Introduction to optimization methods for training SciML models

Alena Kopaničáková, Elisa Riccietti

Introduction to optimization methods for training SciML models

Alena Kopaničáková*and Elisa Riccietti†

June 2, 2026

Contents

*Toulouse-INP, IRIT-APO, ANITI, 2 Rue Charles Camichel, 31000 Toulouse, France; email: alena.kopanicakova@toulouse-inp.fr

†ENS de Lyon, CNRS, Inria, Universitè Claude Bernard Lyon 1, LIP, UMR 5668, 69342, Lyon cedex 07, France; email: elisa.riccietti@ens-lyon.fr

1 Introduction

Optimization is a key ingredient of modern machine learning (ML). Historically, optimization problems have been classified as *convex* or *non-convex*, according to the *properties* of the objective function and of the feasible set. Convex problems are generally more tractable, since global optimality guarantees can be established and any stationary point is also a global minimum. Non-convex problems may possess multiple local minima and saddle points, making optimization significantly more challenging.

Optimization methods have been categorized by the degree to which they exploit derivative information. *First-order* methods rely exclusively on gradient evaluations, whereas (approximate) *second-order* methods incorporate curvature information via Hessian matrices or suitable approximations. However, the massive scale of modern ML problems, combined with the principles of empirical risk minimization, has driven the field toward *stochastic optimization*. Stochastic optimization methods, such as Stochastic Gradient Descent (SGD) [99] and its variants (e.g., AdaGrad [28], and Adam [54]), employ inexpensive noisy gradient evaluations that scale efficiently with the data size. Consequently, much of modern ML focuses on integrating first-order schemes, adaptive gradient methods, and curvature-aware techniques into stochastic optimization frameworks.

This familiar optimization landscape changes substantially in the context of *scientific machine learning* (SciML). Unlike classical ML, where abundant data allows stochastic approximation to perform well, SciML often operates in data-scarce regimes in which physical models must supplement, or even dominate, the available data. As a result, the optimization problems arising in SciML frequently take the form of *physics-informed* formulations, in which the objective function incorporates a partial differential equation (PDE) and boundary or initial conditions (BC or IC). The PDE and BC/ICs are usually imposed in a *soft way* through a penalization term. An alternative is to use *operator-constrained* formulations, in which the physical constraints are embedded directly into the model, typically through the neural network architecture, and are therefore enforced in a *hard way*.

This changes the structure of the objective function compared to classical ML, where the loss decomposes into independent sample contributions, enabling efficient stochastic optimization. In SciML, the differential operator induces global spatio-temporal coupling, creating dependencies across the entire domain so that errors at different collocation points are not independent. This makes stochastic optimization significantly more challenging and highly sensitive to the choice of the samples.

Moreover, the resulting optimization landscape is typically shaped by the spectra of the underlying differential operators rather than by the statistics of a dataset. As a consequence, it is often highly *anisotropic*, meaning that the curvature varies significantly across different directions, and *stiff*, in the sense that the problem involves widely different scales and Hessian eigenvalues spanning several orders of magnitude. Such loss functions may therefore exhibit directions of very large curvature, for instance those induced by high-order derivatives,

alongside nearly flat directions associated with weakly identifiable components of the SciML model.

This structure limits the effectiveness of first-order methods, since it increases sensitivity to the choice of step size, which must remain small to ensure stability along directions of large curvature, thereby slowing progress in flatter directions. It also limits the effectiveness of stochastic methods and often necessitates the use of deterministic or large-batch optimization algorithms, as well as curvature-exploiting approaches such as deterministic quasi-Newton methods.

At the same time, SciML also includes a rapidly growing class of *data-driven* formulations that more closely resemble classical supervised learning. Prominent examples include operator-learning models [68], neural surrogate models trained on high-fidelity PDE simulation data [80], autoencoder-based reduced-order models that learn low-dimensional latent dynamics [61], and neural differential equation frameworks that fit observational or simulated trajectories [15]. Because their training objectives decompose over samples and admit mini-batching, these models inherit much of the optimization behavior of standard ML and benefit directly from well-established stochastic optimization techniques. As a result, effective optimization methods in SciML span two complementary regimes: physics-informed settings and data-driven settings that align more closely with large-scale stochastic ML training.

In this chapter, we first introduce the optimization problems that arise in both traditional ML and SciML, highlighting their structural differences and analytical properties. We then present a foundational overview of deterministic and stochastic optimization methods, explaining how these algorithms can be adapted to address the specific challenges posed by SciML models. Given the breadth of the optimization field, the material covered here does not aim to be exhaustive. Instead, we present a concise introduction to common optimization techniques, demonstrate how they can be adapted to SciML training through practical examples[1], and point toward promising research directions.

Chapter's structure The rest of the chapter is organized as follows: The first three sections form the foundations. In particular, Section 2 introduces the principles of empirical risk minimization and the finite-sum minimization problem, with examples spanning classical ML and SciML. Section 3 discusses the difficulties associated with solving arising minimization problems and introduces the neural tangent kernel (NTK), an essential tool for analyzing the training dynamics. Section 4 presents stochastic gradient descent (SGD), the most widely used method for ML optimization, together with a brief convergence analysis and numerical examples.

The following sections dive deeper into more advanced optimization methods. More precisely, Section 5 covers accelerated and adaptive stochastic gradient methods (momentum, AdaGrad, Adam). Section 6 introduces (approximate) second-order methods, including Hessian-free Newton-Krylov and quasi-

[1]The accompanying notebooks are available at https://github.com/kopanicakova/intro_opt_SciML.git.

Newton schemes, such as L-BFGS.

While Sections 2–6 focus on standard optimization methods for ML, Sections 7–9, present more advanced concepts, which are specific to the SciML context. In particular, Section 7 discusses the impact of regularization and soft-constraints on the performance of the optimization algorithms. Section 8 presents the strategies to mitigate *spectral bias* through input embeddings and multiscale architectures. Section 9 then focuses on sampling strategies that are critical for the training of PINNs. Finally, Section 10 gives an overview of current research directions and open challenges in optimization for SciML/PINNs. Notably, the techniques presented in Sections 7–9 can be applied in conjunction with any optimization method presented in Sections 4–6 and can be therefore followed right after the reader becomes familiar with (S)GD algorithm. A visual roadmap of the chapter is given in Figure 1.1.

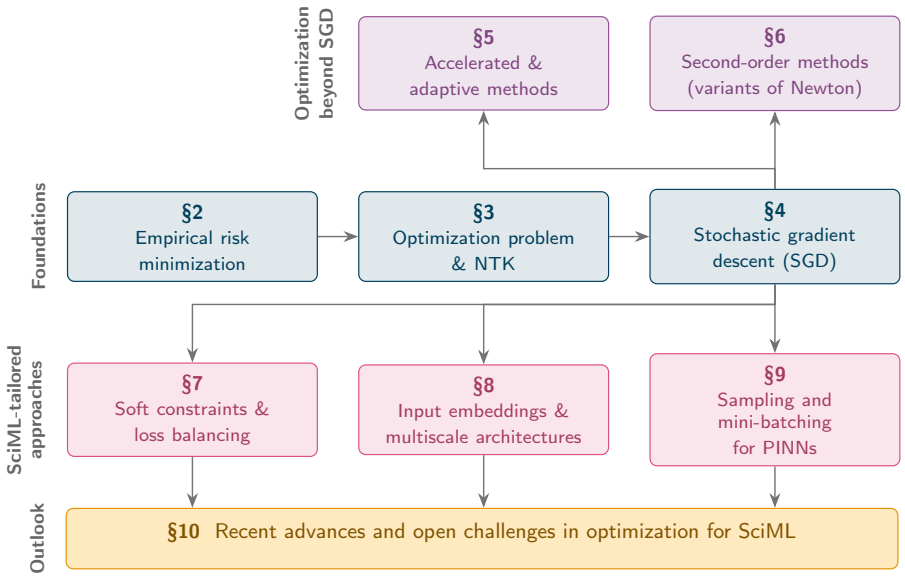

Figure 1.1: Roadmap of the chapter. Sections 2–4 form foundations suitable for a first reading. Sections 5–6 extend the foundations with general-purpose accelerated, adaptive, and second-order methods. Sections 7–9 discuss aspects specific to the training of SciML/PINN models, namely balancing terms related to soft constraints, input embeddings, multiscale architectures, and choice of sampling and mini-batching strategies. Section 10 surveys recent research directions and open challenges.

Notations Throughout the chapter, $\|\cdot\|$ denotes the Euclidean norm, unless otherwise specified.

2 Empirical risk minimization and the finite sum minimization problem

In ML, a model is characterized by a parameter vector $\theta \in \mathbb{R}^n$ and a parameterized mapping $h_\theta : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ denotes the input (feature) space and $\mathcal{Y}$ is the output space. The mapping h_θ transforms an input sample $x \in \mathcal{X}$ into an output prediction $h_\theta(x) \in \mathcal{Y}$, and its form is typically defined by a deep neural network (DNN) whose parameters are collected in the vector θ.

We aim to find a model h_θ that provides the best possible approximation of the desired mapping. The process of searching for such a model is called *training* and requires solving an optimization problem defined by an expected loss over the data distribution. Let (x, y) denote a generic data pair drawn from an unknown probability distribution $\mathcal{P}$ on $\mathcal{X} \times \mathcal{Y}$. Let $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ denote a loss function measuring the model's performance. For example, ℓ may represent a prediction error (supervised learning), a reconstruction error (autoencoders), a divergence between distributions (generative models), or the violation of physical laws (SciML).

The general training objective is to find the parameters θ that minimize the *expected risk* of the model, given by

$$R(\theta) = \int_{\mathcal{X}\times\mathcal{Y}} \ell(h_\theta(x), y)\, \mathrm{d}\mathcal{P}(x, y) = \mathbb{E}_{(x,y)\sim\mathcal{P}}[\ell(h_\theta(x), y)]\,. \tag{1}$$

However, since the distribution $\mathcal{P}$ is usually unknown in practice, learning relies on a finite dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$, consisting of m samples drawn independently and identically from $\mathcal{P}$, i.e., $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} \mathcal{P}$.

Using the dataset $\mathcal{D}$, the expected risk (1) can be approximated, for example, using a Monte-Carlo estimate, yielding the *empirical risk*

$$\widehat{R}_m(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(h_\theta(x_i), y_i).$$

Under standard assumptions (i.i.d. sampling, bounded variance), $\widehat{R}_m(\theta)$ converges to $R(\theta)$ as $m \to \infty$; see, e.g., [114, 103].

The *Empirical Risk Minimization (ERM)* principle consists of minimizing $\widehat{R}_m$, which gives rise to the canonical *finite-sum minimization problem*

$$\min_{\theta\in\mathbb{R}^n} f(\theta) := \frac{1}{m}\sum_{i=1}^{m} f_i(\theta), \qquad \text{where} \qquad f_i(\theta) = \ell(h_\theta(x_i), y_i). \tag{2}$$

Here, each function $f_i(\theta)$ represents the contribution of one sample (x_i, y_i) to the total objective $f(\theta)$. In theoretical analyses, convergence of the iterates generated by solving (2) is typically established by showing that the gradient norm $\|\nabla f(\theta)\|$ approaches zero, indicating that a stationary point has been reached. In the ML context, however, practitioners more commonly monitor the training loss $f(\theta_k)$, the validation error on a held-out dataset, or task-specific

metrics such as accuracy. These practical metrics better reflect generalization performance, which is the ultimate goal of ML training and may not correlate directly with the gradient norm.

2.1 Examples of ERM across learning paradigms

In this section, we illustrate how the abstract ERM formulation translates into practical learning settings by presenting three representative examples.

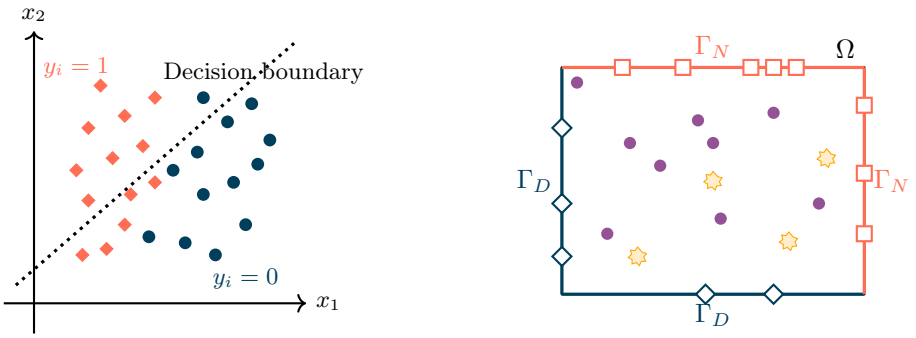

Figure 2.1: *Left:* Example of binary classification in the (x_1, x_2)-plane: Samples from two classes $y_i \in \{0, 1\}$ are shown as orange diamonds and blue circles, respectively. The dotted line indicates the decision boundary separating the two regions. *Right:* Example of a computational domain Ω with sampled collocation points: Dirichlet boundaries Γ_D (diamond), Neumann boundaries Γ_N (square), interior points $\mathcal{D}_\Omega$ (circle), and optional empirical data samples $\mathcal{D}_{\text{data}}$ (star).

2.1.1 Example 1: Supervised learning - classification

In supervised learning, we are given a labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$, where each input vector $x_i \in \mathbb{R}^d$ is associated with an output label y_i, which indicates affiliation to a given class. The goal is to learn the parameters $\theta \in \mathbb{R}^n$ of the model $h_\theta : \mathbb{R}^d \to \mathbb{R}$ to predict the label y_i for a given input x_i and for unseen examples. For simplicity, we restrict our attention to the binary classification setting where $y_i \in \{0, 1\}$.

Logistic regression (linear model) In logistic regression, the model is

$$h_\theta(x_i) = \rho(x_i^\top \theta),$$

where $\rho(t) = \frac{1}{1+e^{-t}}$ is the *sigmoid* activation. The discrepancy between prediction and label is measured by the logistic loss, given as

$$\ell(h_\theta(x_i), y_i) = -\, y_i \log h_\theta(x_i) - (1 - y_i) \log\big(1 - h_\theta(x_i)\big). \tag{3}$$

The ERM objective then becomes

$$\widehat{R}_m(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big[-\,y_i \log \rho(x_i^\top \theta) - (1-y_i)\log\big(1-\rho(x_i^\top\theta)\big)\Big]. \tag{4}$$

This function is smooth and *convex* (strictly, cf. Definition 2, if the features are linearly independent and strongly, cf. Definition 3, if an L^2 regularization term is added). Therefore, its minimization admits a unique minimum (and a unique minimizer in the strictly and strongly convex cases). The practical difficulty of minimizing $\widehat{R}_m$ depends predominantly on the geometry of the data. If the data are well-conditioned and the classes are separable, convergence is fast. If, on the other hand, the samples are nearly collinear or the classes overlap, the optimization problem becomes more challenging. An example of implementing (4) in Python can be found in Code snippet 1.

```python
def logloss(w, X, y, gamma=1e-4):
    # L2-regularized logistic loss averaged over samples
    # Logit z is the product of parameters (w) with input features (X)
    z = X @ w
    p = sigmoid(z)
    eps = 1e-12
    # Binary cross-entropy loss
    # Constant eps is added for numerical safety
    loss = -np.mean(y * np.log(p+eps) + (1-y)*np.log(1-p+eps))
    loss += 0.5 * gamma * np.sum(w**2)  # Add L2 penalty term
    return float(loss)

```

Code snippet 1: Example implementation of the logistic loss with L^2 regularization.

Deep neural network (nonlinear model) If the data are not linearly separable, linear models may be insufficient to capture the underlying structure. In such cases, DNNs provide flexible nonlinear alternatives capable of learning complex nonlinear relationships and feature interactions present in the data.

Let h_θ denote a feed-forward DNN composed of weight matrices $U_1, U_2, \ldots, U_Q$ and bias vectors $b_1, b_2, \ldots, b_Q$, which can be collected into a parameter vector θ as $\theta = \mathrm{vec}(U_1, \ldots, U_Q, b_1, \ldots, b_Q) \in \mathbb{R}^n$. A standard Q-layer network is defined recursively as

$$z_0 = x, \qquad z_j = \rho(U_j z_{j-1} + b_j), \quad j = 1, \ldots, Q-1,$$

with the output

$$h_\theta(x) = \rho(U_Q z_{Q-1} + b_Q). \tag{5}$$

Here, ρ denotes an activation function of the user's choice, e.g., $ReLU$, or $tanh$. A prototypical implementation of a DNN with two layers and $ReLU$ activation function in Pytorch can be found in Code snippet 2.

Given a dataset $\mathcal{D}$ and a loss function ℓ, the empirical risk is now given as

$$\widehat{R}_m(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell(h_\theta(x_i),\, y_i)\,.$$

The choice of the loss function ℓ depends on the task at hand. For example, in a binary classification problem, one can use the logistic loss (3). Here, we emphasize that the use of DNNs makes $\widehat{R}_m$ highly nonconvex. The complexity of the resulting optimization problem strongly depends on the data geometry and on the chosen DNN architecture.

```python
class DNN(nn.Module):
    # Small fully-connected ReLU network
    def __init__(self):
        super().__init__()
        # Linear layer with 2 input neurons and 20 neurons in hidden layer
        self.fc1 = nn.Linear(2, 20)
        self.act = nn.ReLU() # Activation function
        self.fc2 = nn.Linear(20, 1) # Output layer with a single output

    def forward(self, x):
        # Forward pass through the network
        return self.fc2(self.act(self.fc1(x)))
```

Code snippet 2: Example of creating a small DNN in PyTorch.

2.1.2 Example 2: Supervised learning - regression

Regression is a supervised learning paradigm widely used in data-driven SciML. Given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$, the goal is to learn a mapping from inputs x_i to real-valued outputs $y_i \in \mathbb{R}$ that generalizes well to unseen data. For example, the input may consist of observational measurements, while the output might represent the solution of a PDE or a quantity derived from it.

In regression, the discrepancy between the model prediction and the true label is commonly measured using the squared L^2 loss. Defining the stacked data labels $y := \big[y_1, \ldots, y_m\big]^\top$ and model outputs $u(\theta) := \big[h_\theta(x_1), \ldots, h_\theta(x_m)\big]^\top$, the ERM problem takes the following form:

$$\widehat{R}_m(\theta) = \frac{1}{2m} \|u(\theta) - y\|^2. \tag{6}$$

This ERM formulation is commonly referred to as a *least-squares* problem.

If the labels depend linearly on the inputs, this relationship can be modelled using the linear model $h_\theta(x_i) = x_i^\top \theta$. In this particular case, the ERM problem admits the compact representation $\widehat{R}_m(\theta) = \frac{1}{2m} \|X\theta - y\|^2$, where $X \in \mathbb{R}^{m \times d}$ denotes the data matrix. Minimizing this objective corresponds to solving a *linear least-squares* problem. Thus, if X has full column rank, the solution is given in closed form as $\theta = (X^\top X)^{-1} X^\top y$.

If, on the other hand, the model h_θ is nonlinear, e.g., a DNN given in (5), the ERM problem (6) becomes a *nonlinear least-squares* problem. Here, the term "nonlinear" refers to the nonlinear dependence of $h_\theta(x)$ on θ, even though the loss remains quadratic in the residual (misfit). The resulting optimization problem is generally *nonconvex* and may admit multiple local minima and saddle points, making its minimization considerably challenging.

2.1.3 Example 3: Physics-informed neural networks (PINNs)

In physics-informed learning, our goal is to approximate a solution of a differential equation using an ML model [96]. Let $\Omega \subset \mathbb{R}^d$ be a bounded domain with boundary $\partial\Omega = \Gamma_D \cup \Gamma_N$, where Γ_D and Γ_N represent parts where Dirichlet and Neumann boundary conditions (BC) are imposed, respectively. We consider the following PDE:

$$\begin{aligned} \mathcal{N}[u](x) &= q(x), \qquad x \in \Omega, \\ u(x) &= g_D(x), \quad x \in \Gamma_D, \\ \partial_n u(x) &= g_N(x), \quad x \in \Gamma_N, \end{aligned} \tag{7}$$

where $\mathcal{N}$ is a differential operator, q is a source term, and $\partial_n u$ denotes the normal derivative on Γ_N. The goal is to approximate the unknown solution $u : \Omega \to \mathbb{R}$ with a DNN $h_\theta : \Omega \to \mathbb{R}$. The same formulation extends to time-dependent (non-stationary) PDEs by treating time as an additional input coordinate of h_θ. The computational domain Ω is then replaced by the space-time domain $\Omega \times [0, T]$, where Γ_D and Γ_N denote the parts of the spatial boundary where Dirichlet and Neumann conditions are enforced. Moreover, an initial condition (IC) $u(x, 0) = u_0(x)$ is prescribed on $\Omega \times \{0\}$.

In the continuous setting, model quality is quantified by an expected risk functional that penalizes the PDE residual

$$r_\theta(x) := \mathcal{N}[h_\theta](x) - q(x), \tag{8}$$

boundary condition violations, and mismatch with available data, i.e.,

$$\begin{aligned} R(\theta) = \gamma_\Omega \int_\Omega \left|\mathcal{N}[h_\theta](x) - q(x)\right|^2 \mathrm{d}\mathcal{P}_\Omega(x) + \gamma_D \int_{\Gamma_D} \left|h_\theta(x) - g_D(x)\right|^2 \mathrm{d}\mathcal{P}_{\Gamma_D}(x) \\ + \gamma_{\Gamma_N} \int_{\Gamma_N} \left|\partial_n h_\theta(x) - g_N(x)\right|^2 \mathrm{d}\mathcal{P}_{\Gamma_N}(x) \\ + \gamma_{\Omega_{\text{data}}} \int_{\Omega_{\text{data}}} \left|h_\theta(x) - y(x)\right|^2 \mathrm{d}\mathcal{P}_{\text{data}}(x), \end{aligned} \tag{9}$$

where $\gamma_\Omega, \gamma_D, \gamma_{\Gamma_N}, \gamma_{\Omega_{\text{data}}} \in \mathbb{R}^+$.

In practice, the integrals in (9) must be approximated. A common approach is to use Monte-Carlo (MC) quadrature. Thus, we use the following datasets of randomly sampled collocation points

$$\begin{aligned} \mathcal{D}_\Omega &= \{x_j^{(\Omega)}\}_{j=1}^{m_\Omega}, & \mathcal{D}_{\Gamma_D} &= \{x_r^{(D)}\}_{r=1}^{m_D}, \\ \mathcal{D}_{\Gamma_N} &= \{x_k^{(N)}\}_{k=1}^{m_N}, & \mathcal{D}_{\text{data}} &= \{(x_i^{(\text{data})}, y_i)\}_{i=1}^{m_{\text{data}}}, \end{aligned}$$

to obtain the following empirical risk formulation:

$$\widehat{R}_m(\theta) = \frac{\gamma_\Omega}{m_\Omega}\sum_{j=1}^{m_\Omega}\left|\mathcal{N}[h_\theta](x_j^{(\Omega)}) - q(x_j^{(\Omega)})\right|^2 + \frac{\gamma_D}{m_D}\sum_{r=1}^{m_D}\left|h_\theta(x_r^{(D)}) - g_D(x_r^{(D)})\right|^2 + \frac{\gamma_N}{m_N}\sum_{k=1}^{m_N}\left|\partial_n h_\theta(x_k^{(N)}) - g_N(x_k^{(N)})\right|^2 + \frac{\gamma_{\text{data}}}{m_{\text{data}}}\sum_{i=1}^{m_{\text{data}}}\left|h_\theta(x_i^{(\text{data})}) - y_i\right|^2. \tag{10}$$

Code snippet 3 illustrates the computation of the PDE residual loss for the 1D Poisson equation using automatic differentiation in PyTorch.

```python
def pinn_second_derivative(model, x):
    # Compute second derivative by repeated automatic differentiation

    # Enable computation of gradient wrt. network inputs
    xg = x.clone().detach().requires_grad_(True)
    u = model(xg) # Forward pass through the network
    # Compute the first derivative
    du = torch.autograd.grad(u, xg, torch.ones_like(u), create_graph=True)[0]
    # Compute the second derivative
    d2u = torch.autograd.grad(du, xg, torch.ones_like(du),
     create_graph=True)[0]
    return d2u

def loss_fn(uxx, f_rhs):
    # Mean-squared of PDE-residual loss for -u''(x) = f(x)
    return ((uxx + f_rhs) ** 2).mean()
```

Code snippet 3: Pytorch implementation of the PDE residual loss for the 1D Poisson equation $-u''(x) = f(x)$, using automatic differentiation to compute the second order derivative of the network output.

Here, we point out that more efficient quadrature techniques than MC can be used in order to reduce the approximation error. For instance, *quasi-Monte Carlo* (QMC) methods based on low-discrepancy sequences (e.g., Sobol') can achieve approximation errors of order $\mathcal{O}(m^{-1}(\log m)^d)$, compared to the $\mathcal{O}(m^{-1/2})$ rate of standard MC under suitable regularity assumptions [26]. Here, we emphasise that increasing the number of collocation points in PINNs is *not* analogous to adding independent samples in classical ML, as the same deterministic PDE residual is evaluated at additional locations. Due to the global coupling induced by the differential operator, nearby points are often strongly correlated, so oversampling does not yield proportional accuracy gains [81, 69].

As before, in order to find the optimal parameters of model h_θ, we minimize $\widehat{R}_m$. Although h_θ is usually a standard DNN, the choice of the architecture requires particular care because $\widehat{R}_m$ involves *spatial derivatives* of h_θ. In particular, the activation function ρ must be sufficiently smooth to allow stable differentiation through the PDE operator. Weak formulations of PINNs can relax these smoothness requirements [22]. However, for the strong formulation

considered here, which is the most commonly used, smooth activations such as *tanh*, *sigmoid*, needs to be used. Alternatively, one might use *SIREN* (Sinusoidal Representation Networks) architecture [107] with sinusoidal activation functions.

Key Takeaways

Training an ML model is typically formulated as minimizing an empirical risk function that aggregates individual sample losses into a finite-sum objective (2). The structure of this objective depends critically on the model and task. For example, logistic regression yields a convex problem with a unique minimum, whereas DNNs yield highly nonconvex objectives with multiple local minima and saddle points. In PINNs, physical laws are incorporated into the loss as penalty terms, enforcing the PDE and boundary conditions in a soft manner. This introduces global spatio-temporal coupling across the domain, making the resulting optimization problem structurally different from classical ML objectives that naturally decompose over independent samples.

3 Solving the optimization problem

Given a nonlinear, differentiable objective (loss) function $f : \mathbb{R}^n \to \mathbb{R}$, the goal of numerical optimization [86] is to solve

$$\min_{\theta \in \mathbb{R}^n} f(\theta).$$

In the ML context, this problem is often referred to as *training*. Such minimization problems are often solved approximately, until a *stationary point*, i.e., a point at which the gradient is zero, is reached. In most cases, this stationary point will be a *local minimizer*, a point such that the function does not decrease in its neighborhood. However, it is usually not possible to exclude it as a *saddle point*.

To reach a stationary point, one typically employs *iterative algorithms*, also referred to as *optimizers*. Starting from an initial guess θ_0, such methods generate a sequence $\{\theta_k\}_{k \in \mathbb{N}}$, where k denotes the iteration index, that converges to a stationary point $\theta^\star$ through updates of the form

$$\theta_{k+1} = \theta_k + \alpha_k \, p_k,$$

where $p_k \in \mathbb{R}^n$ denotes a search direction and $\alpha_k > 0$ is the step size, often referred to as the *learning rate*.

Many factors influence the convergence speed of the training. On the problem side, these factors include the network architecture, the training data, and the non-convexity of the loss function. On the algorithmic side, they include the choice of the optimization algorithm, its hyperparameters, and the initialization

strategy. The latter determines the initial network parameters θ_0, which play the role of the initial guess for the optimization problem. This choice can strongly affect the optimization trajectory, especially in non-convex settings. Several initialization strategies have been proposed in the literature for the weights of DNNs, e.g., *Xavier* or *Glorot initialization* [33], which is commonly used with tanh activations, and *He initialization* [41], which is typically used with ReLU activations.

3.1 A unifying perspective via preconditioning

Throughout this chapter, we show that many optimization methods can be understood as forms of *preconditioning*, directly linking algorithmic design choices to the conditioning of the underlying optimization problem. Following [21], we distinguish three types of preconditioning depending on the space in which they act. Parameter-space preconditioners act directly on the gradient update, e.g., by rescaling it (block) coordinate-wise (adaptive learning rates) or incorporating curvature information (second-order methods), see Sections 5, 6. Data-space preconditioners transform the data, for example via input transformations, dataset resampling, or reweighting of the loss terms (Sections 7, 8, 9, 10). Function-space preconditioners reshape the geometry of the function represented by the network, e.g., via multiscale architectures (Sections 8, 10).

