a por ejemplo la interpolación bilineal, que pueda aprovechar también las relaciones entre las variables físicas.

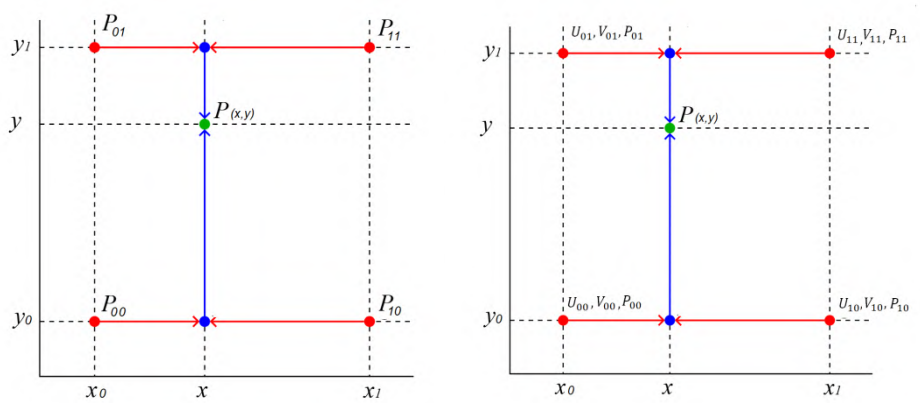


Figura 22: Representación de variables utilizadas en interpolación bilineal (a la izquierda) e interpolación buscada que hace uso de todas las variables (derecha).

La siguiente sección de código se utiliza para seleccionar los datos y almacenarlos en un array de acuerdo con este planteamiento:

```
DOWNSAMPLE_FACTOR = 4
downsampler = nn.AvgPool2d(DOWNSAMPLE_FACTOR)

#Import data
data_raw = np.load('./data/trainData.npy')
data = np.empty((MAX_DATA, 19))  #Allocate spacebefore hand
dataIndex = 0

for i in range(data_raw.shape[0]):
    if dataIndex > MAX_DATA: break
    data_hr = data_raw[i]
    data_lr = downsampler(torch.tensor(data_hr))
    data_hr = np.transpose(data_hr,(0,2,3,1))
    data_lr = np.transpose(data_lr,(0,2,3,1))
    for t in range(data_hr.shape[1]):
        data_iteration_hr = data_hr[t]
        data_iteration_lr = data_lr[t]
        for i in range(0,data_iteration_hr.shape[0]-DOWNSAMPLE_FACTOR):
            for j in range(0,data_iteration_hr.shape[1]-DOWNSAMPLE_FACTOR):
                    i_lr_equivalent = math.floor(i/(DOWNSAMPLE_FACTOR))
                    j_lr_equivalent = math.floor(j/(DOWNSAMPLE_FACTOR))

                    x0 = i_lr_equivalent/data_iteration_lr.shape[0]
                    y0 = j_lr_equivalent/data_iteration_lr.shape[0]
                    x1 = (i_lr_equivalent+1)/data_iteration_lr.shape[0]
                    y1 = (j_lr_equivalent+1)/data_iteration_lr.shape[0]

                    x = i/data_iteration_hr.shape[0]
                    y = j/data_iteration_hr.shape[0]

                    u00 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent][0]
                    u10 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent][0]
                    u01 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent+1][0]
                    u11 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent+1][0]
                    v00 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent][1]
                    v10 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent][1]
                    v01 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent+1][1]
                    v11 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent+1][1]
                    p00 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent][2]
                    p10 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent][2]
                    p01 = data_iteration_lr[i_lr_equivalent][j_lr_equivalent+1][2]
                    p11 = data_iteration_lr[i_lr_equivalent+1][j_lr_equivalent+1][2]
```

```
                    z = data_iteration_hr[i][j][2]
                    new_row = np.array((z, x,y,
                                u00,u10,u01,u11,
                                v00,v10,v01,v11,
                                p00,p10,p01,p11,
                                x0,y0,x1,y1)).reshape(1, -1)
                    if dataIndex >= MAX_DATA: break
                    data[dataIndex] = new_row
                    dataIndex += 1
    del data_hr, data_lr
data[:dataIndex]
```

