Politecnico di Torino

Politecnico di Torino

Aerospace Engineering

Academic Year 2025/2026

A Feed-Forward Neural Network Surrogate for the Preliminary Design of Low-Thrust Interplanetary Transfers

Supervisor:
Prof. Lorenzo Casalino

Co-Supervisors:
Ing. Andrea Musacchio
Ing. Andrea D'Ottavio
Dott. Giorgio Fasano

Candidate:
Giorgio Ruello

Acknowledgements

Desidero ringraziare il Prof. Lorenzo Casalino del Politecnico di Torino, relatore di questa tesi, per la guida e la revisione del lavoro svolto.
Un ringraziamento all'Ing. Stefano Petraz per avermi dato l'opportunità di svolgere la tesi presso Thales Alenia Space Italia e per la fiducia riposta in me.
Ringrazio i miei correlatori aziendali, gli Ing. Andrea Musacchio e Andrea D'Ottavio, per avermi guidato e accompagnato lungo tutto lo svolgimento del lavoro.
Un grazie particolare al Dr. Giorgio Fasano, per il sostegno costante, i preziosi consigli, la revisione della tesi e gli insegnamenti che mi ha trasmesso.
Un ringraziamento, infine, a tutto l'ufficio Mission Analysis & Operations di Thales Alenia Space, che ha reso questi mesi un'esperienza piacevole oltre che formativa.

Un ultimo ringraziamento, primo per importanza, va ai miei genitori, da sempre instancabili sostenitori. A loro, che sono il primo motivo di ogni mio successo, devo tutto.

Abstract

This thesis proposes a surrogate model based on feed-forward neural networks for the rapid generation of optimal guess solutions for low-thrust transfers.

The use of low-thrust propulsion in interplanetary missions, increasingly adopted for its high propulsive efficiency, calls for new methods for the preliminary design of trajectories. At this stage, a large number of mission configurations and launch windows must be explored. However, generating each optimal solution requires solving a nonlinear optimization problem, which has a high computational cost and is not sustainable for extensive trade-off analyses. Machine learning-based surrogate models are one of the most promising strategies for significantly facilitating the task.

The training dataset, consisting of about 20,000 optimal trajectories, was generated using the Sims–Flanagan transcription coupled with the SNOPT7 optimizer and a metaheuristic optimization algorithm. The network adopts a hybrid architecture combining a classification component that assesses transfer feasibility and a regression component that predicts the required Δv and, consequently, the propellant mass.

Applied to Earth–Mars low-thrust transfers, the model achieves highly satisfactory results for both components. Besides accurately reproducing the reference solutions, the network generalizes to engine/spacecraft configurations unseen during training — within the trained ranges and for the same launch dates — and generates dense-grid porkchop plots from a limited dataset; temporal extrapolation beyond the trained interval, however, proves less accurate.

The computational advantage is significant: generating the dataset required about 1630 hours of computation, reduced to roughly 40 hours through multiprocessing, whereas the trained network predicts about 13.5 million points, with values close to the optimum, in approximately one minute.

Overall, the results demonstrate that a standalone surrogate model can rapidly provide valid guess solutions and generalize to configurations not included in training, establishing itself as a practical and efficient tool for the preliminary design of low-thrust interplanetary missions.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Electric propulsion has become an increasingly common technology for interplanetary missions. Missions such as NASA's Deep Space 1 [1] and Dawn [2], JAXA's Hayabusa [3] and Hayabusa2 [4], and ESA's BepiColombo [5] have demonstrated its most valuable characteristic in flight: a specific impulse an order of magnitude higher than that of chemical propulsion, which translates, for a given initial mass, into drastically reduced propellant consumption.

This efficiency, however, comes at a price, which is paid in the design of the trajectory. The available thrust is on the order of millinewtons up to a few newtons: to accumulate the required velocity increment, the thruster must operate continuously for months or years. The maneuver can no longer be approximated as a discrete, impulsive event, as is customary for chemical propulsion, and the trajectory can no longer be decomposed into conic arcs joined by impulses: the design of the transfer becomes an optimal control problem, in which the unknown is the entire thrust law over time. The numerical solution of such a problem (typically by means of direct methods [6, 7], which transcribe it into a large-scale nonlinear programming problem, non-convex and rich in local minima) carries a considerable computational cost: a single optimal trajectory requires seconds to minutes of computation, when the optimizer converges at all.

It is in the preliminary design phase that this cost becomes a structural obstacle. Countless mission alternatives must be surveyed before any of them is worth refining, and the design space must be explored along all the dimensions that define a mission: when to launch, how long to fly, which propulsion system to adopt, how much mass to carry. Each point of this exploration requires the solution of its own optimization problem: the cost of a single evaluation, multiplied by the thousands of points needed, makes an exhaustive survey impractical, forcing in practice sparse samplings or partial analyses. The computational cost thus becomes the factor that limits the breadth of the exploration, precisely in the phase where the widest possible survey would be most valuable.

A promising answer to this obstacle comes from surrogate models [8]: computationally inexpensive mathematical models that, trained on a set of precomputed solutions, approximate the relation between the parameters of a transfer and its outcome, replacing the optimizer in extensive evaluations. Their use in trajectory optimization has attracted growing interest over the past decade, with applications ranging from the approximation of objective functions in evolutionary optimization [9, 10] to the rapid estimation of the cost of low-thrust transfers by means of deep neural networks [11, 12, 13]. In most of these studies, however, the surrogate is trained and employed for a fixed spacecraft configuration: generalization to different propulsion configurations remains comparatively less explored.

It is here that the present thesis is situated. The objective is the development of a surrogate model based on feed-forward neural networks for the rapid generation of estimates close to the optimal solutions computed by the direct solver, with specific application to low-thrust Earth–Mars transfers. The surrogate is conceived as a cascade pipeline of two distinct networks: a classifier, which establishes whether a given combination of launch window and propulsion parameters admits a physically realizable transfer, and a regressor, which for the realizable transfers alone estimates the required Δv and, from it, the final mass of the spacecraft. The two networks are trained on a dataset of optimal trajectories generated by means of the Sims–Flanagan [14], solved by the SNOPT7 optimizer [15] wrapped within a Monotonic-Basin-Hopping global search [16, 17], for eleven propulsion configurations built from real electric thrusters.

The main contributions of this work are fourfold. First, the demonstration that the surrogate generalizes to propulsion configurations never seen during training, provided their parameters lie within the sampled domain: queried on engines absent from the dataset, the model retains performance close to the nominal one, qualifying as a standalone predictive tool rather than a mere interpolator. Second, the generation of dense porkchop plots [18](maps of the transfer cost over a grid of departure dates and times of flight) from a limited training dataset, at a computational gain of roughly seven orders of magnitude per point with respect to the direct solver. Third, the systematic characterization of the limitations of the tool: the interpolating nature of its generalization, the degradation under temporal extrapolation, and the role played by the coverage of the propulsion parameter space. Fourth, the analysis of the influence of the composition of the input vector, which shows that the quality of the generalization is determined not by the number of features, but by their physical nature. The thesis is organized as follows.

Chapter 2 introduces the principles of electric propulsion and the formulation of the trajectory optimization problem, and reviews the two families of numerical methods available for its solution, motivating the choice of direct methods adopted in this work.

Chapter 3 presents the dynamic model and the Sims–Flanagan transcription

by which the optimal control problem is reduced to a finite-dimensional nonlinear programming problem. The chapter closes with a review of surrogate modeling techniques and with the rationale for the adoption of artificial neural networks.

Chapter 4 is devoted to the theoretical foundations of artificial neural networks: the perceptron, the architecture of deep feed-forward networks, activation and loss functions, training by gradient descent and backpropagation, and the techniques employed to counter overfitting.

Chapter 5 describes the generation of the training dataset: the eleven propulsion configurations, the trajectory model, the optimization strategy combining SNOPT7 and Monotonic-Basin-Hopping, the design of the grid, the definition of feasibility, and the parallelization of the computation.

Chapter 6 sets out the implementation of the surrogate: the composition of the input feature vector, the partitioning of the dataset, the architectures of the two networks and the criteria by which they were trained, the metrics adopted, and the protocol by which the cascade pipeline is evaluated.

Chapter 7 reports the results: the performance of the two networks in isolation and in cascade, both in-distribution and on the held-out configurations, the comparison between the predicted and the reference porkchop plots, the computational gain, and a series of complementary analyses probing the robustness and the limits of the surrogate.

Chapter 8 summarizes the results, discusses the practical role of the surrogate in preliminary mission design, and outlines the limitations of the work together with the directions along which it may be continued.

Chapter 2

Low-Thrust Propulsion and Trajectory Optimization

2.1 Principles of Electric Propulsion

2.1.1 The Rocket Equation and Specific Impulse

The starting point for quantifying the performance of any propulsion system is the Tsiolkovsky rocket equation, which relates the velocity increment Δv imparted to a vehicle to the fraction of propellant mass expended:

$$\Delta v = c \ln\left(\frac{m_0}{m_f}\right) = g_0 \, I_{sp} \ln\left(\frac{m_0}{m_f}\right), \tag{2.1}$$

where m_0 and m_f are the initial and final mass, c is the effective exhaust velocity, and g_0 the standard gravitational acceleration. The specific impulse $I_{sp} = c/g_0$, expressed in seconds, measures how efficiently a thruster converts propellant mass into impulse: the higher the value of I_{sp}, the smaller the propellant mass required for a given Δv.

The exponential dependence makes this parameter decisive. For a given mission, doubling the specific impulse sharply reduces the required propellant fraction, freeing mass for the payload or enabling maneuvers that would otherwise be unfeasible. This is precisely the structural advantage of electric propulsion, whose I_{sp} values are an order of magnitude larger than those of chemical propulsion.

2.1.2 The Power Constraint

The fundamental distinction between the two families lies in the energy source. In chemical propulsion the energy is stored in the bonds of the propellant itself

and is released through combustion; power is therefore not a sizing constraint. In electric propulsion [19], by contrast, the propellant is accelerated electrostatically or electromagnetically using energy supplied by an external source (solar arrays or, for deep-space missions, radioisotope generators or nuclear reactors) and the available power becomes the dominant constraint.

The jet power is related to the thrust T and the exhaust velocity by

$$P_{jet} = \tfrac{1}{2}\,\dot{m}\,c^2 = \tfrac{1}{2}\,T\,c. \tag{2.2}$$

Introducing the thruster efficiency η (the ratio of jet power to input electrical power), the required power is

$$P_{in} = \frac{T\,c}{2\eta} = \frac{T\,g_0\,I_{sp}}{2\eta}. \tag{2.3}$$

Equation (2.3) expresses the central trade-off of electric propulsion: for a limited power level, the product of thrust and specific impulse is bounded. Operating at high I_{sp} (the condition required for propellant savings) necessarily entails modest thrust levels, of the order of millinewtons up to a few newtons. Combined with vehicle masses of hundreds or thousands of kilograms, this yields a characteristic acceleration of the order of 10^{-4}–10^{-3} m/s^2, the very condition that gives low-thrust propulsion its name.

2.1.3 Comparison with Chemical Propulsion and Implications

The two technologies thus operate in opposite regimes. Chemical propulsion delivers high thrust over short intervals, which can be approximated as impulsive maneuvers; electric propulsion produces small but sustained thrust over months or years, accumulating Δv continuously. Table 2.1 summarizes the contrast.

	Chemical propulsion	**Electric propulsion**
Typical I_{sp}	200–450 s	1000–5000 s
Thrust	N – kN	mN – N
Thrusting duration	minutes	months – years
Dominant constraint	propellant mass	available power

Table 2.1: Typical operating regimes of chemical and electric propulsion in space [19]

This difference has a direct consequence on trajectory modeling. Whereas in chemical propulsion the maneuvers can be treated as instantaneous velocity

changes at discrete points, electric thrust acts continuously along extended arcs, gradually reshaping the orbit. The trajectory can no longer be decomposed into conic segments joined by impulses, but is instead governed by a continuous control of the thrust over time. It is this feature that makes the optimization of low-thrust trajectories a substantially more complex problem.

2.2 Formulation of the Trajectory Optimization Problem

The design of a low-thrust trajectory is naturally formulated as an *optimal control problem* (OCP) [20]: one seeks the thrust law over time that transfers the vehicle from a prescribed initial state to a prescribed final state, satisfying the equations of motion and the operational constraints, while optimizing a performance index. Unlike impulsive propulsion, where the unknowns are a few discrete parameters (the epochs and magnitudes of the maneuvers), here the unknown is a continuous function of time, which places the problem within the domain of optimal control.

2.2.1 State Variables and Dynamics

The state of the vehicle is described by the position vector, the velocity vector, and the mass,

$$\mathbf{x}(t) = \Big[\, \mathbf{r}(t),\ \mathbf{v}(t),\ m(t) \,\Big]^{\mathsf{T}} \in \mathbb{R}^7, \tag{2.4}$$

and its evolution is governed by dynamics of the general form

$$\dot{\mathbf{x}}(t) = \mathbf{f}\Big(\mathbf{x}(t), \mathbf{u}(t), t\Big), \tag{2.5}$$

where $\mathbf{u}(t)$ is the control variable. For the problem at hand, the function $\mathbf{f}$ corresponds to two-body motion controlled by the thrust, with the mass consumption related to the thrust magnitude through the effective exhaust velocity $c = g_0 I_{sp}$. The explicit dynamic model is developed in Chapter 3.

2.2.2 Control and Constraints

The control variable is the thrust vector, which is conveniently decomposed into magnitude and direction:

$$\mathbf{u}(t) = \mathbf{T}(t) = T(t)\,\hat{\mathbf{u}}(t), \qquad 0 \le T(t) \le T_{\max}, \qquad \|\hat{\mathbf{u}}(t)\| = 1. \tag{2.6}$$

The magnitude is bounded by the maximum thrust the thruster can deliver, whereas the direction is free, save for any pointing constraints.

2.2.3 Boundary Conditions

The initial and final states are prescribed by the positions and velocities of the departure and arrival bodies at the corresponding epochs:

$$\mathbf{r}(t_0) = \mathbf{r}_0, \qquad \mathbf{v}(t_0) = \mathbf{v}_0, \qquad m(t_0) = m_0, \tag{2.7}$$

$$\mathbf{r}(t_f) = \mathbf{r}_f, \qquad \mathbf{v}(t_f) = \mathbf{v}_f. \tag{2.8}$$

Constraining both position and velocity to those of the target body corresponds to a *rendezvous* condition; in the case of a *fly-by*, only the position is constrained. The epochs t_0 and t_f — and hence the transfer time $t_{of} = t_f - t_0$ — may be either fixed or left free, depending on the formulation.

2.2.4 Performance Index

The objective is expressed by a cost functional, which can be cast in three equivalent forms. In its most general, *Bolza* form, the functional comprises a terminal term and an integral term,

$$J = \phi\Big(\mathbf{x}(t_f), t_f\Big) + \int_{t_0}^{t_f} \mathcal{L}\Big(\mathbf{x}(t), \mathbf{u}(t), t\Big)\, \mathrm{d}t. \tag{2.9}$$

The *Lagrange* form retains only the integral term,

$$J = \int_{t_0}^{t_f} \mathcal{L}\Big(\mathbf{x}(t), \mathbf{u}(t), t\Big)\, \mathrm{d}t, \tag{2.10}$$

whereas the *Mayer* form retains only the terminal term,

$$J = \phi\Big(\mathbf{x}(t_f), t_f\Big). \tag{2.11}$$

The three forms are equivalent, as each can be converted into the others through a suitable augmentation of the state vector [20].

The two cases of practical interest for low-thrust transfers are the *maximum final mass* (equivalent to minimum propellant), $J = -\,m(t_f)$, and the *minimum transfer time*, $J = t_f - t_0$. For low-thrust transfers the dominant objective is propellant saving, since the useful mass fraction is the critical design constraint; the maximum final mass, in Mayer form, is therefore the objective adopted throughout the remainder of this work.

2.2.5 Complete Optimal Control Problem

Collecting the terms, the problem is stated as follows: determine the control $\mathbf{u}(t)$, together with any free parameters $t_0,\ t_f$, that minimize J subject to the dynamics, the control constraints, any path constraints, and the boundary conditions. This is an infinite-dimensional problem, generally nonlinear and lacking an analytical solution, whose resolution requires numerical methods.

2.3 Solution Methods: Direct and Indirect Approaches

The numerical methods required to solve the optimal control problem formulated in Section 2.2 are divided into two broad families, distinguished by the order in which optimization and discretization are applied. *Indirect methods* optimize first and discretize afterwards (*optimize-then-discretize*): they derive the necessary conditions of optimality analytically and solve the resulting system numerically. *Direct methods* discretize first and optimize afterwards (*discretize-then-optimize*): they transform the continuous problem into a finite-dimensional parameter optimization problem, solved with nonlinear programming techniques.

2.3.1 Indirect Methods

Indirect methods are grounded in the calculus of variations and in Pontryagin's maximum principle [21, 20]. A vector of costates (or adjoint variables) $\boldsymbol{\lambda}(t)$ is introduced and the Hamiltonian of the system is constructed; the necessary conditions of optimality impose the Euler–Lagrange equations for the evolution of the costates, the optimal control law that minimizes the Hamiltonian, and the appropriate transversality conditions at the boundaries. The result is a *two-point boundary value problem* (TPBVP), in which one must determine the initial values of the costates that satisfy the terminal conditions.

For classical minimum-propellant problems in which the Hamiltonian is affine in the thrust magnitude and the admissible thrust satisfies $0 \leq T(t) \leq T_{\max}$, minimizing the Hamiltonian leads to a *bang–bang* thrust law, with switching between zero and maximum thrust according to the sign of a *switching function*. In more general formulations, such as problems with nonlinear thrust costs, lower thrust bounds, singular arcs, or convexified powered-descent guidance, the optimal thrust may instead vary continuously, and switching-function analysis is not necessarily applicable.

The chief merit of this approach is its high accuracy: the solution satisfies the necessary conditions of optimality in analytical form, and the size of the numerical problem remains contained. On the other hand, indirect methods suffer from a *narrow radius of convergence* and a strong sensitivity to the initial guess of the costates, which are quantities devoid of intuitive physical meaning and therefore difficult to initialize. The presence of bang–bang discontinuities and the need to derive the necessary conditions analytically further limit their flexibility.

2.3.2 Direct Methods

Direct methods circumvent the analytical derivation of the optimality conditions by discretizing the state and control directly on a time grid [6]. The continuous variables are thus replaced by a finite set of parameters, and the cost functional, the dynamics, and the constraints are evaluated at those parameters: the optimal control problem reduces to a *nonlinear programming* (NLP) problem, solved with algorithms such as sequential quadratic programming (SQP) or interior-point methods. The discretization techniques (transcription methods) differ in how the state and control are parametrized and in how the dynamics are enforced as a constraint [7].

Compared with indirect methods, direct methods offer a *wider radius of convergence* and greater robustness with respect to the initial guess, which is moreover expressed in physically interpretable variables (positions, velocities, thrusts) and therefore easier to provide. The handling of path and inequality constraints is natural, as they integrate directly into the NLP formulation. The price is a *higher dimensionality* of the numerical problem and only approximate optimality: the solution converges to the optimal one as the discretization is refined, but does not satisfy the optimality conditions in exact form as in the indirect case.

2.3.3 Choice of Approach

Table 2.2 summarizes the comparison between the two families.

	Indirect methods	**Direct methods**
Principle	optimize-then-discretize	discretize-then-optimize
Theoretical basis	Pontryagin's maximum principle	nonlinear programming
Unknowns	costates (non-intuitive)	state and control (physical)
Radius of convergence	narrow	wide
Accuracy	high (exact optimality)	approximate optimality
Dimensionality	contained	high
Constraint handling	complex	natural

Table 2.2: Comparison of indirect and direct methods

The choice in this work falls on direct methods. The rationale is twofold and tied to the objective of the thesis: the systematic exploration of a large design space requires robustness and reliable convergence over a great number of heterogeneous transfers, where the sensitivity of indirect methods to the initial guess would be

an obstacle; furthermore, the use of physically interpretable variables is consistent with the generation of a trajectory dataset and with the subsequent construction of the surrogate models.

Chapter 3

Dynamic Model

The design of a low-thrust trajectory requires, first of all, a dynamic model describing the motion of the spacecraft under the gravitational attraction of the central body and the continuous thrust of its propulsion system. Consistently with the preliminary design purpose of this work, the simplest model adequate to the task is adopted: the motion is described within the framework of the restricted two-body problem, in a heliocentric inertial reference frame in which the Sun is the primary body [22]. Orbital perturbations (the attraction of the planets, solar radiation pressure, and the non-sphericity of the bodies) are neglected, as their effect is irrelevant at the level of fidelity required at this stage and their inclusion would be incompatible with the extensive exploration of the design space pursued in this work.

The state of the spacecraft is fully described by the position vector $\mathbf{r}$, the velocity vector $\mathbf{v}$, and the mass m, where the position and velocity are expressed through their Cartesian components $\mathbf{r} = (r_x, r_y, r_z)$ and $\mathbf{v} = (v_x, v_y, v_z)$. The thrust is represented by the vector $\mathbf{T} = T\,\hat{\boldsymbol{u}}$, where T is its magnitude and $\hat{\boldsymbol{u}}$ the unit vector defining its direction, subject to the physical limit imposed by the engine,

$$0 \leq T \leq T_{\max}. \tag{3.1}$$

Under the assumptions above, the motion is governed by the differential equations

$$\dot{\mathbf{r}} = \mathbf{v}, \tag{3.2}$$

$$\dot{\mathbf{v}} = -\frac{\mu}{r^3}\,\mathbf{r} + \frac{T}{m}\,\hat{\boldsymbol{u}}, \tag{3.3}$$

$$\dot{m} = -\frac{T}{g_0\, I_{sp}}, \tag{3.4}$$

where $r = |\mathbf{r}|$, μ is the gravitational parameter of the Sun, g_0 is the standard gravitational acceleration, and I_{sp} is the specific impulse of the engine. In the equation for $\dot{\mathbf{v}}$, the first term is the Keplerian gravitational acceleration, while the second is the acceleration produced by the thrust. The last equation expresses the propellant consumption through the mass flow rate, related to the thrust magnitude by the effective exhaust velocity $c = g_0\, I_{sp}$.

The equations of motion formulated in this way constitute a nonlinear system whose integration, for a given continuous thrust law $\Big(T(t), \hat{\boldsymbol{u}}(t)\Big)$, admits no closed-form solution. It is precisely this difficulty that motivates the adoption of a direct transcription method: as described in the following section, the Sims–Flanagan method approximates the continuous thrust with a sequence of impulsive maneuvers, reducing the propagation of the motion to purely Keplerian arcs and turning the optimal control problem into a finite-dimensional nonlinear programming problem.

3.1 The Sims–Flanagan Transcription Method

In this work, the choice fell on a direct method, the Sims–Flanagan method [14, 17], which recasts the optimal control problem into a finite-dimensional parametric optimization problem. The fundamental unit of the transcription is the *leg*, that is the portion of trajectory between two control nodes: in the case of a direct transfer between two planets, the leg begins at the departure planet and ends at the arrival planet. Each leg is subdivided into N segments of equal length

$$\Delta t = \frac{t_f - t_0}{N}, \tag{3.5}$$

where t_0 and t_f are, respectively, the initial and final epoch of the leg. The overall structure of the transcription is illustrated in Figure 3.1.

Within each segment, the continuous thrust that would be applied over the corresponding interval is approximated by a single impulsive maneuver, centered at the midpoint of the segment. The impulse applied in the i-th segment is expressed in the "*up-to-unit* vector control" form [23]

$$\Delta\mathbf{v}_i = \mathbf{u}_i\, \Delta V_{\max,i}, \qquad \|\mathbf{u}_i\| \leq 1, \tag{3.6}$$

where the vector $\mathbf{u}_i \in \mathbb{R}^3$ is the control of the segment and $\Delta V_{\max,i}$ is the maximum velocity increment that the spacecraft can accumulate when operating at maximum thrust over the entire duration of the segment,

$$\Delta V_{\max,i} = \frac{T_{\max}}{m_i}\, \Delta t. \tag{3.7}$$

The constraint $\|\mathbf{u}_i\| \leq 1$ ensures that the impulse does not exceed this physical limit, reproducing the thrust saturation of the real engine; the direction of $\mathbf{u}_i$

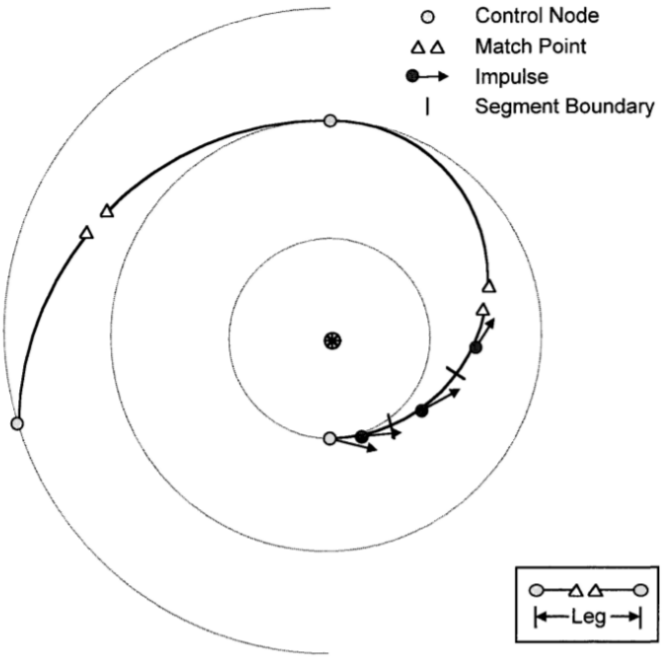


Figure 3.1: Example of the structure of a Sims–Flanagan trajectory with two legs. Reproduced from [14]

represents the thrust orientation, and its magnitude the fraction of thrust delivered. The propellant consumption associated with each impulse is described by the rocket equation,

$$m_{i+1} = m_i \, \exp\left(-\frac{|\Delta \mathbf{v}_i|}{g_0 \, I_{sp}}\right), \tag{3.8}$$

which updates the mass of the spacecraft after the maneuver is applied.

Between one impulse and the next, the spacecraft is subject to no propulsive action and moves along a Keplerian arc, according to the two-body model introduced in the previous section; the propagation thus reduces to a succession of conic arcs interspersed with instantaneous maneuvers, avoiding the numerical integration of the equations of motion. The leg trajectory is propagated in two distinct parts: forward from the initial node and backward from the final node, up to a common *match point*, conventionally located halfway along the leg in time (see Figure 3.1). In general, for an arbitrary combination of the parameters defining the trajectory, the two parts do not perfectly join: the state of the spacecraft propagated forward does not coincide with the one propagated backward at the match point. The

conditions enforcing this junction, together with the constraints on the control, are the subject of the following subsection.

3.1.1 Match-Point Constraints

As anticipated above, propagating the leg forward from the initial node and backward from the final node leads, in general, to two distinct states at the match point. For the trajectory to be physically admissible, that is to represent a continuous and realizable motion, the two states must coincide. Denoting by $\mathbf{x} = (\mathbf{r}, \mathbf{v}, m)$ the full spacecraft state, comprising position, velocity and mass, and by $\mathbf{x}_{\text{fw}}$ and $\mathbf{x}_{\text{bw}}$ the states obtained respectively from the forward and backward propagation up to the match point, the junction condition translates into the vanishing of the mismatch vector

$$\boldsymbol{\varepsilon} = \mathbf{x}_{\text{fw}} - \mathbf{x}_{\text{bw}} = \begin{pmatrix} \mathbf{r}_{\text{fw}} - \mathbf{r}_{\text{bw}} \\ \mathbf{v}_{\text{fw}} - \mathbf{v}_{\text{bw}} \\ m_{\text{fw}} - m_{\text{bw}} \end{pmatrix} = \mathbf{0}. \tag{3.9}$$

This relation collects seven scalar equality constraints, enforcing the continuity of the three position components, the three velocity components and the mass at the match point [23]. A trajectory that satisfies them is said to be feasible; in numerical practice, the vanishing is required to be within a prescribed tolerance, below which the mismatch is regarded as negligible. It is these constraints that implicitly fix the parameters defining the trajectory: an arbitrary combination of such parameters generally yields a non-zero mismatch, and it is the task of the optimizer to find the one that nullifies it.

Alongside the junction constraints, the problem is subject to the control constraints already introduced with the up-to-unit formulation. For each of the N segments, the magnitude of the control vector cannot exceed unity,

$$\|\mathbf{u}_i\| \leq 1, \qquad i = 1, \ldots, N \tag{3.10}$$

These are N inequality constraints, one per segment.

The set of equality constraints at the match point and inequality constraints on the control constitutes the entirety of the problem's constraints. Together with the objective function, presented next, they define the nonlinear programming problem that governs the search for the optimal trajectory.

3.1.2 Objective Function and NLP Formulation

The transcription described above reduces the trajectory to a finite set of parameters, which constitute the decision variables of the optimization problem. For a direct

transfer between two planets, these parameters are collected in the decision vector

$$\mathbf{z} = \left(t_0,\ t_{of},\ m_f,\ \mathbf{v}_\infty^{\mathrm{dep}},\ \mathbf{v}_\infty^{\mathrm{arr}},\ \mathbf{u}_1, \ldots, \mathbf{u}_N\right), \tag{3.11}$$

where t_0 is the departure epoch, $t_{of} = t_f - t_0$ the transfer time, m_f the final mass of the spacecraft, $\mathbf{v}_\infty^{\mathrm{dep}}$ and $\mathbf{v}_\infty^{\mathrm{arr}}$ the velocities relative to the departure and arrival planets, respectively, and $\mathbf{u}_1, \ldots, \mathbf{u}_N$ the control vectors of the N segments. The relative velocities allow the spacecraft to leave the departure planet and reach the arrival planet with a non-zero excess velocity, while the controls define the thrust law along the leg.

The objective of the optimization is the maximization of the final mass m_f. Since the initial mass of the spacecraft m_0 is fixed, maximizing the delivered mass is equivalent to minimizing the propellant consumed over the transfer, $m_0 - m_f$, and is therefore the quantity of primary interest in the design of a low-thrust mission. The objective function is formulated, according to the minimization convention, as

$$J(\mathbf{z}) = -m_f. \tag{3.12}$$

Collecting the objective function and the constraints introduced in the previous subsection, the search for the optimal trajectory reduces to the following nonlinear programming (NLP) problem:

$$\begin{aligned} \min_{\mathbf{z}} \quad & J(\mathbf{z}) = -m_f \\ \text{subject to} \quad & \boldsymbol{\varepsilon}(\mathbf{z}) = \mathbf{0}, \\ & \|\mathbf{u}_i\| \leq 1, \quad i = 1, \ldots, N, \\ & \mathbf{z}_L \leq \mathbf{z} \leq \mathbf{z}_U, \end{aligned} \tag{3.13}$$

in which $\boldsymbol{\varepsilon}(\mathbf{z}) = \mathbf{0}$ collects the seven equality constraints at the match point, $\|\mathbf{u}_i\| \leq 1$ the N inequality constraints on the control, and $\mathbf{z}_L$, $\mathbf{z}_U$ the lower and upper bounds imposed on the decision variables.

The problem formulated in this way is strongly nonlinear and non-convex, and generally exhibits a multiplicity of local minima [17]. Its solution therefore requires a gradient-based NLP solver, supported by a global search strategy capable of exploring the solution space and of converging even from bad initial guesses. The method described in this chapter is realized in practice through the Sims–Flanagan implementation available in the `pykep` library [24]; the specific configuration adopted in this work (the choice of parameters, the bounds on the variables, and the optimization procedure) is detailed in Chapter 5.

3.2 Surrogate Models for Trajectory Optimization

The dynamic model and its Sims–Flanagan transcription, combined with a gradient-based NLP solver and a global search strategy, provide an accurate means of computing a single optimal transfer. Preliminary mission design, however, rarely requires an isolated solution: it demands the exploration of broad grids of departure and arrival dates and of numerous engine and spacecraft configurations, each entailing its own optimization. Repeating the procedure described above over thousands of such cases rapidly becomes computationally prohibitive, and it is this cost that motivates the introduction of a *surrogate model*.

A surrogate model (or metamodel) is a computationally inexpensive mathematical model, built from a set of precomputed solutions, that approximates the functional relationship between the parameters defining the problem (typically the boundary conditions of the transfer) and the output quantities of interest, such as the optimal cost, the propellant mass, or the very feasibility of the solution. Once trained, the surrogate replaces the expensive direct evaluations, providing a nearly instantaneous estimate and allowing the design space to be explored with a drastically reduced number of "exact" evaluations.

The use of surrogate models in aerospace design is well established, and its extension to trajectory optimization has attracted growing interest, particularly for global problems and for the screening of large catalogs of candidate transfers. Their role is not to replace the high-fidelity optimizer, but to complement it: the surrogate provides a fast preliminary mapping of the design space, steering computational resources toward the most promising regions, where the exact solution can subsequently be refined.

3.2.1 A brief overview of surrogate modeling techniques

A wide range of metamodeling techniques is available in the literature, differing in the way the approximation is constructed from the data. A first useful distinction is that between *regression* methods, which approximate the response by minimizing a residual with respect to the data without necessarily passing through them, and *interpolation* methods, which instead enforce an exact fit at the sample points. The most widely used techniques in the optimization of engineering systems are briefly reviewed below; none of them simultaneously satisfies the requirements that the present problem imposes, such as a high-dimensional input space, a large training set, and multiple heterogeneous outputs [8, 25, 26].

- *Response Surface Methodology* (RSM) fits the response with a low-order polynomial, typically linear or quadratic, whose coefficients are estimated by

least squares. It is the simplest and most interpretable option, but it can hardly capture strong nonlinearities, and the number of terms grows rapidly with the input dimensionality — a decisive drawback for the high-dimensional parameter space considered here.

- *Kriging*, or *Gaussian Process Regression*, treats the unknown function as the realization of a Gaussian process and interpolates the samples, offering high accuracy even under strong nonlinearities together with a built-in estimate of the prediction uncertainty. Its training cost, however, scales as $\mathcal{O}(N^3)$ with the number of samples, which becomes prohibitive for datasets of the size used in this work.

- *Radial Basis Functions* (RBFs) construct the interpolant as a linear combination of radially symmetric functions centered at the sample points. The method is effective for moderate-dimensional spaces, but the quality of the approximation strictly depends on the choice of the shape parameter, and the conditioning of the system rapidly deteriorates as the number of points grows.

- *Support Vector Regression* (SVR) seeks a function that deviates from the data within a prescribed margin, exploiting the *kernel trick* to handle nonlinearities. It generalizes well and is robust against overfitting, but it accommodates multiple, heterogeneous outputs (such as the feasibility flag and the Δv of interest here) only awkwardly.

These shared limitations are precisely what motivate the recourse to artificial neural networks, examined next.

3.2.2 Why artificial neural networks?

Artificial neural networks are particularly well suited to the approximation of strongly nonlinear, high-dimensional problems such as trajectory optimization, a context in which they have attracted growing interest [11, 13, 27]. The fundamental theoretical justification lies in their universal approximation property: a feed-forward network with a sufficient number of neurons can approximate any continuous function to arbitrary accuracy [28, 29]. This makes them natural tools for representing the complex, markedly nonlinear relationships that link the parameters of a transfer to its optimal cost, where approximations of fixed form (such as polynomial ones) prove inadequate.

