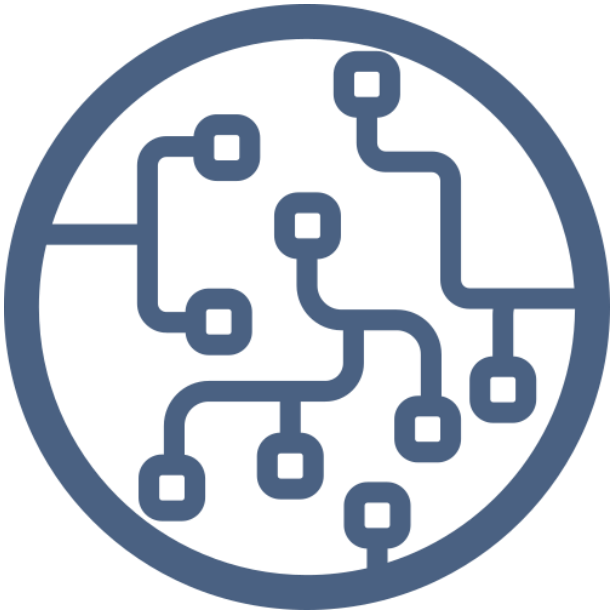
Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN INGENIERÍA DE CONTROL, AUTOMATIZACIÓN Y ROBÓTICA

TRABAJO FIN DE MASTER

DISEÑO Y SIMULACIÓN INTELIGENTES DE SISTEMAS FLUIDODINÁMICOS EN ESTADO NO ESTACIONARIO MEDIANTE DEEP LEARNING



Estudiante: Abucide Armas, Álvaro

Director/Directora: Zulueta Guerrero, Ekaitz

Curso: 2022/2023

Fecha: Bilbao, 10 de julio de 2023

Resumen

La dinámica de fluidos computacional (CFD, del inglés *Computational Fluid Dynamics*) aplicada a flujos turbulentos ha sido objeto de estudio durante los últimos años. Sin embargo, mediante las simulaciones CFD los costes computacionales se elevan considerablemente, siendo inviable el uso de estas técnicas para la resolución de ciertos problemas. Actualmente, se están desarrollando múltiples técnicas de *Deep Learning* (DL) aplicadas a los problemas de CFD. Una de las principales aplicaciones del DL al CFD es el empleo de redes neuronales convolucionales (CNN, del inglés *Convolutional Neural Networks*) para predecir los valores de las magnitudes fluido dinámicas deseadas alrededor de una geometría determinada. En este caso, se ha desarrollado una CNN que predice los campos de velocidades paralela al flujo y vertical y de presión. Se toman dos enfoques: la predicción de los instantes futuros partiendo de uno inicial y la predicción del instante inicial. La CNN adquiere como entradas las características geométricas de la forma analizada y el instante anterior, en el caso del enfoque temporal, siendo las salidas de la red los campos de ambas velocidades y de la presión, obtenidos mediante simulaciones CFD.

Abstract

The application of computational fluid dynamics (CFD) to turbulent flow has lately been a considerable topic of research. Nonetheless, the use of CFD tools results in large computational costs, which implies that, for some applications, CFD may be inviable. Several authors have conducted research applying deep learning (DL) techniques to CFD-based simulations to date. One of the main applications of DL with CFD is the use of convolutional neural networks (CNN) to predict the values of the desired fluid-dynamic magnitudes around a concrete geometry. In this case, a CNN which predicts the streamwise and vertical velocities and pressure fields has been developed. Two approaches are considered: the prediction of the future instants on the basis of an initial sample and the prediction of the initial sample. The CNN takes as inputs the geometric features of the analyzed form and the previous instant, in the case of the temporal approach, being the outputs of the net the coupled velocities and the pressure, obtained by CFD simulations.

Laburpena

Fluxu zurrunbilotsuei aplikatutako fluido konputazionalen dinamika (CFD, ingelesetik *Computational Fluid Dynamics*) aztertu da azken urteetan. CFD simulazioen bidez, ordea, kostu konputazionalak nabarmen igotzen dira, eta bidera ezina da teknika horiek erabiltzea zenbait problema ebazteko. Gaur egun, *Deep Learning* (DL) teknika ugari garatzen ari dira, CFDren arazoei aplikatuta. DLk CFDn duen aplikazio nagusietako bat sare neuronal konboluzionalak (CNN, ingelesetik *Convolutional Neural Networks*) erabiltzea da, geometria jakin baten inguruan nahi diren magnitude fluido dinamikoen balioak iragartzeko. Kasu honetan, CNN bat garatu da fluxuarekiko paraleloa eta bertikala diren abiadura-eremuak eta presio-eremua iragartzeko. Bi ikuspegi hartzen dira: hasierako une batetik abiatuz etorkizuneko uneen iragarpena eta hasierako unearen iragarpena. CNNk sarrera gisa hartzen ditu aztertutako formaren ezaugarri geometrikoak, eta aurreko unea denbora-ikuspegiaren kasuan. Sareko irteerak bi abiaduren eta presioaren eremuak dira, CFD simulazioen bidez lortuak.

Palabras clave:

Deep Learning, dinámica de fluidos computacional, redes neuronales convolucionales, U-Net, flujo turbulento

Índice

Lista de tablas

Lista de ilustraciones

Acrónimos

ADAM Estimación del momento adaptativo, del inglés *Adaptative Moment Estimation*

ANN Red neuronal artificial, del inglés *Artificial Neural Network*

ASO Optimización de formas aerodinámicas, del inglés *Aerodynamic Shape Optimization*

CFD Dinámica de fluidos computacional, del inglés *Computational Fluid Dynamics*

CNN Red neuronal convolucional, del ingés *Convolutional Neural Network*

DL *Deep Learning*

DNN Red neuronal profunda, del inglés *Deep Neural Network*

FRC Canal de la región de flujo, del inglés *Flow Region Channel*

GAN Red neuronal generativa adversaria, del inglés *Generative Adversarial Networks*

HPC Recursos computacionales de alta eficiencia, del inglés *high-performance computing*

IA Inteligencia Artificial

LGR Representación geométrica latente, del inglés *Latent Geometry Representation*

MAE Error medio absoluto, del inglés *Mean Absolute Error*

MSE Error medio cuadrático, del inglés *Mean Squared Error*

RANS Navier-Stokes promediadas por Reynolds, del inglés *Reynolds-Averaged Navier-Stokes*

ReLU Unidad lineal rectificada, del inglés *rectified linear unit*

RMSE Raíz del error cuadrático medio, del inglés *Root Mean Square Error*

RMSprop Propagación de la raíz cuadrática media, del inglés *root mean square propagation*

RNN Red neuronal recurrente, del inglés *Recurrent Neural Network*

SDF Función de distancia con signo, del inglés *Signed Distance Function*

SGD Gradiente de descenso estocástico, del inglés *stochastic gradient descent*

URANS Inestables de Navier-Stokes promediadas por Reynolds, del inglés *Unsteady Reynolds-Averaged Navier-Stokes*

1 Introducción y contexto

La Inteligencia Artificial (IA) se ha convertido en los últimos años en una tecnología de gran utilidad para la resolución de diversos problemas prácticos. Esto se debe, fundamentalmente, a las capacidades de aprendizaje, razonamiento y adaptación de los sistemas inteligentes, que permiten que los métodos de IA estén alcanzando niveles de rendimiento sin precedentes a la hora de resolver problemas computacionales complejos [6]. La IA se ha postulado como una herramienta de valiosa utilidad en aplicaciones como el procesamiento del lenguaje natural, el procesamiento de imágenes, los sistemas de visión artificial, el análisis de imágenes médicas, la mecánica computacional o la ingeniería aerospacial.

La optimización de formas aerodinámicas (ASO, del inglés *Aerodynamic Shape Optimization*) es una forma efectiva de automatizar los procesos de diseño. Apoyándose en la dinámica de fluidos computacional (CFD, del inglés *Computational Fluid Dynamics*) y en recursos computacionales de alta eficiencia (HPC, del inglés *high-performance computing*), las técnicas de ASO permiten diseñar colas, alas o góndolas del motor [28]. Las técnicas de DL aplicadas a la dinámica de fluidos se emplean para el cálculo de parámetros como, por ejemplo, el coeficiente de resistencia (C_D) o la relación entre sustentación y avance o para el cálculo directo de los campos de flujo. Las aplicaciones de DL necesitan de una gran cantidad de datos, que, en el ámbito de la dinámica de fluidos, se extraen de los experimentos y simulaciones numéricas.

En este TFM, se aplica una arquitectura concreta de red neuronal convolucional (CNN, del inglés *Convolutional Neural Networks*) a una serie de situaciones fluido dinámicas de flujo turbulento. Se analiza la capacidad de las redes neuronales de predecir los campos de velocidades paralela al flujo y vertical y de presión, cuando se enfrentan a diferentes velocidades de entrada del fluido o a geometrías variables en tipo, tamaño y orientación.

El SW empleado consiste en una CPU Intel Xeon Gold 5120 para la obtención de todas las simulaciones CFD, una GPU NVIDIA Quadro RTX 6000 para el entrenamiento de la red neuronal, una CPU Intel Core i7-10750H para la generación de los diferentes códigos auxiliares. Los programas empleados son el Star-CCM+ para la generación de las simulaciones CFD y Python 3.9.6 y Matlab 2020b para el desarrollo de SW. Python 3.9.6 se ha ejecutado en el entorno Anaconda.

El CFD requiere de una capacidad computacional muy elevada. Para ciertas aplicaciones, se precisan resultados rápidos para la extracción de las primeras conclusiones

respecto a un problema determinado. Este TFM se focaliza en la aplicación de diferentes técnicas de DL que ahondan en la obtención de resultados rápidos lo más precisos posibles para evaluar los posibles caminos a seguir a la hora de resolver un problema concreto.

2 Alcance y objetivos

Cada simulación fluido dinámica puede presentar unas características muy diferentes respecto al resto. En un análisis lo más básico posible, se puede seleccionar el tipo de fluido y sus valores de velocidad, temperatura, si es un flujo laminar o turbulento... También la geometría a la que se enfrenta el fluido, modificando, por ejemplo, su forma, tamaño, posición u orientación. Todo ello unido a los elevados recursos computacionales que consumen las simulaciones CFD, implica la necesidad de búsqueda de nuevas técnicas que efectúen las simulaciones fluido dinámicas.

Debido a esta problemática, diferentes técnicas de DL se han postulado como alternativas eficientes a las simulaciones fluido dinámicas clásicas. Son muchas y variadas las aplicaciones en las que el DL y en concreto, las redes neuronales, han conseguido resultados con un coste computacional muy reducido y errores, con respecto a las simulaciones clásicas, lo suficientemente pequeños como para considerar al DL como una alternativa eficiente, para ciertas aplicaciones, a las simulaciones CFD.

Este TFM busca aplicar una arquitectura concreta de una CNN para resolver problemas fluido dinámicos con diferentes condiciones del fluido y las geometrías analizadas. En concreto, se buscan soluciones a los siguientes planteamientos:

En primer lugar, se busca predecir un número concreto de instantes futuros partiendo de un estado inicial. Este problema se enfoca en dos situaciones diferentes. En la primera, se consideran velocidades variables de entrada al dominio evaluado. La interacción del fluido se considera siempre sobre una geometría circular, invariable en tamaño y posición. Por tanto, el primer objetivo reside en analizar la capacidad de una CNN para predecir instantes futuros a partir de uno inicial, para velocidades del fluido de entrada al dominio variables. En la segunda situación, el objetivo reside en evaluar si esa misma CNN es capaz de predecir los instantes futuros para un conjunto de geometrías (círculo, elipse, rectángulo, cuadrado, triángulo y triángulo equilátero). Asimismo, se analiza el comportamiento para tamaños, orientaciones y posiciones variables de las geometrías mencionadas.

Las aplicaciones mencionadas en el párrafo anterior dependen en todo momento de un estado inicial a partir del cual se predicen los instantes futuros. Para solucionar este inconveniente, se emplea nuevamente la CNN diseñada para predecir el instante inicial necesario para las otras aplicaciones.

Estas tres aplicaciones persiguen el objetivo global de obtener resultados relativamente precisos que involucren un consumo reducido de recursos computacionales. Para

lograr este objetivo global, se tienen que alcanzar objetivos intermedios, expuestos a continuación.

- El primer objetivo reside en la generación de las simulaciones CFD, necesarias para el entrenamiento de las redes neuronales.
- Posteriormente, se ha de manejar correctamente los datos de las simulaciones CFD para que sirvan de salidas de la CNN. Asimismo, se han de seleccionar el resto de entradas y salidas necesarias.
- El siguiente objetivo consiste en la selección de la arquitectura de la CNN y los valores de los parámetros e hiper-parámetros de la red más convenientes.
- Por último, se han de manejar los resultados obtenidos para extraer las conclusiones pertinentes.

3 Antecedentes bibliográficos o estado del arte

3.1 Introducción a la Inteligencia Artificial y el *Machine Learning*

La Inteligencia Artificial (IA) tiene múltiples enfoques respecto a su definición y a lo que realmente representa. Existen definiciones que hacen referencia a los procesos mentales y al razonamiento y otras, se centran en la conducta. Asimismo, algunas de estas definiciones miden la capacidad de una máquina de igualar el comportamiento humano, mientras que otras se basan en la racionalidad, es decir, la facultad de obrar de forma correcta, según el conocimiento de la máquina [42]. Tomando sistemas que piensan como humanos, aparecen las siguientes definiciones:

- "El nuevo y excitante esfuerzo de hacer que los computadores piensen... máquinas con mentes, en el más sentido literal" [18].
- "[La automatización de] actividades que vinculamos con procesos de pensamiento humano, actividades como la toma de decisions, resolución de problemas, aprendizaje..." [7].

Respecto al enfoque de sistemas que piensan racionalmente, dos definiciones de referencia son las siguientes:

- "El estudio de las facultades mentales mediante el uso de modelos computacionales" [8].
- "El estudio de los cálculos que hacen possible percibir, razonar y actuar" [51].

El tercer grupo se compone de definiciones que referencian a sistemas que actúan como humanos:

- "El arte de desarrollar máquinas con capacidad para realizar funciones que cuando son realizadas por personas requieren de inteligencia" [25].
- "El estudio de como lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor" [40].

Por último, el grupo de definiciones que se centran en la actuación racional de los sistemas:

- “La Inteligencia Computacional es el studio del diseño de agentes inteligentes” [37].
- “IA... está relacionada con conductas inteligentes en artefactos” [35].

Para Flach [11], el Aprendizaje Automático (ML, del inglés *Machine Learning*) es el estudio sistemático de los algoritmos y sistemas que mejoran su conocimiento o su desempeño con la experiencia. El ML se engloba dentro del campo de la IA. Torres [48] expresa de forma generalizada que el ML consiste en desarrollar para cada problema un algoritmo de predicción para un caso de uso particular.

Una red neuronal artificial (ANN, del inglés *Artificial Neural Network*) consiste en una capa de entrada de neuronas, una o más capas ocultas de neuronas, y una capa final de salida [49]. Cuando la red presenta una única capa oculta, se les denomina ANN superficiales, como la del diagrama de la figura 3.1, mientras que las ANN profundas tienen un gran número de capas ocultas. Es en este segundo caso donde entra en juego el concepto de DL, que es un subgrupo del ML, como muestra el diagrama de la figura 3.2.

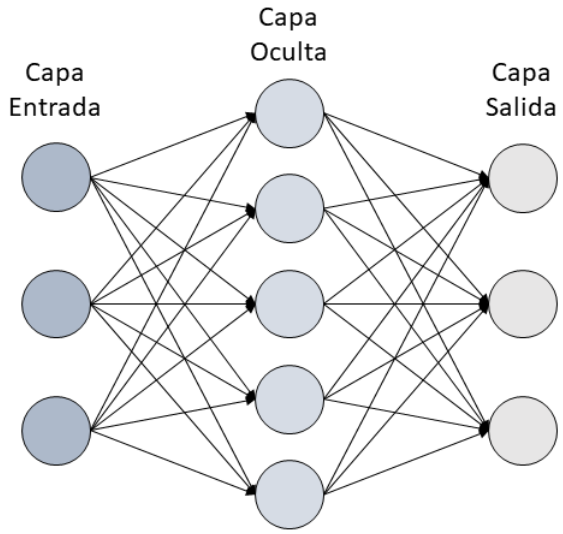


Figura 3.1: Diagrama que representa una ANN simple, que contiene las capas de entrada, oculta y de salida. Si tuviese varias capas ocultas sería una red de DL.

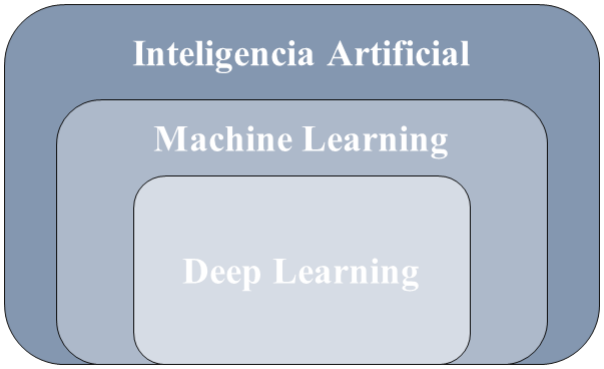


Figura 3.2: Diagrama que muestra que el DL se engloba dentro del ML y, que a su vez, este último, es un subconjunto de IA.

LeCun et al. [26] definen el DL como la herramienta que permite a los modelos computacionales compuestos de múltiples capas de procesamiento aprender representaciones de los datos con múltiples niveles de abstracción. El DL consigue descubrir estructuras complejas en grandes conjuntos de datos utilizando el algoritmo de retropropagación (del inglés *backprogation*) para indicar cómo una máquina debería cambiar sus parámetros internos usados para el cálculo de la representación en cada capa desde la representación de la capa anterior. El concepto de DL junto con los modelos computacionales y los algoritmos que incorpora trata de imitar la arquitectura de las redes neuronales biológicas del cerebro. Cuando el cerebro recive nueva información, la compara con los conocimientos previamente existentes e intenta darle sentido a este nuevo estímulo. El cerebro descifra la información asignando los elementos a varias categorías [20]. El organigrama de la figura 3.3 muestra de forma concisa cómo se organizan los diferentes pasos que se toman en las diversas disciplinas de IA. En el caso de DL, se aprecia la búsqueda de características simples en un primer lugar, para luego, ceñirse a las particularidades de mayor nivel de abstacción. Por último, se lleva a cabo el mapeo de esas características para obtener la salida deseada. A diferencia del DL, cuando se trata de una ANN simple, únicamente se buscan las características en una sola parte y, en el ML, la computadora simplemente se dedica al mapeo de las características.

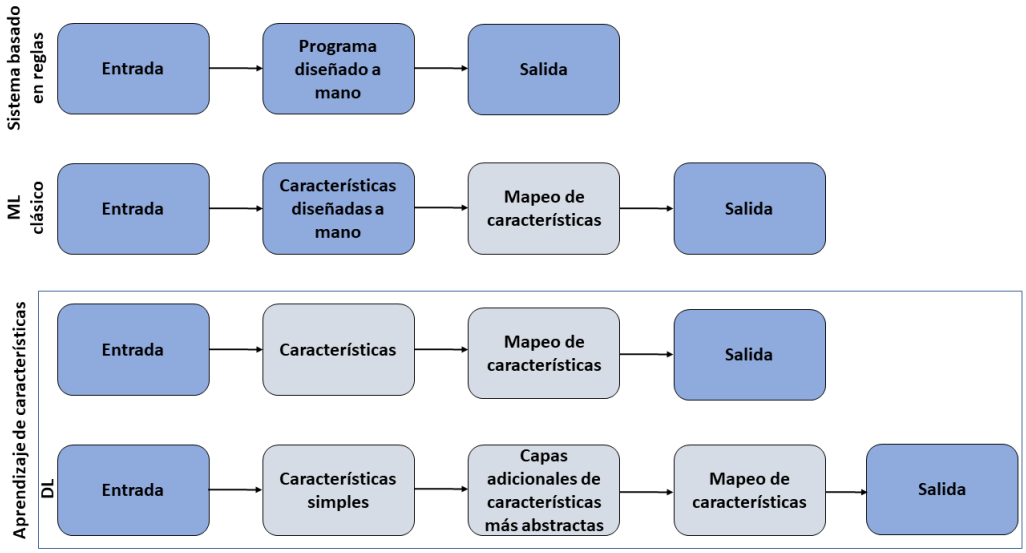


Figura 3.3: Organigrama que muestra cómo las diferentes partes de un sistema de IA se relacionan entre ellas en las diferentes disciplinas de la IA. Las cajas en gris muestran los componentes que son capaces de aprender de los datos.

Las neuronas en DL se pueden definir como los nodos por los que fluyen los datos. El funcionamiento de las neuronas es el siguiente:

- Reciben las señales de una o más entradas, provenientes de los datos de entrada de las salidas de las neuronas anteriores.
- En segundo lugar, realizan una serie de cálculos según la ecuación 3.1.
- Por último, expulsan el resultado a la salida.

$$y = \phi\left(\sum_{j=1}^{n} x_j \omega_j\right), \tag{3.1}$$

donde y es la salida de la neurona, ϕ representa el resultado de la función de activación, n es el número de entradas totales, x_j representa cada una de las entradas de la neurona y ω_j, cada uno de los pesos sinápticos. Para j=0, x_0 es igual a 1 y ω_0 es el sesgo (también se simboliza como b). La 3.4 muestra un diagrama de una neurona artificial.

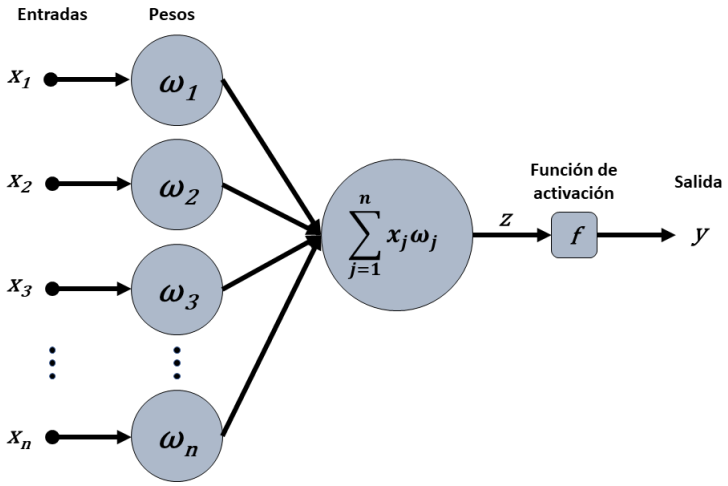


Figura 3.4: Diagrama de una neurona con sus respectivas entradas, pesos sinápticos, función de activación y salida.

3.1.1 Parámetros e hiper-parámetros

En los modelos de ML existen dos tipos de parámetros: los propiamente denominados parámetros, que se inicializan y actualizan durante el proceso de aprendizaje de la red neuronal; y los hiper-parámetros, que han de ser inicializados previamente al inicio del entrenamiento del modelo [24].

Dentro del primer grupo, el llamado parámetros, se encuentran los pesos sinápticos y el sesgo. Los pesos son números reales que indican la pendiente de la recta. El sesgo representa cómo de lejos están las predicciones con respecto del valor de salida deseado.

Los hiper-parámetros, según Yang y Shami [53], se pueden clasificar en dos categorías: aquellos que están relacionados con la construcción de un modelo de DL, llamada hiper-parámetros de diseño de modelo; y los hiper-parámetros de optimización, que se encargan de la optimización y el proceso de entrenamiento del modelo de DL.

Dentro del grupo de hiper-parámetros de diseño de modelo, aparecen algunos como el número de capas ocultas, el número de neuronas por capa, la función de coste, la función de activación y el tipo de optimizador. Como se ha mencionado previamente, las redes neuronales presentan como mínimo tres capas, donde dos de ellas son las de entrada y salida. El concepto de profundidad de una red neuronal depende de dos hiperparámetros, que son el número de capas ocultas y el número de neuronas por capa. Estos dos hiperparámetros dependen de la complejidad y tamaño de los conjuntos de datos del modelo. El tipo de función de coste empleada es un hiperparámetro que adquiere gran relevancia. Algunos ejemplos de función de coste son la entropía cruzada binaria (del inglés *binary cross-entropy*) para los

problemas de clasificación; la entropía cruzada multicategoría (del inglés *multi-class cross-entropy*) para los problemas de clasificación multivariable, y el error medio absoluto (MAE, del inglés *Mean Absolute Error*), también llamado *L1 loss*, el error medio cuadrático (MSE, del inglés *Mean Squared Error*), también *L2 loss*, y la raíz del error cuadrático medio (RMSE, del inglés *Root Mean Square Error*) para los problemas de regresión. Las expresiones de las funciones de coste vienen dadas por las expresiones 3.2, 3.3 y 3.4 respectivamente.

$$MAE = \frac{1}{m}\sum_{i=1}^{m}\left|h(x^{(i)}) - y^{(i)}\right|, \tag{3.2}$$

donde m es el número de muestras, $x^{(i)}$ es la muestra i del conjunto de datos, $h(x^{(i)})$ es la predicción para la muestra i e $y^{(i)}$ es la etiqueta de la muestra i.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}\left|y^{(i)} - \hat{y}^{(i)}\right|, \tag{3.3}$$

donde m es el número de muestras, $y^{(i)}$ es la etiqueta de la muestra i e $\hat{y}^{(i)}$ es la etiqueta predecida para la muestra i.

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(x^{(i)} - y^{(i)}\right)}, \tag{3.4}$$

donde m es el número de muestras, $x^{(i)}$ es la muestra i del conjunto de datos, $h(x^{(i)})$ es la predicción para la muestra i e $y^{(i)}$ es la etiqueta de la muestra i.

El hiperparámetro función de activación se emplea para la propagación de la salida de una neurona hacia delante. Algunos ejemplos de funciones de activación son la función lineal, la sigmoide, la tangente hiperbólica (tanh), la softmax, la unidad lineal rectificada (ReLU, del inglés *rectified linear unit*) o la *softsign*. Las expresiones de estas funciones vienen dadas por las ecuaciones 3.5, 3.6, 3.7, 3.8, 3.9 y 3.10 respectivamente.

$$f(x) = x, \tag{3.5}$$

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{3.6}$$

$$f(x) = tanh(x), \tag{3.7}$$

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \ \ para\ i = 1, ..., n, \tag{3.8}$$

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0, \end{cases} \tag{3.9}$$

$$f(x) = \frac{x}{1 + |x|}. \tag{3.10}$$

Por último, el tipo de optimizador a emplear. La optimización hace referencia a la tarea de minimizar o maximizar una función f(x) alterando el valor de x. La función f(x) a minimizar o maximizar es la previamente explicada función de coste [14]. Algunos ejemplos típicos de optimizadores son el gradiente de descenso estocástico (SGD, del inglés *stochastic gradient descent*), la estimación de momento adaptativo (ADAM, del inglés *adaptative moment estimation*), la propagación de la raíz cuadrática media (RMSprop, del inglés *root mean square propagation*).

Dentro del segundo grupo, el más importante es la tasa de aprendizaje (del inglés *learning rate*), que determina el tamaño de paso en cada iteración para permitir que la función de coste converja. Si la tasa de aprendizaje es muy grande, el proceso de aprendizaje se acelera, pero puede provocar que el gradiente oscile sobre un mínimo local o que no llegue a converger; mientras que una tasa de aprendizaje muy pequeña provoca un incremento del tiempo de aprendizaje. El *dropout* es una técnica que previene el sobreajuste (del inglés *overfitting*). Se encarga de eliminar algunas neuronas y conexiones entre neuronas de forma temporal de una red neuronal [45]. El *mini-batch size* es un hiperparámetro que representa el número de muestras procesadas previas a la actualización del modelo y el número de épocas indica el número de veces que se itera sobre el conjunto de datos completos. Un último hiperparámetro sería el *early stopping*, que sirve para finalizar el entrenamiento del modelo cuando el error de validación no cambia durante una serie de épocas consecutivas.

3.1.2 Tipos y características de ANN

En este apartado, se describen algunas de las estructuras de DL más básicas, en concreto las relacionadas con la aplicación del DL a la dinámica de fluidos. La principal diferencia entre ellas radica en la arquitectura de la red, es decir, la forma en la que las neuronas se organizan dentro de la red. Según la arquitectura, la siguiente lista muestra los tres tipos principales:

- Perceptrón multicapa (MLP, del inglés *Multi-Layer Perceptron*): es un sistema de neuronas simples interconectadas. Se trata de una representación del modelo de mapeo no lineal de un vector de entradas en uno de salidas [12]. Las capas están densamente conectadas entre sí, es decir, cada nodo está conectado con

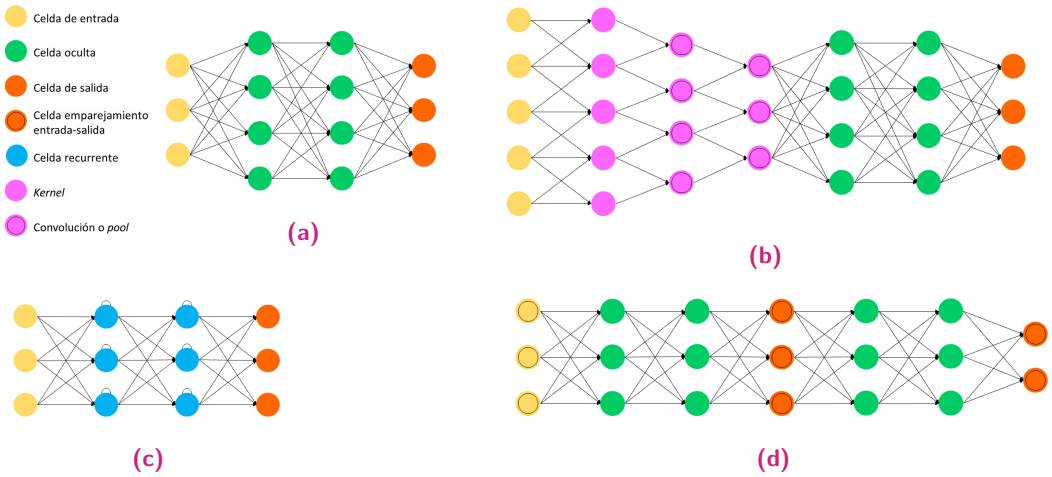


Figura 3.5: Diagramas de las arquitecturas de los principales tipos de redes neuronales: a) MLP, b) CNN, c) RNN y d) GAN.

todas las neuronas de la capa anterior y posterior. La figura 3.5a muestra un diagrama de un MLP con dos capas ocultas.

- Las CNN son un tipo de ANN que están compuestas como mínimo por una capa convolucional. Suelen llevar a cabo operaciones de convolución, *pooling*, tener capas densamente conectadas (del inglés *fully-connected*). Su ventaja principal con respecto a las ANN tradicionales reside en la disminución del número de parámetros necesarios [3] La figura 3.5b muestra un ejemplo de una arquitectura de red de tipo CNN.

- Las redes neuronales recurrentes (RNN, del inglés *Recurrent Neural Networks*) presentan un estado interno creado con los datos de entrada ya empleados por la red. Posteriormente, como salida, proporciona una combinación entre el estado interno y la nueva entrada. A su vez, el estado interno se modifica en cada iteración para los nuevos datos entrantes. Se emplean en tareas que requieren datos secuenciales [43]. En la figura 3.5c se muestra un esquema de una RNN.

- Las red neuronales generativas adversarias (GAN, del inglés *Generative Adversarial Networks*) están formadas por dos modelos: un modelo generativo, G, que captura la distribución de los datos, y un modelo discriminativo D que estima la probabilidad de que una muestra provenga del entrenamiento de los datos antes que de G. El procedimiento de entrenamiento de G es maximizar la probabilidad de D de fallar [15]. En la figura 3.5d se visualiza un ejmeplo de una estructura de tipo GAN.

Las capas principales de una CNN son las capas de entrada y salida y las capas de convolución y *pooling*.

La operación de convolución consiste en detectar y aprender determinados patrones locales en ventanas de dos dimensiones dentro de una imagen. Una vez reconocido un patrón específico en una imagen, se puede identificar esa característica en cualquier parte de la misma. Asimismo, una capa convolucional inicial es capaz de aprender patrones sencillos como aristas, colores o líneas. Posteriormente, otra capa convolucional se vale de esos patrones para aprender jerarquías de patrones. De esta forma, haciendo uso de varias capas de convolución, la red neuronal llega a comprender patrones muy complejos. La figura 3.6 muestra un diagrama que representa la capa de convolución.

Las capas de *pooling* se emplean comúnmente tras la serie de capas convolucionales. A rasgos generales, la operación de *pooling* realiza una simplificación de la información recogida por la capa convolucional y crea una versión condensada de la misma. Para ello, esta capa divide en pequeñas regiones de igual tamaño la capa convolucional logrando reducir el número de conexiones para las siguientes capas. No realiza ningún aprendizaje por sí misma, sino que reduce el número de parámetros a aprender durante las siguientes capas. Como se muestra en la figura 3.7, existen diferentes tipos de *pooling*. Los más típicos son el *max-pooling*, en el que se toma el mayor valor de la región, y el *average-pooling*, donde se toma el valor medio de la región.

Tanto la capa de convolución como la de *pooling* tienen su respectiva operación inversa. Para las capas de convolución, existen las de convolución traspuesta, también denominadas deconvoluciones. Para las capas de *pooling* se encuentran las capas de *unpooling*.

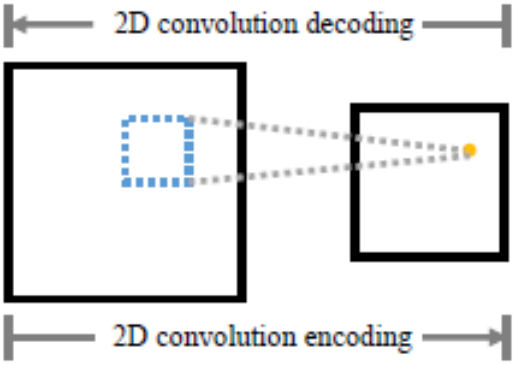


Figura 3.6: Diagrama de una convolución [16].

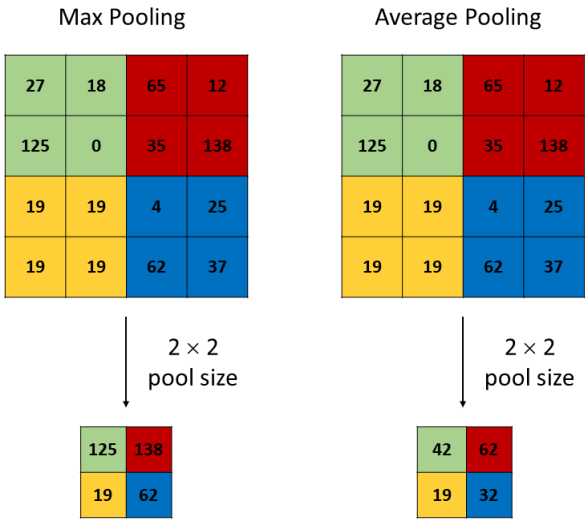


Figura 3.7: Ejemplos de operaciones de *pooling*.

3.2 Dinámica de fluidos Computacional

Las características físicas de cualquier fluido siguen tres leyes fundamentales: la conservación de la masa, F=m·a (segunda ley de Newton) y la conservación de la energía. Estos principios fundamentales se pueden expresar en forma de ecuaciones matemáticas, generalmente a través de ecuaciones diferenciales. El CFD se basa en sustituir soluciones analíticas de ecuaciones en forma cerrada por números. El producto final del CFD es una gran cantidad de números que proporcionan una descripción cuantitativa del problema.

Las soluciones de las simulaciones en CFD requieren generalmente la manipulación reiterada de miles, o incluso millones, de números. Debido a ello, la capacidad computacional necesaria y la demanda de memoria es muy elevada. A pesar de los avances exponenciales que se llevan produciendo durante las últimas décadas en la informática, que habilitan la resolución de problemas con CFD cada vez más detallados y sofisticados, sigue suponiendo una limitación para el desarrollo de productos en un rango variado de aplicaciones, como pueden ser la optimización del diseño aerodinámico y la interacción fluido-estructura [50]. Este hecho, unido al crecimiento de la inteligencia artificial en los últimos años, ha resultado en que muchos autores empleen técnicas de DL para obtener aproximaciones de los resultados de las simulaciones CFD, en los casos en los que se necesitan geometrías muy complejas o mallados muy finos. Otro inconveniente de las aplicaciones CFD es su clara dependencia de la capacidad del usuario para la generación del mallado y el modelo de turbulencia.

Para resolver estos problemas, el DL ha sido un área de estudio enfocada a las simulaciones CFD. Por ejemplo, Tao y Sun [46], Zhang et al. [54] y Yan et al.

[52] alcanzaron la optimización aerodinámica, mejorando la eficiencia de varias geometrías con DL. Dentro de las técnicas de DL aplicadas en el ámbito de la computación de fluidos, destacan dos enfoques diferentes. El primero tiene como objetivo la reducción tiempo de computación de las simulaciones en las que se emplea mallas gruesas. Por ejemplo, Bao et al. [5] aplicaron un enfoque en las características físicas del sistema para mejorar el modelado y la capacidad de simulación de una malla gruesa, y Hanna et al. [17] diseñaron un algoritmo de DL para predecir y mejorar el error de los resultados obtenidos en una malla gruesa.

El segundo enfoque implica el cálculo directo de las características deseadas del fluido. Guo et al [16] crearon una CNN que predice los campos de flujo estacionarios alrededor de objetos sólidos, logrando resultados no tan precisos, pero con predicciones muy rápidas. Ribeiro et al [39], tomando la CNN de Guo et al [16] como referencia, diseñaron una CNN muy precisa para predecir los campos de velocidad y presión de fluidos estacionarios alrededor obstáculos con formas simples, reduciendo el coste computacional en torno a 3 y 5 órdenes de magnitud con respecto a las simulaciones CFD. Kashefi et al [21] diseñaron una ANN para obtener los mismos campos que los estudios mencionados previamente. Los campos se obtuvieron con ligeras modificaciones de las geometrías, ya que este es un parámetro esencial a la hora de optimizar los diseños.

Asimismo, existen otros estudios donde se llevan a cabo predicciones de características del flujo más específicas. Por ejemplo, Ling et al [29] utilizó un enfoque de DL para el modelado de las turbulencias a través de las ecuaciones de Navier-Stokes promediadas por Reynolds (RANS, del inglés Reynolds-Averaged Navier-Stokes). De esta forma, modeló el tensor de anisotropía de Reynolds a través de una red neuronal profunda (DNN, del inglés Deep Neural Network), consiguiendo una mejora importante comparada con los resultados obtenidos en las simulaciones CFD. Lee and You [27] predijeron el desprendimiento de vórtices laminares no estacionarios en un cilindro circular mediante una GAN. Liu et al. [30] y Deng et al. [9], diseñaron métodos para la detección de impactos y vórtices respectivamente mediante técnicas de DL.

Los estudios anteriores están centrados en la predicción de flujos laminares. Este tipo de flujo es más fácil de predecir para una DNN debido a que las partículas del fluido se desplazan de forma paralela en láminas ordenadas mientras que, en los flujos turbulentos, las partículas presentan un movimiento aleatorio y caótico. La arquitectura más habitual en los estudios que contemplan los flujos turbulentos son las CNN. Especialmente con una estructura tipo *autoencoder*, en la que las entradas a la red se reducen a una representación geométrica latente (LGR, del inglés *Latent Geometry Representation*). El LGR se trata de una representación de las características básicas de las entradas iniciales a la red. De esta forma, el *encoder* de la CNN puede mapear de forma más sencilla las características del fluido deseadas. Asimismo, la

estructura U-net creada por Ronneberger et al. [41], en la que se aplica la estructura *autoencoder* conectando cada bloque codificador a su respectivo decodificador. Estos autores emplean dicha estructura para la segmentación de imágenes biomédicas. No obstante, los estudios comentados a continuación demuestran la flexibilidad de aplicación de esta arquitectura a la dinámica de fluidos para flujos turbulentos. Por ejemplo, Fang et al. [10] predijeron el tensor de esfuerzos de Reynolds para flujo turbulento con una DNN, mejorando los resultados obtenidos con el modelo lineal y con el modelo de viscosidad de Eddy cuadrática. Thuerey et al. [47] utiliza una CNN con arquitectura U-net para aproximar los campos de velocidad y presión del modelo de turbulencia Spalart-Allmaras basado en las ecuaciones RANS para un perfil aerodinámico. Abucide-Armas et al. [1] logra tasas de error reducidas en la predicción de los campos de velocidad y presión, a través de una CNN, para fluidos turbulentos y velocidades de entrada al dominio variables. Además, añade una técnica de data-augmentation novedosa aplicando la semejanza dinámica y Portal-Porras et al. [38] desarrolla varias estructuras basadas en CNN para predecir los campos de velocidad para flujos turbulentos.

