Politecnico di Torino

MASTER THESIS

Use of Convolutional Neural Network for multi-fidelity CFD problems

Candidate:
Sebastiano G. I. Marcì

Supervisors:
Prof. Andrea Ferrero
Ph.D. Luca Muscarà
Ph.D. Lorenzo Folcarelli

Department of Mechanical and Aerospace Engineering

October 2025

Abstract

Use of Convolutional Neural Network for multi-fidelity CFD problems

Candidate: Sebastiano G. I. Marcì

POLITECNICO DI TORINO
Master degree in Aerospace Engineering
Department of Mechanical and Aerospace Engineering

In recent years, there has been a growing interest in applying deep learning techniques to Computational Fluid Dynamics (CFD), particularly for accelerating simulations and enhancing data resolution. This work explores the use of Convolutional Neural Networks (CNNs) to reconstruct high-resolution CFD fields from low-resolution inputs. A U-Net architecture was selected due to its proven ability to capture both global context and fine-scale features through its encoder-decoder structure combined with skip connections, facilitating detailed reconstructions. The study focuses on two canonical CFD problems: a bump-in-a-duct configuration and a hydrogen combustion chamber. These cases were chosen to encompass both non-reactive and reactive flow regimes and to test the generalization capability of the model under different physical conditions. The network was trained to predict high-resolution pressure, axial velocity, and heat of reaction fields from their corresponding low-resolution versions, thus performing a super-resolution task. Due to the limited size of the available dataset, several regularization techniques were employed to improve model robustness, including Dropout layers and Batch Normalization. Additionally, different data normalization strategies were tested, such as Z-normalization and Min-Max scaling, to enhance training stability and ensure faster convergence. A multi-network approach was also developed to separately handle the prediction of pressure fields and that of axial velocity and heat release rate, mitigating the risk of feature mixing and allowing each network to specialize in learning specific patterns. The datasets were generated using two solvers: a custom Fortran-based code solving the non-dimensional Euler equations, and the commercial CFD software Ansys Fluent for more complex reactive flow simulations. This combination enabled the creation of a multi-fidelity database, covering a wide range of fluid dynamic behaviors and ensuring better generalization capabilities during training and validation phases. The training process involved careful hyperparameter optimization, early stopping strategies, and loss function selection to prevent overfitting and maximize prediction quality. Quantitative assessments demonstrated prediction errors on the order of

a few percent, while qualitative analyses confirmed the ability of the models to accurately reconstruct the flow structures, even in complex reacting flows. The final phase of the project focused on identifying potential future developments of the proposed framework, particularly exploring the capability of the U-Net architecture to address a broader class of problems. One promising direction involves extending the methodology to correct two-dimensional solutions in order to approximate three-dimensional flow fields. This approach consist of training the network with the 2D planes extracted from the full 3D simulations. Furthermore developments could be searching for techniques aimed at enhancing training performance such as data augmentation, especially in cases involving limited datasets could be implemented to improve the model's generalization and robustness. The individual contribution to this thesis project includes the generation of the datasets, the design and full implementation of the U-Net architecture within a complete Python framework featuring a structured-to-unstructured (and vice versa) grid interpolator, the definition of training strategies, the extension to a multi-network architecture, and the systematic evaluation of the model performance. The work highlights the promising role of deep learning techniques, particularly CNN-based surrogate models, in enhancing CFD simulation capabilities and accelerating future engineering workflows by enabling fast and accurate predictions across different fidelity levels.

Acknowledgements

I would like to express my sincere gratitude to Professor Andrea Ferrero and to PhD students Luca Muscarà and Lorenzo Folcarelli for giving me the opportunity to explore such a stimulating and challenging topic. I also wish to thank them for their constant availability and proactive attitude during our discussion sessions, which have been extremely valuable throughout this journey.
Finally, I would like to thank all those who have supported me over the years and have always been there for me my family, friends, and colleagues. Their presence has been essential in reaching this important milestone.

Contents

List of Figures

List of Tables

Chapter 1

The Rise and Role of Deep Learning in Engineering

Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL), and Neural Networks (NNs) are often used interchangeably in common discourse, but they represent different layers of abstraction and specialization in the field of intelligent systems. Understanding their relationship is crucial, especially when analyzing the motivations behind the recent surge of deep learning applications in engineering.

1.1 From Artificial Intelligence to Deep Learning

Artificial Intelligence is the overarching field that encompasses any technique enabling machines to mimic human intelligence. This includes logic-based systems, symbolic reasoning, expert systems, and data-driven learning approaches. Within AI, Machine Learning emerged as a major subfield focused on developing algorithms that allow machines to improve their performance by learning from data, **without being explicitly programmed.**

Machine learning systems are characterized by their ability to generalize from past examples. However, for many years, their success relied heavily on the quality of input features, a process known as *feature engineering*. Human experts were responsible for designing the most relevant, informative, and quantifiable characteristics of raw data in order to make it interpretable for models such as decision trees, support vector machines (SVM), or shallow neural networks.

For instance, in a typical classification task, data points would be projected into a space defined by two handcrafted features (e.g., (x, y)), and a classifier would learn a rule to separate different classes by a decision boundary. This traditional pipeline was effective but limited in scalability and generalization, especially in high-dimensional data like images, speech, or sensor data.

1.2 The Shift: Why Deep Learning?

Around the early 2010s, a dramatic shift occurred in how learning systems were built. Deep Learning, a subfield of machine learning based on deep

artificial neural networks, began to outperform traditional approaches across a range of tasks, from image recognition to natural language understanding and even scientific computing [12].

This transition was not purely theoretical; it was driven by three practical revolutions:

- **The availability of massive labeled datasets**, such as ImageNet, enabled models to learn complex patterns from millions of examples [13].
- **The rise of GPU computing** provided the computational power needed to train large neural networks efficiently.
- **Advances in training techniques** (e.g., better weight initialization, regularization methods like dropout, and new activation functions like ReLU) made it feasible to train deep architectures that were previously unstable or inefficient [14].

Most importantly, deep learning models introduced a powerful innovation: **automatic feature learning**. Instead of relying on handcrafted features, deep networks learn *hierarchical representations* of data — where lower layers capture simple patterns (e.g., edges in an image), and deeper layers compose these into increasingly abstract concepts (e.g., object parts or categories). This eliminated the bottleneck of manual feature engineering and allowed models to generalize better across tasks.

"Deep learning shifted the focus from manual feature engineering to automated feature learning."[12]

1.3 Clarifying the Terminology

To better understand this evolution, it is useful to clarify how the four main terms — AI, ML, DL, and NNs — are related. These concepts are hierarchically nested:

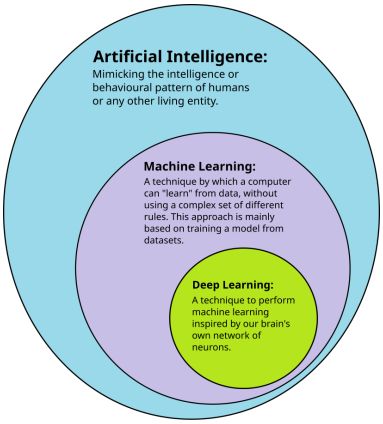

FIGURE 1.1: Diagram showing the relationship between Artificial Intelligence, Machine Learning and Deep Learning. Image from [1].

`Artificial Intelligence` ⊃ `Machine Learning` ⊃ `Deep Learning` ⊃ `Neural Networks`

- **AI** includes all methods that aim to replicate intelligent behavior, whether symbolic (based on logic and explicit rules) or data-driven (based on pattern recognition and statistical inference). It is important to note that many modern AI systems, especially those based on machine learning, are fundamentally *predictive* rather than *reasoning-based*. That is, they learn correlations from data to forecast outcomes, but they do not perform reasoning in the traditional symbolic or causal sense [15].

  "Prediction is not reasoning"

- **ML** is a subset of AI focused on learning patterns from data, enabling predictions and decisions based on experience.
- **DL** is a further specialization within ML that uses multi-layered neural networks to learn complex representations from raw data.
- **NNs** are the fundamental computational structures used in both shallow and deep learning systems, inspired by the biological structure of the brain.

1.4 Why It Matters for Engineering

The adoption of deep learning in engineering stems from its ability to address longstanding limitations in traditional modeling approaches. Engineering problems often involve complex physical systems governed by partial differential equations, nonlinear interactions, and vast volumes of sensor or simulation data. Traditional solvers and empirical models are sometimes too slow or inaccurate for real-time applications or design optimization.

Deep learning offers a data-driven, flexible, and efficient alternative:

- It reduces the reliance on manual feature design and physical assumptions.
- It provides faster inference once trained, enabling real-time deployment.
- It integrates well with existing numerical frameworks, especially through hybrid models (e.g., physics-informed neural networks).

From fluid dynamics and structural analysis to robotics and material science, deep learning is increasingly seen not just as a predictive tool, but as a core component of modern engineering methodologies.

1.5 Growth of Artificial Intelligence in the Aerospace Sector

According to a recent report by Technavio, the global Artificial Intelligence (AI) market in the aerospace sector is expected to grow significantly in the coming years. Specifically, it is estimated to increase by approximately USD 4.69 billion between 2024 and 2028, with a compound annual growth rate (CAGR) of 43.6% [2].

FIGURE 1.2: Size of the Aerospace Artificial Intelligence (AI) Market. Image from [2].

This growth is driven by the rising adoption of AI technologies for process automation, predictive maintenance, flight route optimization, and enhanced operational safety. In the defense sector, AI is increasingly applied in advanced robotics, autonomous drones, and real-time data analysis systems.

Italy plays a strategic role in this landscape, both through its participation in European space programs (such as those promoted by the European Space Agency) and the presence of companies engaged in AI-driven aerospace research and development.

Among the leading global companies active in this sector, the report highlights:

- Airbus SE
- Boeing Co.
- Lockheed Martin Corp.
- Thales SA
- IBM Corp.
- Intel Corp.

- Microsoft Corp.
- NVIDIA

These technological and industrial giants are investing heavily in the development of intelligent systems capable of processing large volumes of data from sensors, satellites, and complex avionics platforms.

1.6 Deep Learning in Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) plays a central role in modern engineering, enabling the simulation and analysis of fluid flows across a variety of applications, from aerospace and automotive design to biomedical and environmental engineering. Traditional CFD methods rely on numerically solving the Navier–Stokes equations using finite volume, finite element, or finite difference schemes. While accurate, these approaches are computationally expensive and often prohibitively slow for high-resolution problems or real-time applications.

In recent years, deep learning has emerged as a promising complementary tool for accelerating CFD simulations, reconstructing high-fidelity flow fields, and enabling data-driven turbulence modeling.

1.6.1 Motivation and Applications

The high computational cost of CFD simulations, especially in unsteady, turbulent, or three-dimensional flows, has led researchers to explore surrogate modeling approaches. Deep learning models, particularly Convolutional Neural Networks (CNNs) and encoder–decoder architectures, have been used to learn mappings between low-resolution and high-resolution flow fields, or between input conditions and output fields.

Several applications have demonstrated the feasibility and advantages of this approach:

- **Super-resolution flow reconstruction:** CNN-based models can reconstruct fine-scale structures of velocity or pressure fields from coarse or downsampled simulations [16].
- **Reduced-order modeling:** Autoencoders and recurrent neural networks (RNNs) are used to build compact representations of flow dynamics for real-time control or parameter studies [17].
- **Turbulence modeling:** Deep learning is being explored as an alternative to traditional subgrid-scale models in Large Eddy Simulation (LES) [18].

1.6.2 Advantages Over Traditional CFD

Deep learning models, once trained, can offer several advantages:

- **Speed:** Inference times are several orders of magnitude faster than solving PDEs numerically.
- **Flexibility:** The same architecture can be adapted across geometries and boundary conditions (given appropriate training data).
- **Integration:** DL models can be embedded within hybrid frameworks, such as Physics-Informed Neural Networks (PINNs) [19], which enforce physical laws during training.

1.6.3 Limitations and Challenges

Despite the promising results, several challenges remain:

- **Generalization:** Neural networks may fail to generalize to flow regimes or geometries not seen during training.
- **Data requirements:** High-fidelity CFD data are expensive to generate and may limit scalability.
- **Physical consistency:** Purely data-driven models can violate conservation laws or produce non-physical artifacts if not properly constrained.

Ongoing research is addressing these limitations through physics-aware architectures, uncertainty quantification, and transfer learning strategies. Nevertheless, deep learning is increasingly becoming a valuable tool in the CFD toolbox, complementing traditional solvers and enabling new capabilities in simulation, optimization, and control.

1.7 Introduction to Artificial Neural Networks

Artificial neural networks are among the foundational concepts in the field of machine learning. Their structure and functioning are inspired by biological neurons:

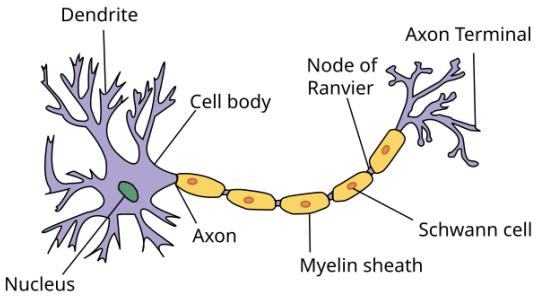

FIGURE 1.3: Biological neuron. Image from [3].

A biological neuron is mainly composed of three elements: the *soma* (cell body), the *dendrites*, which receive signals from surrounding neurons, and the *axon*, which transmits electrical impulses outward. When the sum of stimuli received by the dendrites exceeds a certain activation threshold, the neuron generates an electrical impulse, called a *spike*. This spike travels along the axon to a structure called the *synapse*, where the signal is transmitted to a subsequent neuron—known as the *post-synaptic neuron*—via neurotransmitters. Depending on the type of neurotransmitter, the signal can have an excitatory or inhibitory effect. The human brain contains approximately 10 billion neurons, each connected, on average, to 10,000 other neurons through synapses.

To mimic the information processing and learning capabilities of biological neurons, the concept of the *artificial neuron* is introduced.

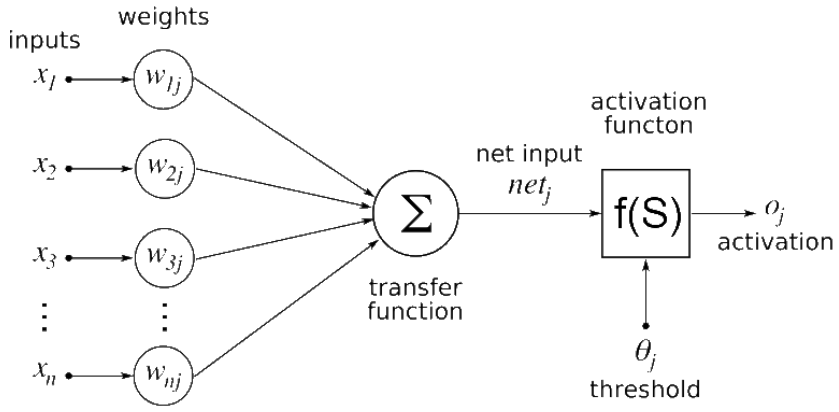

FIGURE 1.4: Artificial neuron diagram. Image from [3].

An artificial neuron receives a set of inputs $x_1, x_2, \ldots, x_n$, each associated with a *synaptic weight* $w_1, w_2, \ldots, w_n$. The weighted sum of these inputs is computed and passed through an *activation function* f, which determines the neuron's output. Depending on the chosen function, the output may be a continuous value (e.g., between 0 and 1, or between -1 and 1) or a discrete value (e.g., 0 or 1, or -1 or +1). The simplest case occurs when the activation function is the identity, in which case the output equals the weighted sum of the inputs. So, a single neuron receives multiple inputs $x_1, x_2, ..., x_n$, each multiplied by an associated weight w_i. The output is computed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right), \tag{1.1}$$

where f is a non-linear activation function (e.g., sigmoid, ReLU).

1.8 Interconnection Examples for Artificial Neurons

Artificial neurons can be interconnected in various topologies. The three main types are:

- **Fully connected network**: Every neuron is connected to all other neurons in the network. This architecture is less commonly used compared to others.

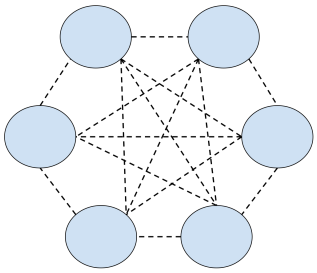

FIGURE 1.5: Fully Connected NN. Image from [4]

- **Feedforward network**: Composed of at least three layers—an *input layer*, one or more *hidden layers*, and an *output layer*. Data flows in one direction only, from input to output, without forming cycles.

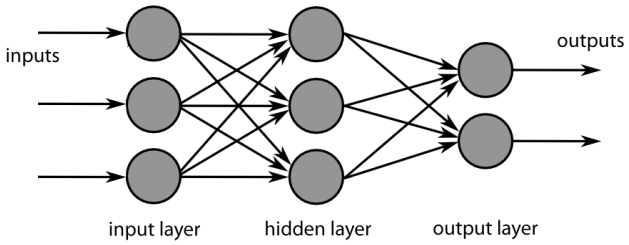

FIGURE 1.6: Recurrent NN. Image from [5].

- **Recurrent neural network (RNN)**: Characterized by feedback connections, where one or more outputs are reintroduced as inputs through a set of additional neurons. This architecture is particularly well-suited for processing signals or sequential information.

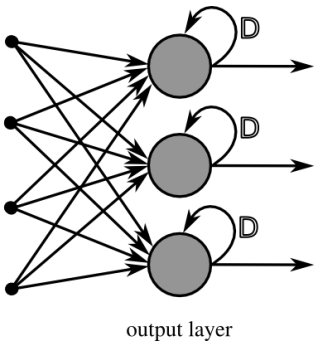

FIGURE 1.7: Recurrent NN.. Image from [6].

These structures enable artificial neural networks to learn complex representations from data. In subsequent sections, we will examine activation functions in more detail, as they play a crucial role in defining the behavior of artificial neurons.

1.9 Activation Functions

What is an activation function? As previously mentioned, the activation function f is the function that allows us to compute the output y of the i-th neuron. This function f receives as input the weighted sum of the inputs x_j, where j is the index of the inputs connected to the i-th neuron, and w_j are the weights associated with each connection between input j and neuron i:

$$a = \sum_{j=1}^{n} w_{ji} x_j$$

We can make a broad distinction between types of activation functions: we separate the identity function from non-linear activation functions.

1.9.1 Identity Function

The identity activation function is the simplest case, where the output y of the neuron coincides with the weighted sum of its inputs:

$$f(a) = a$$

This function does not introduce any non-linearity into the network.

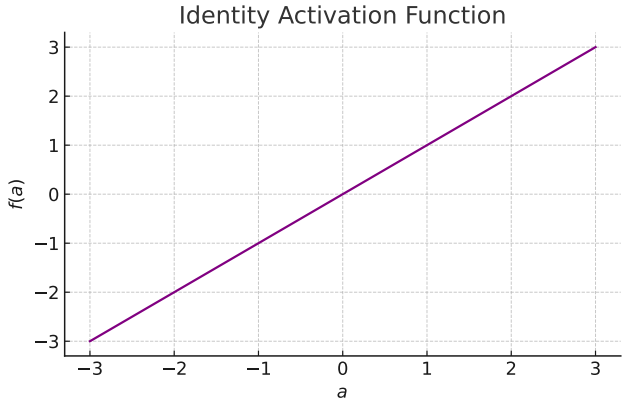

FIGURE 1.8: Identity activation function.

1.9.2 Binary Threshold Function

The binary threshold function returns an output of 1 when the weighted sum a exceeds a given threshold θ, and 0 otherwise:

$$y_i = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{if } a < \theta \end{cases}$$

Graphically, the function has a flat value of 0 for all $a < \theta$ and jumps to 1 for all $a \geq \theta$. When $\theta = 0$, this jump occurs exactly at the origin, but in general θ can take any real value, shifting the step point accordingly.