Unlike methods such as Kriging, whose cost grows unfavorably with the sample size, neural networks scale effectively to large datasets and, in fact, benefit from larger datasets during training compared to traditional interpolators. This is particularly relevant when, as in the present work, a substantial set of thousands

of precomputed optimal solutions is available, which would challenge the memory and computational limits of exact interpolating methods.

Neural networks also handle high-dimensional input and output spaces with ease, without the explosion in the number of terms that characterizes polynomial approaches. They are, in fact, remarkably flexible with respect to both the number of inputs and the number of outputs: a single network can be trained to predict several, even heterogeneous, outputs simultaneously, provided that its structure adequately captures the underlying nature of the problem; alternatively, distinct networks, each dedicated to a specific task, can be combined into a single predictive scheme. The latter is the strategy adopted in the present work, in which a classifier that assesses the feasibility of a transfer is connected to a regressor that estimates the optimal quantities — a modularity hardly attainable with the techniques described earlier.

Finally, once training is complete, evaluating the network is nearly instantaneous, which makes it ideal as a surrogate for the extensive exploration of the design space. Taken together, these features (universal approximation capability, scalability to large amounts of data, flexible handling of multiple and heterogeneous outputs, and speed at inference time) motivate the choice of neural networks as the surrogate model adopted in this work.

Chapter 4

Artificial Neural Networks

4.1 Deep Learning and Artificial Neural Networks

Over the past few decades, the field of artificial intelligence has undergone a profound transformation driven by the rise of *deep learning*, a subfield of machine learning based on the use of artificial neural networks with multiple layers of data processing. The term "deep" refers to the depth of these architectures, that is the number of successive transformation layers through which data passes before producing an output.

Machine learning, in a broader sense, encompasses a family of techniques that enable a computational system to learn patterns and relationships from data without being explicitly programmed to do so. Within this framework, three main learning paradigms are commonly distinguished:

- **supervised learning**, in which the model is trained on a labeled dataset — i.e., a collection of known input/output pairs — with the objective of learning a mapping function that generalizes to unseen data;
- **unsupervised learning**, in which the model must autonomously discover latent structures in unlabeled data;
- **reinforcement learning**, in which an agent learns to make optimal decisions by interacting with an environment and receiving reward signals.

The problem addressed in this thesis falls naturally within the supervised learning paradigm, as a large set of labeled optimal solutions (each mapping the parameters of a transfer to the corresponding optimal solution) is built for training; the following discussion is therefore restricted to this setting.

Artificial Neural Networks (ANNs) constitute the class of models underlying deep learning. At a high level of abstraction, they draw inspiration from the functioning of biological nervous systems: just as the human brain processes information through a network of interconnected neurons, an ANN is composed of elementary computational units, *artificial neurons*, organized into layers and connected through parameterized weights.

The fundamental principle is that, by composing a sufficient number of simple nonlinear processing units, it is possible to approximate arbitrarily complex functions. This is formally stated by the *Universal Approximation Theorem*, proved by Cybenko in 1989 [28]: a neural network with at least one hidden layer of sufficient width and nonlinear activation functions is capable of approximating any continuous function on a compact subset of $\mathbb{R}^n$, to any desired degree of accuracy. This result provides theoretical justification for the use of neural networks as surrogate models for functional approximation problems, as in the case addressed in this thesis.

In practice, the approximation capacity of a network depends not only on its width, but also on its depth: networks with multiple hidden layers, known as *deep neural networks* (DNNs), tend to require significantly fewer parameters than shallow networks to approximate functions with complex hierarchical structure [30]. The presentation that follows adopts the pedagogical progression of the MIT course *Introduction to Deep Learning* [31]; the primary sources of the individual results are cited where they are introduced.

4.2 The Perceptron

The basic building block of any artificial neural network is the *perceptron*, or artificial neuron, first introduced by Frank Rosenblatt in 1958 [32]. Despite its conceptual simplicity, the perceptron encapsulates the core computational principle upon which all modern deep learning architectures are built.

As illustrated in Figure 4.1, a single perceptron receives a vector of n real valued inputs $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$, each associated with a corresponding scalar weight $w_i \in \mathbb{R}$, which determines how strongly each input contributes to the prediction. The first operation performed by the neuron is a weighted sum of its inputs, to which an additive term called bias $b \in \mathbb{R}$ is added:

$$z = \sum_{i=1}^{n} x_i w_i + b = \mathbf{w}^T \mathbf{x} + b \tag{4.1}$$

The bias term plays a crucial role: it allows the neuron to shift its activation threshold independently of the input values, thereby increasing the expressive flexibility of the model. The quantity z is referred to as the pre-activation of the neuron.

The pre-activation z is then passed through a *nonlinear activation function* $g(\cdot)$, which produces the final output $\hat{y}$ of the perceptron:

$$\hat{y} = g(z) = g\Big(\mathbf{x}^T\mathbf{w} + b\Big) \tag{4.2}$$

The nonlinearity introduced by $g(\cdot)$ is essential: without it, the perceptron would reduce to a purely linear transformation of its inputs. The role of the activation function and the choice among the available options are discussed in Section 4.4.

The learnable parameters of a perceptron are therefore the weight vector $\mathbf{w} \in \mathbb{R}^n$ and the bias b, collectively denoted as $\boldsymbol{\theta} = \{\mathbf{w}, b\}$. The training process, described in Section 4.6, consists in adjusting these parameters iteratively so as to minimize a suitable measure of error between the network predictions and the true target values.

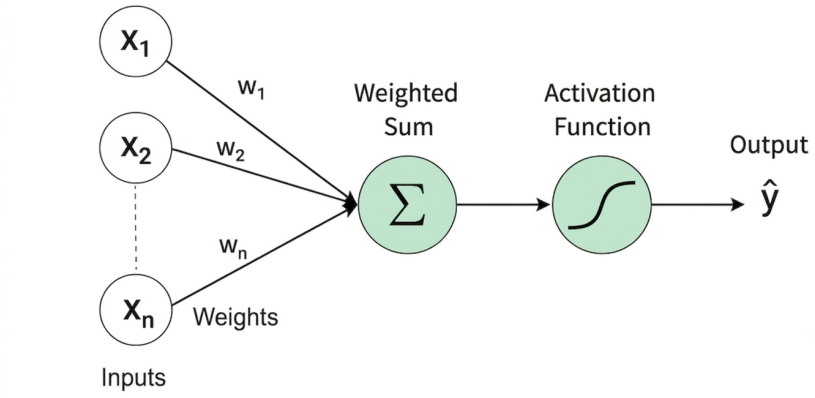


Figure 4.1: Schematic representation of a single perceptron

It is straightforward to extend the formulation described above to the case of *multiple-output perceptron*, in which a layer of k neurons processes the same input vector $\mathbf{x} \in \mathbb{R}^n$ in parallel, as illustrated in Figure 4.2.

In this configuration, each neuron $j \in \{1, \ldots, k\}$ maintains its own weight vector $\mathbf{w}_j \in \mathbb{R}^n$ and bias b_j, and independently computes:

$$z_j = \mathbf{w}_j^T\mathbf{x} + b_j, \qquad y_j = g(z_j) \tag{4.3}$$

This can be expressed compactly in matrix form by stacking the weight vectors as rows of a weight matrix $\mathbf{W} \in \mathbb{R}^{k\times n}$ and the biases as a vector $\mathbf{b} \in \mathbb{R}^k$:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}, \qquad \mathbf{y} = g(\mathbf{z}) \tag{4.4}$$

where $g(\cdot)$ is applied elementwise to the pre-activation vector $\mathbf{z} \in \mathbb{R}^k$, yielding the output vector $\mathbf{y} \in \mathbb{R}^k$. This matrix formulation constitutes the computational

primitive of modern neural network implementations. Crucially, it also represents the natural abstraction of a *fully connected layer*, which is the building block from which deeper architectures are constructed, as discussed in the following section.

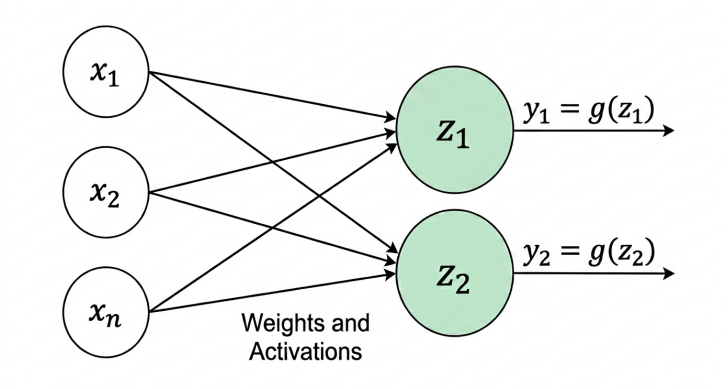


Figure 4.2: Multi-output perceptron with n inputs and $k = 2$ output neurons

4.3 Deep Neural Networks

The multi-output layer formulation introduced in the previous section constitutes the fundamental computational unit from which more complex architectures can be assembled. By stacking multiple such layers in sequence, each feeding its output as the input to the next, one obtains a *feed-forward neural network* — also referred to as a *multilayer perceptron* (MLP). The term *feed-forward* reflects the fact that information flows strictly in one direction, from the input layer through a series of intermediate layers to the output layer, with no feedback connections or cycles.

When the number of intermediate layers is sufficiently large, the network is referred to as a *deep neural network* (DNN), and the field concerned with training such architectures is precisely what is known as deep learning. The depth of the network (i.e., the number of layers) is one of the primary factors governing its capacity: deeper architectures are capable of learning hierarchical representations of the input data, in which successive layers extract features of increasing abstraction. This hierarchical composition of transformations is what makes DNNs particularly effective at approximating complex, high-dimensional functions from data [30].

4.3.1 Architecture

A deep feed-forward neural network is organized into three distinct types of layers, as illustrated in Figure 4.3:

- **Input layer**: the first layer of the network, which receives the raw input vector $\mathbf{x} \in \mathbb{R}^n$, where n denotes the number of input features. This layer performs no computation; it simply passes the input data to the first hidden layer.

- **Hidden layers**: the intermediate layers of the network, indexed $\ell = 1, 2, \ldots, L-1$, which are not directly exposed to either the input or the output. Each hidden layer applies a learned affine transformation followed by a nonlinear activation function to its input, progressively building more abstract internal representations of the data. The number of hidden layers defines the *depth* of the network, while the number of neurons in each layer defines its *width*.

- **Output layer**: the final layer of the network, indexed $\ell = L$, which produces the network's prediction $\hat{\mathbf{y}} \in \mathbb{R}^{n_L}$, where n_L is the number of output neurons. The choice of activation function for this layer depends on the nature of the task: for regression problems, a linear activation is typically employed, so that the output is an unconstrained real valued vector.

Each layer ℓ is characterized by a weight matrix $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ and a bias vector $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$, where n_ℓ denotes the number of neurons in layer ℓ. The forward pass through the network is defined by the following recursive computation, applied for $\ell = 1, 2, \ldots, L$:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \tag{4.5}$$

$$\mathbf{h}^{(\ell)} = g^{(\ell)}\left(\mathbf{z}^{(\ell)}\right) \tag{4.6}$$

where $\mathbf{h}^{(0)} = \mathbf{x}$ is the input vector, $\mathbf{z}^{(\ell)}$ is the pre-activation vector of layer ℓ, $\mathbf{h}^{(\ell)}$ is its post-activation output, and $g^{(\ell)}(\cdot)$ is the activation function applied elementwise at layer ℓ. The final network output is $\hat{\mathbf{y}} = \mathbf{h}^{(L)}$.

4.3.2 Fully Connected Layers

Each layer described above is a fully connected layer (also called a *dense layer*): every neuron in layer ℓ receives as input the full output vector of layer $\ell - 1$, and is connected to every neuron in layer $\ell + 1$. This all-to-all connectivity pattern is what gives rise to the dense weight matrix $\mathbf{W}^{(\ell)}$. The total number of learnable parameters of a fully connected network is:

$$|\boldsymbol{\theta}| = \sum_{\ell=1}^{L} (n_\ell \cdot n_{\ell-1} + n_\ell) \tag{4.7}$$

where the first term accounts for the weights and the second for the biases at each layer. The set of all learnable parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$ is what the training process optimizes, as described in Section 4.6.

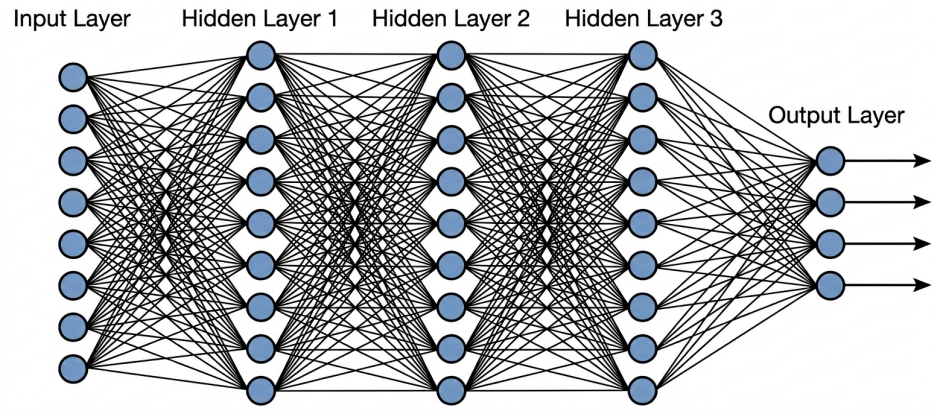


Figure 4.3: Architecture of a deep feed-forward neural network with one input layer, three hidden layers, and one output layer

4.4 Activation Functions

As anticipated in the previous sections, the activation function $g(\cdot)$ is the nonlinear component applied to the output of each neuron following the affine transformation. Its role is fundamental: in the absence of nonlinearity, any composition of linear layers would remain a globally linear transformation, regardless of the network depth, effectively nullifying the expressive advantage of deep architectures. It is the activation function that endows the network with the ability to approximate complex mappings, as established by the Universal Approximation Theorem [28, 29].

The activation function is applied elementwise to the pre-activation vector $\mathbf{z}^{(\ell)}$ of each hidden layer, producing the output $\mathbf{h}^{(\ell)}$. The activation function used in the output layer depends on the type of task being solved. For regression problems, such as the one considered in this thesis, a linear activation function is typically used because it allows the network to predict any real valued output without restricting the range of possible predictions.

Several activation functions have been proposed in the literature, each exhibiting different properties that make it more or less suited to specific tasks and architectural contexts. Among the most popular are:

- **Sigmoid**: maps its input to the interval $(0, 1)$ according to $g(z) = \frac{1}{1+e^{-z}}$. Historically associated with binary classification models, it tends to suffer

from the *vanishing gradient* problem for large input magnitudes, which can slow down training in deep networks.

- **Hyperbolic tangent** (*tanh*): defined as $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, it produces zero-centered outputs in the interval $(-1, 1)$, which is generally preferable to the sigmoid for hidden layers as it mitigates issues related to asymmetric gradient updates.
- **Rectified Linear Unit** (**ReLU**): defined as $g(z) = \max(0, z)$, it is currently the most widely adopted activation function in hidden layers, owing to its computational simplicity, sparsity of activation, and favorable properties during backpropagation.

The choice of activation function is a hyperparameter of the model and can significantly affect both the performance and the stability of the training process.

4.5 Loss Function

Once the network architecture and the forward pass mechanism have been defined, it is necessary to introduce a means of quantifying how far the network predictions deviate from the true target values. This is the role of the **loss function** (or cost function), denoted $\mathcal{L}(\boldsymbol{\theta})$, which maps the set of network parameters $\boldsymbol{\theta}$ to a non-negative scalar: the closer the predictions are to the expected values, the lower the value of the loss.

Formally, given a training dataset of N input/output pairs $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$, the loss function measures the aggregated prediction error of the network over the entire dataset:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell\Big(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\Big) \tag{4.8}$$

where $\ell(\cdot, \cdot)$ is a pointwise error function comparing the prediction $\hat{\mathbf{y}}^{(i)}$ with the corresponding target $\mathbf{y}^{(i)}$. The objective of training is to find the parameters $\boldsymbol{\theta}$ that minimize $\mathcal{L}(\boldsymbol{\theta})$, so as to make the network predictions as accurate as possible. The minimization process is discussed in detail in the following section.

As with activation functions, several loss functions have been proposed in the literature, each better suited to a specific type of problem. For regression tasks, the most commonly used are the *Mean Squared Error* (MSE), which penalizes errors quadratically:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \Big\| \hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)} \Big\|^2 \tag{4.9}$$

and the *Mean Absolute Error* (MAE), which measures the average error in absolute value and is less sensitive to outliers:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}\left\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\right\|_1 \tag{4.10}$$

For classification tasks, the most adopted loss function is the *cross-entropy*, which measures the divergence between the probability distribution predicted by the network and the true class distribution. In the binary case, the network output is passed through a sigmoid activation function so as to produce a probability $\hat{y}^{(i)} \in (0, 1)$, and the *binary cross-entropy* loss is defined as:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y^{(i)}\log\hat{y}^{(i)} + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right] \tag{4.11}$$

The choice of loss function is therefore closely tied to the nature of the problem at hand and represents one of the fundamental design hyperparameters of a neural network.

4.6 Training and Optimization

Training a neural network is the process by which the parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$ are iteratively updated so as to minimize the loss function over the training dataset. An untrained network, whose parameters are randomly initialized, produces essentially arbitrary predictions; it is through training that the network acquires the ability to extract patterns from data and generalize to previously unseen examples.

The training process consists of a cyclic sequence of operations, repeated for a sufficient number of iterations to drive the loss toward a minimum. Each iteration comprises the following steps:

1. **Forward pass**: the input vector $\mathbf{x}^{(i)}$ is propagated through all layers of the network, sequentially applying the affine transformations and activation functions described in the previous sections, until the prediction $\hat{\mathbf{y}}^{(i)}$ is produced at the output layer.

2. **Loss computation**: the prediction $\hat{\mathbf{y}}^{(i)}$ is compared to the target value $\mathbf{y}^{(i)}$ through the loss function $\mathcal{L}(\boldsymbol{\theta})$, yielding a scalar measure of the error incurred by the network.

3. **Backpropagation**: the gradient of the loss with respect to all network parameters, $\nabla_{\boldsymbol{\theta}}\mathcal{L}$, is computed by propagating the error signal backward through the layers via the chain rule of differential calculus. This step is discussed in detail in Section 4.6.2.

4. **Parameter update**: the network parameters are updated in the direction opposite to the gradient, so as to reduce the value of the loss.

These four steps constitute a single update pass, and are repeated for a number of *epochs* (where one epoch corresponds to a complete pass over the entire training dataset) until a stopping criterion is met.

4.6.1 Gradient Descent

The fundamental method for minimizing the loss function is *gradient descent*. The underlying idea is conceptually straightforward: the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ points in the direction of steepest ascent of the loss in the parameter space; by moving in the opposite direction, the parameters are updated toward a local minimum. The update rule is given by:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}}\mathcal{L} \tag{4.12}$$

where $\eta > 0$ is the *learning rate*, a hyperparameter controlling the step size of each update. The choice of learning rate is critical: a value that is too large may cause oscillations or divergence during training, whereas a value that is too small may result in extremely slow convergence or cause the optimization to become trapped in unfavorable local minima.

In its classical formulation, the gradient is computed over the entire training dataset before performing a single parameter update. This approach, known as *batch gradient descent*, is computationally prohibitive for large datasets, as it requires processing all N training samples before taking a single step.

A more practical alternative is *Stochastic Gradient Descent* (SGD), in which the gradient is estimated from a single training sample at a time, updating the parameters after each sample. Although this introduces variance in the gradient estimate, the higher update frequency often leads to faster convergence in practice.

The most widely adopted compromise is *mini-batch gradient descent*: the dataset is partitioned into fixed-size subsets of size $B \ll N$, called *mini-batches*, and the gradient is estimated on each mini-batch. This approach combines the computational efficiency of SGD with a more stable gradient estimate compared to the purely stochastic case, and represents today the standard for training deep neural networks.

4.6.2 Backpropagation

The efficient computation of the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is made possible by the *backpropagation* algorithm [33], which exploits the layered structure of the network and the

chain rule of differential calculus to propagate the error signal from the output layer back toward the input layer [30, 34].

For each layer ℓ, the local error signal (commonly referred to as the *delta*) is defined as the partial derivative of the loss with respect to the pre-activation vector:

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \tag{4.13}$$

Starting from the output layer $\ell = L$, where $\boldsymbol{\delta}^{(L)}$ is computed directly from the chosen loss function and output activation, the error signal is propagated backward recursively through the network:

$$\boldsymbol{\delta}^{(\ell)} = \left(\mathbf{W}^{(\ell+1)}\right)^T \boldsymbol{\delta}^{(\ell+1)} \odot g'^{(\ell)}\left(\mathbf{z}^{(\ell)}\right) \tag{4.14}$$

where $\odot$ denotes the elementwise product and $g'^{(\ell)}$ is the derivative of the activation function at layer ℓ, evaluated at the pre-activation $\mathbf{z}^{(\ell)}$ computed during the forward pass. Once the delta vectors have been obtained for all layers, the gradients with respect to the learnable parameters are given by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \left(\mathbf{h}^{(\ell-1)}\right)^T, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \tag{4.15}$$

These gradients are then used by the gradient descent update rule of Equation 4.12 to adjust the parameters of each layer. The backpropagation algorithm thus provides an efficient and exact method for computing $\nabla_{\boldsymbol{\theta}} \mathcal{L}$, with a computational cost that scales linearly with the number of layers, making it the cornerstone of training in modern deep learning [30].

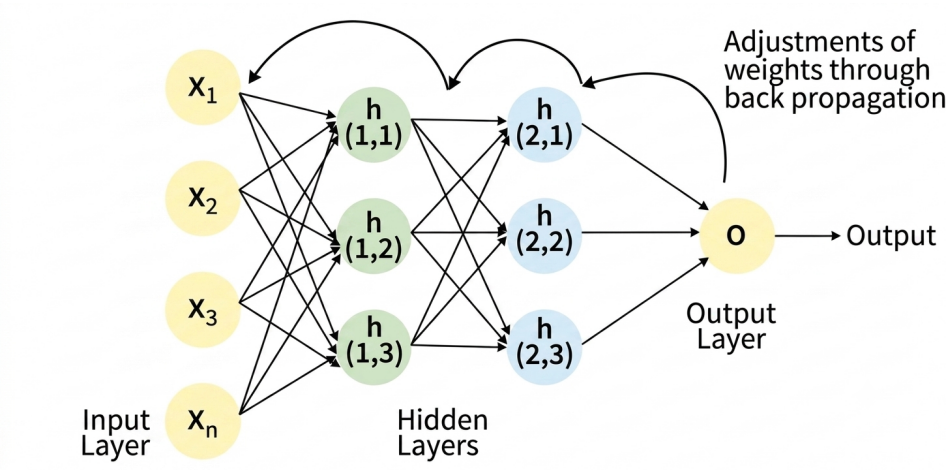


Figure 4.4: Schematic of the backpropagation algorithm

Once the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ has been computed via backpropagation, an optimization algorithm is required to use it for updating the network parameters. Although gradient descent represents the conceptual starting point, in practice more sophisticated algorithms are employed to improve stability and convergence speed. Among the most commonly used are *SGD with momentum*, *Adagrad*, *RMSprop*, *Adadelta* and *Adam* (*Adaptive Moment Estimation*) [35], which combines the advantages of SGD with momentum and RMSprop and is today the most widely adopted optimization algorithm for training deep neural networks [30].

4.7 Overfitting and Regularization

A critical aspect of neural network training is the ability of the model to generalize, that is, to produce accurate predictions not only on the training data, but also on new, previously unseen examples. When a model is too complex relative to the quantity or representativeness of the available data, it tends to memorize the training examples rather than learning the underlying structure of the problem, thereby losing the ability to generalize to new inputs. This phenomenon is known as *overfitting*: the model achieves excellent performance on the training set, but generalizes poorly to new data. The opposite case, known as *underfitting*, occurs when the model is too simple to capture the underlying structure of the data, resulting in poor performance on both the training set and new examples. These two failure modes, along with the ideal intermediate regime, are illustrated in Figure 4.5.

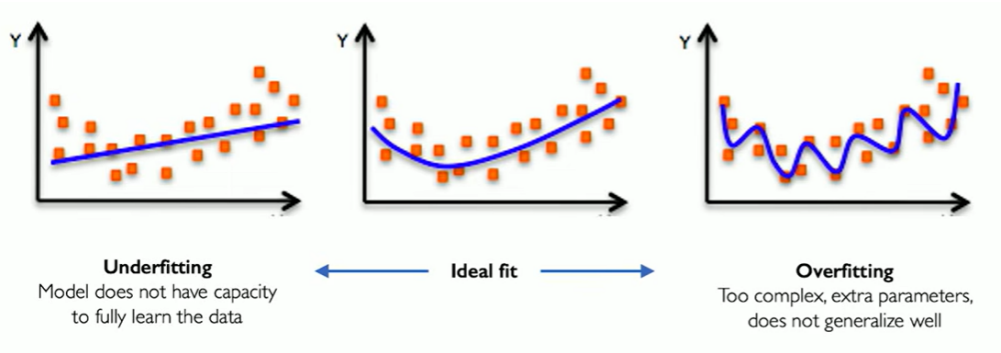


Figure 4.5: Illustration of underfitting, ideal fit, and overfitting. Adapted from © MIT 6.S191: Introduction to Deep Learning [31]

4.7.1 Training, Validation, and Test Set

In order to detect and mitigate overfitting, the available dataset is typically partitioned into three distinct subsets:

- **Training set**: the portion of data used to update the network parameters during training.
- **Validation set**: a subset held separate from training, used to monitor the model performance during training without influencing its parameters. It allows early detection of overfitting and guides the selection of hyperparameters.
- **Test set**: a completely independent set of data, used exclusively during the final evaluation phase to estimate the true performance of the model on examples never seen during training or model selection.

The distinction between the validation and test sets is fundamental: using the test set during the development phase would introduce a form of data leakage, undermining the validity of the final performance estimate.

4.7.2 Regularization Techniques

Regularization encompasses a set of techniques aimed at reducing overfitting and improving the generalization ability of the model without excessively reducing its expressive capacity. Among the most widely used in the context of deep neural networks are *dropout* and *early stopping*.

- **Dropout** [36] consists of randomly deactivating a fraction p of the neurons in a hidden layer (along with their connections) at each training step, where p is a hyperparameter known as the *dropout rate*. By doing so, the network cannot rely on specific units and is forced to learn more distributed and robust representations. During inference, dropout is disabled and the weights are rescaled by a factor $(1 - p)$ to compensate for the larger number of active neurons.
- **Early stopping** monitors the behavior of the loss on the validation set during training and stops the process when it ceases to improve, that is when the validation loss begins to increase while the training loss continues to decrease, as illustrated in Figure 4.6. This divergence point between the two curves signals that the network has begun to overfit the training data. Early stopping is particularly valued for its simplicity and for the fact that it requires no modification to the network architecture.

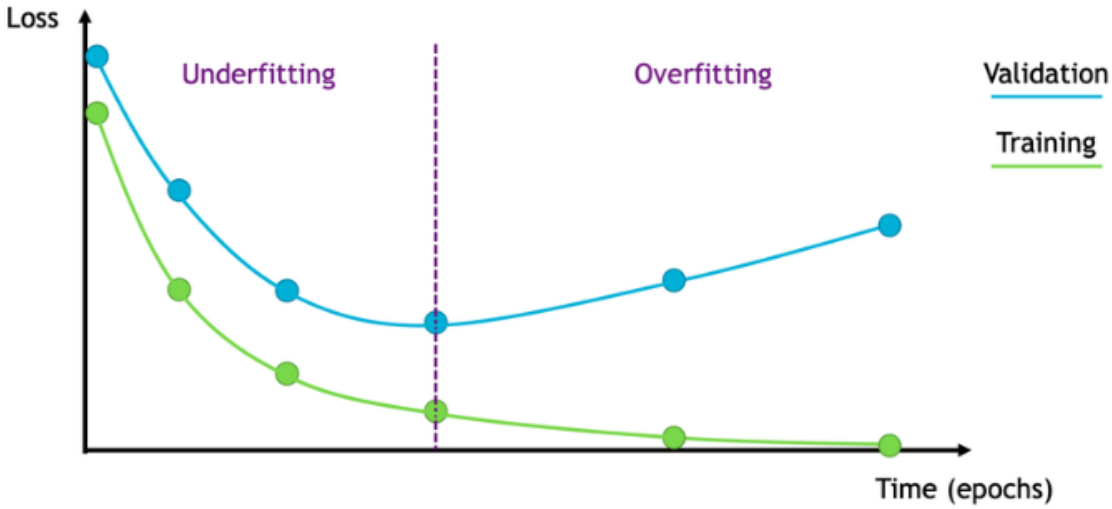


Figure 4.6: Training and validation loss as a function of the number of epochs. Early stopping interrupts training at the point where the two curves begin to diverge

Chapter 5

Dataset Generation

The previous chapters have motivated the adoption of a neural network surrogate model as a means of accelerating the exploration of the design space of low-thrust interplanetary transfers. Such a surrogate, however, is only as effective as the data on which it is trained: the quality, coverage and consistency of the training dataset directly determine the network's ability to generalize. This chapter is devoted to the generation of that dataset.

The aim is to construct a set of optimal Earth–Mars transfers large and varied enough to allow the network to learn the relationship between the conditions of a transfer (departure epoch, time of flight and propulsion system parameters) and its outcome. This outcome is twofold: the network must first determine whether a given transfer is feasible at all, and then, for the feasible ones, predict its cost in terms of propellant. Once trained, the network is able to reproduce this mapping for an arbitrary number of new points at a computational cost that is negligible compared with that of direct optimization, which for every single trajectory requires the solution of a high-dimensional non-linear optimization problem. It is precisely this gap in cost, seconds against fractions of a millisecond per point, that makes the surrogate a useful tool for preliminary mission design.

The generalization capability required of the network is of two kinds. The first is generalization *within* a given mission configuration, that is, the prediction of the cost of transfers not present in the training grid. The second, more ambitious, is generalization *across* different system configurations: the network, once exposed to a set of propulsion configurations during training, should be able to predict the behavior of configurations never seen before, provided their parameters fall within the range covered by the data. To enable and verify this second form of generalization, the dataset was not generated for a single configuration, but for a set of eleven distinct configurations, eight of which are used for training and three held out as a test bench on unseen parameters.

For each configuration the dataset was generated using the same procedure:

the solution of a low-thrust optimization problem over a regular grid of departure epochs and times of flight, yielding a set of optimal trajectories and, for each of them, its associated cost. The final product of this procedure is not only the training data, but also a reference *porkchop plot* for every configuration, generated entirely by the optimizer. These porkchop plots provide a useful means of assessing graphically whether the network has captured the underlying physics of the problem, by comparing them with those produced by the trained surrogate.

5.1 Mission Configurations

The literature offers very few references to Earth–Mars missions actually flown with low-thrust propulsion: this is a transfer regime that, although extensively studied at the theoretical and preliminary design level, does not yet possess a body of real flight data sufficient to define a set of reference configurations. In the absence of such data, the configurations used in this work were constructed by assembling realistic components: for each one an electric thruster actually available on the market was taken, and a spacecraft initial mass consistent with the class of that thruster and with the type of mission was associated with it. The selected thrusters span a wide range of technologies and performance levels, including the BHT-1500 [37], NEXT-C [38], NSTAR, T6, SPT-140, RIT-2X [39] and AEPS [40] units in single and multiple configurations. The resulting set of configurations therefore covers a broad spectrum, ranging from small, very low-thrust satellites, through medium-sized vehicles, up to large spacecraft with markedly higher thrust. Two of the configurations are moreover inspired, in the order of magnitude of their parameters, by real interplanetary missions such as ExoMars and BepiColombo. The complete set of eleven configurations, with their respective specifications, is reported in Table 5.1.

The dynamically relevant quantity is not the initial mass in itself, but the initial thrust acceleration $a_0 = T_{\max}/m_0$, together with the effective exhaust velocity $c = g_0 I_{sp}$. It is these two parameters that govern the shape of the optimal trajectory: two vehicles with identical a_0 and c, but different initial masses, follow geometrically identical transfers and require the same Δv (within the two-body model adopted here), while the propellant mass consumed simply scales with m_0. For this reason a_0 and c — and not the catalog parameters taken individually — will be the quantities chosen as input features of the network. This choice is well illustrated by configurations cfg4 and cfg8, which mount the same thruster with identical $T_{\max}$ and I_{sp} but very different masses: for the same c, the different a_0 makes them two dynamically distinct cases.

Of the eleven configurations, eight are intended for training and three are held out to assess the network's ability to generalize to unseen propulsion parameters.

The split was constructed so that the three held-out configurations lie *inside* the range covered by the training data in the space of the relevant features: their values of a_0 and c both fall within the ranges of the eight training configurations. The test bench therefore verifies the network's ability to *interpolate* within the physical space it has actually learned (the form of generalization of interest for preliminary mission design) rather than a more fragile extrapolation beyond the boundaries of the data.

For each configuration, 2275 optimal trajectories were computed according to the procedure described in the following sections. The set of eight training configurations therefore constitutes a *training dataset* of $2275 \times 8 = 18\,200$ points, while the complete set of eleven configurations amounts to $2275 \times 11 = 25\,025$ trajectories.

	Thruster	m_0 **[kg]**	$T_{\max}$ **[N]**	I_{sp} **[s]**	a_0 **[m/s^2]**	c **[km/s]**
cfg0	BHT-1500	700	0.100	1700	1.43×10^{-4}	16.67
cfg1	NEXT-C	1200	0.236	4155	1.97×10^{-4}	40.75
cfg2	NSTAR (th.)	600	0.080	2800	1.33×10^{-4}	27.46
cfg3	T6	3600	0.360	3900	1.00×10^{-4}	38.25
cfg4	NSTAR	1240	0.092	3200	7.42×10^{-5}	31.38
cfg5	SPT-140	2600	0.280	1800	1.08×10^{-4}	17.65
cfg6	RIT-2X	600	0.100	3500	1.67×10^{-4}	34.32
cfg7	AEPS $\times 3$	3000	1.200	2600	4.00×10^{-4}	25.50
cfg8	NSTAR	480	0.092	3200	1.92×10^{-4}	31.38
cfg9	AEPS $\times 2$	2000	0.600	2600	3.00×10^{-4}	25.50
cfg10	NEXT-C (th.)	650	0.090	2400	1.38×10^{-4}	23.54

Table 5.1: Mission configurations used for dataset generation. Thruster parameters are representative operating points drawn from published performance data [19, 38] and manufacturer datasheets [37]; throttled configurations (th.) adopt a reduced operating point within the documented throttle range. The first eight configurations are used for training; the last three are held out to test generalization to unseen propulsion parameters

5.2 Trajectory Model

The optimization problem associated with each grid point was formulated using the `pykep` library [24], through the `trajopt.direct_pl2pl` class. This implements the direct Sims–Flanagan transcription of a planet-to-planet transfer, whose theoretical formulation was presented in Chapter 3; here the focus is solely on its practical use

and on the choice of parameters. As already explained, the trajectory is divided into a finite number of segments, on each of which the continuous thrust arc is reduced to an impulsive velocity change, and continuity between the leg propagated forward from departure and the one propagated backward from arrival is enforced at a junction point (*match point*).