Para realizar una comprobación de la equivalencia entre árboles y una limpieza de árboles duplicados que nos ayude a diversificar la población se opta por simplificar este problema asumiendo heurísticamente que dos árboles que tengan el mismo valor de *fitness* son idénticos, ya que la probabilidad de un valor idéntico de dos funciones distintas en un número almacenado con coma flotante con 64 bits es extremadamente baja, y el coste de comprobar realmente los árboles haciendo un recorrido en profundidad o en anchura aumenta drásticamente el coste temporal del algoritmo. Esta es de hecho la primera aproximación que se utilizó, pero en este punto el algoritmo ya mostraba limitaciones en la complejidad computacional y los tiempos de entrenamiento eran inasumibles en el hardware utilizado. Alternativas como desarrollar funciones de similitud también son descartadas ya que implican un recorrido por los árboles y la naturaleza de algunas operaciones matemáticas que tienen propiedades distributivas, asociativas o distributivas implica una gran cantidad de expresiones matemáticas equivalentes con árboles distintos que deben ser consideradas. Así, la aproximación de utilizar fitness como métrica para comparar los árboles, aunque tiene limitaciones teóricas, resulta suficiente en la práctica y permite reutilizar el cálculo del valor de fitness que debe hacerse para el proceso de selección.

Se definen las funciones utilizadas en el proceso de selección. La función *create_random_tree* nos permite inicializar un árbol de una profundidad dada. Con las funciones *evaluate_tree* y *fitness* calculamos, utilizando recursividad, el valor de un árbol ante unos parámetros de entrada. Se programan también funciones para el proceso de selección. La función *select_population* realiza una selección de una población de árboles dada mediante la selección iterativa del mejor árbol de una serie de subconjuntos de la población general (función *tournament_selection*). Finalmente, se disponen de funciones para eliminar árboles duplicados en una población dada e imprimir un árbol en forma de ecuación.

```
# Function to generate a random constant between -10 and 10
def random_constant():
    return random.uniform(-10, 10)

def create_random_tree(depth=3):
    if depth == 0 or (depth != -1 and random.random() > 0.5):
        if random.random() < 0.5:
            return Node(random_constant())  # Return a constant
        else:
            var = random.choice(variables)
            return Node(var)  # Return a variable
    else:
        operation = random.choice(list(operations.keys()))
        if operation in two_operads_op:
            return Node(operation, create_random_tree(depth-1), create_random_tree(depth-1))
        else:
            return Node(operation, create_random_tree(depth-1), None)

def evaluate_tree(tree, input_value):
    if tree.value in operations:
        left_val = evaluate_tree(tree.left, input_value)
        right_val = evaluate_tree(tree.right, input_value) if tree.right != None else None
        if right_val != None:
```

```
            return operations[tree.value](left_val, right_val)
        else:
            return operations[tree.value](left_val)
    elif tree.value in variables:
        i = variables.index(tree.value)
        return input_value[i]
    else: #Is constant
        return tree.value

def fitness(tree, data, printPred=False):
    error = 0
    for datapoint in data:
        y = datapoint[0]
        x = datapoint[1:]
        try:
            prediction = evaluate_tree(tree, x)
            if printPred: print("Pred value", prediction)
            if np.iscomplex(prediction):
               return float('inf')  # Penalize invalid equations
            else:
                error += (prediction - y) ** 2 #Squared error
        except: #Exception as e:
            return float('inf')  # Penalize invalid equations
    return   error/data.shape[0] if not np.isnan(error) else float('inf') # Penalize invalid equations
#Selection
def tournament_selection(population, fitness_values, tournament_size=10):
    tournament = random.sample(population, tournament_size)
    sample_index = np.random.choice(len(population),tournament_size)
    tournament = np.take(population, sample_index)
    tournament_fitness = np.take(fitness_values, sample_index)

    fittest_index = np.argmin(tournament_fitness)
    return tournament[fittest_index]

def select_population(current_population, fitness_values, size=5):
    new_population = []
    for _ in range(size):
        individual = tournament_selection(current_population,fitness_values)
        new_population.append(individual)
    return new_population

def remove_duplicate_trees_by_fitness(population, fitness_values):
    unique_trees = []
    unique_fitness_values = []

    for idx, tree in enumerate(population):
        if not any(abs(fitness_values[idx] - other_fitness) < FITNESS_THRESHOLD
                   for other_fitness in unique_fitness_values):
            unique_trees.append(tree)
            unique_fitness_values.append(fitness_values[idx])
    return unique_trees, unique_fitness_values
def print_population(population):
    sample_index = np.random.choice(data.shape[0],round(DATA_SAMPLE/10))
    for individual in population:
        print(fitness(individual, np.take(data,sample_index,0)), " | z =",tree_to_equation(individual))
```