A pesar de la existencia de algunos estudios que analizan dominios tridimensionales, como los estudios Guo et al. [16] y el de Nowruzi et al. [36], la gran mayoría de estudios de este tipo se centran en geometrías bidimensionales. Esto se debe principalmente a los limitados recursos computacionales existentes para las geometrías 3D [34]. Para evitar este problema, Mohan et al. [34] desarrollaron una infraestructura basada en DL que reduce la geometría para analizar posteriormente las características del fluido.

En la mayoría de los estudios mencionados previamente no se tiene en cuenta la evolución temporal de las características del fluido. En DL, se emplean RNN para los casos en los que las entradas y salidas de la red dependen de instantes anteriores. Agostini [2] predice la evolución temporal de la velocidad paralela al flujo con un modelo *autoencoder*. Por otro lado, Maulik et al. [33], Gonzalez y Balajewicz [13] y King et al. [22] predijeron algunas propiedades del fluido con un enfoque basado en la evolución temporal de las mismas.

En este trabajo, se utiliza una CNN basada en la arquitectura U-net con una estructura de tipo *autoencoder*. La CNN propuesta es entrenada con los datos de los campos de velocidad horizontal y vertical y de presión obtenidos con técnicas de CFD. El objetivo es aprovechar estos datos para entrenar la CNN para conseguir predicciones de los instantes futuros con tasas de error pequeñas, teniendo en cuenta la dependencia del estado actual de las características de un fluido con respecto al estado anterior. En cuanto al testeo de la red, se toma un instante inicial aleatorio de los datos de CFD a partir del cual se calculan las predicciones de los siguientes 10 instantes. Cada resultado obtenido es comparado con su correspondiente instante calculado con CFD para obtener la precisión de la red.

4 Desarrollo de la solución

El trabajo se focaliza en la utilización de técnicas de DL para simulaciones fluido dinámicas, principalmente para flujos turbulentos. En primer lugar, se estudia la capacidad de una CNN de predecir instantes futuros de los campos de velocidades paralela al flujo y vertical y de presión para velocidades de entrada al dominio computacional variables. El segundo caso se enfoca en la predicción de dichos campos variando las geometrías analizadas y sus características. Asimismo, se genera un modelo neuronal para la predicción del primer instante a partir del cual se generan los instantes futuros.

4.1 Geometría circular con velocidad de entrada variable

4.1.1 Simulaciones numéricas

En este trabajo, se estudian las oscilaciones periódicas a largo plazo en un flujo turbulento. Por ello, las simulaciones se llevan a cabo en base a las ecuaciones inestables de Navier-Stokes promediadas por Reynolds (URANS, del inglés *Unsteady Reynolds-Averaged Navier-Stokes*). Las ecuaciones URANS se obtienen mediante el siguiente proceso:

Las ecuaciones de Navier-Stokes para un flujo incompresible se filtran a lo largo del tiempo de acuerdo a la ecuación 4.1.

$$\frac{\partial \langle u_i \rangle}{\partial t} + \frac{\partial}{\partial x_j}(\langle u_j u_i \rangle) = -\frac{1}{\rho}\frac{\partial \langle p \rangle}{\partial x_i} + \upsilon \frac{\partial^2 \langle u_i \rangle}{\partial x_k^2}. \tag{4.1}$$

Después, se introduce en la ecuación 4.1 el tensor de esfuerzos turbulentos, dado por la ecuación 4.2.

$$\tau_{ij} = \langle u_i \rangle \langle u_j \rangle - \langle u_j u_i \rangle . \tag{4.2}$$

La ecuación URANS final queda:

$$\frac{\partial \langle u_i \rangle}{\partial t} + \frac{\partial}{\partial x_j}(\langle u_i \rangle \langle u_j \rangle) = -\frac{1}{\rho}\frac{\partial \langle \tau_{ij} \rangle}{\partial x_i} + \frac{\partial \langle p \rangle}{\partial x_j} + \upsilon \frac{\partial^2 \langle u_i \rangle}{\partial x_k^2}. \tag{4.3}$$

Explicaciones más detalladas sobre las ecuaciones URANS se proporcionan, por ejemplo, en el estudio de Iaccarino et al. [19].

Star-CCM+ [44] ha sido empleado para la generación de las simulaciones CFD. Cada una de las simulaciones es de un 1 segundo con una frecuencia de muestreo de $2{\cdot}10^{-4}$. Esto proporciona un total de 5000 muestras por cada simulación. Para cada muestra, se han calculado los campos de las velocidades paralela al flujo y vertical y el campo de presión. Estos datos se guardan en archivos de tipo CSV, donde cada uno de ellos contiene los valores de presión y velocidades horizontal y vertical para cada punto del dominio dado por las coordenadas X e Y. Los resultados de las simulaciones CFD son los empleados para el entrenamiento, validación y testeo de la CNN. El dominio considerado es un rectángulo bidimensional con un cilindro circular en el medio del dominio, ver Aramendia et al. [4]. El flujo avanza desde la entrada al dominio, situada en la parte izquierda del mismo, hasta la salida, situada en el extremo derecho del dominio. Los extremos superior e inferior y el cilindro circular no tienen asignadas condiciones de deslizamiento. El diámetro del círculo (D) es de 10 mm, y su centro se encuentra a una distancia 5D de la entrada al dominio y de las paredes deslizantes. El tamaño del dominio es un rectángulo de dimensiones iguales a 100×256 mm. La vista detallada del dominio computacional y sus dimensiones se proporciona en la figura 4.1.

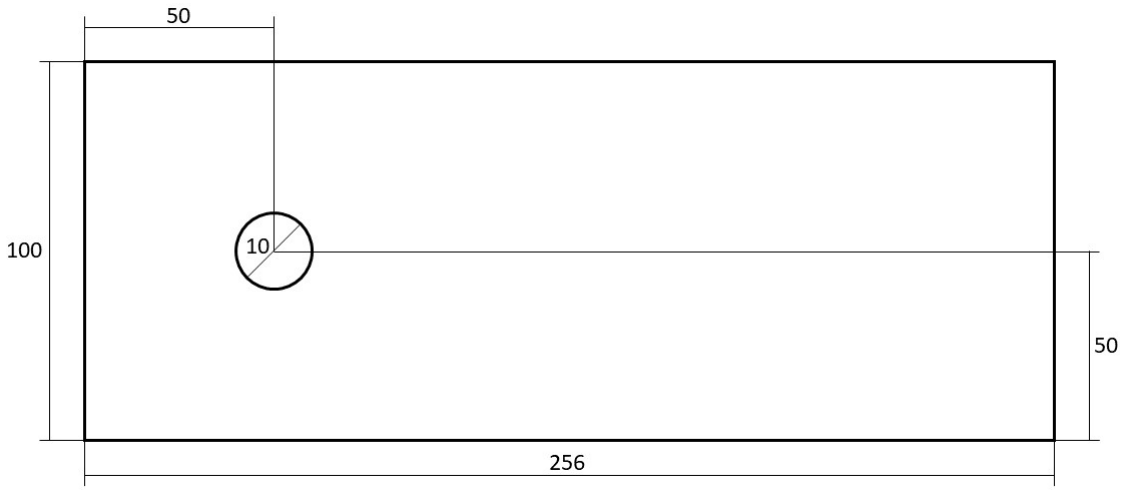


Figura 4.1: Dominio computacional de las simulaciones con velocidad de entrada variable (sin escalar).

Dentro del dominio descrito en el párrafo anterior, se ha construido un mallado compuesto de celdas poliédricas. La mayoría de celdas se ubican en la zona posterior al obstáculo en el sentido de movimiento del flujo, buscando una mejor captura de los vórtices situados detrás del círculo. Asimismo, un control volumétrico ha sido diseñado para refinar la malla alrededor del obstáculo para poder mantener el valor y+ por debajo de 1. La figura 4.2 muestra el mallado descrito.

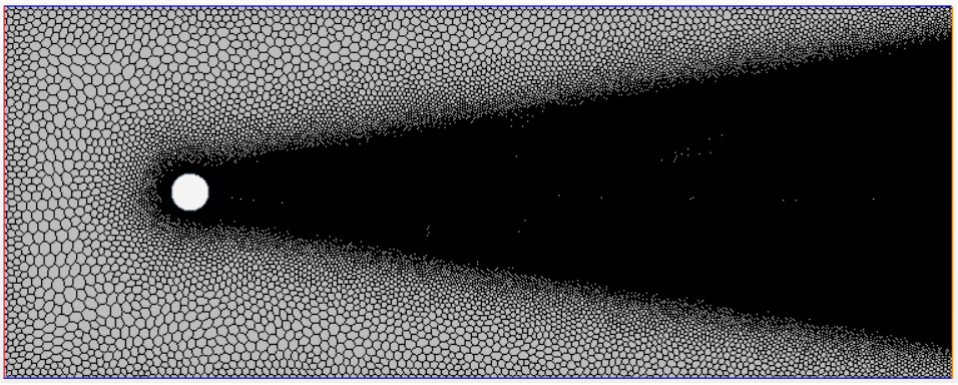


Figura 4.2: Distribución del mallado alrededor de la geometría circular.

Con respecto al fluido, se considera un flujo de aire incompresible turbulento inestable. La densidad (ρ) del fluido elegido es 1,18415 kg/m^3 y su viscosidad dinámica (μ) es igual a 1,85508·10^{-5} Pa·s. Ambas magnitudes se asumen como constantes. La velocidad en la entrada (u_∞) varía entre 5 m/s y 25 m/s, con un intervalo de 5 m/s entre cada simulación. Dando un total de 5 simulaciones. El número de Reynolds de las simulaciones varía entre 3200 y 16000, dependiendo del caso y de acuerdo a la ecuación 4.4.

$$Re = u_\infty D/\nu, \tag{4.4}$$

donde u_∞ indica la velocidad del fluido, D el diámetro del círculo y ν, la viscosidad cinemática del fluido. Para que el tamaño de los datos de CFD encajen con el tamaño de las entradas de la CNN, se interpolan en una red de 79×172.

4.1.2 Red neuronal convolucional

La red neuronal escogida para este estudio es una CNN. Estas redes, como se ha descrito previamente son muy eficaces en la obtención de patrones de imágenes a nivel de píxel a partir de unos determinados datos de entrada. Las imágenes digitales son, en esencia, matrices. Los campos de velocidad y presión también son matrices; por tanto, la ventaja de encontrar patrones es la que permite la predicción de los campos analizados.

En este caso, el problema ha sido analizado con un enfoque basado en el tiempo. Este enfoque se justifica en que, en la dinámica de fluidos, el estado de un fluido en un instante de tiempo, t, es fuertemente dependiente de su estado anterior, t-1. Es decir, el transitorio de los estados de un fluido adquiere una gran importancia a la hora de analizar las características del mismo. Este concepto se implementa en la CNN mediante la adición de una nueva entrada a la red neuronal que contenga el estado del fluido analizado en el instante t-1.

La arquitectura considerada es una U-Net propuesta por Ronneberger et al. [41] para la segmentación de imágenes médicas. Ribeiro et al. [39] demostraron que este tipo

de arquitectura es perfectamente aplicable a la predicción de los campos de velocidad y presión de un fluido. La ventaja de la U-Net reside en que reproduce resultados de mayor precisión gracias a las conexiones directas entre las características de la geometría codificada y las capas decodificadoras. Esta arquitectura consiste en una serie de capas de convolución que llevan a cabo una compresión de la información de las geometrías para obtener una versión condensada de los datos, provocando que la CNN sea capaz de detectar los patrones destacables existentes en el conjunto de datos con una mayor sencillez. Esta versión reducida de las geometrías recibe el nombre de LGR. Posteriormente, por medio de capas de deconvolución se realiza el mapeo entre la LGR y las variables de interés, en este caso, u_x, u_y y p. Estas operaciones de deconvolución transcurren hasta que el LGR adquiere el tamaño del CFD original. Además, el número de canales de salida es igual al número de variables de interés. En el caso en estudio, a pesar de existir tres variables de interés, únicamente existe un canal de salida debido a que cada una de las tres variables se estudia por separado. La adición de un enfoque temporal provoca que un análisis de las tres variables de forma simultánea sea erróneo, ya que se mezclarían informaciones de las variables de interés entre ellas.

La arquitectura de la red presenta una estructura de tipo *autoencoder*, puesto que las entradas de la red se codifican hasta proporcionar el LGR que, posteriormente, se decodifica hasta obtener las salidas deseadas. La figura 4.3 representa de manera gráfica la arquitectura U-Net con una estructura de tipo *autoencoder*.

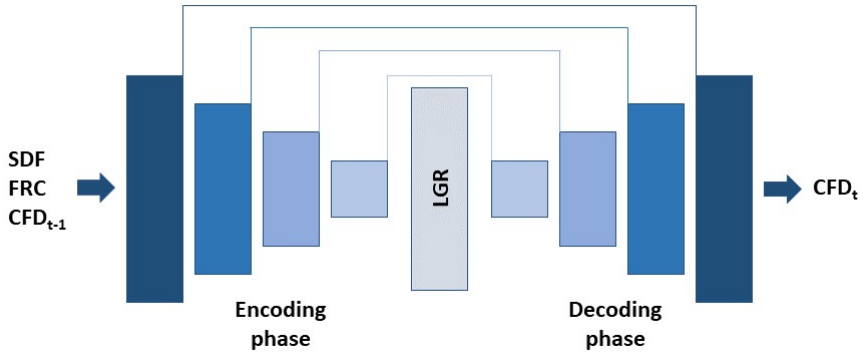


Figura 4.3: Arquitectura U-Net.

A continuación, se describe en profundidad la arquitectura del *autoencoder*. La parte de codificación está constituida por 4 bloques codificadores, donde cada uno de ellos está compuesto por 3 capas de convolución. Los parámetros de las capas de convolución son los siguientes:

- Filtros: 8, 16, 32 y 32 para cada bloque codificador respectivamente.

- Tamaños de las ventanas: 3×3, 5×5 o 7×7, dependiendo de cada uno de los modelos entrenados.

Cada bloque codificador tiene el siguiente orden en sus capas:

- En primer lugar, una capa de convolución con la entrada igual al número de filtros del anterior bloque y la salida igual al número de filtros actual más una capa ReLU. En el primer bloque codificador, la primera subcapa contiene 3 filtros en su entrada, que corresponde con el número de entradas a la red neuronal (la función de distancia con signo (SDF, del inglés *Signed Distance Function*), el canal de la región de flujo (FRC, del inglés *Flow Region Channel*) y CFD_{t-1}).

- Posteriormente, una operación de convolución con igual número de filtros en la entrada y la salida más una capa ReLU.

- Por último, se lleva a cabo el mismo proceso que el anterior punto añadiendo, tras la capa ReLU , una capa de *maxpooling*.

La parte decodificadora está formada por 4 bloques decodificadores formados por sucesivas capas de deconvolución, *maxunpooling* y ReLU para mapear la LGR en la salida deseada en cada caso. Las características de las capas de deconvolución son las mismas que las de convolución de la parte codificadora a excepción de los filtros que van en orden inverso para obtener el mapeo de la LGR de forma correcta. El orden de las capas en cada uno de los decodificadores es el siguiente:

- Operación de deconvolución con la entrada igual al doble de filtros correspondiente y la salida igual al número de filtros correspondiente más una capa ReLU.

- Una capa de deconvolución con entrada y salida igual al número de filtros más una capa ReLU.

- Una capa de maxunpooling más una capa de deconvolución con entrada igual al número de filtros correspondiente y salida igual al número de filtros del bloque siguente más una capa ReLU. En el ultimo bloque decodificador, la capa de *maxunpooling* no existe y la deconvolución tiene una única salida, la variables u_x, u_y o p dependiendo del caso a evaluar.

La figura 4.4 representa la arquitectura de la U-Net de forma detallada.

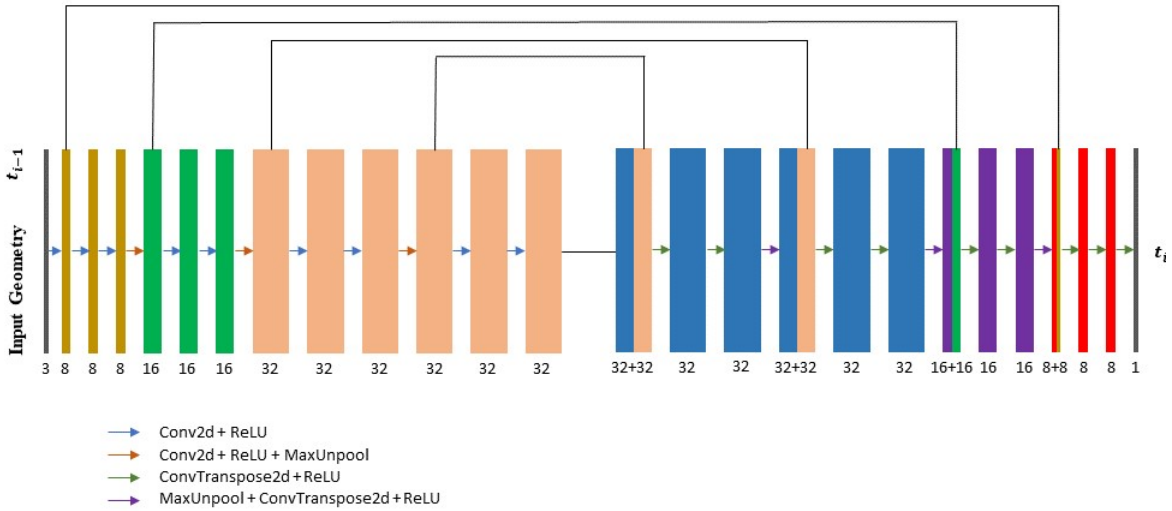


Figura 4.4: Arquitectura detallada de la CNN.

4.1.3 Entradas de la red neuronal

Para esta red neuronal se consideran 3 entradas. El primer par corresponde con la SDF y el FRC. La entrada SDF hace referencia a la forma geométrica tanto del obstáculo como de la malla mientras que en el FRC se indican las características del fluido a lo largo de las diferentes posiciones de la malla. La entrada restante representa la muestra de la variable analizada en el instante anterior (CFD_{t-1}).

La capa de entrada FRC es un canal de múltiples categorías que presenta información sobre las condiciones de contorno del dominio. Se trata de una matriz donde, a través de 5 categorías diferentes, se indican las zonas por las que puede transcurrir un fluido. Las categorías son las siguientes:

- 0 para el obstáculo.
- 1 para la región de flujo libre.
- 2 para las paredes superior e inferior que cumplen la hipótesis de no deslizamiento.
- 3 para la condición de velocidad constante de entrada.
- 4 para la condición límite de flujo cero.

Figura 4.5: Diagrama representativo del FRC.

La entrada SDF consiste en una función matemática que mide la distancia relativa entre cualquier punto de la rejilla y cualquier punto del contorno de una figura geométrica cerrada. Según Guo et al. [16], la SDF proporciona una representación universal para diferentes figuras geométricas y funciona eficientemente con las redes neuronales. Otro tipo de representaciones de las geometrías como los límites o los parámetros geométricos no son efectivas para las redes neuronales. La expresión matemática de esta función viene dada por:

$$SDF(x) = \begin{cases} d(x, \theta\Omega)\ if\ x\ \epsilon\ \Omega \\ -d(x, \theta\Omega)\ if\ x\ \epsilon\ \Omega^c, \end{cases} \tag{4.5}$$

donde Ω es un subconjunto de un espacio métrico, X, con métrica, d, y $\vartheta\Omega$ es el límite de Ω. Para cada x ϵ X:

$$d(x, \theta\Omega) := \underset{y\epsilon\theta\Omega}{inf}\ d(x, y), \tag{4.6}$$

donde inf es el ínfimo. A las posiciones interiores del obstáculo (Ω^c) se le asignan valores de distancia negativos. La figura 4.6 muestra la SFD generada para este estudio con un código de MATLAB [32]. Este permite elegir la posición y tamaño del círculo y el tamaño de la rejilla.

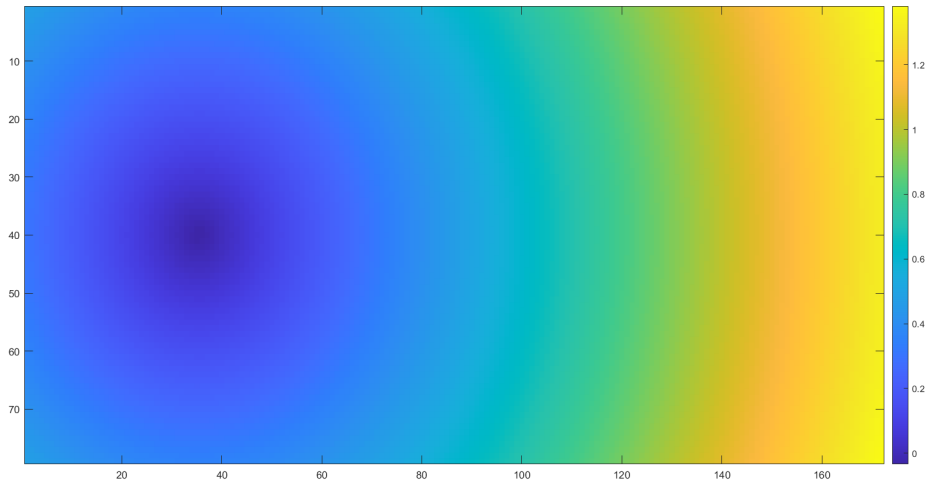


Figura 4.6: SDF de un círculo.

Durante el entrenamiento de la CNN, los resultados obtenidos de las simulaciones CFD se utilizan como la tercera entrada. La muestra del instante t-1 con respecto a la muestra de CFD analizada en ese momento es el introducido como entrada. Esto permite a la CNN aprender los patrones existentes durante los transitorios de un estado a otro. A la hora del testeo de la CNN, el resultado predicho en el instante anterior (t-1) se emplea como referencia para la nueva predicción; es decir, en el testeo las entradas son la SDF, el FRC y la predicción en t-1.

4.1.4 Parámetros del entrenamiento

El optimizador escogido para el entrenamiento de la red es el AdamW, dado por el algoritmo 1. Es un algoritmo basado en el algoritmo Adam, que actualiza el vector de gradiente y el gradiente cuadrático utilizando la media móvil exponencial [23]. Los coeficientes β_1 y β_2 son los factores de olvido (*forgetting factors*) para los gradientes y los segundos momentos de los gradientes, respectivamente. El valor proporcionado a ambos factores de olvido es de 0,5. El optimizador AdamW es una versión actualizada del optimizador Adam, que mejora la regularización desacoplando el decaimiento de los pesos de la actualización basada en gradiente [31].

Algorithm 1 Algoritmo AdamW

Entradas: γ(lr), $\beta_1, beta_2(betas)$, θ_0(params), $f(\theta)$(objetivo), $\epsilon\lambda$(*weight decay*), *amsgrad*, *maximize*

Inicializar: $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment)

for t=*1* ***to*** **... do**

  if *maximize:*

    $g_t \leftarrow - \bigtriangledown_\theta f_t(\theta_{t-1})$

  else

    $g_t \leftarrow \bigtriangledown_\theta f_t(\theta_{t-1})\ \theta_t \leftarrow \theta t - 1 - \gamma\lambda\theta_{t-1}$

  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$

  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$

  if *amsgrad*

    $\widehat{v_t}^{max} \leftarrow max(\widehat{v_t}^{max}, \widehat{v_t})$

    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}^{max}} + \epsilon)$

  else

    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

end

return θ_t

Los valores de los hiper-parámetros empleados en el entrenamiento de los diferentes modelos se muestran en la tabla 4.1.

Parámetro	**Valores**
Arquitectura	[8, 16, 32, 32]
Tamaño de las ventanas	3 5 7
Función de coste	L1-norm
Tasa de aprendizaje (lr)	0,001 0,0001
Decaimiento de pesos	0,005
Batch size	32 64
Ratio entrenamiento-test	0,7-0,3
Nº de épocas	1000

Tabla 4.1: Conjunto de valores seleccionados para la búsqueda de los hiper-parámetros adecuados para la red que predice los instantes futuros para velocidades de entrada al dominio variables.

4.2 Geometrías variables

4.2.1 Simulaciones numéricas para geometrías variables

Nuevamente se emplea Star-CCM+ [44] para simular los campos de velocidades paralela al flujo y vertical y de presión. Se consideran las siguientes geometrías: círculo, elipse, cuadrado, rectángulo, triángulo y triángulo equilátero. La duración de cada una de las simulaciones es de 0,1 segundos con una frecuencia de muestreo de $2 \cdot 10^{-4}$. Se generan 500 muestras para cada geometría, proporcionando un total de 3000 muestras. En cada muestra se calculan los valores de velocidad y presión para cada punto del dominio (X, Y). Estos valores se almacenan en un archivo con formato CSV. El dominio considerado es un rectángulo bidimensional, donde la geometría y las paredes se consideran como superficies no deslizantes. El tamaño del dominio es de 128×256 mm. La figura 4.7 representa el dominio computacional y sus dimensiones.

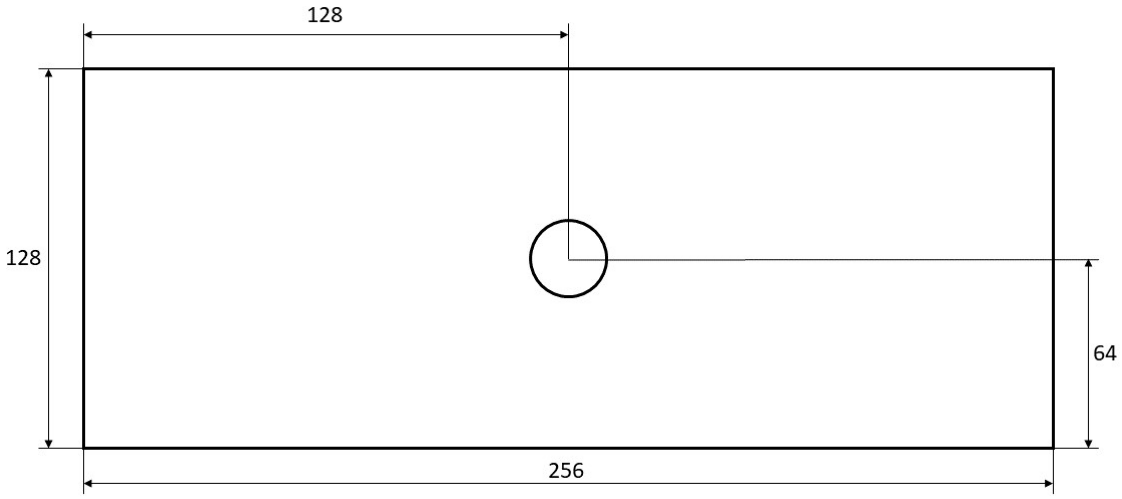


Figura 4.7: Dominio computacional de la simulaciones con geometrías variables (sin escalar).

Se ha construido un mallado compuesto de celdas poliédricas dentro del dominio descrito. La mayoría de celdas se ubican alrededor de la geometría y en las paredes del dominio. Asimismo, se ha sido diseñado un control volumétrico para refinar la malla alrededor del obstáculo para poder mantener el valor y+ por debajo de 1. La figura 4.8 muestra el mallado descrito.

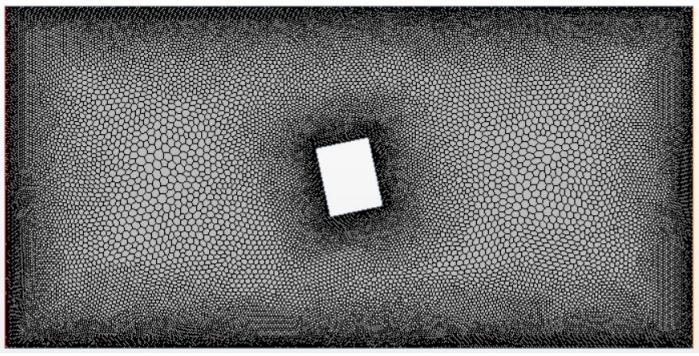


Figura 4.8: Ejemplo de la distribución del mallado alrededor de una geometría.

El fluido considerado es un flujo de aire incompresible turbulento inestable con una densidad (ρ) de 1,18415 kg/m^3 y una viscosidad dinámica (μ) igual a 1,85508·10^{-5} Pa·s. Ambas magnitudes se asumen como constantes. La velocidad de entrada es de 5 m/s. Para que el tamaño de los datos de CFD encajen con el tamaño de las entradas de la CNN, se interpolan en una red de 79×172.

4.2.2 Red neuronal convolucional

La arquitectura U-Net empleada en 4.1.2 se toma como referencia para los entrenamientos de esta sección. Aquí se analizan dos variantes de esta arquitectura. En primer lugar, se reutiliza el enfoque temporal para predecir los estados futuros de los campos de velocidades y presión. Posteriormente, se focaliza en la predicción de un instante determinado de la terna de campos a partir de las entradas de la red, que representan las características geométricas del obstáculo y el dominio computacional.

La primera variante corresponde con la representada en la figura 4.3. Se presentan como entradas las matrices SDF y FRC y la muestra t-1 del campo analizado. Un único decodificador mapea la información en la LGR, que se decodifica para obtener el campo estudiado en el instante t.

La segunda variante se diferencia de la primera en que, a través de tres decodificadores, se obtienen los tres campos analizados. Asimismo, las entradas de la red son únicamente las matrices SDF y FRC. La figura 4.9 representa gráficamente el *autoencoder* con tres decodificadores. Los entrenamientos con esta segunda variante se realizan para una estructura *autoencoder* de 4 bloques codficadores/decodificadores y para otra de 5. La figura 4.10 representa de forma detallada la estructura U-Net con una configuración de 4 bloques codificadores/decodificadores y 3 decodificadores.

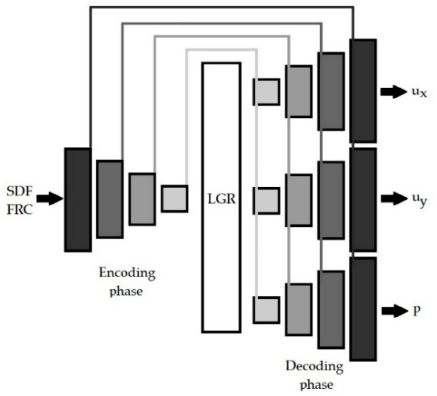


Figura 4.9: Arquitectura U-Net con 3 decodificadores.

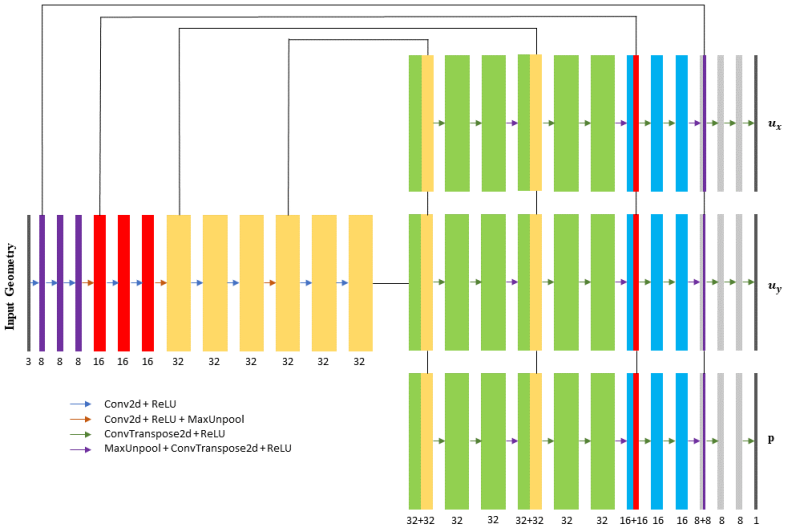


Figura 4.10: Arquitectura detallada de la CNN con 3 decodificadores.

4.2.3 Parámetros del entrenamiento

El optimizador seleccionado es el AdamW, dado por el algoritmo 1. Ambos factores de olvido se inicializan en 0,5. Para buscar la mejor combinación de los hiperparámetros para la red que predice las muestras futuras y la que predice la muestra inicial, se entrenan las redes con todas las combinaciones posibles de los valores dados por las tablas 4.2 y 4.3 respectivamente.

Parámetro	Valores
Arquitectura	[8, 16, 32, 32]
Tamaño de las ventanas	3 5 7
Función de coste	L1-norm
Tasa de aprendizaje (lr)	0,001 0,0001
Decaimiento de pesos	0,005
Batch size	32 64 128
Ratio entrenamiento-test	0,7-0,3
Nº de épocas	2000

Tabla 4.2: Conjunto de valores seleccionados para la búsqueda de los hiper-párametros adecuados para la red que predice los instantes futuros de los campos con geometrías variables.

Parámetro	Valores
Arquitectura	[8, 16, 32, 32] [8, 16, 16, 32, 32]
Tamaño de las ventanas	3 5 7 9
Función de coste	L1-norm
Tasa de aprendizaje (lr)	0,001 0,0001
Decaimiento de pesos	0,005
Batch size	32 64 128
Ratio entrenamiento-test	0,7-0,3
Nº de épocas	1000 2000

Tabla 4.3: Conjunto de valores seleccionados para la búsqueda de los hiper-párametros adecuados para la red que predice la muestra inicial.

4.2.4 *Data augmentation*

El entrenamiento de la red para predecir la primera muestra de los campos de velocidades y presión necesita de una gran cantidad de datos. No obstante, tomando un instante determinado, únicamente se tienen seis muestras, una por cada geometría. Por ello, se opta por aplicar una técnica de *data augmentation* que permita incrementar dicha cantidad de datos para el entrenamiento.

Las técnicas de *data augmentation* se emplean asiduamente en aplicaciones de DL. Consisten en la generación de datos sintéticos realistas con el objetivo de aumentar la cantidad de datos para el proceso de aprendizaje de una red neuronal. La técnica más utilizada consiste en añadir transformaciones geométricas y perturbaciones a los datos reales. Sin embargo, este sencillo procedimiento no puede ser aplicado en este estudio. Como solución a este inconveniente, se aplica la técnica de *data augmentation* expuesta en el estudio de Abucide et al. [1].

Esta técnica de *data augmentation* se aplica teniendo en cuenta la teoría de la semejanza de la dinámica de fluidos. Para cualquier simulación, si el número de Reynolds se mantiene constante en cada caso, la nueva velocidad de entrada al dominio y las nuevas velocidades y presiones de los campos pueden ser calculadas. El número de Reynolds se obtiene a partir de la 4.4.

Asumiendo que el número de Reynolds es constante en cada uno de los casos, la nueva velocidad de entrada para cada caso se puede calcular mediante la ecuación 4.7. Esto es debido a que el fluido y las condiciones de contorno se mantienen constantes y a que la densidad y la viscosidad dinámica no ejercen influencia ninguna sobre la velocidad. Como consecuencia, aplicando ligeras modificaciones al tamaño de las geometrías, la cantidad de datos de entrada para la CNN se incrementa considerablemente.

$$u^*_{\infty i} = \frac{D1}{Di} u_{\infty 1} \tag{4.7}$$

Las expresiones de los nuevos campos de velocidades y presión vienen dadas por las ecuaciones 4.8, 4.9 y 4.10.

$$\hat{u}_{xi}(\hat{x}, \hat{y}) = \frac{u_x i}{u^*_{\infty i}} \left(\frac{x_i}{D_i}, \frac{y_i}{D_i} \right) \tag{4.8}$$

$$\hat{u}_{yi}(\hat{x}, \hat{y}) = \frac{u_y i}{u^*_{\infty i}} \left(\frac{x_i}{D_i}, \frac{y_i}{D_i} \right) \tag{4.9}$$

$$\hat{p}_i(\hat{x}, \hat{y}) = \frac{p_x i}{u^{*2}_{\infty i} \rho} \left(\frac{x_i}{D_i}, \frac{y_i}{D_i} \right) \tag{4.10}$$

donde $\hat{u}_{xi}$, $\hat{u}_{yi}$ y $\hat{p}_i$ representan los nuevos campos de velocidades y presión y $\frac{x_i}{D_i}$ y $\frac{y_i}{D_i}$, las nuevas coordenadas dentro del dominio. Como establece la teoría de la semejanza, el tamaño del dominio cambia proporcionalmente a la modificación del tamaño de la geometría. Esto se muestra en la ecuación 4.11, donde se muestra la equivalencia para dos puntos concretos del mallado.

$$\hat{x} = \frac{x_1}{D_1} = \frac{x_2}{D_2} \tag{4.11}$$

Con esta información y mediante un código que interpola los campos originales, la cantidad de datos se incrementa fácilmente. La ventaja de este método de *data augmentation* reside en que evita la generación de nuevas simulaciones CFD de larga duración y elevados recursos computacionales. Se han tomado las primera

muestra de los campos de velocidades paralela al flujo y vertical y de presión simulados mediante CFD para cada una de las seis geometrías. Después, se han multiplicado las dimensiones de las geometrías por cincuenta valores diferentes de una variable llamada factor de tamaño, que varían entre 0,93 y 1,07. A continuación, se ha aplicado el procedimiento explicado en este apartado y se han obtenido 300 muestras de CFD para el entrenamiento.

5 Análisis de resultados

5.1 Resultados de la red aplicada a una geometría circular con velocidad de entrada variable

La red neuronal ha sido entrenada para las distintas combinaciones de los valores de los hiper-parámetros dados por la tabla 4.1. En la tabla 5.1 se muestran todos los modelos neuronales entrenados con los valores de los hiper-parámetros y la duración de los entrenamientos para cada variable. Asimismo, en la figuras 5.1 y 5.2 se muestran las gráficas que comparan los errores medios y máximos obtenidos en los testeos de cada modelo. Por cada variable se genera un modelo neuronal individual. Esto permite escoger tres modelos independientes según que combinación de los hiper-parámetros haya proporcionado mejores resultados. Cada modelo neuronal ha sido entrenado con los mismos datos para el entrenamiento y para el test. En este caso, los modelos seleccionados son los de ID igual a 1, 9 y 8 para la velocidad paralela al flujo, la velocidad vertical y la presión, respectivamente. La siguiente configuración de los filtros ha sido empleada para todos los entrenamientos: [8, 16, 32, 32].

Los mejores modelos para la predicción de las muestras futuras de la velocidad paralela al flujo son el 1 y el 6. Pese a que el modelo 6 presenta menor error medio, el error máximo es elevado. Por tanto, el primer modelo es el seleccionado. En el caso de la velocidad vertical, los mejores modelos son el 9 y el 10. Se escoge el modelo 9 por proporcionar un error máximo menor que las predicciones del décimo modelo. En el caso de la presión, los modelos 7, 8 y 9 son los que proporcionan los mejores resultados. Se escoge el modelo con ID igual a 8, debido a presentar menor error medio que el modelo 7 y menor error máximo que el modelo 9.

ID	Tamaño ventana	Lr	Batch size	N.º épocas	Duración entrenamiento v_x (h)	Duración entrenamiento v_y (h)	Duración entrenamiento p (h)
1	3	0.001	32	1000	6,73	6,61	6,61
2	5	0.001	32	1000	7,21	7,09	6,95
3	7	0.001	32	1000	9,47	9,30	9,32
4	3	0.001	64	1000	5,56	5,55	5,51
5	5	0.001	64	1000	6,40	6,18	6,29

ID	Tamaño ventana	Lr	Batch size	N.º épocas	Duración entrenamiento v_x (h)	Duración entrenamiento v_y (h)	Duración entrenamiento p (h)
6	7	0.001	64	1000	8,97	8,69	8,74
7	3	0.0001	32	1000	7,41	6,97	6,98
8	5	0.0001	32	1000	9,59	7,26	7,26
9	7	0.0001	32	1000	7,10	9,41	9,44
10	3	0.0001	64	1000	7,40	6,98	6,93
11	5	0.0001	64	1000	9,60	7,30	7,27
12	7	0.0001	64	1000	7,11	9,43	9,43

Tabla 5.1: Identificador de cada modelo, combinación de los hiper-parámetros utilizados en cada uno de los modelos entrenados y duración de los entrenamientos de cada variable.