Here, we note that this classification is not a strict partition. Indeed, many strategies admit interpretations in more than one space (e.g., some domain decomposition methods act in both parameter and function space). It should therefore be viewed as a helpful classification perspective rather than a strict taxonomy. Figure 3.1 provides a schematic summary and serves as a roadmap for the strategies discussed in this chapter. Section 10 returns to this perspective, surveying recent advances and open challenges in large-scale settings along each axis.

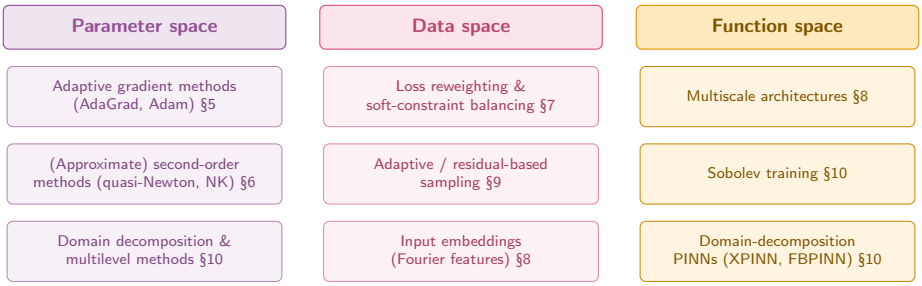

Figure 3.1: Overview of the preconditioning classification. Optimization strategies discussed in this chapter are classified as preconditioners acting in one of the three spaces: parameter, data, or function space. Section references indicate where each strategy is presented in detail.

3.1.1 Parameter-space preconditioning

Assuming f is sufficiently regular, most optimization methods build search directions from the derivatives of f, namely the *gradient* $\nabla f(\theta)$ and the *Hessian* $H(\theta) = \nabla^2 f(\theta) \in \mathbb{R}^{n \times n}$. The update rule usually has the following form:

$$\theta_{k+1} = \theta_k - \alpha_k \, M_k^{-1} \nabla f(\theta_k),$$

where $\alpha_k > 0$ and M_k^{-1} is a preconditioner that rescales the gradient.

Based on the information the preconditioner M_k^{-1} uses, optimization methods are classically divided into three large classes:

1. *First-order methods*: These methods use only gradient information. As a consequence, their iteration cost is low, but they may require many iterations to converge. The canonical example is the gradient descent (GD) method, whose iterates are defined as follows:

$$\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k), \tag{11}$$

   where $M_k = I$.

2. *Second-order methods*: These methods incorporate curvature information from the Hessian or its approximations, providing much faster local convergence at a higher per-iteration cost. The most notable example is Newton's method, whose iterates are formally defined as follows:

$$\theta_{k+1} = \theta_k - \alpha_k H(\theta_k)^{-1} \nabla f(\theta_k), \tag{12}$$

   where $M_k = H(\theta_k)$. The main cost associated with this method lies in evaluating the Hessian and solving the associated linear system $H(\theta_k)p_k = -\nabla f(\theta_k)$ to compute the search direction p_k. The memory footprint is also a concern as storing $H(\theta_k)$ explicitly requires $\mathcal{O}(n^2)$ memory, which is prohibitive for large DNNs.

3. *Adaptive (preconditioned) gradient methods*: Adaptive gradient methods address the challenge of balancing computational efficiency with curvature-awareness by performing updates of the form

$$\theta_{k+1} = \theta_k - \alpha_k M_k^{-1} \nabla f(\theta_k), \tag{13}$$

   where M_k is a diagonal preconditioner. As we will see in Section 5, practical optimizers such as AdaGrad [28] and Adam [54] construct diagonal M_k from running estimates of the first and/or second moments of past gradients, thereby avoiding the need to compute and store the exact Hessian, while still exploiting curvature information.

3.1.2 Data-space preconditioning

A preconditioner M^{-1} can also be applied to the model's input x, which alters both the search direction and its magnitude. For a model h_θ and a loss function ℓ, the sample-wise objective becomes $f_i(\theta) = \ell\big(h_\theta(M^{-1}x),\, y\big)$, with gradient

$$\nabla_\theta f_i(\theta) = J_{h_\theta}(M^{-1}x)\, M^{-1}\, \nabla_z \ell.$$

Such transformations reshape the geometry of the input distribution and can significantly affect the conditioning and convergence behavior of the optimization process. Loss reweighting (Section 7), adaptive sampling (Section 9), and input embeddings such as Fourier features (Section 8) act in this space.

3.1.3 Function-space preconditioning

Function-space preconditioning refers to transformations that modify the geometry of the optimization problem directly in the space of functions represented by the DNN. This viewpoint is natural for SciML, where the learning objective is often defined through a differential operator $\mathcal{N}$. Let u denote the DNN output. For each sample, the objective reads $f(u) = \ell(\mathcal{N}(u), q)$, and the functional gradient is

$$\nabla_u f = \mathcal{N}^*(\mathcal{N}(u) - q)\,,$$

where $\mathcal{N}^*$ denotes the formal adjoint of $\mathcal{N}$ with respect to the L^2 inner product, defined by $\langle \mathcal{N}u, v\rangle = \langle u, \mathcal{N}^* v\rangle$ for all sufficiently smooth u, v.

A function-space preconditioner applies an operator $\mathcal{M}^{-1}$ to this gradient:

$$u_{k+1} = u_k - \alpha_k\, \mathcal{M}^{-1} \nabla_u f,$$

where ideally $\mathcal{M} \approx \mathcal{N}^*\mathcal{N}$. This mirrors classical PDE preconditioning and motivates several architectural and algorithmic strategies in SciML, in particular multiscale architectures (Section 8) as well as Sobolev training and domain-decomposition PINNs (Section 10). Unlike in linear PDE solvers, however, the operator $\mathcal{N}$ is composed with a nonlinear neural representation, making the design and analysis of effective function-space preconditioners substantially more challenging [21].

3.2 Neural tangent kernel

The Neural Tangent Kernel (NTK) is a theoretical tool introduced in [47] for studying the training dynamics of DNNs, especially in the "infinite-width" limit (i.e., for networks with layers of infinite width). It connects neural network training with kernel methods, providing a framework for understanding how GD shapes the function learned by a DNN. In particular, the eigenvalue spectrum of the NTK governs the dynamics and conditioning of optimization methods and provides insights into the training behavior of DNNs. For PINNs, NTK can be used to show that the optimization problem inherits stiffness (stiffness refers to the presence of widely separated scales - typically in time or space - that make numerical simulation difficult, since some components of the solution evolve very rapidly, while others evolve slowly) from the underlying differential operator $\mathcal{N}$, as well as structural biases induced by the network architecture. These effects make training notoriously challenging and often render first-order methods such as GD inefficient.

3.2.1 Function-space view: GD as kernel flow

In this section, we introduce the NTK from a function-space viewpoint. Let $h_\theta : \Omega \to \mathbb{R}$ be a DNN with parameters $\theta \in \mathbb{R}^n$. During GD training, the network updates its parameters, and the NTK describes how these parameter updates translate into changes in the network's output.

Continuous-time kernel dynamics The continuous-time version of GD, known as *gradient flow*, updates the parameters according to

$$\dot{\theta}(t) = -\nabla_\theta f(\theta(t)),$$

where $\dot{\theta}(t)$ denotes the derivatives of $\theta(t)$ with respect to t. We are interested in the induced evolution of the function h_θ. Using the chain rule, and assuming that f depends on θ only through h_θ, we get

$$\dot{h}_\theta(x) = \nabla_\theta h_\theta(x)^\top \, \dot{\theta}(t) = -\nabla_\theta h_\theta(x)^\top \, \nabla_\theta f(\theta(t)). \tag{14}$$

If we consider the gradient of the loss, we have

$$\nabla_\theta f = \int_\Omega \nabla_\theta h_\theta(x') \, \frac{\delta f}{\delta h(x')} \, dx', \tag{15}$$

where $\delta f / \delta h(x')$ is the functional derivative of f with respect to $h(x')$. Combining (14) with (15) gives

$$\dot{h}_\theta(x) = -\int_\Omega K_\theta(x, x') \, \frac{\delta f}{\delta h(x')} \, dx',$$

where

$$K_\theta(x, x') = \langle \nabla_\theta h_\theta(x), \nabla_\theta h_\theta(x') \rangle$$

is the *NTK*. Thus, when GD updates the parameters θ, the corresponding evolution of the function h_θ is governed by the kernel K_θ. In other words, *gradient flow in parameter space induces a kernel gradient flow in function space with respect to the NTK metric.*

Discrete-time NTK dynamics The same argument can be developed in a discretized setting, leading to the empirical NTK. This can be simply shown by considering the empirical squared loss, given in (6). In this case, the GD step with fixed step size $\alpha > 0$ reads as

$$\theta_{k+1} = \theta_k - \alpha \, \nabla f_m(\theta_k) = \theta_k - \frac{\alpha}{m} J_k^\top \big(u(\theta_k) - y\big), \tag{16}$$

where

$$J_k = \begin{bmatrix} \nabla_\theta h_{\theta_k}(x_1)^\top \\ \vdots \\ \nabla_\theta h_{\theta_k}(x_m)^\top \end{bmatrix} \in \mathbb{R}^{m \times n}$$

is the Jacobian of h_{θ_k} evaluated at all data points.

Following the linearization argument of [47, 62], we expand h_θ around θ_k:

$$h_{\theta_{k+1}}(x) \approx h_{\theta_k}(x) + \nabla_\theta h_{\theta_k}(x)^\top (\theta_{k+1} - \theta_k). \tag{17}$$

Since in our notation the predicted outputs evaluated on the training set satisfy $u(\theta_k) = h_{\theta_k}(x_{1:m})$, the linearization directly applies to $u(\theta_k)$ as well. Substituting the GD update (16), into this linearization then yields an approximate evolution equation for the predicted outputs:

$$u(\theta_{k+1}) \approx u(\theta_k) - \alpha\, \Theta_k\, (u(\theta_k) - y), \qquad \Theta_k := \frac{1}{m} J_k J_k^\top, \tag{18}$$

where $\Theta_k \in \mathbb{R}^{m \times m}$ is the empirical NTK at iteration k.

This indicates that training evolves like kernel GD in the reproducing kernel Hilbert space (RKHS) induced by the NTK. In sufficiently wide DNNs, or in the near-stationary regime (e.g., for small α), the NTK remains nearly constant during training. In this case, the dynamics reduce to kernel GD with a fixed kernel, in turn providing a tractable mathematical description of deep learning training. We report below two examples of NTK evaluation in PyTorch, for a simple regression problem (Code snippet 4) and for a PINN trained to approximate the solution of the heat equation (Code snippet 5). For PINNs, nested automatic differentiation is required, as the PINN loss itself involves spatial derivatives of the network output computed via autograd, and the NTK computation requires an additional differentiation with respect to the network parameters.

```python
def ntk_naive(net, X):
    # Compute Theta = (1/m) J J^T using naive loop
    params = [p for p in net.parameters() if p.requires_grad]
    rows = []
    # Loop over all data samples
    for i in range(X.shape[0]):
        out = net(X[i:i+1]).squeeze() # Forward pass through the network
        # Get the gradient of each output wrt. parameters
        grads = torch.autograd.grad(out, params, retain_graph=False)
        rows.append(torch.cat([g.reshape(-1) for g in grads]))
    J = torch.stack(rows, dim=0)  # Stack together all gradients
    return (1.0 / X.shape[0]) * (J @ J.T), J
```

Code snippet 4: Example of computing the NTK for a simple regression problem. More elaborate implementations are discussed in Section 3.6.

```python
def f_source(xt):
    # Evaluate source term
    x, t = xt[..., 0], xt[..., 1]
    return torch.exp(-t) * (PI**2 - 1.0) * torch.sin(PI * x)

def residual_at(net, xt):
```

```
    # Evaluation of the PDE residual, which uses NESTED autograd, i.e.,
    # the inner grad call must keep the graph alive (create_graph=True) so
    # that  u_t and u_xx remain differentiable  in theta for computation of
     NTK
    xt = xt.detach().clone().requires_grad_(True)

    u = net(xt).squeeze(-1) # Evaluate the network output
    # Compute gradients with respect to input
    grads = torch.autograd.grad(u.sum(), xt, create_graph=True)[0]
    u_x, u_t = grads[:, 0], grads[:, 1]
    u_xx = torch.autograd.grad(u_x.sum(), xt, create_graph=True)[0][:, 0]
    return u_t - u_xx - f_source(xt) # Residual evaluation


def ntk_naive_pinn(net, xt):
    # Same as ntk_naive function, but per-sample function is the PDE residual
    params = [p for p in net.parameters() if p.requires_grad]
    rows = []
    # Loop over all data samples
    for i in range(xt.shape[0]):
        # Evaluate residual for each data point
        r_i = residual_at(net, xt[i:i+1]).squeeze()
        # Compute gradients of residual wrt. parameters
        grads = torch.autograd.grad(r_i, params, retain_graph=False,
     allow_unused=True)
        flat = [(g if g is not None else torch.zeros_like(p)).reshape(-1)
                for g, p in zip(grads, params)]
        rows.append(torch.cat(flat))
    J = torch.stack(rows, dim=0)  # Stack together all gradients
    return (1.0 / xt.shape[0]) * (J @ J.T), J
```

Code snippet 5: Example of evaluating the NTK for PINN (heat equation). Note that, since computing the loss requires the evaluation of the residual, computing the NTK requires nested autograd.

3.2.2 Parameter-space viewpoint

The function-space analysis above can be complemented by a parameter-space perspective. For instance, for the empirical least-square loss (6), the gradient and Hessian are given as

$$\nabla \hat{R}_m(\theta_k) = \tfrac{1}{m} J_k^\top (u(\theta_k) - y), \qquad H(\theta_k) = \tfrac{1}{m} J_k^\top J_k + (\text{higher-order terms}).$$

If we are in the NTK regime, i.e., the network is wide and approximately linear in θ (cf. (17)), the higher-order terms are small, and H can be approximated by the Gauss-Newton (GN) approximation $\frac{1}{m} J_k^\top J_k$. Since the empirical NTK is $\Theta_k = \frac{1}{m} J_k J_k^\top$, it follows that $H(\theta_k)$ *and* Θ_k *share the same nonzero eigenvalues.* This creates a direct correspondence between NTK conditioning in function space and Hessian conditioning in parameter space. In particular, it implies that small eigenvalues of Θ_k imply small eigenvalues of $H(\theta_k)$, so that flat directions in the objective function landscape correspond exactly to slow-learning directions in function space.

As a consequence, if the NTK and the Hessian are ill-conditioned, this negatively impacts the convergence of the first-order methods. For example, let us consider the GD iteration (11) applied to a strongly convex quadratic model with constant Hessian H and step size $0 < \alpha < 2/\lambda_{\max}(H)$. It is well known that the GD iterates satisfy the following classical contraction bound (see, e.g., [86, Section 3.2]):

$$\|\theta_{k+1} - \theta^\star\| \leq \left(\frac{\kappa(H) - 1}{\kappa(H) + 1}\right) \|\theta_k - \theta^\star\|, \qquad \kappa(H) = \frac{\lambda_{\max}(H)}{\lambda_{\min}(H)}.$$

Thus, if $\lambda_{\min}(H)$ is very small, the contraction factor approaches one and progress becomes arbitrarily slow. Similar observations extend to stochastic gradient methods, which we discuss in the upcoming sections.

3.3 Explicit mode-wise error decay and spectral bias

The eigenvalues of Θ_k determine the contraction rates of the corresponding error modes during GD training. To understand this behavior, let us again consider the empirical least-squares loss (6). The function-space error is defined as $e_k := u(\theta_k) - y$. Using NTK-induced update (18), we can obtain a linear recursion of the following form:

$$e_{k+1} \approx (I - \alpha\Theta_k)\, e_k. \tag{19}$$

In regimes where Θ_k varies slowly, we may approximate Θ_k locally with constant Θ, i.e., $\Theta_k \approx \Theta$, yielding

$$e_k \approx (I - \alpha\Theta)^k e_0,$$

which is the standard linear kernel gradient-flow approximation used in [47, 62].

Let the eigendecomposition of Θ be

$$\Theta = Q\Lambda Q^\top, \qquad \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_m),$$

where q_i denotes the i-th eigenvector (column of Q) and λ_i is the associated eigenvalue. Because $\{q_i\}_{i=1}^m$ form an orthonormal basis of $\mathbb{R}^m$, we may expand the initial error as $e_0 = \sum_{i=1}^m c_i\, q_i$. Applying the recursion $e_{k+1} = (I - \alpha\Theta_k)e_k$ repeatedly, then yields

$$e_k = \sum_{i=1}^m c_i(1 - \alpha\lambda_i)^k q_i \qquad \text{and} \qquad \|e_k\|^2 = \sum_{i=1}^m c_i^2(1 - \alpha\lambda_i)^{2k}.$$

Thus, each coefficient c_i associated with eigenvector q_i is multiplied by the scalar factor $(1 - \alpha\lambda_i)$ at every iteration. This implies that the i-th component decays geometrically at rate $(1 - \alpha\lambda_i)^k$. Hence, if λ_i is large, the factor $1 - \alpha\lambda_i$ is significantly below one, which leads to a rapid decrease of the corresponding error mode. In contrast, if λ_i is very small, then $1 - \alpha\lambda_i \approx 1$, so the contraction per iteration is extremely small.

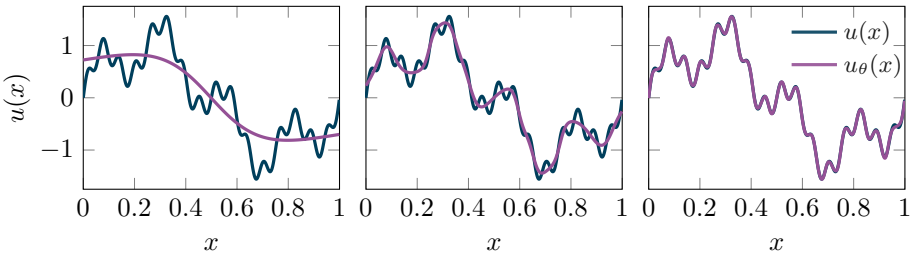

Figure 3.2: Example of spectral bias to learn target function u_θ (purple) composed of low-, medium-, and high-frequencies. The network prediction u_θ (purple) is reported at training iterations 100 (left), 1,500 (middle), and 7,500 (right).

This imbalance in decay rates across different eigenmodes is referred to as *spectral bias* [93, 122]. DNNs learn components of the solution that align with large NTK eigenvalues (typically low-frequency or smooth structures) much faster than the components associated with small eigenvalues (often high-frequency components). As a consequence, the smallest nonzero NTK eigenvalue $\lambda_{\min}(\Theta)$ governs the slowest decaying error and directly controls the overall speed of learning.

Here, we emphasize that different DNN architectures induce different NTKs and, therefore, exhibit different implicit biases. Specialized architectures can be designed to mitigate spectral bias; see, for example, Fourier features [111], discussed in Section 8 and Numerical Example #9.

We report the Python code to evaluate the condition number of a kernel matrix K in Code snippet 6. We remind that the matrix K depends on the chosen optimization method, e.g., for GD, $K = \Theta_k$.

```python
def cond_number(K):
    # Compute singular values (more numerically stable than eigenvalues)
    evals = torch.linalg.svdvals(K.cpu()).numpy()
    eps = 1e-9
    # Filter out numerical noise and zero eigenvalues
    pos = evals[evals > eps]
    if len(pos) == 0:
        return np.inf, pos
    # Condition number = sigma_max / sigma_min
    return pos.max() / pos.min(), pos
```

Code snippet 6: Computing the condition number of the kernel matrix K.

Numerical Example #1: *Spectral bias.* To illustrate the spectral bias phenomenon in practice, we consider nonlinear regression as specified in Section 2.1.2 to learn the function $u(x) = \sin(2\pi x) + 0.5\,\sin(8\pi x) + 0.2\,\sin(32\pi x)$, which is a sum of sinusoids of increasing frequency. The training is performed using the GD method. As we can see from Figure 3.2, in early training (itera-

tion 100), the network fits only the low frequencies. Medium frequencies appear later (iteration $1,500$), and only after many more iterations, the network begins to capture the high-frequency oscillations (iteration $7,500$).

3.4 Kernels of adaptive and second-order methods

We have seen that the conditioning of the NTK and the Hessian plays a central role in determining the training dynamics of the GD method. A similar analysis extends to adaptive and second-order methods. Using the same linearization as in (18), and writing $\nabla \hat{R}_m(\theta_k) = \frac{1}{m} J_k^\top \big(u(\theta_k) - y\big)$, the update rule (13) gives

$$\theta_{k+1} - \theta_k = -\frac{\alpha_k}{m}\, M_k^{-1} J_k^\top \big(u(\theta_k) - y\big),$$

where M_k^{-1} denotes the preconditioning matrix used by the optimizer. As before, we linearize the network output h_θ around θ_k at every training point and stack the resulting approximations into the vector $u(\theta)$. This yields the functional update

$$u(\theta_{k+1}) \approx u(\theta_k) - \alpha_k\, \Theta_k^{(M_k)} \big(u(\theta_k) - y\big),$$

with the *adaptive (or preconditioned) kernel*

$$\Theta_k^{(M_k)} := \frac{1}{m}\, J_k\, M_k^{-1}\, J_k^\top. \tag{20}$$

Thus, every adaptive and second-order method can be interpreted in function space as performing a kernel GD with kernel $\Theta_k^{(M_k)}$. The eigenvalues of $\Theta_k^{(M_k)}$ therefore govern the contraction rates of the error for a given optimizer. In the special case of standard GD ($M_k = I$), we recover $\Theta_k^{(M_k)} = \Theta_k$, whose eigenvalues $\{\lambda_i\}$ produce per-iteration decay factors of the form $(1 - \alpha\lambda_i)^k$. Adaptive and second-order optimizers reshape these decay rates through the spectrum of $\Theta_k^{(M_k)}$. When M_k is chosen well, the resulting spectrum is significantly more balanced, narrowing the gap between the fastest- and slowest-contracting modes, so that the singular values are more similar in magnitude. This phenomenon, often referred to as *spectral flattening*, improves the effective conditioning of the optimization problem $\big(\kappa(\Theta_k^{(M_k)}) < \kappa(\Theta_k)\big)$ and typically leads to faster and more stable training.

Numerical Example #2: *Preconditioning.* To illustrate the effect of preconditioning on the optimization dynamics, we consider a nonlinear regression problem as specified in Section 2.1.2. Our goal is to learn the function $\sin(\pi x_1) + \cos(\pi x_2)$ from inputs (x_1, x_2) using a simple DNN with one hidden layer. The model is trained using three different optimization methods. Specifically, we consider first-order GD, Adam as a representative of adaptive first-order methods (Section 5.2 below), and a damped Gauss-Newton (GN) method as an approximate second-order optimization (Section 3.2.2). GD corresponds to the choice $M_k = I$, while Adam employs a diagonal preconditioner M_k, see the update rule (28)). The damped GN method (or Levenberg-Marquardt

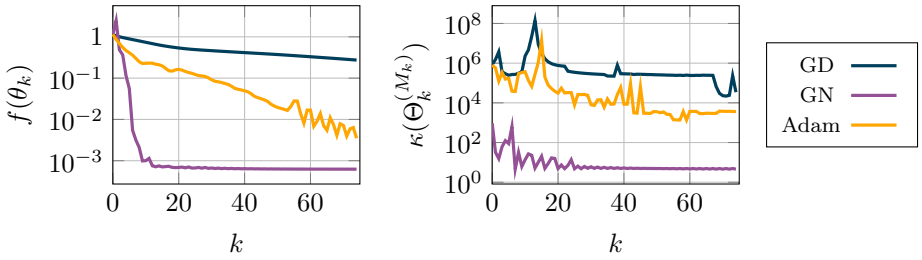

Figure 3.3: Comparison of the convergence behaviors of GD, the adaptive gradient method (Adam), and the Gauss-Newton (GN) method. *Left:* Evolution of the function value over the iterations. *Right:* Condition numbers of the corresponding empirical kernels $\kappa(\Theta_k^{(M_k)})$.

method) [86, ch. 10.3] uses $M_k = J_k^\top J_k + \beta I$, where J_k denotes the Jacobian of the model outputs with respect to the parameters and $\beta > 0$ is a damping parameter.

Figure 3.3 reports the evolution of the objective function and the condition number of the associated kernel $\Theta_k^{(M_k)}$. As we can see, GD produces the most ill-conditioned kernel, which correlates with slow convergence. Adam's diagonal preconditioning partially alleviates this issue, leading to slightly faster convergence. Notably, the GN method yields a substantially more balanced kernel spectrum, resulting in significantly smaller values of $\kappa(\Theta_k^{(M_k)})$ and the fastest empirical decrease of the objective function among the methods considered. To demonstrate the computation of $\Theta_k^{(M_k)}$ in practice, we provide in Code Snippet 7 an explicit construction for the damped GN method.

```
def compute_GN_kernel(model, X, beta):
        # Compute GN kernel: K_GN = J * (J^T J + lambda I)^{-1} * J^T
        # Damping is controlled by parameter beta
        B = J.t() @ J + beta * torch.eye(P, dtype=torch.float64)
        B_inv = torch.linalg.inv(B)
        K_gn = J @ B_inv @ J.t()
```

Code snippet 7: Example of constructing the kernel matrix $K = \Theta_k^{(M_k)}$ for the (damped) GN method.

3.5 Impact of the physical constraints on the loss landscape and the NTK of PINNs

A key difficulty in training PINNs arises from the fact that the loss function does not act directly on the network output h_θ, but rather on its derivatives through the PDE operator. This has a profound influence on the geometry of the objective function landscape and on the conditioning of the NTK, and

ultimately on the behaviour of employed optimization methods.

To understand this effect, let us consider the decomposition of h_θ in the Fourier components, i.e.,

$$h_\theta(x) = \sum_\omega \hat{h}(\omega) e^{i\omega x}$$

with $\hat{h}$ being the Fourier transform of h_θ. Let us now consider a Fourier mode $e^{\mathrm{i}\omega x}$ present in $h_\theta(x)$.

Applying p derivatives multiplies this mode by ω^p, so

$$\partial_x^p h_\theta \sim \omega^p, \qquad J_k \sim \omega^p, \qquad H(\theta) \approx \tfrac{1}{m} J_k^\top J_k \sim \omega^{2p}.$$

Thus, each Fourier mode of frequency ω approximately satisfies

$$H(\theta)\, q_\omega \approx \omega^{2p} q_\omega.$$

This produces a Hessian spectrum whose eigenvalues span

$$\lambda_{\min}(H(\theta_k)) \approx \omega_{\min}^{2p}, \qquad \lambda_{\max}(H(\theta_k)) \approx \omega_{\max}^{2p},$$

where $\omega_{\min}$ and $\omega_{\max}$ are the smallest and largest frequencies that the network effectively represents. The condition number thus scales as $\kappa(H) \approx \left(\frac{\omega_{\max}}{\omega_{\min}}\right)^{2p}$. As a consequence, as the differential order p increases, the exponent $2p$ causes the ill-conditioning to worsen dramatically. This explains why higher-order PDEs exhibit far more severe NTK/Hessian pathologies than first- or second-order PDEs.

The same scaling directly affects the curvature of the PINN objective function. Low-frequency modes create shallow, nearly flat directions in the loss landscape, while high-frequency modes produce extremely steep ravines. This anisotropy makes the optimization problem stiff and highly challenging to solve. First-order methods tend to move predominantly along low-frequency directions due to spectral bias, while second-order methods must cope with strong curvature anisotropy.

To illustrate these concepts, we consider two DNNs with identical architectures and parameter values to learn the solution of a one-dimensional Poisson equation. The first model is trained in a purely *data-driven* setting, formulated as a nonlinear regression problem using samples of the exact solution (Section 2.1.2). The second model is trained as a *PINN* (Section 2.1.3) by minimizing the PDE residual and the associated BC. Figure 3.4 compares the associated loss landscapes. Although both models approximate the same function u, the PINN loss landscape exhibits markedly more anisotropic curvature, while the data-driven objective remains comparatively smoother, albeit with several flat or low-curvature regions.