The transfer is defined between Earth and Mars. The vehicle parameters (initial mass, maximum thrust and specific impulse) are fixed, for each of the points of a given configuration, to the values of that configuration reported in Table 5.1. The departure epochs and times of flight, by contrast, are not fixed: they are defined as decision variables within a narrow window around the grid node, so as to leave the optimizer a local margin in the search for the minimum cost solution. The choice of the remaining arguments deserves a more detailed discussion.

Number of segments (`nseg`). The trajectory is discretized into a finite number of segments, on each of which the continuous thrust arc is reduced to a single velocity impulse. The magnitude of this impulse is proportional to the thrust acceleration and to the segment duration: for a given number of segments, a higher acceleration therefore concentrates a larger velocity change into each discrete impulse, and the piecewise approximation of the continuous thrust becomes correspondingly coarser. The adequate number of segments thus depends on the thrust level of the configuration.

For the low and medium-thrust configurations, twenty segments (`nseg = 20`) represent a balanced choice: high enough to represent faithfully the thrust profiles typical of these transfers, while low enough to keep the dimension of the optimization problem, and hence the computational cost per point, limited. For the two highest-thrust configurations, cfg7 and cfg9, this same discretization proved too coarse: the velocity change per segment becomes large enough that twenty impulses can no longer represent the continuous thrust with sufficient accuracy, and the match-point constraints struggle to be satisfied within tolerance even where a feasible trajectory exists. For these two configurations the number of segments was therefore doubled to `nseg = 40`, refining the discretization so as to restore an accurate representation of the higher-thrust arcs. The role of this choice in ensuring convergence is discussed in Section 5.3.1.

Zero excess velocities (`vinf_dep = vinf_arr = 0`). The hyperbolic excess velocity is set to zero at both departure and arrival. The vehicle is therefore assumed to start on the heliocentric orbit of the Earth, with its same velocity, and the transfer ends with a complete *rendezvous* — in position and velocity — with the orbit of Mars. This is a standard first-order assumption for preliminary mission design: it neglects the contribution of the launch (C_3) and that of any approach

phase at arrival, isolating the cost of the heliocentric low-thrust transfer alone, which is the quantity of interest for this work.

Low fidelity (`hf = False`). The model is used in its low-fidelity version: the thrust on each segment is approximated as a single velocity impulse applied to the segment, with the arcs between one impulse and the next flown as Keplerian arcs. This too is an approximation consistent with the preliminary design phase: in exchange for a limited loss of accuracy with respect to the high-fidelity propagation of the thrust arc, it appreciably reduces the cost of evaluating the problem, that is a decisive aspect when several thousand trajectories must be solved per configuration.

The resulting decision vector has dimension 69 for the 20-segment configurations and 129 for the 40-segment configurations: the departure and arrival epochs, the final mass, the three components of the excess velocity at departure and the three at arrival (the latter six constrained to zero by the rendezvous condition) and the three components of the thrust unit vector for each of the segments. The objective of the optimization is the maximization of the final mass, equivalent to the minimization of the propellant consumed and hence of the transfer Δv.

The problem thus defined (*user defined problem*) was converted into a numerical optimization problem by means of the `PyGMO` library [24], which provides its interface to the solver algorithm. Continuity at the match point is imposed as a set of equality constraints; a solution is considered valid when the maximum violation of these constraints falls below the tolerance $c_{tol} = 10^{-4}$. This threshold defines, at the optimizer level, the boundary between a feasible trajectory and a failure in the search for the solution.

5.3 Optimization Strategy

The solution of the problem at each grid node is entrusted to the combination of two algorithms of complementary nature: a local gradient-based optimizer, SNOPT7, wrapped within a stochastic global search strategy, Monotonic-Basin-Hopping (MBH).

SNOPT7. SNOPT (*Sparse Nonlinear OPTimizer*) is a solver for large-scale constrained nonlinear optimization problems, based on the sequential quadratic programming (SQP) method, that is an iterative approach that solves a nonlinear constrained problem as a sequence of quadratic subproblems, each approximating the original one around the current iterate. Accordingly, at each iteration SNOPT builds a quadratic approximation of the Lagrangian function and a linear approximation of the constraints, solves the resulting quadratic subproblem to

determine a search direction, and advances along that direction by reducing a merit function that balances the minimization of the objective and the satisfaction of the constraints. The algorithm exploits the sparse structure of the constraint Jacobian, a feature that makes it particularly efficient on problems such as the one at hand, in which each segment directly influences only a small number of constraints. SNOPT is a *local* optimizer: starting from an initial point, it converges towards the minimum of the basin of attraction in which that point falls, with no guarantee that this coincides with the global minimum. The quality of the solution it returns therefore depends critically on the quality of the starting point.

Monotonic-Basin-Hopping. MBH is a global-search metaheuristic conceived precisely to overcome this limitation. It does not replace the local optimizer, but wraps it, employing it repeatedly: starting from a local solution, it perturbs its decision vector and restarts the local optimizer from the perturbed point; if the new solution improves on the previous one, it becomes the new reference, otherwise it is discarded and the incumbent is perturbed again. By iterating this procedure, the algorithm "hops" from one basin of attraction to another, progressively exploring the landscape of solutions in search of better optima. This mechanism is illustrated in Figure 5.1. Its configuration is governed by three elements: the inner local algorithm, which in our case is SNOPT7; the magnitude of the perturbation applied to the decision vector at each hop, which regulates the trade-off between local and global exploration [41]; and the stopping criterion, expressed as the number of consecutive iterations without improvement of the incumbent, beyond which the search is considered concluded.

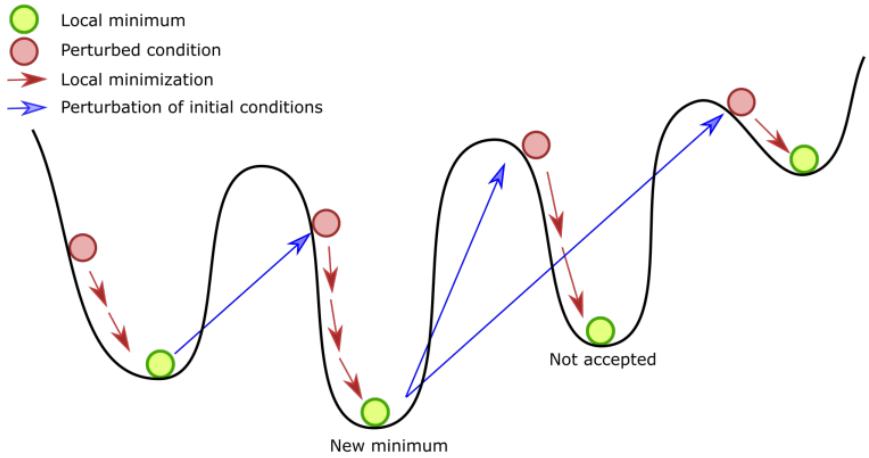


Figure 5.1: Schematic of the Monotonic-Basin-Hopping procedure

Combination of the two algorithms. The choice to combine the two algorithms comes from the nature of the problem. The optimization of low-thrust trajectories gives rise to a strongly multi-modal landscape [17]: the solution space presents numerous local minima, corresponding to qualitatively different thrust profiles (for instance with a different number of revolutions or a different arrangement of the thrust arcs), separated by regions of infeasibility. On a landscape of this kind, applying SNOPT alone from a single initial point would lead, in most cases, to an arbitrary local minimum, with a high probability of missing the minimum-cost solution or, worse, of failing to find any feasible solution even when one exists. By wrapping SNOPT within MBH, the ability of the former to converge quickly and accurately within a basin is combined with the ability of the latter to explore different basins: the result is a procedure that preserves the efficiency of local optimization but mitigates its dependence on the starting point, appreciably increasing the probability of identifying the optimal trajectory, or a good approximation of it, for each node of the grid.

5.3.1 Tier Escalation

The effectiveness of MBH depends, besides the perturbation and the stopping criterion already described, on a third element introduced by the PyGMO interface: the *population*. In PyGMO an algorithm does not operate on a single point, but on a set of candidate solutions (the population) each accompanied by its own fitness value. The best solution in the set constitutes the *champion*, which MBH takes as the initial incumbent to be perturbed. The size of the population therefore determines the richness of the initial sampling of the decision space: a larger population generates a greater number of random starting points, increasing the probability that at least one falls within a favorable basin of attraction.

Even with the SNOPT–MBH combination, the stochastic nature of the search means that a single attempt may fail despite the existence of a feasible solution: the algorithm may not have explored the correct basin within the allowed number of iterations, returning a *false negative*. For a dataset intended for the training of a neural network surrogate this is a serious problem: a point erroneously classified as infeasible distorts the boundary between the feasible and infeasible regions that the network must learn. It is therefore essential to make it as unlikely as possible that a failure reflects a mere lack of convergence of the optimizer rather than a true absence of solution.

For this reason the solution at each node does not rely on a single attempt, but on an *escalation* strategy organized into three levels of increasing intensity. Each level is defined by four elements: the number of MBH iterations, the magnitude of the perturbation, the population size and the number of independent *seeds*, that is, of repeated runs with a different seed of the random number generator. The logic is

as follows: one starts from the least expensive level; if it returns a feasible solution, this is accepted and the search on the node concludes. Otherwise one moves to the next level, more costly but more thorough, up to the third level, which repeats the search from several independent seeds in order to neutralize the statistical bad luck of a single start. Only after all levels have been exhausted without success a node is declared a failure.

This strategy is quite expensive: a node is declared a failure only after the search, repeated and progressively intensified, has been carried out in full without returning a solution. The computational cost concentrates precisely on the nodes for which no solution is found: whenever a node is solved, the search stops at the level that returns the solution, whereas when no solution is found the algorithm is forced to traverse all three levels in full (all the iterations, all the seeds) before it can conclude. The regions where the optimizer fails to find a solution are therefore the most expensive to map.

The parameters of the three levels are reported in Table 5.2. For most configurations the *standard* strategy was employed. The two highest-thrust configurations, cfg7 and cfg9, were instead treated differently in two respects: the finer discretization already introduced in Section 5.2 (`nseg = 40`) and a more aggressive escalation strategy. The two changes address two distinct problems, and are discussed together below.

Higher-thrust configurations. The two highest-thrust configurations required a different treatment for two distinct reasons, one concerning the feasibility of the solution and the other its quality.

The first, and decisive for convergence, is the discretization of the trajectory. As explained in Section 5.2, the velocity impulse assigned to each Sims–Flanagan segment grows with the thrust acceleration; at the thrust levels of cfg7 and cfg9 the twenty-segment discretization used for the other configurations becomes too coarse, and the match-point constraints fail to be satisfied within tolerance even where a feasible trajectory exists. It was this coarseness, and not any intrinsic difficulty of the high-thrust regime, that caused the optimizer to fail at nodes where a solution was in fact present. Doubling the number of segments to forty removes the cause of these failures, restoring an accurate representation of the higher-thrust arcs; the resulting increase in convergence is the principal effect, and is reflected in the near complete feasible regions of cfg7 and cfg9 (Section 5.7).

The second concerns the quality of the solutions rather than their existence. A higher thrust acceleration grants the vehicle greater control authority, and with it a larger number of qualitatively different feasible transfers (with a different number of revolutions, or a different arrangement of thrust and coast arcs). The optimization landscape is correspondingly richer and more multi-modal, populated by a greater number of local minima, so that the search for the global optimum,

rather than a merely local one, becomes more delicate. To increase the probability of locating the global optimum on such a landscape, a more aggressive escalation strategy was adopted for these two configurations (Table 5.2): more iterations, larger perturbations, more numerous populations and, above all, a considerably greater number of independent seeds at the last level. This strategy is probably more meticulous than strictly necessary, but the additional computational cost is a deliberate, conservative choice aimed at maximizing the quality of the dataset for the two configurations whose solution space is hardest to explore exhaustively.

Level	MBH iterations	Perturbation	Population	Seeds
Standard (cfg0–cfg6, cfg8, cfg10) – `nseg=20`				
normal	6	0.05	10	1
strong	10	0.05	20	1
multi-seed	10	0.08	20	3
Aggressive (cfg7, cfg9) – `nseg=40`				
normal	10	0.05	15	1
strong	25	0.07	25	2
multi-seed	30	0.12	25	8

Table 5.2: Parameters of the three-level escalation strategy

5.3.2 Warm-Start Chaining

The escalation strategy described in the previous section improves the reliability of the search, at a computational cost that grows the more it is intensified. Part of this cost can be recovered by exploiting a structural property of the dataset: the regularity of the $t_0 \times t_{of}$ grid. Two adjacent nodes along the time of flight axis share the same departure epoch and differ only by a small increment in the time of flight; within the two-body model adopted here, their optimal trajectories are consequently very similar, and the decision vector that solves one constitutes an excellent starting point for the other.

This observation is the basis of the *warm-start chaining* technique. Instead of tackling each node from an entirely random population, the solution of the previous node along the row is added to the population for the next one. Concretely, each row of the grid (corresponding to a fixed departure epoch) is traversed in sequence as the time of flight increases: whenever a node converges, its decision vector is stored and supplied as a *warm start* to the following node, where, as anticipated in Section 5.3.1, it is inserted into the population as an additional individual alongside

the randomly generated ones. If a node does not converge, the last valid solution available along the row continues to serve as the starting point for the subsequent nodes, until a new one is found.

The advantage is twofold. On the one hand, the computational cost is reduced: starting from a point already close to the optimum, the local optimizer converges in fewer iterations, and typically already at the first level of the escalation, without having to resort to the more expensive levels. On the other hand, the quality and consistency of the solutions are improved: by anchoring each node to the solution of the previous one, the continuity of the family of trajectories along the row is favored, reducing the risk that contiguous nodes converge to qualitatively different local minima and produce an artificially discontinuous cost map.

The chaining is deliberately confined within the single row, that is, at increasing time of flight and constant departure epoch. The different rows, by contrast, remain independent of one another: this independence, besides preventing the propagation of a possibly poor-quality solution across the entire grid, is also what allows the dataset generation to be parallelized by assigning distinct rows to distinct processes, as described in Section 5.6.

5.4 Grid Design

For each configuration the dataset is generated by sampling the (t_0, t_{of}) plane on a regular grid of $65 \times 35 = 2275$ nodes: 65 departure epochs and 35 times of flight, uniformly spaced. The choice of the extremes of the two axes responds to distinct criteria.

Departure epoch. The departure-epoch axis is identical for all configurations: 65 nodes uniformly distributed over a window of 1742 days, from $t_0 = 10958.0$ to $t_0 = 12700.0$ MJD2000, with a step of about 27 days. The width of this window arises from a trade-off. On the one hand, the aim was to cover at least two Earth–Mars synodic periods — the synodic period being about 780 days — so that the resulting porkchop plot displays at least two complete favorable launch windows and makes their periodicity recognizable. On the other hand, extending the interval further was avoided, since each additional departure epoch entails the solution of an entire row of optimization problems, with the resulting computational cost. The adopted window represents the point of balance between these two requirements: sufficient to visualize the periodic structure of the porkchop, without unnecessarily burdening the generation.

Time of flight. The time of flight axis, by contrast, was adapted to each configuration. All configurations share the same interval width — 700 days divided

into 35 nodes, with a step of about 20 days — but its location varies from one configuration to another. The reason is that the feasibility region in the (t_0, t_{of}) plane, and in particular its *lower edge* or *apex* (the minimum time of flight for which a feasible transfer exists at a favorable launch window) lies at different times of flight depending on the propulsive capability of the vehicle. A configuration with low thrust acceleration has weak thrust relative to its own mass and requires long times of flight to accumulate the impulse needed to close the transfer; its feasibility region, and the corresponding lower edge, therefore lie at high values of t_{of}. A configuration with high acceleration, conversely, closes the transfer in shorter times, and its apex lies at much lower times of flight. This behavior is clearly visible in the values reported in Table 5.3: one moves from the 150–850 days of the highest-acceleration configuration (cfg7) to the 900–1600 days of the lowest-acceleration one (cfg4).

The time of flight interval of each configuration was therefore centered so as to frame the lower edge of its respective feasibility region, so that the boundary between the feasible and infeasible regions is clearly visible and sharply defined within the sampled domain. This choice serves the objective of the chapter: the feasibility boundary is the structure that the network must learn with the greatest accuracy, and it is therefore essential that the dataset sample it adequately for each configuration.

	Thruster	a_0 **[m/s^2]**	t_{of} **[days]**
cfg0	BHT-1500	1.43×10^{-4}	350 – 1050
cfg1	NEXT-C	1.97×10^{-4}	300 – 1000
cfg2	NSTAR (throttled)	1.33×10^{-4}	400 – 1100
cfg3	T6	1.00×10^{-4}	550 – 1250
cfg4	NSTAR	7.42×10^{-5}	900 – 1600
cfg5	SPT-140	1.08×10^{-4}	450 – 1150
cfg6	RIT-2X	1.67×10^{-4}	350 – 1050
cfg7	AEPS ×3	4.00×10^{-4}	150 – 850
cfg8	NSTAR	1.92×10^{-4}	300 – 1000
cfg9	AEPS ×2	3.00×10^{-4}	200 – 900
cfg10	NEXT-C (throttled)	1.38×10^{-4}	350 – 1050

Table 5.3: Time of flight ranges adopted for each configuration

5.5 Feasibility Definition and Cost Threshold

The notion of feasibility adopted in this work is the result of two distinct levels of filtering: the first, purely geometric-dynamic, imposed at the optimizer level; the second, of an economic nature, applied a posteriori. Their separation is a deliberate methodological choice.

Geometric-dynamic feasibility. At the optimizer level, a trajectory is feasible when there exists, within the prescribed tolerances, a solution that satisfies continuity at the match point: a transfer physically realizable by the vehicle with the available thrust. At this level no constraint is imposed on propellant availability. Consistently, the dry mass of the vehicle was set to zero in all configurations: in this way the optimizer never discards a solution because of propellant depletion, but is free to identify, for each grid node, the geometrically realizable maximum-final-mass transfer, whatever its cost. The aim is not to lose information: one wants to know the cost of *every* physically possible transfer, even the very expensive ones, and to defer the judgment on their suitability to a later stage.

Cost threshold. That judgment is entrusted to a second filter, applied a posteriori to the results of the optimization: a threshold on the cost of the transfer, expressed as Δv. Trajectories that converge but require a Δv greater than $\Delta v_{cutoff} = 9000$ m/s are considered infeasible from a practical standpoint. The value of the threshold was set so as to represent an excessive cost for an Earth–Mars transfer: transfers exceeding this limit, although geometrically realizable, require a propellant mass so large as to be of little practical interest for a real mission. This setup reflects the logic of preliminary design: the optimizer establishes *how much* a given transfer costs, while the threshold translates that cost into a judgment of practical feasibility, which a mission analyst may possibly re-tune according to the specific constraints of their own mission.

The joint application of the two filters defines the physical-feasibility label associated with each grid point: a point is physically feasible when the optimizer converges to a solution *and* the corresponding Δv does not exceed the threshold. This is the label on which the learning of the surrogate is based.

Effect of the threshold on the feasibility boundary. The choice of a threshold on Δv also has a beneficial effect on the structure of the dataset, which is directly linked to the network's ability to learn it. As the geometric feasibility boundary is approached (that is, the minimum times of flight and the least favorable departure epochs) the Δv required by the transfer grows very rapidly, tending to extremely high values in the proximity of the boundary itself. Without the threshold, the dataset would contain a tail of extreme Δv transfers, concentrated in a narrow band

adjacent to the boundary, and the transition between the feasible and infeasible regions would be blurred and lacking a sharp demarcation. The threshold acts precisely on this band: by cutting the extreme-cost transfers, it replaces a gradual and ill-defined transition with a sharp boundary, placed at a well-defined value of Δv. This benefits both components of the surrogate. The classifier must learn a more definite, and therefore more easily separable, feasible/infeasible boundary. The regressor, trained on the feasible transfers alone, works on a limited and better conditioned Δv range, free of the tail of extreme values which would interfere with its training.

5.6 Parallelization

The generation of the dataset requires the solution of 2275 optimization problems per configuration, for a total of 25 025 trajectories across the eleven configurations. Each solution, owing to the escalation strategy described in section 5.3.1, may require numerous invocations of the local optimizer, and the overall cost would be prohibitive if the nodes were solved one at a time. The generation was therefore parallelized.

The unit of parallelization is the single row of the grid, that is, the set of nodes at a fixed departure epoch and increasing time of flight. This choice follows directly from the structure introduced in the previous sections: the warm-start chaining links the nodes of a given row in a sequential chain that cannot be broken, whereas the rows are independent of one another. The row is therefore the smallest portion of work that can be executed autonomously, while keeping the chaining within it intact: each row is solved sequentially by a single process, and different rows are solved in parallel by different processes.

The computation was distributed over 46 worker processes, executed in parallel on as many cores. The 65 rows of the grid are assigned to the workers through a dynamic queue: each worker receives a row, solves it in full and, as soon as it has finished, takes another from those still to be processed, until the queue is empty. This dynamic load distribution (rather than a static, predefined assignment of rows to workers) is preferable because the cost of a row is not known a priori and varies appreciably from row to row: a worker that quickly completes an “easy” row does not remain idle, but immediately moves on to the next one. To avoid the oversubscription of the cores (the competition between workers for computing resources that would degrade performance) the number of internal threads of the linear algebra libraries was limited to one per worker, so that each process occupies exactly one core.

It is important to stress that parallelization over 46 cores does not entail a reduction of the computation time by a factor of exactly 46. Several factors combine

to produce a speed up lower than the ideal one. In the first place, the number of rows (65) is not a multiple of the number of workers (46): the generation in effect proceeds in two waves, a first one that engages all 46 workers and a second that occupies only the remaining 19, leaving the other cores idle for the entire duration of the last row processed. Secondly, and more substantially, the cost of the rows is not uniform: as observed in Section 5.3.1, the rows that cross infeasible regions are far more expensive, since the algorithm must exhaust all levels of the escalation before concluding. Because the total time is determined by the slowest row of each wave, and not by the average time, the imbalance of the load further reduces the efficiency with respect to the ideal case. To these is added a fixed start up cost, due to the initialization of the environment and the import of the optimization libraries in each process. The effective speed-up nonetheless remains largely sufficient to make the generation of the entire dataset tractable, reducing the times from weeks to about 40 hours.

5.7 The Resulting Dataset

The application of the procedure described in this chapter to all eleven configurations produces the *complete dataset* of 25 025 points (2275 per configuration), each accompanied by the parameters of the transfer, the outcome of the optimization and, for the feasible points, the corresponding Δv cost. Table 5.4 summarizes its main statistics, configuration by configuration.

The most relevant indicator for the purposes of training is the fraction of physically feasible points, which lies between 56% and 71% across all configurations. This value is the result of the tuning of the time of flight window described in Section 5.4: having centered each interval on the lower edge of the respective feasibility region (and thus having each grid start a comparable amount before the minimum feasible time of flight) the fraction of feasible points turns out to be homogeneous across the configurations, despite their very different propulsive parameters. This homogeneity is in itself a favorable factor, as it prevents the surrogate from being exposed to configurations with markedly different feasibility distributions, which would unbalance its learning.

The level around which this fraction settles also represents a good compromise for learning. A feasibility fraction that is too high, close to the totality of the points, would provide the classifier with very few examples of the infeasible class, preventing it from accurately learning the boundary between the two regions; a fraction that is too low, conversely, would leave the regressor with an insufficient number of feasible transfers on which to train. The balance obtained, with a majority of feasible points alongside a non negligible minority of infeasible ones, simultaneously provides an adequate number of examples to both components of the surrogate:

enough infeasible points for the classifier to capture the feasibility boundary, and enough feasible points for the regressor to learn the cost map accurately.

	a_0 [$\mathbf{m/s^2}$]	c [**km/s**]	**feas** [%]	$\overline{\Delta v}$	Δv_{min}	Δv_{max}
cfg0	1.43×10^{-4}	16.67	60.1	6098	5656	8988
cfg1	1.97×10^{-4}	40.75	59.0	6066	5636	8993
cfg2	1.33×10^{-4}	27.46	61.1	6084	5655	8992
cfg3	1.00×10^{-4}	38.25	59.4	6152	5658	8998
cfg4	7.42×10^{-5}	31.38	65.4	6274	5669	8998
cfg5	1.08×10^{-4}	17.65	56.0	6214	5659	8987
cfg6	1.67×10^{-4}	34.32	63.4	6037	5647	8994
cfg7	4.00×10^{-4}	25.50	66.6	6124	5641	8997
cfg8	1.92×10^{-4}	31.38	61.3	6067	5639	8989
cfg9	3.00×10^{-4}	25.50	70.5	6074	5631	8983
cfg10	1.38×10^{-4}	23.54	57.7	6116	5659	8981

Table 5.4: Summary statistics of the generated dataset, by configuration. The feasibility fraction (feas.) is the percentage of physically feasible points; the Δv values (mean, minimum and maximum, in m/s) refer to the feasible points only

The two configurations with the highest acceleration, cfg7 and cfg9, exhibit the highest feasibility fractions of the whole set (66.6% and 70.5% respectively). This is consistent with the specific treatment applied to these two configurations (Section 5.3.1): the finer discretization (`nseg = 40`) and the more aggressive escalation strategy together reduce the incidence of false negatives, that is, of genuinely feasible points erroneously classified as infeasible because the optimizer failed to converge. The finer discretization is the principal factor, removing the systematic convergence failures caused by the coarse twenty-segment model, while the more thorough search further reduces the residual cases. For the configurations solved with the standard treatment, by contrast, the presence of some residual false negatives cannot be excluded.

A qualitative confirmation of this behavior is obtained from the porkchop plots generated from the dataset, reported in Figure 5.2. Each plot shows, in the (t_0, t_{of}) plane, the Δv cost of the transfer through a color scale, with the infeasible points overlaid: the red crosses mark the nodes at which the optimizer found no solution (*snopt_failure*), while the yellow circles mark the nodes that converged but with a Δv exceeding the threshold (*non-physical*, $\Delta v > 9000$ m/s). The latter, consistently with what was discussed in Section 5.5, arrange themselves in a narrow band along the edge of the feasible region, where the Δv grows rapidly: they show, in the plot, where the threshold has cut the most expensive transfers, and do not represent an

error.

The visible trace of the false negatives is of a different nature, and is to be sought in the low-Δv regions, the dark-colored ones, where a feasible transfer certainly exists. In the porkchop plots of the two high-thrust configurations, cfg7 and cfg9, these regions are entirely filled: no red crosses appear in them, a sign that the optimizer found a solution throughout these regions. This confirms the effectiveness of the treatment applied to these two configurations (Section 5.3.1): with the original twenty-segment model and the standard escalation strategy, the porkchop plots of these same two configurations were dotted with red crosses (*snopt_failure*) even within the dark-colored, low-Δv band, where a solution is known to exist (Figure 5.3) — a clear manifestation of false negatives, caused primarily by the coarse discretization and practically eliminated by the combination of the finer discretization and the more thorough search. In the porkchop plots of the remaining configurations, solved with the standard treatment, sporadic red crosses are observed instead within the low-Δv regions (isolated nodes declared infeasible in the midst of a clearly feasible neighborhood) which constitute the visible manifestation of the residual false negatives.

These reference porkchop plots, generated entirely by the optimizer, constitute a benchmark against which those produced by the surrogate will be assessed: their comparison provides a direct, graphical measure of how well the neural network has captured the physics of the problem.

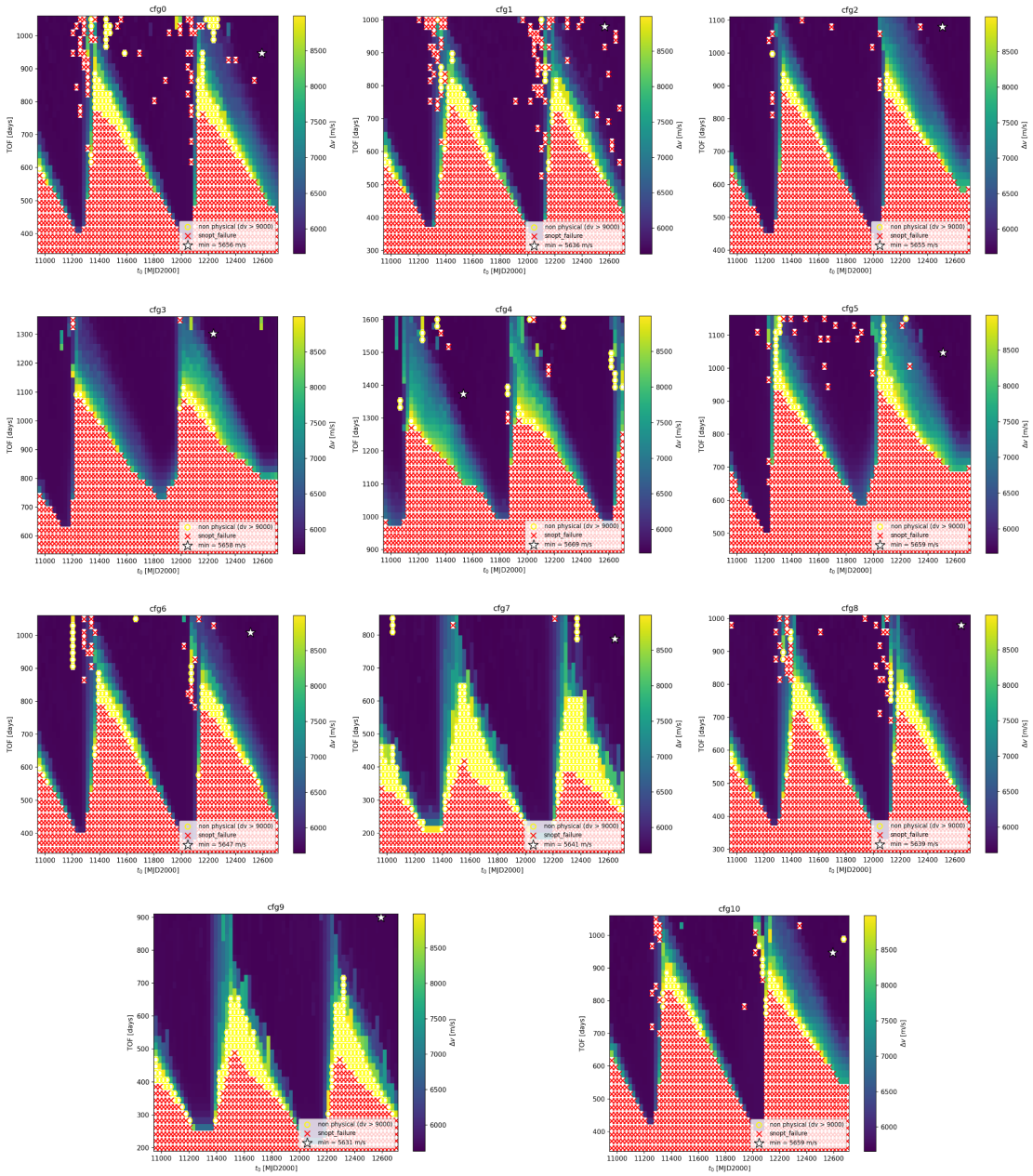


Figure 5.2: Reference porkchop plots generated from the dataset, in the (t_0, t_{of}) plane, one per configuration (cfg0 to cfg10)

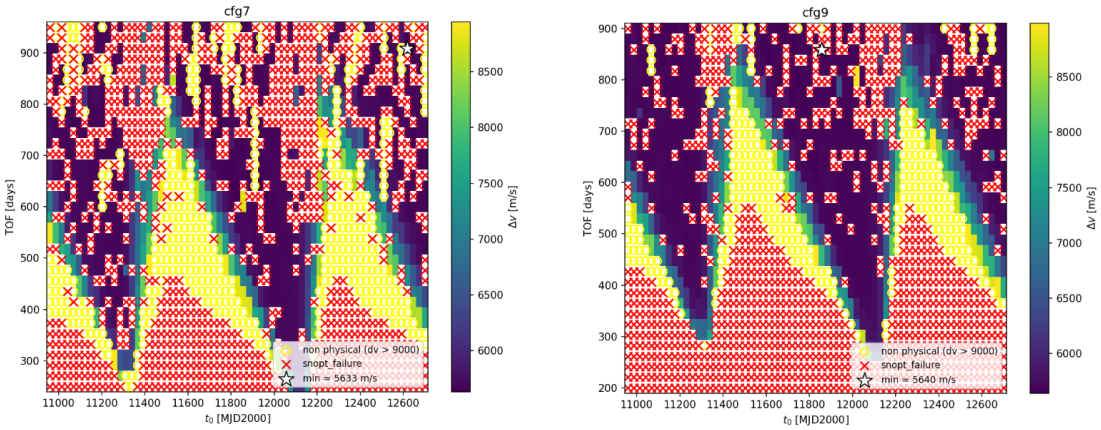


Figure 5.3: Porkchop plots of cfg7 and cfg9 with the standard escalation strategy and twenty segments

Chapter 6

Analyses

Once the dataset described in the previous chapter had been generated, its exploitation toward the goal of the present work (the rapid exploration of the design space and the generation of dense porkchop plots at a modest computational cost) was entrusted to a machine-learning surrogate model based on artificial neural networks.

The adopted strategy consists in the implementation of a pipeline composed of two distinct neural networks, each dedicated to a specific task. The first is a classification network, tasked with distinguishing the parameter combinations that admit a physically realizable trajectory (feasible) from those that do not (infeasible); the second is a regression network, devoted to estimating the Δv (and, through it, the propellant mass) only for the feasible solutions. Consistently with this division of roles, the two networks are trained on different subsets of the dataset: the classifier on the entire set of samples, comprising both feasible and infeasible points, and the regressor exclusively on the feasible points, for which a value of Δv is defined. Once trained, the two networks are arranged in cascade: the classifier acts as an upstream filter, and only the points it recognizes as feasible are forwarded to the regressor for the estimation of the Δv, while the remaining ones are discarded. It is in this configuration that the pipeline is evaluated as a whole, so as to measure its end-to-end performance and the actual interaction between the two networks. The overall workflow, from the generated dataset through the training of the two networks to cascade inference, is illustrated in Figure 6.1.