To better integrate the threshold into the function, we can rewrite it as:

$$y_i = f\left(\sum_{j=1}^{n} w_{ji}x_j - \theta\right)$$

This expression introduces the notion of the **bias**. The bias term θ allows us to shift the activation threshold and can be modeled as an additional input $x_0 = 1$ with an associated weight $w_{0i} = -\theta$. Thus, the neuron output becomes:

$$y_i = f\left(\sum_{j=0}^{n} w_{ji}x_j\right)$$

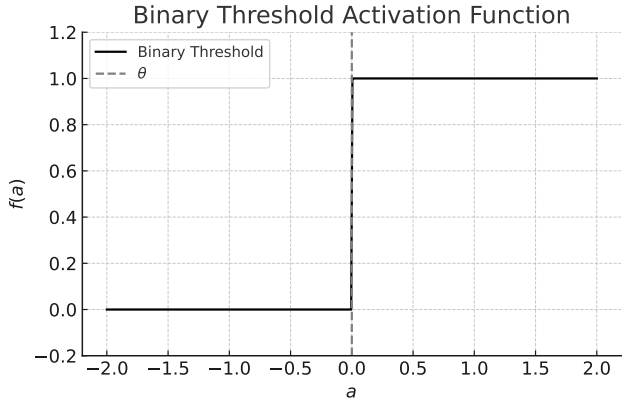

FIGURE 1.9: Binary threshold activation function.

However, this function is not differentiable at $a = \theta$, which can create issues in gradient-based learning. For this reason, differentiable activation functions are often preferred.

1.9.3 Piecewise Linear Function

One such differentiable function is the piecewise linear function, defined as:

$$f(a) = \begin{cases} 0 & \text{if } a < -1 \\ 0.5(a+1) & \text{if } -1 \leq a \leq 1 \\ 1 & \text{if } a > 1 \end{cases}$$

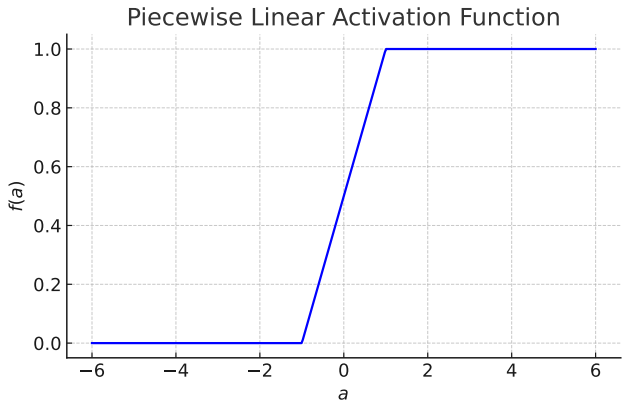

FIGURE 1.10: Piecewise linear activation function.

This function is differentiable (except at the boundary points) and maps input values into the range $[0, 1]$.

1.9.4 Sigmoid (Logistic) Function

Another commonly used differentiable function is the sigmoid (or logistic) function:

$$f(a) = \frac{1}{1 + e^{-a}}$$

This function smoothly transitions from 0 to 1, and is differentiable everywhere, including at 0. It also maps all real-valued inputs to the range $(0, 1)$.

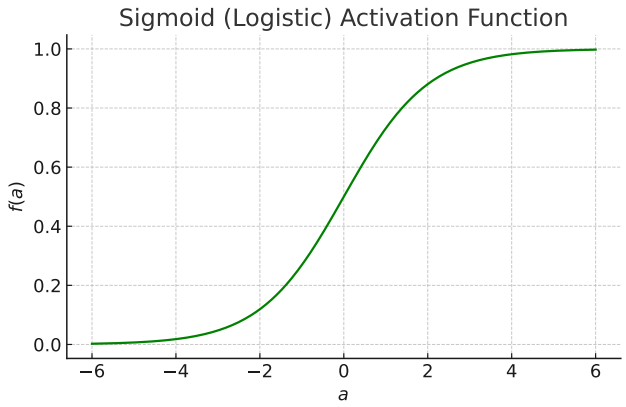

FIGURE 1.11: Sigmoid (logistic) activation function.

1.9.5 ReLU (Rectified Linear Unit)

ReLU is another widely used activation function, defined as:

$$f(a) = \max(0, a)$$

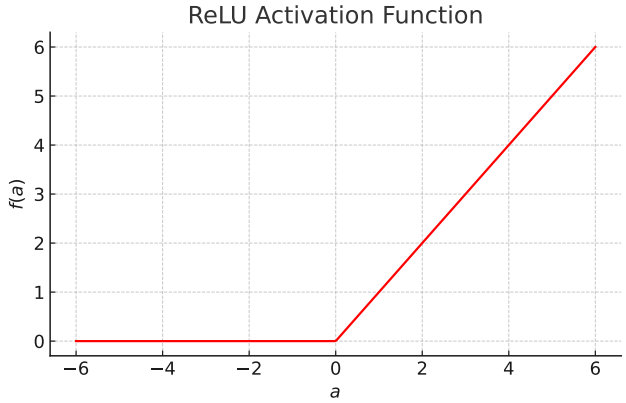

FIGURE 1.12: ReLU activation function.

ReLU outputs zero for negative inputs and is linear for positive inputs. Though not differentiable at zero, it performs well in practice and is computationally efficient.

1.9.6 ELU (Exponential Linear Unit)

The ELU function is defined as:

$$f(a) = \begin{cases} a & \text{if } a \geq 0 \\ \alpha(e^a - 1) & \text{if } a < 0 \end{cases}$$

where α is a positive parameter that controls the value to which an ELU saturates for negative inputs. ELUs help avoid the issue of dying neurons encountered with ReLU and provide a smoother gradient for negative values.

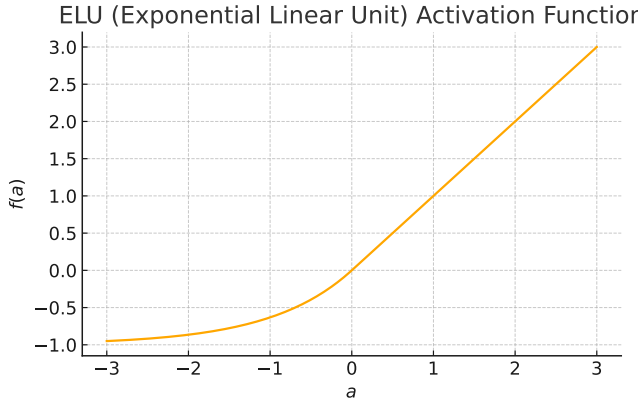

FIGURE 1.13: ELU (Exponential Linear Unit) activation function.

The Role of Bias

The bias term θ allows each neuron to shift its activation threshold. As seen earlier, it can be implemented as an additional input $x_0 = 1$ with a trainable weight $w_{0i} = -\theta$, enabling the neuron to generalize better and adapt to more complex data distributions.

This formulation leads to a more uniform structure for all neurons and simplifies the learning process, especially when using optimization algorithms like backpropagation.

1.10 Supervised Learning

One of the main paradigms for training neural networks is **supervised learning**. In this framework, the network is presented with a set of examples composed of input–output pairs, where the output represents the desired response (also called *target*). The goal of learning is to adjust the synaptic weights so that the network produces outputs that closely match the targets.

Learning, in this context, means modifying the weights of the connections between neurons. A neural network is said to be learning when the values of its weights are being updated.

Consider a network with n input neurons and p output neurons. Let the output vector of the network be:

$$\mathbf{y} = [y_1, y_2, \ldots, y_p]^T$$

and the corresponding target vector for a given input pattern be:

$$\mathbf{t} = [t_1, t_2, \ldots, t_p]^T$$

If we are given a training set composed of M input–output pairs (also called patterns), the error for a single pattern k is defined as:

$$E^{(k)} = \frac{1}{2} \sum_{j=1}^{p} (t_j^{(k)} - y_j^{(k)})^2$$

This is called **MSE**. The total error over the entire training set is then:

$$E = \sum_{k=1}^{M} E^{(k)} = \frac{1}{2} \sum_{k=1}^{M} \sum_{j=1}^{p} (t_j^{(k)} - y_j^{(k)})^2$$

In supervised learning the **loss function** E (or *cost function*), quantifies the discrepancy between the predicted outputs of the network and the corresponding target values. The choice of the loss function is crucial, as it directly influences how the network learns.

From a geometrical perspective, the loss function defines a multidimensional surface over the space of all possible weight configurations. This surface typically presents numerous peaks and valleys, corresponding to local maxima and minima.

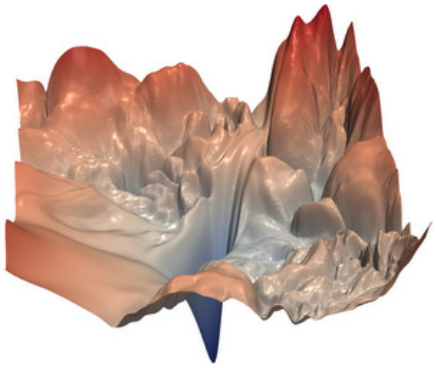

FIGURE 1.14: Loss function surface example. Image from [7]

Training a neural network becomes, therefore, an **optimization problem**, where the objective is to find a configuration of weights that minimizes the loss function.

In practical terms, minimizing the loss means adjusting the synaptic weights so that the outputs of the network approach the target values as closely as possible across all input patterns. Various optimization algorithms, such as gradient descent, are employed to iteratively update the weights in the direction that most reduces the loss, ideally reaching a (local or global) minimum of the error surface.

1.11 Gradient Descent

As said before, the learning process can be formulated as an **optimization problem** where the objective is to minimize the error function E with respect to the weights. A common method to perform this minimization is the **Gradient Descent** algorithm.

Gradient Descent updates the weights in the direction opposite to the gradient of the error:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

where:

- w_{ji} is the weight connecting input i to output neuron j,
- η is the learning rate,
- $\frac{\partial E}{\partial w_{ji}}$ is the partial derivative of the error with respect to the weight.

1.12 The Delta Rule (Widrow-Hoff Rule)

In order to understand how training works, the **Delta Rule** is a particular case of gradient descent applied to a linear neuron. It states that the weight

update should be proportional to the product of the input and the error signal:

$$\Delta w_{ji} = \eta \delta_j x_i$$

where:

- $\delta_j = t_j - y_j$ is the error term for neuron j,
- x_i is the i-th input.
- η is called **learning rate**

1.13 Equivalence Between Delta Rule and Gradient Descent

We now show the equivalence between the Delta Rule and Gradient Descent for a linear neuron. Suppose the output of neuron j is:

$$y_j = \sum_{i=1}^{n} w_{ji} x_i$$

Then, the derivative of the error E with respect to weight w_{ji} is:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ji}}$$

From the definition of E, we have:

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j) = -\delta_j$$

and:

$$\frac{\partial y_j}{\partial w_{ji}} = x_i$$

Combining the two:

$$\frac{\partial E}{\partial w_{ji}} = -\delta_j x_i \quad \Rightarrow \quad \Delta w_{ji} = \eta \delta_j x_i$$

which is exactly the Delta Rule. Therefore, the Delta Rule can be seen as a specific application of the Gradient Descent method for linear neurons.

Weight update

In practice, different strategies can be used to update weights:

- **Batch learning:** weights are updated after the entire training set has been processed.

- **Online (stochastic) learning:** weights are updated after each individual training example.

Further refinements to this rule include techniques such as momentum and adaptive learning rates, which will be discussed in the next section.

1.14 Backpropagation

While the Delta Rule and gradient descent provide a foundation for understanding learning in single-layer networks, training deep neural networks requires a more general approach. The **backpropagation** algorithm extends these ideas to multilayer architectures by efficiently computing the gradient of the loss function with respect to all weights in the network.

The core idea behind backpropagation is to apply the *chain rule* of calculus to propagate the error from the output layer back through the hidden layers. This allows the network to assign blame to each weight according to its contribution to the final output error.

Let us denote the loss function as L, typically the mean squared error or cross-entropy, depending on the task. The update rule for a generic weight w_{ij} is:

$$\Delta w_{ij} = -\eta \frac{\partial L}{\partial w_{ij}}$$

where η is the learning rate. Backpropagation makes this computation feasible by recursively calculating gradients, layer by layer, starting from the output.

The algorithm proceeds in two main phases:

- **Forward pass:** The input is propagated through the network to compute the output and the corresponding loss.
- **Backward pass:** The gradients of the loss with respect to the weights are computed by applying the chain rule, and the weights are updated accordingly.

FIGURE 1.15: Forward and Backward passes.

Backpropagation became a cornerstone of modern deep learning with the rise of computational resources and large datasets, enabling the training of deep architectures such as convolutional neural networks and recurrent neural networks [20, 12].

An example: linear unit backpropagation

Consider a simple linear unit, where the output is computed as:

$$y = wx + b$$

Given a loss function $L(y, t)$, where t is the target, we compute the gradients with respect to the parameters w, b, and the input x. Using the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} = \frac{\partial L}{\partial y} \cdot x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b} = \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x} = \frac{\partial L}{\partial y} \cdot w$$

These derivatives show how the error signal $\frac{\partial L}{\partial y}$, computed at the output, is propagated backward to update both weights and bias, and to pass the gradient with respect to x for use in earlier layers.

Weight and bias update rule

Once the gradients have been computed, the weights and biases are updated using gradient descent. With learning rate η:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} = w - \eta \cdot \frac{\partial L}{\partial y} \cdot x$$

$$b \leftarrow b - \eta \frac{\partial L}{\partial b} = b - \eta \cdot \frac{\partial L}{\partial y}$$

This update reduces the loss by adjusting parameters in the direction of steepest descent.

Adding a non-linear activation: sigmoid case

In most networks, the output passes through a non-linear activation. Consider the sigmoid activation:

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{with } z = wx + b$$

The derivative of the sigmoid function is a well-known expression. Given:

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

its derivative with respect to z is:

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \sigma(z)(1-\sigma(z)).$$

This compact form is particularly useful during backpropagation, as it allows us to express the derivative in terms of the output $y = \sigma(z)$:

$$\sigma'(z) = y(1-y).$$

So, computing:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y} \cdot \frac{dy}{dz} = \frac{\partial L}{\partial y} \cdot \sigma(z)(1-\sigma(z)) = \frac{\partial L}{\partial y} \cdot y(1-y)$$

Then, using the chain rule again:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot x, \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$$

These expressions demonstrate how the activation function modulates the gradient during backpropagation.

Vanishing gradient problem

The **vanishing gradient** problem occurs when gradients become extremely small as they are propagated backward through deep networks. This is common with activation functions like sigmoid or tanh, whose derivatives are bounded between 0 and 1.

As the gradient is passed backward through multiple layers, each with small derivative terms, the product of these terms can become very small. For example, in a network with multiple layers using sigmoid activations, the gradient with respect to the input of an early layer takes the form:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y^{(n)}} \cdot \prod_{l=1}^{n} \sigma'(z^{(l)}) \cdot w^{(l)}$$

Since each term $\sigma'(z^{(l)}) \in (0, 0.25]$, the product tends to zero as the number of layers n increases, leading to vanishing gradients.

This results in negligible updates in early layers, slowing or even preventing learning. To mitigate this, modern architectures often use alternative activation functions such as ReLU, or structural techniques like residual connections to preserve gradient flow.

Automatic differentiation

Modern deep learning frameworks such as PyTorch and TensorFlow implement **automatic differentiation**, which computes gradients efficiently and accurately by recording the sequence of operations during the forward pass and applying the chain rule in reverse. This allows the training of complex networks without the need for manual gradient derivation.

1.15 Learning Rate

The **learning rate** η is a fundamental hyperparameter in training neural networks using gradient-based optimization. It determines the size of the steps taken in the direction of the negative gradient of the error function.

The generic weight update rule using gradient descent is:

$$\Delta w_{ji} = -\eta \frac{\partial L}{\partial w_{ji}}$$

where:

- w_{ji} is the weight from input i to neuron j,
- L is the loss function,
- $\frac{\partial L}{\partial w_{ji}}$ is the partial derivative of the error with respect to w_{ji},
- η is the learning rate.

Impact of Learning Rate

Choosing an appropriate learning rate is critical for effective and efficient training:

- **Too large** η: The updates may overshoot the minimum of the error function, potentially leading to divergence or oscillatory behavior. This is particularly problematic in regions of the error surface that are steep or narrow.
- **Too small** η: The updates become very small, slowing down the convergence significantly. Although the training process may remain stable, it becomes inefficient and may take a large number of iterations to reach an acceptable minimum.

The following schematic illustrates these behaviors:

- Large η: unstable convergence or divergence.
- Small η: slow but steady convergence.
- Moderate η: optimal balance between speed and stability.

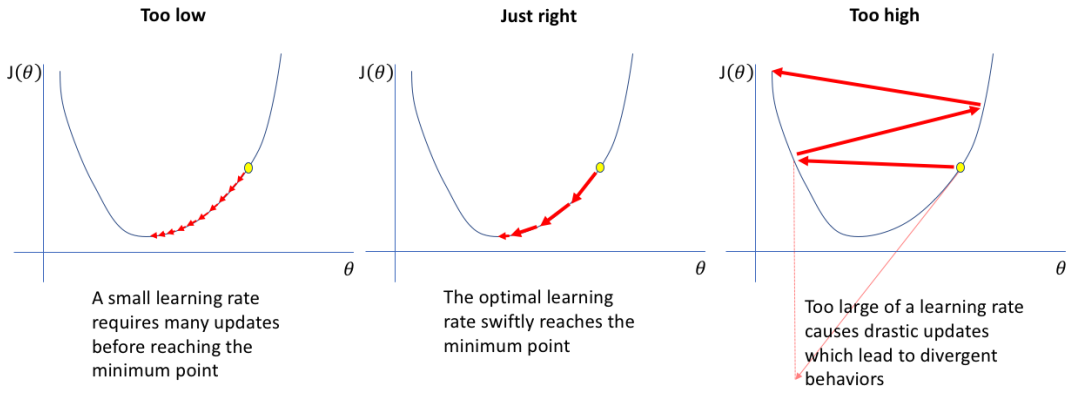

FIGURE 1.16: Learning rate effects during training. Image from [8]

Adaptive Learning Rate Strategies

In practice, instead of using a fixed learning rate, various **adaptive strategies** are often employed to improve performance:

- **Learning rate scheduling:** Start with a higher learning rate and gradually decrease it as training progresses. This allows for rapid convergence in early phases and fine-tuning near the minimum. Examples include:
  - Step decay
  - Exponential decay
  - Inverse time decay
- **Learning rate warm-up:** Start with a very small learning rate and gradually increase it during the initial iterations. This can help stabilize training in networks with high variance in gradients.
- **Per-parameter adaptive methods:** Algorithms like AdaGrad, RMSProp, and Adam automatically adjust the learning rate for each weight individually based on past gradients. These methods often yield faster convergence, especially in large-scale and sparse problems.

Learning rate Selection

In simpler models or academic examples, the learning rate is often chosen empirically. A typical range for η might be:

$$\eta \in [10^{-4}, 10^{-1}]$$

However, the optimal value depends heavily on the architecture, dataset, loss surface, and presence of other optimization techniques.

Summary

The learning rate is one of the most influential **hyperparameters** in training neural networks. It governs the trade-off between convergence speed and training stability. An effective learning rate strategy, whether fixed, decaying, or adaptive, is crucial for successful model training.

In the next section, we will address other challenges in training, such as how to organize and present the data to the network, including techniques like batch and online learning.