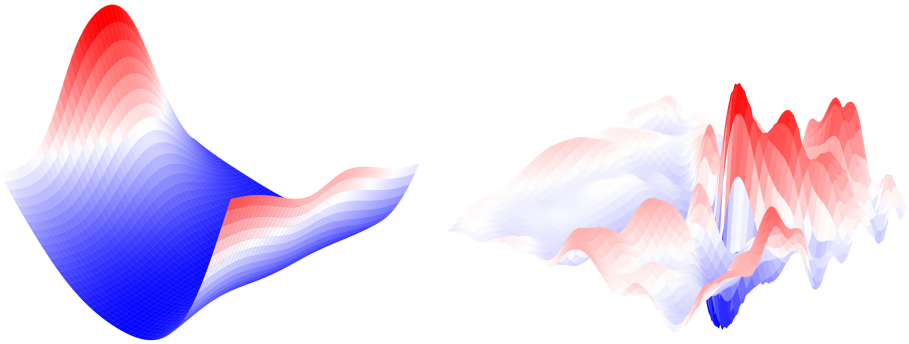

Figure 3.4: Two-dimensional projection [64] of the objective function landscape for the same DNN under two training regimes for 1D Poisson problem, where we learn a map from right-hand side to a solution. *Left:* Data-driven nonlinear regression. *Right:* PINN.

3.6 Practical computations with the NTK

All NTK arguments in this chapter rely on quantities derived from the empirical kernel

$$\Theta_k \;=\; \tfrac{1}{m}\, J_k J_k^\top \in \mathbb{R}^{m\times m}, \qquad (J_k)_{ij} \;=\; \tfrac{\partial h_{\theta_k}(x_i)}{\partial \theta_j}, \qquad J_k \in \mathbb{R}^{m\times n}, \tag{21}$$

where m is the number of inputs $\{x_i\}_{i=1}^m$ and $\theta \in \mathbb{R}^n$. While this representation is conceptually simple, computing the empirical NTK for modern DNNs is generally challenging due to the prohibitive computational and memory costs associated with forming and storing the Jacobian J_k, whose size scales as mn.

In practice, one of the following three standard strategies can be employed to compute $\Theta_k^{(M_k)}$ exactly, each suited to a different computational regime:

- **Naive autograd loop**: We construct J_k row by row by performing a backward pass for each sample. This approach is straightforward to implement and requires only standard automatic differentiation tools. However, it incurs m backward passes, leading to significant runtime overhead due to repeated graph traversals and poor hardware utilization. Its memory cost is $\mathcal{O}(mn)$, if J_k is stored explicitly, and it is typically impractical except for very small datasets.

- **Jacobian contraction**: Instead of performing a naive loop over the samples, the full Jacobian J_k can be assembled in a vectorized manner. This can be achieved by using batched automatic differentiation, which applies the gradient computation to all the samples in the batch at once. The NTK is then obtained via the matrix product $\Theta_k = \frac{1}{m} J_k J_k^\top$. This approach is often significantly faster than the naive method. However, it still requires storing J_k, leading to $\mathcal{O}(mn)$ memory usage, and the matrix multiplication itself can become a bottleneck. This approach is preferred

when both m and n are moderate and the matrix fits in the memory. An example of such calculation in PyTorch via the `torch.func.vmap` function is presented in Code snippet 8.

- **Matrix-free NTK-vector products** [87]: To reduce memory requirements, one can directly exploit the action of the kernel Θ_k on a vector $v \in \mathbb{R}^m$, i.e., the matrix-vector product $\Theta_k v$, without forming Θ_k explicitly. This approach leverages forward- and reverse-mode automatic differentiation: a `vjp` computes $J_k^\top v$ in parameter space via a single backward pass, while a `jvp` maps this back to output space via a forward pass, yielding $J_k J_k^\top v = m\Theta_k v$. Crucially, neither J_k nor Θ_k need to be formed explicitly, requiring only $\mathcal{O}(n)$ memory. We refer the reader to Code Snippet 9 for a PyTorch implementation.

The trade-off between Jacobian contraction and the matrix-free formulation is governed by the number of required matrix-vector products and memory constraints. If the full matrix Θ_k is needed (e.g., for direct eigendecomposition or visualization), Jacobian contraction is preferable. If only partial spectral information (e.g., extremal eigenvalues) or a small number of linear solves requiring Θ_k is desired, the matrix-free approach becomes advantageous. In practice, a few dozen matvecs often suffice when Krylov methods are used for the linear solves, making this approach cheaper than the contraction method once the cost of forming $J_k J_k^\top$ dominates.

```python
def fnet_single(params, x):
    # Scalar network output for a single input
    return functional_call(net_reg, params, (x.unsqueeze(0),)).squeeze()

def ntk_contraction(net, X, fn_single):
    # Vectorized approach: assemble J in one pass, then form Theta = (1/m) J J^T
    params = {k: v.detach() for k, v in net.named_parameters()}
    # Apply vmap over samples
    # Function jacrev returns the gradient of the scalar h_theta wrt theta
    jac = vmap(jacrev(fn_single, argnums=0), (None, 0))(params, X)
    cols = [jac[k].reshape(X.shape[0], -1) for k in jac]
    J = torch.cat(cols, dim=1)
    return (1.0 / X.shape[0]) * (J @ J.T), J

# Get NTK using contraction approach
Theta_contr, J_contr = ntk_contraction(net_reg, x_reg, fnet_single)
```

Code snippet 8: Example of evaluating the empirical NTK via Jacobian contraction using `torch.func.vmap` function.

```python
def make_ntk_matvec(net, X, fn_batch):
    # Return a closure Theta v
    # Argument fn_batch must take a params dict and return the batched output

```

```
    params = {k: v.detach() for k, v in net.named_parameters()}
    m_local = X.shape[0]
    def matvec(v):
        # Step 1: J^T v via reverse-mode
        _, vjp_fn = vjp(lambda p: fn_batch(p, X), params)
        (jtv,) = vjp_fn(v)
        # Step 2: J (J^T v) via forward-mode
        _, theta_v = jvp(lambda p: fn_batch(p, X), (params,), (jtv,))
        return theta_v / m_local
    return matvec

def fnet_batch(params, X):
    # Vector-valued network prediction
    # Returns the m predictions stacked into (m,)
    return functional_call(net_reg, params, (X,)).squeeze(-1)

# Get NTK*v product
matvec = make_ntk_matvec(net_reg, x_reg, fnet_batch)
# Build full Theta from the matrix-free form by applying it to the canonical
    basis (we evaluate m matvecs)
Theta_implicit = torch.stack([matvec(torch.eye(m)[i]) for i in range(m)],
    dim=1)
```

Code snippet 9: Example of matrix-free computation of NTK-vector product.

Finally, we note that in large-scale settings, even matrix-free NTK computations may become prohibitively expensive. In such cases, several approximations have been proposed, for instance via random parameter projections [39], last-layer NTKs [62], or closed-form infinite-width NTKs [7, 88].

Numerical Example #3: *Evaluating the empirical NTK.* Figure 3.5 reports the wall time (left) and the dominant tensor memory (right) of the three exact approaches for computing the NTK. Experiments are performed using a four-layer tanh-MLP with 32 width ($n = 3{,}265$ parameters). We consider the number of inputs m from 32 to $1,024$. The reported memory is associated with the dominant tensor allocation, i.e., $J_k \in \mathbb{R}^{m \times n}$ for the naive loop and contraction approach. For the matrix-free approach, only a constant working set of size $\sim 4n$ is required.

As we can see from the observed results, the naive loop is never competitive. Jacobian contraction is the practical default whenever mn fits in memory, scaling almost linearly in m in both time and memory. Moreover, we also observe that the matrix-free approach is the qualitative outlier in terms of the memory requirements. Its size is set by the network size, not by m, and stays constant throughout the sweep. Combined with the flat per-matvec cost, this makes it the only viable algorithm for medium- to large-scale DNNs.

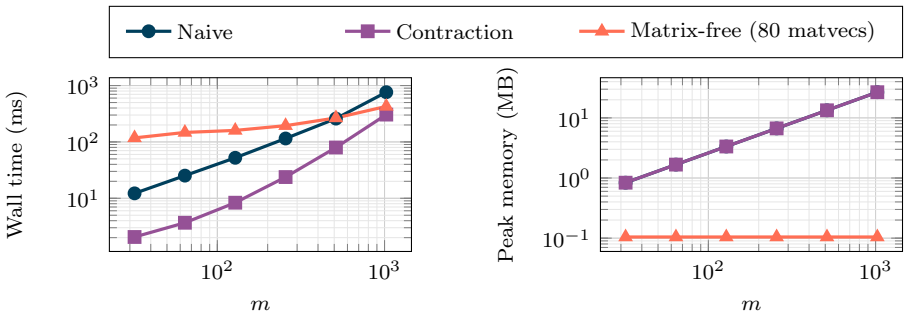

Figure 3.5: Cost of computing the empirical NTK for a four-layer tanh-MLP of width 32 ($n = 3{,}265$ parameters) as a function of the number of inputs m. *Left:* Wall time. *Right:* Dominant tensor memory.

Key Takeaways

The NTK is a fundamental tool for understanding the difficulties related to the solution of problems involving parametric models. It connects parameter-space optimization to function-space learning dynamics: the eigenvalues of Θ_k govern how quickly each error mode contracts during training. An ill-conditioned NTK, caused by widely separated eigenvalues, leads to slow convergence, as low-frequency modes learn much faster than high-frequency ones (*spectral bias*). Adaptive and second-order methods can improve convergence by reshaping the spectrum of the preconditioned NTK kernel $\Theta_k^{(M_k)}$, reducing its condition number and accelerating training.

4 Stochastic gradient descent (SGD)

Stochastic Gradient Descent (SGD) is the most widely used optimization method in modern ML, and plays an equally central role in data-driven SciML. Its popularity stems from its simplicity, scalability, and ability to train high-dimensional models on massive datasets. This widespread use is largely due to the limitations of classical GD, which, when applied to the finite-sum problem (2), requires evaluating the full gradient:

$$\nabla f(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(\theta).$$

When the dataset size (m) is large, evaluating the full gradient quickly becomes computationally prohibitive. However, real-world datasets are highly redundant, and computing every per-sample gradient at each iteration usually provides little additional benefit.

These considerations motivate the shift from classical deterministic optimization to *stochastic optimization*. The finite-sum structure of ML problems allows one to replace the full gradient with a cheaper approximation. The stochastic gradient (SGD) method randomly chooses a sample at each iteration and utilizes just the corresponding gradient. Its iterations thus read as follows:

$$\theta_{k+1} = \theta_k - \alpha_k \nabla f_{i_k}(\theta_k), \tag{22}$$

where $\alpha_k > 0$ is a given step size. The symbol i_k denotes an index drawn uniformly at random from $\{1, \ldots, m\}$ at iteration k.

Each iteration of the SGD method is thus very cheap, involving only the computation of the gradient $\nabla f_{i_k}(\theta_k)$, corresponding to one sample, rather than all the m terms. Since the index i_k is chosen randomly, the generated sequence depends on the random sequence $\{i_k\}$ and is not uniquely determined from θ_0. Note that each direction $-\nabla f_{i_k}(\theta_k)$ might not be one of descent for f (in the sense of yielding a negative directional derivative), but we can show that if it is a descent direction in expectation (the expected value of the scalar product with $\nabla f(\theta_k)$ is negative), then the sequence $\{\theta_k\}$ can be guided toward a minimizer of f. This comes from the fact that $\nabla f_{i_k}(\theta_k)$ is an unbiased estimator of $\nabla f(\theta_k)$, i.e.,

$$\mathbb{E}[\nabla f_{i_k}(\theta_k) \mid \theta_k] = \sum_{i=1}^{m} \frac{1}{m} \nabla f_i(\theta_k) = \nabla f(\theta_k).$$

The SGD method is summarized in Algorithm 1. Note that progress is in practice measured in epochs rather than iterations, where one epoch represents a complete pass over the training dataset, corresponding to m iterations. In contrast, the standard GD method, which evaluates the gradient using all m samples, performs exactly one iteration per epoch, which is why it is often referred to as batch gradient descent.

Algorithm 1 Stochastic Gradient (SGD) Method

1: **Given:** A dataset $\{x_i, y_i\}_{i=1}^m$, the objective functions $\{f_i\}_{i=1}^m$, an initial iterate $\theta_0 \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Randomly choose $i_k \in \{1, \ldots, m\}$ ▷ Sample-index selection
4: $p_k = -\nabla f_{i_k}(\theta_k)$ ▷ Search direction computation
5: Choose $\alpha_k > 0$ ▷ Step size selection
6: $\theta_{k+1} = \theta_k + \alpha_k p_k$ ▷ Iterate update
7: **end for**

The SGD method often exhibits rapid progress during the early stages of training but may stagnate as training proceeds. This behavior is commonly attributed to the inherent noise in stochastic gradient estimates, typically measured in terms of the variance

$$\sigma^2(\theta) := \mathbb{E}\left[\|\nabla f_{i_k}(\theta) - \nabla f(\theta)\|^2\right]. \tag{23}$$

This noise can help the iterates escape local minima and saddle points, but it also limits the attainable optimization accuracy, as discussed below. However, since ERM only approximates the expected risk, excessively optimizing the empirical objective may be detrimental and lead to overfitting, i.e., degraded performance on unseen data. Consequently, this limitation is often not critical in practice.

Mini-batch methods provide an intermediate approach between full-gradient methods and SGD. Whereas SGD relies on a single sample at each iteration, mini-batch methods compute gradient estimates using a small subset $\mathcal{I}_k \subset \{1, \dots, m\}$ of the samples at each iteration. The mini-batch $\mathcal{I}_k$ is typically drawn uniformly without replacement at each epoch, giving rise to the following update rule:

$$\theta_{k+1} = \theta_k - \frac{\alpha_k}{|\mathcal{I}_k|} \sum_{i \in \mathcal{I}_k} \nabla f_i(\theta_k).$$

This allows one to exploit some parallelism when computing mini-batch gradients, as each sample contribution can be evaluated independently. Moreover, a straightforward generalization of the variance formula (23) to the mini-batch setting shows that the variance of the gradient estimate decreases as $|\mathcal{I}_k|$ increases. As a consequence, mini-batch methods are easier to tune in terms of step size selection. Indeed, as we will see in the following section, the product of the step size and of the variance impacts how closely the iterates can approach an optimum. Since the variance decreases with larger mini-batch sizes, larger step sizes can be used, yielding faster convergence with lower optimization error.

In practice, selecting an appropriate step size is challenging, as its value directly affects the method's convergence. The stepsize α_k is often chosen heuristically. For example, a small constant value may be fixed, which, however, can ensure just convergence to a neighborhood of an optimum. Most often, a step size schedule is used to decrease α_k progressively, see Numerical Example #4.

4.1 Theoretical results

In this section, we present key convergence results for SGD applied to problem (2), and outline one representative proof to illustrate the main techniques. Additional results and proofs can be found in [32, 12]. Throughout, we denote by $\theta^\star$ a solution of (2). Note, the discussed results remain valid if the stochastic gradient ∇f_{i_k} is replaced by any unbiased gradient estimator g_k satisfying $\mathbb{E}[g_k \mid \theta_k] = \nabla f(\theta_k)$.

The stated results mainly depend on the assumptions we make about the objective function. We always assume the function to be bounded from below. One quite common assumption is smoothness.

Definition 1. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be L-smooth if there exists $L > 0$ such that, for all $\theta_1, \theta_2 \in \mathbb{R}^n$,*

$$\|\nabla f(\theta_1) - \nabla f(\theta_2)\| \leq L \|\theta_1 - \theta_2\|.$$

Assumption 1. *Each function $f_i : \mathbb{R}^n \to \mathbb{R}$ in problem* (2) *is L_i-smooth. We denote $L_{\max} := \max_{i=1,\dots,m} L_i$.*

Another important assumption, which, however, restricts the class of functions under study, is the convexity assumption. This assumption generally does not hold in the most general neural network training problem, but there are contexts in which it plays an important role, see for instance [11, 8].

Definition 2. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be convex if for all $\theta_1, \theta_2 \in \mathbb{R}^n$, for all $t \in [0, 1]$*

$$f(t\theta_1 + (1-t)\theta_2) \leq tf(\theta_1) + (1-t)f(\theta_2).$$

f is said to be strictly *convex if for all $\theta_1, \theta_2 \in \mathbb{R}^n$, for all $t \in (0, 1)$*

$$f(t\theta_1 + (1-t)\theta_2) < tf(\theta_1) + (1-t)f(\theta_2).$$

Definition 3. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be μ-strongly convex if there exists $\mu > 0$ such that for all $\theta_1, \theta_2 \in \mathbb{R}^n$*

$$f(\theta_2) \geq f(\theta_1) + \nabla f(\theta_1)^T(\theta_2 - \theta_1) + \frac{\mu}{2}\|\theta_1 - \theta_2\|^2.$$

When needed, we will assume the following.

Assumption 2. *Each function $f_i : \mathbb{R}^n \to \mathbb{R}$ in problem* (2) *is convex.*

Convexity of the function usually makes the solution of the problem easier, since the landscape will be more benign (for instance, all the local minima are global minima for convex functions). With this assumption, we can obtain stronger convergence results. For instance, we will see that when f is convex, we can prove a convergence result on the sequence of functions (even on the sequence of iterates when f is strongly convex), while in the nonconvex case, we can just show a result on the norm of the gradient.

Another important factor influencing convergence is the variance of the gradient estimate.

Assumption 3. *Assume that the variance of the gradient estimate in* (23) *is bounded: $\sigma^2(\theta) \leq \sigma^2$ for all θ.*

We will see that a non-zero variance will negatively impact the convergence of optimization algorithms, which is why variance reduction techniques [12] may be used to obtain better stochastic directions and relax assumptions on the step size, which also affects the results. In this chapter, we consider both constant ($\alpha_k = \alpha$ for all k) and vanishing ($\alpha_k = \mathcal{O}(1/k)$) step sizes.

We will see that in all the results, the key constants that determine the convergence rates (smoothness $L_{\max}$, strong convexity μ, and the SGD variance) are directly related to the eigenvalues of the Hessian $H(\theta)$. In particular, we recall that, if f is μ-strongly convex, then

$$L_{\max} = \sup_\theta \lambda_{\max}(H(\theta)), \quad \mu = \inf_\theta \lambda_{\min}(H(\theta)), \quad \kappa(H(\theta)) = \frac{\lambda_{\max}(H(\theta))}{\lambda_{\min}(H(\theta))}.$$

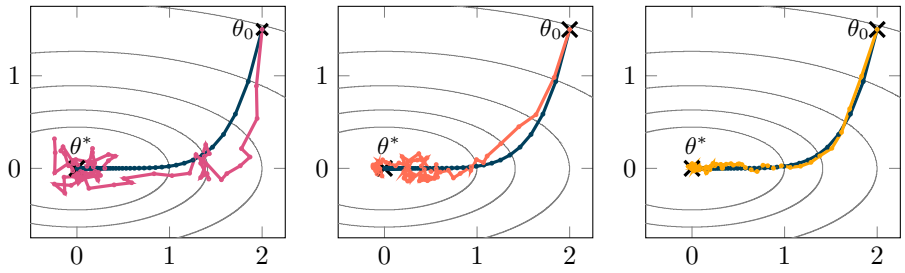

Figure 4.1: Trajectories of GD (blue) and SGD on a smooth quadratic function for different noise levels $\sigma \in \{\sigma_0,\ \sigma_0/2,\ \sigma_0/4\}$ (purple, orange, yellow). Higher noise magnitudes enlarge the region in which SGD oscillates around the minimizer θ^*, while lower noise confines the iterates closer to the GD path.

Thus, the magnitudes of $\lambda_{\min}(H(\theta))$ and $\lambda_{\max}(H(\theta))$, as well as their ratio $\kappa(H(\theta))$, play a crucial role in interpreting the obtained theoretical convergence guarantees. The convergence theorems show in particular how SGD deteriorates with increasing $\kappa(H(\theta))$.

4.1.1 Convergence for strongly convex and smooth functions

The first result applies to strongly convex functions, i.e., the ones with the most favorable landscape.

Theorem 1. *Let assumptions 1, 2 and 3 hold, and assume that f is μ-strongly convex. Assume that $\{\theta_k\}_{k\in\mathbb{N}}$ is the sequence generated by SGD (22) with a constant step size $0 < \alpha_k = \alpha < \frac{1}{2L_{\max}}$ for all k. Then, for $k \geq 1$, one has*

$$\mathbb{E}[\|\theta_k - \theta^\star\|^2] \leq (1-\alpha\mu)^k \|\theta_0 - \theta^\star\|^2 + \frac{2\alpha}{\mu}\sigma^2.$$

Proof. From the definition of the step (22) it holds that

$$\begin{aligned}\|\theta_{k+1} - \theta^\star\|^2 &= \|\theta_k - \alpha\nabla f_{i_k}(\theta_k) - \theta^\star\|^2 \\ &= \|\theta_k - \theta^\star\|^2 - 2\alpha\langle\nabla f_{i_k}(\theta_k), \theta_k - \theta^\star\rangle + \alpha^2\|\nabla f_{i_k}(\theta_k)\|^2.\end{aligned}$$

Taking the conditional expectation given θ_k yields

$$\mathbb{E}[\|\theta_{k+1} - \theta^\star\|^2 \mid \theta_k] = \|\theta_k - \theta^\star\|^2 - 2\alpha\langle\nabla f(\theta_k), \theta_k - \theta^\star\rangle + \alpha^2\mathbb{E}[\|\nabla f_{i_k}(\theta_k)\|^2 \mid \theta_k].$$

From the definition of strong convexity it follows

$$\mathbb{E}[\|\theta_{k+1} - \theta^\star\|^2 \mid \theta_k] \leq (1-\alpha\mu) - 2\alpha(f(\theta_k) - f(\theta^\star)) + \alpha_k^2\sigma^2.$$

Taking full expectation and using a variance transfer (see [32, lemma 4.20]) gives the main recursion:

$$\begin{aligned}\mathbb{E}[\|\theta_{k+1} - \theta^\star\|^2] &\leq (1-\alpha_k\mu)\,\mathbb{E}[\|\theta_k - \theta^\star\|^2] + 2\alpha_k^2\sigma^2 + 2\alpha(2\alpha L_{\max} - 1)\mathbb{E}[f(\theta_k) - f(\theta^\star)] \\ &\leq (1-\alpha_k\mu)\,\mathbb{E}[\|\theta_k - \theta^\star\|^2] + 2\alpha_k^2\sigma^2,\end{aligned}$$

where we have used the assumption on the step size. Recursively applying the above and summing up the resulting geometric series gives

$$\begin{aligned}\mathbb{E}[\|\theta_{k+1} - \theta^\star\|^2] &\leq (1-\alpha\mu)^k \|\theta_0 - \theta^\star\|^2 + 2\sum_{j=0}^{k-1}(1-\alpha\mu)^j \alpha^2\sigma^2 \\ &\leq (1-\alpha\mu)^k \|\theta_0 - \theta^\star\|^2 + \frac{2\alpha\sigma^2}{\mu}.\end{aligned}$$

□

This result tells us that SGD is not guaranteed to converge with a constant step size, since the upper bound on the error does not go to zero when k increases. Instead, the error stabilizes on the constant value $\frac{2\alpha\sigma^2}{\mu}$. If the variance of the gradient estimate were zero, the generated sequence would converge in expectation to the solution at a fast linear rate (i.e., the error from one iterate to the next would decrease exponentially). A gradient estimate with nonzero variance leads to the addition of a constant positive term on the right-hand side, which breaks the monotone decrease of the error and leads the sequence to converge to a ball around the optimum, rather than exactly to the optimum itself.

This means that the method exhibits linear convergence, similar to classical GD, at a rate depending on the learning rate α. This phase continues until the iterates enter a neighborhood of the optimum of radius proportional to $\alpha\sigma^2$; see Figure 4.1 for an illustration. In addition, a large α leads to a fast initial decrease but to a larger steady-state error, while a small value leads to a slighter decrease, but we can eventually come closer to the minimum.

Note that the eigenvalues of the Hessian also impact the convergence. A small $\mu \approx \lambda_{\min}(H(\theta_k))$ makes the linear rate $(1-\alpha\mu)$ close to 1, while it simultaneously amplifies the variance term $(2\alpha/\mu)\sigma^2$. Thus, SGD cannot make progress in flat directions, where the curvature is very small ($\lambda_{\min}(H(\theta_k)) \approx 0$), and instead accumulates noise there.

4.1.2 Convergence for convex and smooth functions

In the following theorem we state an ergodic result for SG, applied to convex functions, with fixed step size [32].

Theorem 2. *Let assumptions 1, 2 and 3 be satisfied. Assume that $\{\theta_k\}_{k\in\mathbb{N}}$ is the sequence generated by SGD (22) with constant step size $0 < \alpha_k = \alpha \leq \frac{1}{4L_{\max}}$, for all k. Define, for every $K \geq 1$,*

$$\bar{\theta}_K := \frac{1}{K}\sum_{k=1}^{K}\theta_k.$$

Then,

$$\mathbb{E}[f(\bar{\theta}_K) - f^\star] \leq \frac{\|\theta_0 - \theta^\star\|^2}{\alpha K} + 2\alpha\sigma^2, \tag{24}$$

where $f^\star = \inf f$. *In particular, if for a fixed* K *we set* $\alpha = \frac{\alpha_0}{\sqrt{K}}$ *with* $\alpha_0 \leq \frac{1}{4L_{\max}}$, *the two terms are balanced, giving*

$$\mathbb{E}[f(\bar{\theta}_K) - f^\star] = \frac{\|\theta_0 - \theta^\star\|^2}{\alpha_0\sqrt{K}} + \frac{2\alpha_0\sigma^2}{\sqrt{K}} = \mathcal{O}\left(\frac{1}{\sqrt{K}}\right).$$

We can see that the decrease is slower than in the strongly convex case (it is sublinear) and that the variance has the same effect as for strongly convex functions. In this case, the largest eigenvalue of the Hessian plays a role. Since the step size α is limited by $\alpha \leq \frac{1}{4L_{\max}} \leq \frac{1}{4\lambda_{\max}(H(\theta_k))}$, a large $\lambda_{\max}(H(\theta_k))$ (and hence a large condition number) forces very small step sizes. Moreover, reducing α also increases the $1/(\alpha K)$ term in (24), delaying convergence.

Once again, convergence cannot be achieved with constant step sizes, but the effect of the variance can be counteracted by a vanishing step size, as shown in the following theorem [32].

Theorem 3. *Let assumptions 1, 2 and 3 be satisfied. Assume that* $\{\theta_k\}_{k\in\mathbb{N}}$ *is the sequence generated by SGD* (22) *with a vanishing step size* $\alpha_k = \alpha_0/\sqrt{k+1}$ *with* $\alpha_0 \leq \frac{1}{4L_{\max}}$, *and define*

$$\bar{\theta}_K := \frac{1}{\sum_{k=0}^{K-1}\alpha_k}\sum_{k=0}^{K-1}\alpha_k\theta_k.$$

Then, for $f^\star = \inf f$,

$$\mathbb{E}[f(\bar{\theta}_K) - f^\star] \leq \frac{5\|\theta_0 - \theta^\star\|^2}{4\alpha_0\sqrt{K}} + \sigma^2\frac{5\alpha_0\log(K+1)}{\sqrt{K}} = \mathcal{O}\left(\frac{\log(K+1)}{\sqrt{K}}\right).$$

4.1.3 Convergence for nonconvex and smooth functions

In this section, we just assume that the stochastic gradients are Lipschitz continuous. Note that in this nonconvex setting, we cannot prove global optimality results. Nevertheless, we can still obtain bounds on the stationarity of the algorithm, stated in the following theorem [12].

Theorem 4. *Let Assumptions 1 and 3 hold. Assume that* $\{\theta_k\}_{k\in\mathbb{N}}$ *is the sequence generated by SGD* (22) *with a constant step size* $0 < \alpha_k = \alpha \leq \frac{1}{L_f}$, *where* L_f *is the Lipschitz constant of the gradient of* f. *Then, for any* $K \geq 1$,

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}[\|\nabla f(\theta_k)\|^2] \leq \frac{2(f(\theta_0) - f^\star)}{\alpha K} + L_f\alpha\sigma^2,$$

where $f^\star = \inf f$. *In particular, choosing* $\alpha = \min\{\frac{1}{L_f}, \frac{1}{\sqrt{K}}\}$, *yields*

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}[\|\nabla f(\theta_k)\|^2] \leq \frac{2(f(\theta_0) - f^\star)}{\sqrt{K}} + \frac{L_f\sigma^2}{\sqrt{K}} = \mathcal{O}\left(\frac{1}{\sqrt{K}}\right).$$

Consequently, for a given $\epsilon > 0$, if $K = \mathcal{O}(\epsilon^{-2})$ then

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}[\|\nabla f(\theta_k)\|^2] = \mathcal{O}(\epsilon).$$

This means that we need $\mathcal{O}(\epsilon^{-2})$ iterations to drive the mean gradient norm below the tolerance ϵ. Note that the constant hidden in the $\mathcal{O}(1/\sqrt{K})$ term depends on $L_f = \sup_\theta |\lambda_{\max}(H(\theta))|$. Thus, a large curvature in some directions (large L_f) makes $\mathcal{O}(1/\sqrt{K})$ progress much slower in practice.