La figuras 5.3, 5.4 y 5.5 muestran la comparativa gráfica de los resultados obtenidos mediante las predicciones de la CNN y de las simulaciones CFD. Asimismo, las figuras 5.6, 5.7 y 5.8 muestran los histogramas en los que se comparan la distribución de los datos de las predicciones de la CNN y los valores de CFD. La tabla 5.2 muestra la comparativa de la media aritmética y la desviación estándar para los valores de las simulaciones CFD y la predicciones de la CNN, para las tres variables analizadas. Puesto que las predicciones dependen la predicción de la muestra inicial, en las predicciones más cercanas al instante 50 se incrementan considerablemente los errores absolutos. Específicamente, hasta la muestra 20 los resultados son bastantes precisos. Después, aparece un incremento de los errores absolutos, especialmente en la velocidad paralela al flujo y en los casos con velocidades de entrada al dominio elevadas. Las velocidades de entrada superiores influyen en que los cambios de los valores de los campos entre dos muestras consecutivas sean mayores. Esto dificulta que la CNN prediga los valores de los instantes posteriores, provocando un incremento del error absoluto.

Velocidad entrada (m/s)	Método	CFD			CNN		
		v_x (m/s)	v_y (m/s)	p (Pa)	v_x (m/s)	v_y m/s	p (Pa)
5	μ	4,9951	0,0075	2.1723	5,4124	-0,0285	2,8495
	σ	1,2170	1,1167	4,3208	1,3490	1,1234	5,1211
10	μ	9,9913	-0,0013	5,2600	10,1065	-0,0376	6,5888
	σ	2,3040	1,8506	14,1083	2,2233	1,8575	14,5295
15	μ	14,9948	0,0190	23,3207	14,7106	-0,0316	28,1998
	σ	3,0165	3,6250	33,4826	2,9791	3,5796	38,5308

20	μ	19,9890	0,0665	44,3182	18,8420	-0,1063	49,2844
	σ	4,6631	4,7345	64,6778	4,6450	4,4262	69,8038
25	μ	24,9808	0,0353	71,2020	22,2070	-0,2575	57,7787
	σ	6,3783	5,8095	105,2168	6,2542	5,4596	85,2597

Tabla 5.2: Media aritmética y desviación estándar de las simulaciones CFD y las predicciones de la CNN por cada velocidad de entrada al dominio.

5.2 Resultados de la red que predice los instantes futuros aplicada a geometrías variables

La tabla 4.2 indica los valores de los hiper-parámetros que se han seleccionado para el entrenamiento de los modelos neuronales. Mediante combinaciones de los valores escogidos se han generado una serie de modelos neuronales. Después, se comprueba los valores de error medio y máximo en cada caso. En las tablas 5.3, 5.4 y 5.5 se muestran los mejores modelos ordenados según el mínimo error medio conseguido para cada una de las tres variables. En este caso, se pueden seleccionar diferentes combinaciones de los hiper-parámetros para el entrenamiento de cada variable, puesto que se generan tres modelos independientes. Todos los entrenamientos presentan 2000 épocas con la arquitectura [8, 16, 32, 32]. El criterio seguido para escoger el modelo adecuado se basa en los errores máximos y mínimos proporcionados por los testeos de cada modelo neuronal. Además, las muestras para entrenamiento y test han sido las mismas para cada modelo neuronal.

En los casos de la velocidad vertical y la presión, el error medio y el error máximo mínimos corresponden los modelos neuronales con ID igual a 5 y 12 respectivamente. Sin embargo, para la velocidad paralela al flujo, los dos primeros modelos neuronales con el menor error medio, proporcionan errores máximos demasiados elevados. Los modelos neuronales con ID igual a 12 y 15 son relativamente similares, proporcionando cada uno menor error medio y mayor error máximo y viceversa. El modelo neuronal con ID 15 es el elegido, por ser el de menor error máximo.

ID	Tamaño ventana	Lr	Batch size	Duración entrenamiento (h)	Error medio v_x (m/s)	Error máx. v_x (m/s)
10	3	0,0001	32	1,70	0,1413	33,4453
13	3	0,0001	64	1,35	0,1561	43,7074
12	7	0,0001	32	2,83	0,1715	15,6595
15	7	0,0001	64	2,60	0,2053	14,8938
4	3	0,001	64	1,37	0,2188	2705,84
11	5	0,0001	32	1,67	0,2217	21,416

ID	Tamaño ventana	Lr	Batch size	Duración entrenamiento (h)	Error medio v_x (m/s)	Error máx. v_x (m/s)
3	7	0,001	32	2,74	0,2510	73,7285
14	5	0,0001	64	1,51	0,2847	48,7618
6	7	0,001	64	2,59	0,3729	1848,65
5	5	0,001	64	1,50	0,5728	39398,8

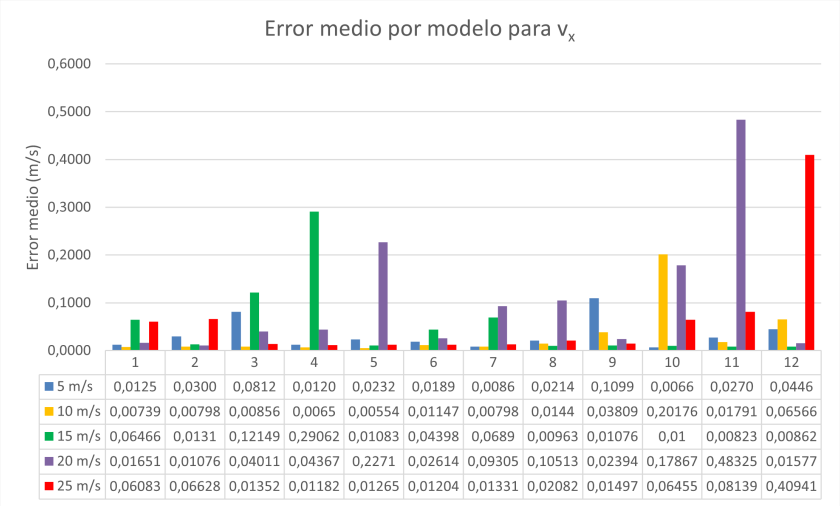
Tabla 5.3: 10 mejores entrenamientos para predicción de los instantes futuros del campo de velocidad paralela al flujo.

ID	Tamaño ventana	Lr	Batch size	Duración entrenamiento (h)	Error medio v_y (m/s)	Error máx. v_y (m/s)
5	5	0,001	64	1,45	0,0585	9,7446
15	7	0,0001	64	2,55	0,0651	10,5586
12	7	0,0001	32	2,72	0,0651	32,4277
2	5	0,001	32	1,56	0,0674	21,7514
14	5	0,0001	64	1,46	0,0976	18,6669
7	3	0,001	128	1,40	0,0992	35,5557
9	7	0,001	128	2,69	0,0999	15,4869
1	3	0,001	32	1,59	0,1016	802,375
8	5	0,001	128	1,49	0,1094	90,4429
16	3	0,0001	128	1,41	0,1358	34,6491

Tabla 5.4: 10 mejores entrenamientos para la predicción de los instantes futuros del campo de velocidad vertical.

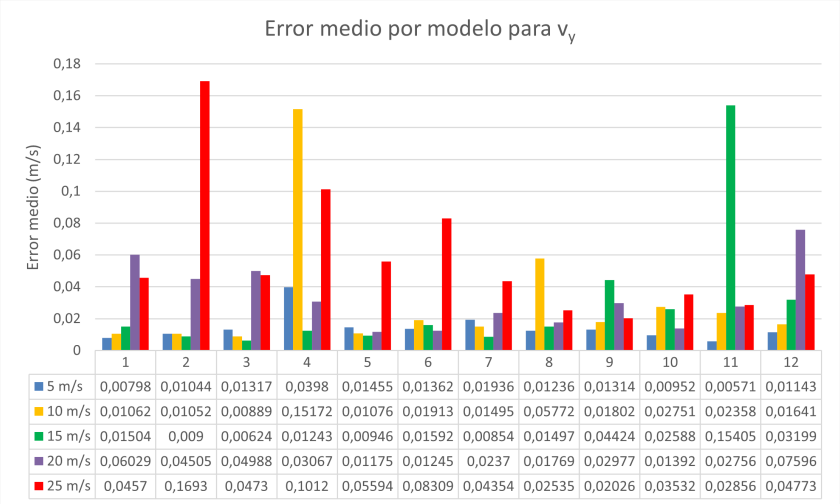
ID	Tamaño ventana	Lr	Batch size	Duración entrenamiento (h)	Error medio p (Pa)	Error máx. p (Pa)
12	7	0,0001	32	2,72	0,7139	112,09
4	3	0,001	64	1,24	0,8828	132,73
11	5	0,0001	32	1,57	0,9542	119,92
1	3	0,001	32	1,58	0,9748	466,44
14	5	0,0001	64	1,46	0,9933	117,70
5	5	0,001	64	1,44	1,0869	356,00
15	7	0,0001	64	2,54	1,0900	212,32
7	3	0,001	128	1,40	1,1218	549,21
18	7	0,0001	128	2,71	1,1887	108,50
2	5	0,001	32	1,55	1,3195	196,38

Tabla 5.5: 10 mejores entrenamientos para predicción de los instantes futuros del campo de presión.



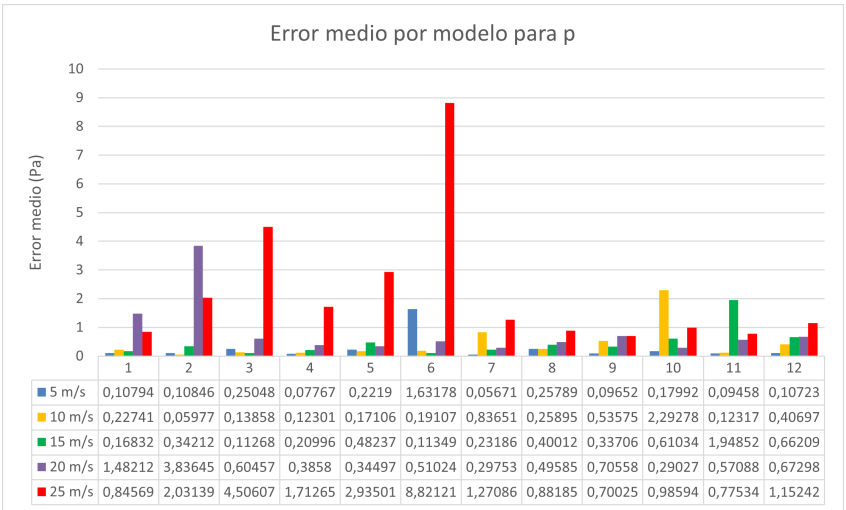
	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	0,0125	0,0300	0,0812	0,0120	0,0232	0,0189	0,0086	0,0214	0,1099	0,0066	0,0270	0,0446
10 m/s	0,00739	0,00798	0,00856	0,0065	0,00554	0,01147	0,00798	0,0144	0,03809	0,20176	0,01791	0,06566
15 m/s	0,06466	0,0131	0,12149	0,29062	0,01083	0,04398	0,0689	0,00963	0,01076	0,01	0,00823	0,00862
20 m/s	0,01651	0,01076	0,04011	0,04367	0,2271	0,02614	0,09305	0,10513	0,02394	0,17867	0,48325	0,01577
25 m/s	0,06083	0,06628	0,01352	0,01182	0,01265	0,01204	0,01331	0,02082	0,01497	0,06455	0,08139	0,40941

(a)



	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	0,00798	0,01044	0,01317	0,0398	0,01455	0,01362	0,01936	0,01236	0,01314	0,00952	0,00571	0,01143
10 m/s	0,01062	0,01052	0,00889	0,15172	0,01076	0,01913	0,01495	0,05772	0,01802	0,02751	0,02358	0,01641
15 m/s	0,01504	0,009	0,00624	0,01243	0,00946	0,01592	0,00854	0,01497	0,04424	0,02588	0,15405	0,03199
20 m/s	0,06029	0,04505	0,04988	0,03067	0,01175	0,01245	0,0237	0,01769	0,02977	0,01392	0,02756	0,07596
25 m/s	0,0457	0,1693	0,0473	0,1012	0,05594	0,08309	0,04354	0,02535	0,02026	0,03532	0,02856	0,04773

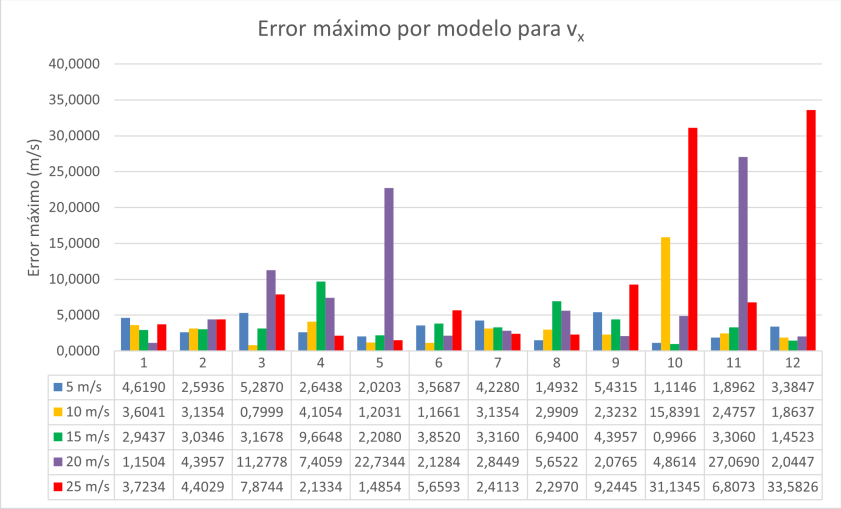
(b)



	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	0,10794	0,10846	0,25048	0,07767	0,2219	1,63178	0,05671	0,25789	0,09652	0,17992	0,09458	0,10723
10 m/s	0,22741	0,05977	0,13858	0,12301	0,17106	0,19107	0,83651	0,25895	0,53575	2,29278	0,12317	0,40697
15 m/s	0,16832	0,34212	0,11268	0,20996	0,48237	0,11349	0,23186	0,40012	0,33706	0,61034	1,94852	0,66209
20 m/s	1,48212	3,83645	0,60457	0,3858	0,34497	0,51024	0,29753	0,49585	0,70558	0,29027	0,57088	0,67298
25 m/s	0,84569	2,03139	4,50607	1,71265	2,93501	8,82121	1,27086	0,88185	0,70025	0,98594	0,77534	1,15242

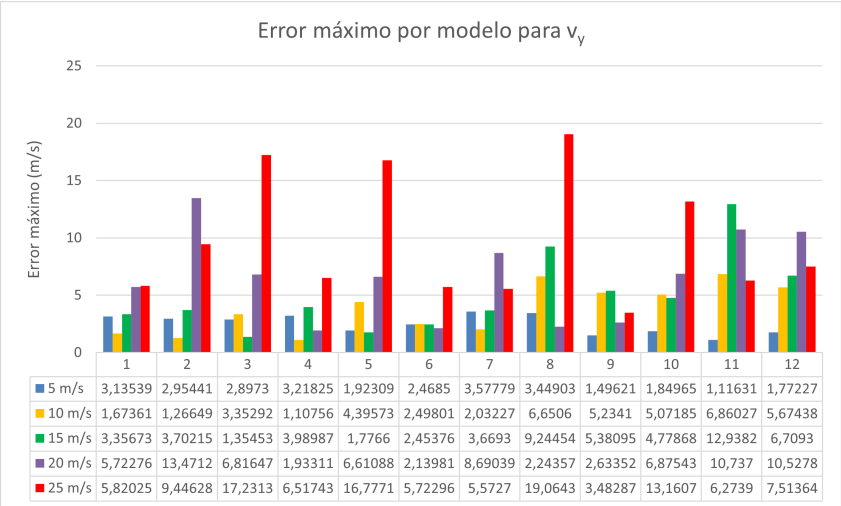
(c)

Figura 5.1: Errores medios de cada modelo neuronal entrenado para cada una de las velocidades de entrada al dominio. a) Velocidad paralela al flujo, b) velocidad vertical y c) presión.



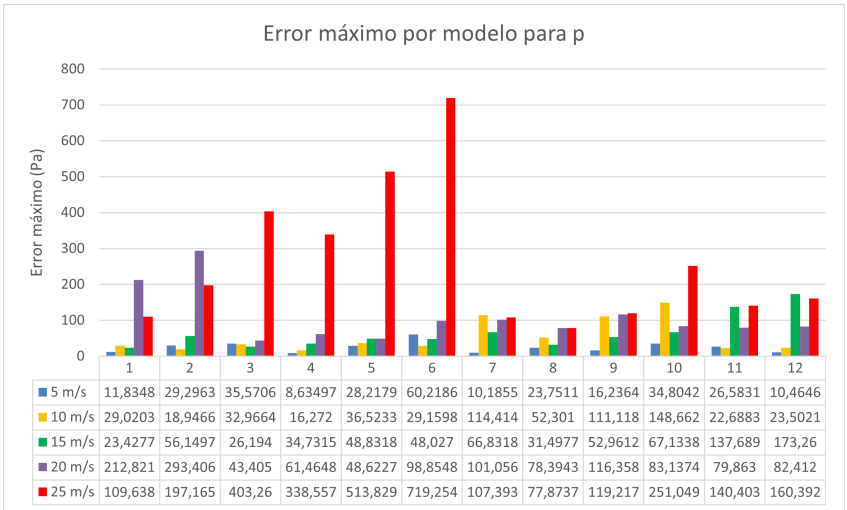
	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	4,6190	2,5936	5,2870	2,6438	2,0203	3,5687	4,2280	1,4932	5,4315	1,1146	1,8962	3,3847
10 m/s	3,6041	3,1354	0,7999	4,1054	1,2031	1,1661	3,1354	2,9909	2,3232	15,8391	2,4757	1,8637
15 m/s	2,9437	3,0346	3,1678	9,6648	2,2080	3,8520	3,3160	6,9400	4,3957	0,9966	3,3060	1,4523
20 m/s	1,1504	4,3957	11,2778	7,4059	22,7344	2,1284	2,8449	5,6522	2,0765	4,8614	27,0690	2,0447
25 m/s	3,7234	4,4029	7,8744	2,1334	1,4854	5,6593	2,4113	2,2970	9,2445	31,1345	6,8073	33,5826

(a)



	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	3,13539	2,95441	2,8973	3,21825	1,92309	2,4685	3,57779	3,44903	1,49621	1,84965	1,11631	1,77227
10 m/s	1,67361	1,26649	3,35292	1,10756	4,39573	2,49801	2,03227	6,6506	5,2341	5,07185	6,86027	5,67438
15 m/s	3,35673	3,70215	1,35453	3,98987	1,7766	2,45376	3,6693	9,24454	5,38095	4,77868	12,9382	6,7093
20 m/s	5,72276	13,4712	6,81647	1,93311	6,61088	2,13981	8,69039	2,24357	2,63352	6,87543	10,737	10,5278
25 m/s	5,82025	9,44628	17,2313	6,51743	16,7771	5,72296	5,5727	19,0643	3,48287	13,1607	6,2739	7,51364

(b)



	1	2	3	4	5	6	7	8	9	10	11	12
5 m/s	11,8348	29,2963	35,5706	8,63497	28,2179	60,2186	10,1855	23,7511	16,2364	34,8042	26,5831	10,4646
10 m/s	29,0203	18,9466	32,9664	16,272	36,5233	29,1598	114,414	52,301	111,118	148,662	22,6883	23,5021
15 m/s	23,4277	56,1497	26,194	34,7315	48,8318	48,027	66,8318	31,4977	52,9612	67,1338	137,689	173,26
20 m/s	212,821	293,406	43,405	61,4648	48,6227	98,8548	101,056	78,3943	116,358	83,1374	79,863	82,412
25 m/s	109,638	197,165	403,26	338,557	513,829	719,254	107,393	77,8737	119,217	251,049	140,403	160,392

(c)

Figura 5.2: Errores máximos de cada modelo neuronal entrenado para cada una de las velocidades de entrada al dominio. a) Velocidad paralela al flujo, b) velocidad vertical y c) presión.

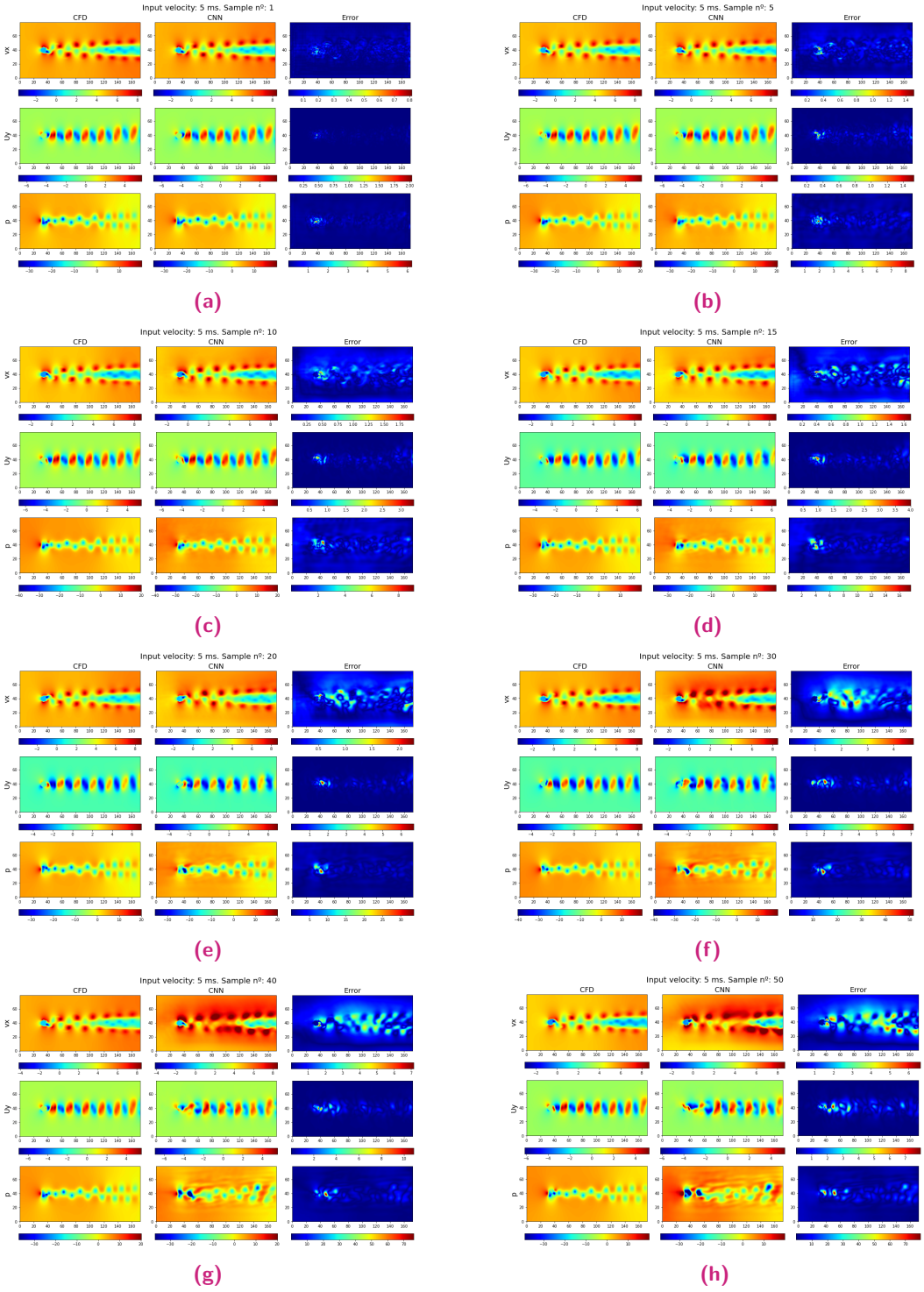


Figura 5.3: Predicciones de la geometría circular con una velocidad de entrada de 5 m/s de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

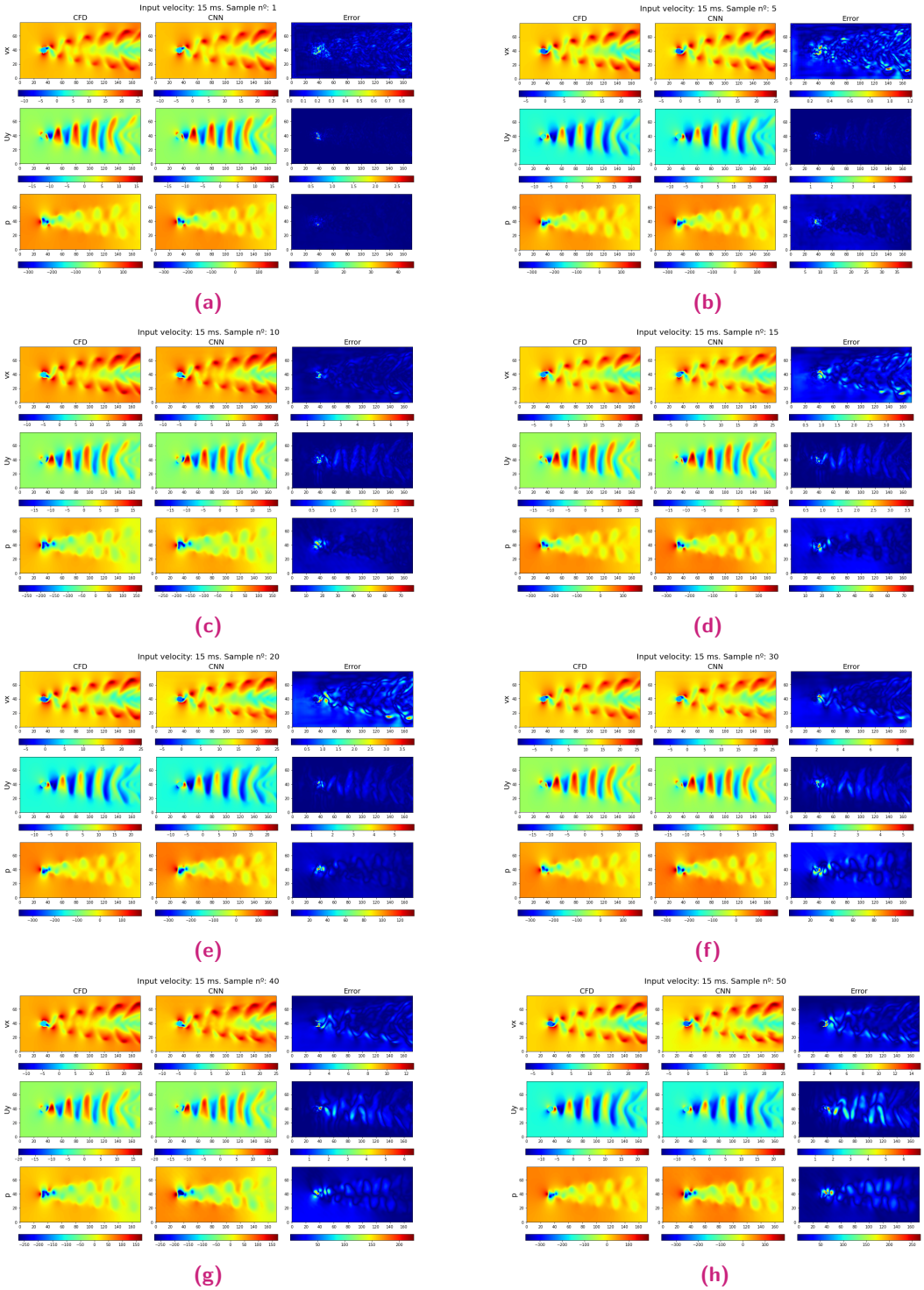


Figura 5.4: Predicciones de la geometría circular con una velocidad de entrada de 15 m/s de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

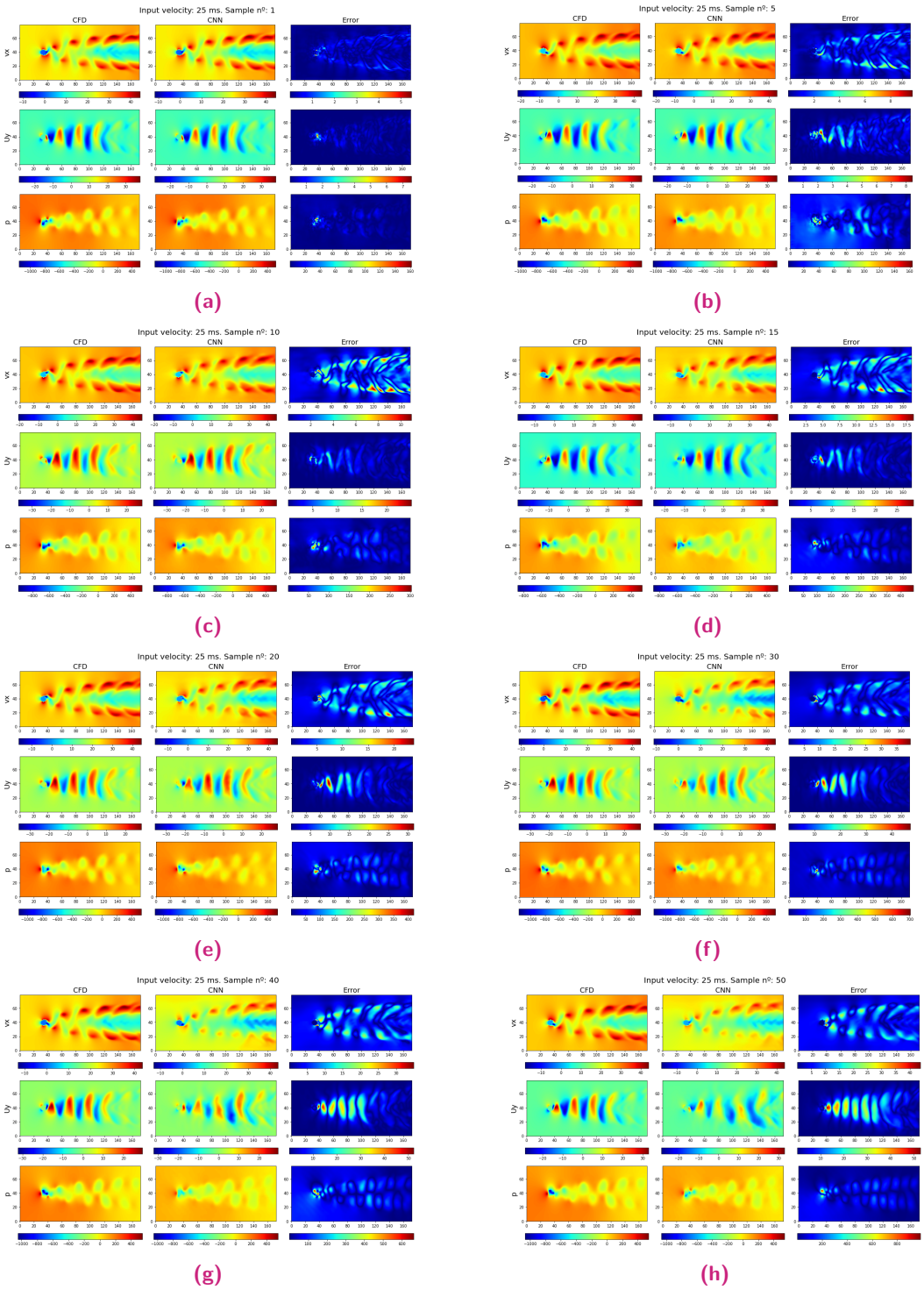


Figura 5.5: Predicciones de la geometría circular con una velocidad de entrada de 25 m/s de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

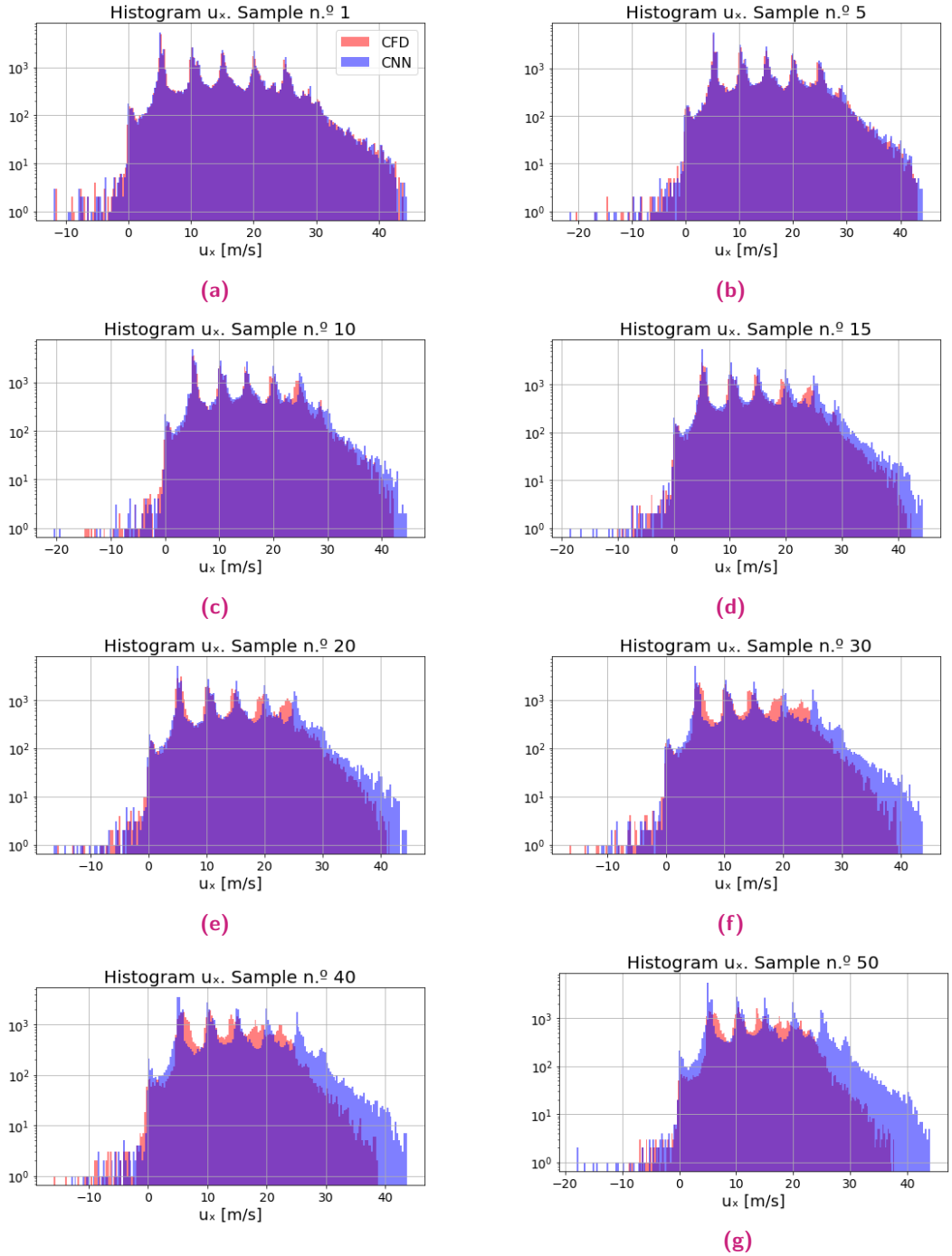


Figura 5.6: Histogramas de la distribución de los datos de CFD y de la CNN del campo de velocidad paralela al flujo alrededor de la geometría circular para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

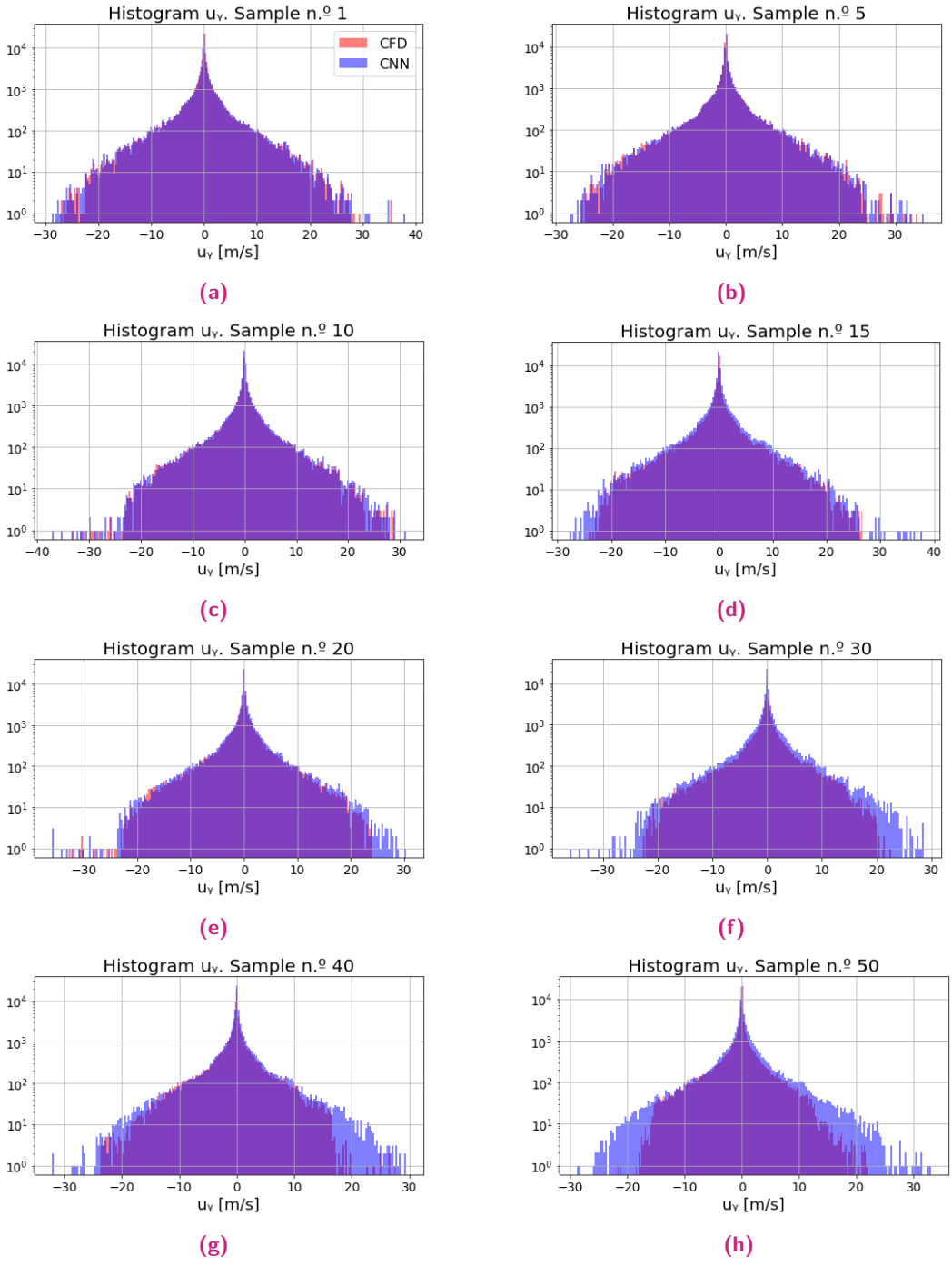


Figura 5.7: Histogramas de la distribución de los datos de CFD y de la CNN del campo de velocidad vertical alrededor de la geometría circular para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

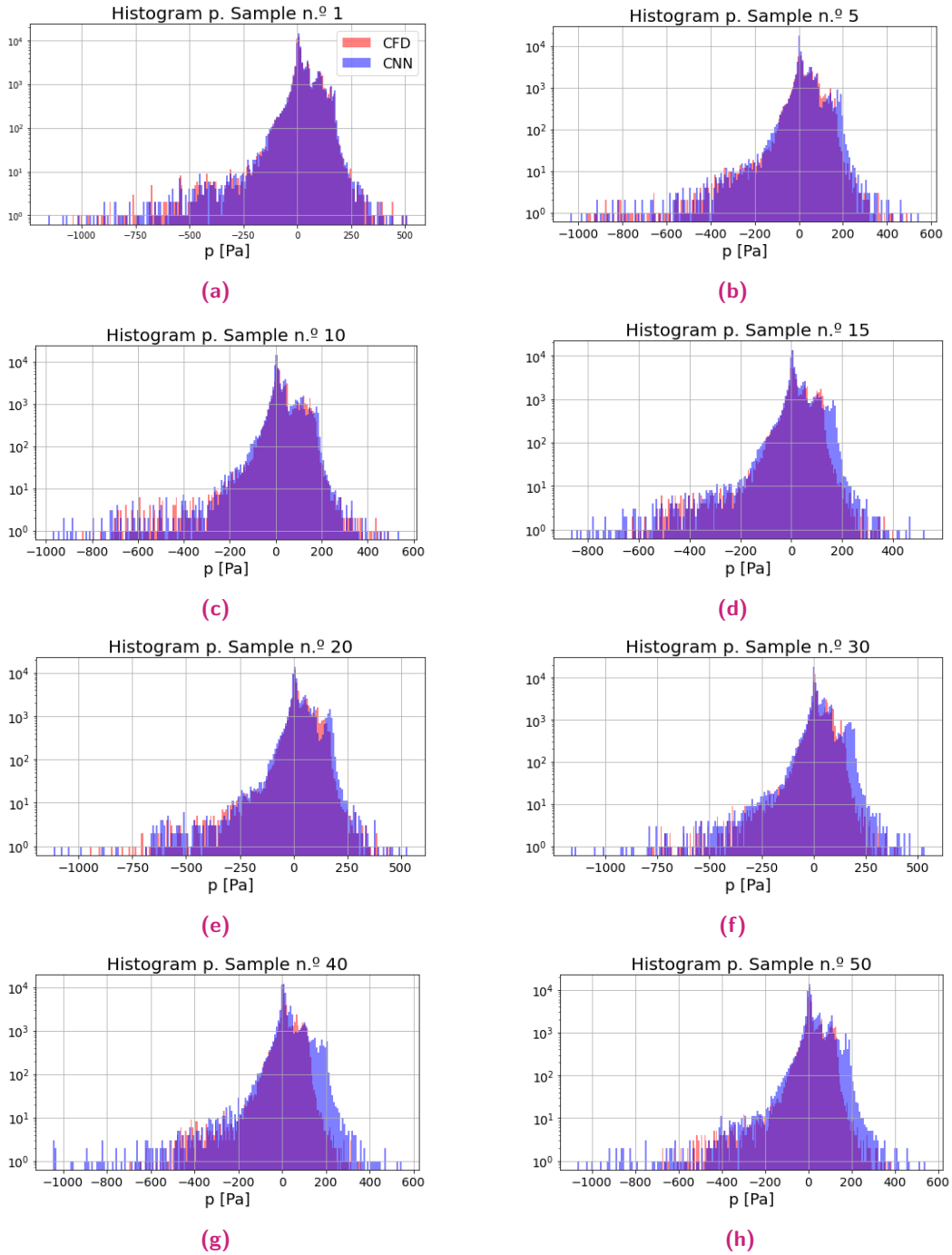


Figura 5.8: Histogramas de la distribución de los datos de CFD y de la CNN del campo de presión alrededor de la geometría circular para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

En las figuras 5.9, 5.10 y 5.11 se muestra gráficamente la comparación de los resultados obtenidos por la CNN con respecto a las simulaciones proporcionadas por el CFD para el círculo, la elipse y el cuadrado. Asimismo, en las figuras 5.12, 5.13 y 5.14 se muestran los histogramas para las tres variables, que comparan todos los valores dados por la CNN con respecto al CFD. Por otro lado, la tabla 5.6 contiene los valores cuantitativos de la media aritmética y la desviación estándar de las simulaciones CFD y de los tests de la CNN, donde se observa que el modelo neuronal presenta una precisión considerable. Los errores absolutos en las predicciones de los tres campos se incrementan a lo largo de las muestras, puesto que dependen de un estado inicial y la aparición de un error excesivo en un cierto punto, influye considerablemente en el error que producirán las predicciones futuras en dicho punto. Durante las primeras 20 muestras predichas, el error absoluto es relativamente pequeño; sin embargo, en la zona posterior al contorno de las geometrías, surgen errores absolutos elevados, que se acumulan a lo largo de las predicciones. En este caso, los vórtices son captados adecuadamente. La CNN proporciona tasas de error reducidas en esta zona, incluso en los instantes más avanzados.