1.16 Momentum

One of the main challenges when using gradient descent is its inefficiency in navigating complex error surfaces, especially in regions with narrow valleys or where the gradient frequently changes direction. A widely adopted solution to mitigate these issues is the introduction of a **momentum** term in the weight update rule. In fact, standard gradient descent may exhibit slow convergence, particularly when the error surface has steep but narrow ravines. In such cases, the gradient may point in a zig-zag pattern, resulting in inefficient updates. Momentum helps by accumulating a velocity vector that continues to move in the consistent direction of past gradients.

Mathematical Formulation

Let $\Delta w_{ji}^{(n)}$ denote the weight update at iteration n. The update rule with momentum is:

$$\Delta w_{ji}^{(n+1)} = \eta \delta_j x_i + \alpha \Delta w_{ji}^{(n)}$$

$$w_{ji}^{(n+1)} = w_{ji}^{(n)} + \Delta w_{ji}^{(n+1)}$$

where:

- η is the learning rate,
- δ_j is the error signal for neuron j,
- x_i is the input to the neuron,
- $\alpha \in [0, 1)$ is the **momentum coefficient**,
- $\Delta w_{ji}^{(n)}$ is the previous weight update.

Effect of Momentum

The momentum term α effectively smooths the trajectory of weight updates. If the gradient continues to point in the same direction, the accumulated update becomes larger, allowing the network to move faster in that direction.

Conversely, if the gradient direction changes, momentum helps to dampen oscillations.

The benefits of momentum include:

- Faster convergence in shallow but consistently sloped regions.
- Improved ability to escape from small local minima.
- Reduction of oscillations in directions with high curvature.

Practical Considerations

Typical values of α range from 0.5 to 0.9. When momentum is used, it is common to slightly reduce the learning rate η to avoid overly large updates. The combination of a moderate learning rate and well-tuned momentum often results in significantly improved training performance.

Momentum is also a key component of more sophisticated optimization algorithms, such as Nesterov Accelerated Gradient (NAG) and Adam, which adapt the momentum concept in more advanced ways.

1.16.1 Optimization Algorithms in Practice

In practical applications, gradient descent is rarely used in its basic form. Instead, training is typically performed using more advanced **optimization algorithms**, which extend or improve the basic gradient descent rule by incorporating additional information such as gradient history, per-parameter adaptation, or momentum.

Some of the most widely used optimizers include:

- **Stochastic Gradient Descent (SGD):** A basic version of gradient descent where the weights are updated after each training sample. It is often combined with momentum to improve convergence.
- **SGD with Momentum:** Enhances standard SGD by adding a fraction of the previous weight update to the current one, helping to accelerate convergence in consistent directions and dampen oscillations.
- **RMSProp:** Adjusts the learning rate for each weight individually based on a moving average of squared gradients, making it suitable for non-stationary objectives and deep networks.
- **Adam (Adaptive Moment Estimation):** Combines the benefits of RMSProp and momentum. It maintains both a moving average of the gradients and the squared gradients, and adapts the learning rate accordingly. Adam is widely used due to its efficiency and robustness in many different tasks.

These optimizers automatically handle many aspects of the training dynamics, often making the learning process more stable and faster compared to plain gradient descent.

1.17 Regularization Techniques

Deep neural networks are powerful function approximators, but they are also prone to **overfitting**, a situation in which the model performs well on the training data but poorly on unseen test data. This typically occurs when the network has too many parameters relative to the amount of available data, allowing it to memorize the training set instead of learning general patterns.

Regularization refers to a set of techniques that constrain or penalize model complexity during training. The goal is to improve the generalization ability of the network by discouraging overly flexible or sensitive solutions.

A Motivating Example: Suppose we train a neural network to fit a small dataset of noisy measurements. Without regularization, the network may learn to interpolate all training points perfectly, including noise, leading to high variance and poor performance on new data. Regularization techniques, such as L2 penalty or Dropout, force the network to learn smoother or more robust representations, yielding better test accuracy.

1.17.1 L2 Regularization (Weight Decay)

L2 regularization adds a penalty proportional to the square of the weights to the loss function:

$$E_{\text{total}} = E_{\text{data}} + \lambda \sum_{i,j} w_{ji}^2$$

This encourages the network to keep the weights small, which generally leads to simpler models and improved generalization [21].

1.17.2 Dropout

Dropout is a stochastic regularization technique introduced by Srivastava et al. [22]. During training, each neuron (except for output neurons) is temporarily "dropped" with a probability p, meaning its output is set to zero:

- This forces the network to learn redundant pathways and discourages reliance on specific neurons.
- At inference time, the full network is used, and the activations are scaled to account for dropout during training.

Mathematically, for each hidden unit h_i during training:

$$h_i' = h_i \cdot z_i \quad \text{where } z_i \sim \text{Bernoulli}(1-p)$$

Here:

- h_i is the original output of the i-th hidden unit.

- z_i is a random binary variable drawn from a Bernoulli distribution with probability $1 - p$ of being 1 (i.e., the unit is kept) and p of being 0 (i.e., the unit is dropped).
- h'_i is the output after dropout is applied.

This means:

$$h'_i = \begin{cases} h_i & \text{with probability } 1 - p \\ 0 & \text{with probability } p \end{cases}$$

During **training**, dropout is applied by randomly setting some units to zero at each forward pass. During **testing**, dropout is not applied.

Dropout is especially effective in large networks and prevents co-adaptation of neurons. It will be used in later chapters to regularize U-Net architectures.

1.17.3 Batch Normalization as Implicit Regularization

Batch Normalization (BN), originally introduced to accelerate training [23], has also been found to have a regularizing effect. BN normalizes the activations of each mini-batch (a small subset of the training dataset used to compute an estimate of the gradient during one iteration of training) to have zero mean and unit variance:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y^{(k)} = \gamma \hat{x}^{(k)} + \beta$$

where μ_B and σ_B^2 are the batch mean and variance, and γ, β are learnable parameters.

BatchNorm improves gradient flow, stabilizes training, and adds noise due to mini-batch statistics, which acts as an implicit form of regularization. We will make use of BN in convolutional architectures like U-Net to improve generalization and convergence speed.

1.17.4 Early Stopping

Early stopping halts the training process when performance on a validation set starts to degrade. It prevents overfitting by stopping before the network begins to memorize noise in the training data.

1.17.5 Data Augmentation

Data augmentation enriches the training set by applying transformations to the input data (e.g., rotations, flips, scaling). While not a model-based regularization method, it improves generalization by exposing the network to a broader variety of examples. Insights will be discussed in Chapter 6.

Chapter 2

U-Net: A Convolutional Architecture for Scientific Computing

U-Net is a convolutional neural network (CNN) architecture originally developed for biomedical image segmentation, but its design has proven highly effective across a wide range of image-to-image translation tasks. In particular, its use in scientific domains such as Computational Fluid Dynamics (CFD) has recently gained significant traction, due to its ability to efficiently approximate complex spatial mappings. U-Net was introduced in 2015 by Ronneberger et al. [9], extending the concept of Fully Convolutional Networks (FCNs) [24]. Unlike traditional CNNs aimed at classification, U-Net was designed specifically for pixel-wise prediction tasks. Its architecture enabled accurate segmentations even when trained on small datasets, a critical requirement in medical imaging. In order to understand how U-net works it is useful to present how a CNN works.

2.1 Understanding Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are designed to process data with a grid-like structure, making them particularly effective for visual data analysis. This section introduces the fundamental components and operations that define CNNs and how they relate to U-Net's architecture [25].

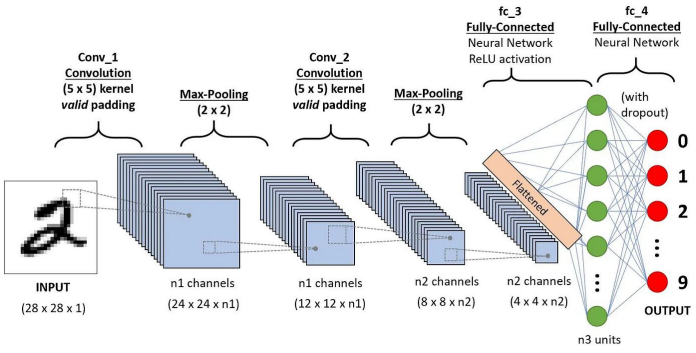

FIGURE 2.1: CNN structure

A CNN works with:

- **Tensors**: Data in CNNs are represented as *tensors*, which are generalizations of matrices to higher dimensions. A typical image input is a 3D tensor of shape $(H \times W \times C)$, where H and W denote spatial dimensions and C is the number of channels (e.g., RGB or physical variables like velocity and pressure in CFD).
- **Neurons and Layers**: A neuron performs a weighted sum of inputs, adds a bias, and applies a non-linear activation function. Neurons are organized in layers, with each layer performing a specific transformation on the input tensor. Layers can be convolutional, pooling, or fully connected, depending on their role.
- **Learnable Parameters**: The learnable parameters in CNNs are the *weights* (or *kernels*) and *biases*. In convolutional layers, each filter (kernel) slides across the input tensor, detecting spatial features. These parameters are optimized during training.

2.1.1 Input Layer

The input layer holds the raw data without performing computations. In classical CNN, each channel may represent RGB layers. In CFD, inputs channels may represent fields like pressure or velocity, with each channel corresponding to a different variable.

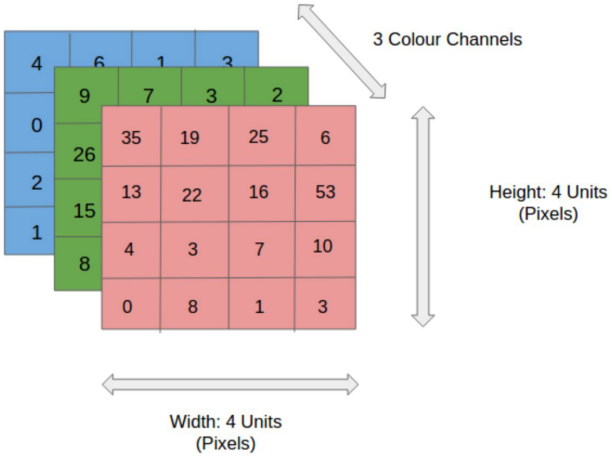

FIGURE 2.2: Typical input representation with multiple channels.

2.1.2 Convolutional Layers and Core Hyperparameters

Convolutional layers are the core building blocks of CNNs. Each layer applies a set of learnable filters (or kernels) that slide over the input, computing dot products between the filter weights and local patches of the input. This operation captures local spatial features such as edges, corners, or patterns.

As we go deeper into the network, these filters learn to detect increasingly complex and abstract features. Each convolution operation reduces or maintains the spatial resolution depending on the stride and padding used.

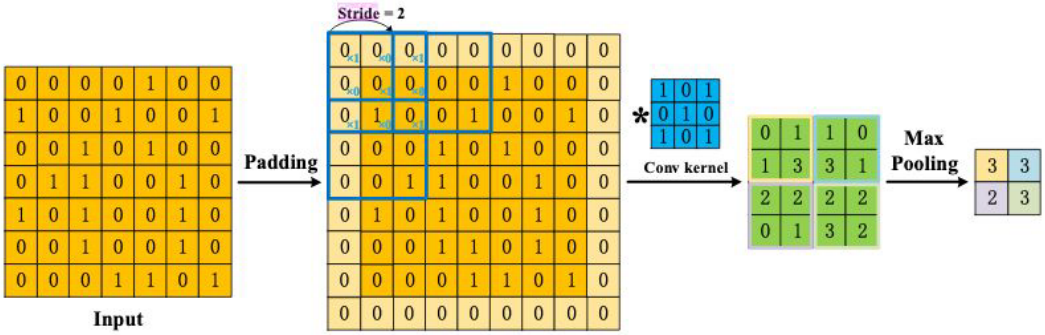

FIGURE 2.3: Typical convolutional operation

Each convolutional neuron performs an elementwise dot product between a filter (kernel) and a local patch of the input tensor, followed by summing the results and applying an optional bias. This operation is repeated spatially across the input, producing a two-dimensional activation map. Multiple filters result in multiple activation maps, which are then stacked to form the output tensor.

Worked Example: Understanding the Convolution Mechanism Let us consider a simple example with a 3×3 input matrix and a 2×2 kernel, using a **stride** of 1 and **no padding**. The kernel is applied (or *slid*) across the input matrix from left to right and top to bottom, extracting localized features. At each position, the overlapping elements of the kernel and the input region are multiplied elementwise, then summed. This result forms a single element in the output activation map.

Input I**:**

$$I = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 1 & 2 \\ 1 & 0 & 1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

The stride of 1 means the kernel moves one cell at a time, resulting in a 2×2 output. We compute the output values as follows:

$$\begin{aligned} O_{1,1} &= (1 \cdot 1) + (2 \cdot 0) + (3 \cdot (-1)) + (1 \cdot 1) = 1 + 0 - 3 + 1 = -1 \\ O_{1,2} &= (2 \cdot 1) + (0 \cdot 0) + (1 \cdot (-1)) + (2 \cdot 1) = 2 + 0 - 1 + 2 = 3 \\ O_{2,1} &= (3 \cdot 1) + (1 \cdot 0) + (1 \cdot (-1)) + (0 \cdot 1) = 3 + 0 - 1 + 0 = 2 \\ O_{2,2} &= (1 \cdot 1) + (2 \cdot 0) + (0 \cdot (-1)) + (1 \cdot 1) = 1 + 0 + 0 + 1 = 2 \end{aligned}$$

Output O**:**

$$O = \begin{bmatrix} -1 & 3 \\ 2 & 2 \end{bmatrix}$$

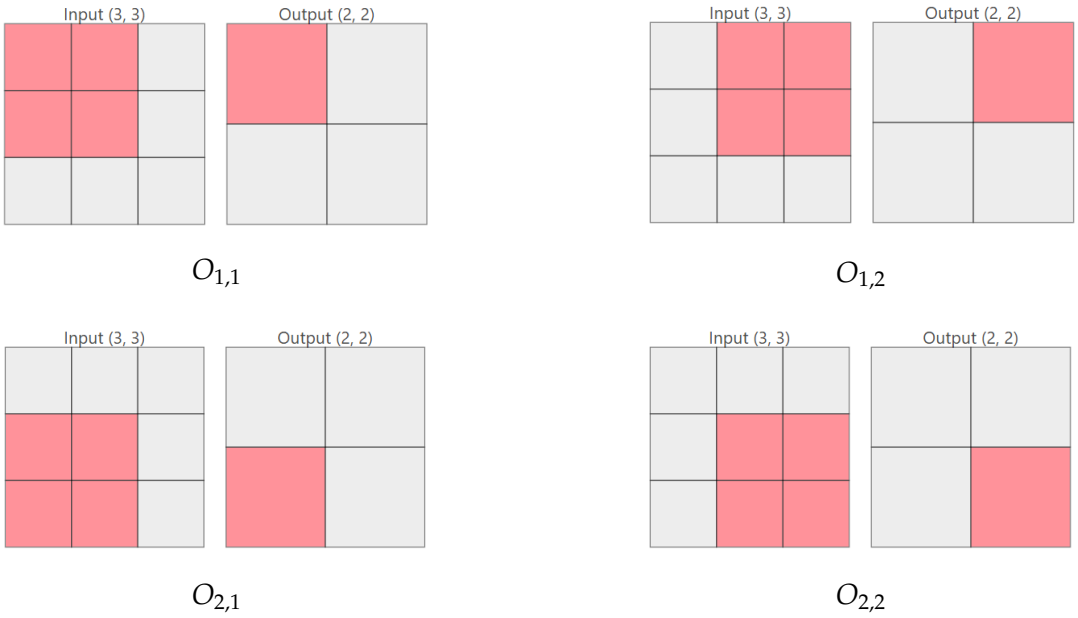

FIGURE 2.4: Examples of different convolution outputs. Each image represents a localized result $O_{i,j}$ from the application of the kernel over the input tensor. Note that the stride is 1

This output is the activation map generated by the convolution with the given kernel. The stride determines how far the kernel shifts in each step: a stride of 1 allows overlapping regions to be scanned, producing high-resolution outputs. Larger strides would skip over more positions, reducing the output size accordingly.

With multiple filters, this process is repeated for each filter, and the resulting activation maps are stacked along the depth dimension.

Multi-Channel Convolution

In convolutional layers with multiple input channels (e.g., RGB images with 3 channels), each filter consists of multiple $k \times k$ kernels—one per input channel. Each kernel is convolved with its corresponding input channel, and the results are summed elementwise to form the final activation map.

For example, in the TinyVGG architecture, a convolutional layer with 10 output channels applied to an input with 3 channels requires $3 \times 10 = 30$ unique kernels. Each output channel aggregates contributions from all input channels, resulting in rich and expressive learned features [25].

Padding

Padding is used to control the spatial size of the output tensor.

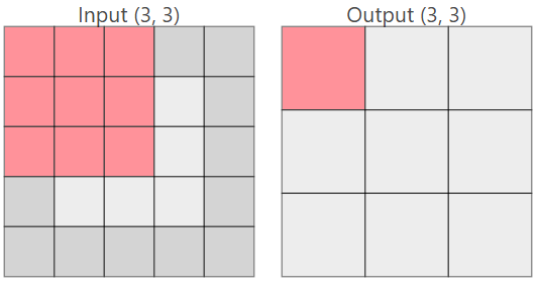

FIGURE 2.5: Example of padding: the original 3x3 matrix (light grey) is expanded in a 5x5

Without padding, the convolution operation reduces the spatial resolution of the input. Zero-padding is the most common approach, which adds rows and columns of zeros around the input tensor. This allows for:

- Preservation of input dimensions ("same" convolution)
- Inclusion of edge information
- Deeper architectures without rapid spatial reduction

Zero-padding is widely used in architectures such as AlexNet and ResNet [25].

Kernel Size

The kernel size, or filter size, determines the receptive field of the convolution. Small kernels such as 3×3 are widely used as they allow capturing fine details while maintaining a manageable number of parameters. Stacking several small-kernel layers can simulate the effect of a larger receptive field while maintaining efficiency.

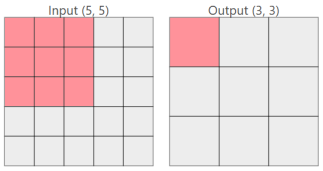

FIGURE 2.6: Kernel (in rose) size 3x3 and relative output

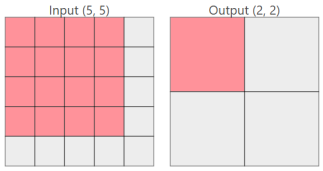

FIGURE 2.7: Kernel size 4x4 and relative output

Larger kernels (e.g., 7×7) cover more spatial area but may lose local information and reduce output size more aggressively. The choice of kernel size depends on the dataset and task but is a critical architectural hyperparameter.

Stride

The stride defines the number of pixels the filter moves at each step. A stride of 1 performs dense scanning of the input and produces large activation maps. Larger strides skip over input positions, reducing the output size and computational cost at the expense of detail.

In TinyVGG, for instance, convolutional layers typically use a stride of 1 to maximize feature resolution [25].

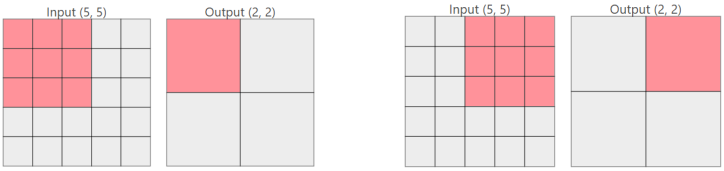

FIGURE 2.8: Example of convolution with stride = 2. The kernel skips two position at each step, reducing the output resolution.

Pooling Layer

Pooling layers follow convolutional layers to reduce the spatial resolution of activation maps while retaining dominant features. The most common form is *max pooling*, which selects the maximum value within a local window (e.g., 2×2). Other pooling operations are *min pooling* or *mean pooling*. Pooling improves computational efficiency and introduces translation invariance, helping the model generalize to shifted versions of patterns.

$$\text{MaxPooling} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = 4 \quad \text{(for window } 2 \times 2\text{)}$$

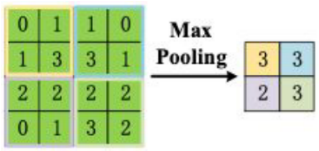

FIGURE 2.9: Kernel size 4x4 and relative output

This operation is typically performed with a stride equal to the window size, ensuring non-overlapping downsampling.