As we have seen, the Hessian's eigenvalues critically affect the theoretical results across all considered settings. Notably, small eigenvalues $\lambda_{\min}(H(\theta))$ lead to slow contraction in the convex setting, while the variance scales as $1/\lambda_{\min}(H(\theta))$, amplifying noise as $\lambda_{\min}(H(\theta))$ decreases. At the same time, large eigenvalues enforce very small step sizes. These observations explain why first-order methods, such as (S)GD, struggle when training ill-conditioned problems, such as PINNs.

4.2 Comparison of computational cost: GD vs. SGD

Since the iterations of SGD are much cheaper than those of classic GD, one may expect that it is always preferable to use SGD. However, the inexactness introduced in the gradient evaluation comes at the cost of a slower convergence rate. For instance, if f is μ-strongly convex, GD is known to converge linearly [9, Theorem 10.29], meaning that

$$f(\theta_k) - f^\star \leq \left(1 - \frac{\mu}{L}\right)^k (f(\theta_0) - f^\star).$$

The total number of iterations to drive the training error below a given threshold $\epsilon > 0$ is thus proportional to $\log(1/\epsilon)$. Since the per-iteration cost is proportional to m (due to the need to compute the full gradient), the total work required to obtain the accuracy ϵ is proportional to $m\log(1/\epsilon)$.

The convergence rate of the SGD method is slower than that of the GD method. For instance, in the strongly convex case with i_k drawn uniformly from $\{1, \ldots, m\}$ and step size $\alpha_k = \mathcal{O}(1/k)$, SGD converges sublinearly in expectation [12, Theorem 4.7]:

$$\mathbb{E}(f(\theta_k) - f^\star) = O(1/k).$$

However, the per-iteration cost of SGD does not depend on the dataset size m, making it attractive in large-scale settings. Specifically, the total work to reach accuracy ϵ is proportional to $1/\epsilon$ for SGD, compared to $m\log(1/\epsilon)$ for GD. Thus, while SGD may be less efficient for small to moderate m, it becomes increasingly advantageous as m grows large.

If the gradient estimate is computed using a mini-batch of data, a different trade-off between cost and convergence rate arises. The per-iteration cost scales

with the mini-batch size, while the variance of the gradient estimate decreases relative to plain SGD, leading to improved convergence. Thus, the main difference between these methods lies in step quality: GD performs fewer but more accurate and more expensive steps, whereas SGD takes many cheap but noisy steps, and mini-batch methods provide a middle ground.

The same comparison applies to memory requirements. In particular, both GD and SGD require $\mathcal{O}(n)$ memory for the parameters and the gradient estimate. Storing the full dataset requires $\mathcal{O}(md)$ memory, where d is the per-sample feature dimension. In the mini-batch setting, only the current $|\mathcal{I}_k|$ samples need to reside in memory, which is the practical reason why mini-batching is the default strategy once m samples no longer fit in GPU memory.

Numerical Example #4: *SGD implementation.* This numerical example demonstrates how the theoretical results discussed in Section 4.1 manifest in practice. To illustrate the behaviour of SGD, we consider the logistic regression problem introduced in Section 2.1 with a synthetic dataset. Figure 4.2 shows the effect of the mini-batch size on the behaviour of the SGD method. With a fixed learning rate and very small batches, SGD initially makes rapid progress but soon stagnates as the stochastic variance in the gradient estimates becomes dominant, preventing a consistent decrease in the loss. In contrast, larger batches reduce the sampling noise and produce search directions that more closely approximate the full gradient. A prototypical implementation of the SGD in NumPy can be found in Code snippet 10.

```
def run_sgd(X, y, gamma=1e-4, epochs=300, batch_size=8,
    alpha0=1e-3, scheduler=None, seed=0, f_opt=None):
   # Run SGD with a fixed batch size and (optional) learning-rate scheduler

    n, d = X.shape # Number of sample (n) and feature dimension (d)
    rng = np.random.default_rng(seed) # Set seed for reproducibility
    w = np.zeros(d) # Initial guess for parameters

    for ep in range(epochs): # Run over epochs
        #  Use learning-rate scheduler. If None, fixed at alpha0
        lr = alpha0 if scheduler is None else scheduler(ep, alpha0)

        idx = rng.permutation(n)
        Xs, ys = X[idx], y[idx] # Suffle samples

        for start in range(0, n, batch_size): # Run over all mini-batches
            end = min(start + batch_size, n) # Get mini-batch
            # SGD step direction
            d_step = -lr * grad_logloss(w, Xs[start:end], ys[start:end],
     gamma)
            w = w + d_step  # Update the parameters
```

Code snippet 10: Example of SGD implementation in NumPy.

In practice, two standard mechanisms can be used to mitigate the effect of the variance of the stochastic search directions, and to drive the residual

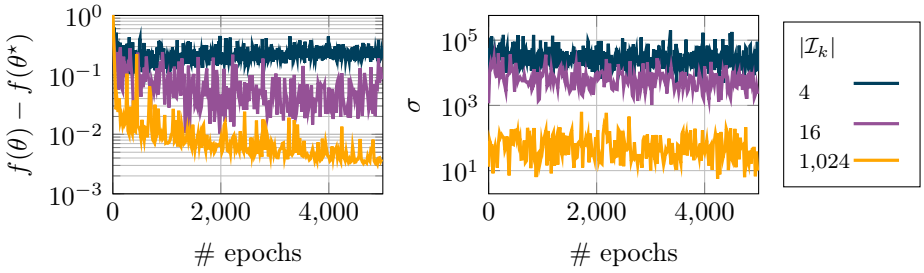

Figure 4.2: Effect of batch size on convergence of the mini-batch GD method with fixed $\alpha_k = 10^{-3}$. *Left:* Suboptimality $f(\theta_k) - f(\theta^\star)$ over epochs for three batch sizes ($|\mathcal{I}| = \{4; 16; 1,024\}$). *Right:* Corresponding relative variance of subsampled gradient.

variance-related term in the convergence results to zero. Their effect is illustrated in Figure 4.3. The first mechanism is *learning rate scheduling*, where the step size α_k is incrementally reduced during training, see the first function in Code snippet 11 for a possible implementation. In our experiments, we employ a *polynomial decay* scheduler of the following form:

$$\alpha_k = \frac{\alpha_0}{(1 + \tau k)^{pw}},$$

with initial learning rate $\alpha_0 = 10^{-3}$, decay coefficient $\tau = 2{\cdot}10^{-2}$, and exponent $pw = 1$. Reducing the learning rate decreases the magnitude of stochastic perturbations and, as predicted theoretically, it enforces a gradual reduction of variance [99, 12]. Note that other schedules, such as step decay, exponential decay, or cosine annealing, are commonly used in modern practice [67].

```python
def poly_scheduler(epoch, alpha0, decay=5e-3, power=1):
    # Polynomial decay learning-rate schedule
    return alpha0 / ((1 + decay * epoch) ** power)

def batch_size_scheduler(epoch, bs_start, bs_end, frac):
   # Linear batch scheduler from bs_start to bs_end at frac rate
        frac = epoch / (epochs - 1) if epochs > 1 else 1.0
        bs = int(bs_start + frac * (bs_end - bs_start))
        return max(1, min(bs, n))
```

Code snippet 11: Example of implementation of a polynomial learning-rate scheduler and a progressive batch-size scheduler.

The second mechanism is intended to reduce the variance of the gradient estimates by a *batch-size scheduler*, which increases the accuracy of each gradient estimate by progressively increasing the size of the batches, see the second function provided in Code snippet 11 for a possible implementation. In this way, early steps are cheap, computed on a few samples, and become increasingly ac-

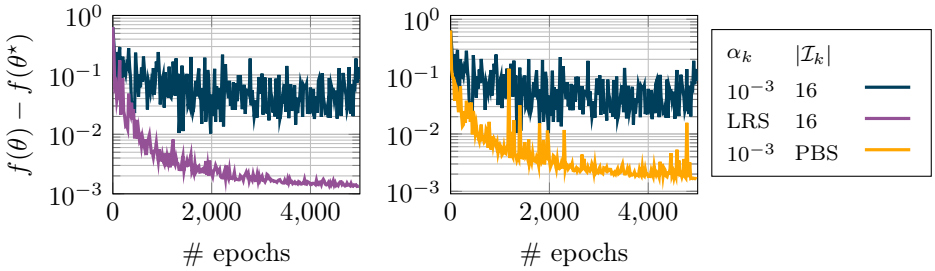

Figure 4.3: Example of reducing SGD noise by using learning rate scheduling (LRS) and progressive batch-size (PBS) increase. The SGD (blue) utilizes $|\mathcal{I}_k| = 16$ and $\alpha_k = 10^{-3}$. The SGD with LRS (purple) utilizes polynomial decay of α_k, while SGD with PBS progressively increases the batch size $|\mathcal{I}_k|$.

curate but more expensive as the iterations approach a minimum, allowing for a more precise final solution. In our experiment, the mini-batch size evolves linearly from $|\mathcal{I}_0| = 16$ to $|\mathcal{I}_T| = 1,024$ over T epochs as

$$|\mathcal{I}_k| \;=\; |\mathcal{I}_0| \;+\; \frac{k}{T}\big(|\mathcal{I}_T| - |\mathcal{I}_0|\big), \qquad k = 0, \dots, T.$$

The learning rate is kept constant at $\alpha_k = \alpha_0$, enabling progress in regimes where small-batch SGD with a fixed step size would stagnate [108, 104].

Figure 4.3 demonstrates the performance of these two approaches, compared to SGD with $\alpha_k = 10^{-3}$ and $|\mathcal{I}_k| = 16$. As we can see, both learning rate decay and batch-size growth strategies effectively reduce the variance of the search direction, thereby restoring convergence. However, they differ subtly in practice: learning rate decay improves stability but slows down the method permanently, whereas batch-size growth preserves the step magnitude and reduces noise without sacrificing asymptotic speed, but progressively increases the iteration cost.

Key Takeaways

Stochastic gradient descent (SGD) and its mini-batch variants minimize finite-sum objectives by replacing the full gradient with a cheap estimate computed on small random subsets of data. The *variance* of this estimate governs convergence. In particular, large variance limits the attainable accuracy with fixed step sizes, while small variance permits larger steps and faster progress. Two standard strategies to counteract the effect of variance are *step size decay* and *batch size growth*, both of which restore convergence in expectation, at the cost of slower iterations or higher per-iteration cost, respectively.

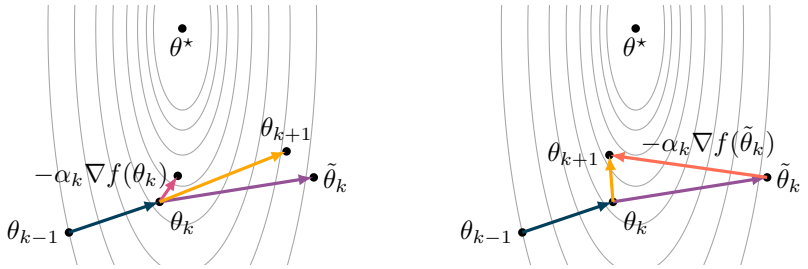

Figure 5.1: Sketch of heavy-ball momentum (left) and NAG (right) iteration.

5 Accelerated and adaptive stochastic gradient methods

The main limitation of standard first-order methods is their slow convergence, often attributed to the difficulty of selecting an appropriate step size. This issue becomes even more pronounced in large-scale SciML, where physical constraints, large models, noisy gradients, and heterogeneous data make choosing a suitable step size both challenging and essential for stable and efficient training. To address these challenges, a wide range of accelerated and adaptive variants of GD and SGD have been developed. In this section, we present the most widely used ones. From the unifying preconditioning viewpoint introduced in Section 3.1, the adaptive gradient methods discussed below act as diagonal preconditioners in parameter space.

5.1 Momentum

The purpose of momentum is to mitigate a key limitation of GD, namely that the parameter update at iteration k only depends on local information on the gradient, and does not exploit the history of previous updates. Incorporating past information can provide valuable insight into the objective function landscape along the optimization trajectory and can significantly improve convergence.

Momentum techniques keep memory of past gradients and accumulate them. To account for time, they use an exponentially weighted average to give more weight to recent gradients. The update at iteration k then becomes

$$p_k = \beta\, p_{k-1} + (1-\beta)\, g_k,$$

where p_k is the new search direction and g_k is any gradient approximation at iteration k. The momentum parameter $\beta \in (0,1)$ controls the weighting between past and current gradients. The gradient approximation g_k is typically obtained by sub-sampling the training set, and may correspond to the full gradient ($|\mathcal{I}_k| = m$), a mini-batch gradient ($1 < |\mathcal{I}_k| < m$), or a stochastic gradient ($|\mathcal{I}_k| = 1$).

The term $(1-\beta)$ is often replaced with the learning rate, and a dynamically adjusted parameter β_k may be used instead of a fixed β to scale p_{k-1}, so the

general update is given as

$$\theta_{k+1} = \theta_k - \alpha_k g_k + \beta_k(\theta_k - \theta_{k-1}), \tag{25}$$

where the scalar sequence $\{\alpha_k\}$ is either predetermined or set dynamically.

Heavy ball If $\alpha_k = \alpha$ and $\beta_k = \beta$ for some constants $\alpha > 0$ and $\beta \in (0, 1)$ and for all $k \in \mathbb{N}$, the update (25) is referred to as the *heavy ball method*. An alternative interpretation of this method is obtained by expanding the update:

$$\theta_{k+1} = \theta_k - \alpha \sum_{j=1}^{k} \beta^{k-j} g_j.$$

Thus, each step can be viewed as an exponentially weighted average of past gradients. Notably, larger values of β place greater emphasis on gradients from earlier iterations. In practice, β is typically chosen close to 0.9.

Nesterov accelerated gradient (NAG) Another widely used variant of momentum is the Nesterov accelerated gradient (NAG). The core idea behind NAG is that if the current parameter vector is θ_k, then the momentum term alone (i.e., ignoring the term with the gradient) is about to nudge the parameter vector by $\beta_k(\theta_k - \theta_{k-1})$. Therefore, it makes sense to compute the gradient at $\theta_k + \beta_k(\theta_k - \theta_{k-1})$, instead of at the old position θ_k.

Written as a two-step procedure, the NAG update is given as

$$\tilde{\theta}_k = \theta_k + \beta_k(\theta_k - \theta_{k-1}) \qquad \text{and} \qquad \theta_{k+1} = \tilde{\theta}_k - \alpha_k \nabla f(\tilde{\theta}_k),$$

which can be reformulated as

$$\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k + \beta_k(\theta_k - \theta_{k-1})) + \beta_k(\theta_k - \theta_{k-1}). \tag{26}$$

The main difference with classical momentum is thus the order of computation (momentum step versus gradient step), see also Figure 5.1.

Cost and convergence The cost of (S)GD methods with momentum is comparable to that of classical (S)GD, but their convergence is generally faster. For example, in the deterministic strongly convex case, the convergence rate of GD (cf. Theorem 1 with $\sigma = 0$) improves from $\mathcal{O}((1-\frac{\mu}{L})^k)$ to $\mathcal{O}((1-\sqrt{\frac{\mu}{L}})^k)$, see[90] for details. In the stochastic case, the convergence neighborhood size increases from $\mathcal{O}(\alpha\sigma^2)$ (cf. Theorem 1) to $\mathcal{O}(\frac{\alpha\sigma^2}{1-\beta})$, where α and β denote the step size and the momentum parameter, respectively. Thus, momentum improves the convergence rate but enlarges the noise-dominated convergence neighborhood, which can degrade the final accuracy [123].

5.2 Adaptive gradient algorithms: AdaGrad, Adam

To overcome the sensitivity of SGD to the choice of the learning rate, several adaptive optimization algorithms have been proposed. These methods adjust the learning rate of each parameter individually based on past gradient information. The most notable examples are AdaGrad and Adam.

Adaptive Gradient(AdaGrad) [28] AdaGrad updates parameters θ at iteration k in a component-wise manner, i.e.,

$$\theta_{k+1}(i) = \theta_k(i) - \frac{\alpha}{\sqrt{G_k(i) + \epsilon}}\, g_k(i), \tag{27}$$

where the symbol (i) denotes the i-th component of a vector. The symbol α denotes the global learning rate, and ϵ is a small constant to avoid division by zero. The vector g_k stands for the gradient approximation at iteration k and G_k contains the sum of squares of past gradients for each parameter, i.e., $G_k(i) = \sum_{j=1}^{k} g_j(i)^2$. All together, this gives rise to a preconditioned gradient method with diagonal M_k, where $(M_k)(i,i) = \sqrt{G_k(i)} + \epsilon$.

AdaGrad gives larger updates to infrequent parameters and smaller updates to frequent ones, which can be particularly useful for sparse data. However, the accumulated squared gradients in G_k increase indefinitely and can cause the effective learning rate to decrease excessively over time.

Adaptive Moment Estimation (Adam) [54] Adam addresses AdaGrad's vanishing learning rate by maintaining exponentially decaying averages of past gradients (first moment) and squared gradients (second moment):

$$\begin{aligned} m_k(i) &= \beta_1 m_{k-1}(i) + (1-\beta_1) g_k(i), \\ v_k(i) &= \beta_2 v_{k-1}(i) + (1-\beta_2)(g_k(i))^2, \end{aligned}$$

which are then bias-corrected as

$$\hat{m}_k(i) = \frac{m_k(i)}{1-\beta_1^k}, \qquad\qquad \hat{v}_k(i) = \frac{v_k(i)}{1-\beta_2^k},$$

where β_1 and β_2 are decay rates for the moving averages. The bias correction is necessary because m_k and v_k are initialized at zero, causing their early estimates to be biased toward zero. The factors $(1-\beta_1^k)^{-1}$ and $(1-\beta_2^k)^{-1}$ compensate for this effect by rescaling the estimates to the appropriate magnitude; their influence vanishes as $k \to \infty$.

The moments $\hat{m}_k$ and $\hat{v}_k$ are then used to update the current iterate in the following, component-wise, manner:

$$\theta_{k+1}(i) = \theta_k(i) - \alpha \frac{\hat{m}_k(i)}{\sqrt{\hat{v}_k(i)} + \epsilon}, \tag{28}$$

giving a diagonal preconditioned update with $(M_k)(i,i) = (\sqrt{\hat{v}_k(i)} + \epsilon)/\hat{m}_k(i)$. Adam maintains an exponentially decaying moving average of squared gradients, which remains bounded and thus avoids the vanishing learning-rate issue inherent to AdaGrad's cumulative accumulation of squared gradients. As a result, Adam combines the benefits of AdaGrad's per-parameter learning rates with momentum's acceleration, making it robust to noisy or sparse gradients. It is nowadays implemented in all the ML libraries, and it is one of the most widely used optimizers.

Cost and convergence Both AdaGrad and Adam have the same per-iteration time complexity as (S)GD, with a small memory overhead. AdaGrad stores the accumulated squared-gradient sum G_k, while Adam stores the first and second moments m_k and v_k.

Regarding convergence, AdaGrad is fairly well understood. For convex Lipschitz functions the rate is $\mathcal{O}(K^{-1/2})$ [28, 127], improving to $\mathcal{O}(\log(K)/K)$ for strongly convex functions. In the nonconvex setting, AdaGrad achieves $\min_{k \le K} \|\nabla f(\theta_k)\|^2 = \mathcal{O}(K^{-1/2})$ [24], matching the optimal SGD rate under comparable assumptions.

The convergence theory for Adam is more involved. Even for convex problems, the original Adam scheme can fail to converge [97], as aggressive decay of the second-moment estimate can discard useful historical gradient information. Modern variants exist, such as AMSGrad [97], which enjoy convergence guarantees but tend to underperform Adam empirically. Despite its incomplete theoretical foundation, Adam remains one of the most effective and widely used optimizers in practice.

Numerical Example #5: *SGD, NAG, AdaGrad and Adam.* We now empirically compare the performance of SGD, NAG, AdaGrad, and Adam, on the same logistic-regression problem as considered in Numerical Example #4. All methods are run in the full-batch regime ($|\mathcal{I}_k| = 6,000$), ensuring that differences in convergence arise solely from the update rule rather than stochastic sampling effects. Step sizes are chosen to guarantee stable behavior: SGD and NAG use a small step size $\alpha_k = 10^{-4}$, whereas AdaGrad and Adam employ significantly larger values $\alpha_k = 10^{-2}$. Code snippet 12 provides a prototypical implementation of the four optimizers used in this comparison.

```
def run_optimizer( X, y, lam=1e-4, epochs=300, batch_size=16,
     optimizer="SGD", eta0=1e-3,beta=0.9, beta1=0.9, beta2=0.999, eps=1e-8,
     seed=0, f_opt=None):
    # Unified driver that runs SGD / Momentum / NAG / AdaGrad / Adam
    # At each iteration a mini-batch gradient g is computed and an update
     rule is applied according to `optimizer`

    n, dim = X.shape # Dimension of input features and parameters
    rng = np.random.default_rng(seed) # Seed for reproducibility
    w = np.zeros(dim) # Initialization of parameters

    # State variables
```

```
    v = np.zeros_like(w)         # Momentum / NAG velocity
    G = np.zeros_like(w)         # AdaGrad sum of squared grads
    m = np.zeros_like(w)         # Adam first moment
    v2 = np.zeros_like(w)        # Adam second moment
    k = 0                        # Global step counter


    for ep in range(epochs): # Run over epochs
        idx = rng.permutation(n) # Suffle dataset
        Xs, ys = X[idx], y[idx]

        for start in range(0, n, batch_size): # Run over all batches
            end = min(start + batch_size, n)
            Xb, yb = Xs[start:end], ys[start:end] # Select mini-batch
            k += 1

            if(optimizer != "NAG"):
              # Evaluate gradient using current iterate w and mini-batch
                g = grad_logloss(w, Xb, yb, lam)

            if optimizer == "SGD":
                step = -eta0 * g # SGD search direction evaluation

            elif optimizer == "momentum":
                v = beta * v + g # Momentum evaluation
                step = -eta0 * v  # SGD with momentum step

            elif optimizer == "NAG":
                # Computation of look-ahead point
                w_look = w - eta0 * beta * v
                # Evaluation of gradient at look-ahead point
                g = grad_logloss(w_look, Xb, yb, lam)
                v = beta * v + g # Momentum computation
                step = -eta0 * v # NAG step

            elif optimizer == "AdaGrad":
                G += g * g # Updating gradient history
                step = -eta0 * g / (np.sqrt(G) + eps) # Adagrad step

            elif optimizer == "Adam":
                m = beta1 * m + (1 - beta1) * g # First-moments
                v2 = beta2 * v2 + (1 - beta2) * (g * g) # Second-moments
                m_hat = m / (1 - beta1 ** k) # Bias corrected first-moments
                v_hat = v2 / (1 - beta2 ** k) # Bias corrected second-moments
                step = -eta0 * m_hat / (np.sqrt(v_hat) + eps) # Adam step

            else:
                raise ValueError("Unknown optimizer")

            w += step # Parameter update with search direction (step)
```

Code snippet 12: Implementation of SGD, momentum, NAG, AdaGrad, and Adam in NumPy.

Figure 5.2 shows that standard SGD exhibits the slowest convergence. NAG

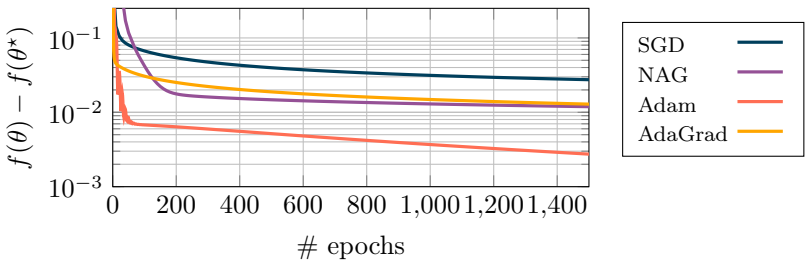

Figure 5.2: Convergence of SGD, NAG, AdaGrad, and Adam on the logistic regression problem of Numerical Example #4.

and AdaGrad achieve faster progress during the initial phase, but both ultimately remain limited by the ill-conditioning. Adam, by contrast, exhibits the fastest overall convergence, demonstrating that adaptive preconditioning can stabilize training while enabling the use of larger constant learning rates.

We emphasize that optimal step size choices and the relative performance of these optimizers are problem-dependent and must be tuned in practice. Moreover, in deep-learning applications, methods that progress slowly or conservatively (such as SGD) may explore the landscape more thoroughly and often yield solutions with better generalization performance. As a result, identifying the most suitable optimization method and tuning it appropriately remains a key practical challenge in ML workflows.

Key Takeaways

Momentum techniques are designed to improve the convergence rate of (S)GD by incorporating past information in the update. Adaptive gradient methods can be thought of as preconditioned gradient methods, i.e., their updates are obtained by the gradient one, premultiplied by an invertible matrix intended to approximate the inverse of the Hessian, to include curvature information and speed up convergence. In adaptive gradient methods, the preconditioner (the invertible matrix) is diagonal and built from only first-order information (first- and second-order moments of the gradients).

6 Beyond first-order optimization

While first-order methods such as SGD and Adam dominate modern ML, they often struggle in SciML settings, motivating the use of curvature-aware approaches. Second-order methods, such as Newton's method, can converge much faster near a local minimizer, which is particularly beneficial in SciML, where the loss landscape is often stiff and highly anisotropic due to underlying differ-

ential operators. Coupled with *globalization strategies*, such as line search [86] or trust-region methods [18], these methods are ensured to convergence from arbitrary starting points. However, their high computational cost makes them impractical at scale, leading to the development of scalable alternatives such as Hessian-free inexact Newton, subsampled Newton, or quasi-Newton methods.

6.1 Hessian-free inexact Newton–Krylov methods

Recall that the Newton step (12) requires solving the linear system

$$H(\theta_k)\, p_k = -\nabla f(\theta_k) \tag{29}$$

for the search direction p_k, which is computationally challenging due to both the memory and computational cost of forming and inverting $H(\theta_k)$.

Hessian-free inexact Newton–Krylov methods [57] provide a scalable means of incorporating curvature information without explicitly forming or storing the Hessian. Rather than solving (29) exactly, these methods compute an approximate Newton step satisfying

$$\|H(\theta_k)p_k + \nabla f(\theta_k)\| \leq \eta \,\|\nabla f(\theta_k)\|,$$

with $\eta > 0$, using an iterative Krylov solver (e.g., conjugate gradient), which requires only matrix-vector products $H(\theta_k)v$. If the tolerance η is progressively tightened as the iterations proceed, inexact Newton methods achieve superlinear convergence [25]. A concrete example is the *Eisenstat–Walker* choice of η, given as [29]

$$\eta_k = \frac{|\,\|\nabla f(\theta_k)\| - \|\nabla f(\theta_{k-1}) + H(\theta_{k-1})p_{k-1}\|\,|}{\|\nabla f(\theta_{k-1})\|},$$

which adapts η_k automatically based on how well the previous linear system was solved, avoiding both over-solving and under-solving.

A key advantage of Newton–Krylov methods is that Krylov solvers require only Hessian-vector products. This is particularly attractive in ML, where Hessian-vector products can often be computed efficiently using automatic differentiation or adjoint techniques. Such ideas have recently been employed for regression problems in SciML [128], demonstrating higher model accuracy or lower computational cost than adaptive first-order methods.

Cost and convergence Each Newton–Krylov iteration is more expensive than a GD iteration, as it requires several inner Krylov iterations, each involving two Hessian-vector products of cost comparable to a gradient evaluation. In practice, only a few inner iterations suffice to achieve the required accuracy on the Newton direction, and the improved convergence behavior typically compensates for the additional per-iteration cost. The memory footprint remains comparable to that of first-order methods, since the Hessian is never explicitly formed, making Newton–Krylov one of the few practically scalable second-order approaches for large DNNs.

Since fast local convergence is guaranteed only in a neighborhood of a solution, these methods are often used in combination with globalization strategies to ensure convergence from arbitrary starting points.