Método	**CFD**			**CNN**		
	v_x (m/s)	**v_y (m/s)**	**p (Pa)**	**v_x (m/s)**	**v_y (m/s)**	p (Pa)
Media aritmética (μ)	5,0283	-0,0538	8,4882	5,0413	-0,0452	8,5360
Desviación estándar (σ)	1,9109	1,7205	17,0865	1,9940	1,7725	16,9417

Tabla 5.6: Media aritmética y desviación estándar de las 50 muestras predichas por la CNN y simuladas mediante CFD.

5.3 Resultados de la red que predice la muestra inicial para geometrías variables

La red ha sido entrenada para todas las combinaciones de los valores de los hiperparámetros proporcionados por la tabla 4.3. En las tablas 5.7, 5.8 y 5.9 se muestran los resultados de los 10 mejores entrenamientos para las variables v_x, v_y y p respectivamente. A la hora de escoger el modelo entrenado óptimo, se observa que la siguiente combinación de los valores de los hiper-parámetros proporciona los mejores resultados:

- Arquitectura: [8, 16, 16, 32, 32]
- Tamaño de ventana: 3.
- Lr: 0,001.
- *Batch size*: 32

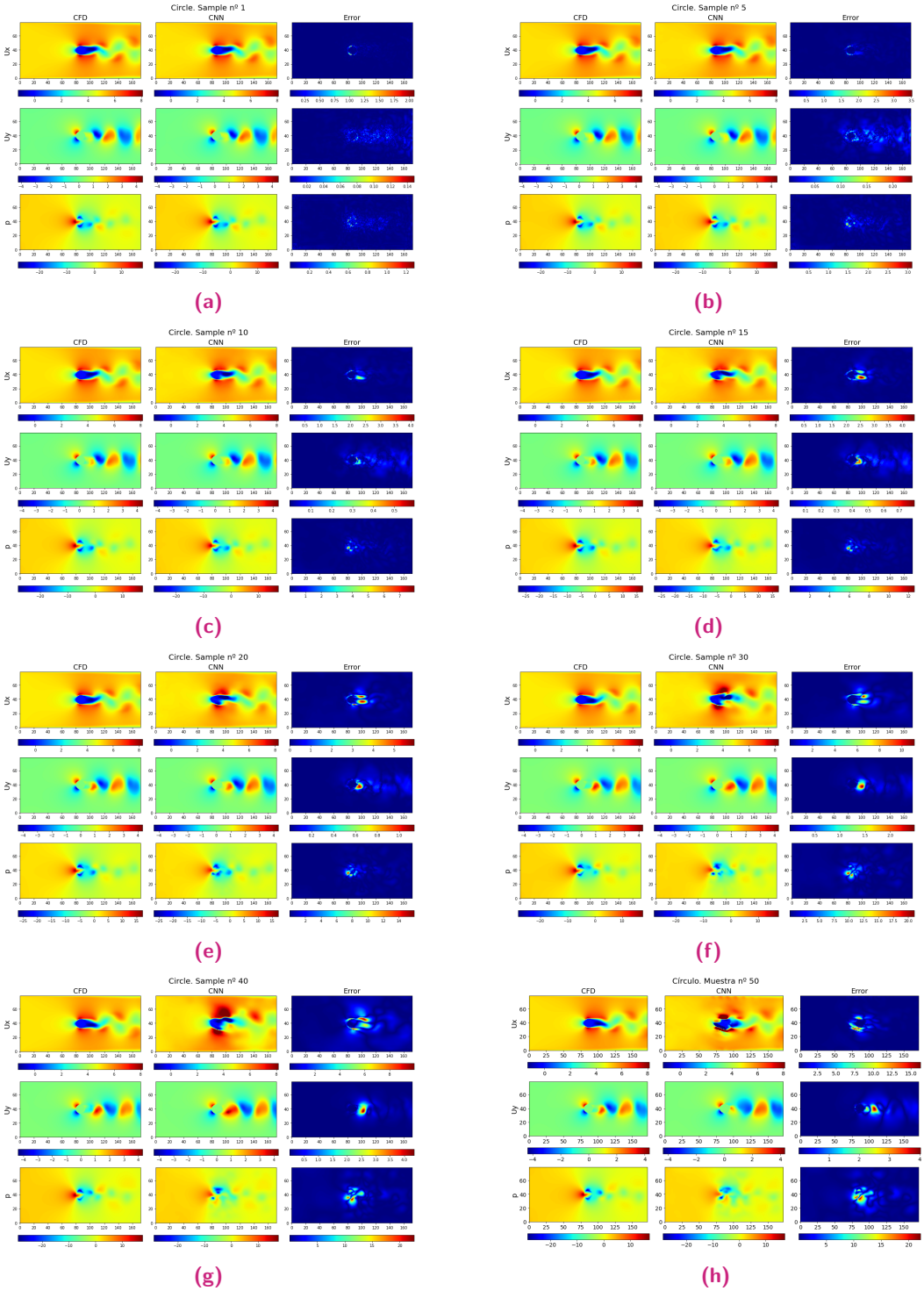


Figura 5.9: Predicciones de la geometría circular de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

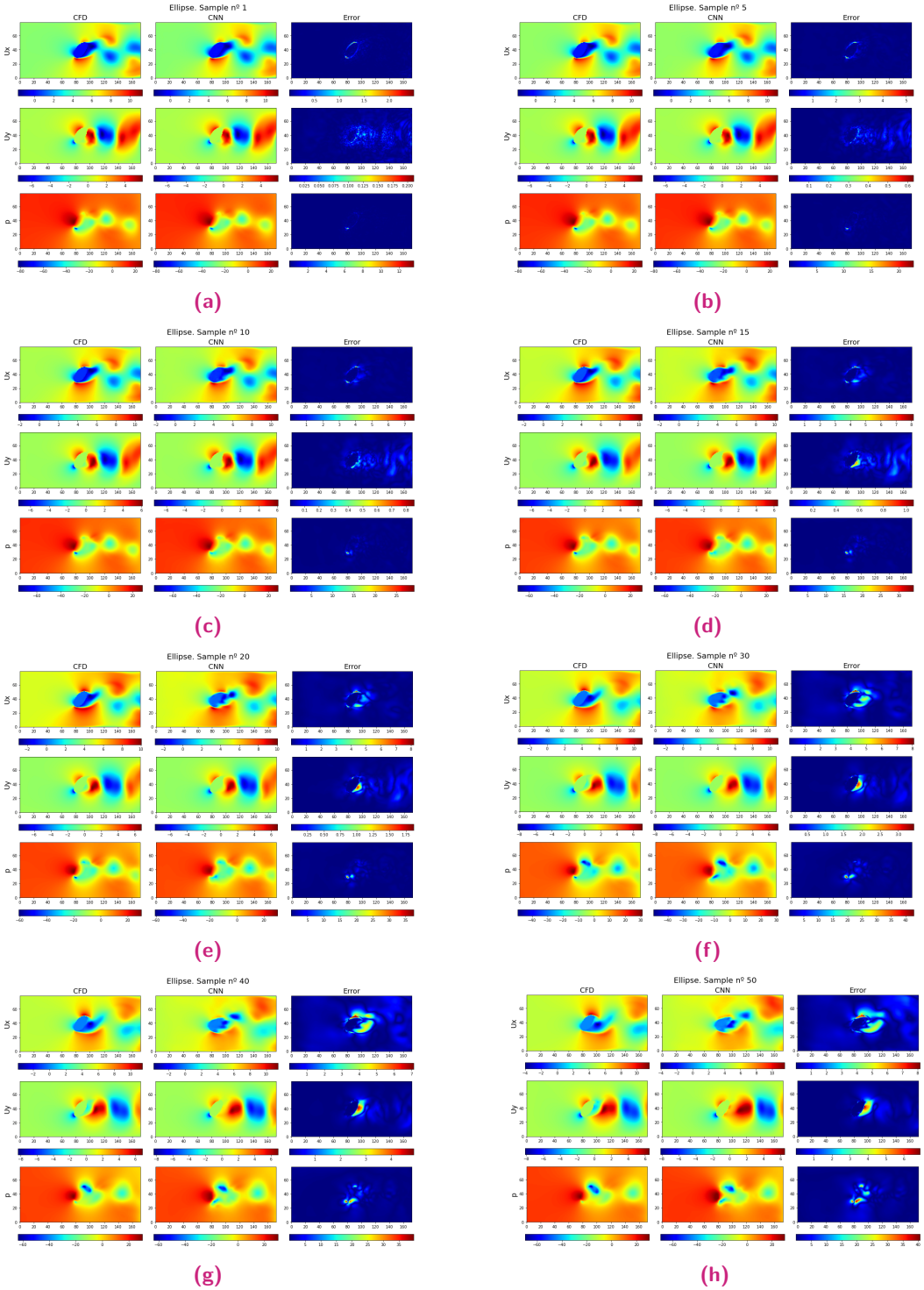


Figura 5.10: Predicciones de la elipse de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

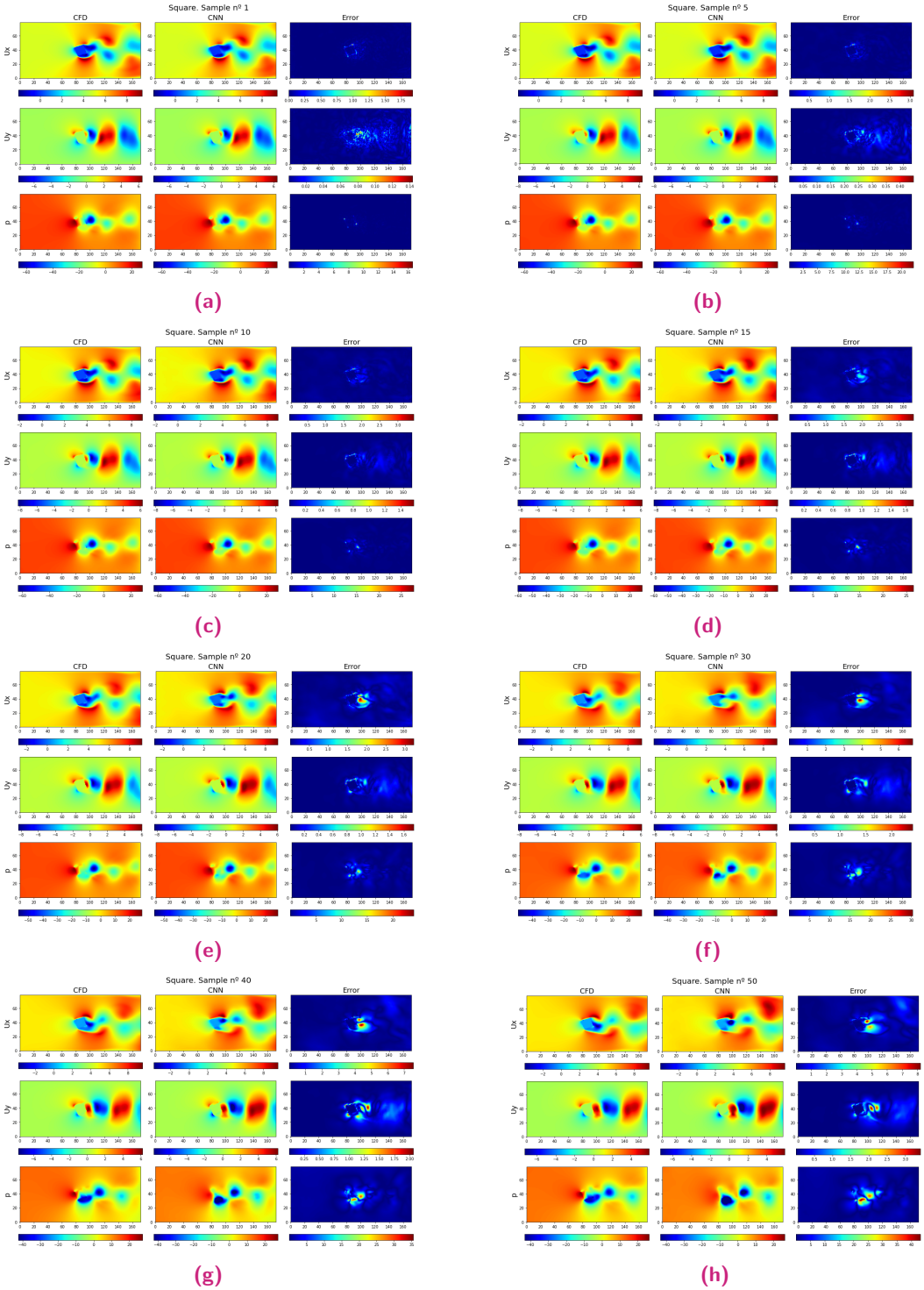


Figura 5.11: Predicciones del cuadrado de las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40, h) 50.

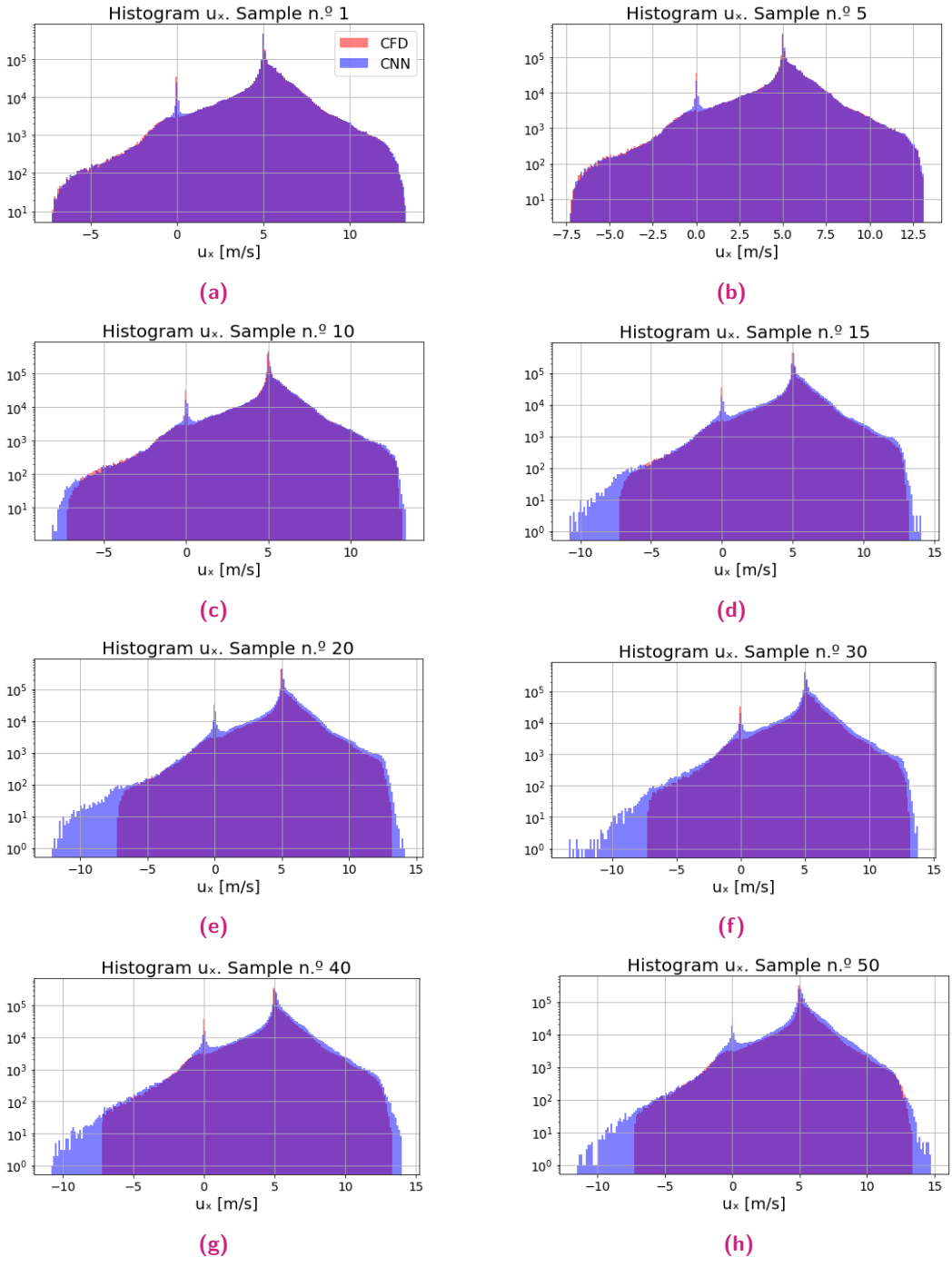


Figura 5.12: Histogramas de la distribución de los datos de CFD y de la CNN del campo de velocidad paralela al flujo para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

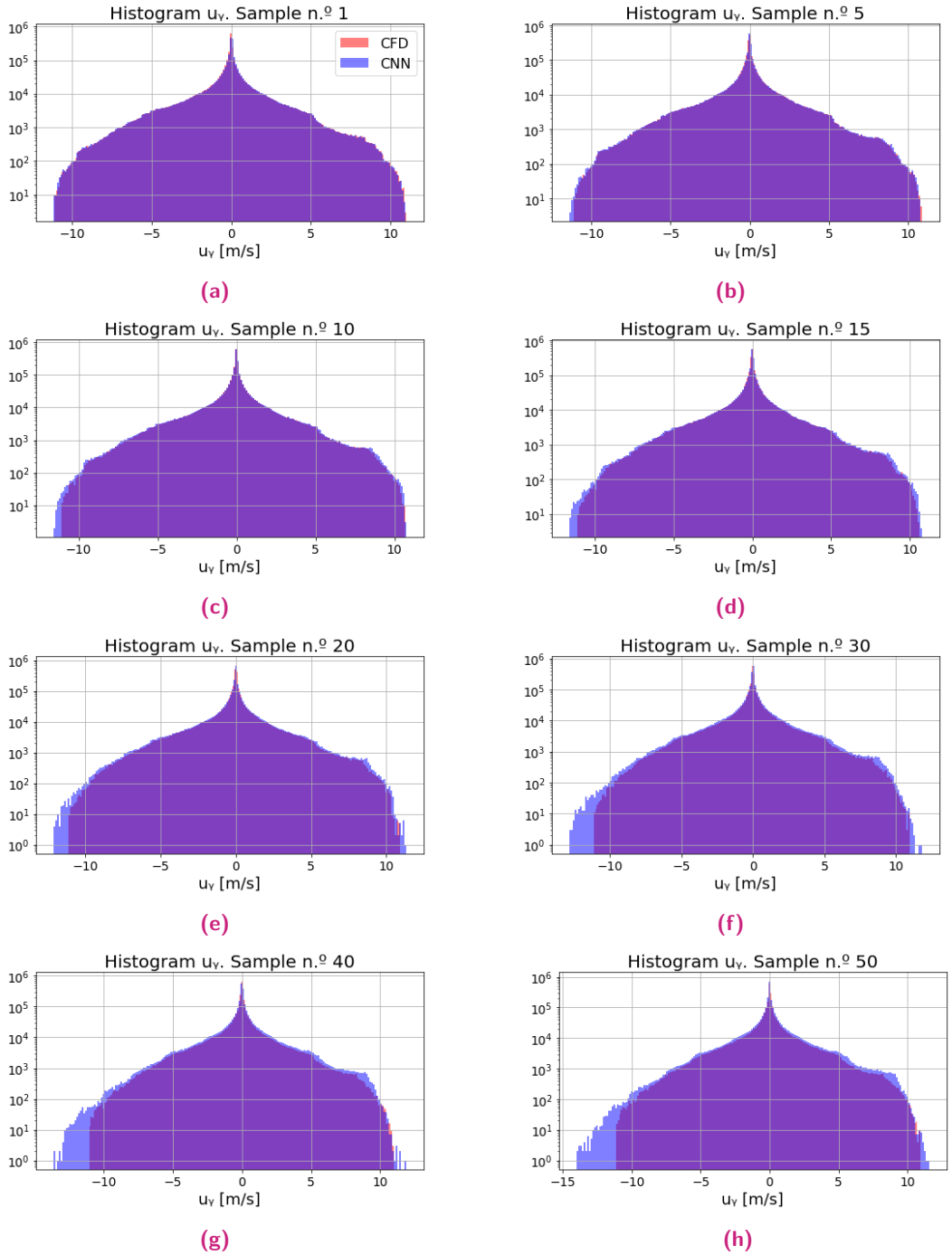


Figura 5.13: Histogramas de la distribución de los datos de CFD y de la CNN del campo de velocidad vertical para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

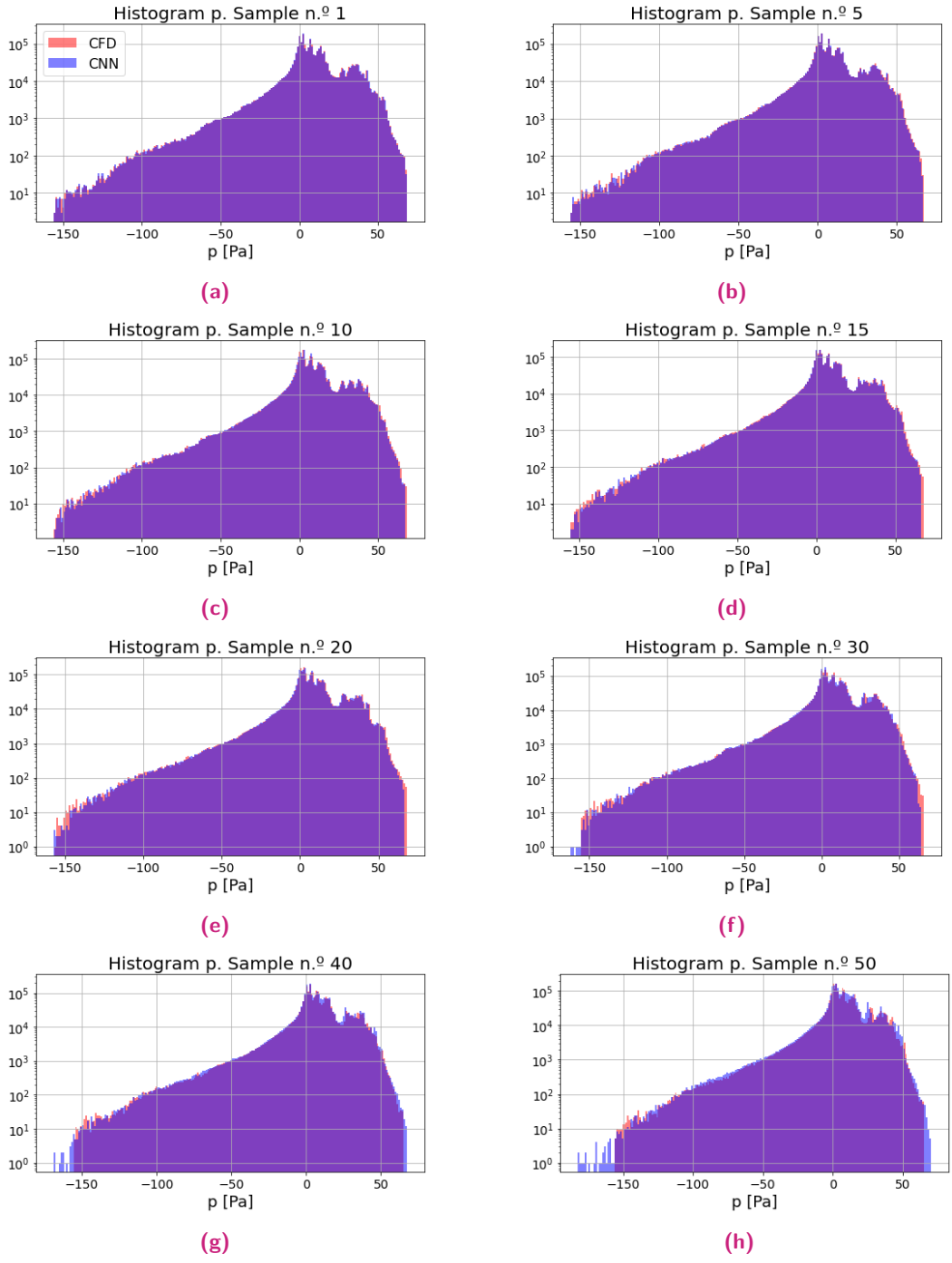


Figura 5.14: Histogramas de la distribución de los datos de CFD y de la CNN del campo de presión para las muestras número: a) 1, b) 5, c) 10, d) 15, e) 20, f) 30, g) 40 y h) 50.

- Número de épocas: 2000

Esta combinación es la adecuada, puesto que se trata del mejor modelo para la velocidad paralela al flujo y la presión y el segundo mejor, para la velocidad vertical. Cada modelo neuronal ha sido ejecutado con las mismas muestras para entrenamiento y test.

Arquitectura	**Tamaño ventana**	Lr	**Batch size**	**N.º épocas**	**Duración entrenamiento (min)**	**Error medio v_x (m/s)**	**Error máx. v_x (m/s)**
[8, 16, 16, 32, 32]	3	0,001	32	2000	19,66	0,0528	2,4436
[8, 16, 32, 32]	5	0,001	32	2000	22,33	0,0550	2,5234
[8, 16, 16, 32, 32]	5	0,001	32	2000	21,60	0,0678	3,0944
[8, 16, 32, 32]	3	0,001	32	2000	19,45	0,0698	3,2475
[8, 16, 32, 32]	7	0,001	32	1000	13,84	0,0705	2,5926
[8, 16, 32, 32]	9	0,0001	32	2000	37,20	0,0707	2,9895
[8, 16, 16, 32, 32]	5	0,001	32	1000	10,88	0,0746	2,9101
[8, 16, 32, 32]	7	0,0001	32	2000	27,85	0,0793	4,3594
[8, 16, 32, 32]	3	0,001	128	2000	17,02	0,0811	4,0057
[8, 16, 16, 32, 32]	9	0,0001	32	2000	34,19	0,0831	3,7460

Tabla 5.7: 10 mejores entrenamientos para la velocidad paralela al flujo.

Arquitectura	**Tamaño ventana**	Lr	**Batch size**	**N.º épocas**	**Duración entrenamiento (min)**	**Error medio v_y (m/s)**	**Error máx. v_y (m/s)**
[8, 16, 32, 32]	7	0,0001	32	2000	27,85	0,0281	2,2092
[8, 16, 16, 32, 32]	3	0,001	32	2000	19,66	0,0288	2,6517

Arquitectura	Tamaño ventana	Lr	Batch size	N.º épocas	Duración entrenamiento (min)	Error medio v_y (m/s)	Error máx. v_y (m/s)
[8, 16, 32, 32]	5	0,001	32	2000	22,33	0,0292	2,4026
[8, 16, 32, 32]	7	0,001	32	1000	13,84	0,0310	2,4049
[8, 16, 32, 32]	3	0,001	32	2000	19,45	0,0333	3,9388
[8, 16, 32, 32]	5	0,001	64	2000	20,31	0,0343	2,6450
[8, 16, 16, 32, 32]	5	0,001	32	2000	21,60	0,0353	2,0954
[8, 16, 32, 32]	9	0,0001	32	2000	37,20	0,0365	2,2067
[8, 16, 32, 32]	5	0,001	32	1000	11,19	0,0371	1,8295
[8, 16, 16, 32, 32]	7	0,001	32	1000	12,88	0,0397	4,2255

Tabla 5.8: 10 mejores entrenamientos para la velocidad vertical.

Arquitectura	Tamaño ventana	Lr	Batch size	N.º épocas	Duración entrenamiento (min)	Error medio p (Pa)	Error máx. p (Pa)
[8, 16, 16, 32, 32]	3	0,001	32	2000	19,66	0,2474	16,1953
[8, 16, 32, 32]	7	0,001	32	1000	13,84	0,2604	18,4768
[8, 16, 32, 32]	5	0,001	32	2000	22,33	0,2643	16,4145
[8, 16, 16, 32, 32]	9	0,001	64	2000	32,99	0,2736	19,4038
[8, 16, 16, 32, 32]	5	0,001	32	2000	21,60	0,2770	22,1191
[8, 16, 32, 32]	5	0,001	64	2000	20,31	0,2814	17,5238
[8, 16, 32, 32]	3	0,001	32	2000	19,45	0,3048	17,0342

Arquitectura	Tamaño ventana	Lr	Batch size	N.º épocas	Duración entrenamiento (min)	Error medio p (Pa)	Error máx. p (Pa)
[8, 16, 32, 32]	7	0,0001	32	2000	27,85	0,3080	23,0767
[8, 16, 16, 32, 32]	9	0,0001	32	2000	34,19	0,3269	27,6613
[8, 16, 32, 32]	5	0,001	32	1000	11,19	0,3314	19,8795

Tabla 5.9: 10 mejores entrenamientos para la presión.

En la figura 5.15 se muestra una predicción del modelo escogido para cada una de las geometrías analizadas. Para evaluar cuantitavimante la precisión del modelo neuronal, mediante la tabla 5.10 se muestran los valores de la media aritmética y la desviación estándar de las simulaciones con CFD y con la CNN. Asimismo, en la figura 5.16 se muestra la distribución de los valores de los campos analizados obtenidos con la CFD y la CNN. Ambas permiten alcanzar la conclusión de que el modelo neuronal es considerablemente preciso con respecto a las simulaciones CFD.

Método	CFD			CNN		
	v_x (m/s)	v_y (m/s)	p (Pa)	v_x (m/s)	v_y (m/s)	p (Pa)
Media aritmética (μ)	5,0674	-0,0644	5,9574	5,0414	-0,0628	6,0475
Desviación estándar (σ)	1,6867	1,5074	15,0737	1,6797	1,5142	15,0853

Tabla 5.10: Media aritmética y desviación estándar de las predicciones de la CNN de la primera muestra y de las simulaciones CFD de la muestra inicial.

5.4 Análisis del coste computacional

En esta sección se lleva a cabo la comparativa de los tiempos computacionales necesarios para obtener los resultados de los campos de velocidades y de presión mediante las simulaciones CFD y las predicciones de los modelos neuronales.

En el caso de los modelos neuronales que predicen los instantes futuros del fluido con velocidad de entrada variable, el tiempo de entrenamiento ha sido de 6,73, 9,41 y 7,26 horas, dando un total de 23,40 horas de entrenamiento. La predicción de

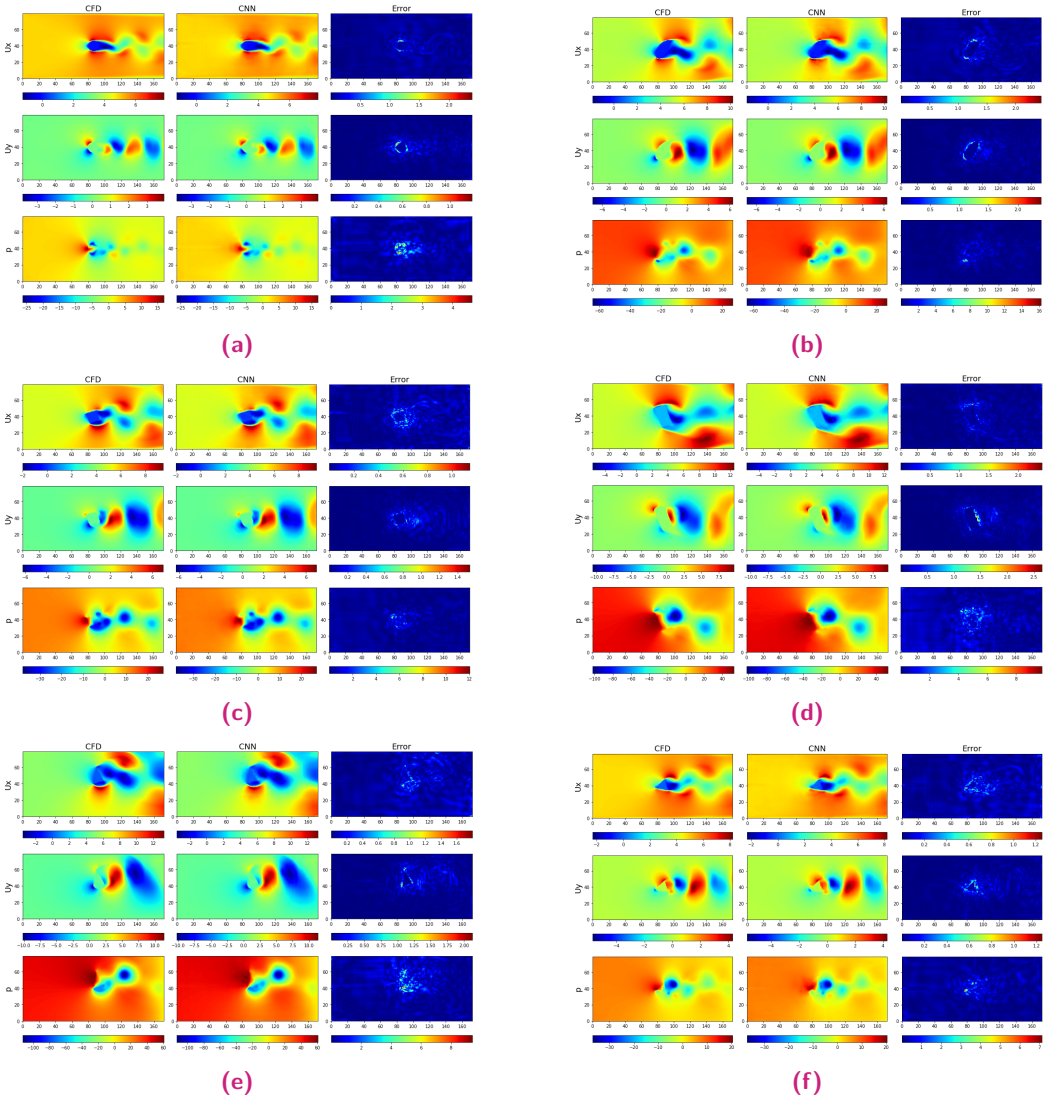


Figura 5.15: Predicciones de la CNN para a) círculo, b) cilindro, c) cuadrado, d) rectángulo, e) triángulo y f) triángulo equilátero.

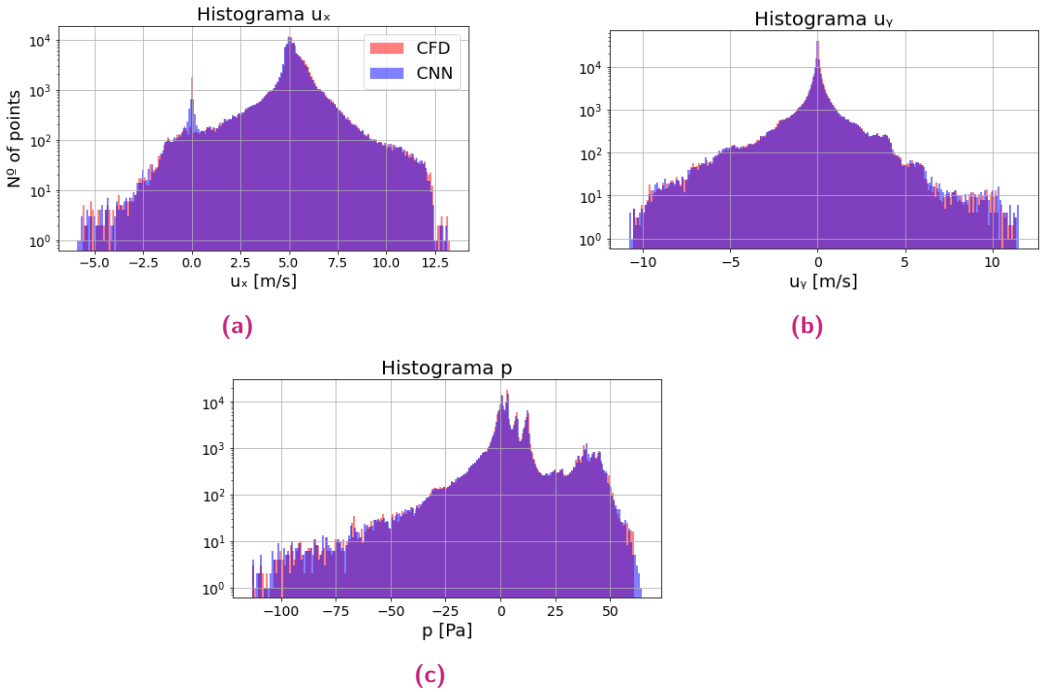


Figura 5.16: Distribución de los datos para: a) v_x, b) v_y y c) p.