This chapter is devoted to the description of the implementation of this pipeline and of the protocol by which its performance was evaluated. The tools and strategies adopted for data preparation and splitting, the definition of the architectures and the training of the two networks, the choice of the performance metrics, and the evaluation protocol are presented in turn, with particular attention to the distinction between in-distribution performance and the ability to generalize to spacecraft configurations never seen during training. The entire pipeline was implemented

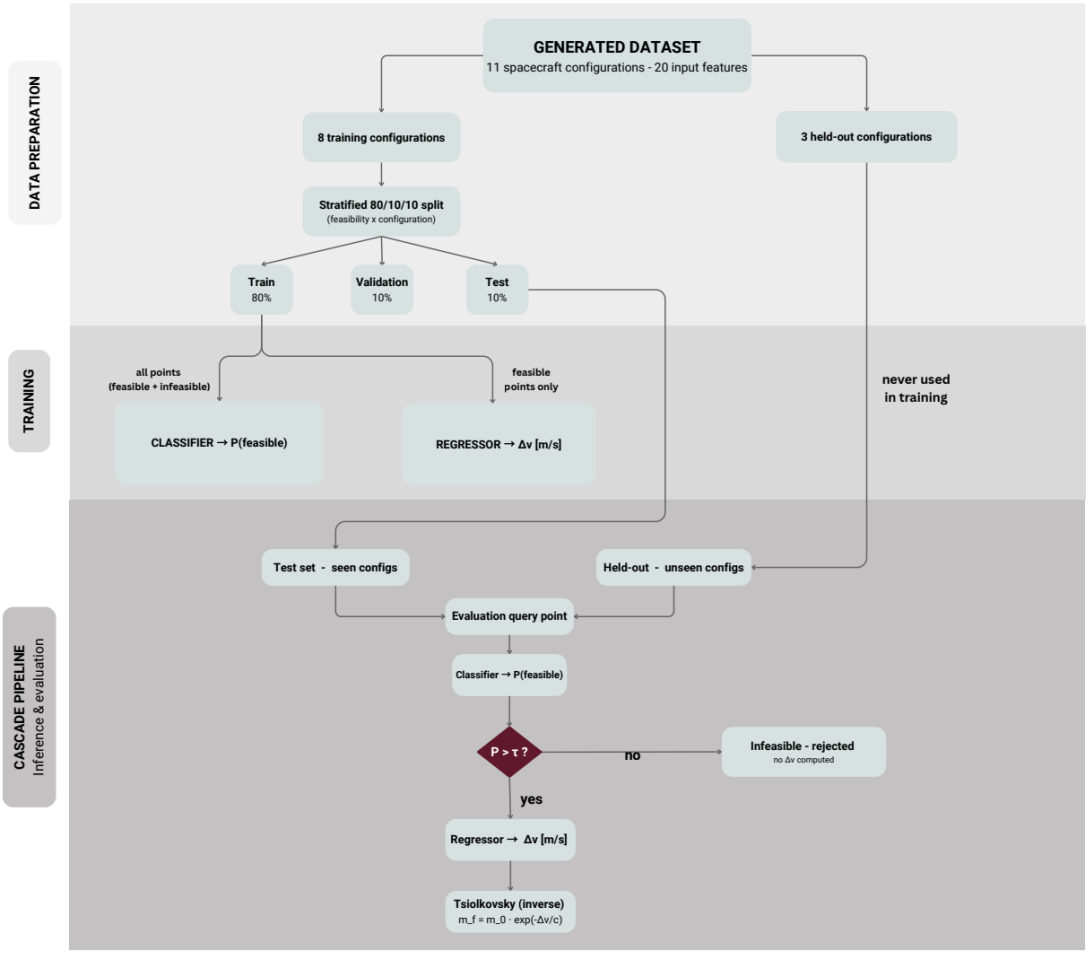


Figure 6.1: Overview of the two-network surrogate pipeline

in Python, relying on the `PyTorch` library [42, 43] for the definition, training, and evaluation of both neural networks. The results obtained are presented and discussed in Chapter 7.

6.1 Input Feature Vector

The dataset generated in the previous chapter constitutes the common starting point for the training of both networks. As described therein, each sample corresponds to a node of the (t_0, t_{of}) grid for a given spacecraft configuration, and records the outcome of the low-thrust trajectory optimization: its physical feasibility and, when the solution is feasible, the corresponding Δv.

The two parameters t_0 and t_{of} are not, however, supplied to the networks directly

as a bare pair of numbers. Starting from them, for each grid node a set of quantities that explicitly describe the geometry of the transfer was computed, again by means of the `pykep` library [24]: the heliocentric position vectors of the Earth at departure and of Mars at arrival ($\mathbf{r}_1$, $\mathbf{r}_2$), their respective velocity vectors ($\mathbf{v}_1$, $\mathbf{v}_2$), the true anomalies of the two planets (ν_E and ν_M), the phase angle between Earth and Mars (ϕ), and a periodic encoding of the synodic phase through its sine and cosine ($\sin\psi$, $\cos\psi$). To these is added the time of flight t_{of}, supplied directly. A total of 18 orbital features is thus obtained.

The choice of these quantities (and in particular the fact of supplying some that are in reality derivable from others) was made for a reason. On a purely informational level, the entire orbital geometry is uniquely determined by the values t_0 and t_{of} alone: given the ephemerides, positions, velocities, and anomalies all follow from them, and the phase angle and true anomalies are in turn derivable from the state vectors. The orbital features are therefore, strictly speaking, redundant. This redundancy is nonetheless deliberate: by providing the network directly with the relevant physical quantities, rather than with t_0 and t_{of} alone, one avoids forcing it to expend part of its capacity reconstructing internally astrodynamical relations that are already known (the planetary ephemerides, Kepler's equation, the dot product that defines the phase angle). The model's capacity thus remains focused on the genuinely hard task, namely learning the mapping from the transfer geometry and the engine parameters to feasibility and Δv. For the same reason, the departure epoch does not appear in the input vector as an absolute value: an epoch number in MJD2000 is not informative in itself and does not favor generalization, since the network would have no way of relating a large, unbounded integer to the geometry of the transfer. What is physically meaningful is not the epoch as such, but the position of the Earth and Mars within their synodic cycle, which recurs with a period of approximately 780 days. The epoch is therefore converted into a synodic phase angle, $\psi = 2\pi\, t_0/T_{\text{syn}}$, with t_0 expressed in days from the J2000 epoch, which expresses how far the Earth–Mars alignment cycle has advanced at that date. This angle is then supplied to the network not directly, but through its sine and cosine: since an angle of 0 and one of 2π describe the same physical configuration, feeding ψ as a raw number would introduce an artificial discontinuity at the end of each cycle, with two nearly identical geometries mapped onto numerically distant inputs. The pair $(\sin\psi, \cos\psi)$ removes this discontinuity, encoding the phase in a bounded and continuous form in which epochs separated by a full synodic period are correctly recognized as equivalent. Both components are required: the sine alone would be ambiguous, as it takes the same value at two distinct points of the cycle, whereas the pair identifies the phase uniquely.

To the 18 geometric features are added the two parameters that describe the spacecraft configuration: the thrust-to-mass ratio $a_0 = T_{\max}/m_0$ and the effective exhaust velocity $c = g_0 I_{sp}$. Here too, the choice is not to supply the raw parameters

($T_{\max},\ I_{sp},\ m_0$), but rather two physically meaningful combinations of them. The dynamics of the low-thrust transfer depend on the configuration only through these two quantities: a_0 represents the maximum available acceleration, and hence the control authority of the vehicle, while c governs the propellant consumption through Tsiolkovsky's equation. In particular, as far as the shape of the trajectory and the required Δv are concerned, what matters is not the absolute values of $T_{\max}$ and m_0 taken separately, but only their ratio: two configurations differing solely by a common scale factor (the same a_0 and the same c) give rise to the same trajectory and the same Δv. Adopting (a_0, c) in place of ($T_{\max}$, I_{sp}, m_0) therefore eliminates a redundant degree of freedom, the absolute scale, reducing the parameter space from three to two dimensions and presenting the network with a more compact and more readily generalizable description. The initial mass m_0 and the I_{sp} are nonetheless retained separately for each configuration, as they are required to convert the predicted Δv back into the final spacecraft mass (and therefore the mass of propellant), as described in the section devoted to the regressor.

A final consideration concerns the treatment of a_0. Across the eleven configurations considered, the thrust-to-mass ratio varies by several orders of magnitude, exhibiting a strongly asymmetric distribution that is poorly conditioned for linear normalization. For this reason, the feature actually supplied to the network is not a_0 but its logarithm, $\log(a_0)$: the transformation compresses the scale, rendering additive what would otherwise be multiplicative differences, and reflects the fact that the dependence of feasibility and Δv on a_0 is itself markedly nonlinear. The parameter c, which instead varies within a much narrower range (the I_{sp} values of the electric thrusters considered are all of the same order of magnitude) is kept on a linear scale.

The input vector of both networks is therefore composed of 20 features ($\mathbf{x}_{\text{in}} \in \mathbb{R}^{20}$), the 18 orbital ones plus a_0 and c, identical for the classifier and the regressor. It is reported in equation (6.1):

$$\mathbf{x}_{\text{in}} = \Big[\ \underbrace{t_{of},\ \mathbf{r}_1,\ \nu_E,\ \mathbf{r}_2,\ \nu_M,\ \phi,\ \sin\psi,\ \cos\psi,\ \mathbf{v}_1,\ \mathbf{v}_2}_{\text{18 orbital features}},\quad \underbrace{\log(a_0),\ c}_{\text{engine parameters}}\ \Big]^{\mathsf{T}} \tag{6.1}$$

where $\mathbf{r}_1, \mathbf{r}_2, \mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^3$, and $\psi = 2\pi\, t_0 / T_{\text{syn}}$ denotes the synodic phase, with $T_{\text{syn}} \approx 780$ days the Earth–Mars synodic period.

What distinguishes the two networks is solely the output, namely the feasibility label (0 for infeasible solutions, 1 for the feasible ones) for the classifier and the Δv value for the regressor, as summarized in Table 6.1.

Finally, the input vector configuration described here, inclusive of the redundant features, is the one adopted for the main results. In order to verify its actual usefulness, which cannot be taken for granted a priori, a number of trials were also carried out with the same architecture but with alternative input vectors, with

Network	Input	Output
Classifier (FeasibilityNet)	$\mathbf{x}_{\text{in}} \in \mathbb{R}^{20}$	feasibility label $\in \{0, 1\}$
Regressor (DvNet)	$\mathbf{x}_{\text{in}} \in \mathbb{R}^{20}$	Δv [m/s]

Table 6.1: Input and output of the two networks

the aim of assessing whether the redundancy of correlated features constitutes a genuine benefit to performance; these comparisons are reported in the results chapter.

6.1.1 Splitting and Stratification

Prior to training, the dataset was partitioned into three disjoint subsets: a *training* set, on which the network weights are optimized; a *validation* set, used to monitor performance during training without directly influencing the weight updates; and a *test* set, reserved for the final assessment of performance. This separation is standard practice in the training of machine-learning models and serves a precise purpose: to measure the model's true generalization capability, that is, its accuracy on data never seen during training, rather than merely its ability to reproduce the samples on which it was optimized. Evaluating a model on the same data used to train it would in fact yield an optimistically biased estimate, incapable of revealing possible overfitting.

The three subsets play distinct and complementary roles. The training set is the only one actually used to update the weights via gradient descent. The validation set intervenes instead in the decisions that concern the model but not its weights: the monitoring of the loss for early stopping, the selection of the best model, and, in the case of the classifier, the tuning of the decision threshold (as detailed in the dedicated sections). It thus serves as an independent reference during development, but the very fact of guiding such choices makes it indirectly part of the optimization process. For this reason, the final assessment is entrusted to the test set, which contributes in no way either to training or to the modeling decisions, and therefore constitutes the most genuine estimate of performance on new data.

The adopted partition assigns 80% of the samples to training and 10% each to validation and test. This proportion reflects a standard compromise: allocating the largest share of the data to training, so as to provide the networks with as many examples as possible to learn a complex mapping, while at the same time reserving for validation and test fractions numerous enough to make the respective estimates statistically significant.

The splitting is not, however, purely random, but *stratified* with respect to the combination of two criteria: the spacecraft configuration and the feasibility label.

A purely random partition would not guarantee that the proportions of the various classes are preserved across the three subsets; in the presence of an imbalance between feasible and infeasible points, or of configurations represented by differing numbers of samples, a subset could turn out to be unrepresentative — for instance, over or under representing a configuration or a class. Stratification instead requires that each of the three subsets preserve the same proportions as the full dataset with respect to both criteria, ensuring that training, validation, and test are statistically homogeneous with one another and that every configuration is adequately present in each of them.

It is important to emphasize that this splitting is performed on the entire dataset, infeasible points included, and that the resulting partition is common to the two networks: fixing a common *seed* for the random number generator makes it perfectly reproducible and guarantees that the classifier and the regressor share the same assignment of points to the training, validation, and test subsets, and, in particular, the same intact test set. The two networks do not, however, train on identical data: as detailed in the section devoted to the regressor, its training and validation sets are subsequently filtered to the feasible points alone, whereas the classifier uses them in full. What matters is that this filtering is applied only downstream of the common partition and never touches the test set, which therefore remains identical for both networks. Such consistency is essential for the subsequent evaluation of the pipeline as a whole, which requires that the points on which end-to-end performance is measured be exactly the same for the two networks and never seen by either during training.

Of the eleven configurations, eight are dedicated to training and are partitioned into training, validation, and test according to the criteria just described, for a total of 18,200 samples; the remaining three constitute the *held-out* set, with 6,825 samples, and are excluded entirely from the split and hence from training, validation, and test. These latter configurations take no part in any phase of the training and are reserved for a distinct evaluation, aimed at measuring the networks' ability to generalize not to new points of already seen configurations, but to entirely new spacecraft configurations.

6.1.2 Normalization

The twenty features that make up the input vector differ from one another by orders of magnitude and in units of measurement. The components of the heliocentric position vectors take values on the order of 10^{11} meters, those of the velocity vectors on the order of 10^4 meters per second, the anomalies and the phase angle are angular quantities on the order of unity, the sine/cosine encodings lie between -1 and 1, while $\log(a_0)$ and c occupy yet different ranges. Supplying the network with such heterogeneous quantities without any preliminary treatment would be

problematic: the features characterized by larger numerical values would dominate the weighted sums feeding the neurons and the corresponding gradients, steering the optimization toward those directions at the expense of the smaller magnitude features, regardless of their actual physical relevance. The result would be an ill-conditioned optimization problem, with slower and less stable convergence.

To remedy this, the input features were normalized by means of *standardization*, applied through the `StandardScaler` class of the `scikit-learn` library [44]. Standardization transforms each feature so that it has zero mean and unit standard deviation, according to the relation

$$z = \frac{x - \mu}{\sigma}, \tag{6.2}$$

where x is the original value of the feature, μ and σ are respectively its mean and standard deviation, and z is the standardized value. The transformation is applied independently to each of the twenty features, with its own pair of parameters (μ, σ). In this way all features are brought to a common, comparable scale, the optimization problem becomes better conditioned, and the regularization acts uniformly on all inputs.

A crucial aspect concerns the data on which the parameters μ and σ are estimated. To avoid any form of data leakage (that is the inadvertent transfer of information from the evaluation data to the training data) the standardization statistics are computed exclusively on the training subset. The same values (μ, σ) thus obtained are then applied, without any further fitting, to the validation, test, and held-out sets. Estimating the parameters on the latter as well would in fact allow information about data that must remain entirely unknown to the model to leak into the training phase, compromising the genuineness of the subsequent evaluation.

Since, as described earlier, the two networks are trained on different subsets of the dataset, each has its own independent scaler, fitted on the respective training set: that of the classifier is estimated on the entire training set, comprising both feasible and infeasible points, whereas that of the regressor is estimated on the feasible points alone employed for its training. Finally, it should be noted that the standardization described here concerns the input features only; in the case of the regressor, the Δv target is also subject to an analogous scaling, whose treatment is however deferred to the dedicated section, as it is specific to that network.

6.2 Classifier Network

The first of the two networks that make up the pipeline is the classifier, hereafter referred to as FeasibilityNet. Its task is to establish, for a given combination of transfer geometry and engine parameters, whether a physically realizable low-thrust

trajectory exists. This is therefore a binary classification problem: to each input point — the twenty-feature vector described in Section 6.1 — the network assigns a feasibility label, distinguishing the cases in which an admissible solution exists (label "1") from those in which it does not (label "0"). Within the pipeline, this network acts as an upstream filter: only the points it recognizes as feasible are forwarded to the regressor, while the remaining ones are discarded without further processing.

One characteristic of the problem decisively conditions much of the implementation described in what follows: the imbalance between the two classes. As it emerged in the chapter devoted to dataset generation, the feasible points constitute the majority of the samples, whereas the infeasible points represent a minority fraction. This imbalance has precise consequences for the way the network must be trained and, above all, evaluated: some seemingly natural metrics turn out to be misleading in the presence of imbalanced classes, and specific measures must be adopted both in the cost function and in the choice of the decision threshold, so that the minority class is not neglected by the optimization process. These aspects are taken up and justified in the following subsections.

6.2.1 Data Preparation

Applying to the subset of the eight training configurations the stratified splitting strategy described in Section 6.1.1 yields the partition reported in Table 6.2. The counts refer to the entire set of points, feasible and infeasible, on which the classifier is trained.

Configuration	Train	Val	Test
0	1,820	228	227
1	1,820	228	227
2	1,820	227	228
3	1,820	227	228
4	1,820	227	228
5	1,820	228	227
6	1,820	227	228
7	1,820	228	227
Total	**14,560**	**1,820**	**1,820**
Feasible fraction	**61.4%**	**61.4%**	**61.3%**

Table 6.2: Composition of the training, validation, and test partitions across the eight training configurations

Two aspects, already stated as objectives of the stratification, find confirmation here. First, the fraction of feasible points remains practically identical across the three subsets (about 61.4% in training and validation and 61.3% in test) confirming that the proportions between the classes are preserved by the split. Second, the samples are uniformly distributed across the eight configurations, each of which contributes the same number of points (2,275) and identical shares to training, validation, and test; the result is a per-configuration balanced dataset, in which no engine is over or underrepresented in any of the three subsets.

6.2.2 Architecture

The classifier is realized as a fully connected feed-forward neural network (a multilayer perceptron), whose architecture is composed by the sequence of layers of dimensions

$$20 \to 128 \to 64 \to 32 \to 1$$

The input layer receives the twenty feature vector; three hidden layers of decreasing width follow — 128, 64, and 32 neurons, respectively — and finally a single output neuron produces the logit associated with the feasibility prediction, that is, an unbounded real number which, once passed through the sigmoid function, becomes the predicted probability that the point is feasible. The presence of three hidden layers places the network, compact as it is, among the properly *deep* models, capable of learning a hierarchical representation of the features through the composition of several nonlinear transformations.

This architecture was arrived at empirically, through a preliminary experimentation phase in which networks of different depth and width were trained and compared, and the configuration reported above was retained as the one offering the best compromise between accuracy and complexity. Its overall form nonetheless follows two established practices in the design of multilayer perceptrons. The first is the funnel-shaped topology, in which the hidden layers narrow progressively: the wider initial layers extract a large number of intermediate features from the twenty inputs, while the subsequent, narrower ones combine and compress them toward the final binary decision. The second is the choice of layer widths as powers of two, a customary convention that is widely adopted in practice.

The specific sizes reflect a compromise between representational capacity and containment of complexity. The boundary that separates the feasible combinations from the infeasible ones is a strongly nonlinear surface, depending in an intricate way on the transfer geometry and the engine parameters; a weaker network would be unable to represent it, whereas depth makes it possible to build progressively more abstract and discriminative features. On the other hand, the input space is of modest dimensionality (twenty features), and an excessively large model would be not only superfluous but counterproductive: it would increase the risk of overfitting

and slow down inference, in contrast with the very purpose of the surrogate, namely the rapid evaluation of dense grids. The adopted sizes provide a capacity sufficient to model the feasibility boundary, and the total number of trainable parameters is 13,505. This figure denotes the trainable degrees of freedom of the model (the weights and biases adjusted during training) and thus provides a rough measure of its representational capacity; the value is well proportioned to the size of the training set, which favors good generalization.

Each hidden layer consists not of the linear transformation alone, but of a recurring block of four operations applied in sequence: an affine transformation, a batch normalization, a nonlinear activation, and a dropout stage.

Linear transformation (`Linear`). This is the affine transformation $\mathbf{y} = \mathbf{W}\mathbf{x}+\mathbf{b}$ at the core of each layer, in which the trainable weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ map the layer input onto the neurons of the following layer. It is the operation whose parameters are actually learned during training.

Batch normalization (`BatchNorm1d`). As the training progresses and the weights are updated, the distribution of the values flowing through each layer keeps shifting, so that every layer must continually adapt to inputs whose scale changes from one iteration to the next. This is a phenomenon that slows down convergence and makes the network sensitive to the initial choice of the weights. Batch normalization [45] counteracts this by standardizing the activations of a layer before they reach the activation function: for each mini-batch, their mean and variance are computed, and the values are rescaled to zero mean and unit variance, so that each layer always receives inputs on a comparable scale. Two trainable parameters, a scale and a shift, are then applied, allowing the network to adjust the normalized values if this proves advantageous. The result is a faster and more stable training, a reduced sensitivity to weight initialization, and a mild regularizing effect, due to the fact that each sample is normalized using statistics that depend on the other samples of the batch, and therefore vary slightly from one iteration to another.

Activation function (`ReLU`). The function adopted here to introduce the nonlinearity is the Rectified Linear Unit [46], defined as $\mathrm{ReLU}(x) = \max(0, x)$, which leaves positive values unchanged and sets negative ones to zero. Its advantage lies in its derivative. Training propagates the gradient backwards through the network by multiplying the derivatives of the activations layer after layer; with saturating functions such as the sigmoid, whose derivative is everywhere small, this product of small factors makes the gradient vanish before reaching the first layers, which therefore learn very slowly. The ReLU has instead a unit derivative for all positive inputs, so the gradient traverses the network without being attenuated. The ReLU

is applied in all hidden layers, whereas the output layer, as is customary for this type of task, employs no nonlinear activation.

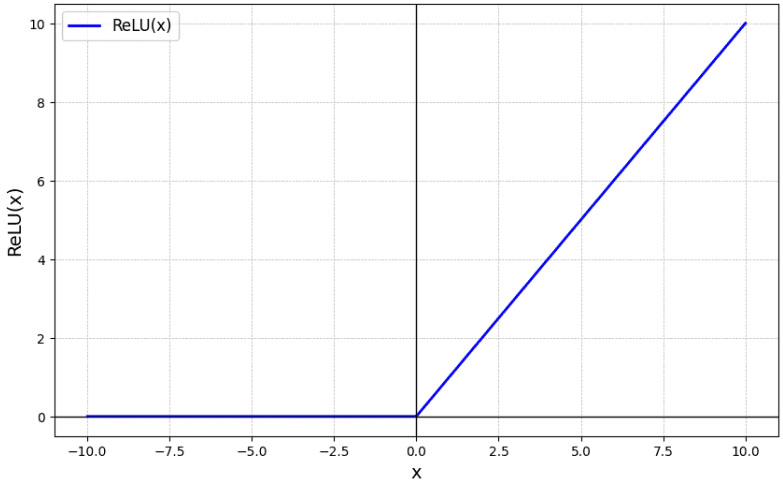


Figure 6.2: Representation of the Rectified Linear Unit (ReLU)

Dropout. During each training step, dropout [36] randomly deactivates a fraction of the neurons (set to 25% in the hidden layers) preventing the neurons from developing excessive co-adaptations. It constitutes one of the principal defenses against overfitting and is active only during training, being disabled at inference time.

A particular feature concerns the output layer: the single final neuron produces an unconstrained real value (the logit), without the explicit application of a sigmoid function within the network. The transformation of the logit into a feasibility probability, by means of the sigmoid function, is delegated to the cost function employed during training, for reasons of numerical stability discussed in the following subsections.

6.2.3 Evaluation Metrics

The evaluation of a binary classifier neural network rests on the comparison between the predicted labels and the true ones. Following the convention adopted in the dataset — in which the label 1 designates the *feasible* class, taken as the **positive** class, and the label 0 the *infeasible* class, taken as the **negative** class — each prediction falls into one of four categories, which are the building blocks of all the metrics described below.

Confusion matrix. The four possible outcomes of a prediction are summarized by the *confusion matrix*, shown in Figure 6.3: the true positives (TP), feasible points correctly recognized as such; the true negatives (TN), infeasible points correctly rejected; the false positives (FP), infeasible points erroneously classified as feasible; and the false negatives (FN), feasible points erroneously discarded. Within the pipeline, the two types of error are not equivalent and carry distinct physical consequences. A false positive corresponds to a point for which the real solver did not find an admissible solution but which the network has classified as feasible; it is then forwarded to the regressor, which assigns it a Δv value of dubious physical meaning, as no converged trajectory is associated with that point. A false negative corresponds instead to a realizable solution erroneously rejected, and hence to an admissible region of the design space that is lost and does not appear in the final map. This asymmetry motivates the attention devoted, in what follows, to the network's ability to recognize both classes correctly, and not merely the majority one.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Figure 6.3: Confusion matrix for the binary feasibility classification

Accuracy. The most immediate metric is the *accuracy*, defined as the fraction of correct predictions over the total:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{6.3}$$

Intuitive as it is, accuracy is a misleading metric in the presence of imbalanced classes. Since the feasible points constitute the majority of the dataset, a trivial classifier predicting feasible for every input would still attain an accuracy equal to the fraction of positives (a seemingly high value) while having learned nothing and being altogether useless as a filter, as it would identify no infeasible point whatsoever. Accuracy alone does not distinguish a competent classifier from one that merely exploits the class imbalance.

Precision, Recall, and F_1 score. A more informative assessment relies on metrics that analyze the two classes separately. The *precision* and the *recall* are defined, for the positive class, as

$$\text{precision} = \frac{TP}{TP + FP}, \qquad \text{recall} = \frac{TP}{TP + FN}. \tag{6.4}$$

Precision measures, among the points predicted as feasible, the fraction that actually is: it answers the question "how reliable are the positive predictions?". Recall measures, among the truly feasible points, the fraction that the network manages to recover: it answers the question "how many of the positive cases are identified?". The two quantities are in tension with each other (making the classifier more conservative typically increases one at the expense of the other) and are therefore summarized by a single metric, the F_1 score, defined as their harmonic mean:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{6.5}$$

Unlike the arithmetic mean, the harmonic mean attains a high value only when *both* quantities are high, penalizing classifiers that completely sacrifice either one. Precision, recall, and F_1 can be computed for each of the two classes by exchanging their roles; of particular relevance in what follows is the F_1 score of the *infeasible* class, which quantifies the network's ability to correctly identify the minority points and which will be employed for the tuning of the decision threshold.

ROC curve. The metrics described so far all depend on a fixed decision threshold, that is, on the probability value above which a point is declared feasible. An assessment of the intrinsic quality of the classifier, independent of the particular threshold chosen, is instead provided by the *Receiver Operating Characteristic* (ROC) curve. The ROC curve is constructed by continuously varying the decision threshold from 1 to 0 and reporting, for each value, the true positive rate (TPR) as a function of the false positive rate (FPR), defined as

$$\text{TPR} = \frac{TP}{TP + FN}, \qquad \text{FPR} = \frac{FP}{FP + TN}. \tag{6.6}$$

The TPR coincides with the recall of the positive class and represents the fraction of feasible points correctly identified; the FPR represents the fraction of infeasible points erroneously classified as feasible. As the threshold varies, the two rates increase or decrease together: a very high threshold makes the classifier extremely selective (TPR and FPR both close to zero), whereas a very low threshold makes it permissive (both close to one). The resulting curve therefore joins the origin (0,0) to the point (1,1), as illustrated in Figure 6.4, and its shape describes the

trade-off between sensitivity and false alarm rate attainable by the classifier. An ideal classifier reaches the top-left corner (0,1) — all positives identified with no false positives — while a classifier devoid of discriminative power, assigning random scores, lies along the diagonal TPR = FPR.

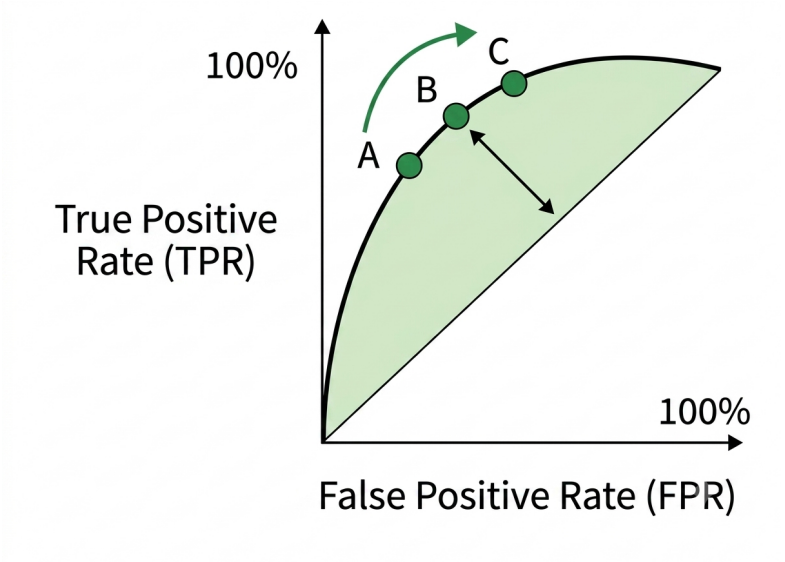


Figure 6.4: ROC curve: the true positive rate as a function of the false positive rate as the decision threshold is varied. Points closer to the top-left corner (from A to C) correspond to more favorable operating points; the diagonal represents a classifier with no discriminative power

Area under the curve (AUC). The area under the ROC curve (AUC) summarizes the entire curve in a single scalar, shown as the shaded region in Figure 6.5, and is the principal metric adopted for monitoring the classifier during training. Its value ranges between 0.5, corresponding to the random classifier, and 1, corresponding to the perfect classifier. The AUC admits a particularly intuitive probabilistic interpretation: it equals the probability that, given a randomly drawn positive point and a randomly drawn negative point, the network assigns the former a higher feasibility score than the latter. In other words, the AUC measures the model's ability to correctly *rank* the points according to their true feasibility, irrespective of any threshold. It is precisely this threshold-independence that makes it well suited as a criterion for model selection and early stopping: by assessing the quality of the ranking produced by the network rather than its decisions at an arbitrary threshold, the AUC provides a stable and robust measure of performance, less sensitive to class imbalance than accuracy and unaffected by the subsequent, independent choice of the operating threshold.

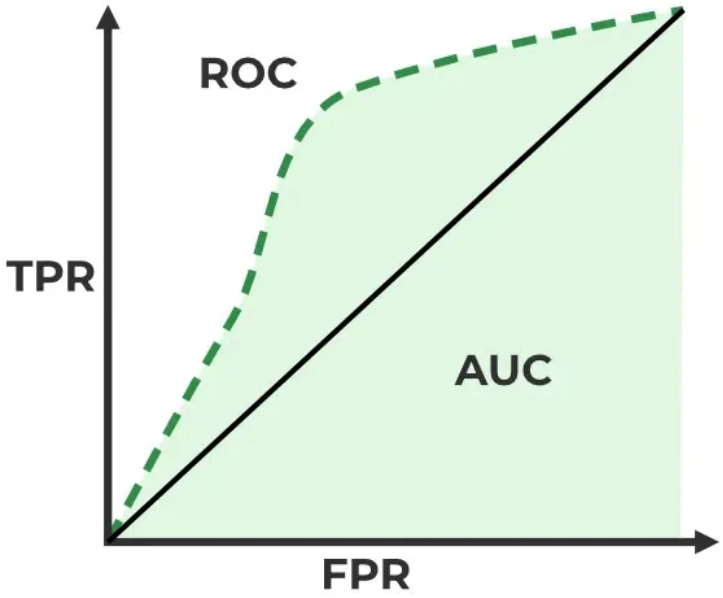


Figure 6.5: The area under the ROC curve (AUC), shaded in green. It ranges from 0.5 for a random classifier (diagonal) to 1 for a perfect one, and equals the probability that a randomly chosen positive point is ranked above a randomly chosen negative one

6.2.4 Training

The training of the classifier consists in the iterative optimization of the network weights so as to minimize a cost function that quantifies its prediction error. The optimization proceeds by gradient descent on mini-batches (subsets of the training set processed one at a time, each comprising 256 samples, a size chosen in proportion to the number of available data) and is repeated for a certain number of epochs, where an epoch denotes a complete pass over the entire training set. At the end of each epoch, performance is monitored on the validation set. The elements that characterize this procedure are described below.

Cost function (`BCEWithLogitsLoss`). As this is a binary classification problem, the adopted cost function is the binary cross-entropy. The implementation employed, `BCEWithLogitsLoss`, combines in a single operation the application of the sigmoid function to the logit produced by the network and the subsequent computation of the cross-entropy. This justifies the choice, anticipated in the subsection on the architecture, of having the network return an unconstrained logit rather than an explicit probability: by merging the two operations, the PyTorch library resorts to a numerically stable formulation that avoids the precision losses inherent in

the separate computation of the logarithm of a sigmoid. For a sample with true label $y \in \{0, 1\}$ and predicted probability $p = \sigma(x)$, where x is the logit, the cross-entropy is

$$\ell = -\Big[\, y \log p + (1 - y) \log(1 - p) \,\Big], \tag{6.7}$$

and the overall loss is the mean of ℓ over all samples of the batch. This function penalizes a prediction the more severely the further the estimated probability departs from the true label, driving the network to produce values close to 1 for feasible points and close to 0 for infeasible ones.

Handling the class imbalance (`pos_weight`). To counteract the class imbalance, the cost function was modified by introducing a weighting factor, named `pos_weight`, on the contribution of the positive class. In the weighted formulation, the term relating to the positive (feasible) samples is multiplied by this factor:

$$\ell = -\Big[\, w \cdot y \log p + (1 - y) \log(1 - p) \,\Big], \tag{6.8}$$

where w is the value of `pos_weight`. The adopted value is $w = n_{\text{neg}}/n_{\text{pos}}$, equal to the ratio between the number of negative and positive samples in the training set. This choice balances the contribution of the two classes to the overall loss: since the feasible (positive) class is the majority in the dataset, the ratio is less than unity (about 0.63) and consequently reduces the weight of the abundant class, preventing it from dominating the optimization and the minority infeasible class from being neglected. In the absence of this correction, the network could attain an overall low loss by concentrating on the feasible points alone, at the expense of the correct identification of the infeasible ones — precisely the error that, as noted, accuracy alone would fail to reveal.

Optimizer (Adam). The update of the weights from the gradients is entrusted to the Adam optimizer (*Adaptive Moment Estimation*) [35], among the most widespread in the training of neural networks. Adam automatically adapts the magnitude of the update of each parameter on the basis of estimates of the first and second moments of the gradients (their mean and their variance, respectively) accumulated over the course of training, combining the benefits of momentum and of adaptive step rescaling. This results in fast and robust convergence and a reduced need for manual tuning, reasons that make it a reference choice. A `weight_decay` term equal to 10^{-4} is associated with the optimizer. Weight decay is a form of L_2 regularization that penalizes large magnitude weights, adding to the cost function a contribution proportional to their squared norm; at each update the weights are thus slightly contracted toward zero. The effect is to discourage

excessively complex solutions and to limit overfitting, in a manner complementary to the dropout and batch normalization already described.