6.2 Subsampled Hessian-free inexact Newton methods

Analogous to mini-batch gradient methods, the computational cost of Hessian-free inexact Newton–Krylov methods can be further reduced by employing subsampled Hessian [100]. In this case, mini-batches are used both for the gradient and for the Hessian approximations, and the step is given as

$$H_{\mathcal{I}_k^H}(\theta_k)p_k = -\nabla f_{\mathcal{I}_k}(\theta_k),$$

where we used index sets $\mathcal{I}_k \subset \{1,\dots,m\}$ and $\mathcal{I}_k^H \subset \{1,\dots,m\}$ to obtain the stochastic gradient and stochastic Hessian estimates as

$$\nabla f_{\mathcal{I}_k}(\theta_k) = \frac{1}{|\mathcal{I}_k|}\sum_{i\in\mathcal{I}_k}\nabla f_i(\theta_k), \qquad H_{\mathcal{I}_k^H}(\theta_k) = \frac{1}{|\mathcal{I}_k^H|}\sum_{i\in\mathcal{I}_k^H}\nabla^2 f_i(\theta_k).$$

Here, we assumed that the mini-batch $\mathcal{I}_k^H$ is uncorrelated with the mini-batch $\mathcal{I}_k$ and $|\mathcal{I}_k^H| < |\mathcal{I}_k|$, since the iteration scheme is more tolerant to noise in the Hessian estimate than it is to noise in the gradient estimate.

If one chooses the subsample size $|\mathcal{I}_k^H|$ small enough, then the cost of each product involving the Hessian approximation can be reduced significantly, thus reducing the cost of each Krylov iteration. Algorithm 2 summarizes the subsampled Newton method. The choice of the step size in this context is often done via line-search strategies [86, ch.3], adaptive techniques that ensure convergence of the method for any starting point.

Similar to stochastic first-order methods, subsampled Newton methods are not particularly well-suited for physics-informed problems due to the global coupling induced by the PDE constraints. However, they may be well suited for data-driven SciML problems, where the loss decomposes over independent training samples, e.g., operator learning, DNN surrogate models trained on simulation data, or latent-space representations learned from large collections of solution snapshots. In such cases, the Hessian admits a natural sample-wise decomposition, making subsampled Newton methods a potentially effective yet largely unexplored alternative to (adaptive) first-order optimization.

Algorithm 2 Subsampled Hessian-free Inexact Newton Method

1: **Given:** A dataset $\{x_i, y_i\}_{i=1}^m$, an objective function $f : \mathbb{R}^n \to \mathbb{R}$, an initial iterate $\theta_0 \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Choose $\mathcal{I}_k^H$ and construct $H_{\mathcal{I}_k^H}(\theta_k)$ ▷ Hessian approximation
4: Choose $\mathcal{I}_k$ and $\nabla f_{\mathcal{I}_k}(\theta_k)$ ▷ Gradient approximation
5: Obtain p_k by approximately solving ▷ Inexact step computation

$$H_{\mathcal{I}_k^H}(\theta_k) p_k = -\nabla f_{\mathcal{I}_k}(\theta_k)$$

6: Choose α_k ▷ Step size selection
7: Set $\theta_{k+1} = \theta_k + \alpha_k p_k$ ▷ Iterate update
8: **end for**

6.3 Quasi-Newton methods

Quasi-Newton methods use the same update rule as Newton's method, but instead of the exact Hessian $H(\theta_k)$, they employ an approximation B_k. The update is then given as $\theta_{k+1} = \theta_k + \alpha_k p_k$, where p_k is such that $B_k p_k = -\nabla f(\theta_k)$. The most well-known quasi-Newton method is perhaps BFGS [86, 65], which allows for directly approximating the inverse of the Hessian $\tilde{B}_k$, so that the iteration update further simplifies by avoiding the need to solve a linear system. Thus, the search direction is obtained as

$$p_k = -\tilde{B}_k \nabla f(\theta_k).$$

At iteration k, the inverse Hessian approximation $\tilde{B}_k$ is constructed recursively, starting from an initial approximation $\tilde{B}_0$, according to

$$\tilde{B}_{k+1} = (I - \varrho_k s_k y_k^T)\, \tilde{B}_k\, (I - \varrho_k y_k s_k^T) + \varrho_k s_k s_k^T, \qquad \varrho_k = \frac{1}{y_k^T s_k}, \tag{30}$$

where

$$s_k := \theta_{k+1} - \theta_k, \qquad y_k := \nabla f(\theta_{k+1}) - \nabla f(\theta_k).$$

If $\tilde{B}_k$ is symmetric positive definite and the curvature condition $y_k^T s_k > 0$ holds, then $\tilde{B}_{k+1}$ is also symmetric positive definite.

Algorithm 3 L-BFGS two-loop recursion

Given: Current gradient $\nabla f(\theta_k)$, pairs $\{s_i, y_i\}_{i=1}^{S}$, $\varrho_i = 1/(y_i^T s_i)$, initial approximation of the inverse of the Hessian $\tilde{B}_k^0$
Set $q = \nabla f(\theta_k)$
for $i = k-1, k-2, \ldots, k-S$ **do**
    Compute $\alpha_i = \varrho_i s_i^T q$
    Update $q = q - \alpha y_i$
end for
Set $r = \tilde{B}_k^0 q$
for $i = k-S, k-S+1, \ldots, k-1$ **do**
    Compute $\beta = \varrho_i y_i^T r$
    Update $r = r + s_i(\alpha_i - \beta)$
end for
return r

L-BFGS For large-scale problems, storing the full matrix $\tilde{B}_k$ is infeasible. This motivates the use of a limited-memory variant, termed L-BFGS, which stores only S most recent secant pairs $\{(s_i, y_i)\}_{i=k-S}^{k-1}$ instead of the full $\tilde{B}_k$. These pairs suffice to approximate the application of $\tilde{B}_k$ to $\nabla f(\theta_k)$ in order to compute the search direction p_k, in turn reducing storage requirements from $O(n^2)$ to $O(nS)$, where n is the number of variables.

To derive an efficient procedure for applying $\tilde{B}_k$ to $\nabla f(\theta_k)$, we rewrite the BFGS inverse update (30) in the compact form as

$$\tilde{B}_{k+1} = V_k^T \tilde{B}_k V_k + \varrho_k s_k s_k^T, \qquad V_k := I - \varrho_k y_k s_k^T. \tag{31}$$

As a consequence, the product $\tilde{B}_k \nabla f(\theta_k)$ can be computed using only inner products and vector additions involving the stored secant pairs $\{(s_i, y_i)\}_{i=k-S}^{k-1}$. Thus, by repeatedly applying (31), we obtain

$$\begin{aligned}\tilde{B}_k = {} & (V_{k-1}^T \cdots V_{k-S}^T)\, \tilde{B}_k^0\, (V_{k-S} \cdots V_{k-1}) \\ & + \varrho_{k-S}(V_{k-1}^T \cdots V_{k-S+1}^T)\, s_{k-S} s_{k-S}^T\, (V_{k-S+1} \cdots V_{k-1}) \\ & + \varrho_{k-S+1}(V_{k-1}^T \cdots V_{k-S+2}^T)\, s_{k-S+1} s_{k-S+1}^T\, (V_{k-S+2} \cdots V_{k-1}) \\ & + \cdots + \varrho_{k-1} s_{k-1} s_{k-1}^T.\end{aligned}$$

This expression leads to a recursive scheme for computing $\tilde{B}_k \nabla f(\theta_k)$ efficiently, namely the two-loop recursion summarized in Algorithm 3. Assuming $\tilde{B}_k^0$ is diagonal, the two-loop recursion requires approximately $4Sn + n$ multiplications. A common choice for $\tilde{B}_k^0$ is a scaled identity $\tilde{B}_k^0 = \tau_k I$, with $\tau_k > 0$. Parameter τ_k is often chosen as $\tau_k = \frac{s_{k-1}^\top y_{k-1}}{y_{k-1}^\top y_{k-1}}$, which matches the local curvature information from the most recent quasi-Newton pair, see [85] for details.

Algorithm 4 outlines the L-BFGS procedure. Note that when using L-BFGS, full gradients are typically required, as the method is sensitive to noise in gradient estimates. While noise-robust variants of BFGS and L-BFGS have been

proposed [105, 46], this is not a significant limitation for physics-based models, which are usually trained in deterministic settings due to global coupling induced by PDE constraints. Consequently, L-BFGS has emerged as one of the most widely used optimizers for PINNs, see a comprehensive benchmark of quasi-Newton methods for PINNs provided in [55].

Cost and convergence Each L-BFGS iteration costs $\mathcal{O}(nS)$, compared to $\mathcal{O}(n)$ for GD, as it applies the two-loop recursion over S stored curvature pairs $(s_i, y_i) \in \mathbb{R}^n \times \mathbb{R}^n$. The memory footprint is $\mathcal{O}(2nS)$, which for a typical choice of $S = 2$–5 is comparable to Adam's within a small constant factor, while avoiding the $\mathcal{O}(n^2)$ cost of full BFGS.

Despite modest overhead, L-BFGS typically converges much faster than GD, often superlinearly in practice. As the method relies on the positive definiteness of the Hessian approximation, a line search satisfying the Wolfe conditions [86] is standard, ensuring both sufficient decrease and the curvature condition required for the two-loop recursion to remain well-defined.

Algorithm 4 L-BFGS

Given: A dataset $\{x_i, y_i\}_{i=1}^m$, an objective function $f : \mathbb{R}^n \to \mathbb{R}$, an initial iterate $\theta_0 \in \mathbb{R}^n$, a memory size $S > 0$
for $k = 0, 1, 2, \ldots$ **do**
  Choose $\tilde{B}_k^0$ ▷ Initial inverse Hessian approximation
  Compute $p_k = -\tilde{B}_k \nabla f(\theta_k)$ by using Algorithm 3 ▷ Step computation
  Choose α_k ▷ Step size selection
  $\theta_{k+1} = \theta_k + \alpha_k p_k$ ▷ Iterate update
  if $k > S$ **then** ▷ Secant pairs update
    Discard $\{(s_{k-S}, y_{k-S})\}$ from the storage
    Compute and save $s_k = \theta_{k+1} - \theta_k$, $y_k = \nabla f(\theta_{k+1}) - \nabla f(\theta_k)$
  end if
end for

6.4 Switching from first- to second-order optimization

Adaptive first-order optimizers such as Adam are highly effective during the early stages of training due to their ability to handle poor gradient scaling, strong anisotropy, and stochastic noise. However, their adaptivity can bias the optimization trajectory and limit the asymptotic accuracy of the learned solution [124, 53, 43]. In contrast, quasi-Newton methods are often more effective in later stages of training, when the terms in (9) vary more smoothly along the optimization trajectory and curvature information can be exploited reliably.

This complementarity motivates hybrid training schedules that combine the robustness of first-order adaptive optimizers with the precision and fast local convergence of quasi-Newton methods. The most common strategy in the PINN literature is the two-stage *Adam → L-BFGS* scheme [95, 118]. Adam provides a robust initialization phase, bringing the parameters θ into a favorable region of

the loss landscape. Once the optimization enters a smoother regime, L-BFGS then refines the solution using curvature-aware updates.

Because the optimal switching point is problem-dependent, switching too early leads to unstable curvature estimates, whereas switching too late wastes iterations. To address this, several heuristics have been proposed to determine the suitable transition point in practice. For example, one might observe that the gradient norm $\|\nabla_\theta f(\theta_k)\|$ is typically noisy during early Adam iterations but stabilizes once the loss landscape becomes smoother. Hence, a persistent plateau in $\|\nabla_\theta f(\theta_k)\|$ over several epochs indicates readiness to switch to L-BFGS [118, 76]. Alternatively, one might explore approaches based on PDE residual saturation [118, 76] or stabilization of adaptive learning rates [118, 101].

Numerical Example #6: *Training of PINNs.* Figure 6.1 illustrates the contrasting optimization behaviors of Adam, L-BFGS, and a hybrid Adam→L-BFGS strategy for a representative PINN example (1D Poisson problem). An implementation of this hybrid strategy is provided in Code Snippet 13. As we can see, Adam often achieves rapid loss reduction during the early, noisy, and highly anisotropic phase of training. However, its progress may slow down once the optimization enters a more stable regime where curvature effects dominate. Standard L-BFGS, when started from the same initialization, fails to make progress at a certain point due to unstable curvature estimates. In contrast, the hybrid Adam→L-BFGS schedule automatically transitions when the gradient norm stabilizes, enabling L-BFGS to exploit accurate curvature information and achieve substantially faster convergence and improved accuracy. Here, the switch from Adam to L-BFGS is triggered when the average gradient norm over 200 consecutive iterations decreases by less than 0.5%.

```
def train_adam_then_lbfgs_automatic_switch(model, x, f_rhs, u_exact,
 adam_max_iters=8000, adam_min_iters=500, lbfgs_iters=1000, lr_adam=1e-4,
 window=200, plateau_tol=0.995, history_size=50, tol_grad=1e-12,
 tol_change=1e-12):
# Train PINN with Adam first, then automatically switch to L-BFGS on
 convergence plateau

# Use Adam optimizer of Pytorch to train parameters of model
opt_adam = optim.Adam(model.parameters(), lr=lr_adam)

def grad_norm():
    # Compute L2 norm of gradients
    return sum(p.grad.norm().item()**2 for p in model.parameters()
               if p.grad is not None)**0.5

# ----- Adam stage: early exploration -----
for k in range(adam_max_iters):
    opt_adam.zero_grad() # Zero gradient storage in Adam
    # Eval derivative required to evaluate PDE
    uxx = pinn_second_derivative(model, x)
    loss = loss_fn(uxx, f_rhs) # Compute PINN loss
    loss.backward() # Perform back-propagation to get gradient

    # Adam optimizer step
```

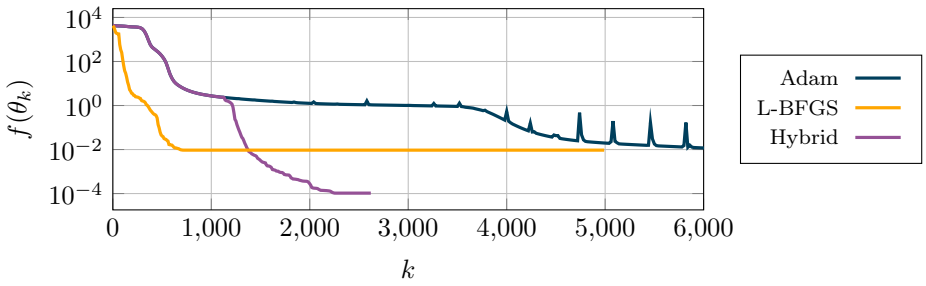

Figure 6.1: PINN training for Poisson problem in 1D using Adam, L-BFGS, and a hybrid Adam→L-BFGS strategy. The hybrid strategy initially follows Adam, then automatically switches to L-BFGS once the gradient norm stabilizes, yielding significantly faster convergence in the later stages.

```
        opt_adam.step()

        L, E = compute_loss_and_error(model, x, f_rhs, u_exact)
        grad_norms.append(grad_norm()) # Store gradient norms

        # Check for convergence plateau using sliding window
        if k > adam_min_iters + window:
            r = np.mean(grad_norms[-window:]) /
    np.mean(grad_norms[-2*window:-window])
            if r > plateau_tol:  # Stagnation detected
                print(f">>> Switching to L-BFGS at iter {k}")
                break

    # ----- L-BFGS stage: final refinement -----
    # Use L-BFGS optimizer of Pytorch to train parameters of model
    opt_lbfgs = optim.LBFGS( model.parameters(), max_iter=1,
     history_size=history_size, line_search_fn="strong_wolfe",
     tolerance_grad=tol_grad, tolerance_change=tol_change)

    for k in range(lbfgs_iters):
        # L-BFGS requires a closure, which return loss, in order to perform
     line-search to determine optimal step-size
        def closure():
            opt_lbfgs.zero_grad() # Zero gradient storage in L-BFGS
            # Eval derivative required to evaluate PDE
            uxx = pinn_second_derivative(model, x)
            loss = loss_fn(uxx, f_rhs) # Compute PINN loss
            loss.backward() # Perform back-propagation to get gradient
            return loss # Return loss

        opt_lbfgs.step(closure)  # Perform L-BFGs optimizer step
```

Code snippet 13: Example of using Pytorch's Adam and L-BFGS optimizers for training of PINN (1D Poisson). Automatic switch between Adam and L-BFGS is also implemented.

Key Takeaways

Second-order methods exploit curvature information contained in the Hessian, enabling faster convergence than first-order methods near a solution. In their classical form, however, they are too expensive for large-scale problems due to the $\mathcal{O}(n^2)$ cost of forming and storing the Hessian. Scalable alternatives, such as Hessian-free Newton–Krylov, subsampled Newton, and quasi-Newton (L-BFGS) methods, recover much of this benefit at a cost that remains roughly linear in the number of parameters. For PINNs, a practical and widely used strategy is to begin training with Adam for robust early-stage progress, then switch to L-BFGS to exploit curvature information and achieve faster final convergence.

7 Soft constraints and loss balancing

The objective function f is frequently composed of N terms as $f(\theta) = \sum_{i=1}^{N} \gamma_i \widehat{R}_i(\theta)$, where $\gamma_i \in \mathbb{R}^+$. This is due to the fact that typically several terms are incorporated into the loss function in order to improve generalization, enhance the well-posedness of the optimization problem, and enable more stable and efficient training. Such terms may include regularization or physical constraints, enforced softly through penalization (recall Section 2). The following subsections discuss the key ideas behind these formulations as well as the strategies for balancing heterogeneous loss terms. From the preconditioning viewpoint introduced in Section 3.1, the discussed loss-reweighting strategies act as data-space preconditioners that reshape the empirical loss and the effective Hessian/NTK spectrum.

7.1 Loss functions with multiple terms

7.1.1 Regularization

Regularization terms are often used to ensure that the learned model generalizes well beyond the training samples/collocation points. They augment the objective with an additional term $\widehat{R}_{\text{reg}}$, leading to the modified objective

$$f(\theta) := \widehat{R}_m(\theta) + \gamma \widehat{R}_{\text{reg}}(\theta),$$

where $\gamma > 0$ is a regularization parameter. A common strategy is to incorporate L^2 or L^1 penalties on the parameters or on the network's output gradients, e.g.,

$$\widehat{R}_{\text{reg}}(\theta) = \|\theta\|^2, \qquad \widehat{R}_{\text{reg}}(\theta) = \|\nabla_\theta h_\theta\|^2.$$

The first term encourages smoother parameter trajectories, while the second directly penalizes sharp variations in the network output.

In PINNs, one can additionally employ physics-guided regularization to control high-frequency components. A frequently used formulation is [19, 109]

$$\widehat{R}_{\mathrm{reg}}(\theta) = \int_{\Omega} \left|\nabla^p h_\theta(x)\right|^2 \mathrm{d}x,$$

where p is the differential order of the operator $\mathcal{N}$. Such derivative- or spectrum-based penalties enforce physically consistent smoothness and often improve both training stability and out-of-distribution generalization.

7.1.2 Soft constraints

A canonical example of introducing soft constraints arises in PINNs (cf. Section 2), where the PDE residual and the BC conditions are added to the loss via penalty terms, i.e.,

$$\widehat{R}_m(\theta) = \gamma_\Omega\, \widehat{R}_\Omega(\theta) + \gamma_D\, \widehat{R}_{\Gamma_D}(\theta) + \gamma_N\, \widehat{R}_{\Gamma_N}(\theta) + \gamma_{\mathrm{data}}\, \widehat{R}_{\mathrm{data}}(\theta),$$

where $\widehat{R}_\Omega, \widehat{R}_{\Gamma_D}, \widehat{R}_{\Gamma_N}$, and $\widehat{R}_{\mathrm{data}}$ denote, respectively, the PDE residual, the Dirichlet and Neumann boundary terms, and the data-misfit term as defined in (10).

The weights $\gamma_\Omega, \gamma_D, \gamma_N, \gamma_{\mathrm{data}} \in \mathbb{R}^+$ control how strongly each constraint is enforced. In the limit $\gamma \to \infty$, the penalty formulation formally recovers the corresponding hard-constraint problem. However, the choice of these weights strongly influences the optimization dynamics. If they are too small, the constraint may fail to be enforced effectively. If they are too large, the loss landscape becomes stiff, with one term dominating the overall gradient and hindering progress in the remaining components. The next subsection discusses how to mitigate this difficulty by systematically balancing the contributions of the heterogeneous loss terms.

7.2 Balancing the loss terms

Selecting appropriate weights $\gamma = (\gamma_1, \ldots, \gamma_N)$ is often challenging. Loss-balancing strategies aim to rescale the contributions of each component so that gradients and curvature are more homogeneous. From the NTK viewpoint of Section 3.2, each γ_i scales the eigenvalues of the i-th NTK block, directly controlling that component's convergence rate. However, imbalanced weights widen the overall spectrum and degrade the conditioning of the composite problem. Loss-balancing strategies can therefore be understood as hand-tuned data-space preconditioners (cf. Section 3.1).

7.2.1 Example: Impact of penalty terms on NTK's conditioning

Let us consider the simplified PINN setting with two loss components, the PDE residual $\widehat{R}_\Omega$ and a boundary term $\widehat{R}_\Gamma$, with corresponding weights γ_Ω and γ_Γ. In the NTK regime, the empirical NTK acquires a block structure

aligned with these two components. More precisely, neglecting the off-diagonal coupling blocks (which are typically small when the two loss components involve disjoint collocation points), the weighted NTK reads as

$$\Theta^{(\gamma)} = \begin{pmatrix} \gamma_\Omega\,\Theta_{\Omega\Omega} & 0 \\ 0 & \gamma_\Gamma\,\Theta_{\Gamma\Gamma} \end{pmatrix}, \tag{32}$$

where $\Theta_{\Omega\Omega}, \Theta_{\Gamma\Gamma}$ denote the per-component empirical NTK blocks.

The condition number of the NTK is then given as

$$\kappa\big(\Theta^{(\gamma)}\big) = \frac{\max\big(\gamma_\Omega\,\lambda_{\max}^{(\Omega)},\ \gamma_\Gamma\,\lambda_{\max}^{(\Gamma)}\big)}{\min\big(\gamma_\Omega\,\lambda_{\min}^{(\Omega)},\ \gamma_\Gamma\,\lambda_{\min}^{(\Gamma)}\big)}, \tag{33}$$

where $\lambda_{\max}^{(i)}, \lambda_{\min}^{(i)}$ are the largest and smallest eigenvalues of the NTK blocks. As we can see, the NTK spectrum is jointly controlled by the spectra of the individual blocks (problem-dependent) and by the relative scaling of the weights $\gamma_\Omega, \gamma_\Gamma$. In particular, even if each block Θ_{ii} is individually well-conditioned, a large mismatch between γ_Ω and γ_Γ can severely inflate $\kappa(\Theta^{(\gamma)})$ and slow convergence. Our overall goal is therefore to explore loss-balancing strategies, which select $\{\gamma_i\}_{i=1}^N$ to reduce $\kappa(\Theta^{(\gamma)})$.

7.2.2 Bilevel approaches

A principled strategy for choosing the weights γ is to treat them as *hyperparameters* that are selected by optimizing some outer performance measure $\mathcal{C}$, such as the validation loss or the mean PDE residual. This leads to the following bilevel optimization problem [40]:

$$\min_{\gamma>0}\ \mathcal{C}(\theta(\gamma)), \qquad \text{s.t.} \quad \theta(\gamma) = \arg\min_\theta \sum_{i=1}^N \gamma_i\,\widehat{R}_i(\theta),$$

where the inner problem determines the model parameters θ for a fixed weights γ, while the outer problem adjusts γ based on the performance measure $\mathcal{C}$.

At the inner optimum $\theta^\star(\gamma)$, the stationarity condition

$$\nabla_\theta \widehat{R}_m(\theta^\star, \gamma) = 0$$

implicitly defines the dependence of $\theta^\star(\gamma)$ on γ. Differentiating this condition with respect to γ yields the *hypergradient* $\nabla_\gamma \mathcal{C}$ whose components are given as

$$\frac{\partial \mathcal{C}}{\partial \gamma_j} = -\left(\nabla_\theta \mathcal{C}(\theta^\star)\right)^\top H^{-1}(\theta^\star, \gamma)\,\nabla_\theta^2 \widehat{R}_j(\theta^\star), \qquad H = \nabla_\theta \widehat{R}_m(\theta^\star, \gamma),$$

which shows that the response of $\theta^\star$ to changes in γ is governed by the inverse Hessian H^{-1} of the inner objective.

In the NTK regime, the inner Hessian H shares its nonzero eigenvalues with the empirical NTK Θ_k. The hypergradient therefore implicitly uses Θ_k^{-1}

to update γ, so that bilevel balancing is in fact an *NTK-aware* update of the weights. It accounts simultaneously for inter-block scale mismatch and intra-block stiffness in $\Theta^{(\gamma)}$. In particular, it drives γ toward values that reduce $\kappa(\Theta^{(\gamma)})$. However, computing or inverting H is prohibitive in practice, especially for large ML models. The heuristic strategies discussed below can be viewed as inexpensive, first-order approximations of this exact update, each capturing only part of its effect on the NTK spectrum.

7.2.3 Adaptive weighting strategies

Adaptive weighting strategies are heuristic techniques that can be used to adjust the weights γ during the optimization process so that the different loss components contribute more evenly.

Gradient-norm balancing The common approach is to employ *global heuristics*, that assign a single coefficient to each loss component. One of the most widely used approaches in this category is *Gradient-norm balancing (GN)* [17], which updates the weights γ in order to equalize the magnitudes of the per-loss gradients with respect to the shared parameters. At each iteration, *GN* computes for each component i the partial gradients $\|\gamma_i \nabla_\theta \widehat{R}_i(\theta)\|$, measuring the contribution of the i-th loss component to the overall gradient. A target value for these quantities is derived from the relative training speed of the losses:

$$\rho_i = \frac{\widehat{R}_i(\theta)/\widehat{R}_i(0)}{\frac{1}{N}\sum_{j=1}^{N} \widehat{R}_j(\theta)/\widehat{R}_j(0)}.$$

GN approach is designed such that it minimizes the surrogate objective

$$L_{\text{grad}}(\gamma) = \sum_{i=1}^{N} \Big| \, \|\gamma_i \nabla_\theta \widehat{R}_i(\theta)\| - \frac{1}{N} \sum_{j=1}^{N} \|\gamma_i \nabla_\theta \widehat{R}_i(\theta)\| \, \rho_i^{\zeta} \, \Big|,$$

with respect to the weights γ, where $\zeta > 0$ controls how strongly the method enforces equalized relative descent rates among the losses. In practice, this step is inexpensive as only a single gradient-descent update is applied to each γ_i using $\partial L_{\text{grad}}/\partial \gamma_i$, followed by a normalization step to maintain weights on a similar scale. This procedure equalizes the influence of the different loss terms without computing second-order information, thus providing a practical and efficient approximation to bilevel loss balancing. Other heuristics in this category include for example uncertainty-based weighting [52, 126] and stage-dependent curricula learning [118].

In the two block-NTK example discussed above (Section 7.2.1), GN rescales the *between-block* contributions. By enforcing that $\|\gamma_i \nabla_\theta \widehat{R}_i\|$ is equal across components, GN approximately drives the weights toward

$$\gamma_\Omega \, \lambda_{\max}^{(\Omega)} \;\approx\; \gamma_\Gamma \, \lambda_{\max}^{(\Gamma)} \;=:\; \mu_{\max},$$

so that the dominant eigenvalues of the two blocks become comparable. Substituting into (33), the condition number of the GN-rescaled NTK simplifies to

$$\kappa\big(\Theta^{(\gamma_{\mathrm{GN}})}\big) \;=\; \frac{\mu_{\max}}{\min\big(\gamma_\Omega\,\lambda_{\min}^{(\Omega)},\;\gamma_\Gamma\,\lambda_{\min}^{(\Gamma)}\big)} \;=\; \max\big(\kappa^{(\Omega)},\;\kappa^{(\Gamma)}\big),$$

where $\kappa^{(i)} := \lambda_{\max}^{(i)}/\lambda_{\min}^{(i)}$ denotes the within-block condition number. GN therefore removes the inter-block scale mismatch in $\Theta^{(\gamma)}$ and brings the overall condition number down to the worst within-block stiffness. However, since each Θ_{ii} itself is unchanged, GN does not address the intra-component stiffness.