50 instantes tarda 1,56, 3,21 y 1,84 segundos. El entrenamiento de los modelos neuronales de la red que predice los instantes futuros para geometrías variables, proporciona unos tiempos de entrenamiento de 2,60, 1,45 y 2,72 horas, dando un total de 6,77 horas. La predicción de 50 instantes lleva 1,76, 3,77 y 6,06 segundos. Por último, el modelo neuronal que predice el instante inicial ha sido entrenado durante 19,66 minutos y las predicciones son de 0,08 segundos. En la tabla 5.11 se muestran las comparativas de la duración de las predicciones por parte de los modelos neuronales con respecto a la duración de las simulaciones CFD. En el caso de las predicciones de las muestras futuras, se compara el tiempo necesario para predecir 50 muestras de las tres variables. Para la predicción de la muestra inicial, se compara el tiempo necesario para predecir una única muestra. Además, se ha de tener en cuenta el tiempo ahorrado gracias a la aplicación de la técnica de *data augmentation*, ya que evita la generación de un mayor número de simulaciones CFD. Esta técnica ahorra la generación de 50 nuevas simulaciones por cada geometría, que se traduce en un ahorro aproximado de 120 segundos por geometría.

Red	Duración simulaciones CFD (s)	Duración predicciones (s)	Reducción de tiempo
Velocidades de entrada variables	540,00	6,61	81,69
Geometrías variables	720,00	11,59	62,12
Muestra inicial	720,00	0,08	9000

Tabla 5.11: Comparación del tiempo de cálculo de los campos de velocidades paralela al flujo y vertical y de presión.

Conclusiones y trabajos futuros 6

6.1 Discusión general

Las simulaciones CFD son de gran utilidad para el estudio del comportamiento de los flujos turbulentos al enfrentarse a determinadas geometrías. No obstante, el coste computacional de estas simulaciones es demasiado elevado para ciertas aplicaciones en las que se requieren resultados rápidos. Asimismo, la precisión de estas simulaciones dependen también de la influencia que tiene el usuario en la generación del mallado y el modelo de turbulencia. Las técnicas de DL, como numerosos estudios demuestran, son eficaces para lograr aproximaciones rápidas y relativamente precisas.

En este estudio, se ha demostrado la flexibilidad de la estructura U-Net en su aplicación a diferentes situaciones fluido dinámicas. Esta estructura es capaz de predecir los campos de velocidades paralela al flujo y vertical y de presión para características variables del flujo. Asimismo, se demuestra no solo la capacidad de predecir una muestra de las variables analizadas, sino que se logra predecir, con errores relativamente pequeños hasta un número concreto de muestras, los instantes futuros del fluido.

6.2 Conclusiones específicas

En primer lugar, se han logrado unas aproximaciones relativamente precisas de los campos de velocidades paralela al flujo y vertical y de presión con una geometría circular y velocidad de entrada del fluido variable durante 20 instantes, partiendo de una inicial. A partir de la muestra 20, aparece un destacado incremento de los errores absolutos, especialmente en la velocidad paralela al flujo y en las velocidades de entrada más elevadas. En la velocidad vertical y la presión, las aproximaciones son generalmente más fidedignas, incluso para las muestras de los instantes más avanzados.

En el caso de la predicción de los instantes futuros de las geometrías variables, ocurre de forma similar al de la velocidad de entrada variable. Hasta la muestra del instante número 20, las predicciones son relativamente fiables. Los errores son más elevados en la velocidad paralela al flujo, que en la velocidad vertical y la presión. En este caso, los vórtices son captados adecuadamente, quedando las mayores tasas de error en las zonas contiguas a las geometrías.

Se confirma que la técnica de *data augmentation* empleada es eficaz para incrementar el número de muestras necesarias para el entrenamiento, de forma que no sea necesario llevar a cabo nuevas simulaciones CFD para el aumento de los datos. Gracias a ello, el modelo neuronal generado predice con precisión los campos de velocidades paralela al flujo, vertical y de presión.

Respecto al objetivo de reducir el coste computacional, el primer modelo neuronal predice 81,69 veces más rápido, el segundo 62,12 veces y el tercero, 9000. En los tres casos se cumple el objetivo de reducir el coste computacional, pero gracias a la técnica de *data augmentation* aplicada, en la predicción de la primera muestra el coste computacional se reduce drásticamente.

6.3 Líneas de trabajo futuras

El campo de la aplicación de DL a las simulaciones CFD es muy amplio. Son muchas y variadas las características fluido-dinámicas con las que se puede trabajar. El comportamiento del fluido varía ampliamente según el tipo de fluido estudiado y sus magnitudes de temperatura, densidad, velocidad, número de Reynolds...

Respecto a futuros estudios, cabe la posibilidad de aplicar la CNN con estructura U-Net a geometría con un comportamiento aerodinámico superior, como los perfiles aerodinámicos. Otro posible enfoque correspondería con la aplicación de técnicas de DL para predecir la mejor forma de un perfil aerodinámico con un flap de Gurney añadido, en busca de la optimización aerodinámica.

Presupuesto y planificación

7

7.1 Presupuesto

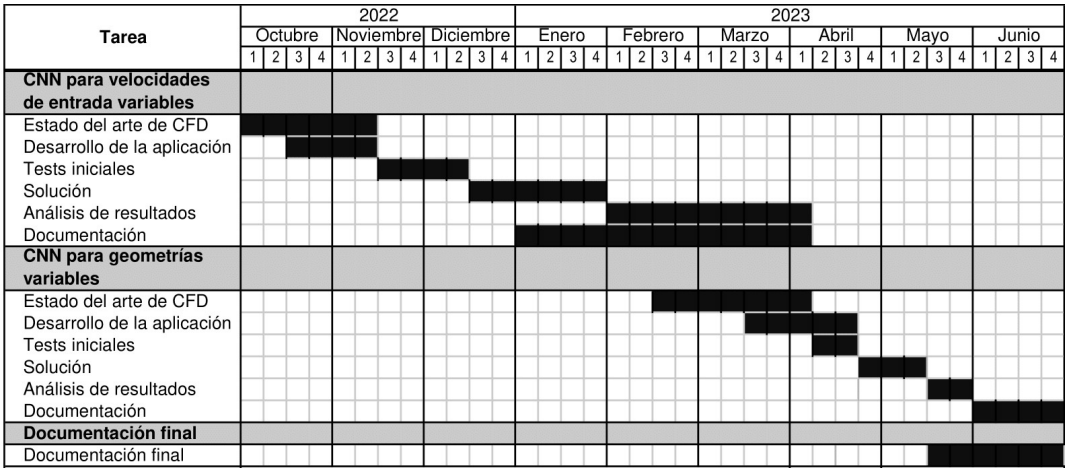
Concepto	Amortización ordenador portátil	Amortización ordenador remoto
Coste producto	1000 €	3000 €
Tiempo de amortización	4 años	4 años
Tiempo de trabajo	1700 horas/año	1700 horas/año
Tiempo de trabajo total	6800 horas	6800 horas
Coste por hora	0,15 €/hora	0,44 €/hora

Tabla 7.1: Amortizaciones.

Concepto	Coste unitario	Cantidad	Coste total	Porcentaje
Costes directos			**25630,20 €**	**72,46 %**
Coste tiempo invertido	50 €/hora	420 horas	21000,00 €	81,93 %
Coste simulaciones CFD	400 €/sim.	11 sim.	4400,00 €	17,17 %
Amortización ordenador portátil	0,15 €/hora	420 horas	63,00 €	0,25 %
Amortización ordenador remoto	0,44 €/hora	380 horas	167,20 €	0,65 %
Costes indirectos			**3844,53 €**	**10,87 %**
		15 % directos	3844,53 €	100,00 %
TOTAL Costes			29474,73 €	
Beneficio industrial			**5894,95 €**	**16,67 %**
		20 % costes	5894,95 €	100,00 %
TOTAL			**35369,68 €**	**100,00 %**
IVA		21 % total	7427,63 €	
TOTAL + IVA			**42797,31 €**	

Tabla 7.2: Presupuesto.

7.2 Planificación



Referencias bibliográficas

[1]Alvaro Abucide-Armas, Koldo Portal-Porras, Unai Fernandez-Gamiz, Ekaitz Zulueta y Adrian Teso-Fz-Betoño. „A Data Augmentation-Based Technique for Deep Learning Applied to CFD Simulations“. En: *Mathematics* 9.16 (4 de ago. de 2021), pág. 1843 (vid. págs. 16, 29).

[2]Lionel Agostini. „Exploration and prediction of fluid dynamical systems using auto-encoder technology“. En: *Physics of Fluids* 32.6 (1 de jun. de 2020), pág. 067103 (vid. pág. 16).

[3]Saad Albawi, Tareq Abed Mohammed y Saad Al-Zawi. „Understanding of a convolutional neural network“. En: *2017 International Conference on Engineering and Technology (ICET)*. 2017 International Conference on Engineering and Technology (ICET). Antalya: IEEE, ago. de 2017, págs. 1-6 (vid. pág. 12).

[4]Iñigo Aramendia, Unai Fernandez-Gamiz, Ekaitz Zulueta Guerrero, Jose Lopez-Guede y Javier Sancho. „Power Control Optimization of an Underwater Piezoelectric Energy Harvester“. En: *Applied Sciences* 8.3 (7 de mar. de 2018), pág. 389 (vid. pág. 18).

[5]Han Bao, Jinyong Feng, Nam Dinh y Hongbin Zhang. „Computationally efficient CFD prediction of bubbly flow using physics-guided deep learning“. En: *International Journal of Multiphase Flow* 131 (oct. de 2020), pág. 103378 (vid. pág. 15).

[6]Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser et al. „Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI“. En: *Information Fusion* 58 (jun. de 2020), págs. 82-115 (vid. pág. 1).

[7]Richard Bellman. *An introduction to artificial intelligence: can computers think?* San Francisco: Boyd & Fraser Pub. Co, 1978. 146 págs. (vid. pág. 5).

[8]Eugene Charniak y Drew V. McDermott. *Introduction to artificial intelligence*. Reading, Mass: Addison-Wesley, 1985. 701 págs. (vid. pág. 5).

[9]Liang Deng, Yueqing Wang, Yang Liu et al. „A CNN-based vortex identification method“. En: *Journal of Visualization* 22.1 (feb. de 2019), págs. 65-78 (vid. pág. 15).

[10]Rui Fang, David Sondak, Pavlos Protopapas y Sauro Succi. „Deep learning for turbulent channel flow“. En: *arXiv:1812.02241 [physics]* (5 de dic. de 2018). arXiv: 1812.02241 (vid. pág. 16).

[11]Peter A. Flach. *Machine learning: the art and science of algorithms that make sense of data*. OCLC: ocn795181906. Cambridge ; New York: Cambridge University Press, 2012. 396 págs. (vid. pág. 6).

[12]M.W Gardner y S.R Dorling. „Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences“. En: *Atmospheric Environment* 32.14 (ago. de 1998), págs. 2627-2636 (vid. pág. 11).

[13]Francisco J. Gonzalez y Maciej Balajewicz. „Deep convolutional recurrent autoencoders for learning low-dimensional feature dynamics of fluid systems“. En: *arXiv:1808.01346 [physics]* (22 de ago. de 2018). arXiv: 1808.01346 (vid. pág. 16).

[14]Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. 775 págs. (vid. pág. 11).

[15]Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza et al. *Generative Adversarial Networks*. arXiv:1406.2661. type: article. arXiv, 10 de jun. de 2014. arXiv: 1406.2661[cs,stat] (vid. pág. 12).

[16]Xiaoxiao Guo, Wei Li y Francesco Iorio. „Convolutional Neural Networks for Steady Flow Approximation“. En: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16: The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco California USA: ACM, 13 de ago. de 2016, págs. 481-490 (vid. págs. 13, 15, 16, 23).

[17]Botros N. Hanna, Nam T. Dinh, Robert W. Youngblood e Igor A. Bolotnov. „Coarse-Grid Computational Fluid Dynamic (CG-CFD) Error Prediction using Machine Learning“. En: *arXiv:1710.09105 [physics]* (25 de oct. de 2017). arXiv: 1710.09105 (vid. pág. 15).

[18]John Haugeland. *Artificial intelligence: the very idea*. 6. print. Bradford books. Cambridge, Mass.: MIT Press, 1993. 287 págs. (vid. pág. 5).

[19]G. Iaccarino, A. Ooi, P.A. Durbin y M. Behnia. „Reynolds averaged simulation of unsteady separated flow“. En: *International Journal of Heat and Fluid Flow* 24.2 (abr. de 2003), págs. 147-156 (vid. pág. 18).

[20]D. Jakhar e I. Kaur. „Artificial intelligence, machine learning and deep learning: definitions and differences“. En: *Clinical and Experimental Dermatology* 45.1 (ene. de 2020), págs. 131-132 (vid. pág. 7).

[21]Ali Kashefi, Davis Rempe y Leonidas J. Guibas. „A Point-Cloud Deep Learning Framework for Prediction of Fluid Flow Fields on Irregular Geometries“. En: *Physics of Fluids* 33.2 (1 de feb. de 2021), pág. 027104. arXiv: 2010.09469 (vid. pág. 15).

[22]Ryan King, Oliver Hennigh, Arvind Mohan y Michael Chertkov. „From Deep to Physics-Informed Learning of Turbulence: Diagnostics“. En: *arXiv:1810.07785 [nlin, physics:physics, stat]* (5 de dic. de 2018). arXiv: 1810.07785 (vid. pág. 16).

[23]Diederik P. Kingma y Jimmy Ba. „Adam: A Method for Stochastic Optimization“. En: (2014). Publisher: arXiv Version Number: 9 (vid. pág. 24).

[24]Max Kuhn y Kjell Johnson. *Applied predictive modeling*. OCLC: ocn827083441. New York: Springer, 2013. 600 págs. (vid. pág. 9).

[25]Ray Kurzweil. *The age of intelligent machines*. 3. print. Cambridge, Mass: MIT Press, 1999. 565 págs. (vid. pág. 5).

[26]Yann LeCun, Yoshua Bengio y Geoffrey Hinton. „Deep learning“. En: *Nature* 521.7553 (28 de mayo de 2015), págs. 436-444 (vid. pág. 7).

[27]Sangseung Lee y Donghyun You. „Prediction of laminar vortex shedding over a cylinder using deep learning“. En: *arXiv:1712.07854 [physics]* (21 de dic. de 2017). arXiv: 1712.07854 (vid. pág. 15).

[28]Jichao Li, Xiaosong Du y Joaquim R. R. A. Martins. *Machine Learning in Aerodynamic Shape Optimization*. arXiv:2202.07141. type: article. arXiv, 14 de feb. de 2022. arXiv: 2202.07141[physics] (vid. pág. 1).

[29]Julia Ling, Andrew Kurzawski y Jeremy Templeton. „Reynolds averaged turbulence modelling using deep neural networks with embedded invariance“. En: *Journal of Fluid Mechanics* 807 (25 de nov. de 2016), págs. 155-166 (vid. pág. 15).

[30]Yang Liu, Yutong Lu, Yueqing Wang et al. „A CNN-based shock detection method in flow visualization“. En: *Computers & Fluids* 184 (abr. de 2019), págs. 1-9 (vid. pág. 15).

[31]Ilya Loshchilov y Frank Hutter. „Decoupled Weight Decay Regularization“. En: (2017). Publisher: arXiv Version Number: 3 (vid. pág. 24).

[32]*MATLAB - MathWorks*. URL: https://www.mathworks.com/products/matlab.html (visitado 20 de nov. de 2022) (vid. pág. 23).

[33]Romit Maulik, Bethany Lusch y Prasanna Balaprakash. „Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders“. En: *Physics of Fluids* 33.3 (1 de mar. de 2021), pág. 037106. arXiv: 2002.00470 (vid. pág. 16).

[34]Arvind Mohan, Don Daniel, Michael Chertkov y Daniel Livescu. „Compressed Convolutional LSTM: An Efficient Deep Learning framework to Model High Fidelity 3D Turbulence“. En: *arXiv:1903.00033 [nlin, physics:physics]* (4 de mar. de 2019). arXiv: 1903.00033 (vid. pág. 16).

[35]Nils J. Nilsson. *Artificial intelligence: a new synthesis*. 5th print. San Francisco, Calif: Kaufmann, 2003. 513 págs. (vid. pág. 6).

[36]Hashem Nowruzi, Hassan Ghassemi y Mahmoud Ghiasi. „Performance predicting of 2D and 3D submerged hydrofoils using CFD and ANNs“. En: *Journal of Marine Science and Technology* 22.4 (dic. de 2017), págs. 710-733 (vid. pág. 16).

[37]David L. Poole, Alan K. Mackworth y Randy Goebel. *Computational intelligence: a logical approach*. New York: Oxford University Press, 1998. 558 págs. (vid. pág. 6).

[38]Koldo Portal-Porras, Unai Fernandez-Gamiz, Ainara Ugarte-Anero, Ekaitz Zulueta y Asier Zulueta. „Alternative Artificial Neural Network Structures for Turbulent Flow Velocity Field Prediction“. En: *Mathematics* 9.16 (14 de ago. de 2021), pág. 1939 (vid. pág. 16).

[39]Mateus Dias Ribeiro, Abdul Rehman, Sheraz Ahmed y Andreas Dengel. „DeepCFD: Efficient Steady-State Laminar Flow Approximation with Deep Convolutional Neural Networks“. En: *arXiv:2004.08826 [physics]* (2 de mayo de 2020). arXiv: 2004.08826 (vid. págs. 15, 19).

[40]Elaine Rich y Kevin Knight. *Artificial intelligence*. 2. ed. New York: McGraw-Hill, 1991. 621 págs. (vid. pág. 5).

[41]Olaf Ronneberger, Philipp Fischer y Thomas Brox. „U-Net: Convolutional Networks for Biomedical Image Segmentation“. En: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. por Nassir Navab, Joachim Hornegger, William M. Wells y Alejandro F. Frangi. Vol. 9351. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, págs. 234-241 (vid. págs. 16, 19).

[42]Stuart J. Russell, Peter Norvig y Ernest Davis. *Artificial intelligence: a modern approach*. 3rd ed. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010. 1132 págs. (vid. pág. 5).

[43]Alex Sherstinsky. „Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network“. En: *Physica D: Nonlinear Phenomena* 404 (mar. de 2020), pág. 132306 (vid. pág. 12).

[44]*Siemens Software*. URL: https://www.plm.automation.siemens.com/global/en/ (visitado 20 de nov. de 2022) (vid. págs. 18, 26).

[45]Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever y Ruslan Salakhutdinov. „Dropout: a simple way to prevent neural networks from overfitting“. En: *The journal of machine learning research* 15.1 (2014). Publisher: JMLR. org, págs. 1929-1958 (vid. pág. 11).

[46]Jun Tao y Gang Sun. „Application of deep learning based multi-fidelity surrogate model to robust aerodynamic design optimization“. En: *Aerospace Science and Technology* 92 (sep. de 2019), págs. 722-737 (vid. pág. 14).

[47]Nils Thuerey, Konstantin Weißenow, Lukas Prantl y Xiangyu Hu. „Deep Learning Methods for Reynolds-Averaged Navier–Stokes Simulations of Airfoil Flows“. En: *AIAA Journal* 58.1 (ene. de 2020), págs. 25-36 (vid. pág. 16).

[48]Jordi Torres. *Python deep learning: introducción práctica con Keras y TensorFlow 2*. OCLC: 1148195914. Barcelona: Marcombo, 2020 (vid. pág. 6).

[49]Sun-Chong Wang. *Interdisciplinary Computing in Java Programming*. OCLC: 852790067. Boston, MA: Springer US, 2003 (vid. pág. 6).

[50]John F. Wendt, John D. Anderson y Von Karman Institute for Fluid Dynamics, eds. *Computational fluid dynamics: an introduction*. 3rd ed. OCLC: ocn288984495. Berlin ; [London]: Springer, 2008. 332 págs. (vid. pág. 14).

[51]Patrick Henry Winston. *Artificial intelligence*. 3rd ed. Reading, Mass: Addison-Wesley Pub. Co, 1992. 737 págs. (vid. pág. 5).

[52]Xinghui Yan, Jihong Zhu, Minchi Kuang y Xiangyang Wang. „Aerodynamic shape optimization using a novel optimizer based on machine learning techniques“. En: *Aerospace Science and Technology* 86 (mar. de 2019), págs. 826-835 (vid. pág. 15).

[53]Li Yang y Abdallah Shami. „On Hyperparameter Optimization of Machine Learning Algorithms: Theory and Practice“. En: *Neurocomputing* 415 (nov. de 2020), págs. 295-316. arXiv: 2007.15745[cs,stat] (vid. pág. 9).

[54]Xinshuai Zhang, Fangfang Xie, Tingwei Ji, Zaoxu Zhu y Yao Zheng. „Multi-fidelity deep neural network surrogate model for aerodynamic shape optimization“. En: *Computer Methods in Applied Mechanics and Engineering* 373 (ene. de 2021), pág. 113485 (vid. pág. 14).

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN INGENIERÍA DE CONTROL, AUTOMATIZACIÓN Y ROBÓTICA

TRABAJO FIN DE MÁSTER

ANEXO A: ESQUEMAS Y PROGRAMAS FUENTE

DISEÑO Y SIMULACIÓN INTELIGENTES DE SISTEMAS FLUIDODINÁMICOS EN ESTADO NO ESTACIONARIO MEDIANTE DEEP LEARNING

Estudiante Abucide, Armas, Álvaro
Director/Directora Zulueta, Guerrero, Ekaitz
Departamento Ingeniería de Sistemas y Automática
Curso académico 2022-2023

Bilbao, 10 de julio de 2023

Esquemas y programas fuente

A

A.1 Carga de archivos CSV e interpolación

```
import pickle
import matplotlib.pyplot as plt
from scipy.io import loadmat
import numpy as np
# from functions.create_dataX_function import create_dataX
# from functions.load_CSVfunc_TIME import loadCSV
# from functions.load_CSVfunc_fromHardDisc import loadCSV
from scipy.interpolate import griddata

import pandas as pd
from os import listdir
from pathlib import Path
import csv
from numba import njit

@njit
def visualize3(sample_y, s):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    fig.set_size_inches(15, 10)
    plt.suptitle("Time instant " + str(s+1), fontsize=20)
    plt.subplot(3, 1, 1)
    plt.title('CFD', fontsize=18)
```

```
    plt.imshow((sample_y[s, 0, :, :]), cmap='jet', vmin = minu,
    ↪  vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Ux', fontsize=18)

    plt.subplot(3, 1, 2)
    plt.imshow((sample_y[s, 1, :, :]), cmap='jet', vmin = minv,
    ↪  vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Uy', fontsize=18)


    plt.subplot(3, 1, 3)
    plt.imshow((sample_y[s, 2, :, :]), cmap='jet', vmin = minp,
    ↪  vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('p', fontsize=18)
    plt.tight_layout()
    plt.show()

def data_manip(dataY):

    Y=np.zeros([len(dataY),3,81,174])
    for i in range(len(dataY)):
        for j in range (3):
            points=dataY[i][:,4:6] #X e Y.
            values=dataY[i][:,j] #Pressure, velocity[i],
            ↪  velocity[j]

            # Interpolación:
            grid_x,
            ↪  grid_y=np.mgrid[min(points[:,0]):max(points[:,0]):174j,
            ↪  min(points[:,1]):max(points[:,1]):81j]
            data_interpolated=griddata(points, values, (grid_x,
            ↪  grid_y), method='linear')
            data_interpolated=np.moveaxis(data_interpolated,0,-1)
            Y[i,j,:,:]=data_interpolated
        print("Interpolación nº: " + str(i + 1))
    Y=Y[:,:,1:80,1:173]
    ## Cambiar el orden de pressure y velocities para que estén en
    ↪  la misma forma
    ## que y en la red neuronal.
```

```
    ## 1º Velocity[i], 2º Velocity[j], 3º Pressure.
    Y=Y[:,[1,2,0],:,:]

    return Y

if __name__=="__main__":
    dataY=[]
    for i in range (3000,5001,1):
        file =
        ↪ pd.read_csv('./Unai-ANN-Abucide-casos20-25-2alamenos4/'
        ↪ +                'XYZ_Internal_Table_Datos_25ms__' +
        ↪ str(i) +
        .csv', header=0)
        data_array = file.to_numpy()
        dataY.append(data_array)
        print ("Archivo nº" + str(i))
        print(file)
    with open('csv_25ms_3000_5000.pkl', 'wb') as f:
        pickle.dump(dataY, f)
```

A.2 Generar las matrices de SDF y FRC

```
function [px, py] = geometry_points_v2(geometry, aug_factor,
↪ size_factor, rotation_angle, position)
    switch geometry
        case "circle"
            rx = 10*172/(180);
            ry = 10*79/(180);
            a = 172/2; % Center in x
            b = 79/2; % Center in y
            th = 0:pi/50:2*pi;
            px = rx * cos(th) + a;
            py = ry * sin(th) + b;
        case "triangle"
            px = [89 85 98];
            py = [59 40 43];
        case "triangle_eq"
            px = [77 92 87];
            py = [37 34 46];
        case "ellipse"
```

```
            px = [77 77 78 78 78 79 80 81 82 83 84 85 86 87 88 89
            ↪  90 91 92 ...
                93 94 94 94 95 95 95 95 95 95 95 95 94 93 92 92 90
                ↪  89 88 ...
                87 86 85 84 83 83 82 81 80 80 79 79 78 78 78];
            py = [35 34 33 32 31 30 30 29 29 30 30 30 31 32 32 33
            ↪  34 35 ...
                36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 50 51
                ↪  51 50 ...
                50 50 49 48 48 47 46 45 44 43 42 41 40 39 38 37];
        case "square"
            px = [77 82 93 89];
            py = [42 32 36 46];
        case "rectangle"
            px = [78 84 96 87];
            py = [48 26 30 52];
        otherwise
            disp("Geometry not considered.")
    end

    % Cambiar la posición de la figura
    px = px + position(1);
    py = py + position(2);

    % Cambiar el tamaño de la figura para el data augmentation
    p = polyshape(px, py);
    [cx, cy] = centroid(p);
    p2 = scale(p, aug_factor, [cx, cy]);
    px = p2.Vertices(:,1)';
    py = p2.Vertices(:,2)';

    % Cambiar el tamaño de la figura
    p = polyshape(px, py);
    [cx, cy] = centroid(p);
    p2 = scale(p, size_factor, [cx, cy]);
    px = p2.Vertices(:,1)';
    py = p2.Vertices(:,2)';

    % Cambiar la orientación de la figura
    p = polyshape(px, py);
    [cx, cy] = centroid(p);
    p2 = rotate(p, rotation_angle, [cx, cy]);
```

```
    px = p2.Vertices(:,1)';
    py = p2.Vertices(:,2)';

end
```

```
geometries = ["circle", "ellipse", "rectangle", "square", "triangle", "triangle_eq"];
% aug_factor = linspace(0.93, 1.07, 50);
aug_factor = 1;
size_factor = 1;
rotation_angle = 1;
position = [1, 1; 1, 1];
for geom = 1:length(geometries)
    for aug = 1:length(aug_factor)
        for size = 1:length(size_factor)
            for rot_angle = 1:length(rotation_angle)
                for pos = 1:length(position)
                    [px, py] = geometry_points_v2(geometries(geom), aug_factor(aug), size_factor(size), ...
                        rotation_angle(rot_angle), position(pos,:));
                    Z_matrix = sdf(px, py)';
            % save("Datos para entrenamiento/SDF_geometrias/" + geometries(geom) + ".mat", "Z_matrix")
            % save("Datos para entrenamiento/SDF_geometrias_data_augmentation/Test/" + geometries(geom) + "_" + aug_factor(aug) + "size_factor" + size_factor(size) + "rot_angle" + rotation_angle(rot_angle) + "pos_x" + position(pos,1) + "pos_y" + position(pos,2) + ".mat", "Z_matrix")
                    frc = FRC_geometries(px, py)';
                    figure
                    imagesc(frc')
            % save("Datos para entrenamiento/FRC_geometrias/" + geometries(geom) + ".mat", "frc")
```

```
            % save("Datos para
            ↪  entrenamiento/FRC_geometrias_data_augmentation/Test/"
            ↪  + geometries(geom) + "_" + aug_factor(aug) +
            ↪  "size_factor" + size_factor(size) + "rot_angle" +
            ↪  rotation_angle(rot_angle) + "pos_x" +
            ↪  position(pos,1) + "pos_y" + position(pos,2) +
            ↪  ".mat", "frc")
                end
            end
        end
    end
end
```

A.3 *Data augmentation*

Generar los datos adicionales de las entradas de la CNN:

```
from scipy.io import loadmat
import numpy as np
from os import listdir
import pickle

def generate_dataX():

    SDF = []
    FRC = []
    filepaths = [f for f in
    ↪  listdir("SDF_geometrias_data_augmentation/Train") if
    ↪  f.endswith('.mat')]
    for file in filepaths:

        ↪  SDF.append(loadmat("SDF_geometrias_data_augmentation/Train/"
        ↪  + file)['Z_matrix'])
    filepaths = [f for f in
    ↪  listdir("FRC_geometrias_data_augmentation/Train") if
    ↪  f.endswith('.mat')]
    for file in filepaths:

        ↪  FRC.append(loadmat("FRC_geometrias_data_augmentation/Train/"
        ↪  + file)['frc'])
```

```
    SDF2 = pickle.load(open("dataX.pkl", "rb"))[len(filepaths), 2,
    ↪  :, :]

    dataX = np.zeros([len(filepaths), 3, 172, 79])

    dataX[:,0,:,:] = np.array(SDF)
    dataX[:,1,:,:] = np.array(FRC)
    dataX[:,2,:,:] = SDF2

    return dataX
```

Generar los datos adicionales de los campos de velocidades paralela al flujo y vertical y de presión:

```
import numpy as np
import pickle
from scipy.interpolate import griddata
from scipy.io import loadmat

geom_index = ["1", "63", "121", "723", "1305", "1923"]

# a = np.linspace(0.93, 1.07, 50)

a = loadmat("aug_factor.mat")["aug_factor"]
a = np.reshape(a, [50])

for geom in geom_index:
    dataY = pickle.load(open('E:/NuevasGeometrias/csv_geom' + geom
    ↪  + '_1001_1500.pkl', 'rb'))
    # Y=np.zeros([len(dataY)*len(a),3,172, 79])
    Y=np.zeros([len(a),3,172, 79])

    print("Geometry " + geom)

    for i in range(3):
        points=dataY[i][:,5:7] #X e Y.
        values_p=dataY[i][:,1] # Pressure (Pa)
        values_vx=dataY[i][:,2] # Velocity x (m/s)
        values_vy=dataY[i][:,3] # Velocity y (m/s)

        print("Interpolación nº : " + str(i + 1) + " out of " +
        ↪  str(len(dataY)))
```

```
219
220         for k in range(len(a)):
221         # for k in range(1):
222
223             grid_x_p, grid_y_p =
                ↪  np.mgrid[min(points[:,0]):max(points[:,0]):256j/a[k],
                ↪  min(points[:,1]):max(points[:,1]):128j/a[k]]
224             data_interpolated_p = griddata(points, values_p,
                ↪  (grid_x_p, grid_y_p), method='linear')/a[k]**2
225
226             print("Pressure data augmentation: " + str(k) + " out
                ↪  of " + str(len(a)))
227
228             grid_x_vx, grid_y_vx =
                ↪  np.mgrid[min(points[:,0]):max(points[:,0]):256j/a[k],
                ↪  min(points[:,1]):max(points[:,1]):128j/a[k]]
229             data_interpolated_vx = griddata(points, values_vx,
                ↪  (grid_x_vx, grid_y_vx), method='linear')/a[k]
230
231             print("Vx data augmentation: " + str(k) + " out of " +
                ↪  str(len(a)))
232
233             grid_x_vy, grid_y_vy =
                ↪  np.mgrid[min(points[:,0]):max(points[:,0]):256j/a[k],
                ↪  min(points[:,1]):max(points[:,1]):128j/a[k]]
234             data_interpolated_vy = griddata(points, values_vy,
                ↪  (grid_x_vy, grid_y_vy), method='linear')/a[k]
235
236             print("Vy data augmentation: " + str(k) + " out of " +
                ↪  str(len(a)))
237
238             points_b_p = np.array([grid_x_p.flatten(),
                ↪  grid_y_p.flatten()]).T
239             values_b_p = data_interpolated_p.flatten()
240
241             points_b_vx = np.array([grid_x_vx.flatten(),
                ↪  grid_y_vx.flatten()]).T
242             values_b_vx = data_interpolated_vx.flatten()
243
244             points_b_vy = np.array([grid_x_vy.flatten(),
                ↪  grid_y_vy.flatten()]).T
245             values_b_vy = data_interpolated_vy.flatten()
```

```
246
247             print("Pressure interpolation initialized")
248
249             grid_x_b_p, grid_y_b_p =
                ↪  np.mgrid[min(points_b_p[:,0]):max(points_b_p[:,0]):174j,
                ↪  min(points_b_p[:,1]):max(points[:,1]):81j]
250             data_interpolated_b_p = griddata(points_b_p, values_b_p,
                ↪  (grid_x_b_p, grid_y_b_p), method='linear')
251             data_interpolated_b_p =
                ↪  data_interpolated_b_p[1:173,1:80]
252
253             print("Pressure interpolation finished")
254             print("Vx interpolation initialized")
255
256             grid_x_b_vx, grid_y_b_vx =
                ↪  np.mgrid[min(points_b_vx[:,0]):max(points_b_vx[:,0]):174j,
                ↪  min(points_b_vx[:,1]):max(points[:,1]):81j]
257             data_interpolated_b_vx = griddata(points_b_vx,
                ↪  values_b_vx, (grid_x_b_p, grid_y_b_vx),
                ↪  method='linear')
258             data_interpolated_b_vx =
                ↪  data_interpolated_b_vx[1:173,1:80]
259
260             print("Vx interpolation finished")
261             print("Vy interpolation initialized")
262
263             grid_x_b_vy, grid_y_b_vy =
                ↪  np.mgrid[min(points_b_vy[:,0]):max(points_b_vy[:,0]):174j,
                ↪  min(points_b_vy[:,1]):max(points[:,1]):81j]
264             data_interpolated_b_vy = griddata(points_b_vy,
                ↪  values_b_vy, (grid_x_b_vy, grid_y_b_vy),
                ↪  method='linear')
265             data_interpolated_b_vy =
                ↪  data_interpolated_b_vy[1:173,1:80]
266
267             print("Vy interpolation finished")
268
269             Y[k, 0, :, :] = data_interpolated_b_vx
270             Y[k, 1, :, :] = data_interpolated_b_vy
271             Y[k, 2, :, :] = data_interpolated_b_p
272
```

```
    with open('dataY_' + geom + '_geometries_augmented_f.pkl',
    ↪  'wb') as f:
        pickle.dump(Y, f)

import numpy as np
import pickle
import random
from create_dataX_augmented import *

def randomize_CFD_input_data():

    y_circle =
    ↪  pickle.load(open("./dataY_1_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD
    y_ellipse =
    ↪  pickle.load(open("./dataY_1305_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD
    y_rectangle =
    ↪  pickle.load(open("./dataY_723_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD
    y_square =
    ↪  pickle.load(open("./dataY_63_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD
    y_triangle =
    ↪  pickle.load(open("./dataY_1923_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD
    y_triangle_eq =
    ↪  pickle.load(open("./dataY_121_geometries_augmented_e.pkl",
    ↪  "rb"))[:50] # CFD

    y_list = [y_circle, y_ellipse, y_rectangle, y_square,
    ↪  y_triangle, y_triangle_eq]
    length = len(y_circle) + len(y_ellipse) + len(y_rectangle) +
    ↪  len(y_square) + len(y_triangle) + len(y_triangle_eq)
    length_list = [len(y_circle), len(y_ellipse), len(y_rectangle),
    ↪  len(y_square), len(y_triangle), len(y_triangle_eq)]
    length1 = len(y_circle)
    length2 = length1 + len(y_ellipse)
    length3 = length2 + len(y_rectangle)
    length4 = length3 + len(y_square)
    length5 = length4 + len(y_triangle)
    print("Long. 1: ", length1)
```

```
    print("Long. 2: ", length2)
    print("Long. 3: ", length3)
    print("Long. 4: ", length4)
    print("Long. 5: ", length5)

    print("Tamaño total: ", length)

    geometries = ["circle", "ellipse", "rectangle", "square",
    ↪  "triangle", "triangle_eq"]

    data_x = generate_dataX()

    lengths = [length1, length2, length3, length4, length5]

    x = np.concatenate((data_x[:length1,:,:,:],
                        data_x[length1:length2,:,:,:],
                        data_x[length2:length3,:,:,:],
                        data_x[length3:length4,:,:,:],
                        data_x[length4:length5,:,:,:],
                        data_x[length5:,:,:,:]))


    y = np.zeros([length, 3, 172, 79])
    y[:length1,:,:,:] = y_list[0]
    y[length1:length2,:,:,:] = y_list[1]
    y[length2:length3,:,:,:] = y_list[2]
    y[length3:length4,:,:,:] = y_list[3]
    y[length4:length5,:,:,:] = y_list[4]
    y[length5:,:,:,:] = y_list[5]

    return y, x
```

A.4 Generar los datos de entrenamiento

Código que organiza los datos de entrada y salida para el entrenamiento de la CNN:

```
import numpy as np
import pickle
import random
from create_dataX import *
```

```
332
333 def randomize_CFD_input_data(variable, n_test):
334     velocities = [5, 10, 15, 20, 25]
335
336     y_circle = pickle.load(open("./Y_geom1_interpolated.pkl",
    ↪  "rb")) # CFD
337     y_ellipse = pickle.load(open("./Y_geom1305_interpolated.pkl",
    ↪  "rb")) # CFD
338     y_rectangle = pickle.load(open("./Y_geom723_interpolated.pkl",
    ↪  "rb")) # CFD
339     y_square = pickle.load(open("./Y_geom63_interpolated.pkl",
    ↪  "rb")) # CFD
340     y_triangle = pickle.load(open("./Y_geom1923_interpolated.pkl",
    ↪  "rb")) # CFD
341     y_triangle_eq = pickle.load(open("./Y_geom121_interpolated.pkl",
    ↪  "rb")) # CFD
342
343     y_list = [y_circle, y_ellipse, y_rectangle, y_square,
    ↪  y_triangle, y_triangle_eq]
344     length = len(y_circle) + len(y_ellipse) + len(y_rectangle) +
    ↪  len(y_square) + len(y_triangle) + len(y_triangle_eq)
345     length_list = [len(y_circle), len(y_ellipse), len(y_rectangle),
    ↪  len(y_square), len(y_triangle), len(y_triangle_eq)]
346     length1 = len(y_circle)-1
347     length2 = length1 + len(y_ellipse) - 1
348     length3 = length2 + len(y_rectangle) - 1
349     length4 = length3 + len(y_square) - 1
350     length5 = length4 + len(y_triangle) - 1
351     print("Long. 1: ", length1)
352     print("Long. 2: ", length2)
353     print("Long. 3: ", length3)
354     print("Long. 4: ", length4)
355     print("Long. 5: ", length5)
356
357     length_list_b = [length1, length2, length3, length4, length5]
358
359     # y = np.zeros([length-5, 1, 172, 79])
360     print("Tamaño total: ", length)
361
362     geometries = ["circle", "ellipse", "rectangle", "square",
    ↪  "triangle", "triangle_eq"]
363
```

```
    data_x = generate_dataX(geometries)

    x=np.zeros([length,2,79,172])
    x[:length1+1,:,:,:]=data_x[0]
    x[length1+1:length2+2,:,:,:]=data_x[1]
    x[length2+2:length3+3,:,:,:]=data_x[2]
    x[length3+3:length4+4,:,:,:]=data_x[3]
    x[length4+4:length5+5,:,:,:]=data_x[4]
    x[length5+5:,:,:,:]=data_x[5]

    x_new = np.zeros([length-6,3,79,172])
    y_new = np.zeros([length-6,1,79,172])

    y_new[:length1], x_new[:length1] = add_t_1_instant(y_list[0],
    ↪  x[:length1+1], variable)
    y_new[length1:length2], x_new[length1:length2] =
    ↪  add_t_1_instant(y_list[1], x[length1+1:length2+2],
    ↪  variable)
    y_new[length2:length3], x_new[length2:length3] =
    ↪  add_t_1_instant(y_list[2], x[length2+2:length3+3],
    ↪  variable)
    y_new[length3:length4], x_new[length3:length4] =
    ↪  add_t_1_instant(y_list[3], x[length3+3:length4+4],
    ↪  variable)
    y_new[length4:length5], x_new[length4:length5] =
    ↪  add_t_1_instant(y_list[4], x[length4+4:length5+5],
    ↪  variable)
    y_new[length5:], x_new[length5:] = add_t_1_instant(y_list[5],
    ↪  x[length5+5:], variable)

    test_index_list = []
    y_train = y_new
    x_train = x_new
    for i in range(n_test):
        y_train, x_train, test_index, length_list_b =
        ↪  test_samples(y_train, x_train, length_list,
        ↪  length_list_b)
        test_index_list.append(test_index)
        print("Test nº: ", i+1)
    return y_train, x_train, y_new, x_new, test_index_list,
    ↪  length_list
```

```
393 def add_t_1_instant(y, x, variable):
394     print("Tamaño original: ", y.shape)
395     if variable == "vx":
396             y2=y[:,:1,:,:] # Select the variable to simulate vx(:1),
                ↪  vy(1:2), p(2:3)
397     elif variable == "vy":
398             y2=y[:,1:2,:,:] # Select the variable to simulate
                ↪  vx(:1), vy(1:2), p(2:3)
399     else:
400             y2=y[:,2:3,:,:] # Select the variable to simulate
                ↪  vx(:1), vy(1:2), p(2:3)
401     print("Tamaño tras selección de la variable: ", y2.shape)
402     y=y2
403     x2=np.zeros([len(y),3,79,172])
404     x2[:,:2,:,:]=x
405     # y=np.moveaxis(y,2,3)
406     x2[:,2:3,:,:]=y # Add the t-1 instant to the input data
407     x=x2[:-1,:,:,:]
408     y=y[1:,:,:,:]
409     print("Tamaño tras eliminar una muestra: ", y.shape)
410     return(y, x)
411
412 def test_samples(y_train, x_train, length_list, length_list_b):
413
414     test_index = []
415
416     random_value_circle = (random.randint(0, length_list[0]-50))
417     random_value_ellipse = (random.randint(0, length_list[1]-50))
418     random_value_rectangle = (random.randint(0, length_list[2]-50))
419     random_value_square = (random.randint(0, length_list[3]-50))
420     random_value_triangle = (random.randint(0, length_list[4]-50))
421     random_value_triangle_eq = (random.randint(0,
        ↪  length_list[5]-50))
422
423
424     test_index = [random_value_circle, random_value_ellipse,
        ↪  random_value_rectangle,
425                   random_value_square, random_value_triangle,
                      ↪  random_value_triangle_eq]
426     y_train = np.delete(y_train, np.add(test_index[1:],
        ↪  length_list_b), 0)
427     print("length y: ", y_train.shape)
```

```
    x_train = np.delete(x_train, np.add(test_index[1:],
    ↪ length_list_b), 0)
    print("length x: ", x_train.shape)
    length_list_b = np.add(length_list_b, [-1 -2 -3 -4 -5])
    # length_list_b = length_list_b.to_list()
    print("Lista de longitudes de array:", length_list_b)
    return y_train, x_train, test_index, length_list_b
```

```
# from random_samples_function import *
from data_generation_for_CNN_first_sample_data_augmentation import
↪ *

# y, x, y_all, x_all, test_index_list, length_list =
↪ randomize_CFD_input_data("p",50)
y, x = randomize_CFD_input_data()


with open("training_test_data_geometries_first_sample_f.pkl", "wb")
↪ as file:
    # pickle.dump([y, x, y_all, x_all, test_index_list,
    ↪ length_list], file)
    pickle.dump([y, x], file)
```