Relation Between Hyperparameters and Output Dimensions The spatial dimension of the output feature map produced by a convolutional layer depends on four key hyperparameters: the **input size** (I), the **kernel size** (K), the **padding** (P), and the **stride** (S). The relationship is given by the formula:

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1$$

where O is the size of the output feature map along one spatial dimension. Padding (P) allows the network to preserve spatial dimensions by adding a border of zeros around the input (there are many padding tecniques). A larger kernel (K) reduces the output size, as it spans a wider region of the input. Stride (S) controls the step with which the kernel moves across the input; increasing the stride reduces the output resolution. Understanding this relationship is essential when designing convolutional networks, particularly for tasks like segmentation or scientific computing where spatial fidelity is important.

Example Let us consider an example with the following values:

- Input size $I = 3$
- Padding $P = 1$
- Kernel size $K = 3$
- Stride $S = 1$

Applying the formula:

$$O = \left\lfloor \frac{3 + 2 \cdot 1 - 3}{1} \right\rfloor + 1 = \left\lfloor \frac{3 + 2 - 3}{1} \right\rfloor + 1 = \lfloor 2 \rfloor + 1 = 3$$

So the output size is 3×3. This example illustrates how using padding can preserve the spatial resolution of the input **when combined with appropriate kernel size and stride settings.**

2.1.3 Output Layer

After passing through several convolutional and pooling layers, the output tensor reaches the final stage of the network: the output layer. At this point, the network must translate the high-level features it has extracted into meaningful predictions, depending on the task at hand.

For classification tasks, such as the one illustrated in Figure 2.1, the feature maps generated by the final convolutional layer are first *flattened* into a one-dimensional vector. This vector is then passed through one or more *fully connected layers*, which act as a traditional feedforward neural network.

The final fully connected layer typically ends with a *softmax* activation function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

where C is the number of output classes and z_i is the logit (raw score) associated with class i. The softmax function converts these logits into a probability distribution over all possible classes, ensuring that:

$$\sum_{i=1}^{C} \text{softmax}(z_i) = 1$$

This probability distribution indicates how confident the network is in classifying the input as belonging to each class. The final predicted label corresponds to the class with the highest probability.

In regression tasks, such as those frequently encountered in CFD, the output layer may instead use a *linear activation* function. In this case, the output consists of one or more continuous values, directly representing physical quantities such as pressure, velocity, or temperature fields.

Thus, the design of the output layer and its activation function is highly dependent on the nature of the task: classification requires probabilistic outputs, while regression demands continuous-valued predictions.

2.2 The Classical U-Net Architecture

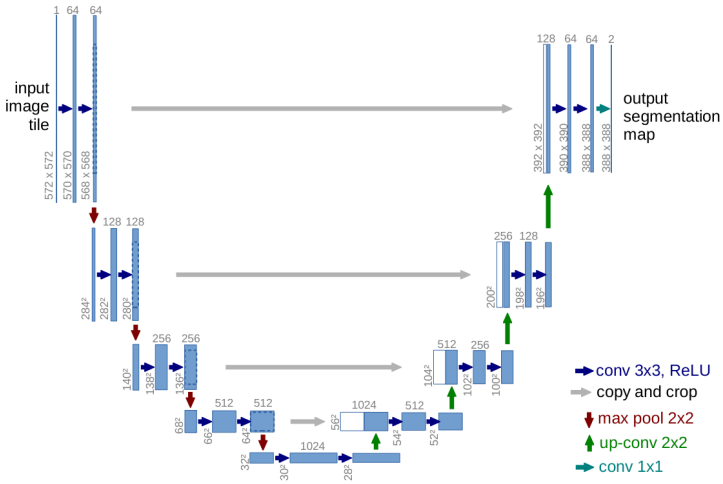

FIGURE 2.10: U-net Network. Image from [9]

The classic U-Net follows a symmetric **encoder-decoder** structure with **skip connections**. It consists of two main parts:

- **Contracting path (encoder)**: This path captures context through successive applications of 3x3 convolutions, each followed by a non-linear activation function and a 2x2 max-pooling operation for downsampling. With each downsampling step, the number of feature channels doubles.
- **Expanding path (decoder)**: This path performs upsampling of the feature maps using interpolation methods (e.g., nearest-neighbor or bilinear) followed by convolutional layers. Each step halves the number of channels and concatenates the result with the corresponding feature maps from the contracting path (skip connections). This fusion helps the network localize features accurately.

The network ends with a 1x1 convolution to map each 64-component feature vector to the desired number of output classes or regression values. This architecture allows the U-Net to combine coarse contextual information with fine spatial details.

2.3 Core Operations in U-Net

The U-Net architecture relies on a sequence of well-established operations in deep learning, we treated many of them in the previous section:

- **Convolution (Conv2D)**: Each convolutional layer applies learnable filters (typically 3x3) to the input feature maps. The result is a set of output channels (**feature maps**) [26].
- **Max Pooling**: A downsampling operation where a window (commonly 2x2) slides over the input, and only the maximum value within the window is retained.
- **Upsampling (Up-convolution)**: In the decoder, spatial resolution is increased using **learned Upsampling**, **nearest neighbor** or **bilinear interpolation**. This is followed by convolutional layers to refine the upsampled features [27]. The first strategy for upsampling, is to insert zeros between the original values, and then apply a learnable filter, such as a convolutional layer, to fill the missing values. For example, given a 3×3 input matrix:

  $$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

  we can create a 5×5 grid by inserting zeros between rows and columns:

  $$\begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 \end{bmatrix}$$

  At this point, a **learnable filter** (such as a 3×3 convolution kernel) is applied to the enlarged grid to generate the final upsampled output, filling those zeros. This allows the network to learn how to best interpolate and combine nearby values during training. This approach provides more flexibility than fixed interpolation methods.

  Alternatively, simpler techniques such as *nearest-neighbor upsampling*, where each value is repeated into a block of fixed size, and *bilinear interpolation*, which computes intermediate values as weighted averages, can also be used especially when learnable parameters are not required or desired.

  For example, using **nearest-neighbor** upsampling with a scaling factor of 2 on the following 2×2 matrix:

  $$X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

produces the upsampled output:

$$\tilde{X} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

In this case, each pixel value is simply copied to a 2×2 block, leading to visible square-shaped regions of constant value. These blocky artifacts, sometimes referred to as "**pixelation**," can degrade visual quality, especially in image generation or segmentation tasks, where smooth transitions are often desired.

- **Concatenation (Skip Connections)**: Encoder feature maps are concatenated with decoder maps at matching resolution. This preserves fine details lost during downsampling and facilitates gradient propagation. They help preserve spatial details by directly passing features from the encoder to the decoder, which is especially useful for preventing the vanishing gradient problem. By providing a shortcut for gradient flow, skip connections ensure efficient training even in deep networks, maintaining crucial information during backpropagation. The model's skip connections combine low-level spatial information from the encoder with high-level contextual information from the decoder, also, they allow for precise segmentation, even when only a limited number of labeled examples are available.

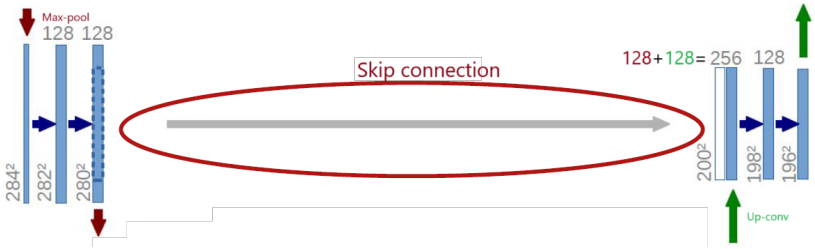

FIGURE 2.11: Skip connection example in a i-th layer of U-net NN

2.4 Activation Functions in U-Net

As said in the previous Chapter, activation functions introduce non-linearity into neural networks, enabling them to learn complex mappings. In U-Net, several activation functions may be used depending on the context and task:

- **ReLU (Rectified Linear Unit)**: ReLU is widely used due to its simplicity and ability to mitigate the vanishing gradient problem. It is the default choice in most U-Net layers [28].
- **ELU (Exponential Linear Unit)**: ELU provides smoother transitions and improves learning in deeper networks [29].

- **Sigmoid**: Maps inputs to the $(0, 1)$ interval, useful in binary segmentation tasks. However, it is prone to saturation and vanishing gradients. It could be a good choice before the output layer if the entire dataset is normalized in the range [0,1] in order to prevent outliers.
- **Linear Activation**: Used in regression tasks, especially in the final layer when the network must predict continuous values (e.g., pressure or velocity fields).

The choice of activation function plays a crucial role in determining the network's convergence behavior and performance.

2.5 The Effectiveness of U-Net Architectures in Computational Fluid Dynamics

In recent years, deep learning architectures have shown great promise in addressing complex problems in Computational Fluid Dynamics (CFD). Among these, the U-Net architecture has emerged as one of the most effective and widely adopted models. Its strength lies in the ability to learn spatially coherent representations from structured data. Originally developed for biomedical image segmentation, U-Net [9] has naturally extended into scientific computing, where many tasks involve mapping between different spatial resolutions or reconstructing complete physical fields from partial inputs.

CFD problems often require predicting high-resolution flow fields (e.g., pressure, velocity, or temperature distributions) starting from low-resolution or sparse data. These problems are inherently spatial and demand models capable of capturing both local features and global context. U-Net is particularly well-suited for this purpose, thanks to the following architectural strengths:

- **Encoder-decoder structure**: Enables the model to extract hierarchical features at multiple scales.
- **Skip connections**: Allow fine-scale spatial details to be retained, which is crucial for accurate flow field reconstruction.
- **Progressive upsampling**: Facilitates effective mapping from coarse to fine resolutions.

This architecture has been successfully applied across a wide range of CFD tasks:

- **Super-resolution reconstruction**: U-Net models enhance coarse velocity and pressure fields, learning the missing fine-scale dynamics.
- **Field prediction from sparse data**: Given limited measurements, such as inlet conditions or scattered sensor data, U-Net can infer the complete flow field [30].

- **Turbulence modeling**: U-Net variants reconstruct or predict turbulent structures, offering a data-driven alternative to traditional closure models [31], [32].

A major advantage of U-Net in this context is its ability to act as an efficient surrogate model. Traditional CFD solvers are computationally expensive, particularly for high-resolution or time-dependent simulations. In contrast, a trained U-Net model can generate accurate flow predictions in a fraction of the time, making it suitable for tasks such as:

- Real-time flow field estimation,
- Optimization and control loops,
- Rapid design space exploration.

In conclusion, the architectural features of U-Net including its deep encoder-decoder pathway, the use of skip connections, and strong spatial generalization align remarkably well with the demands of CFD tasks. Its demonstrated success in super-resolution, field completion, and turbulence modeling underscores its versatility and effectiveness as a learning-based surrogate model. As data-driven methods continue to evolve, U-Net stands out as a cornerstone architecture bridging the gap between classical physics-based solvers and modern machine learning approaches.

Chapter 3

Development of a Python-Based U-Net Framework for Predicting CFD Simulation Fields

The developed code is designed to enhance the resolution of computational fluid dynamics (CFD) solutions by leveraging a deep learning-based super-resolution approach. As illustrated in Figure 3.1, the workflow begins with a coarse 2D mesh used to generate low-fidelity flow field solutions via a traditional CFD solver. These low-resolution outputs serve as input for a U-net model trained to infer high-resolution approximations of the flow field.

The U-net model performs a super-resolution task, mapping the low-fidelity solution to a high-fidelity prediction that corresponds to the results obtained using a fine 2D mesh. The ultimate goal of this code is to approximate high-fidelity CFD solutions with reduced computational cost by bypassing the need for solving the Navier-Stokes equations on a refined grid. This approach enables fast and efficient inference while preserving the essential physical features of the simulated field.The final goal is to achieve the workflow illustrated below:

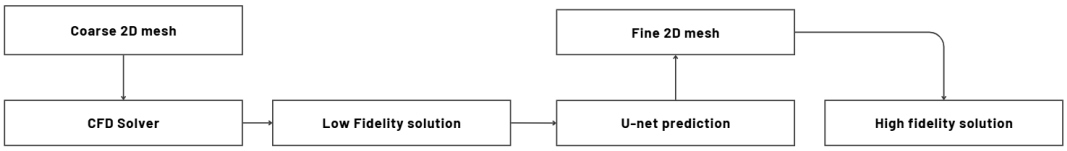

FIGURE 3.1: Goal of the code: from coarse mesh CFD solution to U-net-based high-resolution prediction.

The concept of multifidelity can refer either to an enhancement in the resolution of the predicted field or to an improvement in the underlying physical model. In the latter case, the high-fidelity model may represent a more complex physical representation, such as a 2D section extracted from a 3D RANS simulation or even from a Large Eddy Simulation (LES), thus involving higher physical accuracy and computational cost.

In the final application of the workflow, the user is only required to generate the computational meshes and compute the low-fidelity solution using a standard CFD solver. The rest of the process is handled automatically by the code, which has been previously trained on a dedicated dataset. If the use

of a finer mesh is foreseen, it is assumed to be included during the training phase, allowing the model to learn the mapping from low- to high-fidelity representations accordingly.

3.1 General code workflow

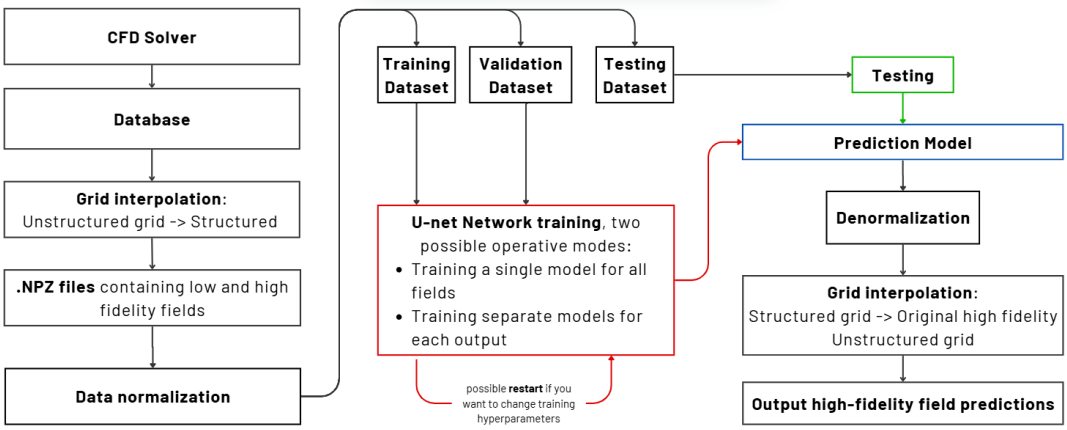

FIGURE 3.2: General code workflow

Figure 3.2 illustrates the general structure of the developed code, which automates the data preparation, training, and inference processes for the super-resolution of CFD fields.

The process begins with a traditional CFD solver, which generates flow field solutions. **These solutions are stored in a database and then interpolated from the original unstructured computational mesh onto a structured grid.** Typically, the interpolation process is carried out over a limited portion of the domain; therefore, the code allows defining a spatial region in which to perform the extrapolation of nodal values and subsequently interpolate them into an N x M matrix. If the selected box exceeds the domain boundaries, the interpolation returns a **mask matrix** as output, with values of 0 corresponding to nodes outside the domain and 1 to those within the domain. This step ensures compatibility with convolutional neural networks, which require structured input data. The interpolated fields are saved in `.npz` files, containing both low-fidelity and high-fidelity versions of the fields.

Next, the data is normalized in a proper way, depending on the specific task, to ensure efficient neural network training. The dataset is then split into **training, validation, and testing** subsets. During the training phase, a U-net architecture is employed. Two operational modes are supported:

- Training a single model that learns to predict all target fields simultaneously.
- Training separate models for each physical output (e.g., pressure, velocity, reaction heat).

Hyperparameter tuning may require restarting the training phase, depending on the user's performance evaluation.

Once trained, the model is used for prediction on the test dataset. The output is then denormalized and interpolated back from the structured grid to the original high-fidelity unstructured grid. This results in a high-resolution prediction that matches the format and fidelity of a traditional CFD output, but with significantly reduced computational cost.

3.2 Customizable U-Net Architecture

The code implements a flexible version of the U-Net architecture, adaptable to a wide range of tasks such as image segmentation, text recognition, or the prediction of physical fields in Computational Fluid Dynamics (CFD) problems. Below is a detailed explanation of its core components.

3.2.1 Model Options

The initial section of the script defines several configurable parameters that govern the model's behavior:

- `flat_class`: if set to `True`, the network ends with a fully connected classifier, suitable for tasks such as text recognition or image classification. This option was included because, in the first version of the code, the model was benchmarked on an image classification task. It was nonetheless kept to make the code versatile for various case studies.
- `constant_channels`: when `True`, the number of filters remains constant across all convolutional layers; otherwise, it increases with depth.
- `filter_max_dim`: specifies the number of filters at the bottom layer of the U-Net.
- `activation_conv`: defines the activation function used in convolutional layers (e.g., ELU).
- `activation_output`: specifies the activation function for the output layer (e.g., Linear, Sigmoid).
- `dropout_rates`: a list that assigns dropout rates to each depth level, useful for regularization.
- `Batch_normalization`: enables or disables the use of batch normalization layers.

3.2.2 Building Blocks

The core component of the model is the `block()` function, which serves both encoding and decoding roles. Each block consists of:

- One or more convolutional layers with optional batch normalization and dropout.
- A max pooling operation (for encoder blocks) or an upsampling operation (for decoder blocks).
- Skip connections, which can be optionally enabled using the `skip` flag.

Decoder blocks additionally concatenate their upsampled output with the corresponding encoder features via skip connections to preserve spatial information, a fundamental principle in U-Net design.

3.2.3 Model Architecture

The function `build_unet()` constructs the entire network given the input shape and number of output classes.

- **Encoder path:** A series of four downsampling blocks that progressively reduce spatial dimensions while increasing feature depth.
- **Bridge:** A central block that connects the encoder and decoder. This section has the highest number of filters and no pooling.
- **Decoder path:** Four upsampling blocks that reconstruct spatial resolution and merge features from the encoder via concatenation.
- **Output:**
  - If `flat_class=True`, the decoder output is flattened and passed through dense layers for classification.
  - Otherwise, a final convolution projects the output to the desired number of channels, with an activation function like Sigmoid for segmentation.