Learning rate and scheduling. The learning rate, which governs the magnitude of the weight updates, was set to an initial value of 10^{-3}, selected after experimenting with several trial values. To improve its behavior in the later stages of training, the learning rate is not kept constant but is adjusted dynamically by means of the `ReduceLROnPlateau` scheduler. This monitors the validation loss and, when it ceases to improve for a preset number of epochs (15), reduces the learning rate by a factor of 0.5. The strategy makes it possible to proceed initially with large steps, which rapidly explore the weight space, and then to progressively refine the search with smaller steps as a minimum is approached, reducing the risk of oscillating around it without converging.

Training loop and early stopping. Each epoch is divided into two phases. In the *training* phase the network operates in training mode (with dropout active and batch normalization using the statistics of the current batch) and, for each mini-batch, the loss is computed, its gradients are backpropagated, and the weights are updated. Before each update, the overall norm of the gradients is clipped to a maximum value of unity (*gradient clipping*), a measure that prevents destabilizing updates due to occasionally very large gradients. In the *validation* phase the network operates instead in evaluation mode (with dropout disabled and batch normalization using the accumulated global statistics) and, without gradient computation, its loss and AUC are evaluated on the validation set.

The stopping criterion adopted is early stopping based on the validation AUC. At the end of each epoch, if the AUC exceeds the best value observed up to that point, the current weights are saved as the best model; if instead no improvement is recorded for a preset number of consecutive epochs (a *patience* of 30), training is halted prematurely. At the end of the procedure, the weights of the best model are restored, rather than those of the last epoch. The choice of the AUC as the reference metric for early stopping, in place of the loss or the accuracy, follows from the properties discussed in the previous subsection. Early stopping additionally serves a twofold purpose: to prevent overfitting, by halting training when generalization performance ceases to improve, and to avoid needlessly prolonging the computation. The quantities monitored epoch by epoch (training loss, validation loss, and validation AUC) give rise to the learning curves reported in the results chapter.

6.2.5 Decision Threshold Tuning

The output of the classifier, as described, is a logit that, through the sigmoid function, is converted into a feasibility probability between 0 and 1. To turn this

probability into a binary decision (feasible or infeasible) a *decision threshold* must be fixed: the points whose estimated probability exceeds the threshold are classified as feasible, the others as infeasible. The choice of this value is not a secondary detail, since it determines the operating point of the classifier along the ROC curve described earlier, and with it the trade-off between false positives and false negatives.

The most immediate choice would be to adopt the conventional threshold of 0.5, classifying as feasible any point deemed more likely feasible than infeasible. That value, however, is optimal only when the classes are balanced and the two types of error are of equal importance. On the one hand, the class imbalance makes the threshold of 0.5 not necessarily optimal; on the other, the correction introduced by the `pos_weight` in the cost function alters the calibration of the probabilities produced by the network, shifting the most appropriate balance point away from the default value. An explicit tuning of the threshold is therefore required.

The threshold was selected downstream of training, on the basis of the performance measured on the *validation set* and not on the test set, which must remain extraneous to any modeling decision in order to guarantee an unbiased final evaluation. As the optimization criterion, the choice was made to maximize the F_1 score of the infeasible class. The rationale lies in the role of the classifier within the pipeline: its most delicate function is to correctly identify the infeasible points, so as to prevent points for which the solver found no solution from being forwarded to the regressor. Maximizing the F_1 score of the infeasible class balances precision and recall precisely on this class, steering the threshold toward an operating point that favors its correct identification, rather than optimizing an overall accuracy dominated by the majority class.

In practice, the procedure explores a discrete set of candidate thresholds, uniformly distributed over the probability interval, and for each one computes the F_1 score of the infeasible class on the validation set; the threshold that maximizes this score is finally adopted. The value thus identified is stored and used in all subsequent evaluation and inference stages, both on the test set and on the held-out configurations, as well as in the execution of the complete pipeline. In this way the threshold becomes a fixed operating parameter of the classifier, determined in a reproducible manner and consistent with the objective of not neglecting the minority class.

6.3 Regressor

The second network that makes up the pipeline is the regressor, hereafter referred to as DvNet, whose task is to estimate the Δv required by the transfer from the same twenty-feature vector described in Section 6.1. Unlike the classifier, which solves a

binary classification problem, DvNet tackles a *regression* problem: its goal is not to assign a discrete label, but to predict a continuous quantity that approximates as faithfully as possible the value obtained from the trajectory optimization. From the predicted Δv the final mass of the vehicle is then calculated through the Tsiolkovsky equation.

Within the pipeline, the regressor operates downstream of the classifier: it is not queried on every point, but only on those that the classifier has recognized as feasible, to which it associates the estimate of the transfer cost. Consistently with this role, and differently from the classifier (trained on the entire dataset) DvNet is trained exclusively on the feasible points, the only ones for which a Δv value is defined. Since the overall setup of this network largely mirrors the one already illustrated for the classifier, in what follows the shared aspects (the optimizer, the structure of each layer's block, the regularization rationale, and the scheduling and early-stopping strategy) are recalled without being derived again in detail, concentrating the treatment on the elements specific to the regressor.

6.3.1 Data Preparation

The regressor starts from the same dataset and from the same stratified 80/10/10 split described in Section 6.1.1. In particular, the split is performed using the same seed of the random number generator employed for the classifier: the partition of the points into training, validation, and test is therefore identical to the one already seen, with 14,560 samples allotted to training and 1,820 each to validation and test. This coincidence is not accidental, but a necessary condition: by guaranteeing that the two networks share exactly the same test set (intact and never seen by either of them) it makes possible the subsequent end-to-end evaluation of the pipeline, discussed in the dedicated section.

To this common basis the single difference specific to the regressor is applied. Since DvNet predicts the Δv, its training and validation sets are filtered to the feasible points alone, discarding the infeasible ones, which possess no target value. The test set, by contrast, is kept intact, with the infeasible points included, precisely because it must serve for the evaluation of the entire pipeline, in which the classifier intervenes upstream. The effect of the filtering on the size of the three subsets is reported in Table 6.3: training is reduced from 14,560 to 8,935 feasible samples and validation from 1,820 to 1,117, while the test remains unchanged at 1,820 points.

A further difference with respect to the classifier concerns the treatment of the *target*. As this is a regression problem, not only the input features but also the quantity to be predicted, the Δv, is standardized before training, by means of a second, dedicated `StandardScaler` [44]. The rationale is the same as for the inputs: the Δv values span a broad range, and bringing them to zero mean and unit variance stabilizes training, keeping the loss and its gradients on a favorable

Subset	After split (all points)	After filtering (feasible only)
Train	14,560	8,935
Validation	1,820	1,117
Test	1,820	1,820 (intact)

Table 6.3: Effect of the feasibility filtering on the size of the regressor's subsets

numerical scale, independent of the absolute order of magnitude of the target. Consistently with what was done for the features, the parameters of this scaler as well are estimated exclusively on the Δv of the feasible training points, so as not to introduce any information coming from validation or test. The network thus learns to predict a Δv in normalized form; at evaluation and inference time, its outputs are mapped back to the original physical unit (meters per second) by applying the inverse transformation of the scaler, before any comparison with the true values or any derived computation, such as the final mass via Tsiolkovsky.

6.3.2 Architecture

The architecture of DvNet takes up the same setup as the classifier, from which it differs only in its dimensions and in the output layer. The network is a fully connected feed-forward multilayer perceptron, whose topology is described by the sequence

$$20 \to 256 \to 128 \to 64 \to 32 \to 1$$

The twenty-feature input layer is followed by four hidden layers of decreasing width (256, 128, 64, and 32 neurons) and a single output neuron. Compared to the classifier, the network is therefore deeper (four hidden layers instead of three) and of greater capacity; the total number of trainable parameters rises accordingly to 49,601. This greater capacity reflects the different nature of the task: the regression of a continuous quantity over a broad range of values is a more elaborate problem than mere binary discrimination, and benefits from a network able to represent finer relations between the input features and the Δv.

Each hidden layer is built with the same block of operations already described for the classifier in Section 6.2.2 — linear transformation, batch normalization, ReLU activation, and dropout — with the same rationale, which is not repeated here. The only substantial difference concerns the output layer: whereas in the classifier the final neuron produces a logit destined for the sigmoid function, in DvNet the output is an unconstrained real value, interpreted directly as the predicted Δv (in normalized form, according to the target scaling described in the previous subsection). No activation function is applied to it, as is customary in

regression problems, in which the output must be able to take on, continuously, any value within the range of the targets.

6.3.3 Evaluation Metrics

The evaluation of a regression model rests on the comparison between the predicted and the true values of the target, and relies on metrics different from those employed for classification, as it must quantify not the correctness of a discrete decision, but the magnitude of the deviation between two continuous quantities. In what follows, y_i denotes the true value of the Δv for the i-th sample, $\hat{y}_i$ the corresponding value predicted by the network, and N the number of samples evaluated; all metrics are computed on the values expressed in the original physical unit, after the application of the inverse transformation of the scaler.

Mean absolute error (MAE). The mean absolute error is the average of the absolute deviations between prediction and true value:

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|. \tag{6.9}$$

It represents the typical error made by the network, expressed in the same unit as the target and therefore directly interpretable: a MAE of a given value indicates by how many meters per second, on average, the prediction departs from the true Δv. All deviations contribute in proportion to their magnitude, without the larger errors being emphasized.

Root mean square error (RMSE). The root mean square error is defined as

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}. \tag{6.10}$$

Also expressed in meters per second, it differs from the MAE in the squaring of the deviations, which assigns a greater weight to large-magnitude errors; as a consequence, it always holds that RMSE $\geq$ MAE, with equality only when all errors are of the same magnitude. The comparison between the two is therefore informative: when the values are close, the errors are homogeneously distributed, whereas an RMSE appreciably larger than the MAE indicates an error distribution with greater dispersion and heavier tails, that is, the presence of a minority of errors of magnitude well above the average, typically attributable to outliers, which weigh disproportionately on the quadratic metric.

Coefficient of determination (R^2). The coefficient of determination measures the fraction of the target variance that the model manages to explain, and is defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{N} (\hat{y}_i - y_i)^2}{\sum_{i=1}^{N} (y_i - \bar{y})^2}, \tag{6.11}$$

where $\bar{y}$ is the mean of the true values. Unlike the previous metrics, R^2 is dimensionless and provides a normalized measure of quality: a value of 1 corresponds to a perfect prediction, a value of 0 to a model that does no better than the simple constant prediction of the mean value, while negative values indicate performance worse than the latter. It thus makes it possible to judge the goodness of the model in relative terms, irrespective of the absolute scale of the target.

Relative error and fraction within a tolerance. The previous metrics are expressed in absolute terms and do not account for the fact that the same deviation, in meters per second, carries a different weight depending on the magnitude of the Δv to which it refers. For an assessment that is comparable across configurations characterized by Δv of differing scale, the mean percentage relative error is introduced,

$$E_{\text{rel}} = \frac{100}{N} \sum_{i=1}^{N} \left| \frac{\hat{y}_i - y_i}{y_i} \right|, \tag{6.12}$$

which expresses the deviation as a fraction of the true value. Alongside it is the percentage of predictions that fall within a preset tolerance (set at 5%) that is, the fraction of samples for which the relative error is below that threshold. This last metric offers an immediate reading of the practical accuracy of the surrogate, quantifying how many of its outputs turn out to be close enough to the true value to be deemed reliable.

Besides the error on the Δv, the evaluation also quantifies the error on the final mass m_f derived from the predicted Δv through the Tsiolkovsky equation; the corresponding results are reported in the dedicated chapter.

6.3.4 Training

The training of DvNet shares most of the setup already described for the classifier in Section 6.2.4, to which the reader is referred for the common aspects. In particular, the weight update is entrusted to the same Adam optimizer with weight decay equal to 10^{-4}, training proceeds by mini-batches of 256 samples with the same unit-norm gradient-clipping strategy, and the learning rate (initialized to 10^{-3}) is dynamically adjusted by the same `ReduceLROnPlateau` scheduler, which

reduces it by a factor of 0.5 when the validation loss ceases to improve. The training loop is articulated into the usual training and validation phases, with the network placed in the respective operating modes. Compared to the classifier, the only two conceptually different elements are the cost function and the early-stopping criterion, which are described below.

Cost function (Huber loss). As this is a regression problem, the cost function is no longer the cross-entropy, but a measure of the deviation between the predicted and the true Δv. The choice fell on the *Huber loss* [47], a hybrid function that combines the advantages of the mean squared error and the mean absolute error. Denoting by $r = \hat{y} - y$ the deviation between predicted and true value, and by δ a threshold parameter, it is defined as

$$L_\delta(r) = \begin{cases} \dfrac{1}{2} r^2 & \text{if } |r| < \delta, \\ \delta \left(|r| - \dfrac{1}{2}\delta \right) & \text{if } |r| \geq \delta. \end{cases} \tag{6.13}$$

Its behavior, illustrated in Figure 6.6, is therefore twofold. For small deviations, within the threshold δ, the function is quadratic and reproduces the mean squared error: the gradient is proportional to the error and attenuates near the minimum, ensuring a fine and stable convergence. For large deviations, beyond the threshold, the function becomes linear and reproduces the mean absolute error: the gradient remains bounded and constant, so that a single sample with a very large error does not generate a disproportionate gradient capable of destabilizing the weight update. The two branches are joined at $|r| = \delta$ with continuity of both the function and its first derivative.

This twofold nature makes the Huber loss particularly suited to the problem at hand. Compared to the pure mean squared error, it is more robust to outliers (those points, discussed earlier, for which the solver's solution is less accurate or the Δv takes on anomalous values) preventing a minority of difficult samples from dominating the training; compared to the pure mean absolute error, it retains the regularity and the precision of the optimization near the minimum, where the deviations are small. It should finally be noted that the loss operates on the normalized target, according to the target scaling described in Section 6.3.1: the adopted value $\delta = 1$ (the default one) therefore corresponds to about one standard deviation of the Δv, placing the transition between the two regimes at the deviations that are large relative to the typical scale of the target.

Early stopping. As for the classifier, training is governed by an early stopping criterion, with saving of the best model and restoration of its weights at the end. The difference lies in the quantity monitored: whereas in the classifier the

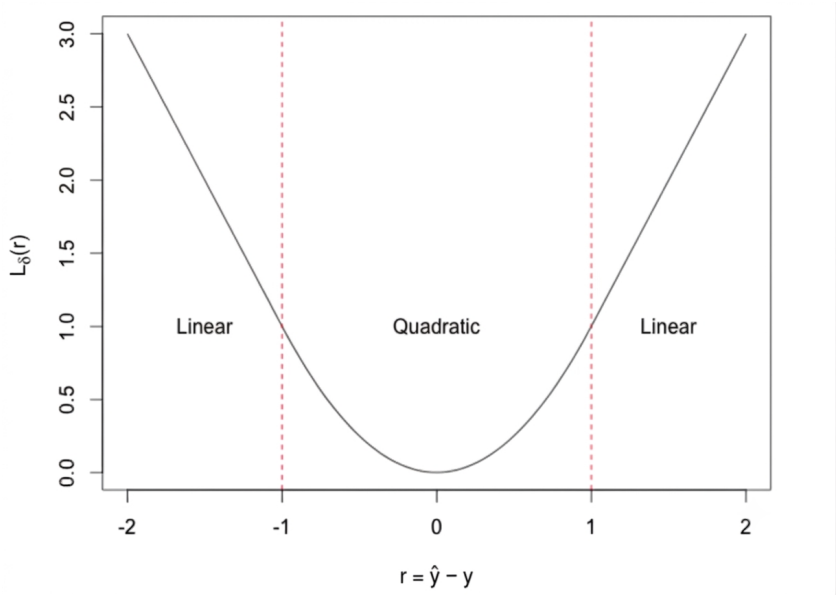


Figure 6.6: The Huber loss function

validation AUC was adopted, in the regressor the monitoring is performed directly on the validation loss. The reason is that, in a regression problem, the loss already constitutes in itself a direct and meaningful measure of the prediction error, free of the ambiguities that in classification made a dedicated metric preferable; there is therefore no reason to resort to a criterion different from the one that training itself minimizes. At each epoch, if the validation loss improves upon the best value observed, the weights are saved; if no improvement is recorded for a number of epochs equal to the patience (40), training is halted. The training and validation loss curves monitored over the course of training are reported in the results chapter.

6.4 Cascade Pipeline

The two networks described in the previous sections have so far been presented and evaluated separately, each in relation to its own task. Within the surrogate system, however, they do not operate independently, but as a single *cascade pipeline*, in which the output of the classifier determines whether and how the regressor is queried. For a generic query point the classifier first produces a feasibility probability and, by applying the previously tuned decision threshold, issues a binary decision. Only the points recognized as feasible are forwarded to the regressor, which estimates their Δv; the points classified as infeasible are instead discarded and receive no estimate. The classifier thus acts as an upstream filter, and the regressor intervenes exclusively on the subset of points that have passed

this filter. The entire flow, from the input point to the final estimate, is illustrated in the diagram of Figure 6.1.

For the points that traverse the whole cascade, the Δv estimate is finally converted into the final mass of the vehicle m_f by means of the Tsiolkovsky equation, in the form

$$m_f = m_0 \exp\left(-\frac{\Delta v}{c}\right), \tag{6.14}$$

where m_0 is the initial mass and $c = g_0 I_{sp}$ the effective exhaust velocity, both proper to the configuration considered and retained for each of them, as anticipated in Section 6.1. The propellant mass consumed during the transfer is obtained by difference as $m_0 - m_f$. For each feasible point, the pipeline therefore returns both the cost in terms of Δv and the final mass at the destination.

The purpose for which the pipeline was conceived is the rapid evaluation of dense grids in the design space, aimed in particular at the generation of *porkchop plots*: maps that report the cost of the transfer in terms of the Δv as a function of the departure date and the time of flight, and that constitute a classic tool in mission analysis for identifying the most favorable launch windows. Whereas the construction of such a map by the direct approach would require the optimization of one trajectory for each node of the grid, the surrogate makes it possible to obtain the same result in times reduced by orders of magnitude, evaluating the entire grid with a single pass of the two networks. It is precisely the evaluation of the pipeline in this usage regime, and not of the two networks in isolation, that constitutes the object of the following subsections.

6.4.1 Compound Error

When the two networks operate in cascade, the errors of each no longer manifest in isolation, but combine: the overall error of the pipeline on a given point depends both on the correctness of the classifier's decision and on the accuracy of the Δv estimate. One speaks in this sense of *compound error*, to emphasize that the performance of the end-to-end system does not coincide with that of the two networks evaluated separately, but emerges from their interaction. In particular, since the regressor acts only downstream of the classifier's gate, it inherits the latter's decisions: an upstream classification error irremediably conditions what happens downstream, regardless of the intrinsic quality of the regressor.

To analyze this interaction, it is useful to trace each point back to one of the four categories of the classifier's confusion matrix, examining what each entails for the overall output of the pipeline.

True positives (TP). These are the genuinely feasible points that the classifier correctly recognizes as such and forwards to the regressor. They constitute the nominal, "good" case of the pipeline: the point traverses the entire cascade and receives a Δv estimate. It is on this category that the actual accuracy of the pipeline in its intended operation is measured, and it is here that the compound error reduces to the regression error alone, there being no classification error to contaminate it.

True negatives (TN). These are the genuinely infeasible points that the classifier correctly rejects. They are discarded and receive no estimate, which is precisely the desired behavior: the gate has performed its function, preventing a point without a solution from reaching the regressor. This category introduces no error into the pipeline output.

False positives (FP). These are the points that the solver did not solve but that the classifier recognized as feasible, forwarding them to the regressor, which returns a Δv estimate for them nonetheless. This category has a twofold nature, which will be examined in depth in the results chapter. On the one hand, it may correspond to genuine classification errors: points truly devoid of a solution, to which a spurious Δv value is assigned, propagating all the way to the final map as an apparently feasible region. On the other hand, however, some of these points might not be errors at all, but rather cases in which the direct solver failed (not converging despite the existence of a realizable solution) and in which the network, having learned the regular trend of the phenomenon from the surrounding points, "corrects" that failure by providing a plausible estimate. In this second sense, the false positives would represent a desirable behavior, a regularizing effect of the pipeline with respect to the artificial discontinuities of the dataset. The distinction between the two situations cannot be determined at this level and is addressed in the results chapter; here it suffices to observe that not all false positives are necessarily to be interpreted as errors.

False negatives (FN). These are the genuinely feasible points that the classifier erroneously discards, not forwarding them to the regressor. Unlike the false positives, they do not introduce spurious values, but rather a coverage deficiency: a realizable solution is lost and does not appear in the final map, which thus turns out to be more conservative than reality. The error here is not on a predicted value, but on the absence of a prediction that ought to have been present.

From this analysis emerges the asymmetry between the two types of error already observed in Section 6.2.3: the false positives introduce into the map regions classified as feasible in the absence of a solver solution (whether spurious or the

result of a correction by the network) while the false negatives subtract real regions from it. The end-to-end evaluation of the pipeline, described in what follows, aims precisely at quantifying the overall incidence of these effects, measuring separately the accuracy of the classification gate and that of the regression on the correctly forwarded points.

6.4.2 Evaluation Protocol

The evaluation of the pipeline is carried out according to two distinct modes, conceived to measure two different capabilities of the surrogate system: the fidelity of its predictions on configurations already seen during training, and its ability to generalize to entirely new spacecraft configurations. The distinction rests on the two sets of data introduced in Section 6.1.1, the test set and the held-out configurations, which are here employed for two complementary purposes.

In-distribution evaluation. The first mode employs the *test set*, that is, the portion of data belonging to the eight training configurations that was set aside and never used either for the weight update or for any modeling decision. Since these configurations are the same ones employed in training (with (t_0, t_{of}) points different from those seen) this evaluation measures the pipeline's ability to interpolate within the domain on which it was trained, that is, to correctly predict feasibility and Δv for new combinations of departure date and time of flight relating to known engines. The test set, kept intact with both its feasible and infeasible points, additionally allows the evaluation of the entire cascade: the infeasible points it contains make it possible to measure the accuracy of the classification gate, while the feasible ones, correctly forwarded, make it possible to measure the regression error. It is in this sense that the test set constitutes the reference for the evaluation of the compound error discussed in the previous subsection.

Generalization evaluation (held-out). The second mode, more stringent, employs the three *held-out* configurations, excluded entirely from the training of both networks. These configurations correspond to engines with parameters (a_0, c) that, while falling within the overall range covered during training, do not coincide with any of the eight known configurations, being located at intermediate positions with respect to them. Their evaluation no longer measures the ability to interpolate between (t_0, t_{of}) points of a known configuration, but rather the ability to generalize along the direction of the configuration parameters: the network is required to correctly predict feasibility and Δv for values of (a_0, c) never seen as a training point, interpolating the learned relation between engine parameters, transfer geometry, and cost across the gaps that separate the training configurations. Since the latter are discrete and sparse in the parameter space, this capability

depends on the continuity and regularity of the relation learned in those regions, where the density of training data may be low. This is the most significant test bench for the practical usefulness of the surrogate: its ability to provide reliable estimates for an engine not present in the dataset determines the possibility of employing it as a general predictive tool, and not as a mere interpolator restricted to the training configurations.

Both modes are applied to the entire pipeline operating in cascade, and not to the two networks in isolation, so as to evaluate its performance in the actual usage regime. The comparison between the results obtained in the two modes finally makes it possible to quantify the possible performance degradation associated with the transition from known to new configurations, which represents the most direct measure of the real generalization capability of the system.

6.4.3 Analyses Conducted

On the basis of the protocol just described, the pipeline was subjected to a series of analyses aimed at characterizing its performance under various aspects. This section describes only their methodological setup; the corresponding results, with the related figures and quantitative measures, are presented and discussed in the following chapter. The analyses fall into a first group, devoted to the evaluation of the surrogate in its intended use, and a second, complementary group, aimed at probing its robustness and reliability beyond nominal conditions.

Predicted porkchop plots. The first and most direct analysis consists in the generation, for each configuration, of the porkchop plots predicted by the pipeline and in their comparison with those obtained from the direct solver. The network's porkchop plots are constructed by evaluating the pipeline on a dense grid in the (t_0, t_{of}) plane, with a step of one day on both axes, a resolution far finer than that of the grid employed during training. The generation exploits the structure of the feature vector: since the eighteen orbital features depend solely on (t_0, t_{of}) and not on the configuration, while the two engine parameters (a_0, c) are constant over the entire grid, the orbital features (whose computation through PyKEP constitutes the most expensive part) are computed only once on the common grid, and the engine parameters proper to each configuration are then simply appended to them. The comparison between the predicted and the real porkchop makes it possible to visually assess the fidelity with which the surrogate reproduces the structure of the transfer cost, including the position and shape of the most favorable launch windows.

Coverage maps in the parameter space. A second analysis aims to visualize how the quality of the prediction is distributed across the engine parameter space. To this end, a map is constructed that reports, for each configuration, the coefficient of determination R^2 of the regressor as a function of its position in the (a_0, c) plane, distinguishing the training configurations from the held-out ones. This representation makes it possible to identify the regions of the parameter space in which the surrogate generalizes with greater or lesser accuracy, and to relate the performance on the held-out configurations to their location with respect to the configurations seen during training, clarifying the link between training data density and generalization capability discussed in the previous subsection.

Pointwise inference. The third analysis consists in the direct, point-by-point comparison between the Δv predicted by the pipeline and that computed by the solver, carried out both on the in-distribution points and on those of the held-out configurations. Unlike the porkchop plots, which offer an overall reading, this analysis quantifies the prediction error on the individual samples, providing a direct measure of the surrogate's accuracy in the two evaluation modes.

Alongside the main analyses are a number of complementary checks, aimed at assessing the behavior of the pipeline under conditions more severe than the nominal ones.

Temporal extrapolation. To probe the behavior of the pipeline outside the temporal interval seen during training, for two of the training configurations (chosen among the training ones, and not among the held-out ones, so as to isolate temporal extrapolation alone without superimposing on it the generalization to a never seen engine) an additional set of reference data was generated over a window of departure dates subsequent to the original one, starting from the day immediately following the last node of the training grid, with the same extent and the same times of flight. The pipeline was then queried on these points and its predictions compared with the solver, so as to assess its reliability in a regime of genuine temporal extrapolation, in which the departure dates no longer fall within the training interval.

Off-grid inference. Since the predicted porkchop plots are generated with a one-day step, whereas the training grid is coarser, it is appropriate to verify the accuracy of the pipeline precisely on the points farthest from the training nodes, where interpolation is most demanding. To this end, for five configurations (the three held-out ones and two training ones) the points located at the centers of the training grid cells, that is, at the maximum distance from the seen samples, were selected. For each configuration, 200 points were sampled, stratified so as to represent the feasible interior regions, the infeasible interior ones, and above all the

boundary regions, where the feasibility frontier makes interpolation most critical. On these points the network's predictions were compared with the solver, so as to validate the surrogate's ability to correctly interpolate between the training nodes.

Robustness to initialization. To assess the sensitivity of the results to the random initialization of the weights, training was repeated using ten different seeds for the random number generator. The comparison between the performances obtained with the different initializations makes it possible to estimate the variability of the results and hence the robustness of the pipeline with respect to this factor.

Learning curves versus dataset size. A further analysis is aimed at evaluating the effect of the training set size on the performance of the pipeline. To this end, training was repeated using increasing fractions of the training set — 25%, 50%, 75%, and 100% of the available data — and for each fraction the corresponding learning curves were traced. The comparison between the results obtained with different amounts of data makes it possible to study their trend and to establish whether, and to what extent, an increase in the dataset size translates into an improvement in performance, or whether the latter has already reached a saturation beyond which the addition of further samples would bring a marginal benefit.

Comparison of architectures and input vectors. Finally, as anticipated in Section 6.1, different variants of the network were compared, both in the size of the architecture and, above all, in the composition of the input vector. This last comparison is aimed in particular at verifying the actual usefulness of the redundant features, assessing whether their inclusion constitutes a genuine benefit to performance with respect to more essential input vectors.

The details and outcomes of all these analyses are reported in the results chapter.

Chapter 7

Results

The previous chapter established the methodological framework of the surrogate system: the composition of the input feature vector, the partitioning of the dataset, the architectures of the two networks, the criteria by which they were trained, and the metrics by which their behavior is assessed, up to the evaluation protocol of the cascade pipeline. The present chapter reports the concrete outcomes of that framework: the numbers, the curves, and the maps that quantify the actual performance of the surrogate and put its reliability to the test. Every analysis presented here finds its methodological justification in Chapter 6, to which the reader is referred for the definition of the quantities and procedures involved; the treatment here concentrates instead on the reading and discussion of the results.

7.1 FeasibilityNet

The first component of the pipeline to be evaluated is FeasibilityNet. Its results are presented in three stages: the learning curves recorded during training, together with the decision threshold selected on the validation set; the classification performance measured in-distribution on the test set; and the generalization to the three held-out configurations, excluded entirely from training. All the outcomes discussed below are collected in Figure 7.1, which gathers the learning curves (top row) and the evaluation results (bottom row) in a single overview.

7.1.1 Learning Curves

The top row of Figure 7.1 reports the quantities monitored over the course of training. The binary cross-entropy loss (top left) decreases monotonically for both the training and the validation set, with a steep drop during the first few epochs followed by a settling onto a plateau slightly below 0.1. The validation AUC

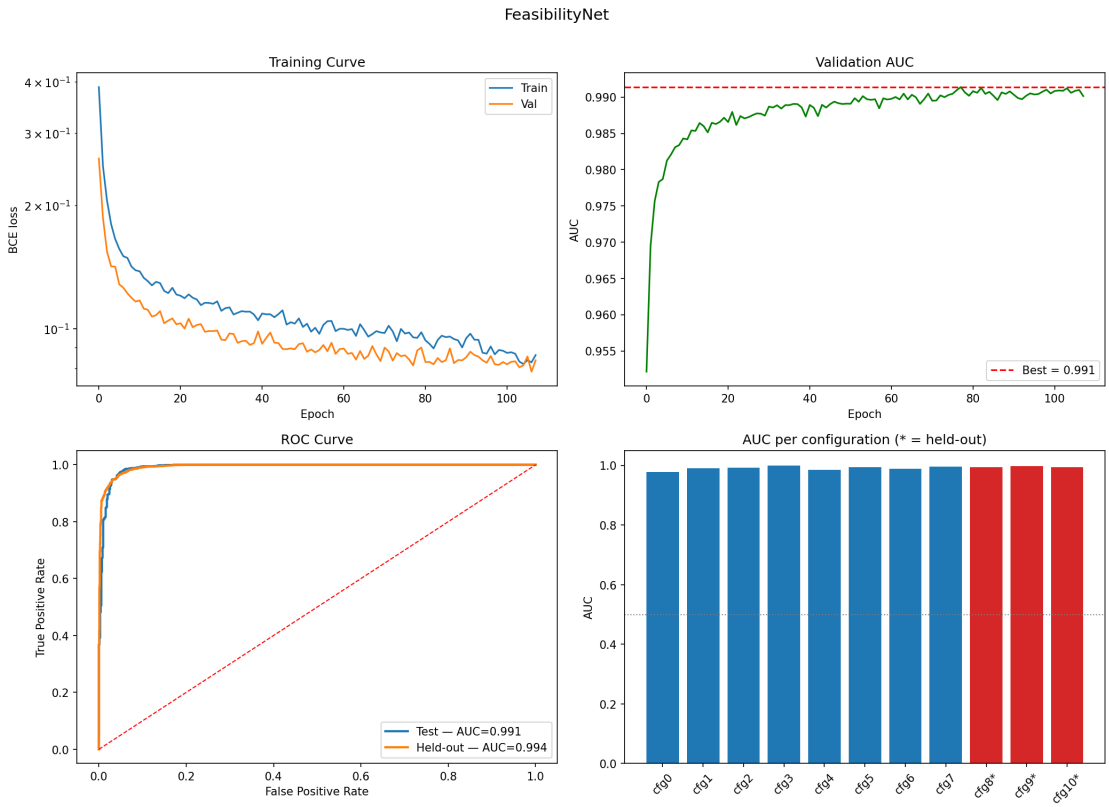


Figure 7.1: Overview of the FeasibilityNet evaluation

(top right) exhibits the complementary behavior: it rises rapidly to about 0.985 within the first twenty epochs and then improves more gradually, stabilizing around 0.990. The best value, 0.9914, is attained at epoch 78; the early-stopping criterion then halts training at epoch 108, after which the weights of the best model are restored. Neither a divergence between the training and validation loss curves nor a degradation of the validation AUC is observed in the final epochs, confirming that early stopping interrupted training before the onset of overfitting.

One feature of the loss curves deserves comment: the validation loss lies consistently below the training loss, from the very first epoch onward (0.388 against 0.260 at epoch 1). Counterintuitive as it seems, a model appearing more accurate on unseen data than on its training data is not an anomaly here, but a well-documented artifact of how the two quantities are computed. Two distinct mechanisms concur.

The first is the asymmetric nature of the regularization. As described in Chapter 6, each block of the network includes dropout and batch normalization, both active during the training phase but disabled at evaluation. In the training phase, dropout randomly deactivates a fraction of the neurons, so that the loss is

measured on a "thinned" network operating at reduced capacity; at evaluation, by contrast, the network employs all its neurons and operates at full capacity. The training loss therefore reflects the performance of a network penalized by the regularization, whereas the validation loss reflects that of the complete network: it is thus to be expected, in the absence of overfitting, that the latter should be lower than the former.

The second mechanism concerns the moment at which the two quantities are measured. The training loss is averaged *over the course* of the epoch, while the weights are continuously updated at each mini-batch; the validation loss is instead computed *at the end* of the epoch, once the weights have already benefited from all the updates of that epoch. Validation is therefore evaluated on a state of the network systematically more advanced than the average state with which the training loss is measured. This effect is greatest in the very first epochs, when each backpropagation step improves the network appreciably, which explains why the gap is already marked at epoch 1 and then tends to shrink as training proceeds and the intra-epoch improvements become smaller. The combination of the two effects fully accounts for the relative positioning of the two curves, which is therefore to be read as a sign of correctly functioning regularization and of the absence of overfitting, and not as a shortcoming. This ordering naturally holds only in the absence of overfitting: if training were prolonged well beyond the early-stopping point, the two curves would eventually diverge, the training loss continuing to fall while the validation loss starts to climb.

From the validation set, according to the criterion described in Chapter 6, the decision threshold that maximizes the F_1 score of the infeasible class was selected, yielding a value of 0.272, appreciably lower than the naive value of 0.5. This shift is consistent with the recalibration of the predicted probabilities induced by the `pos_weight` correction (which, by downweighting the majority feasible class, compresses its predicted probabilities downward) and with the chosen objective, which favors an operating point capable of correctly recovering the minority class. This value is fixed once and for all and employed in every subsequent evaluation, on both the test set and the held-out configurations.

7.1.2 In-Distribution Performance

The classification performance on the test set, comprising 1,820 points drawn from the eight training configurations and never seen during weight optimization, is summarized by the confusion matrix in Table 7.1 and by the per-class metrics in Table 7.2. The overall accuracy is 0.9599 and the AUC 0.9911, the latter matching the value observed on the validation set and confirming the excellent discriminative capability suggested by the ROC curve of Figure 7.1, which closely approaches the top-left corner of the diagram.