Se implementan dos mecanismos para aumentar la diversidad en los modelos. Por un lado, la mutación de nodos, que cambia el contenido (operador u operando) de un nodo de forma aleatoria, y por otro el *crossover* o cruce de nodos (con todos sus hijos) entre dos árboles dados. La siguiente sección de código implementa estas operaciones que alteran un árbol.

```
def mutate(tree, mutation_rate=0.1):
    if (tree == None): return
    if random.random() < mutation_rate:
        if random.random() < 0.5:
            # Mutate the current node's value
            if isinstance(tree.value, (int, float)):  # If it's a constant
                tree.value = random_constant()
            else:
                tree.value = random.choice(list(operations.keys()) + variables)
        else:
            # Replace a subtree
            if random.random() < 0.5 and tree.left:
                tree.left = create_random_tree(depth=random.randint(1, 3))
            elif tree.right:
                tree.right = create_random_tree(depth=random.randint(1, 3))
    else:
        if tree.left != None:
            mutate(tree.left, mutation_rate)
        if tree.right != None:
            mutate(tree.right, mutation_rate)
def crossover(parent1, parent2, crossover_rate=0.1):
    if random.random() < crossover_rate:
        # Randomly choose a node from each parent and swap
        node1 = random.choice(get_all_nodes(parent1))
        node2 = random.choice(get_all_nodes(parent2))
        if (node1 == None or node2 == None): return
        node1.value, node2.value = node2.value, node1.value
        node1.left, node2.left = node2.left, node1.left
        node1.right, node2.right = node2.right, node1.right
```