A.5 CNN

El modelo de la CNN viene dado por los siguientes códigos:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils import weight_norm

def create_layer(in_channels, out_channels, kernel_size, wn=True,
↪ bn=True,
                 activation=nn.ReLU, convolution=nn.Conv2d):
    assert kernel_size % 2 == 1
    layer = []
    conv = convolution(in_channels, out_channels, kernel_size,
    ↪ padding=kernel_size // 2)
    if wn:
        conv = weight_norm(conv)
    layer.append(conv)
```

```
457     if activation is not None:
458         layer.append(activation())
459     if bn:
460         layer.append(nn.BatchNorm2d(out_channels))
461     return nn.Sequential(*layer)
462
463
464 class AutoEncoder(nn.Module):
465     def __init__(self, in_channels, out_channels, kernel_size=3,
        ↪  filters=[16, 32, 64],
466                  weight_norm=True, batch_norm=True,
                     ↪  activation=nn.ReLU, final_activation=None):
467         super().__init__()
468         assert len(filters) > 0
469         encoder = []
470         decoder = []
471         for i in range(len(filters)):
472             if i == 0:
473                 encoder_layer = create_layer(in_channels,
                    ↪  filters[i], kernel_size, weight_norm,
                    ↪  batch_norm, activation, nn.Conv2d)
474                 decoder_layer = create_layer(filters[i],
                    ↪  out_channels, kernel_size, weight_norm, False,
                    ↪  final_activation, nn.ConvTranspose2d)
475             else:
476                 encoder_layer = create_layer(filters[i-1],
                    ↪  filters[i], kernel_size, weight_norm,
                    ↪  batch_norm, activation, nn.Conv2d)
477                 decoder_layer = create_layer(filters[i],
                    ↪  filters[i-1], kernel_size, weight_norm,
                    ↪  batch_norm, activation, nn.ConvTranspose2d)
478             encoder = encoder + [encoder_layer]
479             decoder = [decoder_layer] + decoder
480         self.encoder = nn.Sequential(*encoder)
481         self.decoder = nn.Sequential(*decoder)
482
483     def forward(self, x):
484         return self.decoder(self.encoder(x))
```

```
485     import torch
486 import torch.nn as nn
487 import torch.nn.functional as F
```

```
from torch.nn.utils import weight_norm
from Models.AutoEncoder import create_layer


def create_encoder_block(in_channels, out_channels, kernel_size, wn=True, bn=True,
                         activation=nn.ReLU, layers=2):
    encoder = []
    for i in range(layers):
        _in = out_channels
        _out = out_channels
        if i == 0:
            _in = in_channels
        encoder.append(create_layer(_in, _out, kernel_size, wn, bn, activation, nn.Conv2d))
    return nn.Sequential(*encoder)

def create_decoder_block(in_channels, out_channels, kernel_size, wn=True, bn=True,
                         activation=nn.ReLU, layers=2, final_layer=False):
    decoder = []
    for i in range(layers):
        _in = in_channels
        _out = in_channels
        _bn = bn
        _activation = activation
        if i == 0:
            _in = in_channels * 2
        if i == layers - 1:
            _out = out_channels
            if final_layer:
                _bn = False
                _activation = None
        decoder.append(create_layer(_in, _out, kernel_size, wn, _bn, _activation, nn.ConvTranspose2d))
    return nn.Sequential(*decoder)


def create_encoder(in_channels, filters, kernel_size, wn=True, bn=True, activation=nn.ReLU, layers=2):
    encoder = []
    for i in range(len(filters)):
```

```
525         if i == 0:
526             encoder_layer = create_encoder_block(in_channels,
                ↪  filters[i], kernel_size, wn, bn, activation,
                ↪  layers)
527         else:
528             encoder_layer = create_encoder_block(filters[i-1],
                ↪  filters[i], kernel_size, wn, bn, activation,
                ↪  layers)
529         encoder = encoder + [encoder_layer]
530     return nn.Sequential(*encoder)
531
532 def create_decoder(out_channels, filters, kernel_size, wn=True,
    ↪  bn=True, activation=nn.ReLU, layers=2):
533     decoder = []
534     for i in range(len(filters)):
535         if i == 0:
536             decoder_layer = create_decoder_block(filters[i],
                ↪  out_channels, kernel_size, wn, bn, activation,
                ↪  layers, final_layer=True)
537         # elif i == len(filters):
538         #     decoder_layer = create_decoder_block(velocity_channel,
            ↪  filters[i], kernel_size, wn, bn, activation, layers,
            ↪  final_layer=True)
539         else:
540             decoder_layer = create_decoder_block(filters[i],
                ↪  filters[i-1], kernel_size, wn, bn, activation,
                ↪  layers, final_layer=False)
541         decoder = [decoder_layer] + decoder
542     return nn.Sequential(*decoder)
543
544 class UNetEx(nn.Module):
545     def __init__(self, in_channels, out_channels, kernel_size=3,
        ↪  filters=[16, 32, 64], layers=3,
546          weight_norm=True, batch_norm=True, activation=nn.ReLU,
             ↪  final_activation=None):
547         super().__init__()
548         assert len(filters) > 0
549         self.final_activation = final_activation
550         self.encoder = create_encoder(in_channels, filters,
            ↪  kernel_size, weight_norm, batch_norm, activation,
            ↪  layers)
551         decoders = []
```

```
        for i in range(out_channels):
            decoders.append(create_decoder(1, filters, kernel_size,
            ↪  weight_norm, batch_norm, activation, layers))
        self.decoders = nn.Sequential(*decoders)

    def encode(self, x):
        tensors = []
        indices = []
        sizes = []
        for encoder in self.encoder:
            x = encoder(x)
            sizes.append(x.size())
            tensors.append(x)
            x, ind = F.max_pool2d(x, 2, 2, return_indices=True)
            indices.append(ind)
        return x, tensors, indices, sizes

    def decode(self, _x, _tensors, _indices, _sizes):
        y = []
        for _decoder in self.decoders:
            x = _x
            tensors = _tensors[:]
            indices = _indices[:]
            sizes = _sizes[:]
            for decoder in _decoder:
                tensor = tensors.pop()
                size = sizes.pop()
                ind = indices.pop()
                x = F.max_unpool2d(x, ind, 2, 2, output_size=size)
                x = torch.cat([tensor, x], dim=1)
                x = decoder(x)
            y.append(x)
        return torch.cat(y, dim=1)

    def forward(self, x):
        x, tensors, indices, sizes = self.encode(x)
        x = self.decode(x, tensors, indices, sizes)
        if self.final_activation is not None:
            x = self.final_activation(x)
        return x
```

Funciones para el entrenamiento de la red neuronal:

```
import copy
import torch

def generate_metrics_list(metrics_def):
    list = {}
    for name in metrics_def.keys():
        list[name] = []
    return list

def epoch(scope, loader, on_batch=None, training=False):
    model = scope["model"]
    optimizer = scope["optimizer"]
    loss_func = scope["loss_func"]
    metrics_def = scope["metrics_def"]
    scope = copy.copy(scope)
    scope["loader"] = loader

    metrics_list = generate_metrics_list(metrics_def)
    total_loss = 0
    if training:
        model.train()
    else:
        model.eval()
    for tensors in loader:
        if "process_batch" in scope and scope["process_batch"] is
        ↪  not None:
            tensors = scope["process_batch"](tensors)
        if "device" in scope and scope["device"] is not None:
            tensors = [tensor.to(scope["device"]) for tensor in
            ↪  tensors]
        loss, output = loss_func(model, tensors)
        if training:
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        total_loss += loss.item()
        scope["batch"] = tensors
        scope["loss"] = loss
        scope["output"] = output
        scope["batch_metrics"] = {}
        for name, metric in metrics_def.items():
            value = metric["on_batch"](scope)
```

```
            scope["batch_metrics"][name] = value
            metrics_list[name].append(value)
        if on_batch is not None:
            on_batch(scope)
    scope["metrics_list"] = metrics_list
    metrics = {}
    for name in metrics_def.keys():
        scope["list"] = scope["metrics_list"][name]
        metrics[name] = metrics_def[name]["on_epoch"](scope)
    return total_loss, metrics

def train(scope, train_dataset, val_dataset, patience=10,
 ↪  batch_size=256, print_function=print, eval_model=None,
          on_train_batch=None, on_val_batch=None,
           ↪  on_train_epoch=None, on_val_epoch=None,
           ↪  after_epoch=None):
    epochs = scope["epochs"]
    model = scope["model"]
    metrics_def = scope["metrics_def"]
    scope = copy.copy(scope)

    scope["best_train_metric"] = None
    scope["best_train_loss"] = float("inf")
    scope["best_val_metrics"] = None
    scope["best_val_loss"] = float("inf")
    scope["best_model"] = None

    train_loader = torch.utils.data.DataLoader(train_dataset,
     ↪  batch_size=batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(val_dataset,
     ↪  batch_size=batch_size, shuffle=False)
    skips = 0
    for epoch_id in range(1, epochs + 1):
        scope["epoch"] = epoch_id
        print_function("Epoch #" + str(epoch_id))
        # Training
        scope["dataset"] = train_dataset
        train_loss, train_metrics = epoch(scope, train_loader,
         ↪  on_train_batch, training=True)
        scope["train_loss"] = train_loss
        scope["train_metrics"] = train_metrics
        print_function("\tTrain Loss = " + str(train_loss))
```

```
667         for name in metrics_def.keys():
668             print_function("\tTrain " + metrics_def[name]["name"] +
                ↪  " = " + str(train_metrics[name]))
669         if on_train_epoch is not None:
670             on_train_epoch(scope)
671         del scope["dataset"]
672         # Validation
673         scope["dataset"] = val_dataset
674         with torch.no_grad():
675             val_loss, val_metrics = epoch(scope, val_loader,
                ↪  on_val_batch, training=False)
676         scope["val_loss"] = val_loss
677         scope["val_metrics"] = val_metrics
678         print_function("\tValidation Loss = " + str(val_loss))
679         for name in metrics_def.keys():
680             print_function("\tValidation " +
                ↪  metrics_def[name]["name"] + " = " +
                ↪  str(val_metrics[name]))
681         if on_val_epoch is not None:
682             on_val_epoch(scope)
683         del scope["dataset"]
684         # Selection
685         is_best = None
686         if eval_model is not None:
687             is_best = eval_model(scope)
688         if is_best is None:
689             is_best = val_loss < scope["best_val_loss"]
690         if is_best:
691             scope["best_train_metric"] = train_metrics
692             scope["best_train_loss"] = train_loss
693             scope["best_val_metrics"] = val_metrics
694             scope["best_val_loss"] = val_loss
695             scope["best_model"] = copy.deepcopy(model)
696             print_function("Model saved!")
697             skips = 0
698         else:
699             skips += 1
700         if after_epoch is not None:
701             after_epoch(scope)
702
703     return scope["best_model"], scope["best_train_metric"],
    ↪  scope["best_train_loss"],\
```

```
            scope["best_val_metrics"], scope["best_val_loss"]

def train_model(model, loss_func, train_dataset, val_dataset,
    optimizer, process_batch=None, eval_model=None,
                on_train_batch=None, on_val_batch=None,
                    on_train_epoch=None, on_val_epoch=None,
                    after_epoch=None,
                epochs=100, batch_size=256, patience=10, device=0,
                    **kwargs):
    model = model.to(device)
    scope = {}
    scope["model"] = model
    scope["loss_func"] = loss_func
    scope["train_dataset"] = train_dataset
    scope["val_dataset"] = val_dataset
    scope["optimizer"] = optimizer
    scope["process_batch"] = process_batch
    scope["epochs"] = epochs
    scope["batch_size"] = batch_size
    scope["device"] = device
    metrics_def = {}
    names = []
    for key in kwargs.keys():
        parts = key.split("_")
        if len(parts) == 3 and parts[0] == "m":
            if parts[1] not in names:
                names.append(parts[1])
    for name in names:
        if "m_" + name + "_name" in kwargs and "m_" + name +
            "_on_batch" in kwargs and "m_" + name + "_on_epoch" in
            kwargs:
            metrics_def[name] = {
                "name": kwargs["m_" + name + "_name"],
                "on_batch": kwargs["m_" + name + "_on_batch"],
                "on_epoch": kwargs["m_" + name + "_on_epoch"],
            }
        else:
            print("Warning: " + name + " metric is incomplete!")
    scope["metrics_def"] = metrics_def
    return train(scope, train_dataset, val_dataset,
    eval_model=eval_model, on_train_batch=on_train_batch,
```

```
            on_val_batch=on_val_batch, on_train_epoch=on_train_epoch,
            ↪  on_val_epoch=on_val_epoch, after_epoch=after_epoch,
            batch_size=batch_size, patience=patience)
```

Funciones necesarias para la ejecución del código de la red neuronal.

```
import torch.nn as nn
import numpy as np
from matplotlib import pyplot as plt

def split_tensors(*tensors, ratio):
    assert len(tensors) > 0
    split1, split2 = [], []
    count = len(tensors[0])
    for tensor in tensors:
        assert len(tensor) == count
        split1.append(tensor[:int(len(tensor) * ratio)])
        split2.append(tensor[int(len(tensor) * ratio):])
    if len(tensors) == 1:
        split1, split2 = split1[0], split2[0]
    return split1, split2

def initialize(model, gain=1, std=0.02):
    for module in model.modules():
        if type(module) in [nn.Linear, nn.Conv1d, nn.Conv2d,
        ↪  nn.Conv3d]:
            nn.init.xavier_normal_(module.weight, gain)
            if module.bias is not None:
                nn.init.normal_(module.bias, 0, std)

def visualize(sample_y, out_y, error, s, variable, n_sample,
↪  velocity):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    plt.figure()
    fig = plt.gcf()
```

```
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
    ↪  + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
    ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel(variable, fontsize=18)
    plt.subplot(3, 3, 2)
    plt.title('CNN', fontsize=18)
    plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
    ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 3, 3)
    plt.title('Error', fontsize=18)
    plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
    ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.tight_layout()
    plt.show()
```

Código para el entrenamiento de la red que predice los instantes futuros con velocidades de entrada variables:

```
import os
import json
import torch
import pickle
import time
from train_functions import *
from functions import *
import torch.optim as optim
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx
from Models.UNet import UNet

import numpy as np
from create_channels_weights_1variable import
↪  create_channels_weights
from random_samples_function import *

```

```
import random

import gc

# torch.cuda.empty_cache()
# del variables
# gc.collect()
epoch = [1000]
___batchsize = [32, 64]
learning_rate = [0.0001]
kernelsize = [3, 5, 7]
variable = ["vx", "vy", "p"]
for vb in variable:
    cont = 25
    for bs in range(len(batchsize)):
        for l_r in range(len(learning_rate)):
            for ep in range(len(epoch)):
                for ks in range(len(kernelsize)):
                    device = torch.device("cuda" if
                    ↪  torch.cuda.is_available() else "cpu")
                    with open('training_test_data_' + vb + '.pkl',
                    ↪  'rb') as f:
                        y, x, y_all, x_all, test_index_list,
                        ↪  length_list = pickle.load(f)

                    # Remove the possible NaN data
                    # y_b=y
                    # y[np.isnan(y)] = 0
                    # x[np.isnan(x)] = 0



                    x = torch.FloatTensor(x)
                    y = torch.FloatTensor(y)
                    # y_b = torch.FloatTensor(y_b)

                    # Initial weights calculation
                    # channels_weights =
                    ↪  torch.sqrt(torch.mean(y.permute(0, 2, 3,
                    ↪  1).reshape((len(y)*172*79,1)) ** 2,
                    ↪  dim=0)).view(1, -1, 1, 1).to(device)
                    channels_weights=create_channels_weights(y,
                    ↪  device)
```

```
841                 print("Channel weights: ", channels_weights)
842
843                 # Simulation files directory
844                 simulation_directory = "./Run/"
845                 if not os.path.exists(simulation_directory):
846                     os.makedirs(simulation_directory)
847
848                 train_data, test_data = split_tensors(x, y,
                    ↪  ratio=0.7)
849
850                 train_dataset, test_dataset =
                    ↪  TensorDataset(*train_data),
                    ↪  TensorDataset(*test_data)
851                 test_x, test_y = test_dataset[:]
852
853                 # Parameters
854                 torch.manual_seed(0)
855                 lr = learning_rate[l_r]
856                 kernel_size = kernelsize[ks]
857                 filters = [8, 16, 32, 32]
858                 bn = False
859                 wn = False
860                 wd = 0.005
861                 beta1 = 0.5
862                 beta2 = 0.5
863
864                 model = UNetEx(3, 1, filters=filters,
                    ↪  kernel_size=kernel_size, batch_norm=bn,
                    ↪  weight_norm=wn)
865                 # model.load_state_dict(torch.load(
                    ↪  "Modelo_vx_Final.py")) #Cargamos los pesos
                    ↪  anteriores
866
867                 # Define optimizer
868                 optimizer =
                    ↪  torch.optim.AdamW(model.parameters(), lr=lr,
                    ↪  weight_decay=wd)
869
870                 config = {}
871                 train_loss_curve = []
872                 test_loss_curve = []
873                 train_ux_curve = []
```

```
test_ux_curve = []

def after_epoch(scope):

    train_loss_curve.append(scope["train_loss"])
    test_loss_curve.append(scope["val_loss"])
    train_ux_curve.append(scope[
    "train_metrics"]["ux"])
    test_ux_curve.append(scope[
    "val_metrics"]["ux"])

def loss_func(model, batch):
    x, y = batch
    output = model(x)

    lossu = torch.abs((output[:,0,:,:] -
    y[:,0,:,:])).reshape((output.shape[0],1,
    output.shape[2],output.shape[3]))
    loss = lossu / channels_weights
    return torch.sum(loss), output

ti = time.time()

# Training model
DeepCFD, train_metrics, train_loss,
test_metrics, test_loss = train_model(model,
loss_func, train_dataset, test_dataset,
optimizer,
    epochs=epoch[ep], batch_size=batchsize[ep],
    device=device,
    m_ux_name=vb + "MSE",
    m_ux_on_batch=lambda scope:
    float(torch.sum(scope["output"][:,0,:,:]
    - scope["batch"][1][:,0,:,:])),
    m_ux_on_epoch=lambda scope:
    sum(scope["list"]) /
    len(scope["dataset"]),
    patience=25, after_epoch=after_epoch)

duration = time.time() - ti
```

```
print("Training time: " + str(duration//60) +
↪  "min and " +
↪  str("{:.2f}".format(duration%60)) + "sec")

# Guardar los el modelo
torch.save(model.state_dict(),"Modelo_" + vb +
↪  str(cont) + ".py")

metrics = {}
metrics["train_metrics"] = train_metrics
metrics["train_loss"] = train_loss
metrics["test_metrics"] = test_metrics
metrics["test_loss"] = test_loss
curves = {}
curves["train_loss_curve"] = train_loss_curve
curves["test_loss_curve"] = test_loss_curve
curves["train_ux_curve"] = train_ux_curve
curves["test_ux_curve"] = test_ux_curve
config["metrics"] = metrics
config["curves"] = curves

net_param = {}
net_param["num_epochs"] = epoch[ep]
net_param["learning_ratio"] =
↪  learning_rate[l_r]
net_param["kernel_size"] = kernelsize[ks]
net_param["batch_size"] = batchsize[bs]
config["net_param"] = net_param
config["training_duration"] = duration

velocity = ["05", "10", "15", "20", "25"] # The
↪  simulated velocity
variable = vb # Simulated variable
with open(simulation_directory + "results_ms_"
↪  + vb + str(cont) + ".json", "w") as file:
    json.dump([config, duration, net_param],
    ↪  file)


# Test 5ms
y_5ms = y_all[:length_list[0]]
x_5ms = x_all[:length_list[0]]
```

```
# Test 10ms
y_10ms =
↪  y_all[length_list[0]:sum(length_list[:2])]
x_10ms =
↪  x_all[length_list[0]:sum(length_list[:2])]
# Test 15ms
y_15ms =
↪  y_all[sum(length_list[:2]):sum(length_list[:3])]
x_15ms =
↪  x_all[sum(length_list[:2]):sum(length_list[:3])]
# Test 20ms
y_20ms =
↪  y_all[sum(length_list[:3]):sum(length_list[:4])]
x_20ms =
↪  x_all[sum(length_list[:3]):sum(length_list[:4])]
# Test 25ms
y_25ms = y_all[sum(length_list[:4]):]
x_25ms = x_all[sum(length_list[:4]):]


def test_CNN(n_instantes, x_ms, y_ms,
↪  simulation_directory, velocity, variable,
↪  n_test, vb, cont):

    ti = time.time()

    out = torch.empty((n_instantes,3,172,79))
    x_ms = torch.FloatTensor(x_ms)
    y_ms = torch.FloatTensor(y_ms)
    out[:,0:2,:,:]=x_ms[0,:2,:,:]

    out[0,2,:,:] =
    ↪  DeepCFD(x_ms[n_test:n_test+1].to(device))

    for i in range (1,n_instantes,1):
        print("Sample: " + str(i))
        out[i,2,:,:] =
        ↪  DeepCFD(out[i-1:i].to(device))
        error =
        ↪  (torch.abs((out[:,2:3,:,:].cpu() -
        ↪  y_ms[n_test:n_test+n_instantes].cpu())))
    out2=out[:,2:3,:,:]
```

```
            out2=out2.reshape([n_instantes,1,172,79])

            duration = time.time() - ti
            print("Training time: " + str(duration//60)
            ↪ + "min and " +
            ↪ str("{:.2f}".format(duration\%60)) +
            ↪ "sec")

            out = out2.cpu().detach().numpy()
            error = error.cpu().detach().numpy()
            y = y_ms[ n_test:n_test+n_instantes]
            ↪ .cpu().detach().numpy()

            with open(simulation_directory + "plots_" +
            ↪ velocity + "ms_" + vb + str(cont) +
            ↪ ".pkl", "wb") as file:
                pickle.dump([out, error, y], file)

        n_instantes = 50
        for test in range(len(test_index_list)):
            test_CNN(n_instantes, x_5ms, y_5ms,
            ↪ simulation_directory, velocity[0],
                    variable, test_index_list[test][0],
                    ↪ vb, cont)
            test_CNN(n_instantes, x_10ms, y_10ms,
            ↪ simulation_directory, velocity[1],
                    variable, test_index_list[test][1],
                    ↪ vb, cont)
            test_CNN(n_instantes, x_15ms, y_15ms,
            ↪ simulation_directory, velocity[2],
                    variable, test_index_list[test][2],
                    ↪ vb, cont)
            test_CNN(n_instantes, x_20ms, y_20ms,
            ↪ simulation_directory, velocity[3],
                    variable, test_index_list[test][3],
                    ↪ vb, cont)
            test_CNN(n_instantes, x_25ms, y_25ms,
            ↪ simulation_directory, velocity[4],
                    variable, test_index_list[test][4],
                    ↪ vb, cont)

        cont = cont+1
```

Código para el entrenamiento de la red que predice los instantes futuros para geometrías variables:

```
import os
import json
import torch
import pickle
import time
from train_functions import *
from functions import *
import torch.optim as optim
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx
from Models.UNet import UNet

import numpy as np
from create_channels_weights_1variable import
 ↪  create_channels_weights
from random_samples_function import *

import random

# import gc

# torch.cuda.empty_cache()
# del variables
# gc.collect()

epoch = [2000] # 1000
batchsize = [32, 64, 128] # 32
learning_rate = [0.001, 0.0001] # 0.0001
kernelsize = [3, 5, 7] # 3
variable = ["vy", "p"]

# epoch = [1000]
# batchsize = [32]
# learning_rate = [0.001]
# kernelsize = [3]
# variable = ["vx"]

for vb in variable:
    cont = 200
```

```
    for bs in range(len(batchsize)):
        for l_r in range(len(learning_rate)):
            for ep in range(len(epoch)):
                for ks in range(len(kernelsize)):
                    a = variable.index(vb) + bs + l_r + ep + ks
                    # if cont<20:
                    #     cont = cont + 1

                    # else:
                    device = torch.device("cuda" if
                    ↪  torch.cuda.is_available() else "cpu")
                    with open('training_test_data_geometries_' + vb
                    ↪  + '.pkl', 'rb') as f:
                        y, x, y_all, x_all, test_index_list,
                        ↪  length_list = pickle.load(f)

                    print(ep, l_r, ks)

                    # y = y[:-450]
                    # x = x[:-450]

                    # Remove the possible NaN data
                    # y_b=y
                    # y[np.isnan(y)] = 0
                    # x[np.isnan(x)] = 0

                    # Function that scales data in the (0, 1)
                    ↪  interval
                    def minmax_norm(df):
                        return (df - df.min()) / ( df.max() -
                        ↪  df.min())

                    # x=minmax_norm(x)
                    # y=minmax_norm(y)

                    x = torch.FloatTensor(x)
                    y = torch.FloatTensor(y)
                    # y_b = torch.FloatTensor(y_b)

                    # Initial weights calculation
```

```
# channels_weights =
↪  torch.sqrt(torch.mean(y.permute(0, 2, 3,
↪  1).reshape((len(y)*172*79,1)) ** 2,
↪  dim=0)).view(1, -1, 1, 1).to(device)
channels_weights=create_channels_weights(y,
↪  device)
print("Channel weights: ", channels_weights)

# Simulation files directory
simulation_directory = "./Run/"
if not os.path.exists(simulation_directory):
    os.makedirs(simulation_directory)

plots_directory = "Run/Plots2/"
if not os.path.exists(plots_directory):
    os.makedirs(plots_directory)

plots_subdirectory = "Run/Plots2/" + vb +
↪  str(cont) + "/"
if not os.path.exists(plots_subdirectory):
    os.makedirs(plots_subdirectory)

modelo_directory = "./Modelo2/"
if not os.path.exists(modelo_directory):
    os.makedirs(modelo_directory)
# Spliting dataset into 95% train and 5% test
train_data, test_data = split_tensors(x, y,
↪  ratio=0.9)

train_dataset, test_dataset =
↪  TensorDataset(*train_data),
↪  TensorDataset(*test_data)
test_x, test_y = test_dataset[:]

# Parameters
torch.manual_seed(0)
lr = learning_rate[l_r]
kernel_size = kernelsize[ks]
filters = [8, 16, 32, 32]
bn = False
wn = False
wd = 0.005
```

```
beta1 = 0.5
beta2 = 0.5

model = UNetEx(3, 1, filters=filters,
↪  kernel_size=kernel_size, batch_norm=bn,
↪  weight_norm=wn)
#
↪  model.load_state_dict(torch.load("Modelo_vx_Final.py"))#Ca
↪  los pesos anteriores

# Define opotimizer
optimizer =
↪  torch.optim.AdamW(model.parameters(), lr=lr,
↪  weight_decay=wd)

config = {}
train_loss_curve = []
test_loss_curve = []
train_ux_curve = []
test_ux_curve = []

def after_epoch(scope):

    ↪  train_loss_curve.append(scope["train_loss"])
    test_loss_curve.append(scope["val_loss"])

    ↪  train_ux_curve.append(scope["train_metrics"]["ux"])

    ↪  test_ux_curve.append(scope["val_metrics"]["ux"])

def loss_func(model, batch):
    x, y = batch
    output = model(x)

    lossu = torch.abs((output[:,0,:,:] -
    ↪  y[:,0,:,:])).reshape((output.shape[0],1,output.shape[2
    loss = lossu / channels_weights
    return torch.sum(loss), output

ti = time.time()

# Training model
```

```
1131                    DeepCFD, train_metrics, train_loss,
                        ↪ test_metrics, test_loss = train_model(model,
                        ↪ loss_func, train_dataset, test_dataset,
                        ↪ optimizer,
1132                        epochs=epoch[ep], batch_size=batchsize[bs],
                            ↪ device=device,
1133                        m_ux_name="Ux MSE",
1134                        m_ux_on_batch=lambda scope:
                            ↪ float(torch.sum(scope["output"][:,0,:,:]
                            ↪ - scope["batch"][1][:,0,:,:])),
1135                        m_ux_on_epoch=lambda scope:
                            ↪ sum(scope["list"]) /
                            ↪ len(scope["dataset"]),
1136                        patience=25, after_epoch=after_epoch)
1137
1138                    duration = time.time() - ti
1139                    print("Training time: " + str(duration//60) +
                        ↪ "min and " +
                        ↪ str("{:.2f}".format(duration%60)) + "sec")
1140
1141                    # Guardar los el modelo
1142                    torch.save(model.state_dict(), modelo_directory
                        ↪ + "Modelo_geometries_" + vb + str(cont) +
                        ↪ ".py")
1143
1144                    metrics = {}
1145                    metrics["train_metrics"] = train_metrics
1146                    metrics["train_loss"] = train_loss
1147                    metrics["test_metrics"] = test_metrics
1148                    metrics["test_loss"] = test_loss
1149                    curves = {}
1150                    curves["train_loss_curve"] = train_loss_curve
1151                    curves["test_loss_curve"] = test_loss_curve
1152                    curves["train_ux_curve"] = train_ux_curve
1153                    curves["test_ux_curve"] = test_ux_curve
1154                    config["metrics"] = metrics
1155                    config["curves"] = curves
1156
1157                    net_param = {}
1158                    net_param["num_epochs"] = epoch[ep]
1159                    net_param["learning_ratio"] =
                        ↪ learning_rate[l_r]
```

```
net_param["kernel_size"] = kernelsize[ks]
net_param["batch_size"] = batchsize[bs]
config["net_param"] = net_param
config["training_duration"] = duration

geometries = ["circle", "ellipse", "rectangle",
↪  "square", "triangle", "triangle_eq"] # The
↪  simulated velocity
# variable = "p" # Simulated variable
# n_test = "1" # Number of test with this
↪  variable at this velocity
with open(simulation_directory + "results_" +
↪  vb
          + "_geometries_" + str(cont) +
           ↪  ".json", "w") as file:
    json.dump(config, file)


# Test circle
y_circle = y_all[:length_list[0]]
x_circle = x_all[:length_list[0]]
# Test ellipse
y_ellipse =
↪  y_all[length_list[0]:sum(length_list[:2])]
x_ellipse =
↪  x_all[length_list[0]:sum(length_list[:2])]
# Test rectangle
y_rectangle =
↪  y_all[sum(length_list[:2]):sum(length_list[:3])]
x_rectangle =
↪  x_all[sum(length_list[:2]):sum(length_list[:3])]
# Test square
y_square =
↪  y_all[sum(length_list[:3]):sum(length_list[:4])]
x_square =
↪  x_all[sum(length_list[:3]):sum(length_list[:4])]
# Test triangle
y_triangle =
↪  y_all[sum(length_list[:4]):sum(length_list[:5])]
x_triangle =
↪  x_all[sum(length_list[:4]):sum(length_list[:5])]
# Test equilateral triangle
```

```
y_triangle_eq = y_all[sum(length_list[:5]):]
x_triangle_eq = x_all[sum(length_list[:5]):]


def test_CNN(n_instantes, x_ms, y_ms, simulation_directory, geometries,
             variable, n_test, plots_directory, plots_subdirectory):

    ti = time.time()

    out = torch.empty((n_instantes,3,79,172))
    x_ms = torch.FloatTensor(x_ms)
    y_ms = torch.FloatTensor(y_ms)
    out[:,0:2,:,:]=x_ms[0,:2,:,:]

    out[0,2,:,:] =
     DeepCFD(x_ms[n_test:n_test+1].to(device))

    for i in range (1,n_instantes,1):
        print("Sample: " + str(i))
        out[i,2,:,:] =
         DeepCFD(out[i-1:i].to(device))
        error =
         (torch.abs((out[:,2:3,:,:].cpu() -
         y_ms[n_test:n_test+n_instantes].cpu())))
    out2=out[:,2:3,:,:]
    out2=out2.reshape([n_instantes,1,79,172])

    duration = time.time() - ti
    print("Training time: " + str(duration//60)
     + "min and " +
     str("{:.2f}".format(duration%60)) +
     "sec")

    # for s in range (n_instantes):
        #
         visualize3(y_ms[n_test:n_test+n_instantes].cpu().detach()
        #
         out2.cpu().detach().numpy(),
         error.cpu().detach().numpy(), s,
```

```
            #            variable, n_test,
            ↪ geometries)
        out = out2.cpu().detach().numpy()
        error = error.cpu().detach().numpy()

        ↪ y=y_ms[n_test:n_test+n_instantes].cpu().detach().numpy


        # with open(plots_subdirectory + "plots_" +
        ↪ geometries + "_" + vb + str(n_test) +
        ↪ ".pkl", "wb") as file:
            # pickle.dump([out, error, y], file)

        return out, error, y, geometries, n_test


plots = {}

n_instantes = 50
test_index_list[7][-1] = 300 # Este número de
↪ test estaba fuera de rango
test_index_list[21][-1] = 265
for test in range(30):
    print(test)
    out, error, y, geom, n_test =
    ↪ test_CNN(n_instantes, x_circle,
    ↪ y_circle, simulation_directory,
    ↪ geometries[0],
            vb, test_index_list[test][0],
            ↪ plots_directory,
            ↪ plots_subdirectory)
    plots[vb + '_' + geom + '_' + str(n_test)]
    ↪ = [y, out, error]
    out, error, y, geom, n_test =
    ↪ test_CNN(n_instantes, x_ellipse,
    ↪ y_ellipse, simulation_directory,
    ↪ geometries[1],
            vb, test_index_list[test][1],
            ↪ plots_directory,
            ↪ plots_subdirectory)
    plots[vb + '_' + geom + '_' + str(n_test)]
    ↪ = [y, out, error]
```

```
1243                    out, error, y, geom, n_test =
                        ↪ test_CNN(n_instantes, x_rectangle,
                        ↪ y_rectangle, simulation_directory,
                        ↪ geometries[2],
1244                          vb, test_index_list[test][2],
                              ↪ plots_directory,
                              ↪ plots_subdirectory)
1245                    plots[vb + '_' + geom + '_' + str(n_test)]
                        ↪ = [y, out, error]
1246                    out, error, y, geom, n_test =
                        ↪ test_CNN(n_instantes, x_square,
                        ↪ y_square, simulation_directory,
                        ↪ geometries[3],
1247                          vb, test_index_list[test][3],
                              ↪ plots_directory,
                              ↪ plots_subdirectory)
1248                    plots[vb + '_' + geom + '_' + str(n_test)]
                        ↪ = [y, out, error]
1249                    out, error, y, geom, n_test =
                        ↪ test_CNN(n_instantes, x_triangle,
                        ↪ y_triangle, simulation_directory,
                        ↪ geometries[4],
1250                          vb, test_index_list[test][4],
                              ↪ plots_directory,
                              ↪ plots_subdirectory)
1251                    plots[vb + '_' + geom + '_' + str(n_test)]
                        ↪ = [y, out, error]
1252                    out, error, y, geom, n_test =
                        ↪ test_CNN(n_instantes, x_triangle_eq,
                        ↪ y_triangle_eq, simulation_directory,
                        ↪ geometries[5],
1253                         vb, test_index_list[test][5],
                              ↪ plots_directory,
                              ↪ plots_subdirectory)
1254                    plots[vb + '_' + geom + '_' + str(n_test)]
                        ↪ = [y, out, error]
1255
1256                with open(plots_subdirectory +
                    ↪ "plots_geometries_" + vb + str(cont) +
                    ↪ ".pkl", "wb") as file:
1257                    pickle.dump(plots, file)
```

```
                    cont = cont+1
```

Código para el entrenamiento de la red que predice el primer instante para geometrías variables:

```
from train_functions import *
from functions import *
import torch.optim as optim
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx

import numpy as np
from create_channels_weights import create_channels_weights

# epoch = [1000]
# batchsize = [32, 64, 128]
# learning_rate = [0.001, 0.0001]
# kernelsize = [5, 7, 9]
# filters_ = [[8, 16, 16, 32, 32], [8, 16, 32, 64]]

epoch = [1000, 2000]
batchsize = [32, 64, 132]
learning_rate = [0.001, 0.0001]
kernelsize = [3, 5, 7, 9]
filters_ = [[8, 16, 32, 32], [8, 16, 16, 32, 32]]

cont = 100
for bs in range(len(batchsize)):
    for l_r in range(len(learning_rate)):
        # for sp_rat in range(len(split_ratio)):
        for ep in range(len(epoch)):
            for ks in range(len(kernelsize)):
                for filt in range(len(filters_)):
                    device = torch.device("cuda" if
                    ↪  torch.cuda.is_available() else "cpu")
                    y, x = pickle.load(open(
                    ↪  "./training_test_data_geometries_first_sample_e.pkl",
                    ↪  "rb"))

                    def minmax_norm(df):
```

```
1292                    return (df - df.min()) / ( df.max() -
                        ↪  df.min())
1293
1294                minimize = 0
1295
1296                if minimize == 1:
1297                    for j in range (3):
1298                        if j == 2:
1299                            y[:,j,:,:]=minmax_norm(y[:,j,:,:])
1300                        else:
1301                            x[:,j,:,:]=minmax_norm(x[:,j,:,:])
1302                            y[:,j,:,:]=minmax_norm(y[:,j,:,:])
1303
1304                y_b=y
1305                y[np.isnan(y)] = 0
1306
1307                x = torch.FloatTensor(x)
1308                y = torch.FloatTensor(y)
1309                y_b = torch.FloatTensor(y_b)
1310
1311                channels_weights=create_channels_weights(y_b,
                    ↪  device)
1312
1313                # x = x.permute(0,1,3,2)
1314                # y = y.permute(0,1,3,2)
1315                # channels_weights =
                    ↪  torch.sqrt(torch.mean(y_cw)).view(1, -1, 1,
                    ↪  1).to(device)
1316                # channels_weights =
                    ↪  torch.sqrt(torch.mean(y.permute(0, 2, 3,
                    ↪  1).reshape((981*172*79,3)) ** 2,
                    ↪  dim=0)).view(1, -1, 1, 1).to(device)
1317                print(channels_weights)
1318
1319                 # Simulation files directory
1320                simulation_directory = "./Run/"
1321                if not os.path.exists(simulation_directory):
1322                    os.makedirs(simulation_directory)
1323
1324                plots_directory = "Run/Plots_first_sample/"
1325                if not os.path.exists(plots_directory):
1326                    os.makedirs(plots_directory)
```