	Pred. infeasible	**Pred. feasible**
True infeasible	641	63
True feasible	10	1,106

Table 7.1: Confusion matrix of FeasibilityNet on the in-distribution test set

Beyond the aggregate figures, the per-class breakdown is what best characterizes the behavior of the classifier in the presence of imbalanced classes. Both classes are recognized with high and well balanced quality: the feasible class attains an F_1 of 0.968, the infeasible one an F_1 of 0.946. The fact that the minority (infeasible) class is recovered almost as effectively as the majority one is the tangible result of the measures adopted against the imbalance, and demonstrates that the classifier has not simply learned to exploit the prevalence of the feasible points.

A slight asymmetry between the two error types can also be noted: the network produces 63 false positives against only 10 false negatives, and correspondingly the fraction of points predicted as feasible slightly exceeds the true one.

Evaluation	**Class**	**Precision**	**Recall**	F_1
In-distribution (test)	Infeasible	0.985	0.911	0.946
	Feasible	0.946	0.991	0.968
Held-out	Infeasible	0.961	0.934	0.947
	Feasible	0.962	0.978	0.970

Table 7.2: Per-class precision, recall, and F_1 score of FeasibilityNet on the in-distribution test set and on the held-out configurations

The per-configuration performance, reported in the upper portion of Table 7.4 and in the bar chart of Figure 7.1, confirms that this quality is uniform across the eight training configurations: the AUC never falls below 0.9776 and reaches 0.9999 for configuration 3, with no configuration standing out as an anomaly. The classifier thus performs consistently across the entire range of engine regimes considered, and its accuracy is not driven by a subset of favorable configurations.

7.1.3 Generalization to Held-Out Configurations

The most stringent test of the classifier concerns the three held-out configurations, whose 6,825 points correspond to engines with parameters (a_0, c) never seen during training. The results, reported in the lower portion of Table 7.2 and Table 7.4, are remarkable: the accuracy is 0.9616 and the AUC 0.9935, values even marginally superior to those measured in-distribution. The ROC curve on the held-out set, shown alongside the test one in Figure 7.1, is essentially indistinguishable from it.

The confusion matrix is reported in Table 7.3: the classifier produces 166 false positives and 96 false negatives, preserving the same slight lean toward the over-prediction of feasibility observed in-distribution.

	Pred. infeasible	**Pred. feasible**
True infeasible	2,347	166
True feasible	96	4,216

Table 7.3: Confusion matrix of FeasibilityNet on the held-out configurations

The comparison of the per-class metrics is especially telling: the F_1 score of the infeasible class is 0.947 on the held-out configurations against 0.946 in-distribution, a difference well within statistical noise. The classifier therefore generalizes to engines it has never seen with no loss of quality, a strong indication that it has learned the underlying relation between engine parameters, transfer geometry, and feasibility, rather than memorizing the specific configurations of the training set.

The held-out configurations lie at intermediate positions within the (a_0, c) range spanned by the training set, so that what is measured is the ability to interpolate *across the configuration space*, not to extrapolate beyond it. The near-absence of degradation, with per-configuration AUC values all between 0.9940 and 0.9969, confirms that within the covered range the learned relation is regular enough for the network to fill the gaps between the discrete training configurations reliably.

Config	N	%feas (true)	%feas (pred)	Accuracy	AUC
0	227	59.9	64.8	0.9427	0.9776
1	227	59.0	63.0	0.9427	0.9898
2	228	61.0	64.5	0.9649	0.9923
3	228	59.2	59.2	0.9912	0.9999
4	228	65.4	68.9	0.9649	0.9856
5	227	55.9	59.5	0.9471	0.9935
6	228	63.6	66.7	0.9605	0.9881
7	227	66.5	67.4	0.9648	0.9964
8	2,275	61.3	64.8	0.9609	0.9940
9	2,275	70.5	67.3	0.9578	0.9969
10	2,275	57.7	60.5	0.9662	0.9943

Table 7.4: Per-configuration performance of FeasibilityNet. The upper block reports the eight training configurations (in-distribution test set), the lower block the three held-out ones

7.2 DvNet

Unlike FeasibilityNet, which solves a binary classification problem, DvNet tackles a regression problem, and its evaluation rests on the metrics introduced in Chapter 6: the mean absolute error (MAE), the root mean square error (RMSE), the coefficient of determination R^2, the mean relative error, and the fraction of predictions falling within the $\pm 5\%$ tolerance. As for the classifier, the results are presented by distinguishing the in-distribution performance on the test set from the generalization to the held-out configurations, and are collected in the overview of Figure 7.2.

7.2.1 Learning Curves

The learning curve (Figure 7.2e, bottom left) reports the Huber loss, computed on the normalized target, for the training and the validation set over the epochs. Both decrease smoothly, with the usual steep drop during the first few epochs followed by a progressively slower relaxation; training is halted by early stopping at epoch 208, with the restoration of the weights corresponding to the best validation loss (0.0316). The trend, free of any rise in the validation loss during the final epochs, confirms the absence of overfitting and the appropriateness of the stopping point.

Here too the validation loss lies below the training loss for the entire duration of training, for the same reasons already discussed in detail for FeasibilityNet in Section 7.1.1.

7.2.2 In-Distribution Performance

On the test set, comprising the 1,116 feasible points drawn from the eight training configurations, the regressor attains a coefficient of determination $R^2 = 0.9388$, a MAE of 78.96 m/s, and an RMSE of 177.84 m/s, over a real Δv range spanning from about 5,640 to 9,000 m/s. In relative terms, the mean error is 1.17% and 94.4% of the predictions fall within the $\pm 5\%$ tolerance, as summarized in Table 7.6. The scatter plot of predicted against true Δv (Figure 7.2a) shows the points clustered along the ideal diagonal, with a contained dispersion and no systematic distortion; the distribution of the relative error (Figure 7.2c) is centered on zero ($\mu = -0.08\%$) and narrow ($\sigma = 2.48\%$), confirming the absence of bias and the concentration of the errors around the correct value.

The comparison between MAE and RMSE is itself informative: the latter, more than twice the former, indicates an error distribution with heavier tails than the Gaussian, that is, the presence of a minority of points with deviations well above the average, which weigh disproportionately on the quadratic metric. This behavior, expected and indeed anticipated in the choice of the Huber loss as cost function

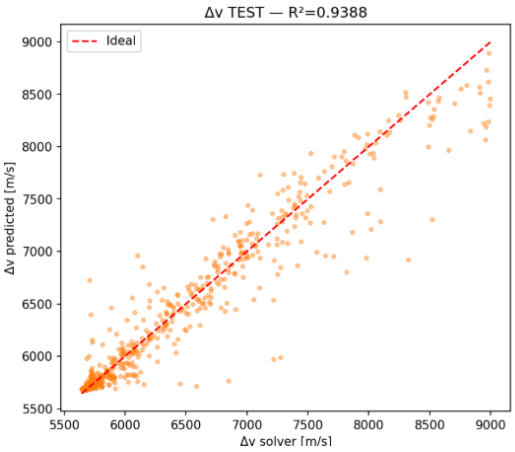
(robust precisely against such outliers), is visible in the scatter plot as a handful of points detached from the main cluster, mostly concentrated at the high values of the Δv range.

The error on the final mass m_f, derived from the predicted Δv through the Tsiolkovsky equation, is even smaller in relative terms: the MAE is 3.85 kg over a real range from 414 to 3,105 kg, for a mean relative error of 0.29%. The reason for this further reduction lies in the exponential nature of the Tsiolkovsky relation: for the values of Δv and effective exhaust velocity at play, the final mass depends in an attenuated manner on variations of the Δv, so that the relative error on m_f turns out to be a fraction of that on Δv.

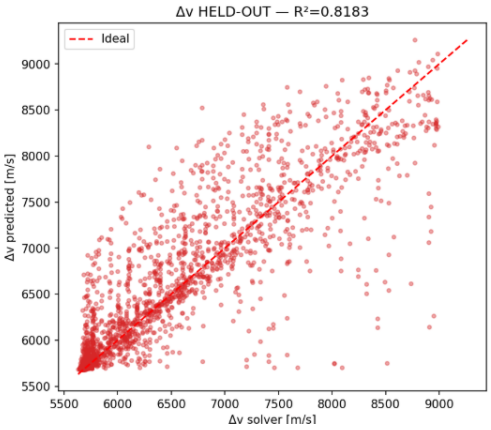
The per-configuration results, reported in the upper portion of Table 7.5, confirm the quality of the regression on the majority of the configurations, but also bring out its variability: seven of the eight training configurations exhibit an R^2 between 0.926 and 0.974, with mean relative errors below 1.4% and fractions within ±5% above 90%. The exception is configuration 4, which stands out as the most challenging of the in-distribution set, with an R^2 of 0.818, a mean relative error of 1.98%, and a MAE of 133.44 m/s, appreciably higher than that of the others. This configuration evidently corresponds to an engine regime in which the relation between transfer geometry and Δv is less straightforward to learn; its behavior will be taken up and framed, together with that of the held-out configurations, in the analysis of the distribution of accuracy across the parameter space in Section 7.3.

Cfg	N	**MAE** Δv **[m/s]**	R^2	**Rel. err. [%]**	**Within ±5% [%]**
0	136	50.94	0.9740	0.76	97.1
1	134	89.07	0.9344	1.30	91.8
2	139	59.59	0.9692	0.89	97.8
3	135	76.35	0.9535	1.15	96.3
4	149	133.44	0.8182	1.98	88.6
5	127	62.78	0.9711	0.91	96.9
6	145	56.60	0.9678	0.85	97.2
7	151	96.73	0.9257	1.40	90.7
8	1,394	96.82	0.8986	1.39	92.7
9	1,605	219.27	0.7122	3.36	76.3
10	1,313	98.16	0.8601	1.38	93.0

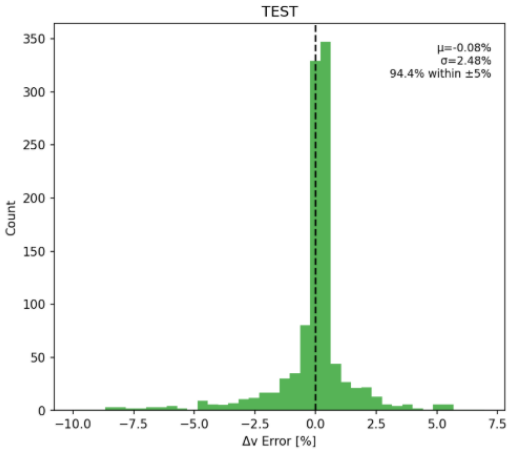
Table 7.5: Per-configuration performance of DvNet. The upper block reports the eight training configurations (in-distribution test set), the lower block the three held-out ones



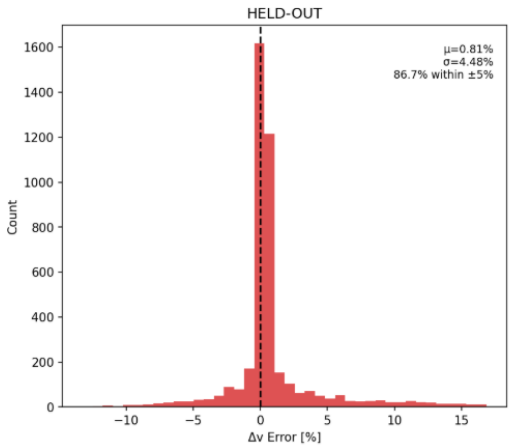
(a) Predicted vs solver Δv, test set



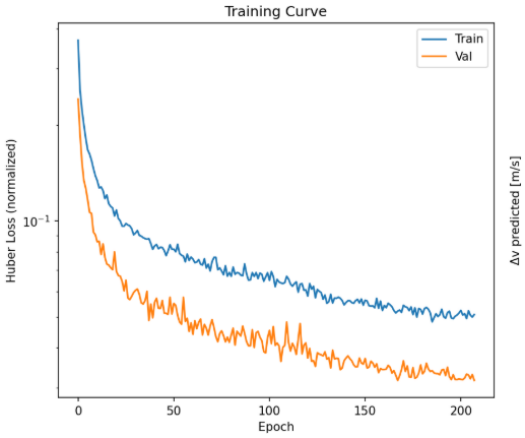
(b) Predicted vs solver Δv, held-out



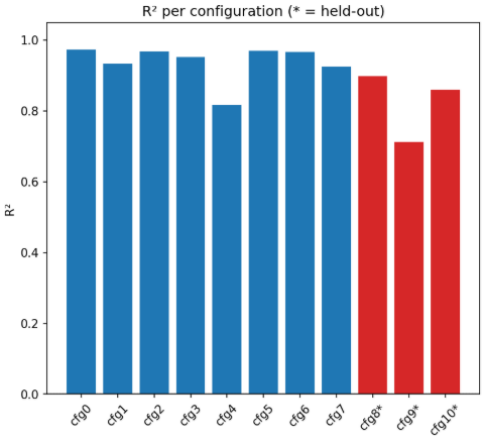
(c) Relative Δv error, test set



(d) Relative Δv error, held-out



(e) Training and validation Huber loss



(f) Per-configuration R^2

Figure 7.2: Overview of the DvNet evaluation

7.2.3 Generalization to Held-Out Configurations

On the three held-out configurations (4,312 feasible points), the performance degrades with respect to the in-distribution case, as expected for the estimation of a continuous quantity on engines never seen during training: $R^2 = 0.8183$, MAE 142.81 m/s, RMSE 318.76 m/s, mean relative error 2.12%, and 86.7% of the predictions within ±5% (Table 7.6). The error on the final mass m_f remains contained, with a MAE of 5.89 kg and a mean relative error of 0.55%, again a fraction of the corresponding Δv error owing to the attenuating effect of the Tsiolkovsky relation. The larger scatter is apparent in the dispersion plots and in the error histograms of Figures 7.2b, 7.2d. The aggregate figure is nonetheless dominated by configuration 9 ($R^2 = 0.712$), markedly harder than the other two which remain closer to the training values (R^2 of 0.899 and 0.860). This disparity between configurations equally excluded from training reflects their position in the parameter space, and is taken up, together with configuration 4, in Section 7.3.

	In-distribution (test)	**Held-out**
Δv [m/s]		
MAE	78.96	142.81
RMSE	177.84	318.76
R^2	0.9388	0.8183
Real range	[5643.3, 8997.4]	[5631.1, 8989.3]
Mean rel. error [%]	1.17	2.12
Within ±5% [%]	94.4	86.7
Final mass m_f [kg]		
MAE	3.85	5.89
Real range	[414.0, 3104.9]	[360.4, 1603.7]
Mean rel. error [%]	0.29	0.55

Table 7.6: Overall performance of DvNet on the Δv and on the derived final mass m_f, for the in-distribution test set and the held-out configurations

7.3 Cascade Pipeline

The end-to-end evaluation of the pipeline follows the protocol of Chapter 6, on the two sets kept intact (feasible and infeasible points alike), so as to assess jointly the classification gate and the regression. This section analyzes the compound error, compares the predicted porkchop plots with those of the solver, characterizes the accuracy across the parameter space, and finally quantifies the computational gain.

7.3.1 Compound Error

When the two networks operate in cascade, the overall error is not the sum of those of the networks in isolation, but emerges from their interaction: the classifier's decision determines which points reach the regressor. Table 7.7 summarizes the end-to-end outcome on the two sets. On the correctly forwarded points, the regressor attains metrics marginally better than the standalone ones (on the test set, the MAE drops from 78.96 to 75.77 m/s): the difference is exactly the false negatives, feasible points discarded by the gate which, having a high mean real Δv (8,001 m/s), belong to the population hardest to predict; their removal leaves the regressor with a marginally easier subset.

The four categories of the confusion matrix contribute differently to the output. True positives and true negatives are the correct cases; the false negatives subtract realizable regions from the map (coverage gaps), whereas the false positives introduce into it a "ghost" Δv, devoid of a converged reference trajectory (7,074 m/s on average on the test set). As anticipated in Chapter 6, the latter are of a twofold nature: either genuine errors, or cases in which the solver failed despite the existence of a solution and the network "corrects" that failure with a plausible estimate. The distinction requires the spatial inspection of the porkchop plots, and is taken up in Section 7.3.3.

On the held-out set the picture is analogous but degraded, and dominated by configuration 9, which exhibits a profile opposite to the others: an anomalously conservative gate (few false positives, many false negatives) and by far the least accurate regression. It is therefore the most challenging on both fronts, consistently with what was already observed.

	Aggregate		Held-out per config		
	Test	**Held-out**	**8**	**9**	**10**
Classification gate					
TP	1,106	4,216	1,390	1,520	1,306
TN	641	2,347	796	659	892
FP	63	166	85	11	70
FN	10	96	4	85	7
Accuracy	0.9599	0.9616	0.9609	0.9578	0.9662
Regression on forwarded points (TP)					
MAE Δv [m/s]	75.77	132.86	95.64	200.40	93.88
RMSE [m/s]	172.12	303.84	232.87	375.44	276.12
Rel. err. [%]	1.13	1.99	1.38	3.13	1.33
Within $\pm 5\%$ [%]	94.8	87.8	92.9	78.4	93.3
m_f rel. err. [%]	0.274	0.507	0.307	0.778	0.404

Table 7.7: End-to-end performance of the cascade pipeline, on the in-distribution test set and the held-out configurations (aggregate and per configuration). Regression metrics are computed on the correctly forwarded points (TP)

7.3.2 Pointwise Inference

As a direct illustration of the behavior of the pipeline on the unseen configurations, Table 7.8 compares the predicted Δv with that of the solver point by point, on a random sample of held-out points, ordered in the table by increasing error. The picture confirms what emerged from the aggregate metrics: on the low-Δv transfers (those closest to the cost minimum, and hence of greatest interest for the identification of the launch windows) the network is extremely reliable, with errors typically below 1% on all three configurations, configuration 9 included. The significant deviations are instead confined exclusively to high Δv and, predominantly but not exclusively on configuration 9, where a few points reach errors of 11–22%. It is these rare outliers that dominate the aggregate metrics of the configuration, depressing its R^2 and mean error well beyond what the quality on the points of interest would suggest. The network, in other words, remains accurate where it matters, and loses precision in the high-cost region, the farthest from the optimum and the least relevant in practice.

Cfg	t_0 **[MJD2000]**	**TOF [days]**	$\Delta v_{\mathbf{real}}$ **[m/s]**	$\Delta v_{\mathbf{pred}}$ **[m/s]**	**Err. [%]**	$m_{f,\mathbf{real}}$ **[kg]**	$m_{f,\mathbf{pred}}$ **[kg]**
8	12020	691	5692.2	5692.5	+0.01	400.37	400.37
8	12646	712	5691.1	5693.6	+0.04	400.39	400.36
10	12455	926	5701.3	5705.8	+0.08	510.16	510.07
10	11883	1050	5703.6	5690.2	−0.24	510.11	510.41
9	11040	715	5669.7	5687.0	+0.31	1601.25	1600.16
9	11257	756	5670.2	5692.8	+0.40	1601.21	1599.80
9	11856	900	5662.4	5690.2	+0.49	1601.70	1599.96
9	12700	612	5659.1	5688.2	+0.51	1601.91	1600.09
10	12183	1029	6463.4	6579.2	+1.79	493.91	491.49
10	12428	926	5925.7	5783.6	−2.40	505.32	508.38
8	11802	547	7150.4	6927.8	−3.11	382.20	384.92
9	12264	735	7295.1	7585.5	+3.98	1502.35	1485.34
8	11557	835	6267.4	5931.5	−5.36	393.10	397.33
10	11394	906	8967.2	8362.1	−6.75	444.07	455.63
10	11502	844	8456.2	7516.7	−11.11	453.81	472.29
9	11448	571	7447.6	8413.7	+12.97	1493.39	1437.87
9	11339	262	6784.9	7751.4	+14.24	1532.72	1475.71
9	11394	406	7024.3	8115.4	+15.53	1518.39	1454.79
8	12128	876	8409.1	6759.8	−19.61	367.17	386.98
9	10958	550	6350.4	7742.5	+21.92	1559.07	1476.22

Table 7.8: Pointwise comparison between predicted and solver Δv on a sample of held-out points

7.3.3 Porkchop Plots

The most direct analysis of the pipeline in its intended use consists in the generation of the predicted porkchop plots and in their comparison with those produced by the direct solver. For conciseness, a representative subset is reported here: configurations 1 and 4 among the training ones, and all three held-out configurations (8, 9, 10), sufficient to illustrate both the overall fidelity of the surrogate and the behaviors most significant for the discussion. Each figure places side by side the solver porkchop, the one predicted by the pipeline, and the map of the pointwise relative error (Figures 7.3, 7.4, 7.5, 7.6, 7.7).

The comparison confirms first of all the geometric fidelity of the surrogate: in every configuration the predicted porkchop accurately reproduces the structure of the transfer cost, the position and shape of the most favorable launch windows, and the periodic pattern tied to the synodic recurrence, with an agreement that holds even on the never-seen configurations.

It is worth emphasizing the resolution of the predicted porkchops. Whereas the solver grid comprises 2,275 nodes per configuration, the grid on which the pipeline is evaluated adopts a step of a single day in both the (t_0, t_{of}) directions, for a total of 1,219,400 points: the network thus generates over two orders of magnitude more evaluations (about 500 times as many) in a fraction of the time. Since the vast majority of these points do not belong to the original dataset grid, a dedicated off-grid inference was carried out on the reliability of the pipeline (precisely at the points farthest from the training nodes, those where interpolation is most demanding) on a sample of these five configurations. The results are positive: the median relative errors remain contained: 0.49%, 0.79%, 0.66%, and 0.71% for configurations 1, 4, 8, and 10, and 1.65% for configuration 9, in line with those of the inference on the grid nodes reported in the preceding sections (and reported here in the figures of the error maps). The densification of the grid therefore does not compromise accuracy, and the pipeline interpolates reliably even far from the nodes seen during training; even configuration 9, the most critical, retains off-grid a median well below its own aggregate mean error: further evidence that its metrics are driven by a few outliers rather than by a widespread difficulty.

Another aspect concerns the nature of the predicted porkchops relative to those of the solver. The solver maps present a granular appearance dotted with empty cells (in white), including isolated pixels located within otherwise feasible regions: points at which the solver did not reach convergence despite lying in a regular neighborhood. The predicted porkchop, by contrast, is smooth and continuous, and such internal gaps are filled with Δv values consistent with the surrounding context. This is the visual manifestation of the behavior anticipated in Chapter 6 and taken up in the discussion of the compound error: a portion of the false positives does not constitute an error, but the network's correction of isolated solver failures,

with a regularizing effect with respect to the artificial discontinuities of the dataset. Configuration 1 offers the clearest example. It remains understood that this reading concerns the false positives internal to feasible regions; nothing excludes that others, located in genuinely solution-free areas, represent true errors.

The error maps finally allow the aggregate metrics of the most difficult configurations to be correctly interpreted. Training configuration 4, which exhibited the lowest R^2 of the in-distribution set, nonetheless shows a predicted porkchop that faithfully captures its geometry, and an error map whose median is merely 0.43%: a sign that the pointwise accuracy is on the whole high and that the aggregate metrics are degraded by a minority of marked deviations, while the low-Δv transfers remain reliable. The same picture, accentuated, characterizes held-out configuration 9: despite the lowest R^2 of the entire study, the median of its error is only 0.58%, against a mean relative error of 3.36% reported in Table 7.5. The gap between median and mean is the unmistakable signature of a distribution dominated by the tails: the majority of the points is predicted with minimal errors, and it is a few outliers that shift the mean. The error map confirms this spatially: the deviations vanish almost everywhere in the low-Δv regions and concentrate instead along the high-Δv ridges, the farthest from the cost optimum. It should moreover be noted that the color scale is truncated at $\pm 10\%$: the darkest points thus reach that limit but may exceed it substantially, as already seen in the pointwise inference of Section 7.3.2, where some deviations surpassed 20%. The overall reading is consistent across all configurations: the surrogate faithfully reproduces the structure of the porkchop plots and is accurate where it matters most, namely in the low-cost regions that host the launch windows, while the error concentrates in the high-Δv region, the least relevant in practice.

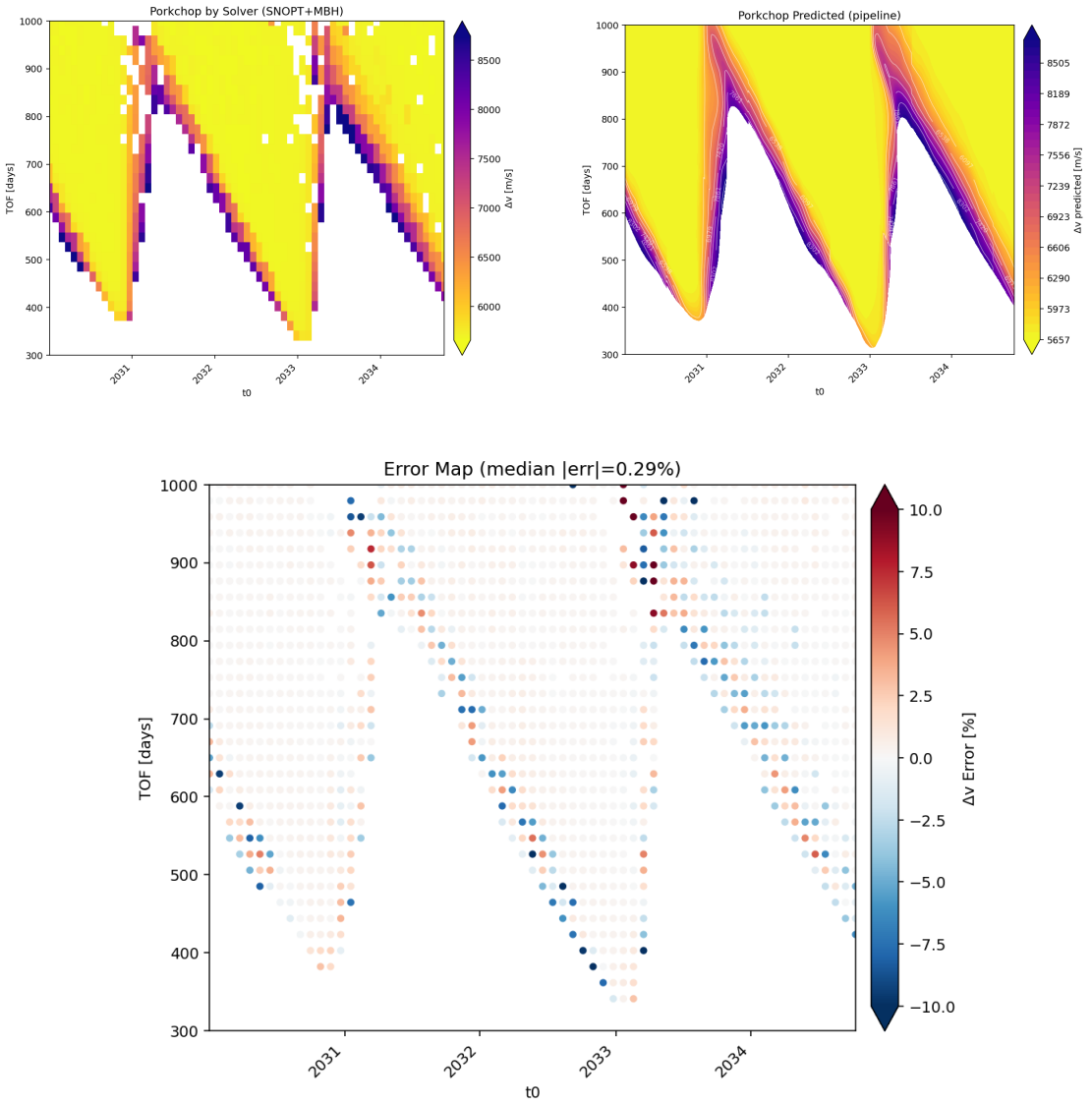


Figure 7.3: Configuration 1 (training): solver porkchop, predicted porkchop, and relative error map

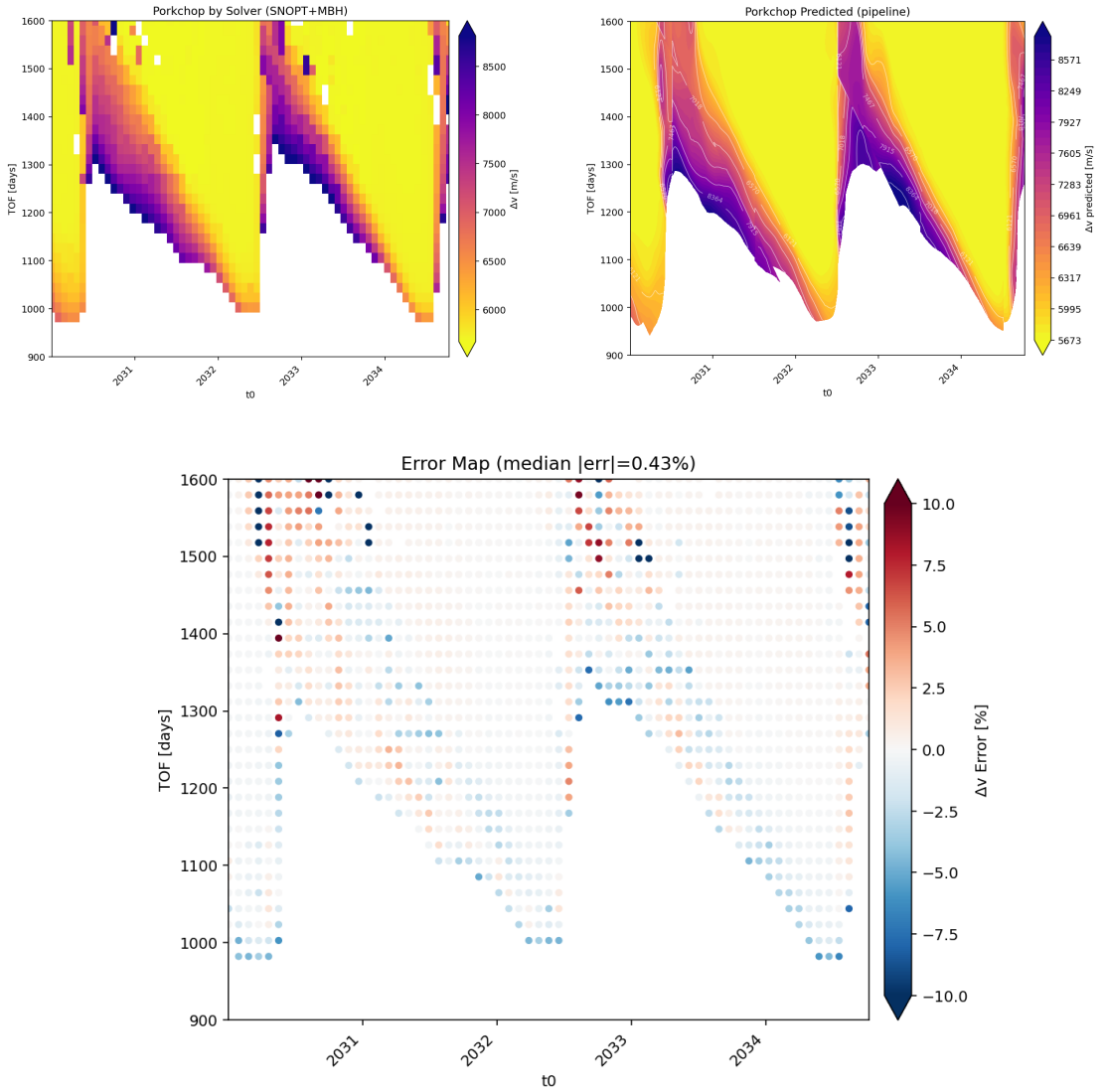


Figure 7.4: Configuration 4 (training): solver porkchop, predicted porkchop, and relative error map

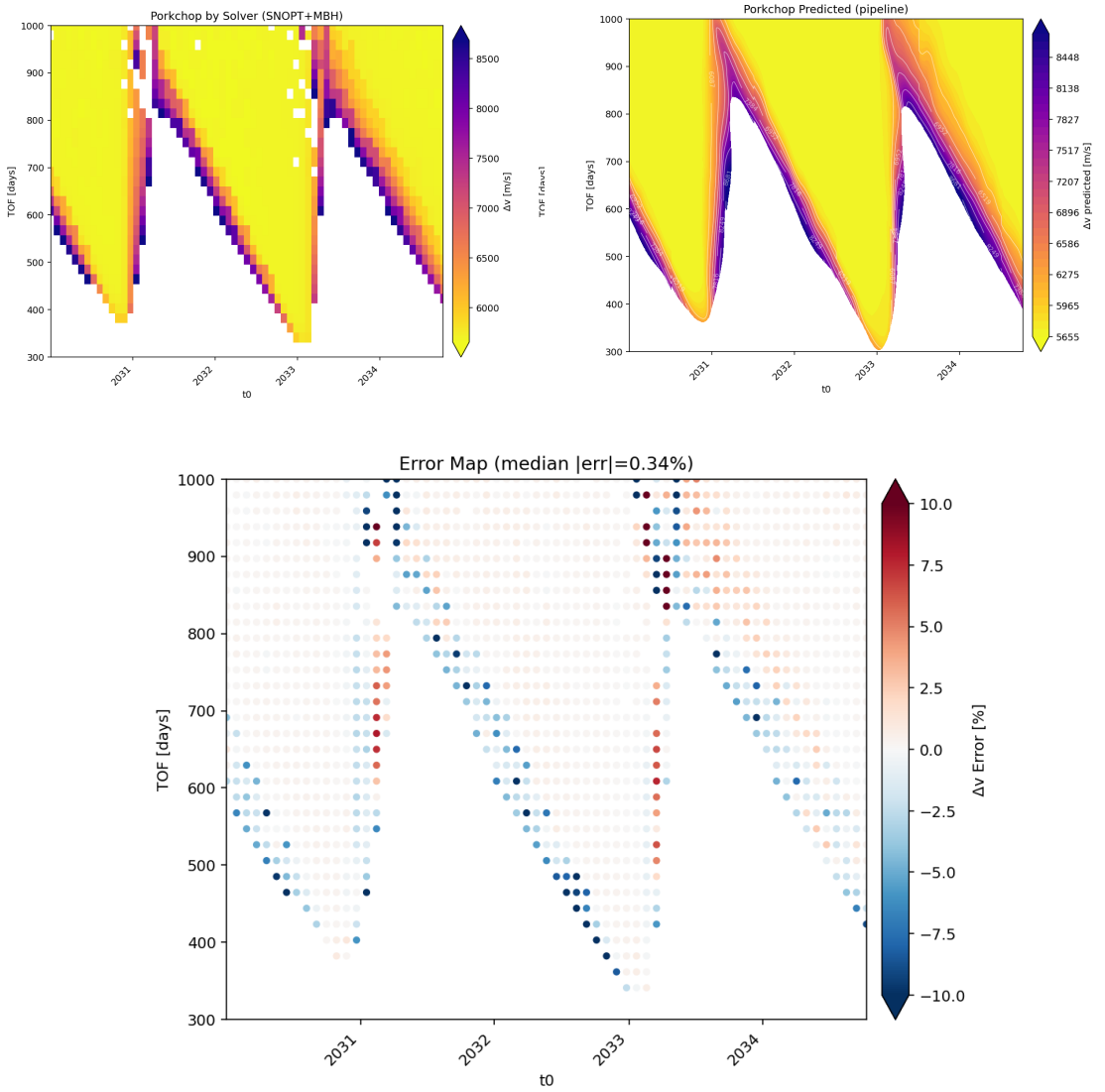


Figure 7.5: Configuration 8 (held-out): solver porkchop, predicted porkchop, and relative error map