Spatially adaptive weighting Using a single global weight for each loss component presumes that all samples contribute comparably. However, in PINNs, the PDE residual is often highly non-uniform, frequently differing by several orders of magnitude across the domain [76]. This mismatch can cause the optimizer to overfit well-resolved regions while failing to sufficiently reduce the error in stiff or under-resolved parts of the domain. *Spatially adaptive weighting* strategies were designed to mitigate this difficulty by assigning location-dependent weight ξ_j to all collocation points, thus replacing the standard residual term by

$$\widehat{R}_\Omega(\theta,\xi) = \frac{1}{m_\Omega}\sum_{j=1}^{m_\Omega}\xi_j\,|\mathcal{N}[h_\theta](x_j) - q(x_j)|^2, \qquad \sum_{j=1}^{m_\Omega}\xi_j = m_\Omega, \quad \xi_j \geq 0.$$

A widely used approach for constructing spatially adaptive weights ξ is residual-based weighting [5, 16], which constructs the weights directly from the pointwise residual $r_\theta(x_j) = \mathcal{N}[h_\theta](x_j) - q(x_j)$. A typical example includes power-law normalization, where the j-th weight takes on the following form:

$$\xi_j = \frac{|r_\theta(x_j)|^\beta}{\frac{1}{m_\Omega}\sum_{k=1}^{m_\Omega}|r_\theta(x_k)|^\beta}, \qquad \beta > 0. \tag{34}$$

This promotes more uniform residual reduction across the domain Ω by upweigting under-resolved regions, i.e., regions with high residual.

Recall the two-block-NTK example discussed in Section 7.2.1. The per-point weights ξ_j rescale rows and columns of the residual block. In particular, let $D_\xi = \mathrm{diag}(\xi_1,\ldots,\xi_{m_\Omega})$. The NTK of residual loss becomes

$$\widetilde{\Theta}_{\Omega\Omega} \;:=\; D_\xi^{1/2}\,\Theta_{\Omega\Omega}\,D_\xi^{1/2}.$$

Using residual-based weights (34), collocation points with larger PDE residuals receive larger weights. Since these points often correspond to directions that are learned slowly, they are associated with small eigenvalues of $\Theta_{\Omega\Omega}$. Thus, re-weighting them increases their impact and can make the spectrum of $\widetilde{\Theta}_{\Omega\Omega}$ more balanced. Thus, the within-block condition number is reduced, i.e.,

$$\kappa_\xi^{(\Omega)} \;:=\; \kappa\big(\widetilde{\Theta}_{\Omega\Omega}\big) \;\leq\; \kappa\big(\Theta_{\Omega\Omega}\big) \;=\; \kappa^{(\Omega)}.$$

As we can see, GN-type and spatially adaptive strategies are complementary. The former addresses inter-block imbalance, while the latter tackles intra-block stiffness. They can therefore be combined, yielding the following condition number:

$$\kappa\big(\Theta^{(\gamma)}\big) \;\approx\; \max\big(\kappa_{\xi}^{(\Omega)},\, \kappa^{(\Gamma)}\big) \;<\; \max\big(\kappa^{(\Omega)},\, \kappa^{(\Gamma)}\big).$$

Numerical Example #7: *Convergence of GD with adaptive weights.* We illustrate the analysis of Section 7.2.1 on a small numerical example with two loss components, the PDE residual $\widehat{R}_\Omega$ and a boundary term $\widehat{R}_\Gamma$, with corresponding weights γ_Ω and γ_Γ. Following the block-NTK structure of Equation (32), we assume that the per-component NTK blocks have spectra

$$\Lambda(\Theta_{\Omega\Omega}) = \mathrm{diag}(0.1,\, 0.04,\, 0.02,\, 0.01), \qquad \Lambda(\Theta_{\Gamma\Gamma}) = \mathrm{diag}(10,\, 5),$$

so that the within-block condition numbers are $\kappa^{(\Omega)} = 10$ and $\kappa^{(\Gamma)} = 2$. The boundary block $\Theta_{\Gamma\Gamma}$ thus has eigenvalues two to three orders of magnitude larger than the residual block $\Theta_{\Omega\Omega}$. This situation commonly occurs while training PINNs, since the boundary functional is often much better conditioned than the differential operator.

To show the dynamics induced by these spectra, we run GD on the auxiliary quadratic function

$$f(\theta) \;=\; \tfrac{1}{2}\,(\theta - \theta^\star)^\top\, \Theta^{(\gamma)}\,(\theta - \theta^\star), \tag{35}$$

unique minimizer of which is $\theta^\star$. The Hessian of (35) is exactly the weighted NTK $\Theta^{(\gamma)}$. Letting $e_k := \theta_k - \theta^\star$ denote the iteration error and using the optimal constant step size for quadratic minimization [86, Section 3.2] $\alpha = 2/(\lambda_{\max}\big(\Theta^{(\gamma)}\big) + \lambda_{\min}\big(\Theta^{(\gamma)}\big))$, the GD iteration $\theta_{k+1} = \theta_k - \alpha\,\nabla f(\theta_k)$ yields the following estimates for the error reduction and convergence rate:

$$\|e_k\|/\|e_0\| \;\le\; \rho^k, \qquad \rho \;=\; \frac{\kappa\big(\Theta^{(\gamma)}\big) - 1}{\kappa\big(\Theta^{(\gamma)}\big) + 1}. \tag{36}$$

Using the convergence metrics (36), we now compare the expected convergence behavior of GD under three weighting strategies: uniform weights (U), GN balancing, and a combined approach pairing GN with spatially adaptive weighting (GN+SA).

Uniform (U) weights ($\gamma_\Omega = \gamma_\Gamma = 1$): The composite spectrum of $\Theta^{(\gamma)}$ is the union of the two block spectra, i.e.,

$$\Lambda\big(\Theta^{(\gamma)}\big) = \{10,\, 5,\, 0.1,\, 0.04,\, 0.02,\, 0.01\}, \qquad \text{and} \qquad \kappa\big(\Theta^{(\gamma)}\big) = \frac{10}{0.01} = 1000.$$

The boundary block dominates while the residual block is essentially frozen. From (36), the contraction factor is $\rho \approx 0.998$, GD thus requires approximately $1,000$ iterations to reduce the error by a factor of ten.

GN balancing ($\gamma_\Omega = 1,\ \gamma_\Gamma = 0.01$): GN drives the weights toward $\gamma_\Omega \lambda_{\max}^{(\Omega)} \approx \gamma_\Gamma \lambda_{\max}^{(\Gamma)}$, i.e., $\gamma_\Gamma \approx 0.01$. The composite spectrum then becomes

$$\Lambda\big(\Theta^{(\gamma_{\mathrm{GN}})}\big) = \{0.1,\, 0.1,\, 0.05,\, 0.04,\, 0.02,\, 0.01\}, \quad \text{and} \quad \kappa\big(\Theta^{(\gamma_{\mathrm{GN}})}\big) = \frac{0.1}{0.01} = 10.$$

Hence, the condition number is reduced by two orders of magnitude and matches exactly $\max(\kappa^{(\Omega)}, \kappa^{(\Gamma)}) = 10$, confirming the prediction that GN collapses the overall conditioning to the worst within-block stiffness. The contraction factor improves to $\rho \approx 9/11 \approx 0.818$, so that only about 12 GD iterations suffice to reduce the error by an order of magnitude.

GN with spatially adaptive weighting (GN+SA): On top of the inter-block balance, considered in the previous paragraph, we now apply spatially adaptive weights within the residual block. After such a rescaling, the residual block spectrum is compressed, e.g., from $\{0.1, 0.04, 0.02, 0.01\}$ to a more uniform $\{0.1, 0.06, 0.05, 0.04\}$, so that the within-block condition number drops to $\kappa(\widetilde{\Theta}_{\Omega\Omega}) \approx 2.5$. The overall condition number then satisfies

$$\kappa\big(\Theta^{(\gamma)}\big) \;\approx\; \max\big(\kappa(\widetilde{\Theta}_{\Omega\Omega}),\, \kappa^{(\Gamma)}\big) \;\approx\; 2.5.$$

As a consequence, only around 3 iterations are needed to obtain the same tenfold error reduction.

Figure 7.1 visualizes the same experiment. The left panel shows the GD error $\|e_k\|/\|e_0\|$ on the auxiliary quadratic (35), for the three strategies. The empirical curves match theoretical rates closely, and the order-of-magnitude reduction in iteration count predicted by the analysis is clearly visible. The right panel displays the composite NTK spectrum for each strategy. As we can see, GN collapses the inter-block gap, while SA weighting additionally compresses the residual block toward a more uniform spectrum.

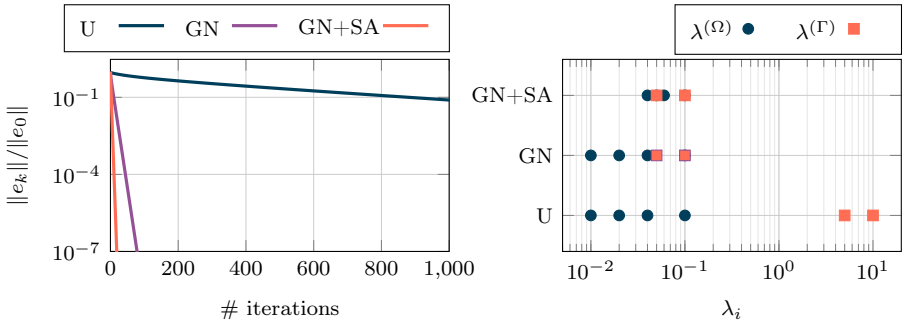

Figure 7.1: Effect of U, GN, and GN+SA strategies on the convergence of GD. *Left:* Relative GD error $\|e_k\|/\|e_0\|$, with $e_k := \theta_k - \theta^\star$, as a function of the iteration number. *Right:* Eigenvalues of $\Theta^{(\gamma)}$ at initialization, with residual-block eigenvalues marked by circles and boundary-block eigenvalues by squares.

Numerical Example #8: *Loss balancing in practice.* Building on the insights from the synthetic quadratic experiment of Numerical Example #7, we now examine how the different weighting strategies perform in practical PINN training. More precisely, we consider the one-dimensional diffusion equation, given as

$$u_t \;=\; u_{xx} + e^{-t}\big(\pi^2 - 1\big)\sin(\pi x), \quad (x,t) \in (-1,1) \times (0,1), \tag{37}$$

with homogeneous Dirichlet conditions $u(\pm 1, t) = 0$ and initial condition $u(x,0) = \sin(\pi x)$, whose manufactured exact solution is $u^\star(x,t) = \sin(\pi x)\, e^{-t}$. The PINN is an MLP with 4 hidden layers of width 32, and tanh activation functions. We utilize 256 residual collocation points and 80 initial and 80 boundary points. Training is performed using the full-batch Adam with learning rate 10^{-3} for 5,000 iterations. Boundary and initial conditions are enforced softly, so the loss has the three terms, i.e., $\widehat{R}_m = \gamma_\Omega \widehat{R}_\Omega + \gamma_{IC}\widehat{R}_{IC} + \gamma_{BC}\widehat{R}_{BC}$.

```python
def comp_losses(model, xt_pde, xt_ic, xt_bc, xi=None):
    # Computes per-component losses (R_Omega, R_IC, R_BC)
    # Inputs xt_pde, xt_ic, xt_bc are collocation points for residual, IC and BC
    # Argument `xi` is a per-point weight vector on the residual collocation set;
    # If None is provided, then the residual term is plain mean-square, i.e., xi_j = 1

    # Evaluate residual loss
    r = residual(model, xt_pde).squeeze()
    if xi is None:
        L_pde = (r ** 2).mean()
    else:
        L_pde = (xi.detach() * r ** 2).mean()

    # Evaluate IC loss
    u_ic_pred = model(xt_ic).squeeze()
    u_ic_true = torch.sin(PI * xt_ic[:, 0])
    L_ic = ((u_ic_pred - u_ic_true) ** 2).mean()

    # Evaluate BC loss
    L_bc = (model(xt_bc) ** 2).mean()
    return L_pde, L_ic, L_bc


def gradnorm_targets(model, xt_pde, xt_ic, xt_bc, xi=None):
    # Return the target weights based on GN rule
    params = [p for p in model.parameters() if p.requires_grad]
    Lp, Li, Lb = comp_losses(model, xt_pde, xt_ic, xt_bc, xi=xi)
    gp = grad_norm(Lp, params)
    gi = grad_norm(Li, params)
    gb = grad_norm(Lb, params)
    gmax = max(gp, gi, gb) + 1e-30
    return {'pde': gmax / (gp + 1e-30),
            'ic':  gmax / (gi + 1e-30),
            'bc':  gmax / (gb + 1e-30)}, (gp, gi, gb)

```

```python

def rba_target(model, xt_pde, beta=1.0):
    # Return the residual-based per-point weights based on size of residual
    r = residual(model, xt_pde).detach().abs().squeeze()
    rb = r.pow(beta)
    return r.shape[0] * rb / (rb.sum() + 1e-30)
```

Code snippet 14: Example of computing weights for loss balancing strategies.

```python
    opt = optim.Adam(model.parameters(), lr=lr)
    gamma = {'pde': 1.0, 'ic': 1.0, 'bc': 1.0}
    xi = torch.ones(xt_pde.shape[0])
    xi_snaps = [(0, xi.detach().cpu().numpy().copy())]

    for k in range(iters):
        # Weight updates every gn_every iterations
        # Update xi using SA/RBA first so the subsequent GradNorm target sees it
        if strategy == 'gn_sa' and (k + 1) xi_new = rba_target(model, xt_pde, beta=rba_beta)
            # Smoothing to deal with oscillatory residuals during the initial phase of training
            xi = (1 - gn_alpha) * xi + gn_alpha * xi_new
            xi_snaps.append((k + 1, xi.detach().cpu().numpy().copy()))
        if strategy in ('gn', 'gn_sa') and (k + 1) xi_used = xi if strategy == 'gn_sa' else None
            tgt, _ = gradnorm_targets(model, xt_pde, xt_ic, xt_bc, xi=xi_used)
            for kk in gamma:
                gamma[kk] = (1 - gn_alpha) * gamma[kk] + gn_alpha * tgt[kk]

        opt.zero_grad()
        xi_used = xi if strategy == 'gn_sa' else None
        Lp, Li, Lb = comp_losses(model, xt_pde, xt_ic, xt_bc, xi=xi_used)
        total = gamma['pde'] * Lp + gamma['ic'] * Li + gamma['bc'] * Lb
        total.backward()
        opt.step()
```

Code snippet 15: Snippet of training loop with adaptive loss scaling.

We now compare the performance of three weighting strategies discussed earlier in this section: U, GN, and GN+SA ($\xi_j \propto |r_\theta(x_j)|^\beta$, $\beta = 1$). An example of their implementation can be found in Code snippets 14 and 15. GN and GN+SA adaptive schemes update weights every 100 Adam iterations. For all three strategies, Figure 7.2 reports the per-component gradient norms $\|\nabla_\theta \widehat{R}_j\|$, where $j = \{\Omega, IC, BC\}$. As we can see, with uniform weights, $\|\nabla_\theta \widehat{R}_\Omega\|$ dominates the composite gradient, especially during the second half of the training. Utilizing GN weighting, the residual norms have a more comparable scale, which allows for more uniform residual reduction across all iterations. The GN+SA approach takes this a step further by addressing both inter-block and

intra-block imbalance simultaneously. In particular, by upweighting collocation points where the residual is currently large, it compresses the intra-block spectrum while sharpening the inter-block balance, allowing for a more substantial reduction of the IC/BC gradients.

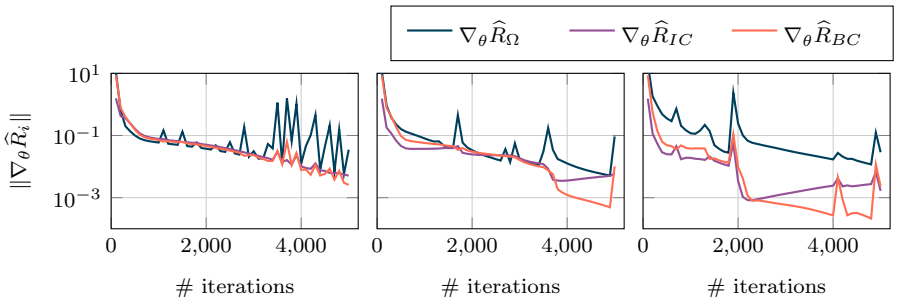

Figure 7.2: Per-component gradient norms during Adam training of the diffusion PINN model. The results for three different weighting strategies are reported: U, GN, and GN+SA (from left to right).

The improvement in the optimization directly translates into a reduction in the test error. To observe this, Figure 7.3 reports the relative L^2 test error $\|h_{\theta_k} - u^\star\|_{L^2} / \|u^\star\|_{L^2}$ as a function of the Adam iteration. As we can see, using weighting strategies not only allows for faster error reduction but also for achieving lower overall error. Compared to the quadratic analysis of Numerical Example #7, which predicted a reduction in the condition number by two orders of magnitude, the practical gains are more modest. This is due to the fact that the NTK evolves throughout training, the block-diagonal structure is only approximate, and Adam's adaptive step sizes partially compensate for gradient imbalance even under uniform weights. Nevertheless, the qualitative ranking of the three strategies is preserved, confirming that the NTK conditioning analysis provides a useful predictor of practical training behavior.

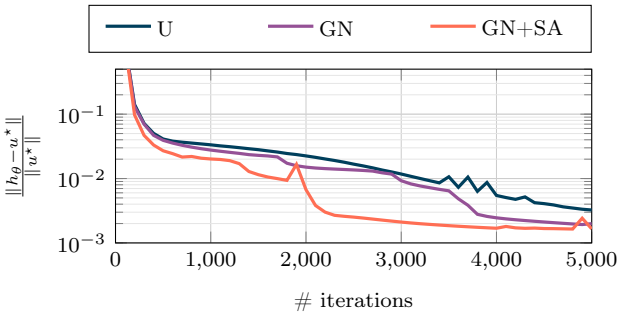

Figure 7.3: Relative L^2 test error of the diffusion PINN against the manufactured solution $u^\star$ during Adam training with different loss weighting strategies.

Key Takeaways

In SciML, the loss function is often composed of multiple heterogeneous terms that encode different regularizations, physical constraints, and data. This is especially prevalent in PINNs, where PDE residuals, initial conditions, boundary conditions, and optional data terms must all be satisfied simultaneously. The loss terms typically differ in magnitude by several orders of magnitude. Treating them with equal weights produces a severely ill-conditioned optimization problem in which the gradient of the dominant term monopolizes the rest, so part of the loss is effectively never minimized. Loss-balancing strategies reweight the terms during training to keep gradient contributions comparably scaled. The resulting well-balanced loss yields a much better-conditioned empirical NTK, leading to faster convergence.

8 Input embeddings and multiscale architectures

Recall from Section 3.3 that standard DNNs often exhibit strong *spectral bias*. As a result, the eigenvalues of the empirical NTK Θ_k can decay rapidly with frequency, so high-frequency components of the residual contract slowly, at a rate $(1-\alpha\lambda_i)^k$ with $\lambda_i \ll \lambda_{\max}$, and may effectively stagnate during training.

The strategies discussed in this section aim to mitigate spectral bias *before* the optimizer is applied by reshaping the eigenstructure of Θ_k. The goal is to flatten the NTK spectrum, or at least to shift the relevant frequency range of the target function into the well-conditioned part of Θ_k. We illustrate this idea with two concrete examples: Fourier feature (FF) embeddings [111] and multiscale network architectures [122]. From the preconditioning viewpoint adopted in Section 3.1, FF embeddings act in data space by transforming the inputs before they enter the network, whereas multiscale architectures act in function space by changing the function representation itself.

8.1 Fourier feature embeddings

FF embeddings were developed based on the observation that spectral bias originates in the input layer [111]. In particular, the first layer $\phi_1(x) = \sigma(W_1 x + b_1)$, with random initialization $W_1 \sim \mathcal{G}(0, \sigma_W^2)$, maps low-dimensional inputs to a narrow band of frequencies determined by σ_W. The subsequent layers, therefore, cannot create frequencies that are not already present at the input, and the resulting NTK has rapidly decaying eigenvalues at high frequencies, irrespective of network depth or width.

The FF embeddings address this limitation by replacing the raw input $x \in \mathbb{R}^d$ with the explicit sinusoidal encoding

$$\Phi(x) = \begin{pmatrix} \cos(2\pi B x) \\ \sin(2\pi B x) \end{pmatrix} \in \mathbb{R}^{2m_F}. \tag{38}$$

Here, $B \in \mathbb{R}^{m_F \times d}$ is a frequency matrix whose rows $b_1, \ldots, b_{m_F}$ are sampled from a fixed distribution, typically $b_i \sim \mathcal{N}(0, \sigma_B^2 I)$, and m_F is the number of Fourier features. The network then operates on the lifted representation, i.e.,

$$h_\theta(x) = \mathrm{DNN}_\theta\big(\Phi(x)\big). \tag{39}$$

Frequency components up to the scale σ_B are therefore injected into the network. Code snippet 16 provides an example of a PyTorch implementation of an MLP with Fourier feature embeddings.

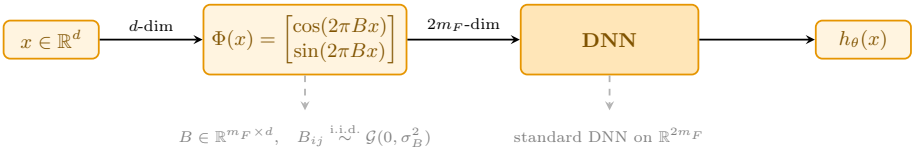

Figure 8.1: FF architecture. The input $x \in \mathbb{R}^d$ is first mapped to a $2m_F$-dimensional vector of cosine and sine features by a *fixed* random matrix B. A DNN then operates on this lifted representation. The embedding explicitly injects sinusoids up to frequency $\sim \sigma_B$.

```python

class FF_MLP(nn.Module):
    # MLP with Fourier-features
    # Matrix B ~ N(0, sigma_B^2 I) is fixed at initialization
    def __init__(self, sigma_B, m_F=M_F):
        super().__init__()
        # Sample values of B
        B = torch.randn(m_F, 1) * float(sigma_B)
        self.register_buffer('B', B)
        # MLP can be any network, but with increased input dimension
        self.core = MLP(in_dim=2 * m_F)

    def embed(self, x):
        Bx = 2.0 * float(np.pi) * x @ self.B.t()
        return torch.cat([torch.cos(Bx), torch.sin(Bx)], dim=1)

    def forward(self, x):
        # Embedding transforms input before passing it through MLP
        return self.core(self.embed(x))
```

Code snippet 16: Example of implementation of an MLP with Fourier features.

8.1.1 NTK of a FF network

FF embeddings transform the NTK into a kernel that can represent frequencies up to the scale σ_B without the strong low-frequency bias exhibited by standard NTKs. To understand this phenomenon, we first show that the NTK depends

on x only through $\Phi(x)$. In particular, for $h_\theta(x) = \mathrm{DNN}_\theta(\Phi(x))$, the embedding Φ does not depend on the network parameters θ, so the parameter gradient is just the DNN's parameter gradient evaluated with $\Phi(x)$, and the NTK is the DNN's NTK evaluated at the embedded points:

$$\Theta_{\mathrm{FF}}(x, x') \;=\; \Theta_{\mathrm{DNN}}\big(\Phi(x),\, \Phi(x')\big).$$

A standard result for wide MLPs with random isotropic weights is that $\Theta_{\mathrm{DNN}}(z, z')$ depends on the two inputs only through their dot product $z^\top z'$ and their norms [47]. For FFs, the norm is fixed, since

$$\|\Phi(x)\|^2 = \sum_{f=1}^{m_F} \big(\cos^2(2\pi b_f^\top x) + \sin^2(2\pi b_f^\top x)\big) = m_F,$$

so the only quantity that varies is $\Phi(x)^\top \Phi(x')$. In other words, the FF network's NTK is fully determined by the inner products $\Phi(x)^\top \Phi(x')$, which we now evaluate explicitly.

Using the relation $\cos a \cos b + \sin a \sin b \;=\; \cos(a - b)$, a direct computation gives $\Phi(x)^\top \Phi(x') \;=\; \sum_{f=1}^{m_F} \cos\big(2\pi b_f (x - x')\big)$. With large m_F and $b_f \sim \mathcal{N}(0, \sigma_B^2 I)$, the law of large numbers replaces the empirical average by its expectation, yielding a Gaussian kernel via the standard random-features identity [94]:

$$\frac{1}{m_F}\, \Phi(x)^\top \Phi(x') \;\approx\; \mathbb{E}_b\big[\cos(2\pi b(x - x'))\big] \;=\; \exp\big(-2\pi^2\, \sigma_B^2\, \|x - x'\|^2\big).$$

In the infinite-width limit, it follows that the NTK of DNN with FF is a Gaussian kernel

$$\Theta_{\mathrm{FF}}(x, x') \;\propto\; \exp\big(-2\pi^2\, \sigma_B^2\, \|x - x'\|^2\big),$$

whose bandwidth is controlled directly by σ_B [111]. The eigenvalues of any translation-invariant kernel are given by the Fourier transform of its profile [94], so for Θ_{FF} they decay as a Gaussian centered at zero frequency. More precisely, denoting by $\omega \in \mathbb{R}^d$ the spatial frequency variable, the eigenvalues satisfy

$$\lambda_\omega^{\mathrm{FF}} \;\propto\; \exp\big(-\tfrac{\|\omega\|^2}{2\sigma_B^2}\big). \tag{40}$$

Thus, inside the band $\|\omega\| \le \sigma_B$, the Gaussian profile is nearly flat. To see this, we substitute $\|\omega\| = \sigma_B$ in (40). The ratio of the eigenvalue at the band edge $\|\omega\| = \sigma_B$ to the eigenvalue at zero frequency $\omega = 0$ is therefore

$$\frac{\lambda_{\sigma_B}^{\mathrm{FF}}}{\lambda_0^{\mathrm{FF}}} = \frac{\exp(-\sigma_B^2/2\sigma_B^2)}{\exp(0)} = \exp(-1/2) \approx 0.6.$$

As a consequence, all frequencies within the band $\|\omega\| \le \sigma_B$ have eigenvalues of comparable magnitude and no frequency is favored over another. Outside the

band $\|\omega\| > \sigma_B$, eigenvalues decay rapidly as $\exp(-\|\omega\|^2/2\sigma_B^2)$, so modes far from the band are effectively invisible to the optimizer.

The appropriate choice of bandwidth σ_B is crucial in practice. If σ_B is too small, the DNN with FF encodings behaves almost like a standard DNN, and high-frequency components remain hard to learn. If σ_B is too large, the model may over-emphasize high frequencies at the expense of large-scale structure. Setting σ_B to match the highest frequency present in the target ensures that every relevant mode lies in the well-conditioned part of the spectrum. The effective condition number of the NTK on that band drops from $\mathcal{O}(e^{\omega_{\max}})$ to $\mathcal{O}(1)$. In this sense, the Fourier feature embedding acts as a *spectral preconditioner* of the NTK in data space. In practice, σ_B should be chosen on the order of the largest relevant frequency in the target, informed by prior knowledge of the target or a short hyper-parameter sweep.

8.2 Multiscale architectures

FF embeddings require knowing σ_B in advance. Moreover, a single bandwidth often struggles to simultaneously resolve features at very different scales. Multiscale architectures address both limitations by employing *parallel subnetworks* operating at different frequency scales [122]. Each subnetwork $q = 1, \dots, Q$ applies a FF map Φ_{σ_q} at its own scale and feeds the transformed input features through a given subnetwork. The subnetwork outputs are then linearly combined as

$$\begin{aligned} h_\theta(x) &= W_{\text{out}} \left[y_1(x);\, y_2(x);\, \cdots;\, y_Q(x) \right] + b_{\text{out}}, \\ y_q(x) &= \text{DNN}_q\big(\Phi_{\sigma_q}(x)\big), \quad q = 1, \dots, Q, \end{aligned} \tag{41}$$

see Figure 8.2 for an example of such architecture. The frequency scales satisfy $\sigma_1 < \sigma_2 < \cdots < \sigma_Q$ and are chosen to cover the frequency range of the target solution. Each subnetwork acts as a frequency specialist, i.e., q-th subnetwork captures features with frequencies of order σ_q, and the final linear layer combines outputs of all subnetworks. The advantage over a single Fourier-feature network is that each subnetwork operates in a well-conditioned regime. Its σ_q matches its target frequency band, thereby avoiding tension between low and high frequencies within a single network.