El entrenamiento se basa en un algoritmo genético que emula a la selección natural y parte de árboles generados de estructura aleatoria. El algoritmo opera sobre los árboles binarios, seleccionándolos y mutándolos durante generaciones. Se evalúa cada árbol mediante una función de ajuste (fitness) que cuantifica la adecuación del árbol a los datos, y en base a este valor se seleccionan a los N modelos con mejor ajuste. En este caso el algoritmo minimiza una función que calcula el error medio cuadrado entre la salida de los modelos y el valor de alta resolución o *ground-truth*. Se selecciona finalmente tras un número determinado de generaciones al modelo individual que tenga un mejor ajuste.

```
# Parameters
tree_initial_depth = 3
population_size = 300
N_SELECTION = round(population_size/3)
DATA_SAMPLE = 50000 #Amount of data used to train each generation, which will be randomly sampled. We have to
do this to deal with the limitations of the algorithm.
max_generations = 100
desired_fitness = 0  # Set a threshold for desired fitness
crossover_rate = 0.3
mutation_rate = 0.3
# Create initial population
population = [create_random_tree(tree_initial_depth) for _ in range(population_size)]
best_individual = None
best_fitness = float('inf')
best_copies = 3

for generation in tqdm(range(max_generations), desc="Optimizing...", ascii=False, ncols=64):
    # Evaluate fitness
    sample_index = np.random.choice(data.shape[0],DATA_SAMPLE)
    fitness_values = [fitness(individual, np.take(data,sample_index,0)) for individual in population]

    # Remove duplicate trees based on fitness
    population, fitness_values = remove_duplicate_trees_by_fitness(population, fitness_values)

    # Find the best individual
    current_best_idx = fitness_values.index(min(fitness_values))
    current_best_individual = population[current_best_idx]
    current_best_fitness = fitness_values[current_best_idx]
    if current_best_fitness < best_fitness:
        best_fitness = current_best_fitness
        best_individual = current_best_individual
        print(f"\nGeneration {generation}: New best fitness = {best_fitness:.2e} Eq: z =
{tree_to_equation(best_individual)}")
    # Check for termination condition
    if best_fitness <= desired_fitness:
        print(f"Desired fitness level reached at generation {generation}")
        break

    # Selection of N best individuals
    new_population = select_population(population, fitness_values, size = N_SELECTION)
    #Explicitly include best individual
    new_population.append(best_individual)
    #Add a few mutations of best individual
    for i in range(best_copies):
        child_best = copy.deepcopy(best_individual)
        mutate(child_best,mutation_rate)
        new_population.append(child_best)

    if generation > 0 and generation % 5 == 0: print_population(new_population[-10:])

    selectedPopulation = len(new_population)
    # Crossover and mutation until filling population
    while (len(new_population) < population_size):
        #Sample two parents and create child
        parent1 = new_population[randrange(selectedPopulation)]
        parent2 = new_population[randrange(selectedPopulation)]
        child1, child2 = copy.deepcopy(parent1), copy.deepcopy(parent2)

        #Crossover and mutate
        if random.random() < 0.5:
            crossover(child1, child2, crossover_rate)
        else:
            mutate(child1,mutation_rate)
            mutate(child2,mutation_rate)
        #Introduce new random tree
        child3 = create_random_tree(tree_initial_depth)
```

```
        new_population.extend([child1, child2, child3])

    population = new_population

print("Fitness: ",fitness(best_individual,data,printPred=False))
print("Best individual:", best_individual)
print("Best fitness:", best_fitness)
print(tree_to_equation(best_individual))
```

2.5.1 Resultados obtenidos

Tras una exploración de los distintos hiperparámetros, el modelo de representación simbólica implementado solo ha sido capaz de encontrar una solución distinta a la interpolación por un vecino cercano y la interpolación bilineal, las cuales además de ser una solución trivial ya conocida no utilizan ninguna otra variable física como parámetro de entrada distinta a la que se está interpolando.

En particular se ha observado que el entrenamiento de estos modelos para el problema planteado es altamente costoso y toma demasiado tiempo para el conjunto de datos utilizados. Además, el hecho de haber definido la función del problema con un alto número de variables de entrada hace que la probabilidad de que aleatoriamente se produzca una variación en el árbol que sea una ecuación válida y mejore el coste es muy baja. Con ello, el uso del modelo implementado de representación simbólica ha sido insatisfactorio y no hemos sido capaces de encontrar relaciones entre las variables físicas que ofrezcan una mejora a una interpolación bilineal trivial.

3. Conclusiones

En este estudio, hemos aplicado técnicas avanzadas de Aprendizaje Computacional para demostrar la posibilidad de aumentar la eficiencia computacional de la realización de simulaciones de dinámica de fluidos sin sacrificar de manera significativa la precisión de los resultados utilizando técnicas de *upsampling* o aumento de resolución de un conjunto de datos.