3.2.4 Model Visualization

If the script is executed directly, a model instance is created with a predefined input shape and number of classes. A summary of the model architecture is printed, and a visual representation is saved, which is particularly useful for understanding the topology and debugging. A tipycal structure of a U-net generated by this model is the following:

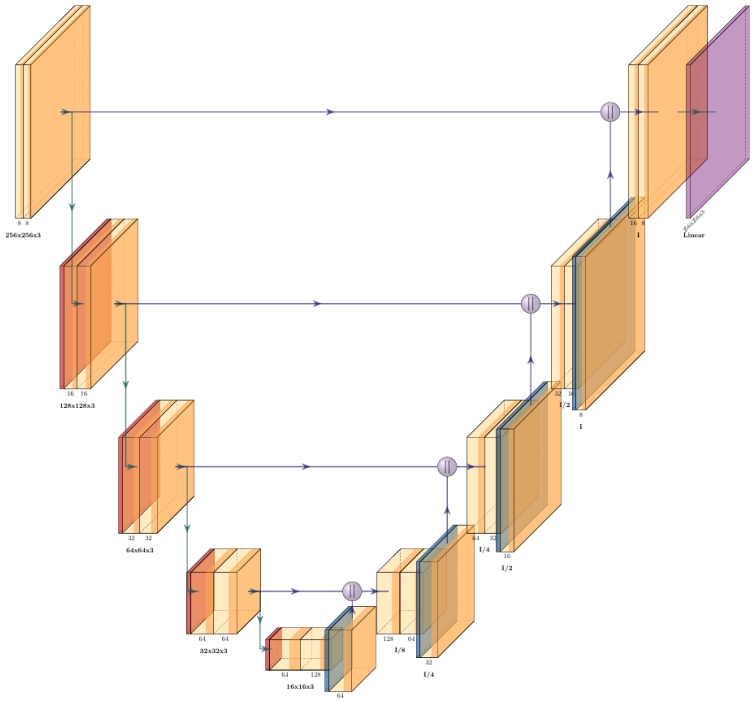

FIGURE 3.3: The U-net strcture generated by the code. In this case, the input images are 256 x 256 with a maximum filter number of 128. The output layer activation function is *linear*

The total number of parameters depends on the complexity of the model. In particular, it can be observed that, once the encoder layers and convolutional blocks are fixed (which must necessarily match those of the decoder), the number of parameters is largely influenced by the number of filters.

TABLE 3.1: Effect of BatchNormalization and Dropout on the model's number of parameters

Filter's number	**Batch Normalization**	**Dropout**	**Trainable Parameters** x10^3	**Non-trainable Parameters** x10^3
32	off	off	21	0
64	off	off	86	0
128	off	off	344	0
256	off	off	1375	0
512	off	off	5496	0
128	on	on	345	1
256	on	on	1377	2.4
512	on	on	5500	5

It can be observed that the increase is exponential as the number of filters is doubled.

Chapter 4

Case Study I: Euler-Based Simulation of a Bump Geometry using a Fortran Solver

The first case study was selected in order to test the network's behavior on the simplest possible simulation scenario. For this reason, an Euler-type solver was chosen to simulate the bump geometry. The code was developed in Fortran during the course "Computational Fluid Dynamics of Propulsion Systems", and both simulations and results had already been validated. This provided a reliable foundation for testing, developing, and improving the Python code responsible for handling interpolations, network training, and back-interpolation.

4.1 Fortran Code Background: Governing Equations

In this section, we introduce the set of equations used to simulate the first test case. The solver is based on the 2D Euler equations, which are solved in non-dimensional form. The non-dimensionalization allows for improved numerical stability and generalization across different flow conditions. It also makes it possible to omit data normalization in the Python code workflow.

Reference Quantities

The following reference quantities are used to non-dimensionalize the equations:

- Reference length: L_{ref}
- Reference velocity: $u_{\text{ref}} = \sqrt{\gamma R T_{\text{ref}}}$
- Reference density: ρ_{ref}
- Reference pressure: $p_{\text{ref}} = \rho_{\text{ref}} u_{\text{ref}}^2$
- Reference temperature: T_{ref}

- Reference time: $t_{\text{ref}} = \frac{L_{\text{ref}}}{u_{\text{ref}}}$

Using these, we define the non-dimensional variables as follows:

$$\bar{x} = \frac{x}{L_{\text{ref}}}, \quad \bar{t} = \frac{t}{t_{\text{ref}}}, \quad \bar{u} = \frac{u}{u_{\text{ref}}}, \quad \bar{\rho} = \frac{\rho}{\rho_{\text{ref}}}, \quad \bar{T} = \frac{T}{T_{\text{ref}}}, \quad \bar{p} = \frac{p}{p_{\text{ref}}} \tag{4.1}$$

Non-dimensional Euler Equations

The conservative form of the 2D Euler equations in non-dimensional variables is written as:

$$\frac{\partial \vec{U}}{\partial \bar{t}} + \frac{\partial \vec{F}}{\partial \bar{x}} + \frac{\partial \vec{G}}{\partial \bar{y}} = 0 \tag{4.2}$$

with the conservative variable vector $\vec{U}$ and flux vectors $\vec{F}$ and $\vec{G}$ defined as:

$$\vec{U} = \begin{pmatrix} \bar{\rho} \\ \bar{\rho}\bar{u} \\ \bar{\rho}\bar{v} \\ \bar{\rho}\bar{E} \end{pmatrix}, \quad \vec{F} = \begin{pmatrix} \bar{\rho}\bar{u} \\ \bar{\rho}\bar{u}^2 + \bar{p} \\ \bar{\rho}\bar{u}\bar{v} \\ \bar{u}(\bar{\rho}\bar{E} + \bar{p}) \end{pmatrix}, \quad \vec{G} = \begin{pmatrix} \bar{\rho}\bar{v} \\ \bar{\rho}\bar{u}\bar{v} \\ \bar{\rho}\bar{v}^2 + \bar{p} \\ \bar{v}(\bar{\rho}\bar{E} + \bar{p}) \end{pmatrix} \tag{4.3}$$

Total Energy and Ideal Gas Law

The non-dimensional total energy per unit volume is expressed as:

$$\bar{\rho}\bar{E} = \frac{\bar{p}}{\gamma - 1} + \frac{1}{2}\bar{\rho}(\bar{u}^2 + \bar{v}^2) \tag{4.4}$$

And the ideal gas law in non-dimensional form becomes:

$$\bar{p} = \bar{\rho}\bar{T} \tag{4.5}$$

These equations form the complete non-dimensional system used in the simulation of the first test case with the Fortran-based Euler solver. The fluxes are calculated with the **ROE** method.

4.2 Computational Domain and Boundary Condition

The computational meshes were generated using the open-source software GMSH. Two levels of fidelity were considered:

- **Low fidelity mesh:** characteristic length $lc = 0.03$, corresponding to approximately 3k nodes.
- **High fidelity mesh:** characteristic length $lc = 0.01$, corresponding to approximately 30k nodes.

The domain is a 2D channel with the following dimensions:

- Length: 1 [–]
- Height: 0.3 [–]
- Bump height b: $[0.01, 0.02, \ldots, 0.10]$

The first important aspect is identifying each simulation case, naming them in a proper way. It has been chosen to identify all simulations using the notation `0(b)0(lc*100)`, combining the bump height and mesh resolution. The dataset includes both the coarse (low fidelity) and fine (high fidelity) results, grouped into single files. **Example:** `0203`, `0201`, `0303`, `0301`, etc.

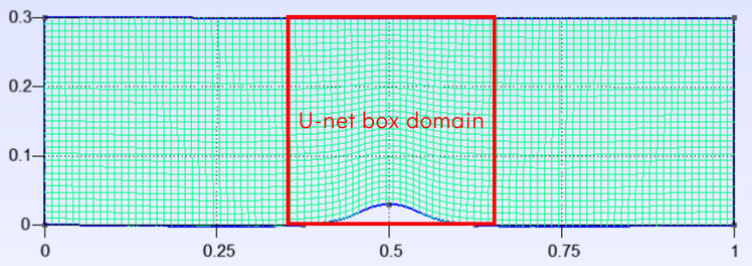

FIGURE 4.1: Low fidelity mesh ($lc = 0.03$) with approximately 3k nodes. Used as input data for the neural network.

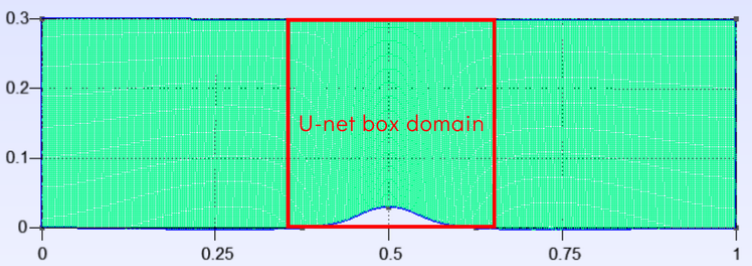

FIGURE 4.2: High fidelity mesh ($lc = 0.01$) with approximately 30k nodes. Used as output data for the neural network.

U-Net box cordinate: Chosen in order to select the portion of domain where the fields change. This permit to interpolate a 256x256 matrix image for each field.

- Along the channel: $(0.35, 0.65)$
- Height: $(0, 0.3)$

Database split:

- 7 training cases (including the highest and lowest bump height)
- 2 validation cases

- 1 test case

Boundary conditions: Fixed for each situation,

- **Walls:** tangency condition
- **Inlet:** $P^* = 1$, $T^* = 1$, $\alpha = 0°$, $M = 0.3$
- **Outlet:** $P^* = 1$, $T^* = 1$, $\alpha = 0°$, $P_{\text{exit}} = 0.9395$

In order to build the dataset the total number of simulation done are 20. Therefore, what varies between the different simulations is the height of the bump, while the boundary conditions remain the same. Naturally, a greater pressure drop will be observed as the bump height increases. The goal of the U-net will be to improve the quality of the solution after performing a first-attempt simulation on the coarse grid, and to correct it in order to interpolate it onto the fine grid, avoiding the need for a full simulation. All of this, during nominal usage, is carried out within a few seconds.

4.3 Code Workflow

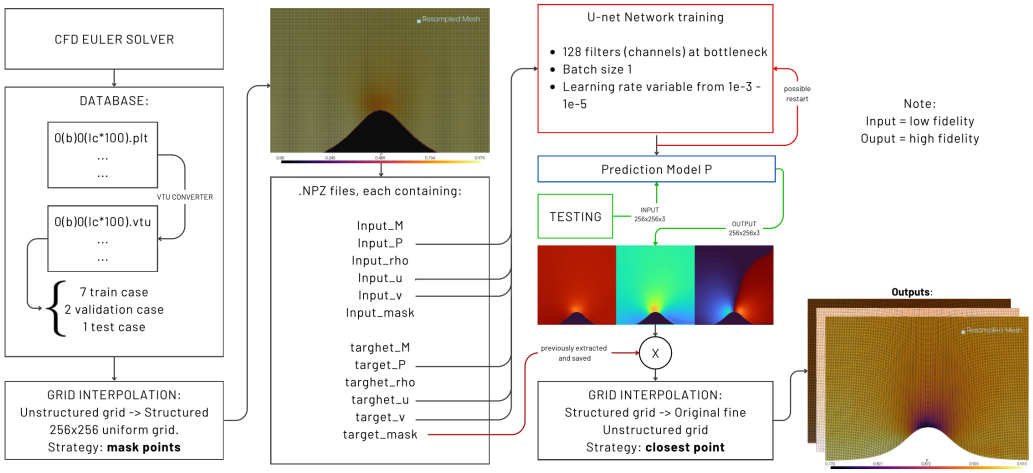

FIGURE 4.3: Workflow for the Bump case. The process begins with data generation using an Euler Fortran solver, followed by grid interpolation, .npz dataset creation, U-Net training, and final re-mapping onto the original mesh. No normalization is needed for this case.

The workflow shown in Figure 4.3 summarizes the complete pipeline developed to enhance the fidelity of Bump case simulations using the supervised learning approach.

The main steps are as follows:

1. **CFD Data Generation**: flow fields are computed using the Euler solver presented, and the simulation results are stored in `.plt` and later converted in `.vtu` format.

2. **Grid Interpolation (Unstructured to Structured)**: the unstructured simulation output is interpolated onto a structured 256×256 **uniform grid** using a *mask points* strategy to handle the computational domain boundaries and later exclude void regions.

3. **Dataset Creation**: each `.npz` file contains both input and target variables such as Mach number (M), pressure (P), density (ρ), and velocity components (u, v), along with a binary mask defining the valid domain. At this stage the mask is stored in order to be used subsequently.

4. **Model Training**: a U-Net convolutional neural network is trained to map low-fidelity inputs to high-fidelity outputs. The architecture uses 128 filters at the bottleneck, a batch size of 1, and a learning rate that decays from 10^{-3} to 10^{-5}. The training can be restarted based on convergence behavior. At this stage, there is no need for Dropout or Batch Normalization. The network receives a 7×256×256×3 tensor, where the first dimension corresponds to the number of bump simulations, the second and third represent the spatial dimensions of the snapshot, and the fourth dimension contains the snapshot channels: pressure, and the u and v velocity components. The rest 2x256x256x3 tensor is used as Validation dataset.

5. **Testing**: the trained model P is tested on a separate test case. The input consists of 3-channel structured grid data ($256 \times 256 \times 3$), and the output is the corresponding high-fidelity prediction with the same shape.

6. **Final Grid Interpolation (Structured to Unstructured)**: the predicted output on the structured grid is mapped back onto the original high-resolution unstructured mesh using a *closest point* interpolation strategy. This ensures consistency with the original CFD geometry and mesh.

4.4 Training

The training process was monitored by visualizing both the loss functions and the feature maps at various levels of the U-Net architecture.

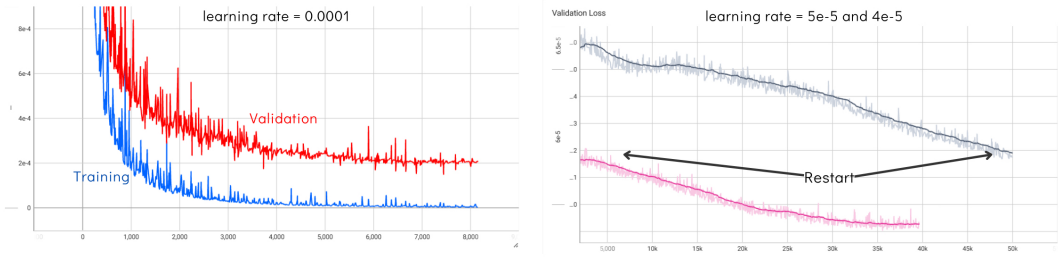

FIGURE 4.4: Training and validation loss trends for different learning rates. Left: learning rate $= 10^{-4}$. Right: use of restarts using learning rates $= 5 \cdot 10^{-5}$ and $4 \cdot 10^{-5}$.

At the end of the training, the following values were achieved:

- `train_loss = 3.72e-6`
- `validation_loss = 5.9e-5`

The training is stopped once high non-zero values inside the bump disappear. t may happen that the values inside the bump are very close to zero, but not exactly zero. This issue can be mitigated by changing the activation function in the output layer, provided that the entire dataset contains values within the range [0,1]. Alternatively, by keeping a linear activation in the output layer, the previously saved mask was applied before performing the backward interpolation. For this type of dataset, this typically occurs when the validation loss drops below $6 \cdot 10^{-5}$.

The possibility of restarting the training with different learning rates allows for controlled experimentation. This approach enables the user to save only the best models, ensuring better control over the final results.

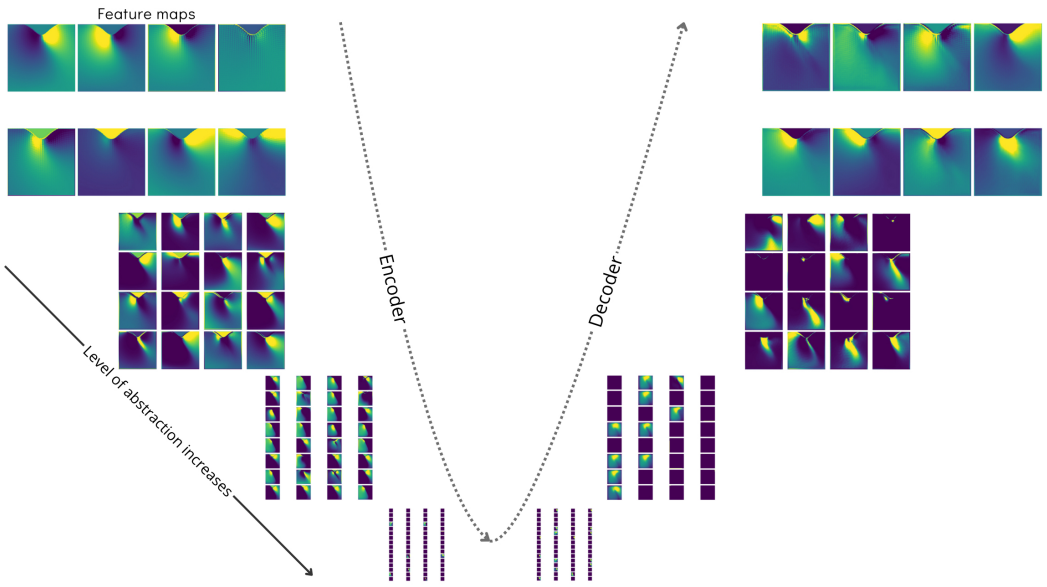

FIGURE 4.5: Visualization of the feature maps across different layers of the U-Net during training. This technique helps to understand the inner workings of the model by highlighting how information is processed at each stage.

Figure 4.5 shows the feature maps extracted from various levels of the U-Net. These maps serve as a tool to partially open the so-called "black box" of deep learning models, providing insight into how the network processes and transforms the input data through its layers. The level of abstraction increases along the encoder path and decreases again through the decoder. As the implementation becomes more complex, having access to these intermediate outputs is essential to debug and detect potential anomalies during training.

4.5 Bump: Results

The performance of the model was evaluated by comparing its predictions with high-resolution CFD results. In the following figures, we assess the model's ability to reconstruct fine-scale flow fields from coarse inputs.

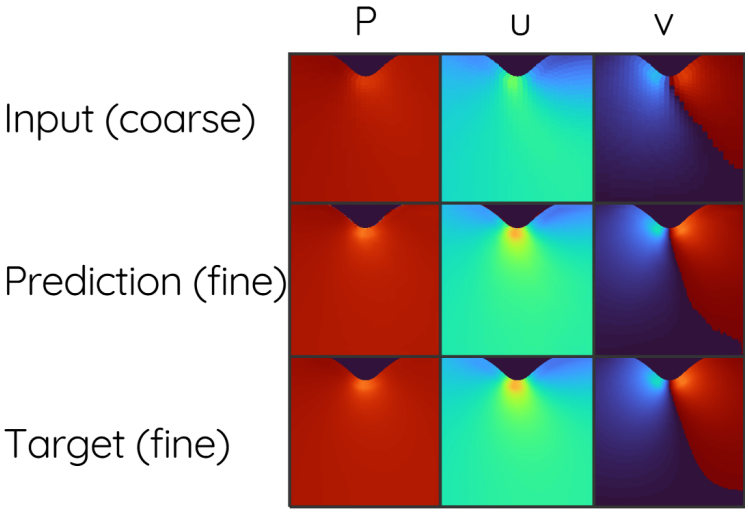

FIGURE 4.6: Standard visual comparison between coarse input, U-Net prediction, and high-resolution CFD target for pressure (P), axial velocity (u), and transverse velocity (v). **Images are flipped due to interpolator transformations**

Figure 4.6 shows the predicted fine-scale fields for pressure (P), axial velocity (u), and transverse velocity (v), given coarse input fields. The U-Net predictions closely resemble the target CFD solutions, capturing both large-scale structures and finer gradients, especially in the velocity fields.

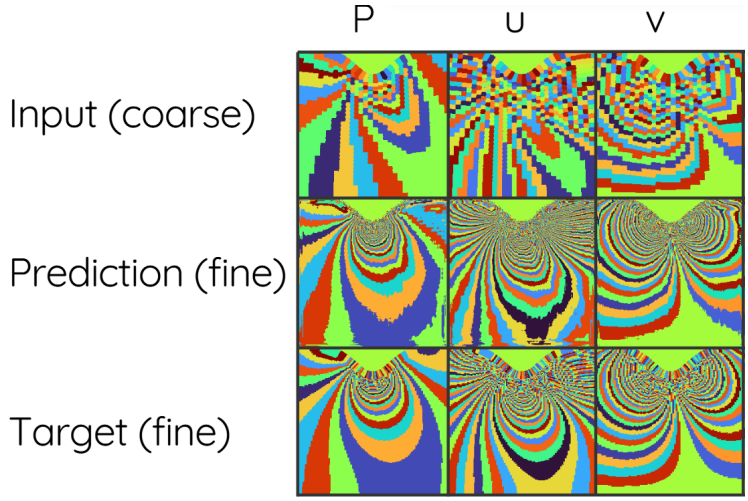

FIGURE 4.7: Same results as in Figure 4.6, with contrast enhancement to highlight fine details. Images are flipped due to interpolator transformations

To highlight the detailed flow features, Figure 4.7 shows the same results with enhanced contrast. This allows a better visualization of complex spatial variations, especially in regions with steep gradients or swirling flow. The domain appears flipped due to PyVista's resampling process; however, this will be automatically corrected during post-processing.