```
plots_subdirectory = "Run/Plots_first_sample/_" + str(cont) + "_/"
if not os.path.exists(plots_subdirectory):
    os.makedirs(plots_subdirectory)

modelo_directory = "./Modelo_first_sample/"
if not os.path.exists(modelo_directory):
    os.makedirs(modelo_directory)

idx = torch.randperm(x.shape[0])
x = x[idx].view(x.size())
y = y[idx].view(y.size())

train_data, test_data = split_tensors(x, y, ratio=0.7)

train_dataset, test_dataset = TensorDataset(*train_data), TensorDataset(*test_data)
test_x, test_y = test_dataset[:]

torch.manual_seed(0)
lr = learning_rate[l_r]
kernel_size = kernelsize[ks]
# filters = [8, 16, 32, 32]
filters = filters_[filt]
bn = False
wn = False
wd = 0.005
beta1 = 0.5
beta2 = 0.5

model = UNetEx(2, 3, filters=filters, kernel_size=kernel_size, batch_norm=bn, weight_norm=wn)
# Define opotimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)

config = {}
```

```
train_loss_curve = []
test_loss_curve = []
train_mse_curve = []
test_mse_curve = []
train_ux_curve = []
test_ux_curve = []
train_uy_curve = []
test_uy_curve = []
train_p_curve = []
test_p_curve = []

def after_epoch(scope):
    train_loss_curve.append(
    ↪  scope["train_loss"])
    test_loss_curve.append( scope["val_loss"])
    train_mse_curve.append(
    ↪  scope["train_metrics"]["mse"])
    test_mse_curve.append(
    ↪  scope["val_metrics"]["mse"])
    train_ux_curve.append(
    ↪  scope["train_metrics"]["ux"])
    test_ux_curve.append(
    ↪  scope["val_metrics"]["ux"])
    train_uy_curve.append(
    ↪  scope["train_metrics"]["uy"])
    test_uy_curve.append(
    ↪  scope["val_metrics"]["uy"])
    train_p_curve.append(
    ↪  scope["train_metrics"]["p"])
    test_p_curve.append(
    ↪  scope["val_metrics"]["p"])

def loss_func(model, batch):
    x, y = batch
    output = model(x)
    lossu = ((output[:,0,:,:] - y[:,0,:,:]) **
    ↪  2).reshape((output.shape[0],1,
    ↪  output.shape[2],output.shape[3]))
    lossv = ((output[:,1,:,:] - y[:,1,:,:]) **
    ↪  2).reshape((output.shape[0],
    ↪  1,output.shape[2],output.shape[3]))
```

```
            lossp = ((output[:,2,:,:] - y[:,2,:,:]) **
            ↪  2).reshape((output.shape[0],1,
            ↪  output.shape[2],output.shape[3]))
            # lossu = torch.abs((output[:,0,:,:] -
            ↪  y[:,0,:,:])).reshape((output.shape[0],1,
            ↪  output.shape[2],output.shape[3]))
            # lossv = torch.abs((output[:,1,:,:] -
            ↪  y[:,1,:,:])).reshape((output.shape[0],1,
            ↪  output.shape[2],output.shape[3]))
            # lossp = torch.abs((output[:,2,:,:] -
            ↪  y[:,2,:,:])).reshape((output.shape[0],1,
            ↪  output.shape[2],output.shape[3]))
            loss = (lossu + lossv +
            ↪  lossp)/channels_weights
            return torch.sum(loss), output

        ti = time.time()

        # Training model
        DeepCFD, train_metrics, train_loss,
        ↪  test_metrics, test_loss = train_model(model,
        ↪  loss_func, train_dataset, test_dataset,
        ↪  optimizer,
            epochs=epoch[ep], batch_size=batchsize[bs],
            ↪  device=device,
            m_mse_name="Total MSE",
            m_mse_on_batch=lambda scope:
            ↪  float(torch.sum((scope["output"] -
            ↪  scope["batch"][1]) ** 2)),
            m_mse_on_epoch=lambda scope:
            ↪  sum(scope["list"]) /
            ↪  len(scope["dataset"]),
            m_ux_name="Ux MSE",
            m_ux_on_batch=lambda scope:
            ↪  float(torch.sum((scope["output"][:,0,:,:]
            ↪  - scope["batch"][1][:,0,:,:]) ** 2)),
            m_ux_on_epoch=lambda scope:
            ↪  sum(scope["list"]) /
            ↪  len(scope["dataset"]),
            m_uy_name="Uy MSE",
```

```
        m_uy_on_batch=lambda scope:
        ↪  float(torch.sum((scope["output"][:,1,:,:]
        ↪  - scope["batch"][1][:,1,:,:]) ** 2)),
        m_uy_on_epoch=lambda scope:
        ↪  sum(scope["list"]) /
        ↪  len(scope["dataset"]),
        m_p_name="p MSE",
        m_p_on_batch=lambda scope:
        ↪  float(torch.sum((scope["output"][:,2,:,:]
        ↪  - scope["batch"][1][:,2,:,:]) ** 2)),
        m_p_on_epoch=lambda scope:
        ↪  sum(scope["list"]) /
        ↪  len(scope["dataset"]), patience=25,
        ↪  after_epoch=after_epoch
        )

duration = time.time() - ti
print("Training time: " + str(duration//60) +
↪  "min and " +
↪  str("{:.2f}".format(duration%60)) + "sec")

# Guardar los el modelo
torch.save(model.state_dict(), modelo_directory
↪  + "Modelo_geometries_" + str(cont) +
↪  "_data_augmentation.py")

metrics = {}
metrics["train_metrics"] = train_metrics
metrics["train_loss"] = train_loss
metrics["test_metrics"] = test_metrics
metrics["test_loss"] = test_loss
curves = {}
curves["train_loss_curve"] = train_loss_curve
curves["test_loss_curve"] = test_loss_curve
curves["train_mse_curve"] = train_mse_curve
curves["test_mse_curve"] = test_mse_curve
curves["train_ux_curve"] = train_ux_curve
curves["test_ux_curve"] = test_ux_curve
curves["train_uy_curve"] = train_uy_curve
curves["test_uy_curve"] = test_uy_curve
curves["train_p_curve"] = train_p_curve
curves["test_p_curve"] = test_p_curve
```

```
config["metrics"] = metrics
config["curves"] = curves

net_param = {}
net_param["num_epochs"] = epoch[ep]
net_param["learning_ratio"] =
↪  learning_rate[l_r]
net_param["kernel_size"] = kernelsize[ks]
net_param["batch_size"] = batchsize[bs]
net_param["arquitecture"] = filters_[filt]
config["net_param"] = net_param
config["training_duration"] = duration

with open(simulation_directory +
↪  "results_geometries_" + str(cont) +
↪  "_data_augmentation.json", "w") as file:
    json.dump([config, duration], file)

test_range = list(range(10, 20))
out = DeepCFD(test_x[test_range].to(device))
error = (torch.abs((out.cpu() -
↪  test_y[test_range].cpu())))#/torch.abs(test_y.cpu())
for s in range(10):

    ↪  visualize2(test_y[test_range].cpu().detach().numpy(),
    ↪  out.cpu().detach().numpy(),
                error.cpu().detach().numpy(), s,
                ↪  epoch[ep],
                ↪  learning_rate[l_r],
                batchsize[bs], kernelsize[ks])
out_b = out.cpu().detach().numpy()
error_b = error.cpu().detach().numpy()
test_y_b =
↪  test_y[test_range].cpu().detach().numpy()
test_x_b =
↪  test_x[test_range].cpu().detach().numpy()

with open(plots_subdirectory +
↪  "plots_geometries" + str(cont) +
↪  "data_augmentation.pkl", "wb") as file:
    pickle.dump([out_b, error_b, test_y_b,
    ↪  test_x_b, test_y, test_x], file)
```

```
                    cont = cont + 1
```

A.6 Test de los modelos neuronales y análisis de los resultados

Test de los modelos neuronales y análisis de los resultados de la CNN que predice los instantes futuros para velocidades variables de entrada:

```
import pickle
import numpy as np
import math
import random
from matplotlib import pyplot as plt
import matplotlib.gridspec as gridspec
import statistics as st
from sklearn import preprocessing
from os import listdir
from tabulate import tabulate
import json
import pandas as pd

def visualize(sample_y, out_y, error, s, n_sample, geometry):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    minev = np.min(error[s, 1, :, :])
    maxev = np.max(error[s, 1, :, :])

    minep = np.min(error[s, 2, :, :])
    maxep = np.max(error[s, 2, :, :])
```

```
1500
1501      plt.figure()
1502      fig = plt.gcf()
1503      plt.suptitle(geometry + ". Sample nº " + str(n_sample+s),
          ↪  fontsize=20)
1504      fig.set_size_inches(15, 10)
1505      plt.subplot(3, 3, 1)
1506      plt.title('CFD', fontsize=18)
1507      plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
          ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
1508      plt.colorbar(orientation='horizontal')
1509      plt.ylabel('Ux', fontsize=18)
1510      plt.subplot(3, 3, 2)
1511      plt.title('CNN', fontsize=18)
1512      plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
          ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
1513      plt.colorbar(orientation='horizontal')
1514      plt.subplot(3, 3, 3)
1515      plt.title('Error', fontsize=18)
1516      plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
          ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
1517      plt.colorbar(orientation='horizontal')
1518
1519      plt.subplot(3, 3, 4)
1520      plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet', vmin
          ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
1521      plt.colorbar(orientation='horizontal')
1522      plt.ylabel('Uy', fontsize=18)
1523      plt.subplot(3, 3, 5)
1524      plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin =
          ↪  minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
1525      plt.colorbar(orientation='horizontal')
1526      plt.subplot(3, 3, 6)
1527      plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin =
          ↪  minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
1528      plt.colorbar(orientation='horizontal')
1529
1530
1531      plt.subplot(3, 3, 7)
1532      plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet', vmin
          ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
1533      plt.colorbar(orientation='horizontal')
```

```
    plt.ylabel('p', fontsize=18)
    plt.subplot(3, 3, 8)
    plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin =
    ↪  minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 3, 9)
    plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin =
    ↪  minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.tight_layout()
    plt.show()


def visualize3(sample_y, s, n_sample, geometry):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle(geometry + ". Sample n.º " + str(n_sample+s),
    ↪  fontsize=20)
    fig.set_size_inches(15, 10)
    plt.suptitle("Time instant " + str(s+1), fontsize=20)
    plt.subplot(3, 1, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow((sample_y[s, 0, :, :]), cmap='jet', vmin = minu,
    ↪  vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Ux', fontsize=18)

    plt.subplot(3, 1, 2)
    plt.imshow((sample_y[s, 1, :, :]), cmap='jet', vmin = minv,
    ↪  vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Uy', fontsize=18)
```

```
1571
1572
1573        plt.subplot(3, 1, 3)
1574        plt.imshow((sample_y[s, 2, :, :]), cmap='jet', vmin = minp,
            ↪  vmax = maxp, origin='lower', extent=[0,172,0,79])
1575        plt.colorbar(orientation='horizontal')
1576        plt.ylabel('p', fontsize=18)
1577        plt.tight_layout()
1578        plt.show()
1579
1580    def visualize_rel_error(sample_y, out_y, error, s, input_velocity
        ↪  ):
1581
1582        # Visualize function with relative error added
1583
1584        minu = np.min(sample_y[s, 0, :, :])
1585        maxu = np.max(sample_y[s, 0, :, :])
1586
1587        minv = np.min(sample_y[s, 1, :, :])
1588        maxv = np.max(sample_y[s, 1, :, :])
1589
1590        minp = np.min(sample_y[s, 2, :, :])
1591        maxp = np.max(sample_y[s, 2, :, :])
1592
1593        mineu = np.min(error[s, 0, :, :])
1594        maxeu = np.max(error[s, 0, :, :])
1595
1596        minev = np.min(error[s, 1, :, :])
1597        maxev = np.max(error[s, 1, :, :])
1598
1599        minep = np.min(error[s, 2, :, :])
1600        maxep = np.max(error[s, 2, :, :])
1601
1602        plt.figure()
1603        fig = plt.gcf()
1604        fig.set_size_inches(15, 10)
1605        plt.suptitle("Time instant " + str(s+1), fontsize=20)
1606        plt.subplot(3, 4, 1)
1607        plt.title('CFD', fontsize=18)
1608        plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
            ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
1609        plt.colorbar(orientation='horizontal')
```

```
    plt.ylabel('Ux', fontsize=18)
    plt.subplot(3, 4, 2)
    plt.title('CNN', fontsize=18)
    plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
    ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 4, 3)
    plt.title('Error', fontsize=18)
    plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
    ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 4, 4)


    plt.subplot(3, 4, 5)
    plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet', vmin
    ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Uy', fontsize=18)
    plt.subplot(3, 4, 6)
    plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin =
    ↪  minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 4, 7)
    plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin =
    ↪  minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 4, 8)


    plt.subplot(3, 4, 9)
    plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet', vmin
    ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('p', fontsize=18)
    plt.subplot(3, 4, 10)
    plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin =
    ↪  minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 4, 11)
    plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin =
    ↪  minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
```

```
1644        plt.colorbar(orientation='horizontal')
1645        plt.subplot(3, 4, 12)
1646        plt.tight_layout()
1647        plt.show()
1648
1649    def test_mean_error(plots):
1650        error = np.zeros([len(plots), 50]) # Habría que poner esto pero
            ↪ me da fallo --> len(plots[id_test][2])])
1651        max_error = np.zeros([len(plots), 50])
1652        variance = np.zeros([len(plots), 50])
1653        col = 0
1654        for key in plots:
1655            for row in range(50):
1656                error[col, row] = np.mean(plots[key][2][row, 0],
                    ↪ dtype=np.float64)
1657                max_error[col, row] = np.amax(plots[key][2][row, 0])
1658                variance[col, row] = np.var(plots[key][2][row, 0], ddof
                    ↪ = 1)
1659            col = col + 1
1660        mean_error = np.mean(error, axis = 0, dtype=np.float64)
1661        max_error_all = np.amax(max_error, axis = 0)
1662        variance_all = np.mean(variance, axis = 0)
1663        me_error = np.mean(mean_error)
1664        v_all = np.mean(variance_all)
1665        ma_error = np.amax(max_error_all)
1666        return mean_error, max_error_all, variance_all, me_error, v_all,
            ↪ ma_error
1667
1668    def plot_feature(feature_type, upper_limit, legend, plot_title,
    ↪ y_label):
1669        for i in range(10):
1670            plt.plot(feature_type[i], '--o', markersize = 3)
1671
1672        plt.grid(axis='both')
1673        plt.ylabel(y_label)
1674        plt.ylim((0,upper_limit))
1675        plt.legend(legend)
1676        plt.title(plot_title)
1677        plt.show()
1678
1679    if __name__ == "__main__":
1680
```

```
1681    plots = []
1682    training_info = []
1683    path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias/"
1684    # path = "C:/Users/alvar/Desktop/Master/ Segundo/TFM/2ª
        ↪  parte/Datos para entrenamiento/Resultados_vx/"
1685
1686    plots_directories = "Plots vx/"#, "Plots vy/", "Plots p/"]
1687    statistics_directories = "Estadisticas vx"#, "Estadisticas vy",
        ↪  "Estadisticas p"]
1688    # filepaths = [f for f in listdir(str(path +
        ↪  plots_directories)) if f.endswith('.pkl')]
1689    # for file in filepaths:
1690    #     data = pickle.load(open(path + plots_directories + file,
        ↪  "rb"))
1691    #     mean_error, max_error, variance, me_error, var_, ma_error
        ↪  = test_mean_error(data)
1692    #     with open(path + statistics_directories +
        ↪  "/plots_geometries_statistics_" + file[-6:-4] + ".json",
        ↪  "wb") as save_file:
1693    #         pickle.dump([mean_error, max_error, variance,
        ↪  me_error, var_, ma_error], save_file)
1694
1695    filepaths_vx = [f for f in listdir(str(path +
        ↪  plots_directories)) if f.endswith('.pkl')]
1696    data_vx = pickle.load(open(path + plots_directories +
        ↪  filepaths_vx[10], "rb"))
1697    filepaths_vy = [f for f in listdir(str(path + "Plots vy/")) if
        ↪  f.endswith('.pkl')]
1698    data_vy = pickle.load(open(path + "Plots vy/" + filepaths_vy[7],
        ↪  "rb"))
1699    filepaths_p = [f for f in listdir(str(path + "Plots p/")) if
        ↪  f.endswith('.pkl')]
1700    data_p = pickle.load(open(path + "Plots p/" + filepaths_p[5],
        ↪  "rb"))
1701
1702    cfd = np.zeros([50, 3, 79, 172])
1703    cnn = np.zeros([50, 3, 79, 172])
1704    error = np.zeros([50, 3, 79, 172])
1705
1706    geometry = 'circle'
1707    samples = ['244', '349', '369']
1708    # samples = ['289', '374', '279']
```

```
    # samples = ['115', '369', '51']
    # samples = ['74', '190', '317']
    # samples = ['169', '62', '428']
    # samples = ['313', '233', '19']

    cfd[:,0,:,:] = data_vx['vx_' + geometry + '_' +
    ↪  samples[0]][0][:,0,:,:]
    cfd[:,1,:,:] = data_vy['vy_' + geometry + '_' +
    ↪  samples[1]][0][:,0,:,:]
    cfd[:,2,:,:] = data_p['p_' + geometry + '_' +
    ↪  samples[2]][0][:,0,:,:]

    cnn[:,0,:,:] = data_vx['vx_' + geometry + '_' +
    ↪  samples[0]][1][:,0,:,:]
    cnn[:,1,:,:] = data_vy['vy_' + geometry + '_' +
    ↪  samples[1]][1][:,0,:,:]
    cnn[:,2,:,:] = data_p['p_' + geometry + '_' +
    ↪  samples[2]][1][:,0,:,:]

    error[:,0,:,:] = data_vx['vx_' + geometry + '_' +
    ↪  samples[0]][2][:,0,:,:]
    error[:,1,:,:] = data_vy['vy_' + geometry + '_' +
    ↪  samples[1]][2][:,0,:,:]
    error[:,2,:,:] = data_p['p_' + geometry + '_' +
    ↪  samples[2]][2][:,0,:,:]

    s = [0, 4, 9, 14, 19, 29, 39, 49]

    for a in range(len(s)):
        visualize(np.moveaxis(cfd, [2,3], [3,2]),
                  np.moveaxis(cnn, [2,3], [3,2]),
                  np.moveaxis(error, [2,3], [3,2]),
                  s[a], 1, "Circle")

    cfd_vx = np.zeros([len(data_vx)*50, 1, 79, 172])
    out_vx = np.zeros([len(data_vx)*50, 1, 79, 172])

    cfd_vy = np.zeros([len(data_vy)*50, 1, 79, 172])
    out_vy = np.zeros([len(data_vy)*50, 1, 79, 172])

    cfd_p = np.zeros([len(data_p)*50, 1, 79, 172])
    out_p = np.zeros([len(data_p)*50, 1, 79, 172])
```

```
    c = 0
    for key in data_vx:
        cfd_vx[c:c+50,:,:,:] = data_vx[key][0]
        out_vx[c:c+50,:,:,:] = data_vx[key][1]
        c = c + 50

    c = 0
    for key in data_vy:
        cfd_vy[c:c+50,:,:,:] = data_vy[key][0]
        out_vy[c:c+50,:,:,:] = data_vy[key][1]
        c = c + 50

    c = 0
    for key in data_p:
        cfd_p[c:c+50,:,:,:] = data_p[key][0]
        out_p[c:c+50,:,:,:] = data_p[key][1]
        c = c + 50

    cfd_vx_mean = np.mean(cfd_vx)
    cfd_vy_mean = np.mean(cfd_vy)
    cfd_p_mean = np.mean(cfd_p)
    cfd_vx_std = np.std(cfd_vx)
    cfd_vy_std = np.std(cfd_vy)
    cfd_p_std = np.std(cfd_p)

    out_vx_mean = np.mean(out_vx)
    out_vy_mean = np.mean(out_vy)
    out_p_mean = np.mean(out_p)
    out_vx_std = np.std(out_vx)
    out_vy_std = np.std(out_vy)
    out_p_std = np.std(out_p)
```

```
import os
import json
import torch
import pickle
import time
from train_functions import *
import torch.optim as optim
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx
```

```
from Models.UNet import UNet
import numpy as np
from matplotlib import pyplot as plt
from os import listdir
os.environ['KMP_DUPLICATE_LIB_OK']='True'

def visualize(sample_y, out_y, error, s, variable, n_sample,
 ↪  velocity):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    # minv = np.min(sample_y[s, 1, :, :])
    # maxv = np.max(sample_y[s, 1, :, :])

    # minp = np.min(sample_y[s, 2, :, :])
    # maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    # minev = np.min(error[s, 1, :, :])
    # maxev = np.max(error[s, 1, :, :])

    # minep = np.min(error[s, 2, :, :])
    # maxep = np.max(error[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
     ↪  + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
     ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel(variable, fontsize=18)
    plt.subplot(3, 3, 2)
    plt.title('CNN', fontsize=18)
    plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
     ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
```

```
1821      plt.colorbar(orientation='horizontal')
1822      plt.subplot(3, 3, 3)
1823      plt.title('Error', fontsize=18)
1824      plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
          ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
1825      plt.colorbar(orientation='horizontal')
1826
1827      # plt.subplot(3, 3, 4)
1828      # plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet',
          ↪  vmin = minv, vmax = maxv, origin='lower',
          ↪  extent=[0,172,0,79])
1829      # plt.colorbar(orientation='horizontal')
1830      # plt.ylabel('Uy', fontsize=18)
1831      # plt.subplot(3, 3, 5)
1832      # plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin
          ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
1833      # plt.colorbar(orientation='horizontal')
1834      # plt.subplot(3, 3, 6)
1835      # plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin
          ↪  = minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
1836      # plt.colorbar(orientation='horizontal')
1837
1838
1839      # plt.subplot(3, 3, 7)
1840      # plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet',
          ↪  vmin = minp, vmax = maxp, origin='lower',
          ↪  extent=[0,172,0,79])
1841      # plt.colorbar(orientation='horizontal')
1842      # plt.ylabel('p', fontsize=18)
1843      # plt.subplot(3, 3, 8)
1844      # plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin
          ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
1845      # plt.colorbar(orientation='horizontal')
1846      # plt.subplot(3, 3, 9)
1847      # plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin
          ↪  = minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
1848      # plt.colorbar(orientation='horizontal')
1849      plt.tight_layout()
1850      plt.show()
1851
1852  def visualize2(sample_y, s, file):
1853
```

```
    plt.figure()
    fig = plt.gcf()
    # fig.set_size_inches(15, 10)

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
    ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.title("Test " + file)
    plt.tight_layout()
    plt.show()

def test_CNN(n_instantes, x_ms, n_test):

    ti = time.time()

    out = torch.empty((n_instantes,3,172,79))
    x_ms = torch.FloatTensor(x_ms)
    out[:,0:2,:,:]=x_ms[0,:2,:,:]

    out[0,2,:,:] = model_prueba(x_ms[n_test:n_test+1])

    for i in range (1,n_instantes,1):
        print("Sample: " + str(i))
        out[i,2,:,:] = model_prueba(out[i-1:i])
    out2=out[:,2:3,:,:]
    out2=out2.reshape([n_instantes,1,172,79])
    tout = time.time()

    duration = tout-ti

    return out2, duration


if __name__ == "__main__":

    path = "E:/CNN Velocidades/Modelos/"


    torch.manual_seed(0)
```

```
lr = 0.0001
# lr = 0.0001
# kernel_size = 3
# kernel_size = 5
kernel_size = 7
filters = [8, 16, 32, 32]
bn = False
wn = False
wd = 0.005
beta1 = 0.5
beta2 = 0.5

model_prueba = UNetEx(3, 1, filters=filters,
↪  kernel_size=kernel_size, batch_norm=bn, weight_norm=wn)
# model_prueba.load_state_dict(torch.load(path +
↪  "Modelo_vy13.py", map_location=torch.device('cpu')))
model_prueba.load_state_dict(torch.load(path + "Modelo_vy27.py",
↪  map_location=torch.device('cpu')))
# model_prueba.load_state_dict(torch.load(path +
↪  "Modelo_p26.py", map_location=torch.device('cpu')))
model_prueba.eval()


path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias
↪  /Resultados_first_sample/tests_first_sample/"
geom_index = [1, 63, 121, 723, 1305, 1923]
geometry = ["circle", "square", "triangle_eq", "rectangle",
↪  "ellipse", "triangle"]
cfd_all = np.zeros([len(geometry)*50, 1, 172, 79])
out_all = np.zeros([len(geometry)*50, 1, 172, 79])
for g_i in range(len(geometry)):
    # cfd = pickle.load(open(path + "Test geometrias no
    ↪  vistas/dataY_1300_1001_1050.pkl", "rb"))
    cfd = pickle.load(open("E:/NuevasGeometrias/Y_geom" +
    ↪  str(geom_index[g_i]) + "_interpolated.pkl", "rb"))
    # filepaths = [f for f in listdir(str(path)) if
    ↪  f.endswith('.pkl')]
    # for file in filepaths:
    #     with open(path + file, 'rb') as f:
    #         input_data.append(pickle.load(f))
    input_data = []
    input_data.append(pickle.load(open(path +
```

```
                                    "input_data_geometry_" +
                                    ↪ geometry[g_i]
                                    +
                                    ↪ "_size_factor_1_rotation_angle_1_variable_vx.
                                    ↪ "rb")))
        x = np.zeros([1,3,172,79])
        x[:,-1,:,:] = input_data[0][0][:,1,:,:].detach().numpy()
        x[:,0,:,:] = input_data[0][1]
        x[:,1,:,:] = input_data[0][2]

            # with open(simulation_directory + "plots_" + velocity
            ↪ + "ms_" + variable + "_Final_v2" + str(n_test) +
            ↪ ".pkl", "wb") as file:
                # pickle.dump([out, error, y], file)

        variable = 'vy'
        n_instantes = 50
        out2, duration = test_CNN(n_instantes, x, 0)
        # cfd = cfd[:,[1,2,0],:,:]
        cfd = torch.tensor(np.moveaxis(cfd, [2,3], [3,2]))
        cfd = cfd[:,2:3,:,:]
        error = abs(cfd[:n_instantes,:,:,:] - out2)
        s = [0, 4, 9, 14, 19, 29, 39, 49]
        for a in s:
            # visualize2(out2.detach().numpy(), s, filepaths[0])
            visualize(cfd.detach().numpy(), out2.detach().numpy(),
                      error.detach().numpy(), a, variable, 1, "5")

        path2 = "E:/NuevasGeometrias
        ↪ /Resultados_NuevasGeometrias/test_nuevas_geometrias/"
        with open(path2 + "test_" + geometry[g_i] + '_' + variable
        ↪ + ".pkl", "wb") as file:
            pickle.dump([cfd.detach().numpy()[:n_instantes],
            ↪ out2.detach().numpy(),
                      error.detach().numpy()], file)

        cfd_all[g_i*50:(g_i+1)*50] = cfd[:50].detach().numpy()
        out_all[g_i*50:(g_i+1)*50] = out2.detach().numpy()

    mean_value_cfd = np.mean(cfd_all[:,0,:,:])
    std_cfd = np.std(cfd_all[:,0,:,:])

```

```
    mean_value_cnn = np.mean(out_all[:,0,:,:])
    std_cnn = np.std(out_all[:,0,:,:])

    sample10 = [x+10 for x in sample1]
    sample15 = [x+15 for x in sample1]
    sample20 = [x+20 for x in sample1]
    sample30 = [x+30 for x in sample1]
    sample40 = [x+40 for x in sample1]
    sample50 = [x+50 for x in sample1]
    sample = [sample1, sample5, sample10, sample15, sample20,
    ↪  sample30,
              sample40, sample50]
    samples = [1, 5, 10, 15, 20, 30, 40, 50]

    for s in range(len(sample)):
        cfd = cfd_all[sample[s]]
        out = out_all[sample[s]]
        gen_data_distribution(cfd, out, samples[s])
```

Test de los modelos neuronales y análisis de los resultados de la CNN que predice los instantes futuros para geometrías variables:

```
import pickle
import numpy as np
import math
import random
from matplotlib import pyplot as plt
import matplotlib.gridspec as gridspec
import statistics as st
from sklearn import preprocessing
from os import listdir
from tabulate import tabulate
import json
import pandas as pd

def visualize(sample_y, out_y, error, s, variable, n_sample,
↪  velocity):

    minu = np.min(sample_y[s, :, :])
    maxu = np.max(sample_y[s, :, :])

    # minv = np.min(sample_y[s, 1, :, :])
```

```
    # maxv = np.max(sample_y[s, 1, :, :])

    # minp = np.min(sample_y[s, 2, :, :])
    # maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, :, :])
    maxeu = np.max(error[s, :, :])

    # minev = np.min(error[s, 1, :, :])
    # maxev = np.max(error[s, 1, :, :])

    # minep = np.min(error[s, 2, :, :])
    # maxep = np.max(error[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
    ↪  + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow(np.transpose(sample_y[s, :, :]), cmap='jet', vmin =
    ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel(variable, fontsize=18)
    plt.subplot(3, 3, 2)
    plt.title('CNN', fontsize=18)
    plt.imshow(np.transpose(out_y[s, :, :]), cmap='jet', vmin =
    ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 3, 3)
    plt.title('Error', fontsize=18)
    plt.imshow(np.transpose(error[s, :, :]), cmap='jet', vmin =
    ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')

    # plt.subplot(3, 3, 4)
    # plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet',
    ↪  vmin = minv, vmax = maxv, origin='lower',
    ↪  extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')
    # plt.ylabel('Uy', fontsize=18)
```

```
    # plt.subplot(3, 3, 5)
    # plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin
    ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')
    # plt.subplot(3, 3, 6)
    # plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin
    ↪  = minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')


    # plt.subplot(3, 3, 7)
    # plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet',
    ↪  vmin = minp, vmax = maxp, origin='lower',
    ↪  extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')
    # plt.ylabel('p', fontsize=18)
    # plt.subplot(3, 3, 8)
    # plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin
    ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')
    # plt.subplot(3, 3, 9)
    # plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin
    ↪  = minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
    # plt.colorbar(orientation='horizontal')
    plt.tight_layout()
    plt.show()


def visualize3(sample_y, s):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    fig.set_size_inches(15, 10)
```

```
    plt.suptitle("Time instant " + str(s+1), fontsize=20)
    plt.subplot(3, 1, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow((sample_y[s, 0, :, :]), cmap='jet', vmin = minu,
    ↪  vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Ux', fontsize=18)

    plt.subplot(3, 1, 2)
    plt.imshow((sample_y[s, 1, :, :]), cmap='jet', vmin = minv,
    ↪  vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Uy', fontsize=18)


    plt.subplot(3, 1, 3)
    plt.imshow((sample_y[s, 2, :, :]), cmap='jet', vmin = minp,
    ↪  vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('p', fontsize=18)
    plt.tight_layout()
    plt.show()

def visualize_rel_error(sample_y, out_y, error, s, input_velocity
↪  ):

    # Visualize function with relative error added

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    minev = np.min(error[s, 1, :, :])
    maxev = np.max(error[s, 1, :, :])
```

```
2108
2109      minep = np.min(error[s, 2, :, :])
2110      maxep = np.max(error[s, 2, :, :])
2111
2112      plt.figure()
2113      fig = plt.gcf()
2114      fig.set_size_inches(15, 10)
2115      plt.suptitle("Time instant " + str(s+1), fontsize=20)
2116      plt.subplot(3, 4, 1)
2117      plt.title('CFD', fontsize=18)
2118      plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
          ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2119      plt.colorbar(orientation='horizontal')
2120      plt.ylabel('Ux', fontsize=18)
2121      plt.subplot(3, 4, 2)
2122      plt.title('CNN', fontsize=18)
2123      plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
          ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2124      plt.colorbar(orientation='horizontal')
2125      plt.subplot(3, 4, 3)
2126      plt.title('Error', fontsize=18)
2127      plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
          ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
2128      plt.colorbar(orientation='horizontal')
2129      plt.subplot(3, 4, 4)
2130
2131
2132      plt.subplot(3, 4, 5)
2133      plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet', vmin
          ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2134      plt.colorbar(orientation='horizontal')
2135      plt.ylabel('Uy', fontsize=18)
2136      plt.subplot(3, 4, 6)
2137      plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin =
          ↪  minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2138      plt.colorbar(orientation='horizontal')
2139      plt.subplot(3, 4, 7)
2140      plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin =
          ↪  minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
2141      plt.colorbar(orientation='horizontal')
2142      plt.subplot(3, 4, 8)
2143
```

```
2144
2145        plt.subplot(3, 4, 9)
2146        plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet', vmin
            ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2147        plt.colorbar(orientation='horizontal')
2148        plt.ylabel('p', fontsize=18)
2149        plt.subplot(3, 4, 10)
2150        plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin =
            ↪  minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2151        plt.colorbar(orientation='horizontal')
2152        plt.subplot(3, 4, 11)
2153        plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin =
            ↪  minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
2154        plt.colorbar(orientation='horizontal')
2155        plt.subplot(3, 4, 12)
2156        plt.tight_layout()
2157        plt.show()
2158
2159    def test_mean_error(plots):
2160        error = np.zeros([len(plots), 50]) # Habría que poner esto pero
            ↪  me da fallo --> len(plots[id_test][2])])
2161        max_error = error
2162        variance = error
2163        col = 0
2164        for key in plots:
2165            for row in range(50): #
2166                error[col, row] = np.mean(plots[key][2][row],
                    ↪  dtype=np.float64)
2167                max_error[col, row] = np.max(plots[key][2][row])
2168                variance[col, row] = np.var(plots[key][2][row], ddof =
                    ↪  1)
2169            col = col + 1
2170        mean_error = np.mean(error, axis = 0, dtype=np.float64)
2171        max_error_all = np.max(max_error, axis = 0)
2172        variance_all = np.mean(variance, axis = 0)
2173        me_error = np.mean(mean_error)
2174        v_all = np.mean(variance_all)
2175        ma_error = np.max(max_error_all)
2176        return mean_error, max_error_all, variance_all, me_error, v_all,
            ↪  ma_error
2177
```

```
def plot_feature(feature_type, upper_limit, legend, plot_title,
↪  y_label):
    for i in range(10):
        plt.plot(feature_type[i], '--o', markersize = 3)

    plt.grid(axis='both')
    plt.ylabel(y_label)
    plt.ylim((0,upper_limit))
    plt.legend(legend)
    plt.title(plot_title)
    plt.show()

def get_sub(x):
    normal =
    ↪  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-=()"
    sub_s = "CDGQwZw"
    res = x.maketrans(''.join(normal), ''.join(sub_s))
    return x.translate(res)

def gen_data_distribution(CFD, out, sample):
## Distribucion de los datos:
    CFD_array = np.array(CFD)
    out_array = np.array(out)

    fig, ax = plt.subplots()
    fig.set_size_inches(7, 4)
    # plt.title('Histogram u{}'.format(get_sub('y')) + '. Sample
    ↪  n.º ' + str(sample+1), fontsize = 20)
    plt.title('Histogram p' + '. Sample n.º ' + str(sample+1),
    ↪  fontsize = 20)
    fig.tight_layout()
    n, bins, patches =
    ↪  plt.hist(CFD_array[:,0,:,:].reshape(len(CFD)*79*172),
                                200, density=False, facecolor='r',
                                alpha=0.5)
    n, bins, patches =
    ↪  plt.hist(out_array[:,0,:,:].reshape(len(out)*79*172),
                                200, density=False, facecolor='b',
                                alpha=0.5)
    # plt.xlabel('u{}'.format(get_sub('y')) + ' [m/s]', fontsize =
    ↪  18)
    plt.xlabel('p [Pa]', fontsize = 18)
```

```
    plt.rc('xtick', labelsize = 14)
    plt.rc('ytick', labelsize = 14)
    if sample == 0:
        plt.legend(['CFD', 'CNN'], fontsize = 16)
    plt.grid(True)
    ax.set_yscale('log')
    path = 'C:/Users/alvar/Desktop/Master/Segundo/TFM/2ª parte/Imagenes para TFM/Instantes futuros geometrias/Histogramas'
    plt.savefig(path + '/Histogram p muestra: ' + str(sample+1) + '.png',  format="png")
    plt.show()

if __name__ == "__main__":

    mean_error_list = []
    max_error_list = []
    variance_list = []
    me_error_list = []
    var_list = []
    ma_error_list = []

    plots = []
    training_info = []
    path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias/"
    # path = "C:/Users/alvar/Desktop/Master/Segundo/TFM/2ª parte/Datos para entrenamiento/Resultados_vy_p/"

    statistics_directory = "Estadisticas p/"
    plots_directory = "Plots p/"

    filepaths = [f for f in listdir(str(path + statistics_directory)) if f.endswith('.json')]
    for file in filepaths:
        mean_error, max_error, variance, me_error, var_, ma_error = pickle.load(open(path + statistics_directory + file, "rb"))
        mean_error_list.append(mean_error)
        max_error_list.append(max_error)
        variance_list.append(variance)
        me_error_list.append(me_error)
        var_list.append(var_)
```

```
        ma_error_list.append(ma_error)
    filepaths = [f for f in listdir(str(path + plots_directory)) if
    ↪  f.endswith('.json')]
    for file in filepaths:
        data = json.load(open(path + plots_directory + file, "rb"))
        training_info.append(data)

    xlabel = []
    training_duration = []
    for i in range(len(training_info)):
        xlabel.insert(-1 ,'Test nº' + str(i+3))

        ↪  training_duration.append(training_info[i]['training_duration']/3600)
    plt.plot(training_duration, '--bo', markersize = 3)
    plt.grid(axis='both')
    # plt.xlabel(xlabel)
    plt.ylabel('Time (min)')
    plt.title('Training duration of each model')
    plt.show()

    table_data = []
    col_names = ["Id", "Train time (h)", "Num_epochs", "Lr",
    ↪  "Kernel_size", "Batch_size",
                "MeErr " + statistics_directory[-3:-1], "MaxErr "
                ↪  + statistics_directory[-3:-1]]
    for i in range(len(me_error_list)):
        table_data.append([filepaths[i][-7:-5],

                           ↪  training_info[i]["training_duration"]/3600,

                           ↪  training_info[i]["net_param"]['num_epochs'],

                           ↪  training_info[i]["net_param"]['learning_ratio'],

                           ↪  training_info[i]["net_param"]['kernel_size'],

                           ↪  training_info[i]["net_param"]['batch_size'],
                          me_error_list[i], ma_error_list[i]])
    table = pd.DataFrame(table_data, columns = col_names)
    # table = tabulate(table_data, headers = col_names)
    print(table)

```

```
    table.to_csv(path + statistics_directory +
    ↪  'training_information.csv')

    # plot_feature(mean_error_list, 70, legend, 'Mean error for
    ↪  each prediction',
    #              'Mean error (m/s)')
    # plot_feature(max_error_list, 40, legend, 'Maximum error for
    ↪  each prediction',
    #              'Max error (m/s)')
    # plot_feature(variance_list, 10, legend, 'Variance for each
    ↪  prediction',
    #              'Variance (m^2/s^2)')

    filepaths = [f for f in listdir(str(path + plots_directory)) if
    ↪  f.endswith('.pkl')]
    # data = pickle.load(open(path + plots_directory +
    ↪  filepaths[10], "rb"))
    # data = pickle.load(open(path + plots_directory + filepaths[7],
    ↪  "rb"))
    data = pickle.load(open(path + plots_directory + filepaths[5],
    ↪  "rb"))


    s = [0, 4, 9, 14, 19, 29, 39, 49]

    for a in s:
          c = 0
          CFD = np.zeros([len(data), 1, 79, 172])
          out = np.zeros([len(data), 1, 79, 172])
          for key in data:
            CFD[c,0,:,:] = data[key][0][a,0,:,:]
            out[c,0,:,:] = data[key][1][a,0,:,:]
            c = c+1
          gen_data_distribution(CFD, out, a)
```