Los modelos de Aprendizaje Profundo desarrollados representan una alternativa prometedora a los enfoques numéricos convencionales, sobre todo en situaciones donde los recursos computacionales son escasos. Sin embargo, es importante reconocer que la aplicabilidad de esta metodología en un contexto físico específico necesita ser examinada cuidadosamente en cada caso, ya que la representación de baja resolución necesita ser suficientemente representativa dada la escala de interés del problema físico. Nuestro trabajo, por tanto, intenta establecer un marco de referencia para evaluar diferentes arquitecturas en los modelos de Aprendizaje Profundo aplicadas a la tarea de aumento de resolución de simulaciones de dinámica de fluidos.

En las comparativas de Aprendizaje Profundo, se han explorado y comparado las distintas arquitecturas bajo las estrategias de optimizar el modelo para inferir directamente el resultado a alta resolución y, alternativamente, optimizar el modelo para inferir las diferencias (residuos) entre una interpolación bicúbica y la representación de alta resolución. Ésta última metodología ha producido los mejores resultados dada la naturaleza y distribución de los datos de la simulación, y se demuestras que es mucho más fácil modelar estos residuos con las arquitecturas implementadas. Por otro lado, el uso de la función de coste que minimiza la diferencia entre los residuos de las ecuaciones de Navier Stokes que rigen la simulación no ha sido satisfactorio, no mostrándose una diferencia significativa con una optimización simple del error medio cuadrado. Se requiere por tanto una mayor profundización e investigación en esta metodología para el problema planteado.

Además, se ha mostrado como, en los dos modelos propuestos de Aprendizaje Profundo utilizando redes convolucionales, el uso de un mayor número de parámetros y, por extensión, de filtros, mientras se utiliza un bajo número de capas, consigue extraer información de los datos de forma más eficiente y aumentar con ello el rendimiento sin aumentar drásticamente el coste computacional. La comparativa hecha sobre número de operaciones y complejidad computacional muestra como estos modelos son más rápidos al entrenar y computar que otras alternativas de un alto número de parámetros y mayor profundidad.

También hemos comprobado como el uso de la técnica de *Sub-Pixel Deconvolution*, mediante una capa convolucional y una operación de *Pixel-Shuffle,* desarrollada por los autores del modelo ESPCNN[38], mejoran el resultado con respecto a la utilización de una capa de convolución traspuesta para aumentar la resolución de los datos de entrada en el modelo propuesto.

La Representación Simbólica, por su parte, nos ha servido como elemento comparativo y ha puesto en evidencia algunas de sus limitaciones al tratar con una gran cantidad de datos y una gran cantidad de variables de entrada. Hemos observado en el desarrollo del trabajo como la computación paralelizada implementada en la librería pytorch y los módulos CUDA hacen este proceso mucho más eficiente. Con ello, no se han logrado los objetivos planteados mediante esta vía dada la dificultad en el problema planteado, las limitaciones en la implementación realizada del algoritmo de búsqueda implementado y las limitaciones en el tiempo disponible.

En comparación, hemos visto como la utilización de modelos de redes convolucionales ofrece una convergencia clara y un mejor rendimiento en la tarea propuesta en los modelos y metodologías utilizados. Por un lado, el entrenamiento utilizando la arquitectura CUDA y la paralelización de las GPU nos ha permitido entrenar más modelos y explorar las opciones con una mayor profundidad en el tiempo de realización de este trabajo. Por otro, la arquitectura general de las CNN es especialmente útil a la hora de capturar patrones que son invariantes en la dimensión espacial y aparecen de forma recurrente en distintas partes del dominio, como es el caso de un fenómeno físico donde no existe ninguna posición privilegiada y todos los puntos del espacio están regidos por las mismas ecuaciones.

Mirando hacia el futuro, existe un vasto campo para explorar en la mejora y optimización de estos modelos. La integración de técnicas más avanzadas de Aprendizaje Profundo, así como la expansión hacia otros tipos de ecuaciones y sistemas físicos, podría abrir nuevas vías de investigación y desarrollo. En concreto existe un gran potencial integrando en los modelos la dimensión temporal de las simulaciones, utilizando la evolución de las variables físicas a lo largo del tiempo. Para ello pueden utilizarse capas de convolución en tres dimensiones, siendo una de ellas la temporal, y modelos recurrentes que consigan almacenar información relevante en el estado interno del modelo.