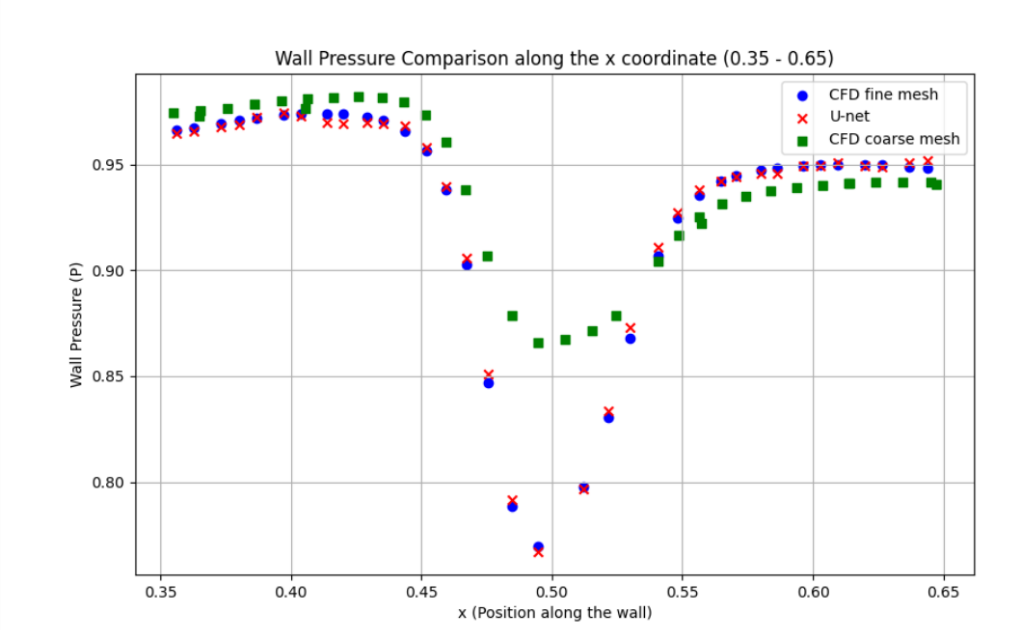

FIGURE 4.8: Wall pressure comparison between coarse CFD input, U-Net prediction, and high-resolution CFD output along $x \in [0.35, 0.65]$.

Figure 4.8 provides a quantitative assessment of model accuracy by comparing the wall pressure distribution along a segment of the domain ($x \in [0.35, 0.65]$). The U-Net prediction (red crosses) matches very closely with the high-resolution CFD data (blue circles), significantly improving over the coarse input (green squares). The model achieves a maximum relative error of 0.61% and a mean relative error of 0.23%, confirming its effectiveness in super-resolving flow fields.

4.6 Issue encountered and final comment

During the training and evaluation of the U-Net model, two main issues were identified that deserve further attention and discussion.

Inactive Kernels

An observed phenomenon during training is the presence of inactive kernels in the convolutional layers. These filters do not activate or produce nearly-zero outputs across the validation set. While not necessarily a malfunction, it

raises concerns about potential inefficiencies in the learned representations. This behavior can be attributed to:

- **Filter Redundancy:** Some filters might become redundant or less useful for image reconstruction. The network might find an efficient representation without utilizing all available filters.
- **Sparse Activation:** Certain filters may specialize in detecting specific features, and thus remain inactive unless such features are present in the input.
- **Dominance of Other Filters:** Strongly activated filters early in training might dominate the reconstruction, reducing the contribution of less active ones.

This issue was visually confirmed by inspecting feature maps from a specific convolutional layer, where several filters showed little to no activation (highlighted in Figure 4.9).

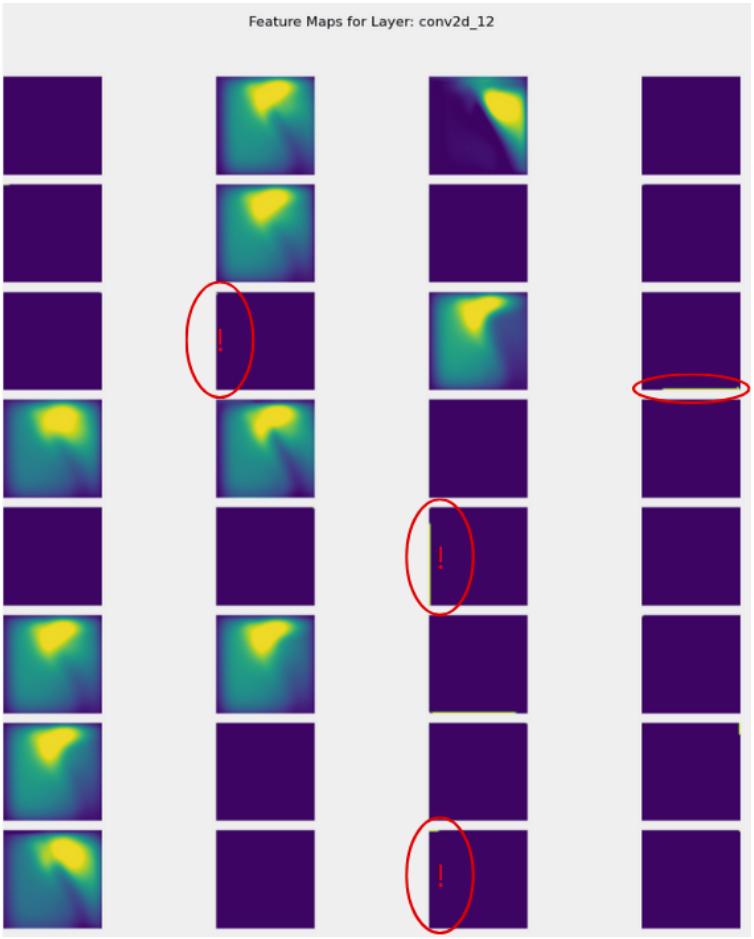

FIGURE 4.9: Feature maps from one of the intermediate convolutional layers. Some filters (circled) exhibit very low or null activation.

Role of the Mask During Post-Processing

Another challenge was found during post-processing. Despite achieving low validation losses, the predicted values inside the geometric bump occasionally deviated from the expected zero value. To prevent propagation of these artifacts during interpolation to the original mesh, a binary mask is applied after prediction. This operation nullifies values inside the bump area to enforce consistency.

This approach serves as a precaution and may be reconsidered if the model learns to correctly reproduce the internal zero values within the bump, making the mask unnecessary in future stages.

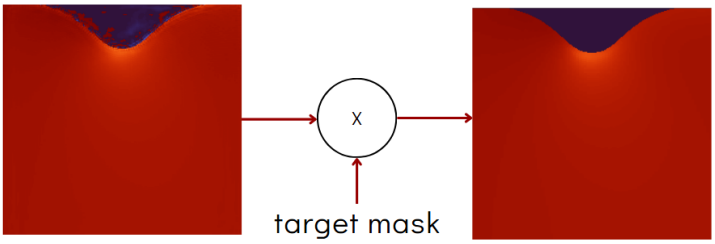

FIGURE 4.10: Illustration of the post-processing mask applied to predicted outputs. The bump region is masked out to ensure correct values during interpolation.

An alternative approach, which was not pursued at this stage of code development, could be to apply an activation function such as the sigmoid, since all values passed during training fall within the range [0,1]. This would allow the suppression of values inside the bump that might be negative and very close to zero. However, this step becomes necessary only if those values are positive and very close to zero.

Chapter 5

Case Study II: AHEAD Hydrogen Burner

The second case study focuses on a lean premixed hydrogen burner, specifically the AHEAD combustor. This configuration was selected due to its relevance in the current research landscape and the availability of a solid dataset of results. The case is particularly suitable for data-driven approaches and machine learning applications thanks to the extensive numerical simulation campaign performed under well-documented conditions. All simulations are carried out in a two-dimensional steady-state axisymmetric framework.

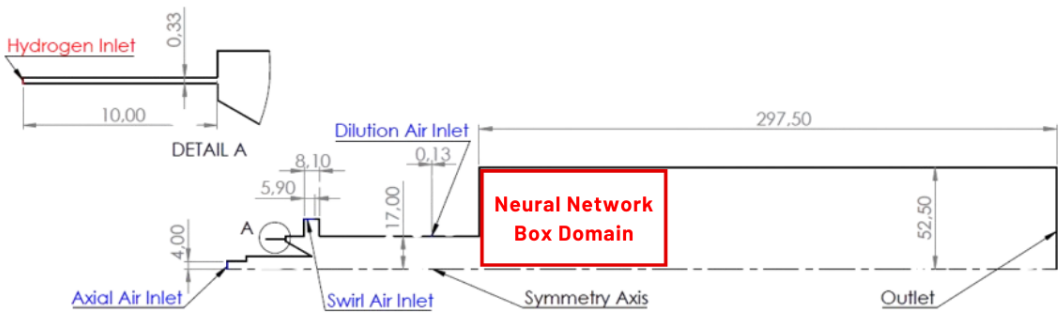

FIGURE 5.1: 2D axisymmetric geometry of the AHEAD combustor, box domain for neural network training, and mesh comparison (fine vs coarse).

The geometry used in this study is a 2D axisymmetric simplification of the AHEAD combustor, preserving the total inlet cross-sectional areas to ensure velocity consistency with the original 3D configuration. The U-Net-based neural network is trained and evaluated on a specific rectangular region of the domain, highlighted in red in Fig. 5.1, with the following physical dimensions:

- Axial direction: $(0.127, 0.265)$ [m]
- Height: $(0, 0.0525)$ [m]

The simulation database comprises 53 different operating conditions. For each configuration, two different mesh resolutions are generated: a fine mesh with approximately 200k nodes and a coarse mesh with approximately 40k nodes:

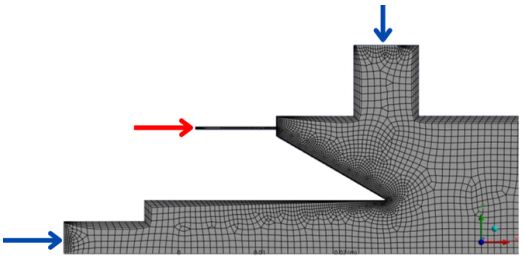

FIGURE 5.2: 40k nodes mesh.

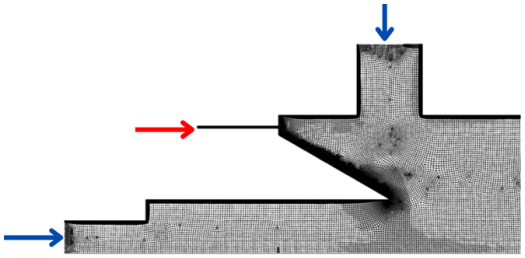

FIGURE 5.3: 200k nodes mesh.

As a result, the total number of CFD simulations required is 106. These cases are organized as follows:

- 44 training cases
- 8 validation cases
- 1 test case

The naming convention used for the dataset is:

`burner_h2_(number_of_cells)_(ID_case).`

For example: `burner_h2_200000_33`.

5.1 High Fidelity Database

The high-fidelity dataset used in this study is based on 2D simulation results obtained from the AHEAD combustor, originally published in *"Multi-Fidelity Modeling of a Lean Premixed Swirl-Stabilized Hydrogen Burner With Axial Air Injection"* [10]. The dataset includes 53 operating points, each characterized by different inlet air mass flow rate, air temperature, and equivalence ratio. The corresponding flame position, normalized with respect to the diameter of the mixing tube, is used as the target quantity.

Each case in the dataset is uniquely identified by an ID and labeled with its operating parameters and resulting normalized flame distance (x_F/D). This high-fidelity dataset serves as the reference for evaluating the accuracy of both the CFD simulations and the multi-fidelity models.

The dataset has been divided as follows:

- **44 training cases**
- **8 validation cases** – selected manually or randomly to ensure coverage of the input domain
- **1 test case** – selected to preserve model generalization

The figure below shows the full table of cases, with red-highlighted IDs corresponding to the manually selected validation set and the green-highlighted ID corresponding to the test case.

ID	$\dot{m}_{Air}$ (kg/h)	T_{Air} (K)	ϕ	x_F / D	ID	$\dot{m}_{Air}$ (kg/h)	T_{Air} (K)	ϕ	x_F / D
1	255	313	0.3	0.277	28	80	453	0.4	0.171
2	255	313	0.4	0.285	29	130	453	0.4	0.293
3	255	313	0.5	0.351	30	150	453	0.4	0.299
4	255	313	0.6	0.450	31	205	453	0.4	0.353
5	255	313	0.7	0.627	32	230	453	0.4	0.417
6	180	453	0.3	0.247	33	93	623	0.4	0.302
7	180	453	0.4	0.262	34	110	623	0.4	0.299
8	180	453	0.5	0.275	35	148	623	0.4	0.350
9	180	453	0.6	0.361	36	82	693	0.4	0.306
10	180	453	0.8	0.615	37	100	693	0.4	0.317
11	180	453	1.0	0.991	38	133	693	0.4	0.324
12	130	623	0.3	0.229	39	130	313	0.6	0.389
13	130	623	0.4	0.233	40	184	313	0.6	0.479
14	130	623	0.5	0.216	41	220	313	0.6	0.529
15	130	623	0.6	0.250	42	295	313	0.6	0.558
16	130	623	0.8	0.474	43	80	453	0.6	0.293
17	130	623	1.0	0.866	44	130	453	0.6	0.399
18	116	693	0.3	0.241	45	150	453	0.6	0.389
19	116	693	0.4	0.228	46	205	453	0.6	0.442
20	116	693	0.5	0.220	47	230	453	0.6	0.555
21	116	693	0.6	0.251	48	93	623	0.6	0.349
22	116	693	0.8	0.463	49	110	623	0.6	0.342
23	116	693	1.0	0.807	50	148	623	0.6	0.362
24	130	313	0.4	0.257	51	82	693	0.6	0.339
25	184	313	0.4	0.313	52	100	693	0.6	0.346
26	220	313	0.4	0.342	53	133	693	0.6	0.346
27	295	313	0.4	0.375					

FIGURE 5.4: Summary of the 53 high-fidelity simulations. In manual split mode, validation cases are highlighted in red, the test case in green. Adapted from [10].

Given the limited size of the dataset, a specific effort was made to evaluate the most effective strategy for splitting the database into training and validation sets. Two approaches were tested: a manual split, aimed at ensuring proper coverage of the parameter space, and a purely random split. The impact of both strategies on model accuracy is analyzed and discussed in the results section. This evaluation helps determine whether a manually guided selection provides advantages over a simpler, automated approach when data availability is limited.

5.2 Summary of Governing Equations

As discussed in [10], the computational model is based on the solution of the compressible Reynolds-Averaged Navier–Stokes (RANS) equations in axisymmetric coordinates, assuming steady-state conditions and neglecting azimuthal derivatives. Favre-averaged quantities are used throughout the formulation, ensuring consistency between mass-weighted averaging and the nonlinear convective terms.

The assumption of axisymmetry implies that all variables are independent of the azimuthal coordinate θ, i.e.,

$$\frac{\partial(\cdot)}{\partial\theta} = 0 \tag{5.1}$$

This allows for a two-dimensional formulation in the (r, z) plane while retaining the contribution of the tangential velocity component v_θ. Such an assumption is valid for geometries and boundary conditions that are rotationally symmetric around the axis, a common configuration in nozzles, burners and ducted flows.

Continuity Equation

$$\nabla \cdot (\rho\tilde{\mathbf{v}}) = 0 \tag{5.2}$$

The continuity equation expresses the conservation of mass. It ensures that the rate of mass entering and leaving any control volume is balanced, reflecting the incompressible or compressible nature of the fluid through the density ρ. In the Favre-averaged form, it guarantees the correct treatment of density fluctuations, which are non-negligible in compressible and reacting flows.

Momentum Equation

$$\nabla \cdot (\rho\tilde{\mathbf{v}}\tilde{\mathbf{v}}) = -\nabla p + \nabla \cdot \boldsymbol{\tau} \tag{5.3}$$

This equation represents the conservation of momentum. The first term on the right-hand side accounts for the pressure gradient, while the second term involves viscous and turbulent stresses through the total stress tensor $\boldsymbol{\tau}$.

The stress tensor is defined as:

$$\tau_{ij} = 2(\mu + \mu_t)\left(S_{ij} - \frac{1}{3}\delta_{ij}\frac{\partial\tilde{v}_k}{\partial x_k}\right) - \frac{2}{3}\rho k\delta_{ij} \tag{5.4}$$

where the effective viscosity $(\mu + \mu_t)$ includes the molecular and turbulent contributions. The last term $-\frac{2}{3}\rho k\delta_{ij}$ represents the isotropic part of the Reynolds stress tensor, proportional to the turbulent kinetic energy k.

The strain-rate tensor is given by:

$$S_{ij} = \frac{1}{2}\left(\frac{\partial\tilde{v}_i}{\partial x_j} + \frac{\partial\tilde{v}_j}{\partial x_i}\right) \tag{5.5}$$

and quantifies the rate of deformation of the fluid elements. The accurate computation of $\boldsymbol{\tau}$ is essential for capturing shear layers, recirculation zones and mixing phenomena typical of confined reacting flows.

Energy Equation

$$\nabla \cdot \left[\rho \tilde{\mathbf{v}} \left(\tilde{h} + \frac{\tilde{v}^2}{2} \right) \right] = \nabla \cdot \left(k_{\text{eff}} \nabla T - \sum_j \tilde{h}_j \mathbf{J}_j + \boldsymbol{\tau} \cdot \tilde{\mathbf{v}} \right) + \Omega \tag{5.6}$$

The energy equation describes the conservation of total enthalpy, including both thermal and kinetic contributions. On the right-hand side, the first term represents conductive and turbulent heat fluxes through the effective thermal conductivity k_{eff}, while the second accounts for enthalpy diffusion due to mass transport of individual species. The term $\boldsymbol{\tau} \cdot \tilde{\mathbf{v}}$ denotes viscous dissipation, and Ω is a generic volumetric heat source, which in reacting flows is associated with chemical heat release. This equation couples thermal, chemical and mechanical effects, making it central to the correct prediction of temperature and heat-release distributions.

Species Transport

For $M - 1$ chemical species, the transport equation reads:

$$\nabla \cdot (\rho \tilde{\mathbf{v}} Y_j) = -\nabla \cdot \mathbf{J}_j + R_j \tag{5.7}$$

where Y_j is the mass fraction of species j. The left-hand side represents convection of each species, while the right-hand side includes molecular and turbulent diffusion ($\mathbf{J}_j$) and the local rate of production or consumption R_j due to chemical reactions.

The species diffusion flux is defined as:

$$\mathbf{J}_j = -\left(D_{j,m} + \frac{\mu_t}{Sc_t} \right) \nabla Y_j - D_{T,j} \nabla T \tag{5.8}$$

Here, $D_{j,m}$ is the molecular diffusivity, while μ_t / Sc_t accounts for turbulent diffusion through an effective turbulent Schmidt number Sc_t. The last term, proportional to the temperature gradient, represents thermal diffusion (Soret effect), which can be relevant in hydrogen or light-mass species transport.