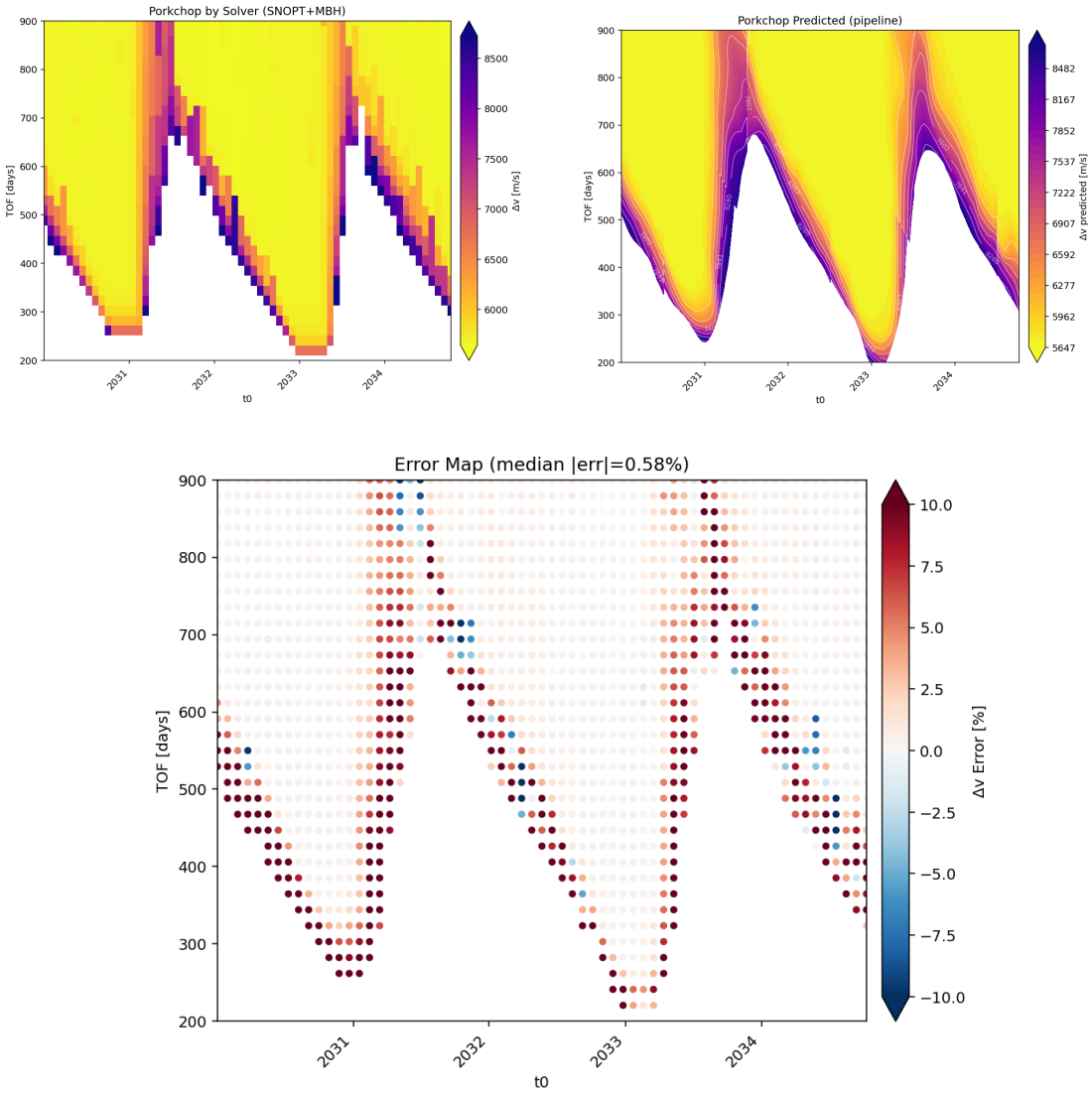


Figure 7.6: Configuration 9 (held-out): solver porkchop, predicted porkchop, and relative error map

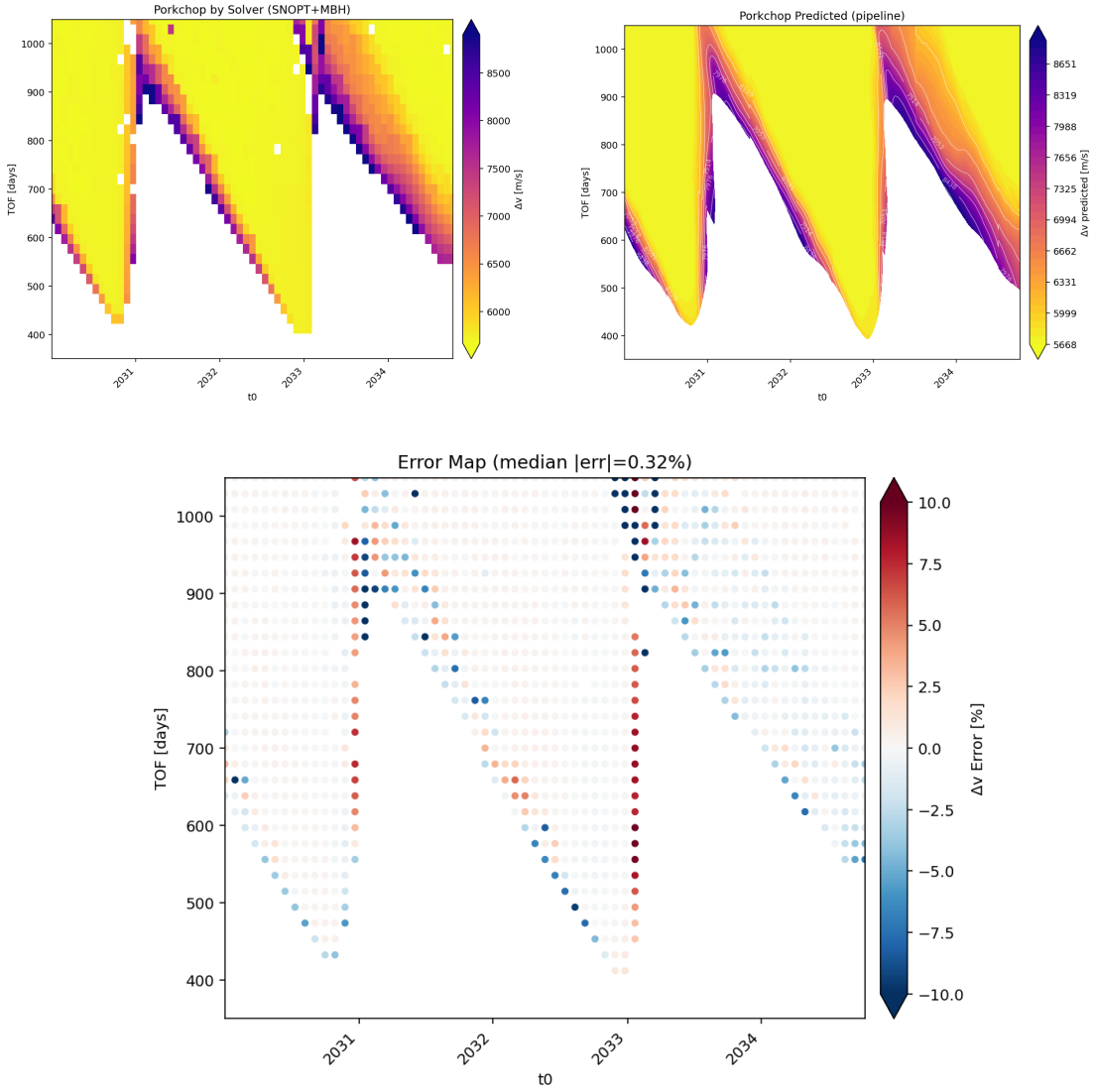


Figure 7.7: Configuration 10 (held-out): solver porkchop, predicted porkchop, and relative error map

7.3.4 Accuracy and Computational Cost

The advantage that constitutes the very rationale of the surrogate is the drastic reduction of computation times against a high accuracy. The generation of the complete dataset (25,025 points) through the direct solver required on average about 235 seconds per point, a value dominated by the infeasible points, which never reach convergence and must run through all the iterations of the algorithm before failing (particularly with the more aggressive escalation strategy, considerably

slower); the average drops to about 67 seconds on the feasible points alone. The evaluation of the pipeline, by contrast, requires about 6×10^{-6} seconds per point: the generation of the dense porkchops at a one-day step for all eleven configurations (13.4 million points in total) was completed in about 80 seconds. The surrogate is therefore faster than the direct solver by roughly seven orders of magnitude per point (Table 7.9), while at the same time retaining the accuracy documented in the preceding sections — relative errors typically below 1–2% in the regions of interest. To this is added the cost of training the two networks, contained within about 30 seconds in total: a speed favored by the relatively modest size of the dataset and by the normalization and regularization techniques adopted, which accelerate convergence.

Direct solver [s/point]	**Surrogate [s/point]**	**Speed-up**
235	6×10^{-6}	$\sim 4 \times 10^{7}$

Table 7.9: Average per-point computational cost of the direct solver and of the surrogate pipeline, with the resulting speed-up

7.3.5 Coverage Map

Figure 7.8 arranges the eleven configurations in the engine parameter plane (a_0, c), colored according to the regressor's R^2 and distinguishing the eight training configurations (circles) from the three held-out ones (stars). The two held-out configurations 8 and 10, in interior positions surrounded by training configurations, are interpolated accurately (R^2 of 0.899 and 0.860), confirming that where the learned relation is well constrained by the surrounding data, generalization to an unseen engine occurs with no appreciable loss. Held-out configuration 9 constitutes instead the most critical case ($R^2 = 0.712$): its location was deliberately chosen in a region less densely covered by the training configurations, so as to subject the interpolating capability of the surrogate to the most severe test; it is there, where the training data are sparsest, that interpolation predictably becomes most uncertain.

An analogous profile, though more contained, is observed among the training configurations: configuration 4 is the least accurate of the in-distribution set ($R^2 = 0.818$). It should be stressed that mere proximity to the edge of the domain does not constitute a sufficient explanation (training configuration 7, though located at the opposite extreme, is predicted with good accuracy ($R^2 = 0.926$)) and that any interpretation in this respect remains, at present, hypothetical. Two elements nonetheless make the greater difficulty of configuration 4 plausible. First, it is the most distant from the other training configurations, which leaves the learned

relation in that region less constrained by neighboring data. Second, configuration 4 belongs to the lowest-thrust regime: it is plausible that in such a regime, in which the vehicle operates with a reduced control margin, the dependence of the Δv on the transfer geometry exhibits sharper gradients and a less regular structure, and is therefore harder for the network to approximate. The map thus offers an overall reading of the link between coverage density, engine regime, and prediction quality, framing the most critical configurations not as anomalies, but as outcomes consistent with their position in the parameter space.

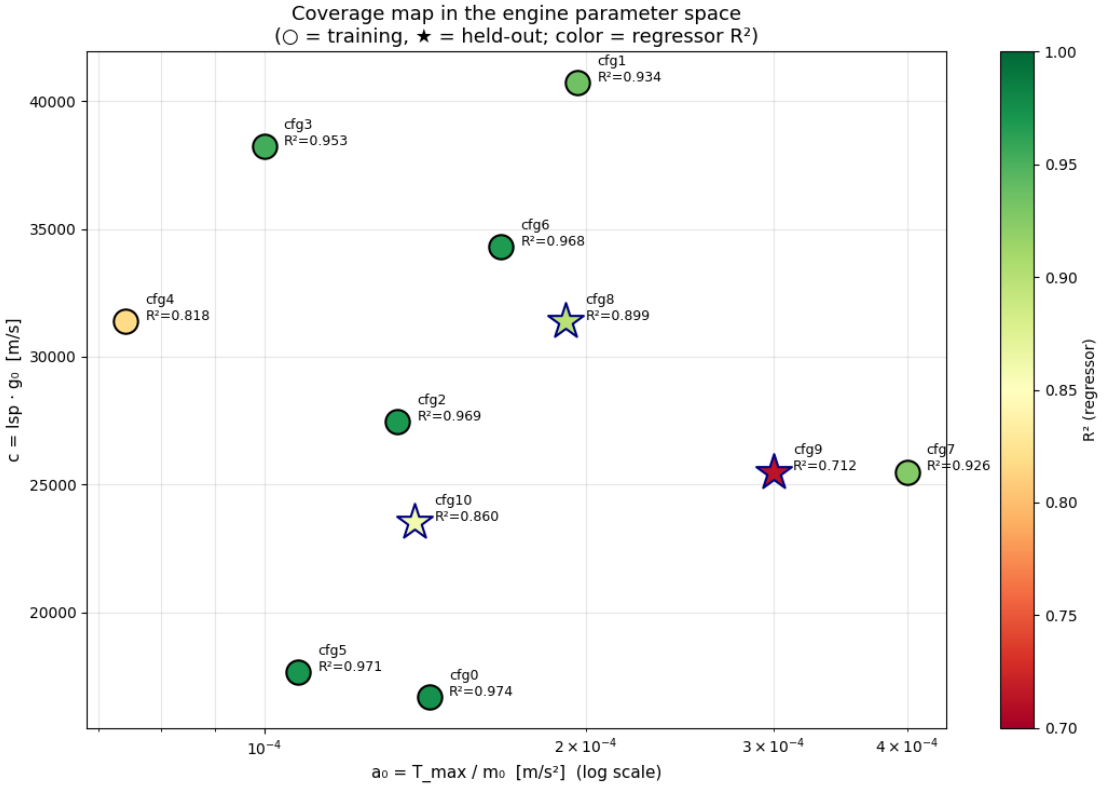


Figure 7.8: Coverage map in the engine parameter space (a_0, c)

7.4 Complementary Analyses

The analyses presented so far have characterized the surrogate in its intended use. This section supplements them with a number of complementary ones, aimed at probing its robustness beyond the nominal conditions and at verifying, a posteriori, some of the design choices adopted.

7.4.1 Robustness to Initialization

To ascertain that the reported performance does not depend on a particular fortunate initialization of the weights, both networks were retrained with ten different seeds of the random number generator, all other conditions being equal. Figure 7.9 summarizes the outcome. The variability of the results is minimal: for FeasibilityNet the test AUC remains within the interval $[0.9911,\ 0.9934]$ and the held-out AUC within $[0.9935,\ 0.9945]$, with standard deviations of the order of 10^{-4}; for DvNet the test R^2 varies within $[0.935,\ 0.945]$ and the held-out R^2 within $[0.810,\ 0.833]$ ($\sigma = 0.003$ and 0.007 respectively). The width of these oscillations is negligible relative to the values at play and, above all, far smaller than the gaps between configurations discussed in the preceding sections, confirming that the performance of the surrogate is a stable property of the model and not the effect of a single favorable realization. It may be noted, incidentally, that the seed employed for the results presented in this chapter is not the one yielding the best performance (indeed it ranks among the least favorable of the set) albeit by an entirely marginal margin.

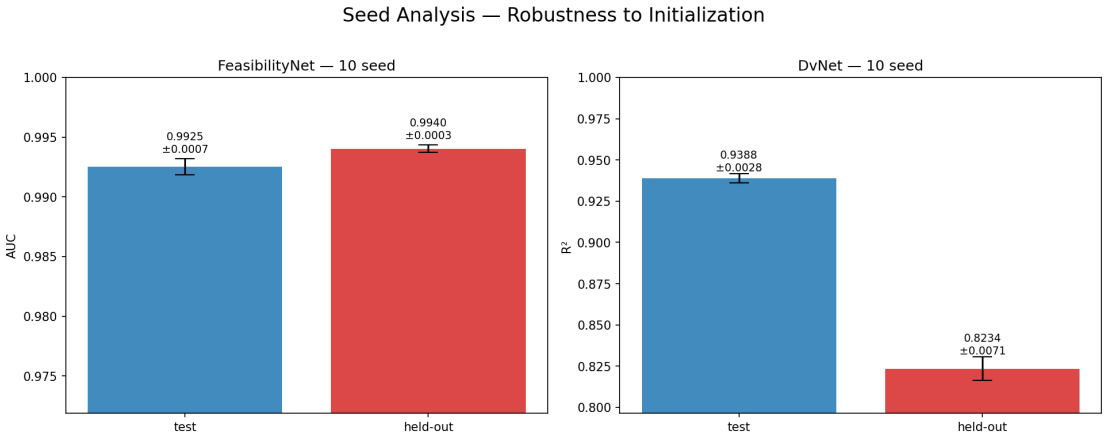


Figure 7.9: Robustness to weight initialization: mean and standard deviation of the principal metrics over ten training seeds

7.4.2 Learning Curves versus Dataset Size

This analysis investigates how the performance depends on the amount of training data. The evaluation was conducted on the regressor alone, as it is charged with the harder task (the estimation of a continuous quantity) and is the one that, as seen, exhibits the widest margins for improvement relative to the classifier. Starting from the frozen split, the network was retrained on increasing fractions of the

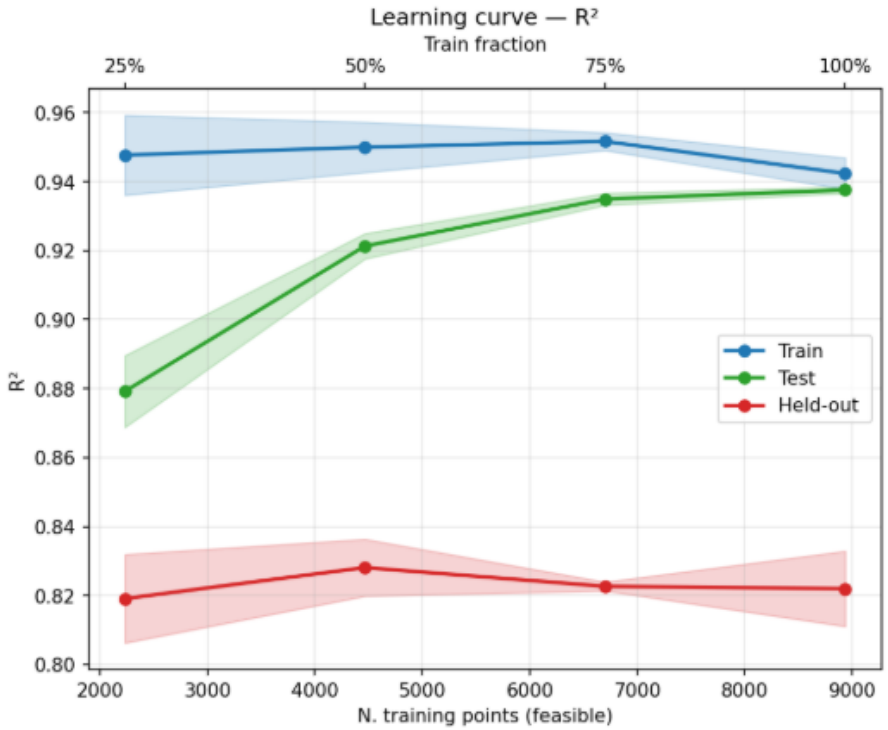
feasible training points (25%, 50%, 75%, 100%, from 2,233 to 8,935 trajectories), evaluating it each time on the fixed test and held-out sets; for robustness, each point was obtained as the mean over three seeds, whose dispersion is shown by the bands in Figure 7.10. The first two panels of the figure also plot the curve relating to the training set, absent from the third: the fraction within ±5% is in fact a measure of generalization, of no interest on the very data the network is optimized on, whereas R^2 and MAE are reported on the training set as well, for reference. It may be noted that the gap between the training and test curves, an indicator of overfitting, progressively narrows as the data increase, nearly vanishing at 100%: a sign that the abundance of data mitigates overfitting and improves generalization within the seen configurations.

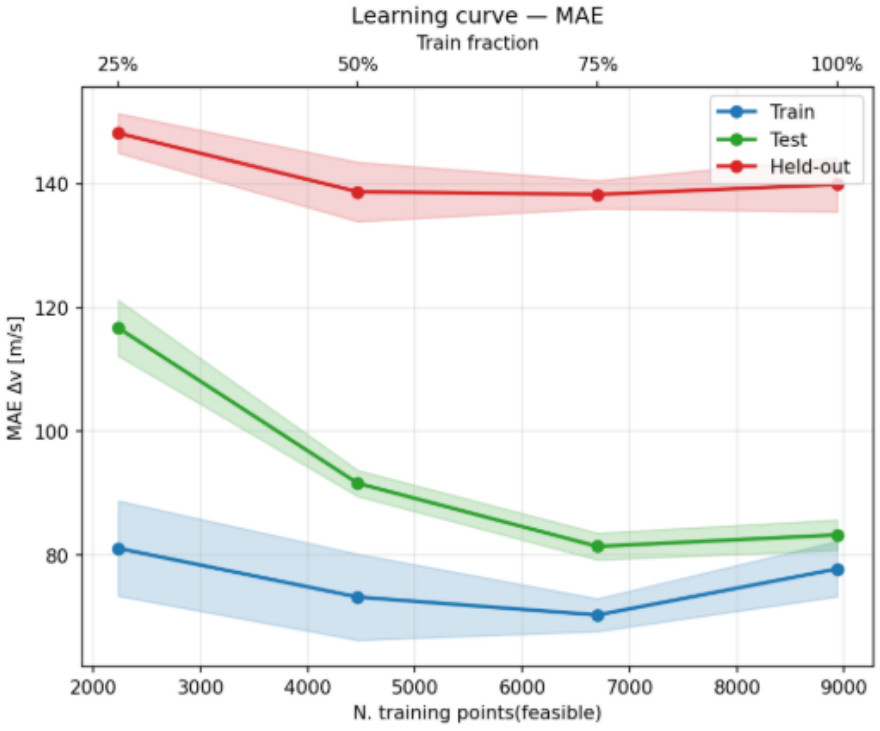
The data reveal two distinct behaviors. On the seen configurations (test), the metrics improve markedly up to 75% of the data — the R^2 rises from 0.879 to 0.935, the MAE falls from 116.7 to 81.4 m/s, and the fraction within ±5% from 89.9% to 94.7% — and then flatten, with negligible gains in passing from 75% to 100%, within the statistical noise across seeds. On the held-out configurations, by contrast, the metrics are nearly constant over the entire interval, with an R^2 stable around 0.82 and oscillations entirely contained within the respective standard deviations.

It follows that, within the explored range, the addition of further training points would not yield a sensible improvement: the network has reached a saturation. The flatness of the held-out curves, in particular, confirms an aspect that emerged repeatedly in this chapter. The data added across the various fractions are supplementary (t_0, t_{of}) points relating to the already seen configurations; the fact that their increase does not alter the generalization to the never-seen configurations indicates that the latter is not limited by the sampling density per configuration, but by the coverage of the engine parameter space. Improving the generalization to new engines would therefore benefit not from more points on the existing configurations, but from additional configurations densifying the (a_0, c) space, in full consistency with what was observed in the coverage map.

Fraction	$n_{\mathbf{train}}$	R^2		**MAE [m/s]**		**Within ±5% [%]**
		Test	**Held-out**	**Test**	**Held-out**	**Test / Held-out**
25%	2,233	0.879	0.819	116.7	148.2	89.9 / 85.8
50%	4,467	0.921	0.828	91.6	138.7	92.9 / 87.4
75%	6,701	0.935	0.823	81.4	138.3	94.7 / 87.4
100%	8,935	0.938	0.822	83.3	139.9	94.8 / 87.9

Table 7.10: Regressor performance as a function of the training set size, mean over three seeds


Learning curve — R²
Train fraction
25%
50%
75%
100%
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0.80
R²
Train
Test
Held-out
2000
3000
4000
5000
6000
7000
8000
9000
N. training points (feasible)


Learning curve — MAE
Train fraction
25%
50%
75%
100%
140
120
100
80
MAE Δv [m/s]
Train
Test
Held-out
2000
3000
4000
5000
6000
7000
8000
9000
N. training points(feasible)

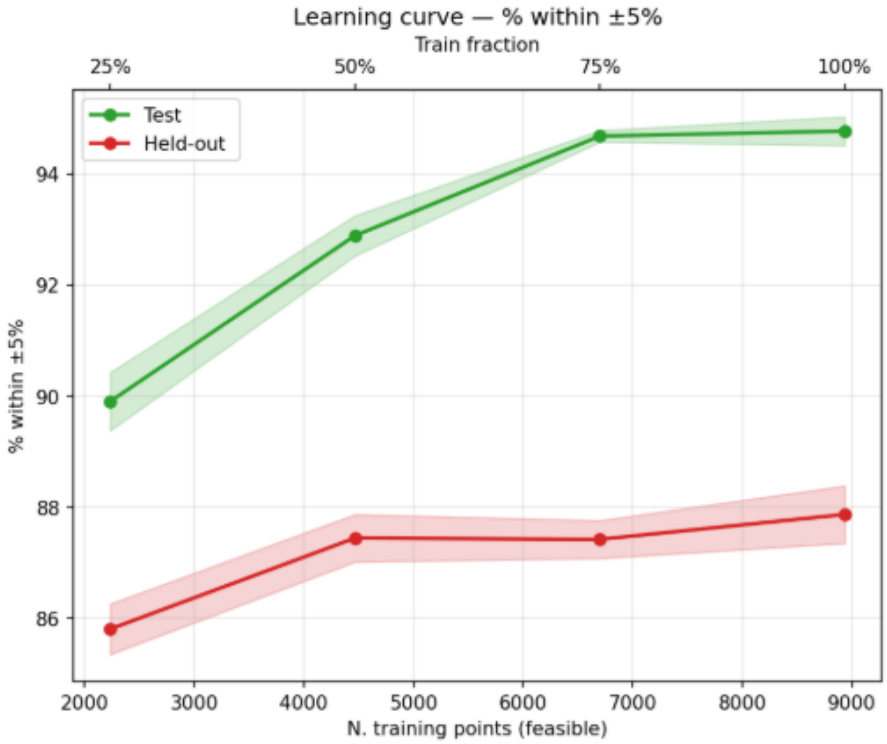


Figure 7.10: Regressor learning curves as a function of the training set size

7.4.3 Temporal Extrapolation

The most ambitious complementary analysis subjects the surrogate to the test of temporal extrapolation, that is, to prediction on departure dates outside the interval covered during training. The check was conducted on two training configurations, 1 and 2, deliberately chosen among the seen ones rather than the held-out ones: the task was already arduous in itself, and adopting known configurations avoids compounding it with the difficulty of generalizing to a never-seen engine. For each, the pipeline was queried on a grid of the same dimensions and step (one day) as those employed so far, but on a t_0 window of equal extent placed immediately after the last node of the training grid. The porkchop thus predicted was compared with the one generated by the solver on the same new dates, following the identical procedure adopted for the initial dataset. The objective was to ascertain whether the network retained any accuracy outside its own temporal interval.

The results, reported in Table 7.11 and illustrated in Figure 7.11 and Figure 7.12, indicate clearly that it does not. In absolute terms the metrics deteriorate markedly with respect to the nominal regime: the R^2 collapses from about 0.95 to 0.36 and 0.24 for the two configurations, the MAE increases by nearly an order of

magnitude (from 60–90 to over 450 m/s), and the fraction within ±5% drops from 95% to about 62%. The classification gate degrades as well, with the feasibility accuracy falling from the ~ 0.96 of the nominal regime to 0.84–0.88 out of range: the porkchops show the network predicting feasible regions in areas the solver leaves empty, with a consequent deterioration of the overall shape of the maps. This outcome is unsurprising and confirms a known property of neural networks: they are interpolators, not extrapolators, and outside the domain of the data seen during training their predictions lose reliability.

Two aspects nonetheless deserve to be noted. The first is that the synodic structure of the problem is still correctly captured: the predicted porkchops reproduce the periodic recurrence of the launch windows in the expected position. This is a direct consequence of the encoding adopted for the input vector: the synodic periodicity is supplied to the network through the sine and cosine components of the phase, which remain valid and bounded even beyond the temporal interval of training; the network does not have to extrapolate them, but recomputes them correctly, so that the geometric backbone of the transfer remains anchored to reliable inputs. The second aspect is that the median error remains relatively low (1.55% and 1.57%), against considerably higher means: as observed elsewhere, this indicates that in the low-Δv regions the errors remain modest, whereas they explode at high Δv, where the bulk of the deterioration concentrates.

The overall picture is unequivocal: temporal extrapolation lies outside the domain of reliable use of the surrogate. For departure dates significantly outside the training interval, the generation of a new dataset and the retraining of the network remain indispensable. This result does not contradict, but rather reinforces, what emerged earlier: the generalization of the surrogate operates by interpolation within the covered domains (both in the configuration space and in the temporal one) and not by extrapolation beyond their boundaries.

	In-range			**Extrapolation**			
Cfg	**MAE**	R^2	±5%	**MAE**	R^2	±5%	**Class. acc.**
1	89.07	0.934	91.8	462.76	0.240	61.6	0.877
2	59.59	0.969	97.8	456.55	0.363	63.6	0.844

Table 7.11: Regressor performance in the nominal range and under temporal extrapolation, for two training configurations. MAE in m/s, fraction within ±5% in percent. The last column reports the out-of-range feasibility accuracy of the classification gate

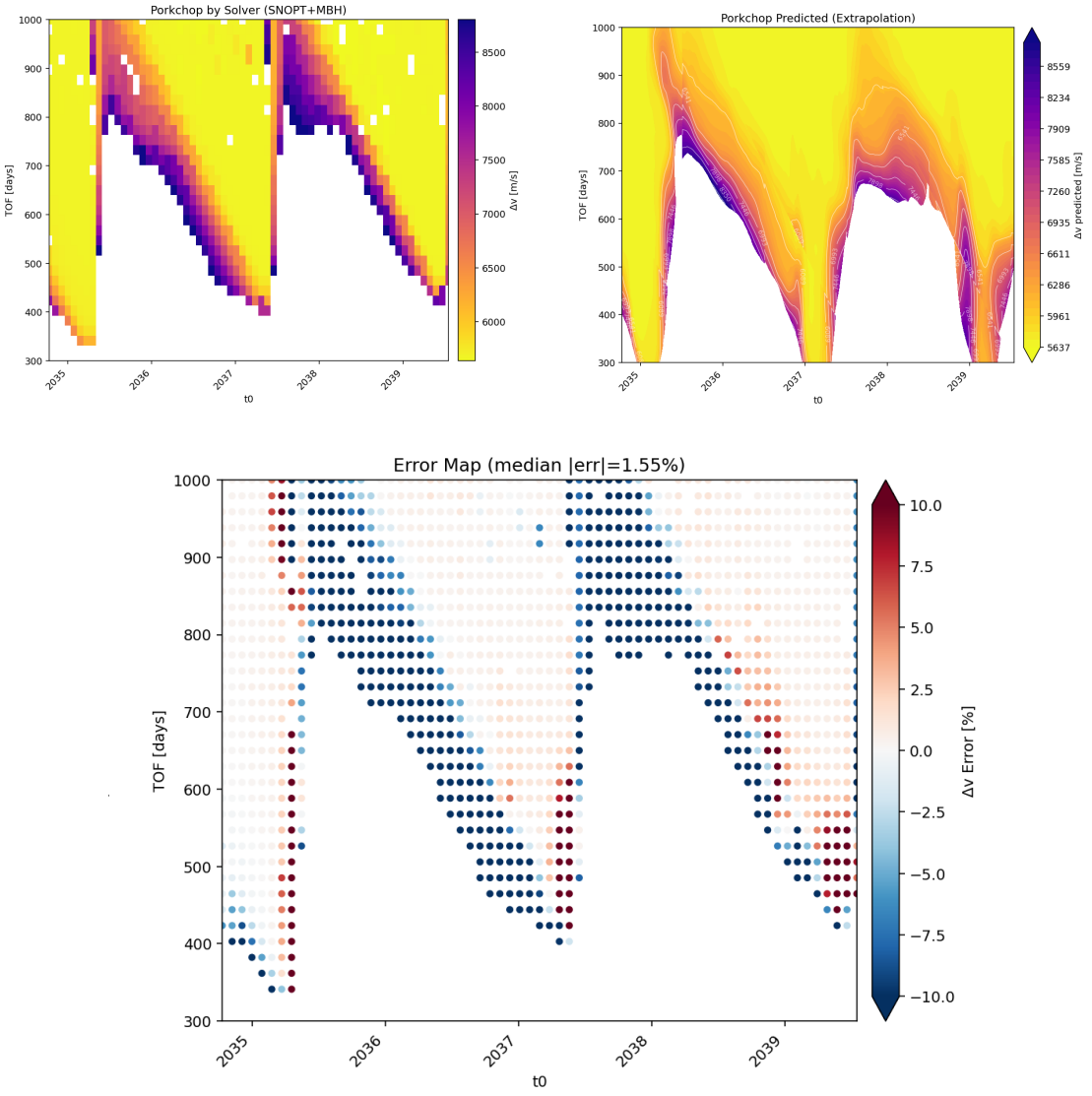


Figure 7.11: Temporal extrapolation, configuration 1: solver porkchop, predicted porkchop, and relative error map

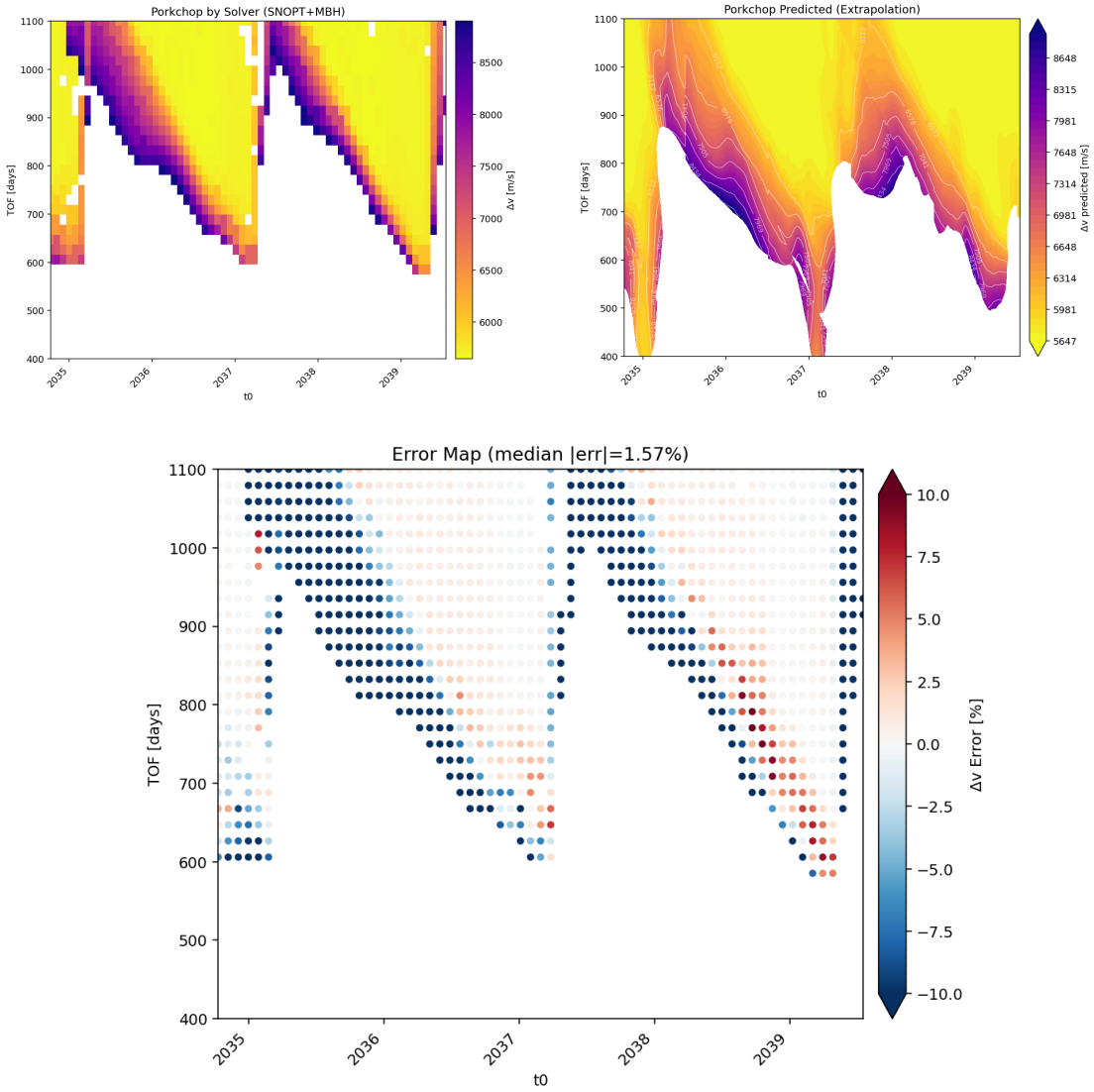


Figure 7.12: Temporal extrapolation, configuration 2: solver porkchop, predicted porkchop, and relative error map

7.4.4 Comparison of Architectures and Input Vectors

Architectures. As anticipated, the architectures of the two networks were determined empirically, by successive trials, comparing configurations of different depth and width. The trials showed that architectures both larger and smaller than the adopted ones worsened the performance: the former tended toward overfitting and lengthened the computation without benefit, the latter lacked the capacity to represent the relation. The chosen architectures therefore represent the best compromise found between generalization capability and computational cost. It should be noted that no exhaustive hyperparameter optimization (such as learning rate, number of neurons, number of layers, dropout rate, batch size) was pursued in search of the model yielding the best possible performance: this was not the aim of the work, which was rather the identification of an applicable and reliable method, and the results achieved with the adopted architectures proved more than satisfactory to that end.