Durante la realización del trabajo ha debido modificarse la planificación debido a retrasos en la obtención de datos mediante las simulaciones, las cuales se solaparon con las etapas de implementación. Además, este retraso continúa también en la primera etapa de entrenamiento, que finalmente se ha realizado paralelamente a la redacción de la memoria del trabajo.

La completitud de este trabajo tiene como objetivo aportar una comparación entre distintas arquitecturas aplicables al problema planteado que permita servir de referencia a la hora de implementar modelos en problemas similares dadas unos recursos de hardware equivalentes a un PC doméstico. Con ello, hemos podido ver como la combinación de Aprendizaje Profundo con técnicas de simulación abre un camino prometedor en la modelización computacional, sugiriendo un futuro en el que los modelos predictivos y de alta precisión son más rápidos y menos costosos de ejecutar, lo que tiene implicaciones positivas en la dinámica de fluidos computacional y otros muchos campos de la ciencia y la ingeniería donde aún se depende de simulaciones mediante métodos numéricos de diferenciales finitos y las limitaciones en el coste computacional de realizar estas simulaciones son aún significativas.

4. Glosario

PDE: acrónimo de *Partial Differential Equations* o Ecuaciones en derivadas parciales.

CFD: acrónimo de *Computational Fluid Dynamics* o Dinámica de fluidos computacional.

GPU: acrónimo de *Graphic Processing Unit* o Unidad de Procesamiento Gráfico, utilizado para referirse a la arquitectura implementada en las tarjetas gráficas.

Learning rate: en español ratio de aprendizaje, es una constante que regula el tamaño del salto en cada paso del descenso de gradiente

Batch: conjunto de datos utilizados en el proceso de optimización de modelos

Mini-batch: subconjunto del *batch* utilizado en un paso del proceso de optimización de modelos.

Kernel: matriz de parámetros utilizada como filtro en las redes convolucionales.

Ground-truth: conjunto de datos utilizado como exactos en el proceso de optimización de modelos de aprendizaje computacional. Nótese que no tienen por qué ser conceptualmente exactos, como es el caso de este trabajo, sino que representan el comportamiento ideal del modelo.

Residuo: diferencia entre un valor “real” o *ground-truth* y un valor inferido por la aproximación de un modelo aproximado.

Outlier: dado un conjunto de datos, los *outliers* son aquellos puntos que se alejan de forma extrema del comportamiento general de la mayoría de los datos.

5. Bibliografía

[1] J. J. Hopfield, «Neural networks and physical systems with emergent collective computational abilities», *Proc. Natl. Acad. Sci. U. S. A.*, vol. 79, n.º 8, pp. 2554-2558, abr. 1982, doi: 10.1073/pnas.79.8.2554.

[2] F. Rosenblatt, «The perceptron: a probabilistic model for information storage and organization in the brain», *Psychol. Rev.*, vol. 65, n.º 6, pp. 386-408, nov. 1958, doi: 10.1037/h0042519.

[3] R. Raina, A. Madhavan, y A. Y. Ng, «Large-scale deep unsupervised learning using graphics processors», en *Proceedings of the 26th Annual International Conference on Machine Learning*, Montreal Quebec Canada: ACM, jun. 2009, pp. 873-880. doi: 10.1145/1553374.1553486.

[4] D. E. Rumelhart, G. E. Hinton, y R. J. Williams, «Learning representations by back-propagating errors», *Nature*, vol. 323, n.º 6088, Art. n.º 6088, oct. 1986, doi: 10.1038/323533a0.

[5] K. O'Shea y R. Nash, «An Introduction to Convolutional Neural Networks». arXiv, 2 de diciembre de 2015. doi: 10.48550/arXiv.1511.08458.