```
import os
import json
import torch
import pickle
import time
from train_functions import *
import torch.optim as optim
```

```
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx
from Models.UNet import UNet
import numpy as np
from matplotlib import pyplot as plt
from os import listdir
os.environ['KMP_DUPLICATE_LIB_OK']='True'

def visualize(sample_y, out_y, error, s, variable, n_sample,
    velocity):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    # minv = np.min(sample_y[s, 1, :, :])
    # maxv = np.max(sample_y[s, 1, :, :])

    # minp = np.min(sample_y[s, 2, :, :])
    # maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    # minev = np.min(error[s, 1, :, :])
    # maxev = np.max(error[s, 1, :, :])

    # minep = np.min(error[s, 2, :, :])
    # maxep = np.max(error[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
     + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
     = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel(variable, fontsize=18)
    plt.subplot(3, 3, 2)
    plt.title('CNN', fontsize=18)
```

```
2353        plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
            ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2354        plt.colorbar(orientation='horizontal')
2355        plt.subplot(3, 3, 3)
2356        plt.title('Error', fontsize=18)
2357        plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
            ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
2358        plt.colorbar(orientation='horizontal')
2359
2360        # plt.subplot(3, 3, 4)
2361        # plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet',
            ↪  vmin = minv, vmax = maxv, origin='lower',
            ↪  extent=[0,172,0,79])
2362        # plt.colorbar(orientation='horizontal')
2363        # plt.ylabel('Uy', fontsize=18)
2364        # plt.subplot(3, 3, 5)
2365        # plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin
            ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2366        # plt.colorbar(orientation='horizontal')
2367        # plt.subplot(3, 3, 6)
2368        # plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin
            ↪  = minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
2369        # plt.colorbar(orientation='horizontal')
2370
2371
2372        # plt.subplot(3, 3, 7)
2373        # plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet',
            ↪  vmin = minp, vmax = maxp, origin='lower',
            ↪  extent=[0,172,0,79])
2374        # plt.colorbar(orientation='horizontal')
2375        # plt.ylabel('p', fontsize=18)
2376        # plt.subplot(3, 3, 8)
2377        # plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin
            ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2378        # plt.colorbar(orientation='horizontal')
2379        # plt.subplot(3, 3, 9)
2380        # plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin
            ↪  = minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
2381        # plt.colorbar(orientation='horizontal')
2382        plt.tight_layout()
2383        plt.show()
2384
```

```
2385 def visualize2(sample_y, s, file):
2386
2387     plt.figure()
2388     fig = plt.gcf()
2389     # fig.set_size_inches(15, 10)
2390
2391     minu = np.min(sample_y[s, 0, :, :])
2392     maxu = np.max(sample_y[s, 0, :, :])
2393
2394     plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
          ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2395     plt.colorbar(orientation='horizontal')
2396     plt.title("Test " + file)
2397     plt.tight_layout()
2398     plt.show()
2399
2400 def test_CNN(n_instantes, x_ms, n_test):
2401
2402     ti = time.time()
2403
2404     out = torch.empty((n_instantes,3,172,79))
2405     x_ms = torch.FloatTensor(x_ms)
2406     out[:,0:2,:,:]=x_ms[0,:2,:,:]
2407
2408     out[0,2,:,:] = model_prueba(x_ms[n_test:n_test+1])
2409
2410     for i in range (1,n_instantes,1):
2411         print("Sample: " + str(i))
2412         out[i,2,:,:] = model_prueba(out[i-1:i])
2413     out2=out[:,2:3,:,:]
2414     out2=out2.reshape([n_instantes,1,172,79])
2415     tout = time.time()
2416
2417     duration = tout-ti
2418
2419     return out2, duration
2420
2421 def gen_data_distribution(CFD, out, sample):
2422 ## Distribucion de los datos:
2423     CFD_array = np.array(CFD)
2424     out_array = np.array(out)
2425
```

```
2426        fig, ax = plt.subplots()
2427        fig.set_size_inches(7, 4)
2428        # plt.title('Histogram u{}'.format(get_sub('y')) + '. Sample
            ↪  n.º ' + str(sample+1), fontsize = 20)
2429        plt.title('Histogram p' + '. Sample n.º ' + str(sample),
            ↪  fontsize = 20)
2430        fig.tight_layout()
2431        n, bins, patches =
            ↪  plt.hist(CFD_array[:,0,:,:].reshape(len(CFD)*79*172),
2432                                    200, density=False, facecolor='r',
2433                                    alpha=0.5)
2434        n, bins, patches =
            ↪  plt.hist(out_array[:,0,:,:].reshape(len(out)*79*172),
2435                                    200, density=False, facecolor='b',
2436                                    alpha=0.5)
2437        # plt.xlabel('u{}'.format(get_sub('y')) + ' [m/s]', fontsize =
            ↪  18)
2438        plt.xlabel('p [Pa]', fontsize = 18)
2439        plt.rc('xtick', labelsize = 14)
2440        plt.rc('ytick', labelsize = 14)
2441        if sample == 1:
2442            plt.legend(['CFD', 'CNN'], fontsize = 16)
2443        plt.grid(True)
2444        ax.set_yscale('log')
2445        # path = 'C:/Users/alvar/Desktop/Master/Segundo/TFM/2ª
            ↪  parte/Imagenes para TFM/Instantes futuros
            ↪  geometrias/Histogramas'
2446        # plt.savefig(path + '/Histogram p muestra: ' + str(sample+1) +
            ↪  '.png',  format="png")
2447        plt.show()
2448
2449    if __name__ == "__main__":
2450
2451        path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias
            ↪  /Modelos/"
2452
2453        torch.manual_seed(0)
2454        # lr = 0.001
2455        lr = 0.0001
2456        # kernel_size = 3
2457        kernel_size = 5
2458        # kernel_size = 7
```

```
filters = [8, 16, 32, 32]
bn = False
wn = False
wd = 0.005
beta1 = 0.5
beta2 = 0.5

model_prueba = UNetEx(3, 1, filters=filters,
↪  kernel_size=kernel_size, batch_norm=bn, weight_norm=wn)
# model_prueba.load_state_dict(torch.load(path +
↪  "Modelo_geometries_vx15.py",
↪  map_location=torch.device('cpu')))
model_prueba.load_state_dict(torch.load(path +
↪  "Modelo_geometries_vy207.py",
↪  map_location=torch.device('cpu')))
# model_prueba.load_state_dict(torch.load(path +
↪  "Modelo_geometries_p205.py",
↪  map_location=torch.device('cpu')))
model_prueba.eval()


path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias/
↪  Resultados_first_sample/tests_first_sample/"
geom_index = [1, 63, 121, 723, 1305, 1923]
geometry = ["circle", "square", "triangle_eq", "rectangle",
↪  "ellipse", "triangle"]
cfd_all = np.zeros([len(geometry)*50, 1, 172, 79])
out_all = np.zeros([len(geometry)*50, 1, 172, 79])
for g_i in range(len(geometry)):
    # cfd = pickle.load(open(path + "Test geometrias no
    ↪  vistas/dataY_1300_1001_1050.pkl", "rb"))
    cfd = pickle.load(open("E:/NuevasGeometrias/Y_geom" +
    ↪  str(geom_index[g_i]) + "_interpolated.pkl", "rb"))
    # filepaths = [f for f in listdir(str(path)) if
    ↪  f.endswith('.pkl')]
    # for file in filepaths:
    #     with open(path + file, 'rb') as f:
    #         input_data.append(pickle.load(f))
    input_data = []
```

```
2486            input_data.append(pickle.load(open(path +
                ↪  "input_data_geometry_" + geometry[g_i] +
                ↪  "_size_factor_1_rotation_angle_1_variable_vx.pkl",
                ↪  "rb")))
2487            x = np.zeros([1,3,172,79])
2488            x[:,-1,:,:] = input_data[0][0][:,1,:,:].detach().numpy()
2489            x[:,0,:,:] = input_data[0][1]
2490            x[:,1,:,:] = input_data[0][2]
2491
2492                # with open(simulation_directory + "plots_" + velocity
                    ↪  + "ms_" + variable + "_Final_v2" + str(n_test) +
                    ↪  ".pkl", "wb") as file:
2493                    # pickle.dump([out, error, y], file)
2494
2495            variable = 'vy'
2496            n_instantes = 50
2497            out2, duration = test_CNN(n_instantes, x, 0)
2498            # cfd = cfd[:,[1,2,0],:,:]
2499            cfd = torch.tensor(np.moveaxis(cfd, [2,3], [3,2]))
2500            cfd = cfd[:,2:3,:,:]
2501            error = abs(cfd[:n_instantes,:,:,:] - out2)
2502            s = [0, 4, 9, 14, 19, 29, 39, 49]
2503            for a in s:
2504                # visualize2(out2.detach().numpy(), s, filepaths[0])
2505                visualize(cfd.detach().numpy(), out2.detach().numpy(),
2506                          error.detach().numpy(), a, variable, 1, "5")
2507
2508            path2 = "E:/NuevasGeometrias/
                ↪  Resultados_NuevasGeometrias/test_nuevas_geometrias/"
2509            with open(path2 + "test_" + geometry[g_i] + '_' + variable
                ↪  + ".pkl", "wb") as file:
2510                pickle.dump([cfd.detach().numpy()[:n_instantes],
                    ↪  out2.detach().numpy(),
2511                          error.detach().numpy()], file)
2512
2513            cfd_all[g_i*50:(g_i+1)*50] = cfd[:50].detach().numpy()
2514            out_all[g_i*50:(g_i+1)*50] = out2.detach().numpy()
2515
2516        # cfd_all = np.array(cfd_all[:].detach().numpy())
2517        # out_all = np.array(out_all.detach().numpy())
2518
```

```
    # mean_error_vx =
    ↪  np.mean(error[:,0,:,:].cpu().detach().numpy())
    # mean_error_vy =
    ↪  np.mean(error[:,1,:,:].cpu().detach().numpy())
    # mean_error_p = np.mean(error[:,2,:,:].cpu().detach().numpy())

    mean_value_cfd = np.mean(cfd_all[:,0,:,:])
    std_cfd = np.std(cfd_all[:,0,:,:])

    mean_value_cnn = np.mean(out_all[:,0,:,:])
    std_cnn = np.std(out_all[:,0,:,:])

    sample10 = [x+10 for x in sample1]
    sample15 = [x+15 for x in sample1]
    sample20 = [x+20 for x in sample1]
    sample30 = [x+30 for x in sample1]
    sample40 = [x+40 for x in sample1]
    sample50 = [x+50 for x in sample1]
    sample = [sample1, sample5, sample10, sample15, sample20,
    ↪  sample30,
              sample40, sample50]
    samples = [1, 5, 10, 15, 20, 30, 40, 50]

    for s in range(len(sample)):
        cfd = cfd_all[sample[s]]
        out = out_all[sample[s]]
        gen_data_distribution(cfd, out, samples[s])
```

Test del modelo neuronal y análisis de los resultados de la CNN que predice el primer instante para las geometrías variables:

```
import pickle
import numpy as np
import math
import random
from matplotlib import pyplot as plt
import matplotlib.gridspec as gridspec
import statistics as st
from sklearn import preprocessing
from os import listdir
from tabulate import tabulate
import json
```

```
import pandas as pd


def visualize(sample_y, out_y, error, s, variable, n_sample,
    velocity):

    minu = np.min(sample_y[s, :, :])
    maxu = np.max(sample_y[s, :, :])

    # minv = np.min(sample_y[s, 1, :, :])
    # maxv = np.max(sample_y[s, 1, :, :])

    # minp = np.min(sample_y[s, 2, :, :])
    # maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, :, :])
    maxeu = np.max(error[s, :, :])

    # minev = np.min(error[s, 1, :, :])
    # maxev = np.max(error[s, 1, :, :])

    # minep = np.min(error[s, 2, :, :])
    # maxep = np.max(error[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
     + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow(np.transpose(sample_y[s, :, :]), cmap='jet', vmin =
     minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel(variable, fontsize=18)
    plt.subplot(3, 3, 2)
    plt.title('CNN', fontsize=18)
    plt.imshow(np.transpose(out_y[s, :, :]), cmap='jet', vmin =
     minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.subplot(3, 3, 3)
    plt.title('Error', fontsize=18)
```

```
2592      plt.imshow(np.transpose(error[s, :, :]), cmap='jet', vmin =
          ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
2593      plt.colorbar(orientation='horizontal')
2594
2595      # plt.subplot(3, 3, 4)
2596      # plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet',
          ↪  vmin = minv, vmax = maxv, origin='lower',
          ↪  extent=[0,172,0,79])
2597      # plt.colorbar(orientation='horizontal')
2598      # plt.ylabel('Uy', fontsize=18)
2599      # plt.subplot(3, 3, 5)
2600      # plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin
          ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2601      # plt.colorbar(orientation='horizontal')
2602      # plt.subplot(3, 3, 6)
2603      # plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin
          ↪  = minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
2604      # plt.colorbar(orientation='horizontal')
2605
2606
2607      # plt.subplot(3, 3, 7)
2608      # plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet',
          ↪  vmin = minp, vmax = maxp, origin='lower',
          ↪  extent=[0,172,0,79])
2609      # plt.colorbar(orientation='horizontal')
2610      # plt.ylabel('p', fontsize=18)
2611      # plt.subplot(3, 3, 8)
2612      # plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin
          ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2613      # plt.colorbar(orientation='horizontal')
2614      # plt.subplot(3, 3, 9)
2615      # plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin
          ↪  = minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
2616      # plt.colorbar(orientation='horizontal')
2617      plt.tight_layout()
2618      plt.show()
2619
2620  def visualize3(sample_y, s):
2621
2622      minu = np.min(sample_y[s, 0, :, :])
2623      maxu = np.max(sample_y[s, 0, :, :])
2624
```

```
    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    fig.set_size_inches(15, 10)
    plt.suptitle("Time instant " + str(s+1), fontsize=20)
    plt.subplot(3, 1, 1)
    plt.title('CFD', fontsize=18)
    plt.imshow((sample_y[s, 0, :, :]), cmap='jet', vmin = minu,
    ↪  vmax = maxu, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Ux', fontsize=18)

    plt.subplot(3, 1, 2)
    plt.imshow((sample_y[s, 1, :, :]), cmap='jet', vmin = minv,
    ↪  vmax = maxv, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('Uy', fontsize=18)


    plt.subplot(3, 1, 3)
    plt.imshow((sample_y[s, 2, :, :]), cmap='jet', vmin = minp,
    ↪  vmax = maxp, origin='lower', extent=[0,172,0,79])
    plt.colorbar(orientation='horizontal')
    plt.ylabel('p', fontsize=18)
    plt.tight_layout()
    plt.show()

def visualize_rel_error(sample_y, out_y, error, s, input_velocity
↪  ):

    # Visualize function with relative error added

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])
```

```
2663
2664     minp = np.min(sample_y[s, 2, :, :])
2665     maxp = np.max(sample_y[s, 2, :, :])
2666
2667     mineu = np.min(error[s, 0, :, :])
2668     maxeu = np.max(error[s, 0, :, :])
2669
2670     minev = np.min(error[s, 1, :, :])
2671     maxev = np.max(error[s, 1, :, :])
2672
2673     minep = np.min(error[s, 2, :, :])
2674     maxep = np.max(error[s, 2, :, :])
2675
2676     plt.figure()
2677     fig = plt.gcf()
2678     fig.set_size_inches(15, 10)
2679     plt.suptitle("Time instant " + str(s+1), fontsize=20)
2680     plt.subplot(3, 4, 1)
2681     plt.title('CFD', fontsize=18)
2682     plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
      ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2683     plt.colorbar(orientation='horizontal')
2684     plt.ylabel('Ux', fontsize=18)
2685     plt.subplot(3, 4, 2)
2686     plt.title('CNN', fontsize=18)
2687     plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
      ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2688     plt.colorbar(orientation='horizontal')
2689     plt.subplot(3, 4, 3)
2690     plt.title('Error', fontsize=18)
2691     plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
      ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
2692     plt.colorbar(orientation='horizontal')
2693     plt.subplot(3, 4, 4)
2694
2695
2696     plt.subplot(3, 4, 5)
2697     plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet', vmin
      ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2698     plt.colorbar(orientation='horizontal')
2699     plt.ylabel('Uy', fontsize=18)
2700     plt.subplot(3, 4, 6)
```

```
2701      plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin =
          ↪  minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2702      plt.colorbar(orientation='horizontal')
2703      plt.subplot(3, 4, 7)
2704      plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin =
          ↪  minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
2705      plt.colorbar(orientation='horizontal')
2706      plt.subplot(3, 4, 8)
2707
2708
2709      plt.subplot(3, 4, 9)
2710      plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet', vmin
          ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2711      plt.colorbar(orientation='horizontal')
2712      plt.ylabel('p', fontsize=18)
2713      plt.subplot(3, 4, 10)
2714      plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin =
          ↪  minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2715      plt.colorbar(orientation='horizontal')
2716      plt.subplot(3, 4, 11)
2717      plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin =
          ↪  minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
2718      plt.colorbar(orientation='horizontal')
2719      plt.subplot(3, 4, 12)
2720      plt.tight_layout()
2721      plt.show()
2722
2723  def test_mean_error(plots):
2724      error = np.zeros([len(plots), 50]) # Habría que poner esto pero
          ↪  me da fallo --> len(plots[id_test][2])])
2725      max_error = error
2726      variance = error
2727      col = 0
2728      for key in plots:
2729          for row in range(50): #
2730              error[col, row] = np.mean(plots[key][2][row],
                  ↪  dtype=np.float64)
2731              max_error[col, row] = np.max(plots[key][2][row])
2732              variance[col, row] = np.var(plots[key][2][row], ddof =
                  ↪  1)
2733          col = col + 1
2734      mean_error = np.mean(error, axis = 0, dtype=np.float64)
```

```
    max_error_all = np.max(max_error, axis = 0)
    variance_all = np.mean(variance, axis = 0)
    me_error = np.mean(mean_error)
    v_all = np.mean(variance_all)
    ma_error = np.max(max_error_all)
    return mean_error, max_error_all, variance_all, me_error, v_all,
    ↪  ma_error

def plot_feature(feature_type, upper_limit, legend, plot_title,
↪  y_label):
    for i in range(10):
        plt.plot(feature_type[i], '--o', markersize = 3)

    plt.grid(axis='both')
    plt.ylabel(y_label)
    plt.ylim((0,upper_limit))
    plt.legend(legend)
    plt.title(plot_title)
    plt.show()

def get_sub(x):
    normal =
    ↪  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-=()"
    sub_s = "CDGQwZw"
    res = x.maketrans(''.join(normal), ''.join(sub_s))
    return x.translate(res)

def gen_data_distribution(CFD, out):
## Distribucion de los datos:
    CFD_array = np.array(CFD)
    out_array = np.array(out)

    fig, ax = plt.subplots()
    fig.set_size_inches(7, 4)
    plt.title('Histogram u{}'.format(get_sub('x')), fontsize = 20)
    fig.tight_layout()
    fig.subplots_adjust(top=0.95)
    n, bins, patches =
    ↪  plt.hist(CFD_array[:,0,:,:].reshape(10*79*172),
                                200, density=False, facecolor='r',
                                 alpha=0.5)
```

```
2772    n, bins, patches =
        ↪  plt.hist(out_array[:,0,:,:].reshape(10*79*172),
2773                                200, density=False, facecolor='b',
2774                                alpha=0.5)
2775    plt.xlabel('u{}'.format(get_sub('x')) + ' [m/s]', fontsize =
        ↪  18)
2776    plt.ylabel('Nº of points', fontsize = 18)
2777    plt.rc('xtick', labelsize = 14)
2778    plt.rc('ytick', labelsize = 14)
2779    plt.legend(['CFD', 'CNN'], fontsize = 18)
2780    ax.set_yscale('log')
2781    # ax.set_xscale('log')
2782    plt.grid(True)
2783    # fig.savefig('./ResultadosFinales/data_distrubution_vx_' +
        ↪  vel_title + '.svg',  bbox_inches='tight')
2784    plt.show()
2785
2786    fig, ax = plt.subplots()
2787    fig.set_size_inches(7, 4)
2788    plt.title('Histogram u{}'.format(get_sub('y')), fontsize = 20)
2789    fig.tight_layout()
2790    n, bins, patches =
        ↪  plt.hist(CFD_array[:,1,:,:].reshape(10*79*172),
2791                                200, density=False, facecolor='r',
2792                                alpha=0.5)
2793    n, bins, patches =
        ↪  plt.hist(out_array[:,1,:,:].reshape(10*79*172),
2794                                200, density=False, facecolor='b',
2795                                alpha=0.5)
2796    plt.xlabel('u{}'.format(get_sub('y')) + ' [m/s]', fontsize =
        ↪  18)
2797    # plt.legend(['CFD', 'CNN'])
2798    plt.rc('xtick', labelsize = 14)
2799    plt.rc('ytick', labelsize = 14)
2800    ax.set_yscale('log')
2801    plt.grid(True)
2802    # fig.savefig('./ResultadosFinales/data_distrubution_vy_' +
        ↪  vel_title + '.svg', bbox_inches='tight')
2803    plt.show()
2804
2805
2806    fig, ax = plt.subplots()
```

```
2807      fig.set_size_inches(7, 4)
2808      plt.title('Histogram p', fontsize = 20)
2809      fig.tight_layout()
2810      n, bins, patches =
          ↪  plt.hist(CFD_array[:,2,:,:].reshape(10*79*172),
2811                                  200, density=False, facecolor='r',
2812                                  alpha=0.5)
2813      n, bins, patches =
          ↪  plt.hist(out_array[:,2,:,:].reshape(10*79*172),
2814                                  200, density=False, facecolor='b',
2815                                  alpha=0.5)
2816
2817      plt.xlabel('p [Pa]', fontsize = 18)
2818      plt.rc('xtick', labelsize = 14)
2819      plt.rc('ytick', labelsize = 14)
2820      # plt.legend(['CFD', 'CNN'])
2821      plt.grid(True)
2822      ax.set_yscale('log')
2823      # fig.savefig('./ResultadosFinales/data_distrubution_p_' +
          ↪  vel_title + '.svg', bbox_inches='tight')
2824      plt.show()
2825
2826  if __name__ == "__main__":
2827
2828      # mean_error_list = []
2829      # max_error_list = []
2830      # variance_list = []
2831      # me_error_list = []
2832      # var_list = []
2833      # ma_error_list = []
2834
2835      # plots = []
2836      training_info = []
2837      path = "E:/NuevasGeometrias/Resultados_NuevasGeometrias
          ↪  /Resultados_first_sample/"
2838
2839      filepaths = [f for f in listdir(str(path) + "Estadisticas/") if
          ↪  f.endswith('.json')]
2840      for file in filepaths:
2841          mean_error, max_error, variance, me_error, var_, ma_error =
              ↪  pickle.load(open(path + "Estadisticas/" + file, "rb"))
2842          # mean_error_list.append(mean_error)
```

```
2843            # max_error_list.append(max_error)
2844            # variance_list.append(variance)
2845            # me_error_list.append(me_error)
2846            # var_list.append(var_)
2847            # ma_error_list.append(ma_error)
2848        filepaths = [f for f in listdir(str(path)) if
        ↪  f.endswith('.json')]
2849        for file in filepaths:
2850            data = json.load(open(path + file, "rb"))
2851            training_info.append(data)
2852
2853        xlabel = []
2854        training_duration = []
2855        for i in range(len(training_info)):
2856            xlabel.insert(-1 ,'Test nº' + str(i+3))
2857            training_duration.append(training_info[i][1]/3600)
2858        plt.plot(training_duration, '--bo', markersize = 3)
2859        plt.grid(axis='both')
2860        # plt.xlabel(xlabel)
2861        plt.ylabel('Time (min)')
2862        plt.title('Training duration of each model')
2863        plt.show()
2864
2865
2866
2867        table_data = []
2868        col_names = ["Id", "Train time (h)", "Num_epochs", "Lr",
        ↪  "Kernel_size", "Batch_size", "Architecture",
2869                     "MeErr_vx", "MeErr_vy", "MeErr_p", "MaxErr_vx",
                     ↪  "MaxErr_vy", "MaxErr_p"]
2870        for i in range(len(mean_error)):
2871            table_data.append([filepaths[i][19:22],
2872                               training_info[i][1]/3600,
2873
                               ↪  training_info[i][0]["net_param"]['num_epochs'],
2874
                               ↪  training_info[i][0]["net_param"]['learning_ratio'],
2875
                               ↪  training_info[i][0]["net_param"]['kernel_size'],
2876
                               ↪  training_info[i][0]["net_param"]['batch_size'],
```

```
2877
                                ↪  training_info[i][0]["net_param"]["arquitecture"],
2878                            mean_error[i][0],  mean_error[i][1],
                                ↪  mean_error[i][2],
2879                            max_error[i][0], max_error[i][1],
                                ↪  max_error[i][2]])
2880        table = pd.DataFrame(table_data, columns = col_names)
2881        # table = tabulate(table_data, headers = col_names)
2882        print(table)
2883
2884        table.to_csv(path + 'training_information.csv')
2885
2886        table_less_mean_error =
            ↪  table.sort_values(by=["MeErr_vx"])#[:10]
2887        table_less_max_error =
            ↪  table.sort_values(by=["MaxErr_vx"])#[:10]
2888        print(table_less_mean_error)
2889        print(table_less_max_error)
2890
2891        # plot_feature(mean_error_list, 70, legend, 'Mean error for
            ↪  each prediction',
2892        #              'Mean error (m/s)')
2893        # plot_feature(max_error_list, 40, legend, 'Maximum error for
            ↪  each prediction',
2894        #              'Max error (m/s)')
2895        # plot_feature(variance_list, 10, legend, 'Variance for each
            ↪  prediction',
2896        #              'Variance (m^2/s^2)')
2897
2898        ### Histograms
2899
2900        filepaths = [f for f in listdir(str(path)) if
            ↪  f.endswith('.pkl')]
2901        best_model_plots = pickle.load(open(path + filepaths[9], "rb"))
2902
2903        cfd = best_model_plots[2]
2904        out = best_model_plots[0]
2905
2906        gen_data_distribution(cfd, out)

2907  import os
2908  import json
```

```
import torch
import pickle
import time
from train_functions import *
import torch.optim as optim
from torch.utils.data import TensorDataset
from Models.UNetEx import UNetEx
from Models.UNet import UNet
import numpy as np
from matplotlib import pyplot as plt
from scipy.io import loadmat
os.environ['KMP_DUPLICATE_LIB_OK']='True'


def visualize(sample_y, out_y, error, s, variable, n_sample,
    velocity):

    minu = np.min(sample_y[s, 0, :, :])
    maxu = np.max(sample_y[s, 0, :, :])

    minv = np.min(sample_y[s, 1, :, :])
    maxv = np.max(sample_y[s, 1, :, :])

    minp = np.min(sample_y[s, 2, :, :])
    maxp = np.max(sample_y[s, 2, :, :])

    mineu = np.min(error[s, 0, :, :])
    maxeu = np.max(error[s, 0, :, :])

    minev = np.min(error[s, 1, :, :])
    maxev = np.max(error[s, 1, :, :])

    minep = np.min(error[s, 2, :, :])
    maxep = np.max(error[s, 2, :, :])

    plt.figure()
    fig = plt.gcf()
    plt.suptitle("Input velocity: " + velocity + " ms. Sample nº: "
     + str(n_sample+s), fontsize=20)
    fig.set_size_inches(15, 10)
    plt.subplot(3, 3, 1)
    plt.title('CFD', fontsize=18)
```

```
2949    plt.imshow(np.transpose(sample_y[s, 0, :, :]), cmap='jet', vmin
        ↪  = minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2950    plt.colorbar(orientation='horizontal')
2951    plt.ylabel(variable, fontsize=18)
2952    plt.subplot(3, 3, 2)
2953    plt.title('CNN', fontsize=18)
2954    plt.imshow(np.transpose(out_y[s, 0, :, :]), cmap='jet', vmin =
        ↪  minu, vmax = maxu, origin='lower', extent=[0,172,0,79])
2955    plt.colorbar(orientation='horizontal')
2956    plt.subplot(3, 3, 3)
2957    plt.title('Error', fontsize=18)
2958    plt.imshow(np.transpose(error[s, 0, :, :]), cmap='jet', vmin =
        ↪  mineu, vmax = maxeu, origin='lower', extent=[0,172,0,79])
2959    plt.colorbar(orientation='horizontal')
2960
2961    plt.subplot(3, 3, 4)
2962    plt.imshow(np.transpose(sample_y[s, 1, :, :]), cmap='jet', vmin
        ↪  = minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2963    plt.colorbar(orientation='horizontal')
2964    plt.ylabel('Uy', fontsize=18)
2965    plt.subplot(3, 3, 5)
2966    plt.imshow(np.transpose(out_y[s, 1, :, :]), cmap='jet', vmin =
        ↪  minv, vmax = maxv, origin='lower', extent=[0,172,0,79])
2967    plt.colorbar(orientation='horizontal')
2968    plt.subplot(3, 3, 6)
2969    plt.imshow(np.transpose(error[s, 1, :, :]), cmap='jet', vmin =
        ↪  minev, vmax = maxev, origin='lower', extent=[0,172,0,79])
2970    plt.colorbar(orientation='horizontal')
2971
2972
2973    plt.subplot(3, 3, 7)
2974    plt.imshow(np.transpose(sample_y[s, 2, :, :]), cmap='jet', vmin
        ↪  = minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2975    plt.colorbar(orientation='horizontal')
2976    plt.ylabel('p', fontsize=18)
2977    plt.subplot(3, 3, 8)
2978    plt.imshow(np.transpose(out_y[s, 2, :, :]), cmap='jet', vmin =
        ↪  minp, vmax = maxp, origin='lower', extent=[0,172,0,79])
2979    plt.colorbar(orientation='horizontal')
2980    plt.subplot(3, 3, 9)
2981    plt.imshow(np.transpose(error[s, 2, :, :]), cmap='jet', vmin =
        ↪  minep, vmax = maxep, origin='lower', extent=[0,172,0,79])
```

```
2982        plt.colorbar(orientation='horizontal')
2983        plt.tight_layout()
2984        plt.show()
2985
2986
2987    def visualize2(sample_y, s, geometry, rotation_angle, size_factor,
        ↪  variable, path):
2988
2989        plt.figure()
2990        fig = plt.gcf()
2991        # fig.set_size_inches(15, 10)
2992
2993        minu = np.min(sample_y[s, variable, :, :])
2994        maxu = np.max(sample_y[s, variable, :, :])
2995
2996        plt.imshow(np.transpose(sample_y[s, variable, :, :]),
            ↪  cmap='jet',
2997                   vmin = minu, vmax = maxu, origin='lower',
                    ↪  extent=[0,172,0,79])
2998        plt.colorbar(orientation='horizontal')
2999        plt.ylabel('vx', fontsize = 14)
3000        plt.title("Test " + geometry + ", size factor: " +
            ↪  str(size_factor)
3001                  + ", rot. angle: " + str(rotation_angle), fontsize =
                    ↪  14)
3002        plt.tight_layout()
3003        plt.savefig(path + "geometry_" + geometry + "_size_factor_" +
            ↪  str(size_factor) + "_rotation_angle_"
3004                    + str(rotation_angle) + "_variable_vx" + ".png" ,
                     ↪  bbox_inches = 'tight')
3005        plt.show()
3006
3007    def get_sub(x):
3008        normal =
            ↪  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-=()"
3009        sub_s = "CDGQwZw"
3010        res = x.maketrans(''.join(normal), ''.join(sub_s))
3011        return x.translate(res)
3012
3013    def gen_data_distribution(CFD, out):
3014    ## Distribucion de los datos:
3015        CFD_array = np.array(CFD)
```

```
3016    out_array = np.array(out)
3017
3018    fig, ax = plt.subplots()
3019    fig.set_size_inches(7, 4)
3020    plt.title('Histogram u{}'.format(get_sub('x')), fontsize = 20)
3021    fig.tight_layout()
3022    fig.subplots_adjust(top=0.95)
3023    n, bins, patches = plt.hist(CFD_array[:,0,:,:].reshape(79*172),
3024                                200, density=False, facecolor='r',
3025                                 alpha=0.5)
3026    n, bins, patches = plt.hist(out_array[:,0,:,:].reshape(79*172),
3027                                 200, density=False, facecolor='b',
3028                                 alpha=0.5)
3029    plt.xlabel('u{}'.format(get_sub('x')) + ' [m/s]', fontsize =
        ↪  18)
3030    plt.ylabel('Nº of points', fontsize = 18)
3031    plt.rc('xtick', labelsize = 14)
3032    plt.rc('ytick', labelsize = 14)
3033    plt.legend(['CFD', 'CNN'], fontsize = 18)
3034    ax.set_yscale('log')
3035    # ax.set_xscale('log')
3036    plt.grid(True)
3037    # fig.savefig('./ResultadosFinales/data_distrubution_vx_' +
        ↪  vel_title + '.svg',  bbox_inches='tight')
3038    plt.show()
3039
3040    fig, ax = plt.subplots()
3041    fig.set_size_inches(7, 4)
3042    plt.title('Histogram u{}'.format(get_sub('y')), fontsize = 20)
3043    fig.tight_layout()
3044    n, bins, patches = plt.hist(CFD_array[:,1,:,:].reshape(79*172),
3045                                 200, density=False, facecolor='r',
3046                                 alpha=0.5)
3047    n, bins, patches = plt.hist(out_array[:,1,:,:].reshape(79*172),
3048                                 200, density=False, facecolor='b',
3049                                 alpha=0.5)
3050    plt.xlabel('u{}'.format(get_sub('y')) + ' [m/s]', fontsize =
        ↪  18)
3051    # plt.legend(['CFD', 'CNN'])
3052    plt.rc('xtick', labelsize = 14)
3053    plt.rc('ytick', labelsize = 14)
3054    ax.set_yscale('log')
```

```
3055        plt.grid(True)
3056        # fig.savefig('./ResultadosFinales/data_distrubution_vy_' +
            ↪  vel_title + '.svg', bbox_inches='tight')
3057        plt.show()
3058
3059
3060        fig, ax = plt.subplots()
3061        fig.set_size_inches(7, 4)
3062        plt.title('Histogram p', fontsize = 20)
3063        fig.tight_layout()
3064        n, bins, patches = plt.hist(CFD_array[:,2,:,:].reshape(79*172),
3065                                    200, density=False, facecolor='r',
3066                                    alpha=0.5)
3067        n, bins, patches = plt.hist(out_array[:,2,:,:].reshape(79*172),
3068                                    200, density=False, facecolor='b',
3069                                    alpha=0.5)
3070
3071        plt.xlabel('p [Pa]', fontsize = 18)
3072        plt.rc('xtick', labelsize = 14)
3073        plt.rc('ytick', labelsize = 14)
3074        # plt.legend(['CFD', 'CNN'])
3075        plt.grid(True)
3076        ax.set_yscale('log')
3077        # fig.savefig('./ResultadosFinales/data_distrubution_p_' +
            ↪  vel_title + '.svg', bbox_inches='tight')
3078        plt.show()
3079
3080    if __name__ == "__main__":
3081
3082        path1 = "E:/NuevasGeometrias/Resultados_NuevasGeometrias/
            ↪  Resultados_first_sample/Modelos/"
3083
3084        torch.manual_seed(0)
3085        lr = 0.001
3086        kernel_size = 3
3087        filters = [8, 16, 16, 32, 32]
3088        bn = False
3089        wn = False
3090        wd = 0.005
3091        beta1 = 0.5
3092        beta2 = 0.5
3093
```

```
model_prueba = UNetEx(2, 3, filters=filters, kernel_size=kernel_size, batch_norm=bn, weight_norm=wn)
model_prueba.load_state_dict(torch.load(path1 + "Modelo_geometries_109_data_augmentation.py", map_location=torch.device('cpu')))
model_prueba.eval()

path2 = "C:/Users/alvar/Desktop/Master/Segundo/TFM/2ª parte/Datos para entrenamiento/"

geometry = ["circle", "ellipse", "square", "rectangle", "triangle", "triangle_eq"]
# geometry = ["test_ellipse"]
# geometry = ["test_square"]
# size_factor = [0.2, 0.4, 0.6, 0.8, 1.2, 1.4, 1.6, 1.8]
size_factor = [1]
# rotation_angle = [-30, -25, -20, -15, -10, -5, 5, 10, 15, 20, 25, 30]
rotation_angle = [1]
position = [1, 1]

# sdf = loadmat("SDF_geometrias_data_augmentation/Test/" + geometry + "_1size_factor" + str(size_factor) + ".mat")['Z_matrix']
# frc = loadmat("FRC_geometrias_data_augmentation/Test/" + geometry + "_1size_factor" + str(size_factor) + ".mat")['frc']

for geom in geometry:
    for size in size_factor:
        for rot in rotation_angle:
            sdf = loadmat("SDF_geometrias_data_augmentation/Test/" + geom + "_1size_factor" +
                          str(size) + "rot_angle" + str(rot) + "pos_x" + str(position[0]) +
                          "pos_y" + str(position[1]) + ".mat")['Z_matrix']
            frc = loadmat("FRC_geometrias_data_augmentation/Test/" + geom + "_1size_factor" +
```

```
                    str(size) + "rot_angle" + str(rot) +
                    ↪ "pos_x" + str(position[0]) +
                   "pos_y" + str(position[1]) +
                   ↪ ".mat")['frc']

# plt.figure()
# plt.imshow(sdf.T)
# plt.figure()
# plt.imshow(frc.T)

x = np.zeros([1,2,172,79])
x[0,0,:,:] = sdf
x[0,1,:,:] = frc

ti = time.time()
out = model_prueba(torch.FloatTensor(x))
tout = time.time()
duration = tout - ti
# path3 =
↪ "E:/NuevasGeometrias/Resultados_NuevasGeometrias/
↪ Resultados_first_sample/test_first_sample/"
# s = 0
# visualize2(out.cpu().detach().numpy(), s, geom,
↪ rot, size, 0, path3)

# directory = path1 + "tests_first_sample/"
# if not os.path.exists(directory):
#     os.makedirs(directory)

path3 =
↪ "E:/NuevasGeometrias/Resultados_NuevasGeometrias/
↪ Resultados_first_sample/tests_first_sample/"

s = 0
visualize2(out.cpu().detach().numpy(), s, geom, rot,
↪ size, 1, path3)
variable = 'vy'
with open(path3 + "input_data_geometry_" +
↪ str(geom)
```

```
                + "_size_factor_" + str(size) +
                ↪ "_rotation_angle_" + str(rot) +
                ↪ "_variable_" + variable + ".pkl",
                ↪ "wb") as f:
        pickle.dump([out, sdf, frc], f)


cfd = pickle.load(open(path3 + "Test geometrias no
↪ vistas/dataY_367_1001.pkl", "rb"))
cfd=cfd[:,[1,2,0],:,:]
cfd = torch.tensor(np.moveaxis(cfd, [2,3], [3,2]))
error = abs(cfd-out)
visualize(cfd.cpu().detach().numpy(),
↪ out.cpu().detach().numpy(),
          error.cpu().detach().numpy(), s, "vx", 0,
          ↪ "5")

mean_error_vx =
↪ np.mean(error[:,0,:,:].cpu().detach().numpy())
mean_error_vy =
↪ np.mean(error[:,1,:,:].cpu().detach().numpy())
mean_error_p =
↪ np.mean(error[:,2,:,:].cpu().detach().numpy())

mean_value_cfd_vx =
↪ np.mean(cfd[:,0,:,:].cpu().detach().numpy())
mean_value_cfd_vy =
↪ np.mean(cfd[:,1,:,:].cpu().detach().numpy())
mean_value_cfd_p =
↪ np.mean(cfd[:,2,:,:].cpu().detach().numpy())

mean_value_cnn_vx =
↪ np.mean(out[:,0,:,:].cpu().detach().numpy())
mean_value_cnn_vy =
↪ np.mean(out[:,1,:,:].cpu().detach().numpy())
mean_value_cnn_p =
↪ np.mean(out[:,2,:,:].cpu().detach().numpy())

gen_data_distribution(cfd.detach().numpy(),
↪ out.detach().numpy())
```