Turbulence and Reaction Modeling

Turbulence is modeled using the k–ω SST (Shear Stress Transport) model, which blends the near-wall accuracy of the k–ω formulation with the robustness of the k–ϵ model in the free stream. This approach provides a good compromise between accuracy and computational cost, particularly suited for separated and recirculating flows in combustors and diffusers.

Combustion is treated using the Eddy Dissipation Concept (EDC), where finite-rate chemistry and turbulence–chemistry interaction are combined:

$$R_j = \frac{\rho \kappa_{fs}}{\tau^*} (Y_j^* - Y_j) \tag{5.9}$$

The source term R_j expresses the relaxation of the local mass fraction Y_j towards its fine-scale equilibrium value Y_j^* over a characteristic time scale τ^*. The fine-scale volume fraction κ_{fs} and time scale are defined as:

$$\kappa_{fs} = \frac{\xi^2}{1-\xi^2}, \quad \xi = C_\xi \left(\frac{\nu\epsilon}{k^2}\right)^{1/4} \tag{5.10}$$

$$\tau^* = C_\tau \left(\frac{\nu}{\epsilon}\right)^{1/2} \tag{5.11}$$

These relations link the turbulence parameters k and ϵ to the chemical time scales, enabling the model to capture the effect of local turbulence intensity on the rate of chemical reactions. In the present formulation, the constants C_ξ and C_τ are calibrated following the standard EDC formulation by Magnussen.

Thermophysical Properties

The specific heat of each species is calculated using NASA7 polynomials:

$$\frac{c_{p,j}(T)}{R} = a_{0,j} + a_{1,j}T + a_{2,j}T^2 + a_{3,j}T^3 + a_{4,j}T^4 \tag{5.12}$$

This formulation allows accurate representation of temperature-dependent specific heats over wide thermal ranges.

The mixture sensible enthalpy is obtained as:

$$\tilde{h} = \sum_j Y_j \tilde{h}_j, \quad \tilde{h}_j = \int_{T_{ref}}^{T} c_{p,j} dT \tag{5.13}$$

ensuring thermodynamic consistency between species enthalpies and mixture energy balance.

The effective conductivity includes both molecular and turbulent contributions:

$$k_{\text{eff}} = k + \frac{c_p \mu_t}{Pr_t} \tag{5.14}$$

where Pr_t is the turbulent Prandtl number, typically assumed constant and calibrated for high-Reynolds reactive flows.

The chemical kinetics follow the Ó Conaire mechanism, which includes 10 species and 19 elementary reactions, optimized for hydrogen–air combustion. This mechanism accurately reproduces the chain-branching behavior and ignition characteristics of hydrogen flames, maintaining manageable computational complexity.

Overall, this formulation provides a consistent framework for simulating reactive, compressible, and turbulent flows in axisymmetric configurations.

The coupling between RANS turbulence modeling, detailed thermochemistry, and multi-species transport enables the solver to capture both aerodynamic structures and chemical heat release phenomena with reasonable computational effort.

5.3 Code Workflow

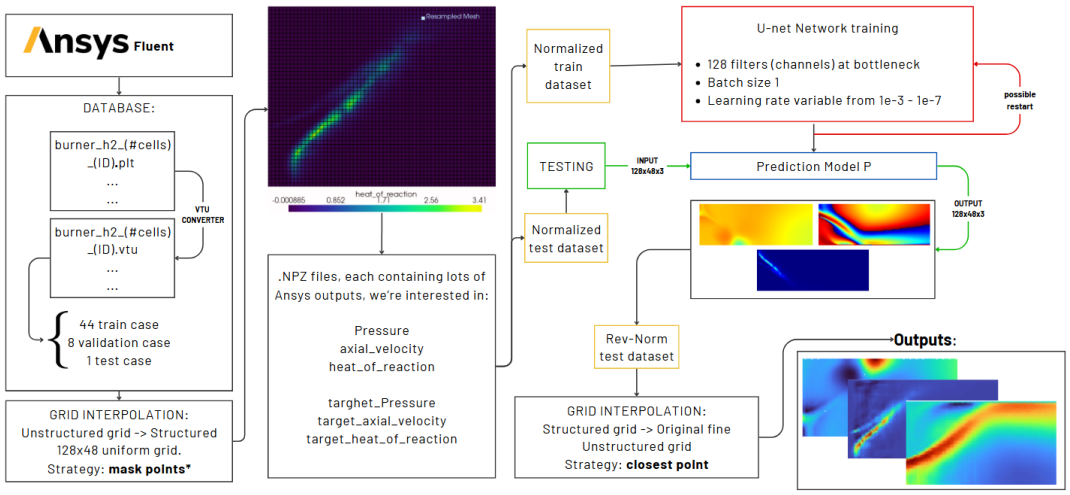

FIGURE 5.5: General workflow for the U-Net training and inference pipeline applied to the hydrogen burner dataset.

The code workflow generally remains consistent with previous implementations, although some additions have been made to handle the specifics of this case study, as illustrated in Fig. 5.5. The primary new step is the normalization of the dataset, which plays a crucial role due to the dimensional nature of the simulation results obtained from Ansys Fluent.

The raw CFD data are generated in both `.plt` and `.vtu` formats for 53 simulation cases, which are then categorized into 44 training, 8 validation, and 1 test case. Each simulation is run at two mesh resolutions (fine and coarse), and results are converted into NumPy-compatible `.npz` files containing several physical fields, such as:

- Pressure
- Axial velocity
- Heat of reaction

These fields are then used to construct input and target arrays for supervised training. The data are interpolated from the original unstructured mesh onto a structured uniform grid of size 128×48, using a masking strategy to identify valid physical regions.

The structured data are normalized to account for differences in units and scales between variables. This step is essential for stable and efficient training of the neural network.

A U-Net architecture is employed, with 128 filters at the bottleneck, a batch size of 1, and a variable learning rate ranging from 10^{-3} to 10^{-7}. The model is trained using the normalized dataset, and predictions are generated on the test case using the same format (input shape $128 \times 48 \times 3$).

The model output is then reverse-normalized and mapped back onto the original fine unstructured grid using a closest-point interpolation strategy. This allows the comparison between the network predictions and the original high-fidelity CFD results on the native mesh.

The final output includes the predicted high-resolution fields for pressure, axial velocity, and heat of reaction, visually and quantitatively evaluated against the reference solution.

5.4 Normalization Strategies and U-Net Model Enhancements

The preprocessing of CFD data prior to training the U-Net model includes a crucial normalization step, necessary due to the dimensional nature of the quantities output by the Ansys simulations. Without normalization, the large scale differences between physical variables could impair training stability and convergence.

In early versions of the model, the normalization strategy applied to the pressure field did not lead to satisfactory results in terms of prediction accuracy. This prompted a search in the literature for more effective normalization approaches specifically designed for CFD datasets. An alternative strategy was identified and adapted, inspired by the method proposed in [33], though not applied identically.

The normalization strategies adopted are variable-specific:

- **Pressure:** Taking inspiration from [33], each pressure field snapshot is normalized based on its input mean value, then scaled within the range $[0, 1]$ using the global minimum and maximum values across the entire dataset.
- **Axial velocity:** Normalized globally to the range $[0, 1]$ using the minimum and maximum values from all cases.
- **Heat of reaction:** Similarly normalized globally to $[0, 1]$.

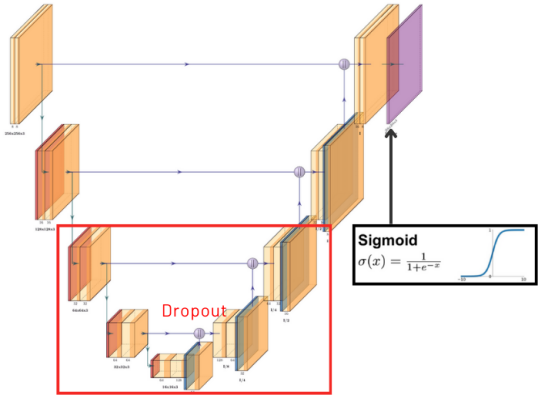

FIGURE 5.6: Aarchitectural improvements to the U-Net model.

Model Additions

To improve generalization and model robustness, the following architectural and training strategies were introduced in the U-Net:

- **Dropout:** A regularization technique that randomly deactivates a fraction of neurons during training, mitigating overfitting.
- **Sigmoid Activation in Output Layer:** Since the outputs are scaled to the $[0, 1]$ range, a sigmoid activation function is applied at the final layer to enforce this constraint naturally.
- **Batch Normalization:** Applied after each convolutional layer, this helps stabilize and accelerate training by ensuring that activations maintain zero mean and unit variance across mini-batches. It also reduces internal covariate shift.

In addition, the prediction is performed in parallel using two separate U-Net models. Each model is trained independently on a subset of the output variables and is executed concurrently during inference. This design choice enables a modular and scalable architecture, allowing the model to focus on specific physical fields and optimize the learning process.

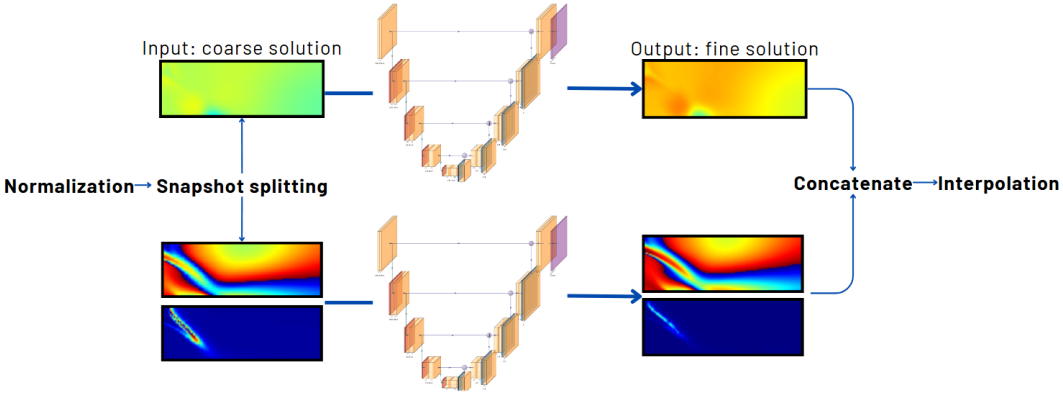

FIGURE 5.7: Parallel execution of two U-Net models, each predicting different subsets of physical quantities from normalized coarse input snapshots.

5.5 Training Procedure and Convergence

The training of the neural network models was conducted separately for two tasks: one for pressure prediction, and another for the simultaneous prediction of axial velocity and heat of reaction. Each model was trained using the Adam optimizer with adaptive learning rate scheduling. The loss function adopted was the Mean Squared Error (MSE), calculated between the predicted and true values on the normalized data.

A **custom learning rate scheduler** was implemented within the training code to manage the variation of the learning rate at different epoch stages. This scheduler allows predefined learning rate decay across user-defined epoch steps, contributing to smoother convergence and enhanced model generalization.

The final hyperparameters selected for training are reported in Table 5.1.

Parameter	Value
Batch size	6
Max filters number	128
Dropout	Active in the 3rd and 4th U-Net layers
Random state	1
Learning rate	From 2×10^{-4} to 5×10^{-5}
Epoch steps	[150, 2000, 2000, 2000]

TABLE 5.1: Summary of the final training parameters.

The convergence of the models is reported in Fig. 5.8, where both training and validation losses are plotted as a function of epochs. The model predicting the pressure field shows a smooth and rapid convergence, reaching a low validation loss early and stabilizing. The model trained on axial velocity and heat of reaction exhibits a more irregular early behavior due to the complexity of learning two coupled quantities, but stabilizes as the training progresses.

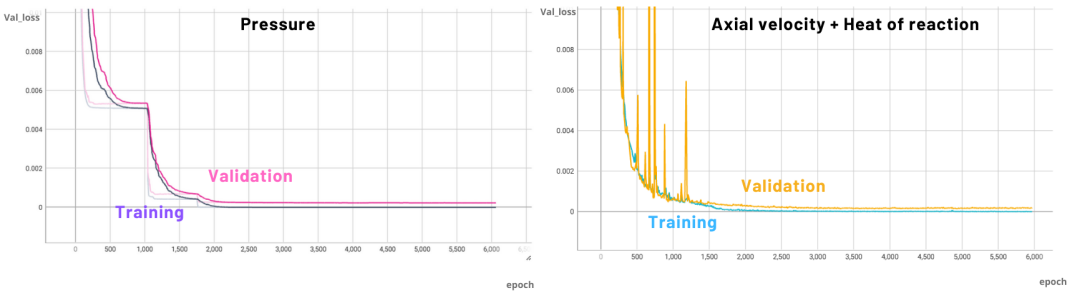

FIGURE 5.8: Training and validation loss history for both neural networks. Left: pressure prediction. Right: axial velocity and heat of reaction.

Overall, the training configuration led to stable convergence in both cases, with the model reaching satisfactory accuracy without signs of overfitting.

The use of dropout and batch normalization contributed significantly to training robustness.

5.6 Results

The performance of the trained models was evaluated on the test case using relative norm errors computed against the high-resolution CFD solution (fine grid). Two norm metrics were used: the ℓ_2 norm (global error) and the ℓ_∞ norm (maximum local error). Results were also compared to the original coarse grid solution to highlight the improvement offered by the neural network.

Among all predicted quantities, the axial velocity consistently showed the best agreement with the reference solution. This behavior can be attributed to its smoother spatial distribution and less abrupt gradients compared to pressure and heat of reaction. Pressure predictions followed closely, while heat of reaction proved to be the most challenging due to its high spatial variability and sharp peaks.

Metric	$\|\cdot\|_2$ (%)	$\|\cdot\|_\infty$ (%)
Neural Network vs Fine Grid	0.82	3.13
Coarse Grid vs Fine Grid	8.11	38.57

TABLE 5.2: Comparison of relative norm errors for Pressure.

The pressure variable presented significant challenges during training. Only with the improved architecture and the use of parallel networks was it possible to achieve the results shown in Fig. 5.9. The three selected axial sections are those where most of the discrepancies tend to occur. The model demonstrates a substantial improvement over the coarse simulation, achieving a much closer match to the fine grid data.

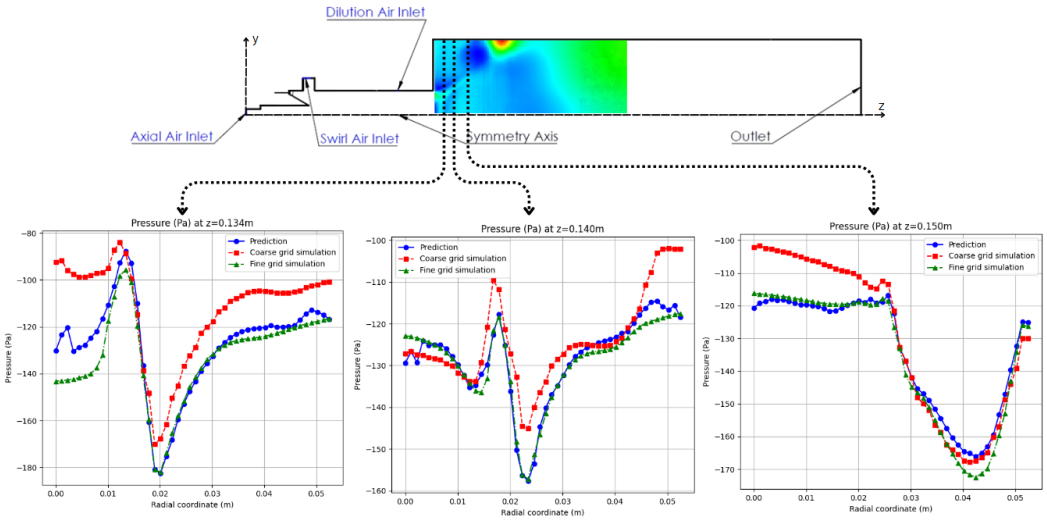

FIGURE 5.9: Radial pressure profiles at three axial positions ($z = 0.134\,\text{m}$, $z = 0.140\,\text{m}$ and $z = 0.150\,\text{m}$).

Metric	$\|\cdot\|_2$ (%)	$\|\cdot\|_\infty$ (%)
Neural Network vs Fine Grid	0.71	1.45
Coarse Grid vs Fine Grid	1.51	6.08

TABLE 5.3: Comparison of relative norm errors for Axial Velocity.

As shown in Fig. 5.10, axial velocity is the variable that is most accurately predicted by the model. This is consistent with its behavior in CFD simulations, where axial velocity tends to exhibit smoother and more regular structures. Nevertheless, the network's prediction still provides a meaningful reduction in error compared to the coarse mesh.

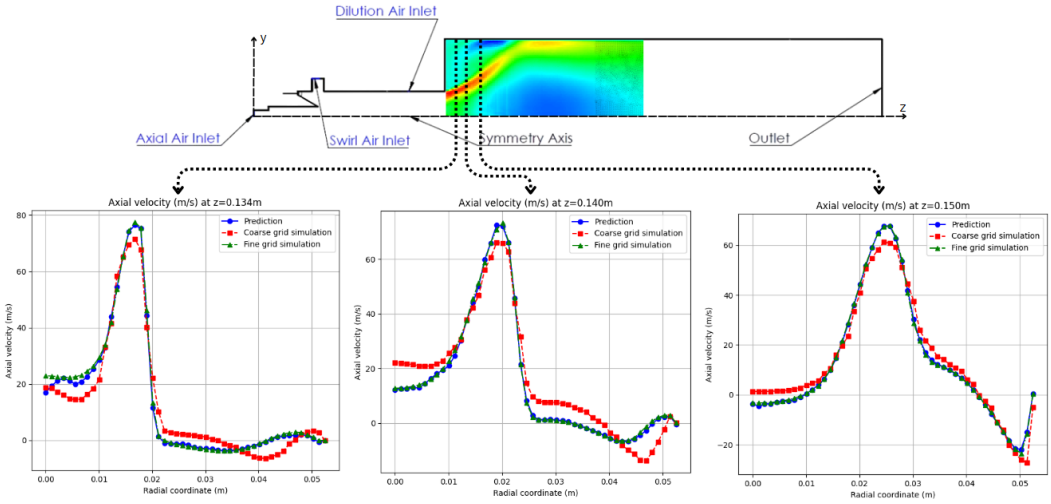

FIGURE 5.10: Radial axial velocity profiles at three axial positions.

Metric	$\|\cdot\|_2$ (%)	$\|\cdot\|_\infty$ (%)
Neural Network vs Fine Grid	3.25	3.96
Coarse Grid vs Fine Grid	855.76	631.89

TABLE 5.4: Comparison of relative norm errors for Heat of Reaction.

Finally, the heat of reaction results, illustrated in Fig. 5.11, represent the most remarkable achievement. The U-Net model is capable of predicting flame front localization and intensity far better than the coarse CFD simulation, resulting in error reductions that are orders of magnitude lower. This highlights the model's capacity to generalize even complex, highly localized phenomena.

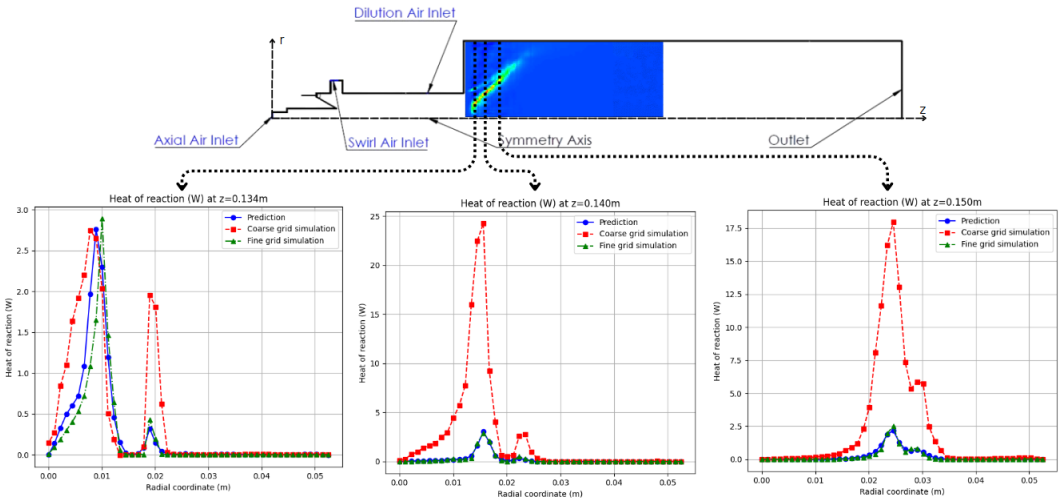

FIGURE 5.11: Radial profiles of heat of reaction at three axial positions.