[6] M. D. Zeiler, D. Krishnan, G. W. Taylor, y R. Fergus, «Deconvolutional networks», en *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Francisco, CA, USA: IEEE, jun. 2010, pp. 2528-2535. doi: 10.1109/CVPR.2010.5539957.

[7] «ImageNet classification with deep convolutional neural networks | Communications of the ACM». Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: https://dl.acm.org/doi/10.1145/3065386

[8] W. Zhiqiang y L. Jun, «A review of object detection based on convolutional neural network», en *2017 36th Chinese Control Conference (CCC)*, jul. 2017, pp. 11104-11109. doi: 10.23919/ChiCC.2017.8029130.

[9] S. Park y Y.-G. Shin, «Generative convolution layer for image generation», *Neural Netw.*, vol. 152, pp. 370-379, ago. 2022, doi: 10.1016/j.neunet.2022.05.006.

[10] S. Nikolopoulos, I. Kalogeris, y V. Papadopoulos, «Non-intrusive Surrogate Modeling for Parametrized Time-dependent PDEs using Convolutional Autoencoders». arXiv, 23 de abril de 2021. doi: 10.48550/arXiv.2101.05555.

[11] P. Ren, C. Rao, Y. Liu, J.-X. Wang, y H. Sun, «PhyCRNet: Physics-informed convolutional-recurrent network for solving spatiotemporal PDEs», *Comput. Methods Appl. Mech. Eng.*, vol. 389, p. 114399, feb. 2022, doi: 10.1016/j.cma.2021.114399.

[12] J. Kubalík, E. Derner, y R. Babuška, «Toward Physically Plausible Data-Driven Models: A Novel Neural Network Approach to Symbolic Regression», *IEEE Access*, vol. 11, pp. 61481-61501, 2023, doi: 10.1109/ACCESS.2023.3287397.

[13] A. Chen y G. Lin, «Robust data-driven discovery of partial differential equations with time-dependent coefficients». arXiv, 2 de febrero de 2021. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: http://arxiv.org/abs/2102.01432

[14] R. Majumdar, V. Jadhav, A. Deodhar, S. Karande, L. Vig, y V. Runkana, «Symbolic Regression for PDEs using Pruned Differentiable Programs».