Input vectors. Of greater interest is the comparison between different compositions of the input vector, aimed at verifying the actual usefulness of the choices made in its definition, in particular the redundancy of correlated features and the adoption of derived rather than raw engine parameters. In addition to the official vector $\mathbf{x}_{\text{in}}$ described in Chapter 6, four variants were trained and evaluated, obtained by adding or removing groups of features:

$$\mathbf{x}_{\text{in},1} = \left[\, t_{of},\ \mathbf{r}_1,\ \mathbf{r}_2,\ \mathbf{v}_1,\ \mathbf{v}_2,\ \log(a_0),\ c \,\right]^{\mathsf{T}} \tag{7.1}$$

$$\mathbf{x}_{\text{in},2} = \left[\, t_{of},\ \mathbf{r}_1,\ \nu_E,\ \mathbf{r}_2,\ \nu_M,\ \phi,\ \sin\psi,\ \cos\psi,\ \mathbf{v}_1,\ \mathbf{v}_2,\ T_{\max},\ I_{sp},\ m_0 \,\right]^{\mathsf{T}} \tag{7.2}$$

$$\mathbf{x}_{\text{in},3} = \left[\, t_{of},\ \nu_E,\ \nu_M,\ \phi,\ \sin\psi,\ \cos\psi,\ \log(a_0),\ c \,\right]^{\mathsf{T}} \tag{7.3}$$

$$\mathbf{x}_{\text{in},4} = \left[\, t_{of},\ \mathbf{r}_1,\ \nu_E,\ \mathbf{r}_2,\ \nu_M,\ \phi,\ \mathbf{v}_1,\ \mathbf{v}_2,\ \log(a_0),\ c \,\right]^{\mathsf{T}} \tag{7.4}$$

The vector $\mathbf{x}_{\text{in},1}$ retains only the state vectors together with the derived engine parameters, discarding the anomalies, the phase angle, and the synodic encoding; $\mathbf{x}_{\text{in},2}$ is identical to the official vector in its geometric part, but supplies the engine parameters in raw form $(T_{\max}, I_{sp}, m_0)$ in place of $\log(a_0)$ and c; $\mathbf{x}_{\text{in},3}$ is a minimal vector comprising only the angular and temporal quantities with the derived parameters, without state vectors; and $\mathbf{x}_{\text{in},4}$ is identical to the official one but deprived of the sine and cosine components of the synodic phase. The results obtained with each vector are summarized in Table 7.12, through the classifier accuracy and the regressor R^2, on the test set and on the held-out configurations.

On the test set all the vectors lead to substantially equivalent performance, with classifier accuracies between 95% and 96% and regressor R^2 close to 0.94. It is

Input vector	# feat.	Classifier acc.		Regressor R^2	
		Test	Held-out	Test	Held-out
$\mathbf{x}_{\text{in}}$ (official)	20	0.960	0.962	0.939	0.818
$\mathbf{x}_{\text{in},1}$ (state only)	15	0.961	0.954	0.938	0.821
$\mathbf{x}_{\text{in},2}$ (raw params)	21	0.961	0.892	0.938	0.140
$\mathbf{x}_{\text{in},3}$ (minimal)	8	0.964	0.963	0.937	0.836
$\mathbf{x}_{\text{in},4}$ (no synodic)	18	0.953	0.954	0.938	0.798

Table 7.12: End-to-end performance of the pipeline with the different input vector compositions

on the held-out configurations that the differences emerge clearly, and from them precise conclusions can be drawn.

The most decisive factor is the choice between derived and raw engine parameters. The vector $\mathbf{x}_{\text{in},2}$, which employs the raw $(T_{\max}, I_{sp}, m_0)$, causes a vertical collapse of the generalization: the held-out regressor R^2 collapses to 0.14, against the 0.82 of the official vector. The per-configuration analysis reveals that the collapse is almost entirely attributable to configuration 9, whose R^2 falls to -0.96, while configurations 8 and 10 remain at plausible values (0.70 and 0.85). This is not anomalous behavior, but the most eloquent experimental confirmation of the choice to adopt the derived parameters, motivated in Chapter 6: supplying the raw values leaves the network to discover on its own that the dynamics are governed by the ratio $a_0 = T_{\max}/m_0$, and not by the three absolute values individually. Configuration 9, isolated and never seen, occupies in the raw space a region not covered by the training set, and in the absence of the scale normalization performed by a_0 the network fails to map it; adopting instead $(\log(a_0), c)$, the same configuration falls at a point interpretable by interpolation, and the R^2 rises back to 0.71. The raw vector thus causes the collapse of precisely the configuration most sensitive to the absolute scale, demonstrating directly the necessity of the transformation.

The second relevant factor is the synodic encoding. Its removal ($\mathbf{x}_{\text{in},4}$) lowers the held-out R^2 from 0.82 to 0.80: a real, though modest, effect that confirms the contribution of the periodic representation of time to generalization.

There finally remains the question from which the analysis set out: the usefulness of the redundancy of correlated features. On this the data are clear and redundancy does not improve the performance. The minimal eight-feature vector $\mathbf{x}_{\text{in},3}$ ($R^2 = 0.836$) and the fifteen-feature vector $\mathbf{x}_{\text{in},1}$ ($R^2 = 0.821$) equal or surpass the official twenty-feature vector ($R^2 = 0.818$): the addition of the redundant features, far from favoring learning, leaves the performance substantially unchanged. The official vector is not, in absolute terms, the one that maximizes the metrics.

Although the official vector is not the one that maximizes performance, it nonetheless represents a reliable solution. The empirical trials indicate that, in this case, the redundancy of correlated features brings no benefit to the network. What the analysis clearly demonstrates is that the quality of generalization is determined not by the number of features, but by their nature: it is the physically motivated transformations (the adoption of $\log(a_0)$ and c and the periodic encoding of time) that make the difference, not the multiplicity of correlated parameters.

Effect of the input vector on extrapolation. The outcome of the comparison between vectors suggests a further check: repeating the temporal extrapolation using, in place of the official vector, the minimal eight-feature vector $\mathbf{x}_{\text{in},3}$. The results, reported in Figure 7.13 and Figure 7.14, are appreciably better: the extrapolation R^2 rises from 0.24 to 0.56 for configuration 1 and from 0.36 to 0.65 for configuration 2, the median error falls below 1.2%, and the predicted porkchops appear cleaner and more regular. A plausible explanation is consistent with what was just observed: the minimal vector is composed almost exclusively of angular and periodic quantities (anomalies, phase angle, synodic encoding), which presumably remain better conditioned even beyond the temporal interval of training, whereas the official vector also includes the absolute Cartesian components of position and velocity, whose values might progressively leave the seen domain as one moves further in time. It could therefore be that, by reducing the number of features subject to extrapolation, the minimal vector attenuates the degradation. It remains understood that this is a relative improvement: the absolute performance stays modest and the earlier conclusion does not change, but the observation is of interest, as it might indicate that the composition of the input influences not only the nominal accuracy, but also the behavior of the surrogate at the boundaries of its domain.

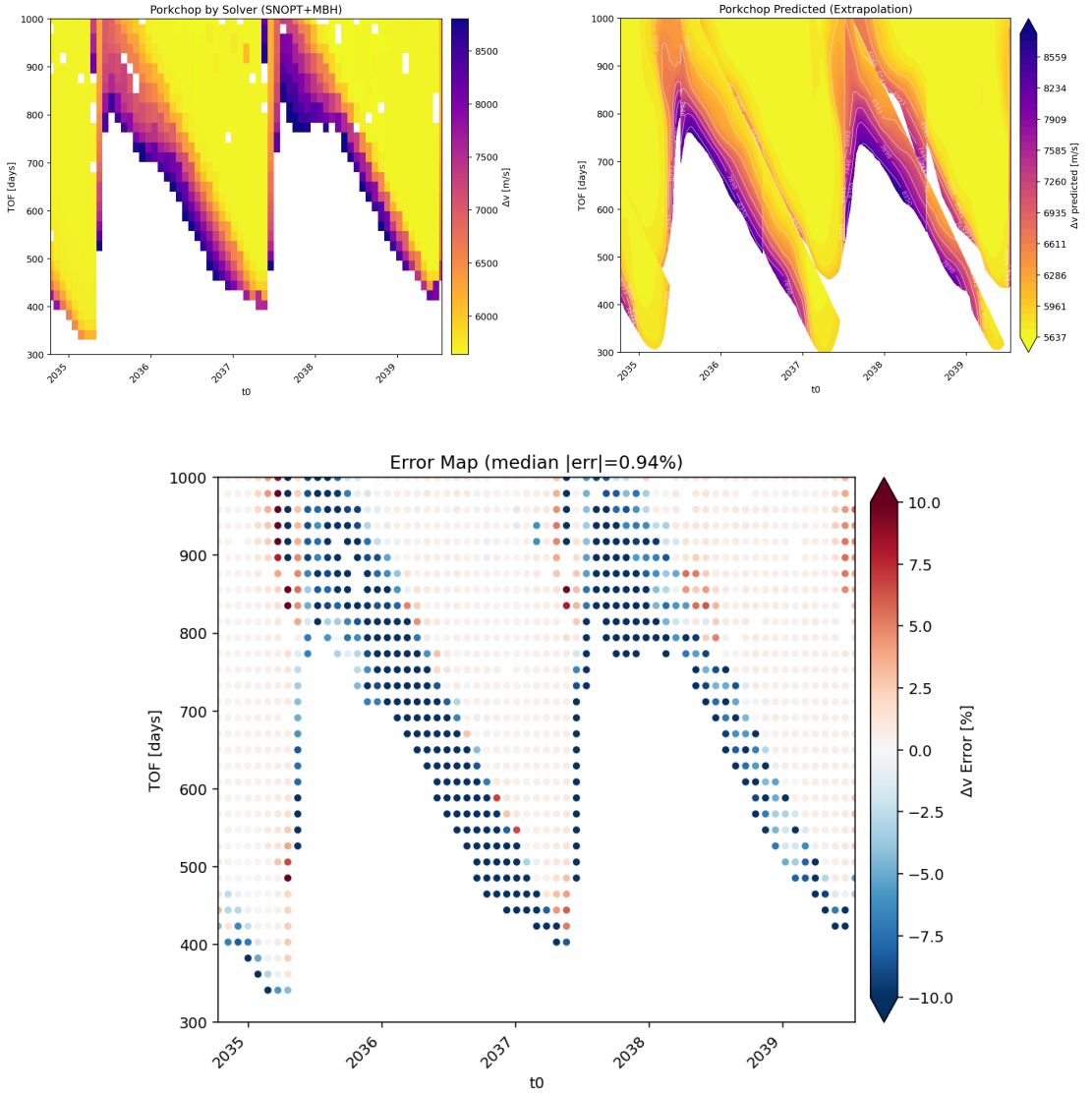


Figure 7.13: Temporal extrapolation, configuration 1 with $\mathbf{x}_{\text{in},3}$: solver porkchop, predicted porkchop, and relative error map

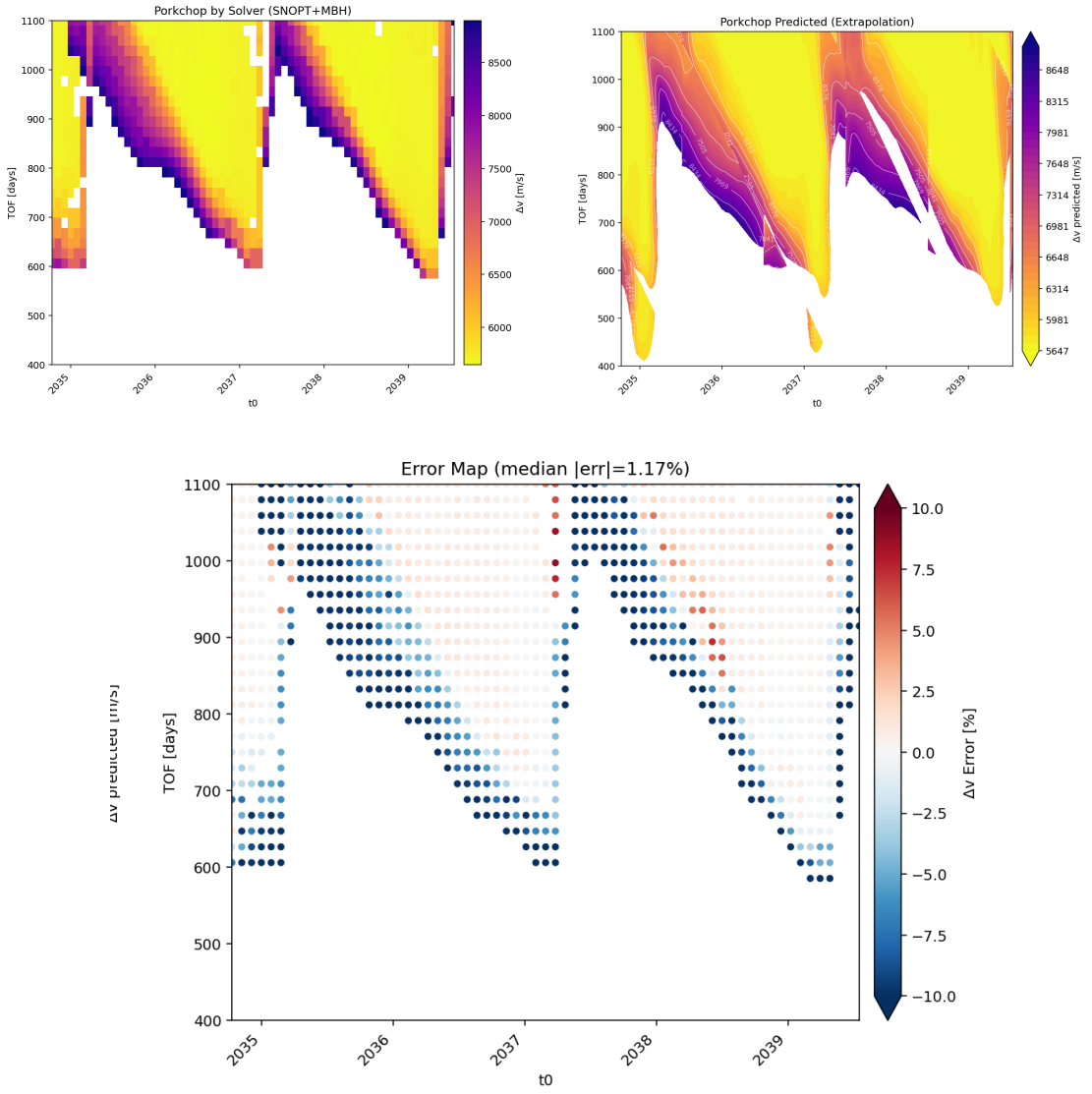


Figure 7.14: Temporal extrapolation, configuration 2 with $\mathbf{x}_{\text{in},3}$: solver porkchop, predicted porkchop, and relative error map

7.4.5 Effect of Configuration Coverage

A final comparison isolates the role of the coverage of the parameter space. The two networks, with the same architecture and official input vector, were retrained using six configurations instead of eight (excluding configuration 7, at the far right of the domain) and evaluated on three new held-out configurations. By restricting the extent of the space to be covered, the six configurations sample

it comparatively more uniformly. The held-out results improve with respect to the eight-configuration case: the classifier accuracy rises to 0.97 (AUC 0.994) and the regressor R^2 to 0.862, as illustrated in the coverage map of Figure 7.15. Of particular note is the left edge of the domain, where configuration 4 previously recorded the lowest value: here the R^2 is now markedly higher, the opposite extreme that widened the space to be interpolated being no longer present. The comparison confirms what already emerged from the learning curves: what most benefits an interpolator is not the absolute quantity of data, but the density and uniformity with which they cover the parameter space.

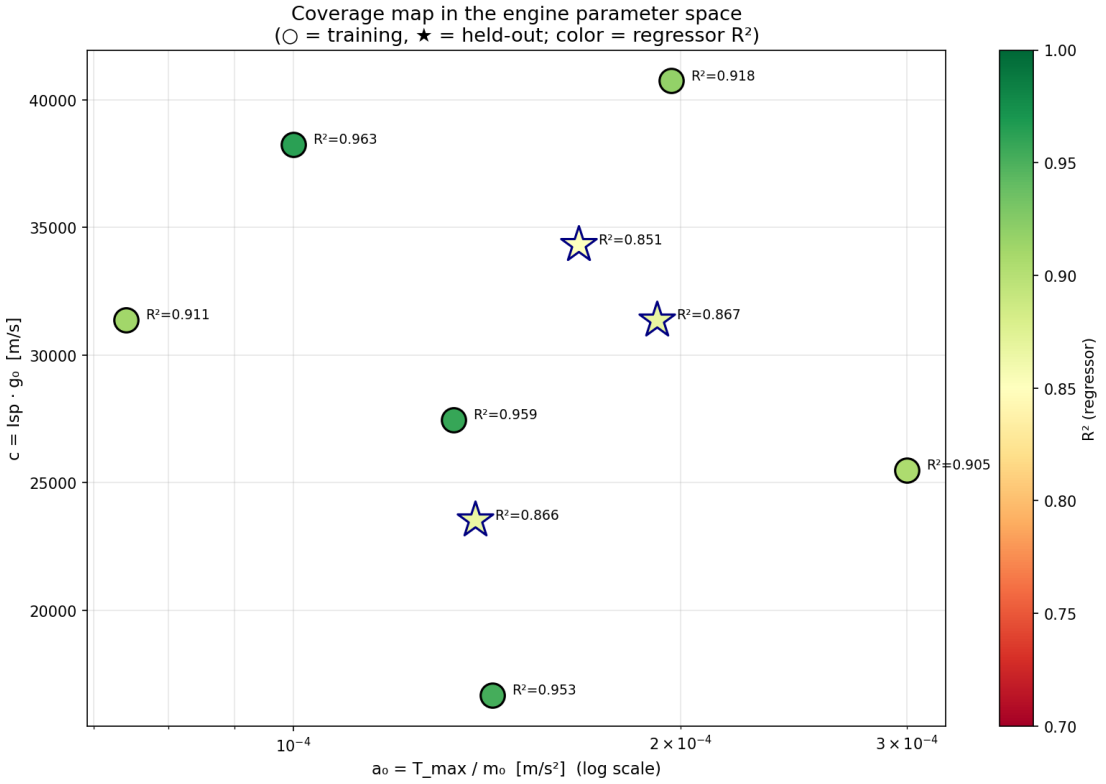


Figure 7.15: Coverage map in the engine parameter space with six training configurations and three new held-out ones. Labels are omitted to avoid confusion with the configuration numbering used elsewhere

Chapter 8

Conclusions

The preliminary design of low-thrust transfers is computationally demanding by its nature: every combination of launch window and propulsion configuration calls for the solution of a non-convex optimization problem, at a cost that renders the extensive exploration of the design space prohibitive. This work has addressed that obstacle through a surrogate model based on feed-forward neural networks, arranged as a cascade pipeline of two networks: a classifier, which discriminates the parameter combinations admitting a physically realizable transfer from those that do not, and a regressor, which estimates, for the feasible solutions alone, the transfer cost in terms of Δv and, through the Tsiolkovsky equation, the final mass delivered. Both networks were trained on a dataset of optimal Earth–Mars transfers, generated by means of the Sims–Flanagan transcription coupled with the SNOPT7 optimizer and wrapped within a Monotonic-Basin-Hopping global search.

Turning to the results, on the configurations seen during training both networks perform excellently: the classifier attains an AUC close to unity, recovering the minority class of the non-realizable points with nearly the same quality as the majority one, while the regressor estimates the Δv with a coefficient of determination of 0.94 and a mean relative error of about one percent, further reduced on the final mass by the attenuating effect of the Tsiolkovsky relation.

The most significant result concerns the generalization to propulsion configurations never seen during training. Queried on engines absent from the training set, but whose parameters lie within the covered domain, the surrogate retains performance close to the nominal one: the classifier suffers no appreciable degradation, while the regressor's coefficient of determination decreases from 0.94 to 0.82, a drop dominated by a minority of outliers at high Δv, the median error remaining below one percent. It is this capability that qualifies the model as a standalone predictive tool, and not as a mere interpolator confined to the configurations of the dataset. Its domain of validity should nonetheless be stated precisely: this is generalization by interpolation within the sampled domain, not extrapolation beyond it, as the

temporal extrapolation analysis demonstrates unambiguously.

In its intended use, the pipeline reproduces the porkchop plots of the solver faithfully (the structure of the transfer cost, the position and shape of the launch windows, their synodic recurrence) even for unseen configurations, and at a computational cost roughly seven orders of magnitude lower. It moreover fills the isolated convergence failures that dot the solver maps with values consistent with their context, a regularizing effect with respect to the artificial discontinuities of the dataset. This gap in cost is best understood in terms of a division of labor: the surrogate dominates the exploratory phase (broad, dense, and tolerant of error, since its purpose is to provide orientation) while pointwise refinement remains the province of the high-fidelity optimizer, invoked only where the surrogate has indicated that doing so is worthwhile. The contribution of the proposed model does not lie in replacing the optimizer, but in shifting the bulk of the work from the second phase to the first.

The limitations of the tool, rather than weaknesses, constitute the natural directions along which the work may be continued.

First, the surrogate predicts the feasibility of a transfer and its cost, but not the trajectory itself: of the entire decision vector, it determines every component except the controls, precisely the part the solver must determine from scratch. Extending it to the prediction of the controls would supply SNOPT with a complete guess, whose benefit should then be measured directly, as a reduction in iterations with respect to a random initialization. A complementary route exploits the impulsive Lambert transfer, a simpler and well-understood problem on which the neural surrogate approach was validated at a preliminary stage of this work, and where the nearly flawless generalization attained confirmed the effectiveness of the methodology. Its solution, obtainable in closed form, provides the velocities at the endpoints of the transfer arc, quantities that belong to the decision vector of the low-thrust problem: it could therefore serve as a physically informed initialization, to be refined rather than taken as an approximate solution.

Second, the coverage map and the learning curves converge on a single conclusion: the generalization to unseen engines is limited not by the sampling density of the individual configuration, which has reached saturation, but by the coverage of the (a_0, c) space. This reframes the question in terms of sampling efficiency: what is the minimum number of configurations, and their optimal placement, sufficient for the network to learn the underlying relation over a prescribed region of the parameter space? Since each configuration costs a full grid of optimizations, identifying the smallest sufficient set would reduce the cost of dataset generation in direct proportion, while the region so covered would continue to define the boundary of validity of the surrogate.

Third, the assumptions proper to a preliminary design phase can be relaxed: non-zero excess velocities would include the contribution of the launch and of

the approach phase, and the high-fidelity propagation would restore an accurate representation of the continuous thrust. The extension to other pairs of bodies lies in the same direction. In all these cases only the underlying model changes, while the methodology of the surrogate remains unaltered.

Taken as a whole, this work demonstrates that a surrogate model based on neural networks can rapidly provide estimates close to the optimal solutions computed by the solver, and generalize to propulsion configurations not included in training, provided its domain of validity is respected. The cost of a single evaluation thus ceases to be the factor that limits the breadth of the exploration, and the preliminary design of low-thrust interplanetary transfers gains a tool capable of returning in moments what direct optimization would require weeks to compute.

Bibliography

[1] M. D. Rayman, P. Varghese, D. H. Lehman, and L. L. Livesay. «Results From the Deep Space 1 Technology Validation Mission». In: *Acta Astronaut.* 47.2–9 (2000), pp. 475–487.

[2] M. D. Rayman, T. C. Fraschetti, C. A. Raymond, and C. T. Russell. «Dawn: A Mission in Development for Exploration of Main Belt Asteroids Vesta and Ceres». In: *Acta Astronaut.* 58.11 (2006), pp. 605–616.

[3] J. Kawaguchi, A. Fujiwara, and T. Uesugi. «Hayabusa—Its Technology and Science Accomplishment Summary and Hayabusa-2». In: *Acta Astronaut.* 62.10–11 (2008), pp. 639–647.

[4] S.-i. Watanabe, Y. Tsuda, M. Yoshikawa, S. Tanaka, T. Saiki, and S. Nakazawa. «Hayabusa2 Mission Overview». In: *Space Sci. Rev.* 208.1 (2017), pp. 3–16.

[5] A. Anselmi and G. E. Scoon. «BepiColombo, ESA's Mercury Cornerstone Mission». In: *Planet. Space Sci.* 49.14–15 (2001), pp. 1409–1420.

[6] J. T. Betts. «Survey of Numerical Methods for Trajectory Optimization». In: *J. Guid. Control Dyn.* 21.2 (1998), pp. 193–207.

[7] F. Topputo and C. Zhang. «Survey of Direct Transcription for Low-Thrust Space Trajectory Optimization With Applications». In: *Abstr. Appl. Anal.* 2014 (2014), p. 851720.

[8] N. V. Queipo, R. T. Haftka, W. Shyy, T. Goel, R. Vaidyanathan, and P. K. Tucker. «Surrogate-Based Analysis and Optimization». In: *Prog. Aerosp. Sci.* 41.1 (2005), pp. 1–28.

[9] C. Ampatzis and D. Izzo. «Machine Learning Techniques for Approximation of Objective Functions in Trajectory Optimisation». In: *Proc. IJCAI-09 Workshop Artificial Intelligence in Space*. Pasadena, CA, 2009, pp. 1–6.

[10] D. Hennes, D. Izzo, and D. Landau. «Fast Approximators for Optimal Low-Thrust Hops Between Main Belt Asteroids». In: *Proc. IEEE Symp. Series Computational Intelligence (SSCI)*. Athens, Greece, Dec. 2016, pp. 1–7.

[11] A. Mereta, D. Izzo, and A. Wittig. «Machine Learning of Optimal Low-Thrust Transfers Between Near-Earth Objects». In: *Hybrid Artificial Intelligent Systems (HAIS 2017)*. Vol. 10334. Lecture Notes in Computer Science. Cham, Switzerland: Springer, 2017, pp. 543–553.

[12] Y.-h. Zhu and Y.-Z. Luo. «Fast Evaluation of Low-Thrust Transfers via Multilayer Perceptions». In: *J. Guid. Control Dyn.* 42.12 (2019), pp. 2627–2637.

[13] A. Forestieri and L. Casalino. «Neural Network-Based Optimization of LEO Transfers». In: *Aerospace* 11.11 (2024), p. 879.

[14] J. A. Sims and S. N. Flanagan. «Preliminary Design of Low-Thrust Interplanetary Missions». In: *AAS/AIAA Astrodynamics Specialist Conf.* Girdwood, AK, Aug. 1997.

[15] P. E. Gill, W. Murray, and M. A. Saunders. «SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization». In: *SIAM J. Optim.* 12.4 (2002), pp. 979–1006.

[16] R. H. Leary. «Global Optimization on Funneling Landscapes». In: *J. Global Optim.* 18.4 (2000), pp. 367–383.

[17] C. H. Yam, D. Di Lorenzo, and D. Izzo. «Low-Thrust Trajectory Design as a Constrained Global Optimization Problem». In: *Proc. Inst. Mech. Eng. G, J. Aerosp. Eng.* 225.11 (2011), pp. 1243–1251.

[18] R. C. Woolley and C. W. Whetsel. «On the Nature of Earth-Mars Porkchop Plots». In: *AAS/AIAA Space Flight Mechanics Meeting*. Kauai, HI, Feb. 2013.

[19] D. M. Goebel and I. Katz. *Fundamentals of Electric Propulsion: Ion and Hall Thrusters*. JPL Space Science and Technology Series. Hoboken, NJ: Wiley, 2008.

[20] Jr. A. E. Bryson and Y.-C. Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Revised. Washington, DC: Hemisphere, 1975.

[21] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes*. New York, NY: Interscience, 1962.

[22] D. A. Vallado. *Fundamentals of Astrodynamics and Applications*. Fourth. Space Technology Library. Hawthorne, CA: Microcosm Press, 2013.

[23] D. H. Ellison, J. A. Englander, M. T. Ozimek, and B. A. Conway. «Analytical Partial Derivative Calculation of the Sims–Flanagan Transcription Match Point Constraints». In: *AAS/AIAA Space-Flight Mechanics Meeting*. Santa Fe, NM, Jan. 2014.

[24] D. Izzo. «PyGMO and PyKEP: Open Source Tools for Massively Parallel Optimization in Astrodynamics (the Case of Interplanetary Trajectory Optimization)». In: *Proc. 5th Int. Conf. Astrodynamics Tools and Techniques (ICATT)*. Noordwijk, The Netherlands, 2012.

[25] G. G. Wang and S. Shan. «Review of Metamodeling Techniques in Support of Engineering Design Optimization». In: *J. Mech. Des.* 129.4 (2007), pp. 370–380.

[26] A. I. J. Forrester, A. Sóbester, and A. J. Keane. *Engineering Design via Surrogate Modelling: A Practical Guide*. Chichester, U.K.: Wiley, 2008.

[27] C. Sánchez-Sánchez and D. Izzo. «Real-Time Optimal Control via Deep Neural Networks: Study on Landing Problems». In: *J. Guid. Control Dyn.* 41.5 (2018), pp. 1122–1135.

[28] G. Cybenko. «Approximation by Superpositions of a Sigmoidal Function». In: *Math. Control Signals Syst.* 2.4 (1989), pp. 303–314.

[29] K. Hornik, M. Stinchcombe, and H. White. «Multilayer Feedforward Networks Are Universal Approximators». In: *Neural Netw.* 2.5 (1989), pp. 359–366.

[30] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[31] A. Amini and A. Soleimany. *MIT 6.S191: Introduction to Deep Learning*. Massachusetts Institute of Technology. 2020. URL: `http://introtodeeplearning.com`.

[32] F. Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain». In: *Psychol. Rev.* 65.6 (1958), pp. 386–408.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. «Learning Representations by Back-Propagating Errors». In: *Nature* 323.6088 (1986), pp. 533–536.

[34] C. M. Bishop. *Pattern Recognition and Machine Learning*. New York, NY: Springer, 2006.

[35] D. P. Kingma and J. Ba. «Adam: A Method for Stochastic Optimization». In: *Proc. 3rd Int. Conf. Learning Representations (ICLR)*. San Diego, CA, May 2015.

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. «Dropout: A Simple Way to Prevent Neural Networks From Overfitting». In: *J. Mach. Learn. Res.* 15.56 (2014), pp. 1929–1958.

[37] *BHT-1500 Hall Effect Thruster*. Datasheet, Version 1.0. Natick, MA: Busek Co. Inc., Aug. 2021. URL: `https://www.busek.com`.

[38] J. Fisher, B. Ferraiuolo, J. Monheiser, et al. «NEXT-C Flight Ion System Status». In: *AIAA Propulsion and Energy Forum*. Aug. 2020.

[39] J. Porst, C. Altmann, C. Arnold, et al. «The RIT 2X Propulsion System: Current Development Status». In: *35th Int. Electric Propulsion Conf. (IEPC)*. Atlanta, GA, Oct. 2017.

[40] R. Shastry et al. «12-kW Advanced Electric Propulsion System Hall Current Thruster Qualification and Production Status». In: *J. Electr. Propuls.* 4 (2025), p. 61.

[41] J. A. Englander and A. C. Englander. «Tuning Monotonic Basin Hopping: Improving the Efficiency of Stochastic Search as Applied to Low-Thrust Trajectory Optimization». In: *Proc. 24th Int. Symp. Space Flight Dynamics (ISSFD)*. Laurel, MD, May 2014.

[42] A. Paszke, S. Gross, F. Massa, et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32 (NeurIPS)*. Curran Associates, 2019, pp. 8024–8035.

[43] *PyTorch*. Accessed: Jul. 13, 2026. The PyTorch Foundation. URL: `https://pytorch.org`.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. «Scikit-Learn: Machine Learning in Python». In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.

[45] S. Ioffe and C. Szegedy. «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift». In: *Proc. 32nd Int. Conf. Machine Learning (ICML)*. Lille, France, 2015, pp. 448–456.

[46] V. Nair and G. E. Hinton. «Rectified Linear Units Improve Restricted Boltzmann Machines». In: *Proc. 27th Int. Conf. Machine Learning (ICML)*. Haifa, Israel, 2010, pp. 807–814.

[47] P. J. Huber. «Robust Estimation of a Location Parameter». In: *Ann. Math. Stat.* 35.1 (1964), pp. 73–101.