5.7 Comparison Between Model Variants

To analyze the impact of model architecture and training choices, several configurations were tested. Table 5.5 summarizes the setup parameters for five cases, including filter count, batch size, dropout use, and learning rate. The most effective configuration (Case 5) combined parallel U-Net models with improved normalization.

Case options	1	2	3	4 (Only Pressure)	5 (Parallel U-net)
Filters	128	128	256	512	128
Batchsize	8	8	8	6	6
Dropout	off	on	on	[0, 0.1, 0.1, 0.1]	[0, 0.0, 0.1, 0.1]
Learning rate	1e-4	1e-4	1e-4	[4e-4, 2e-4]	[2e-4 : 5e-5]

TABLE 5.5: Model architecture and training setup for each tested case.

The relative error performance for each configuration is shown in Table 5.6, along with baseline errors from the coarse grid.

case rel error	coarse	1	2	3	4	5
ℓ_2 - Pressure	8.1	1.75	717	13.26	3.23	0.82
ℓ_∞ - Pressure	38.57	239.55	1613	29.25	1.2	3.13
ℓ_2 - Axial velocity	1.51	2.89	12.72	/	/	0.71
ℓ_∞ - Axial velocity	6.08	0.54	13.57	/	/	1.45
ℓ_2 - Heat of reaction	855.76	2.63	109	/	/	3.25
ℓ_∞ - Heat of reaction	631.89	28.94	23.27	/	/	3.96

TABLE 5.6: Comparison of relative errors for all tested configurations and coarse baseline.

This comparative analysis confirms the effectiveness of the final selected model in Case 5. It achieves the lowest error in all metrics while keeping a reasonable computational cost thanks to its dual-stream U-Net design and refined normalization strategy.

5.8 Computational Cost Analysis

The construction of the high-fidelity database required a total of **24 hours of CFD simulation** on a laptop equipped with an **Intel Core i7-8565U (8th Gen, 4 physical cores / 8 threads)**. The wall-clock execution time was **24:00:00**, corresponding to about **96 CPU-hours** of computational effort on 4 CPU cores. This step represents the *offline cost* of the methodology: it is performed once, prior to the neural network training, and provides the ground-truth data used for supervised learning.

The subsequent training of the neural networks was performed on the same hardware configuration. All times reported here correspond to **measured wall-clock times**, with TensorFlow automatically exploiting multi threading across CPU cores. It should be stressed that these times are *approximate*, since the training runs were manually stopped once a satisfactory accuracy was reached rather than at full convergence. Furthermore, the exact computational cost is influenced not only by the model size but also by the choice of **hyperparameters** (e.g., batch size, learning rate schedule, dropout, and number of epochs), meaning that the reported values should be interpreted as indicative rather than absolute. The computational cost is strongly dependent on the model size, specifically on the number of convolutional filters. In particular:

- A **U-Net with 128 filters** ($\sim$340k trainable parameters) required approximately **03:00:00** of training ($\sim$24 CPU-hours).
- A **U-Net with 512 filters** ($\sim$5 million trainable parameters) required about **13:00:00** ($\sim$104 CPU-hours).
- The **parallel U-Net configuration** (two independent networks, each with 128 filters, $\sim 2 \times$ 340k parameters) required about **06:00:00** ($\sim$48 CPU-hours).

These results show that the training cost scales with the number of parameters, but not linearly: the 512-filter model, while significantly larger, benefits from improved efficiency of multi-threaded operations, reducing the average time per parameter. It should be noted that all training was performed on CPU only; the use of GPU acceleration would drastically reduce the computational cost, enabling larger architectures and faster experimentation. A direct comparison highlights that the **offline CFD phase (96 CPU-hours)** required a similar order of magnitude of resources as the heaviest ML training run (104 CPU-hours), confirming that database generation is the true computational bottleneck of the workflow. However, once the dataset is available,

multiple training runs can be performed at a much lower additional cost. Moreover, once the network is trained, the **inference cost** to obtain a high-fidelity field from a low-resolution input is practically negligible, requiring only **0.1–0.2 seconds per field on CPU**. This represents a drastic speed-up compared to running a new CFD simulation.

Task / Model	**Parameters**	**Wall-clock Time**	**CPU-hours**
High-fidelity database (CFD)	–	∼24:00:00	96
U-Net (128 filters)	$\sim 340 \times 10^3$	∼03:00:00	∼24
Parallel U-Net (2×128 filters)	$\sim 2 \times 340 \times 10^3$	∼06:00:00	∼48
U-Net (512 filters)	$\sim 5 \times 10^6$	∼13:00:00	∼104

TABLE 5.7: Computational cost in terms of wall-clock time and CPU-hours. Training times are approximate, as runs were stopped once a satisfactory accuracy was reached and also depend on hyperparameter settings.

Chapter 6

Future Developments

The work presented in this thesis opens the way to several possible developments, both in terms of methodology and practical applications. In the following, some research directions are discussed that could significantly extend the current framework.

6.1 GPU and Heterogeneous Hardware Support

One of the most immediate improvements concerns the implementation of support for **GPU computing**. At present, the training has been performed using CPUs, which makes the process particularly time-consuming. The adoption of GPUs, or even heterogeneous hardware such as multi-GPU clusters, would considerably reduce the computational time required for training. This acceleration would make it possible to increase the size of the training datasets, to explore deeper and more complex architectures, and to perform more extensive **hyperparameter optimization**, ultimately enhancing both the accuracy and the generalization capability of the network.

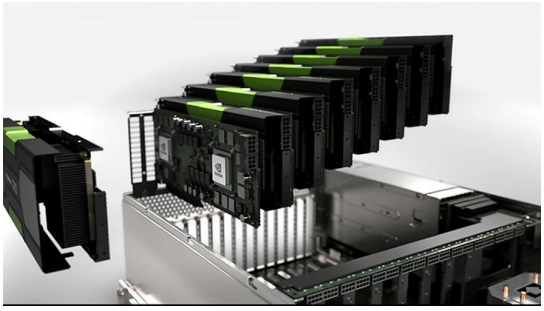

FIGURE 6.1: Example of an NVIDIA multiple GPU workstation

Furthermore, GPU acceleration would open the door to more advanced training strategies that are currently limited by CPU performance. These include **large-scale data augmentation**, **multi-resolution learning**, and the use of **ensemble methods that require repeated model training**. **Parallelization across multiple GPUs** could also allow **distributed training**, making it possible to handle three-dimensional datasets of significant size, which are otherwise impractical on standard hardware.

FIGURE 6.2: Tensorflow python library supports NVIDIA Cuda platform

In addition, GPU-enabled workflows would facilitate the integration of more sophisticated deep learning techniques, such as **adversarial training** or transfer learning, which can further improve model robustness and adaptability to new flow configurations. From a practical perspective, this computational upgrade would also shorten the experimental cycle, allowing for faster iterations between model development, testing, and validation against CFD simulations. Such efficiency gains would not only accelerate research but also increase the feasibility of applying these models to industrial problems, where quick turnaround times are often essential.

6.2 From 2D to 3D Fields

Another promising development is the extension **from two-dimensional to three-dimensional fields**. A possible strategy consists in **extracting two-dimensional slices from a three-dimensional dataset, training the U-Net on these slices, and then reconstructing the 3D solution by combining the learned features**. This slice-based strategy would allow **reusing** the well-established two-dimensional architectures, while progressively approaching the complexity of three-dimensional learning without requiring a complete redesign of the model.

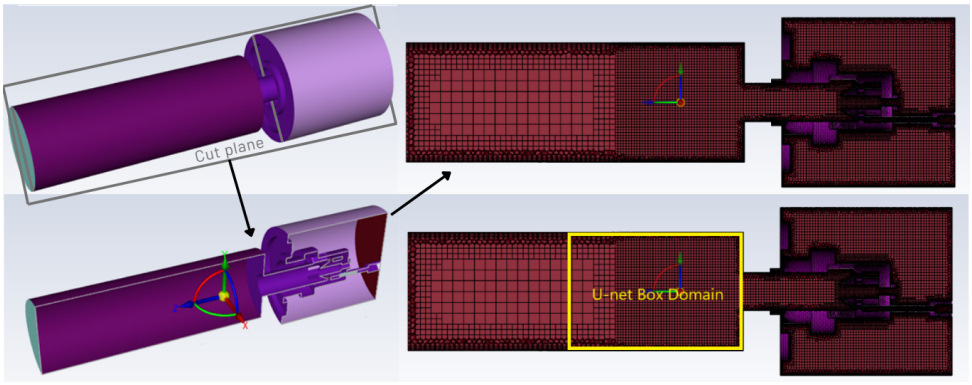

FIGURE 6.3: Conceptual representation of the 2D slice extraction from the original three-dimensional field.

Such an approach would make it possible to address more realistic problems in computational fluid dynamics, such as **three-dimensional turbulent flows, vortex interactions, or reactive combustion phenomena**, while maintaining acceptable computational costs. It would also provide a flexible compromise between model accuracy and computational feasibility, since the training process could be distributed across a large number of slices, enabling efficient parallelization.

In addition, the reconstruction of three-dimensional solutions from two-dimensional predictions could be enriched by consistency constraints across adjacent slices, ensuring physical continuity and coherence in the final volumetric fields. This idea could be further extended by employing hybrid strategies, where a first stage reconstructs 3D volumes from 2D slices and a second stage refines them using a dedicated 3D network.

The transition to 3D also opens up the possibility of directly applying **three-dimensional convolutional neural networks**, which, although more computationally demanding, can naturally capture volumetric structures, correlations, and anisotropies of fluid flows. Combined with GPU or multi-GPU acceleration, this would pave the way for handling complex simulations of industrial relevance, such as combustors, nozzles, or turbomachinery components, where three-dimensional effects are dominant.

6.3 Artificial Data Augmentation

Given the limited size of the available dataset, a valuable improvement would be the implementation of artificial data augmentation techniques. Transformations such as rotations, reflections, scaling, noise injection or synthetic perturbations consistent with physical symmetries could be applied to the training samples. These operations would enrich the dataset without requiring additional CFD simulations, thereby reducing the computational cost associated with data generation.

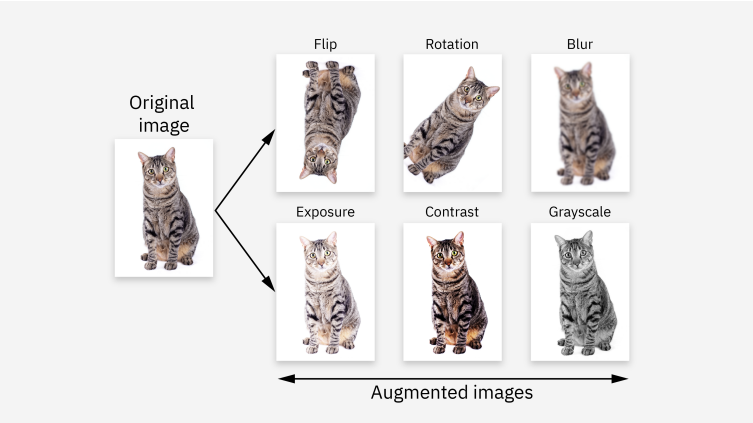

FIGURE 6.4: Data Augmentation techniques. Image from IBM.

Beyond simple geometric transformations, domain-specific augmentation strategies could be introduced to better capture the physics of fluid dynamics

problems. For instance, perturbations in boundary conditions, variations in inlet profiles, or controlled modifications of local flow features could be artificially embedded in the data to mimic the natural variability of CFD fields. Such tailored augmentations would not only increase the effective size of the dataset, but also **expose the network to a broader diversity of flow patterns**, turbulence structures, and nonlinear interactions.

Moreover, data augmentation could serve as a regularization technique, mitigating the risk of overfitting when training on relatively small datasets. By presenting the network with slightly different but physically consistent versions of the same sample, the model would be encouraged to focus on invariant and generalizable flow features rather than memorizing case-specific details. This in turn would improve the robustness of predictions and facilitate extrapolation to new geometries or flow regimes.

6.4 Increase in Physical Model Complexity

Another possible development concerns the extension of the methodology towards models of increasing physical complexity. In the present work, the neural network was trained to reproduce high-resolution solutions within a fixed physical framework. A natural evolution would consist in exploiting the learned mapping **capability of the U-Net to correct or bridge between different levels of physical modeling.**

For instance, the network could be trained to infer the correction terms needed to transform a solution computed with the inviscid Euler equations into an equivalent field that accounts for viscous or turbulent effects, as described by RANS or LES formulations. Similarly, it could learn to enhance RANS-based predictions by approximating sub-grid features typical of DES or LES simulations.

Such an approach would allow the network to act as a **data-driven closure model**, capable of enriching low-fidelity simulations with information learned from higher-fidelity databases. This would open the way to hybrid physics–machine learning frameworks, where neural networks provide physical corrections rather than merely spatial resolution enhancements, ultimately improving the predictive capability of CFD tools across multiple modeling levels.

A further step could involve the application of this methodology to more challenging flow regimes, such as supersonic and hypersonic flows, where strong shocks and discontinuities are present. In these regimes, standard convolutional networks may struggle to represent sharp gradients or capture discontinuous features accurately, often leading to excessive smoothing of shock waves or spurious oscillations near discontinuities.

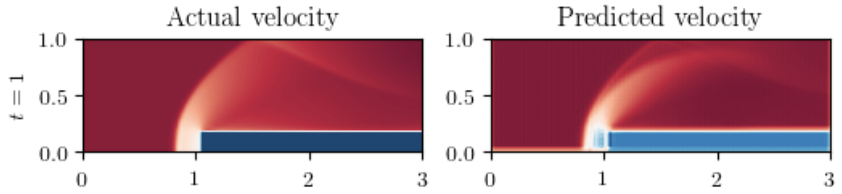

FIGURE 6.5: Supersonic velocity flow-field example. Work from [11]

Overcoming these limitations may require tailored architectures or loss functions explicitly designed to preserve discontinuities and ensure physical consistency in the presence of shocks. Successfully addressing such challenges would greatly expand the applicability of deep-learning-based super-resolution techniques to high-speed aerodynamics and aerospace propulsion, where accurate handling of compressibility effects and shock–boundary layer interactions is of critical importance.

Conclusions

The work presented in this thesis has explored the application of convolutional neural networks, and in particular the U-Net architecture, to problems of computational fluid dynamics (CFD) with the aim of achieving multi-fidelity super-resolution.

An important outcome of this work is the demonstration that, once the offline training stage is completed, the online prediction phase is extremely fast compared to traditional CFD solvers. This feature confirms the potential of machine learning methods as powerful tools for accelerating design and analysis in fluid dynamics, reducing the need for costly simulations while maintaining a satisfactory level of accuracy.

The separation of outputs into different networks (e.g. pressure versus coupled velocity and heat release) has also been shown to improve stability and convergence, offering a modular approach adaptable to different physical contexts. In addition, preliminary strategies for dataset augmentation and extrapolation towards three-dimensional fields have been outlined, opening the way to more realistic applications in aerospace propulsion and energy systems.

In summary, the thesis confirms the feasibility of applying deep learning architectures such as U-Net to CFD, bridging the gap between low- and high-fidelity solutions. **The results are promising and suggest that, with larger datasets and more advanced hardware resources, these methods could become a standard complement to classical numerical solvers.** Future developments may include GPU-based training on heterogeneous hardware, systematic artificial data augmentation, and the extension to full three-dimensional geometries with increasing physical complexity.

Final remark: This work represents a first step towards the integration of machine learning within CFD workflows. Although many challenges remain to be addressed, particularly in terms of generalization and physical consistency, **the results obtained highlight a research direction of great potential, both from an academic and an industrial perspective.**

Bibliography

[1] Mitesh.pilot, "Ai-ml-dl.svg." https://commons.wikimedia.org/wiki/File:AI-ML-DL.svg, 2022. Licensed under CC BY-SA 4.0. Accessed on 2025-03-28.

[2] Technavio, "Aerospace artificial intelligence (ai) market - growth analysis 2024–2028," 2024. Accessed on April 1, 2025.

[3] Wikimedia Commons contributors, "Neuron diagram," 2008. Accessed: 2025-03-30.

[4] Wikimedia Commons contributors, "Fully connected neural network," 2018. Accessed: 2025-03-30.

[5] Wikimedia Commons contributors, "Multi-layer neural network diagram (english)," 2017. Accessed: 2025-03-30.

[6] Wikimedia Commons contributors, "Recurrent neural network diagram (english)," 2017. Accessed: 2025-03-30.

[7] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Neural Information Processing Systems*, 2018.

[8] J. Jordan, "A visual explanation of learning rate in neural networks." https://www.jeremyjordan.me/nn-learning-rate/, 2017. Accessed: 2025-03-31.

[9] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, 2015.

[10] L. Folcarelli, A. Spagnolo, F. Dicech, A. Ferrero, F. Masseni, and D. Pastrone, "Multi-fidelity modeling of a lean premixed swirl-stabilized hydrogen burner with axial air injection," in *AIAA SCITECH 2025 Forum*, p. 0941, 2025.

[11] A. Hussain, "Image-based cfd using deep learning." https://github.com/afzalhussain23/Image-Based-CFD-Using-Deep-Learning, 2021. GitHub repository.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[14] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," *MIT Press*, 2016. Book.

[15] G. Marcus, "The next decade in ai: Four steps towards robust artificial intelligence," *arXiv preprint arXiv:2002.06177*, 2020.

[16] K. Fukami, K. Fukagata, and K. Taira, "Super-resolution reconstruction of turbulent flows with machine learning," *Journal of Fluid Mechanics*, vol. 870, pp. 106–120, 2019.

[17] A. T. Mohan, D. V. Gaitonde, and R. W. Grout, "Deep learning for compact and interpretable reduced-order models of fluid flows," *AIAA Scitech 2018 Forum*, 2018.

[18] K. Duraisamy, G. Iaccarino, and H. Xiao, "Turbulence modeling in the age of data," *Annual Review of Fluid Mechanics*, vol. 51, pp. 357–377, 2019.

[19] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[21] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, PMLR, 2015.

[24] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3431–3440, 2015.

[25] H. S. Kim and Tobias, "Cnn explainer: Learn convolutional neural networks." `https://poloclub.github.io/cnn-explainer/`, 2020.

[26] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[27] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, 2016.

[28] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[29] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[30] K. Fukami, K. Fukagata, and K. Taira, "Machine-learning-based spatio-temporal super resolution reconstruction of turbulent flows," *Journal of Fluid Mechanics*, vol. 909, p. A9, 2021.

[31] S. Lee and D. You, "Modeling of turbulent wake flows using deep learning with embedded physical invariance," *Physics of Fluids*, vol. 32, no. 3, p. 035104, 2020.

[32] L. Muscarà, M. Cisternino, A. Ferrero, A. Iob, and F. Larocca, "A comparison of local and global strategies for exploiting field inversion on separated flows at low reynolds number," *Applied Sciences*, vol. 14, no. 18, p. 8382, 2024.

[33] L. Q. Tuyen, P.-H. Chiu, and C. Ooi, "U-net-based surrogate model for evaluation of microfluidic channels," *arXiv preprint arXiv:2105.05173*, 2021.