arXiv, 13 de marzo de 2023. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: http://arxiv.org/abs/2303.07009
[15] M. Maslyaev, A. Hvatov, y A. Kalyuzhnaya, «Data-driven PDE discovery with evolutionary approach», vol. 11540, 2019, pp. 635-641. doi: 10.1007/978-3-030-22750-0_61.
[16] D. Angelis, F. Sofos, y T. E. Karakasidis, «Artificial Intelligence in Physical Sciences: Symbolic Regression Trends and Perspectives», *Arch. Comput. Methods Eng.*, vol. 30, n.º 6, pp. 3845-3865, jul. 2023, doi: 10.1007/s11831-023-09922-z.
[17] M. Raissi, P. Perdikaris, y G. E. Karniadakis, «Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations», *J. Comput. Phys.*, vol. 378, pp. 686-707, feb. 2019, doi: 10.1016/j.jcp.2018.10.045.
[18] R. Majumdar, V. Jadhav, A. Deodhar, S. Karande, L. Vig, y V. Runkana, «Physics Informed Symbolic Networks». arXiv, 20 de diciembre de 2022. doi: 10.48550/arXiv.2207.06240.
[19] S.-M. Udrescu y M. Tegmark, «AI Feynman: a Physics-Inspired Method for Symbolic Regression». arXiv, 15 de abril de 2020. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: http://arxiv.org/abs/1905.11481
[20] J. A. Guerrero Barberán, «Computational Fluid Dynamics Simulations». 16 de enero de 2024. [En línea]. Disponible en: https://kaggle.com/datasets/443e92c7a44dfa09b448cc6acd1057b9e2b5e061afe1b2b757d609631cb6e2bb
[21] Jose Antonio Guerrero Barberán, «Repositorio Github-TFG J.A.Guerrero». [En línea]. Disponible en: https://github.com/jguerbar/TFG_JoseAntonioGuerreroBarberan_UOC
[22] B. Rehm, D. Consultant, A. Haghshenas, A. S. Paknejad, y J. Schubert, «CHAPTER TWO - Situational Problems in MPD», en *Managed Pressure Drilling*, B. Rehm, J. Schubert, A. Haghshenas, A. S. Paknejad, y J. Hughes, Eds., Gulf Publishing Company, 2008, pp. 39-80. doi: 10.1016/B978-1-933762-24-1.50008-5.
[23] «OpenFOAM». OpenCFD. [En línea]. Disponible en: https://www.openfoam.com/
[24] «FEATool Multiphysics». [En línea]. Disponible en: https://www.featool.com
[25] «Python.org», Python.org. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: https://www.python.org/
[26] A. Paszke *et al.*, «PyTorch: An Imperative Style, High-Performance Deep Learning Library». arXiv, 3 de diciembre de 2019. doi: 10.48550/arXiv.1912.01703.
[27] M. Abadi *et al.*, «TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems». arXiv, 16 de marzo de 2016. doi: 10.48550/arXiv.1603.04467.
[28] C. Dong, C. C. Loy, y X. Tang, «Accelerating the Super-Resolution Convolutional Neural Network». arXiv, 1 de agosto de 2016. doi: 10.48550/arXiv.1608.00367.
[29] K. He, X. Zhang, S. Ren, y J. Sun, «Deep Residual Learning for Image Recognition». arXiv, 10 de diciembre de 2015. doi: 10.48550/arXiv.1512.03385.

[30] E. Lauga, M. P. Brenner, y H. A. Stone, «Microfluidics: The no-slip boundary condition». arXiv, 28 de septiembre de 2005. Accedido: 13 de enero de 2024. [En línea]. Disponible en: http://arxiv.org/abs/cond-mat/0501557

[31] C. Dong, C. C. Loy, K. He, y X. Tang, «Image Super-Resolution Using Deep Convolutional Networks». arXiv, 31 de julio de 2015. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: http://arxiv.org/abs/1501.00092

[32] J. Kim, J. K. Lee, y K. M. Lee, «Accurate Image Super-Resolution Using Very Deep Convolutional Networks». arXiv, 11 de noviembre de 2016. Accedido: 25 de diciembre de 2023. [En línea]. Disponible en: http://arxiv.org/abs/1511.04587

[33] W. Shi *et al.*, «Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network». arXiv, 23 de septiembre de 2016. doi: 10.48550/arXiv.1609.05158.

[34] V. Badrinarayanan, A. Kendall, y R. Cipolla, «SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation». arXiv, 10 de octubre de 2016. Accedido: 16 de enero de 2024. [En línea]. Disponible en: http://arxiv.org/abs/1511.00561

[35] P. Ren, C. Rao, Y. Liu, J.-X. Wang, y H. Sun, «PhyCRNet: Physics-informed convolutional-recurrent network for solving spatiotemporal PDEs», *Comput. Methods Appl. Mech. Eng.*, vol. 389, p. 114399, feb. 2022, doi: 10.1016/j.cma.2021.114399.

[36] D. P. Kingma y J. Ba, «Adam: A Method for Stochastic Optimization». arXiv, 29 de enero de 2017. Accedido: 15 de enero de 2024. [En línea]. Disponible en: http://arxiv.org/abs/1412.6980

[37] «pytorch-OpCounter». [En línea]. Disponible en: https://github.com/Lyken17/pytorch-OpCounter

[38] W. Shi *et al.*, «Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network», n.º arXiv:1609.05158. arXiv, 23 de septiembre de 2016. doi: 10.48550/arXiv.1609.05158.