Code snippet 17 provides an example of a PyTorch implementation of the multiscale architecture (41) with Q parallel subnetworks.

```
class MultiscaleFF(nn.Module):
    # Q parallel FF subnetworks with bandwidths `sigmas`, combined by a
     linear layer
    # Each subnetwork: FF(sigma_q) + MLP -> R^{width}
    def __init__(self, sigmas, m_F=M_F, width=WIDTH, depth=DEPTH):
        super().__init__()
        self.Q = len(sigmas) # Number of subnetworks
        self.subnetworks = nn.ModuleList()
        for s in sigmas:
            # Generation of a fixed embedding matrix with particular sigma
```

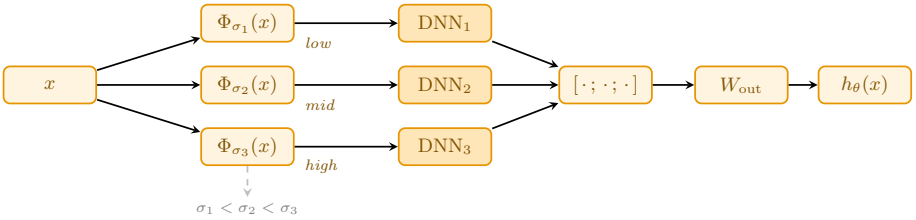

Figure 8.2: Multiscale architecture with $Q = 3$ parallel subnetworks, each applying a Fourier feature map Φ_{σ_q} at a different scale $\sigma_1 < \sigma_2 < \sigma_3$. The subnetwork outputs are concatenated and fused by a final linear layer.

```
            B = torch.randn(m_F, 1) * float(s)
            # Each subnetwork is MLP with a specific FF embeddings
            subnetwork = nn.Module()
            subnetwork.B = nn.Parameter(B, requires_grad=False)
            layers = [nn.Linear(2 * m_F, width), nn.Tanh()]
            for _ in range(depth - 1):
                layers += [nn.Linear(width, width), nn.Tanh()]
            subnetwork.mlp = nn.Sequential(*layers)
            self.subnetworks.append(subnetwork)

        # The last layer used for concatenation of subnetworks outputs
        self.head = nn.Linear(self.Q * width, 1)

    def forward(self, x):
        feats = []
        # Forward propagation is carried out through all subnetworks
        for br in self.subnetworks:
              # Each subnetwork is using specific FF encoding
            Bx = 2.0 * float(np.pi) * x @ br.B.t()
            phi = torch.cat([torch.cos(Bx), torch.sin(Bx)], dim=1)
            feats.append(br.mlp(phi))
        cat = torch.cat(feats, dim=1)
        # Head is combining outputs of all subnetworks
        return self.head(cat)
```

Code snippet 17: Example of multiscale architecture with Q parallel subnetworks.

8.2.1 NTK of multiscale networks

The preconditioning mechanism behind the multiscale networks architectures becomes transparent in the NTK regime. While FFs reshape Θ into a kernel concentrated at a chosen frequency band, the multiscale architecture factors it into independently conditioned bands. Both are concrete instances of *spectral preconditioning*, the first acting on the input (data-space) and the second on the function representation (function-space). Their effectiveness in PINN training has been demonstrated on a wide range of stiff and oscillatory benchmarks [122,

111].

Because the Q subnetworks share no parameters apart from the output weights W_{out}, the empirical NTK of (41) admits an approximate block decomposition, with cross-block coupling entering only through W_{out}, i.e.,

$$\Theta_{\text{MS-FF}}(x, x') \;\approx\; \sum_{q=1}^{Q} \|w_q\|_2^2 \, \Theta_{\text{FF},\,\sigma_q}(x, x'),$$

where $w_q \in \mathbb{R}^{\text{width}}$ denotes the q-th block of the output weight vector $W_{\text{out}} \in \mathbb{R}^{Q\cdot\text{width}}$, corresponding to the contribution of the q-th subnetwork to the scalar output. Moreover, $\Theta_{\text{FF},\,\sigma_q}$ denotes a single-scale FF kernel at bandwidth σ_q. Neglecting the cross-block coupling, this gives the following condition number for $\Theta_{\text{MS-FF}}$:

$$\kappa(\Theta_{\text{MS-FF}}) \;\approx\; \max_{1\le q\le Q} \kappa\big(\Theta_{\text{FF},\,\sigma_q}\big).$$

Hence, $\kappa(\Theta_{\text{MS-FF}})$ is bounded by the worst-conditioned single subnetwork. Since each σ_q is matched to its target frequency band, each within-band NTK $\Theta_{\text{FF},\sigma_q}$ is well-conditioned, i.e., $\kappa(\Theta_{\text{FF},\sigma_q}) \approx \mathcal{O}(1)$, and consequently $\kappa(\Theta_{\text{MS-FF}}) \approx \mathcal{O}(1)$ as well. Consequently, the overall conditioning is governed by these favorable within-band condition numbers, rather than by the large condition number induced by spanning all frequencies at once. Thus, stratifying the network across scales effectively replaces a single ill-conditioned NTK with a collection of well-conditioned blocks, one per frequency band.

Numerical Example #9: *Fourier features and multiscale architectures.*

To make the spectrum-reshaping mechanism of Sections 8.1–8.2 concrete, we revisit the spectral-bias setup of Numerical Example #1 with a multi-frequency target, given as

$$u^\star(x) \;=\; 0.7\,\sin(2\pi x) \;+\; 0.5\,\sin(4\pi x) \;+\; 0.3\,\sin(8\pi x).$$

We train a small MLP (three hidden layers of width 64, tanh activations) to fit the target function under different input encodings. We consider the following architectures: i) a *plain MLP* operating on raw inputs $x \in \mathbb{R}$; ii) an MLP with FF embeddings (38) with $m_F = 32$ and bandwidths $\sigma_B \in \{2, 4, 8\}$; iii) a multiscale FF network (MS-FF) with three parallel subnetworks with the bandwidths $\sigma_B \in \{2, 4, 8\}$. All networks are trained on $N = 1{,}024$ points sampled uniformly in $[0, 1]$ using Adam for 20,000 iterations.

As we can see from Figure 8.3 on the left, the plain-MLP NTK spectrum decays sharply. In MLPs with FF embeddings, FFs inject a fixed band of frequencies and roughly flatten the spectrum within that band. The bandwidth σ_B directly controls which frequencies are emphasized. MS-FF concatenates the three bands and, in turn, gives rise to the smallest condition number $\kappa(\Theta)$, see Table 1.

Figure 8.3 on the right, and Table 1 also demonstrate that MS-FF matches the best single-FF in L^2 error. Its three subnetworks partition the active band,

so the result is dominated by within-band conditioning rather than the global frequency range. Here, we emphasize that the benefit of using MS-FF is not that it always beats the best matched single-FF (it does not, and on this target FF $\sigma_B = 4$ is essentially tied with it). Rather, it is a safe choice when the target's active band (the range of spatial frequencies that significantly contribute to the solution $u^\star(x)$) is unknown a priori. Note that every other single-FF in the table is worse than MS-FF on the considered target.

Architecture	L^2 error	$\kappa(\Theta_k)$ at $k = 0$
MLP	9.58×10^{-4}	1.4×10^{11}
FF, $\sigma_B = 2$	9.99×10^{-5}	2.5×10^{10}
FF, $\sigma_B = 4$	$\mathbf{5.54 \times 10^{-5}}$	1.4×10^{10}
FF, $\sigma_B = 8$	1.62×10^{-4}	3.0×10^{8}
MS-FF $\sigma_B = \{2, 4, 8\}$	$\mathbf{5.59 \times 10^{-5}}$	$\mathbf{3.0 \times 10^{7}}$

Table 1: L^2 error and condition number of the empirical NTK at initialization.

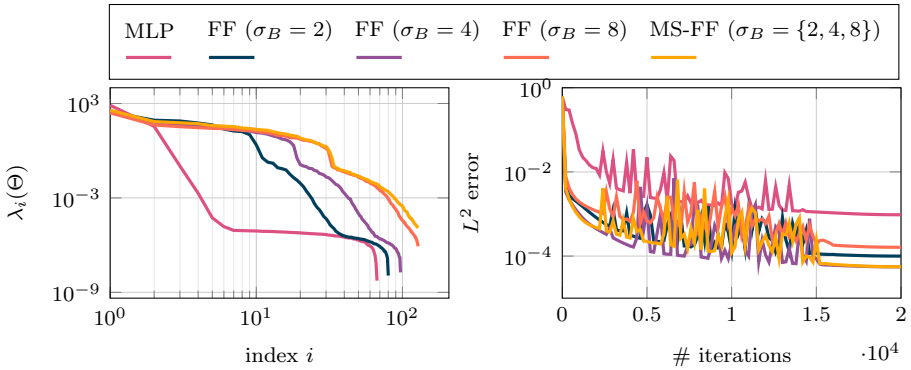

Figure 8.3: *Left:* Empirical NTK eigenvalue spectrum at initialization. *Right:* L^2 error of h_{θ_k} against the target $u^\star$ during training. Plain MLP plateaus at $\sim 10^{-3}$, while the FF and MS-FF networks enable convergence to a more accurate solution.

Key Takeaways

Solving PDEs with sharp features, oscillatory solutions, and multiscale physics is particularly challenging due to *spectral bias*: DNNs tend to learn low-frequency content first, while high-frequency components are learned significantly more slowly. This occurs because the eigenvalues of

the empirical NTK decay rapidly with the frequency of the target function. Both FF embeddings and multiscale network architectures mitigate this difficulty by acting as *spectral preconditioners*. FF embeddings act as a data-space preconditioner, reshaping the NTK's spectrum by transforming the input data. Multiscale architectures act as a function-space preconditioner, reshaping the spectrum further by partitioning the frequency range across independent subnetworks. In both cases, the objective is to align the relevant frequencies with the well-conditioned part of the NTK spectrum, thereby reducing the effective condition number and accelerating convergence.

9 Sampling and mini-batching strategies for PINNs

The approximation quality of PINNs depends crucially on *how collocation points are sampled* (discretization error) and *how mini-batches are constructed* during stochastic optimization process (mini-batch sampling error). Although largely orthogonal, both reshape the spectrum of the empirical NTK and therefore influence the convergence of the underlying optimizer. From the preconditioning viewpoint of Section 3.1, the collocation point sampling strategies discussed in this section act as data-space preconditioners that reshape the effective NTK spectrum.

9.1 Choice of collocation points

As discussed in Section 2.1.3, PINNs enforce the PDE residual and boundary conditions only at a finite set of collocation points, so the resulting optimization problem is intrinsically determined by the sampling strategy. Recall from Section 3.2 that each collocation point contributes a rank-one term to the empirical NTK, given as $\Theta_k = \frac{1}{m}\sum_{j=1}^{m} \nabla_\theta h_{\theta_k}(x_j)\, \nabla_\theta h_{\theta_k}(x_j)^\top$, so the distribution of collocation points directly shapes its spectrum.

Undersampling relevant regions (e.g., boundary layers or sharp interfaces) leaves the associated Jacobian directions poorly represented, leading to small or vanishing eigenvalues of the NTK. Conversely, oversampling smoother regions concentrates the spectrum in a few dominant directions. Thus, both effects inflate $\kappa(\Theta_k)$, resulting in ill-conditioning. The sampling distribution, therefore, directly controls the spectral properties of the NTK and, by extension, the difficulty of solving the optimization problem. Even with a fixed architecture, convergence can vary dramatically depending solely on the choice of collocation points. The design of the collocation set is thus not only a modeling choice, but also a key optimization parameter, particularly in problems with strong spatial heterogeneity.

9.1.1 Adaptive choice of collocation points

A practical way to mitigate NTK/Hessian ill-conditioning caused by poor sampling is to adaptively relocate or enrich the collocation set in regions identified as problematic. This approach is closely related to adaptive mesh refinement in classical numerical methods for PDEs [115].

In practice, adaptive resampling of collocation points is performed every T iterations, after which training resumes from the current parameters θ, creating a feedback loop between optimization and sampling. At each refinement step $t+1$, a new collocation set $\mathcal{D}_\Omega^{(t+1)} = \{x_j^{(\Omega,t+1)}\}_{j=1}^{m_\Omega}$ is constructed by sampling from the distribution with density

$$p^{(t+1)}(x) = \frac{\big(\eta^{(t+1)}(x)\big)^\beta}{\int_\Omega \big(\eta^{(t+1)}(\tilde{x})\big)^\beta \,\mathrm{d}\tilde{x}}, \tag{42}$$

where $\eta^{(t+1)}(x)$ is a user-specified indicator function, for example, the pointwise PDE residual $|r_\theta(x)|$, with $r_\theta(x)$ defined in (8). The parameter $\beta > 0$ controls the sharpness of the sampling distribution. Larger values of β concentrate new points in regions where $\eta^{(t+1)}$ is large, while smaller values promote broader exploration of the domain.

This adaptive sampling strategy can be interpreted as importance sampling for the residual empirical risk. By weighting each collocation point with $\frac{1}{p^{(t+1)}}$, we obtain an unbiased estimator [83] of the continuous residual integral $\int_\Omega \big|\mathcal{N}[h_\theta](x) - q(x)\big|^2 \,\mathrm{d}\mathcal{P}_\Omega(x)$, given as

$$\widehat{R}_\Omega(\theta) = \frac{1}{m_\Omega} \sum_{j=1}^{m_\Omega} \frac{1}{p^{(t+1)}(x_j^{(\Omega)})} \big|\mathcal{N}[h_\theta](x_j^{(\Omega)}) - q(x_j^{(\Omega)})\big|^2. \tag{43}$$

Conditioning of the NTK From a spectral viewpoint, adaptive sampling improves the conditioning of the empirical NTK/Hessian. Recall from Section 3.2 that the NTK is defined as a sum of positive semidefinite rank-one terms, i.e., $\Theta^{(m)} = \frac{1}{m}\sum_{j=1}^m J_j J_j^\top$, where m denotes the number of collocation points and $J_j := J(x_j)$.

Adding a new collocation point yields

$$\Theta^{(m+1)} = \frac{m}{m+1}\Theta^{(m)} + \frac{1}{m+1} J_{m+1} J_{m+1}^\top, \qquad J_{m+1} J_{m+1}^\top \succeq 0.$$

Now, let $A := \frac{m}{m+1}\Theta^{(m)}$, and $B := \frac{1}{m+1} J_{m+1} J_{m+1}^\top$, so that $\Theta^{(m+1)} = A + B$ with $B \succeq 0$. By Weyl's inequality for Hermitian matrices [44, Theorem 4.3.1], it follows that

$$\lambda_i(A+B) \geq \lambda_i(A), \qquad i = 1, \ldots, n.$$

Therefore,

$$\lambda_i(\Theta^{(m+1)}) \geq \lambda_i\left(\frac{m}{m+1}\Theta^{(m)}\right) = \frac{m}{m+1}\,\lambda_i(\Theta^{(m)}) \geq 0, \qquad i = 1, \ldots, n.$$

Hence, in practice, when the new Jacobian J_{m+1} introduces a direction not yet represented in $\{J(x_j)\}_{j=1}^m$, the smallest eigenvalue $\lambda_{\min}(\Theta^{(m)})$ increases sharply, while $\lambda_{\max}(\Theta^{(m)})$ is nearly unaffected. To understand why, we therefore assume that the new direction is approximately aligned with the eigenvector $q_{\min}$ associated with $\lambda_{\min}(\Theta^{(m)})$, i.e., with the direction that is most deficient in the current Jacobian span. By applying the standard first-order perturbation formula for symmetric matrices [44, Section 6.3], the shift in each eigenvalue λ_i satisfies

$$\lambda_i(\Theta^{(m+1)}) \approx \frac{m}{m+1}\lambda_i(\Theta^{(m)}) + \frac{1}{m+1}\left(q_i^\top J_{m+1}\right)^2. \tag{44}$$

For $\lambda_{\min}$, the product $q_{\min}^\top J_{m+1}$ is large by assumption, so the correction term dominates and $\lambda_{\min}$ increases sharply. For $\lambda_{\max}$, however, the leading eigenvector $q_{\max}$ corresponds to directions already well-represented in the existing Jacobian. Since J_{m+1} was selected for its novelty, it is nearly orthogonal to $q_{\max}$, i.e., $q_{\max}^\top J_{m+1} \approx 0$. The correction term $q_{\max}^\top J_{m+1}$ is therefore negligible, and $\lambda_{\max}$ is increased only by the mild rescaling factor $\frac{m}{m+1}$, leaving $\lambda_{\max}$ nearly unchanged for large m.

Since the smallest eigenvalue $\lambda_{\min}(\Theta^{(m)})$ increases sharply, while $\lambda_{\max}(\Theta^{(m)})$ is nearly unaffected, the condition number $\kappa(\Theta^{(m)})$ is improved. Geometrically, the quadratic form $\theta^\top\Theta^{(m)}\theta$ determines the level sets of the loss landscape. Improving $\kappa(\Theta^{(m)})$ corresponds to making these level sets more isotropic, which in turn permits larger stable step sizes and faster convergence of the GD method.

9.1.2 Example: Adaptive Sampling for 1D Poisson problem

To illustrate how adaptive sampling can improve conditioning, we consider the one-dimensional Poisson problem on $\Omega = (0,1)$, given as $-u''(x) = g(x),\ u(0) = u(1) = 0$, with a localized forcing term

$$g(x) = 10\, e^{-100(x-0.9)^2},$$

which induces sharp gradients in the solution near $x \approx 0.9$.

We approximate the solution by a simple two-parameter surrogate,

$$h(x;\theta) = \theta_1\,\phi_1(x) + \theta_2\,\phi_2(x), \qquad \phi_1(x) = x(1-x), \qquad \phi_2(x) = x^2(1-x)^2.$$

The residual and its Jacobian with respect to the parameters θ are given by

$$r(x;\theta) = -h''(x;\theta) - g(x), \qquad J(x) = \nabla_\theta r(x;\theta) = \begin{bmatrix} -\phi_1''(x) \\ -\phi_2''(x) \end{bmatrix}.$$

The NTK is defined as $\Theta^{(m)} = \frac{1}{m}\sum_{j=1}^m J(x_j)J(x_j)^\top$, where m denotes the number of collocation points.

Uniform sampling Let the collocation points be uniformly spaced as

$$x_j \in \{0.1,\ 0.3,\ 0.5,\ 0.7,\ 0.9\}.$$

Most of these points lie in regions where $g(x)$ and $r(x)$ are small, resulting in Jacobians that are nearly collinear. Consequently, the resulting NTK, $\Theta^{(5)} = \frac{1}{5}\sum_{j=1}^{5} J(x_j)J(x_j)^\top$, has eigenvalues $\lambda_{\max} \approx 4.1$ and $\lambda_{\min} \approx 0.02$, yielding a condition number $\kappa(\Theta^{(5)}) \approx 200$.

Adaptive refinement Now, let us suppose that a residual-based sampling identifies the high-error region near $x \approx 0.9$ and enriches the collocation set to

$$x_j \in \{0.1,\, 0.3,\, 0.5,\, 0.7,\, 0.85,\, 0.9,\, 0.95\},$$

as illustrated in Figure 9.1. The additional points contribute Jacobian directions aligned with the localized feature in $g(x)$. The refined NTK, $\Theta^{(7)} = \frac{1}{7}\sum_{j=1}^{7} J(x_j)J(x_j)^\top$, exhibits $\lambda_{\max} \approx 4.0$ and $\lambda_{\min} \approx 0.30$, in turn reducing the condition number to $\kappa(\Theta^{(7)}) \approx 13$. Note that the smallest eigenvalue increases significantly, while the largest remains nearly unchanged.

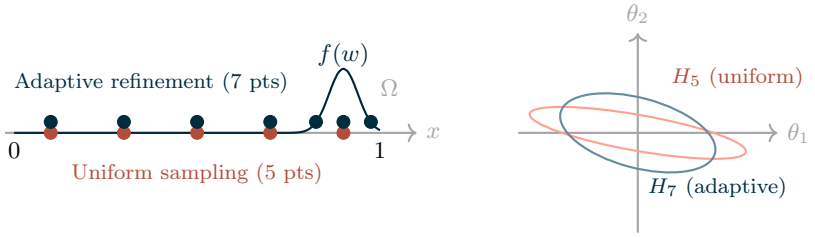

Figure 9.1: *Left:* Domain $\Omega = (0, 1)$ with localized forcing near $x \approx 0.9$. Uniform sampling is depicted in orange, while adaptive refinement is depicted in blue. *Right:* Level sets of $\theta^\top H_m \theta = 1$ with $H_m := H(x_m)$ become more isotropic after refinement, reflecting improved conditioning.

9.1.3 Examples of practical refinement strategies

We now briefly review two refinement strategies commonly used in practice [125]. Both strategies utilize the indicator function $\eta^{(t)}(x) = |r_\theta(x)|$, and are specified as follows:

- *RAD (Residual-based Adaptive Distribution):* Every T iterations, the entire collocation set $\mathcal{D}_\Omega^{(t)}$ is *redrawn* from

$$p^{(t+1)}(x) \;\propto\; \frac{|r_\theta(x)|^\beta}{\mathbb{E}_{x\sim\mathcal{U}(\Omega)}[|r_\theta(x)|^\beta]} \;+\; c,$$

  where $\beta > 0$ controls how aggressively the density concentrates on high-residual regions and $c \geq 0$ adds a uniform floor that preserves exploration. As the collocation points are completely redrawn, the budget of m_Ω collocation points is held fixed across all refinement steps. Note that this

sampling strategy is stochastic, and when combined with the importance weights $1/p^{(t+1)}(x_j^{(\Omega)})$ in $\widehat{R}_\Omega$, it yields an unbiased estimator (43).

- *RAR-G (Residual-based Adaptive Refinement - Greedy):* This approach corresponds to taking the limit $\beta \to \infty$ in (42). More precisely, every T iterations, the residual $|r_\theta(x)|$ is evaluated on a dense pool of candidates drawn from Ω. The q points with the largest $|r_\theta(x)|$ are then appended to $\mathcal{D}_\Omega^{(t)}$, yielding $\mathcal{D}_\Omega^{(t+1)} = \mathcal{D}_\Omega^{(t)} \cup \{x_{m_\Omega^{(t)}+1}, \ldots, x_{m_\Omega^{(t)}+q}\}$ and $m_\Omega^{(t+1)} = m_\Omega^{(t)} + q$. In this case, the collocation set grows monotonically, hence the name greedy. By the Weyl inequality, each appended $J(x_{m+i})J(x_{m+i})^\top$ to $\Theta^{(m)}$ can only increase the eigenvalues of $\Theta^{(m)}$, associated with the largest gain along the worst-residual direction. Since the points are selected greedily rather than by importance sampling, the resulting estimator of $\widehat{R}_\Omega$ is therefore biased toward high-residual regions.

Code snippet 18 provides an example of implementing the RAD and RAR-G sampling strategies in Python. For more advanced indicators and adaptive schemes, we refer to [125, 116].

```python
def rar_g_select(model, k, pool=10000):
    # Greedy top-k: pick the candidates with the largest |residual|
    cand = sample_uniform(pool) # Sample candidate pool of samples
    r = residual(model, cand).detach().abs().squeeze() # Evaluate residual
    idx = torch.topk(r, k=k).indices
    return cand[idx] # Return points with top-k largest residual

def rad_select(model, m, pool=10000, k_exp=1.0, c=1.0):
    # Importance sampling: draw m points without replacement from
    # p(x) proportional to |r(x)|^k_exp / mean(|r|^k_exp) + c
    cand = sample_uniform(pool) # Sample candidate pool of samples
    r = residual(model, cand).detach().abs().squeeze() # Evaluate residual
    rk = r.pow(k_exp)
    p = rk / (rk.mean() + 1e-30) + c
    p = p / p.sum() # Construct distribution density
    # Sample m points using distribution $p$
    idx = torch.multinomial(p, m, replacement=False)
    return cand[idx] # Return points chosen by importance sampling

# Inside the training loop, we use adaptive sampling strategy every
    resample_every steps
if (k + 1) if   strategy == 'rar':
    # Expand the existent set of collocation points
    xt = torch.cat([xt, rar_g_select(model, n_add=1)], 0).detach()
  elif strategy == 'rad':
    # Replace the existent set of collocation points
    xt = rad_select(model, m=N_RAD).detach()
```

Code snippet 18: Example of residual-based adaptive sampling strategies (RAR-G and RAD). Every `resample_freq` iterations, RAR-G appends one new point to the dataset $\mathcal{D}_\Omega$ while RAD redraws a whole new dataset.

Numerical Example #10: *Adaptive sampling on a 1D diffusion PINN.* The toy example discussed in Section 9.1.2 is intentionally low-dimensional: with only two parameters and a hand-picked feature near $x \approx 0.9$, the conditioning argument is essentially algebraic. We now repeat the same comparison using PINN, trained to approximate the solution of the same one-dimensional time-dependent diffusion problem as considered in Numerical Example #8. Thus, we consider the following problem:

$$u_t \;=\; u_{xx} + e^{-t}\big(\pi^2 - 1\big)\sin(\pi x), \quad (x,t) \in (-1,1) \times (0,1),$$

with $u(\pm 1, t) = 0$, $u(x,0) = \sin(\pi x)$ and manufactured exact solution $u^\star(x,t) = \sin(\pi x)\, e^{-t}$. The setup follows [125, Section 3.2]: a tanh-MLP with 4 hidden layers of width 32, trained by full-batch Adam (learning rate 10^{-3}), with 80 initial-condition points and 80 boundary-condition points held fixed during training.

We compare three strategies for constructing the residual collocation set. Uniform sampling uses $m_\Omega = 30$ collocation points drawn once at $t = 0$. We also consider the RAR-G and RAD strategies described above with an update interval of 1,000 iterations. All three runs use the same network initialization and the same initial 30 interior collocation points.

Figure 9.2 reports the relative L^2 test error with respect to the manufactured solution $u^\star$. Three approaches behave identically until iteration 1,000, since no refinement has yet been triggered. After approximately 2,000 iterations, Adam with uniform sampling plateaus, since with only 30 fixed collocation points, the residual loss has already been minimized along the directions spanned by the corresponding Jacobians, and Adam can no longer extract additional signal.

In contrast, RAR-G continues to make progress because each refinement step adds a collocation point whose Jacobian is aligned with the largest-residual direction, expanding the lower end of the empirical NTK spectrum via a rank-one update. For RAD, the entire set of collocation points is refreshed every 1,000 iterations, so the empirical NTK is continually enriched with new directions throughout training.

9.2 Mini-batch sampling

Having discussed how the global set of collocation points should be chosen, we now turn to a finer question: how should subsets of these points be selected at each iteration of the stochastic optimization process? As discussed in Section 4, SGD relies on stochastic gradient estimates computed using mini-batches. For classical ML tasks with i.i.d. data, these mini-batch gradients provide an unbiased estimator of the full gradient, with variance controlled by the batch size. In PINNs, however, selecting informative and stable mini-batches is substantially more delicate, for several reasons:

1. *Non-i.i.d. sampling:* When collocation points are generated using structured sampling strategies such as quasi-Monte Carlo or adaptive schemes, the samples are no longer independent. As a consequence, classical

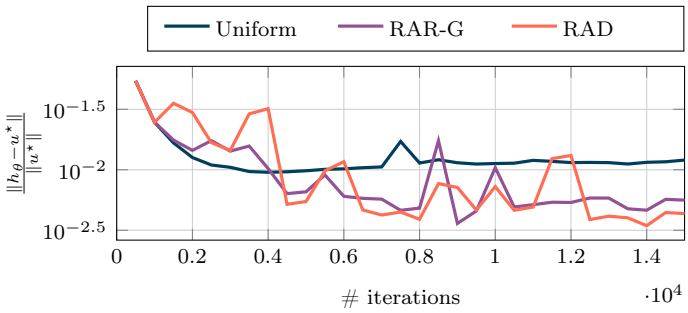

Figure 9.2: Relative L^2 test error of the PINN solution with respect to the exact solution $u^\star$ during Adam training. Uniform sampling is compared with the adaptive RAR-G and RAD sampling approaches.

variance-based analyses of SGD do not apply directly. In the non-i.i.d. setting, the accuracy of mini-batch gradient estimates is no longer controlled by the variance of the estimator (as in the classical i.i.d. case), but rather by the discrepancy (a measure of uniformity of the point set over the domain) and the dependence properties of the sampling scheme [84].

2. *Spatially correlated and heterogeneous sampling:* The PDE residual is spatially correlated and typically highly nonuniform across the domain. Mini-batches that contain points in steep-gradient regions generate disproportionately large updates, while batches drawn from smoother regions produce much smaller ones. This heterogeneity leads to inconsistent optimization steps and can induce oscillations or even divergence.

3. *Highly unbalanced variance:* For a squared-residual objective, the mini-batch gradient estimator can be written as

$$g_{\mathcal{I}_k}(\theta) = \frac{1}{|\mathcal{I}_k|} \sum_{x_j \in \mathcal{I}_k} \nabla_\theta r_\theta(x_j)\, r_\theta(x_j), \qquad \phi_\theta(x) := \nabla_\theta r_\theta(x)\, r_\theta(x). \tag{45}$$

Under i.i.d. Monte Carlo sampling, the variance scales as

$$\mathrm{Var}[g_{\mathcal{I}_k}(\theta)] = \frac{1}{|\mathcal{I}_k|} \mathrm{Var}[\phi_\theta(x)]\,, \qquad \mathbb{E}[\|\phi_\theta(x)\|^2] = \mathbb{E}\big[r_\theta(x)^2\, \|\nabla_\theta r_\theta(x)\|^2\big]\,.$$

In PDE-constrained losses, $\nabla_\theta r_\theta$ involves derivatives of order p of the network's output h_θ and may therefore grow rapidly with the frequency content of the solution, recall Section 3.5. As a result, higher-order PDEs or solutions with sharp layers can produce large batch-to-batch fluctuations, destabilizing early training and slowing convergence.

4. *Interaction with NTK conditioning:* Even if the stochastic gradient variance is moderate, the mini-batch update is still filtered through the NTK.

From (19), we know that each update is projected onto the eigenbasis of the NTK. Directions corresponding to small NTK eigenvalues are thus effectively frozen:

$$(I - \alpha\Theta_k)\, q_i = (1 - \alpha\lambda_i)\, q_i \approx q_i \qquad \text{for } \lambda_i \ll 1,$$

so that SGD makes little progress in these components. However, stochastic gradient noise still projects onto these poorly conditioned subspaces, where curvature is nearly zero, leading to noise accumulation and spurious oscillations. Since the gradient noise $\epsilon_k = g_{\mathcal{I}_k} - \nabla\widehat{R}_m$ is isotropic on average, it projects equally onto all eigendirections of the NTK. In well-conditioned directions, the signal dominates the noise. In poorly conditioned directions, where the signal is nearly zero, the noise dominates and causes spurious oscillations. As a consequence, optimization may appear to stagnate despite rapid progress in better-conditioned directions.

To avoid these difficulties, practitioners often use the *entire* set of collocation points during training, so PINNs are typically trained in a *deterministic* or *very large-batch* regime. Nevertheless, several works report that mini-batch training can improve generalization in PINNs [120].

9.2.1 Practical mini-batching strategies

We now discuss several practical mini-batching strategies for PINNs. To this aim, let us recall Theorem 4. Adapted to the mini-batch setting, i.e., taking into account that $\mathrm{Var}[g_{\mathcal{I}_k}(\theta)] = \sigma^2(\theta)/|\mathcal{I}_k|$, it states that for a smooth nonconvex objective $\widehat{R}$ and SGD with *bounded-variance* (Assumption 3) and *unbiased* gradient estimates, we have

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}[\|\nabla\widehat{R}(\theta_k)\|^2] \;\le\; \frac{2\left(\widehat{R}(\theta_0) - \widehat{R}^\star\right)}{\sqrt{K}} \;+\; \frac{L_f\,\sigma^2}{\sqrt{K}\,|\mathcal{I}_k|}, \tag{46}$$

with L_f the Lipschitz constant of the gradient of $\widehat{R}$ and $\widehat{R}^\star = \inf\widehat{R}$. The first term in (46) is the initialization gap, while the second is the mini-batch variance contribution that scales as $1/|\mathcal{I}_k|$.

Utilizing standard i.i.d. mini-batching while training PINNs can break both assumptions of this theorem. First, even with uniformly randomly sampled collocation points, the stochastic gradient is generally biased (i.e., $\mathbb{E}[g_{\mathcal{I}_k}(\theta)] \neq \nabla\widehat{R}(\theta_k)$) if the proportions of sampled points from $\{\Omega, \Gamma_D, \Gamma_N\}$ do not match the corresponding loss weights in $\widehat{R} = \gamma_\Omega\widehat{R}_\Omega + \gamma_{\Gamma_D}\widehat{R}_{\Gamma_D} + \gamma_{\Gamma_N}\widehat{R}_{\Gamma_N}$. Second, the PDE residual r_θ involves up-to-p-th order derivatives of h_θ, so the term $\nabla_\theta r_\theta(x)\, r_\theta(x)$ grows rapidly with the frequency content of the solution (see also (45)) and inflates σ^2 in ways the bounded-variance assumption cannot capture.

We now discuss some practical mini-batch construction strategies, which can be viewed as tools for i) ensuring that the unbiasness assumption of Theorem 4

is satisfied; ii) tightening the initialization gap and iii) tightening the variance bound. Throughout this discussion, the mini-batch $\mathcal{I}_k$ is constructed from a given collocation pool $\mathcal{D} = \mathcal{D}_\Omega \cup \mathcal{D}_{\Gamma_D} \cup \mathcal{D}_{\Gamma_N}$.

- *Unbiased gradients estimators:* The simplest way to obtain unbiased gradient estimates is to sample the mini-batches according to the loss weights defining $\widehat{R}$. Thus, given $\widehat{R} = \sum_S \gamma_S \widehat{R}_S$, $S \in \{\Omega, \Gamma_D, \Gamma_N\}$, we shall sample mini-batches using fixed proportions from $\{\Omega, \Gamma_D, \Gamma_N\}$ matching the weights $\{\gamma_S\}$. Taking expectations over uniform sampling within each subset yields $\mathbb{E}[g_{\mathcal{I}_k}(\theta)] = \sum_S \gamma_S \nabla_\theta \widehat{R}_S(\theta) = \nabla_\theta \widehat{R}(\theta)$, so the stochastic gradient estimator is unbiased.

  For example, for an equally weighted PINN loss over $\{\Omega, \Gamma_D, \Gamma_N\}$, the mini-batch should contain one third of its samples from each subset.

- *Initialization-gap reduction:* A complementary strategy to improve convergence is to target the initialization gap $\widehat{R}(\theta_0) - \widehat{R}^\star$ directly, by decomposing the problem into a sequence of progressively more difficult subproblems. This is analogous to continuation and homotopy methods in classical numerical analysis [2], as well as curriculum learning in classical ML [10]. In practice, this approach can be realized, for example, by gradually increasing the complexity of the forcing terms or boundary conditions [60].

  For time-dependent PDEs, the most natural realization is to respect the causal structure of the PDE, as described below. The main idea is to fit earlier times before progressively introducing later ones [119, 89, 75]. To this end, we restrict the temporal support of the interior/residual component of the mini-batch to $\{(x,t) \in \mathcal{I}_k^\Omega \mid t \leq T_{\text{cur}}(k)\}$ where t is current time and $T_{\text{cur}}(k)$ increases during training. Thus, instead of solving the full problem from a random initialization, we construct a sequence of progressively more difficult subproblems, each initialized from the solution of the previous one. As a result, each stage starts from a significantly better parameter estimate, leading to a much smaller effective initialization gap than the naive quantity $\widehat{R}(\theta_0^{\text{random}}) - \widehat{R}^\star$, where θ_0^{random} is a random initial guess.

- *Variance reduction:* Reducing the variance of the stochastic gradient estimates $g_{\mathcal{I}_k}$ can substantially improve the performance of the underlying optimization algorithms. The most straightforward approach is to increase the mini-batch size $|\mathcal{I}_k|$ as much as computationally feasible, cf. Numerical Example #4. More sophisticated strategies would include constructing mini-batches via random reshuffling methods [79], sampling with low-discrepancy sequences [84], or employing importance sampling techniques [83]. We also note that classical variance-reduction approaches, such as control-variate estimators widely used in Monte Carlo methods and reinforcement learning, have so far received little attention in the

PINN literature. Similarly, stochastic variance-reduction methods including SVRG [50] and SAGA [23] remain largely unexplored in PINN context. Their effective integration into PINN training may therefore provide promising directions for future research.

Among various variance reduction strategies, *stratified sampling* offers a particularly natural fit for PINNs, since the spatial structure of the domain can be exploited directly to reduce gradient variance. The idea is to partition the domain into a fixed grid of C cells $\{\Omega_c\}_{c=1}^{C}$, with weights $w_c = |\Omega_c|/V$, where V is the volume of the full domain, and to enforce that each mini-batch draws a fixed quota of samples from every cell. This removes the between-cell variance that arises from random fluctuations in spatial coverage under i.i.d. sampling.

More precisely, let $\phi(x) = \nabla_\theta r_\theta(x)\, r_\theta(x) \in \mathbb{R}^p$ denote the per-sample gradient contribution, and let $\sigma^2 = \mathrm{tr}(\mathrm{Cov}(\phi))$ denote its total variance. Denoting by $\mu_c = \mathbb{E}[\phi \mid x \in \Omega_c]$ and $\sigma_c^2 = \mathrm{tr}(\mathrm{Cov}(\phi \mid x \in \Omega_c))$ the mean and total variance of ϕ over cell Ω_c, and by $\mu = \sum_c w_c \mu_c$ the global mean, the total variance decomposes as

$$\sigma^2 = \underbrace{\sum_{c=1}^{C} w_c \sigma_c^2}_{\text{within-cell}} + \underbrace{\sum_{c=1}^{C} w_c \|\mu_c - \mu\|^2}_{\text{between-cell}}. \tag{47}$$

An i.i.d. mini-batch of size $|\mathcal{I}_k|$ yields variance $\sigma^2/|\mathcal{I}_k|$, retaining both terms. A stratified estimator, by enforcing a fixed quota of samples from each cell, eliminates the between-cell term $\sum_c w_c \|\mu_c - \mu\|^2$ by construction, yielding

$$\mathrm{Var}(g_{\mathcal{I}_k}) = \frac{1}{|\mathcal{I}_k|} \sum_{c=1}^{C} w_c \sigma_c^2,$$

which is always smaller than or equal to $\sigma^2/|\mathcal{I}_k|$, with equality only when all cell means μ_c are identical.

In PINNs, this can be particularly beneficial since $\phi(x)$ is typically highly heterogeneous in space (e.g., near boundary layers or underfitted regions). As a result, the between-cell variance is often large, and stratification can significantly reduce gradient variance at no additional computational cost per iteration.

Code snippet 19 provides an example of implementing the aforementioned mini-batching strategies in Python.

```
# Using equal loss-component fractions yield an unbiased gradient estimator
# for the equally-weighted loss R = R_Omega + R_IC + R_BC
STRAT_FRAC = (1/3, 1/3, 1/3)

# Causal schedule: linearly grow the admissible time horizon from T_MIN
```

```python
# at k=0 to t=1 once k >= T_WARM * K (fraction of total training)
T_MIN, T_WARM = 0.05, 0.3
def t_cur(k, K):
    # Current time horizon for causal time-marching at iteration k
    return T_MIN + (1.0 - T_MIN) * min(1.0, k / max(1, T_WARM * K))

def _split(B, frac):
    # Split batch budget B into (pde, ic, bc) counts
    # BC absorbs rounding so that bs_pde + bs_ic + bs_bc = B exactly
    bs_pde = round(B * frac[0])
    bs_ic  = round(B * frac[1])
    bs_bc  = B - bs_pde - bs_ic
    return bs_pde, bs_ic, bs_bc

def stratified_batch(xt_pde, xt_ic, xt_bc, B, **kwargs):
    # Loss-stratified sampling: fixed (1/3, 1/3, 1/3) split across interior
     part of the domain, ICs and BCs
    bs_pde, bs_ic, bs_bc = _split(B, STRAT_FRAC)
    return (xt_pde[torch.randperm(len(xt_pde))[:bs_pde]],
            xt_ic [torch.randperm(len(xt_ic)) [:bs_ic ]],
            xt_bc [torch.randperm(len(xt_bc)) [:bs_bc ]])


def sample_spatial(xt_pool, K):
    # Spatial-stratified sampling: partition the domain into a K_x x K_t
    # grid with K = K_x x K_t  cells and draw one point per cell
    # Choose grid dimensions to match the 2:1 aspect ratio of
     Omega=[-1,1]x[0,1]
    K_t = max(1, int(round((K / 2) ** 0.5)))
    K_x = max(1, int(round(K / K_t)))
    K   = K_x * K_t
    x, t = xt_pool[:, 0], xt_pool[:, 1]
    # Assign each pool point to a cell index in {0, ..., K-1}
    ix    = torch.clamp(((x + 1) / 2 * K_x).long(), 0, K_x - 1)
    it    = torch.clamp((t * K_t).long(), 0, K_t - 1)
    cells = ix * K_t + it
    # For each occupied cell, pick one point uniformly at random
    chosen = []
    for c in range(K):
        ci = (cells == c).nonzero(as_tuple=True)[0]
        if len(ci) > 0:
            chosen.append(ci[torch.randint(len(ci), (1,))].item())
    return torch.tensor(chosen, dtype=torch.long)


def loss_and_spatial_batch(xt_pde, xt_ic, xt_bc, B, **kwargs):
    # Loss and spatial-stratified sampling: loss split + spatial-cell
     stratification within the interior batch
    bs_pde, bs_ic, bs_bc = _split(B, STRAT_FRAC)
    idx_pde = sample_spatial(xt_pde, bs_pde)
    return (xt_pde[idx_pde],
            xt_ic[torch.randperm(len(xt_ic))[:bs_ic]],
            xt_bc[torch.randperm(len(xt_bc))[:bs_bc]])

def loss_and_causal_batch(xt_pde, xt_ic, xt_bc, B, k, K):
   # Loss-stratified sampling and causal time-marching: restrict interior
     and BC samples to t <= T_cur(k)
```

```
    # Progressively expanding the time horizon
    bs_pde, bs_ic, bs_bc = _split(B, STRAT_FRAC)
    Tk       = t_cur(k, K)
    # Find interior and BC points satisfying the causal constraint t <= Tk
    mask_pde = (xt_pde[:, 1] <= Tk).nonzero(as_tuple=True)[0]
    mask_bc  = (xt_bc [:, 1] <= Tk).nonzero(as_tuple=True)[0]
    # If fewer eligible points are available than needed, sample with
     replacement
    if len(mask_pde) >= bs_pde:
        idx_pde = mask_pde[torch.randperm(len(mask_pde))[:bs_pde]]
    else:
        idx_pde = mask_pde[torch.randint(len(mask_pde), (bs_pde,))]
    if len(mask_bc) >= bs_bc:
        idx_bc = mask_bc[torch.randperm(len(mask_bc))[:bs_bc]]
    else:
        idx_bc = mask_bc[torch.randint(len(mask_bc), (bs_bc,))]
    idx_ic = torch.randperm(len(xt_ic))[:bs_ic]
    return xt_pde[idx_pde], xt_ic[idx_ic], xt_bc[idx_bc]
```

Code snippet 19: Implementation of mini-batch construction strategies: loss-stratified, spatial-stratified, and causal time-marching.

Numerical Example #11: ***Impact of mini-batch construction strategies on PINNs training.*** In this example, we consider an advection-diffusion PDE, given as

$$u_t + c\,u_x - \mu\,u_{xx} = g(x,t), \qquad (x,t) \in (-1,1) \times (0,1),$$

with $c = 1$, $\mu = 10^{-2}$, so the problem is strongly advection-dominated. A manufactured exact solution is $u^\star(x,t) = \exp\big(-50(x + 0.5 - ct)^2\big)$ that advects from $x = -0.5$ to $x = +0.5$ across the time horizon. The source g and the non-homogeneous IC and BC are chosen such that $u^\star$ exactly satisfies the PDE. The PINN model is an MLP with four hidden layers of width 32, trained by Adam with a fixed learning rate $\alpha = 10^{-3}$ for 15,000 iterations. The collocation pool $\mathcal{D} = \mathcal{D}_\Omega \cup \mathcal{D}_{\Gamma_D} \cup \mathcal{D}_{\Gamma_N}$ contains $m_\Omega = 4{,}096$ i.i.d. uniform interior points and $m_{\text{IC}} = m_{\text{BC}} = 80$ equispaced initial condition and boundary points. On each iteration, Adam utilizes a mini-batch $\mathcal{I}_k \subset \mathcal{D}$ constructed either uniformly or by loss-component stratification. In addition, we consider variants that incorporate causal time-marching (loss stratification plus interior $t \le T_{\text{cur}}(k)$ growing linearly from $t = 0.05$ to $t = 1$ over the first 30% of training) and spatial stratification.

Figure 9.3 reports the relative L^2 test error and the training loss trajectories for all considered mini-batching strategies. As we can see, employing more advanced mini-batching strategies can improve the L^2 test error, compared to using the standard uniform approach.

Figure 9.4 demonstrates the performance of all considered mini-batching strategies for varying batch sizes. As shown, increasing the batch size significantly improves the accuracy of the trained PINN. We can also see that for very small batches ($|\mathcal{I}_k| \in \{4, 8, 16\}$), the uniform approach diverges, whereas more

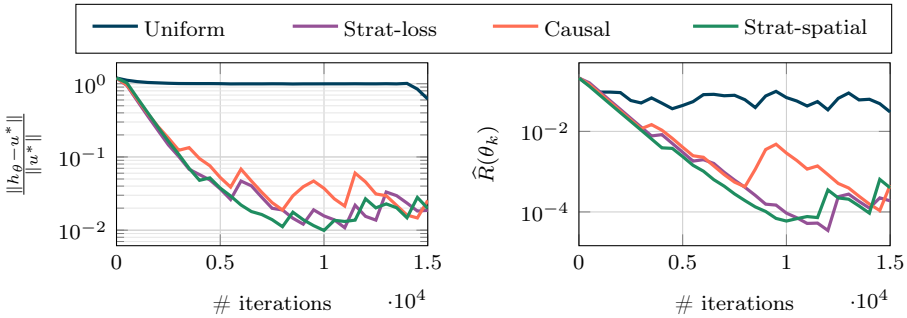

Figure 9.3: Training of PINN using Adam with $|\mathcal{I}_k| = 16$ for an advection-dominated PDE. The experiment is performed across four mini-batching strategies: uniform, loss-component stratified (Strat-loss), causal, and spatial stratification (Strat-spatial). *Left:* Relative L^2 test error. *Right:* Training loss.

advanced strategies enable Adam to converge. For small-to-medium batches ($|\mathcal{I}_k| \in \{16, 32, 64, 128\}$), the performance of the uniform mini-batching strategy becomes more competitive. In the large-batch regime ($|\mathcal{I}_k| \in \{256, 512\}$), the uniform approach recovers and achieves comparable accuracy as the advanced mini-batch construction approaches. Overall, these results indicate that advanced mini-batching techniques are particularly important under tight computational and memory budgets, i.e., $|\mathcal{I}_k| \lesssim 64$. Here, we note that the larger gains in accuracy are expected for stiffer evolution PDEs, such as those considered in [119, 20].

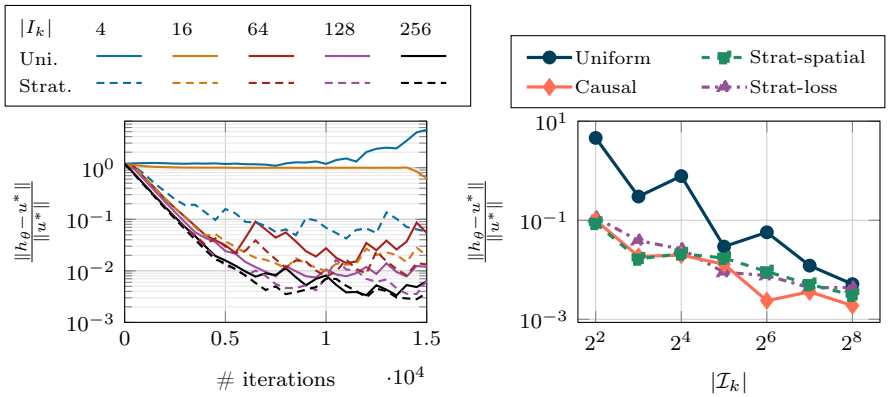

Figure 9.4: Relative test L^2-error of PINN for advection-diffusion example with respect to varying batch sizes $|\mathcal{I}_k|$. *Left:* Error observed during Adam's training with uniform and stratified-spatial mini-batching. *Right:* Final test error obtained using Adam with varying mini-batching strategies.

Key Takeaways

The choice of sampling and mini-batching strategies significantly influences the effectiveness of PINN training. For collocation points, adaptive resampling enriches the collocation set in poorly resolved regions, in turn reducing the condition number of the empirical NTK and improving convergence of the underlying optimizer. For mini-batches, loss-stratified sampling can be used to construct unbiased gradient estimates by enforcing fixed proportions of samples from each loss component $\{\Omega, \Gamma_D, \Gamma_N\}$, while spatial stratification can reduce gradient variance by ensuring uniform spatial coverage. In addition, causality-aware training can reduce the effective initialization gap by decomposing the problem into a sequence of progressively more difficult subproblems. Altogether, this leads to faster and more reliable convergence of stochastic optimizers applied to PINNs.

10 Towards scalable training for SciML: Recent advances and open challenges

In the preceding sections, we reviewed foundational optimization methods (SGD and its accelerated, adaptive, and second-order variants) together with SciML-specific design choices in loss formulation, model architecture, and sampling strategy. In this final section, we survey recent trends and open challenges in optimization for large-scale SciML training, organized according to whether the preconditioning strategy acts in parameter, data, or function space (cf. Section 3.1). Our review focuses on how classical scientific computing and ML techniques transfer to SciML, identifies their current limitations, and highlights open research directions that remain to be explored. Given the breadth and rapid evolution of the field, we provide a structured overview of selected research directions rather than a comprehensive survey.

10.1 Data-space preconditioning

Data-space preconditioning encompasses all strategies that improve optimization by transforming the inputs, reweighting the loss, or reshaping the sampling distribution, without modifying the network architecture or the optimizer itself.

Classical data-space preconditioning techniques in classical ML that are also widely used by SciML practitioners include feature scaling and standardisation [129], data augmentation [71], and equation non-dimensionalisation [118]. However, in SciML settings, these ubiquitous techniques often lack a rigorous theoretical foundation. For instance, in contrast to standard ML, normalization may interact in subtle ways with the PDE structure, or the constraint enforcement, potentially leading to undesirable optimization dynamics [51].

Other input transformations specifically designed for SciML have been motivated by taking into account physical considerations. In particular, input representations that enrich the coordinate system by introducing high-frequency components have been successfully proposed to mitigate the spectral bias of DNNs. Prominent examples include Fourier feature mappings [111] (see Section 8.1), positional encodings [92], or graph embeddings [72]. While these techniques substantially accelerate the learning of oscillatory or multiscale solutions, the choice of frequencies, bandwidths, or embedding structures remains largely problem-dependent and empirical, leaving systematic selection criteria as an open research direction [93, 120].

In the context of PINNs, *sampling strategies* also play a central role in determining how residual and BC's collocation points influence the conditioning of the underlying optimization problem (see Section 9). Consequently, many advanced sampling strategies can be interpreted as data-space preconditioners. Popular examples include variance-reduced and importance sampling strategies [83, 20], adaptive and residual-based sampling [125], or failure-informed or active sampling approaches [31]. Similarly, *adaptive weighting* strategies for balancing loss terms play a key role in stable and efficient PINN training (see Section 7). Existing methods dynamically adjust weights based on gradient magnitudes, NTK spectra, or residual statistics to alleviate stiffness and imbalance during optimization (e.g., [77, 110, 6, 121]). Despite their practical effectiveness, both sampling and weighting strategies remain largely heuristic, and developing principled, theoretically grounded alternatives constitutes an open research direction.

10.2 Parameter-space preconditioning

Parameter-space preconditioners improve optimization by modifying how parameter updates are computed, for example, by incorporating curvature information to transform the gradient direction, or by restricting updates to lower-dimensional, better-conditioned subspaces of the full parameter space.

Established optimization techniques such as AdaGrad [28], RMSProp [112], and Adam [54] are computationally efficient and robust in many large-scale ML applications, yet their behavior in SciML settings remains poorly understood. In particular, recent studies have reported gradient pathologies and unexpected optimizer dynamics in stiff PDE-constrained problems, suggesting nontrivial interactions between adaptive scaling and PDE-induced spectral properties [102].

A complementary class of methods attempts to approximate the inverse Hessian or Fisher information matrix. Algorithms such as natural gradient descent [3], K-FAC [74], Shampoo [38], SOAP [117], or MUON [66] maintain structured curvature approximations, often exploiting Kronecker or layerwise factorizations to remain tractable. However, due to their awareness of curvature, these approaches remain prohibitive in large-scale settings owing to substantial memory and computational overhead [128].

Specific to SciML are preconditioners that exploit its connections to numerical linear algebra and scientific computing. Several recent approaches

adapt large-scale preconditioning techniques from PDE solvers to DNN training, with a particular emphasis on *domain-decomposition and multilevel methods*. These methods provide implicit preconditioning by restricting updates to lower-dimensional, better-conditioned parameter subspaces, which is especially appealing in SciML, where parameter groups often align with physical modes or operator components.

In domain-decomposition approaches [56], the parameter space is often partitioned into blocks (e.g., layers, or channels), and optimization proceeds via additive or multiplicative Schwarz-type iterations; see, e.g., [58, 63, 37, 4, 106]. Closely related multilevel approaches equip the parameter space with a hierarchy of coarse-to-fine DNN representations, analogous to mesh hierarchies in multigrid methods [113]. Networks with fewer parameters, obtained for example by reducing network width, or depth, provide inexpensive approximations of curvature and enable the construction of coarse search directions that can accelerate convergence [30, 59, 14, 1, 91, 13].

Despite the promising empirical performance of several domain-decomposition and multilevel methods, the design of effective coarse spaces, inter-level transfer operators, and principled parameter decompositions with convergence guarantees in stochastic regimes remains an important open problem, with recent progress including multilevel AdaGrad [35, 34] and multilevel adaptive regularization techniques [73].

10.3 Function-space preconditioning

As introduced in Section 3.1, function-space preconditioners modify the geometry of the optimization problem directly in the space of functions represented by the DNN, for example, by changing the inner product or norm in which the loss is measured.

One example of function-space preconditioning is Sobolev training [19], which augments the loss with derivative information, thereby modifying the inner product in function space. In the SciML context, variants of this idea have been developed to guide PINNs toward solutions that minimize errors in Sobolev norms, often improving convergence rates and robustness relative to standard L_2 losses [109]. Nevertheless, the choice of derivative orders, relative weightings, and their interaction with PDE stiffness and BC conditions remains largely experimental, and a systematic theory linking Sobolev loss design to operator spectra and optimization conditioning remains an active area of research.

Beyond loss modification, function-space preconditioning can also be achieved by restructuring the functional basis itself, drawing again inspiration from domain-decomposition and multilevel methods in classical numerical analysis. Here, we note that some of the techniques discussed below admit a dual interpretation as they act simultaneously in function space and parameter space, and are discussed here primarily from the function-space viewpoint. In particular, in the context of PINNs, several domain-decomposition variants explicitly partition the physical domain and assign separate DNNs to each subregion, as in XPINNs [48], cPINNs [49], FBPINNs [82], or APINNs [45]. From a function-

space perspective, this implicitly constructs a basis of locally supported functions, yielding a block-structured representation that improves conditioning and stabilizes the optimization process. From a parameter-space perspective, the domain partitioning induces a block-diagonal structure on the parameter space, analogous to a Schwarz-type preconditioners. It is worth noting that these methods also enable parallelization and facilitate the representation of localized or multiscale features, often making PDE problems that are otherwise inaccessible to monolithic PINNs tractable.

Similarly, multiscale architectures such as those discussed in Section 8.2 can be interpreted as function-space preconditioners since by combining subnetworks operating at different frequency scales, they partition the functional basis across frequency bands, reducing spectral bias and improving NTK conditioning. More broadly, several recent multilevel strategies construct hierarchies of function spaces to train SciML models more efficiently. By first learning a coarse approximation and then refining the function space, these approaches also reduce spectral bias and accelerate convergence. Representative examples include multilevel training for data-driven models [70] and multifidelity PINNs [98, 78], block-structured coarse-to-fine training [36], and multilevel domain-decomposition methods [27], while related multiresolution and wavelet- or Fourier-based architectures [42] achieve similar effects by separating low- and high-frequency components.

Despite the empirical success of these approaches, systematic design principles for subdomain partitioning, interface conditions, coarse space selection, and convergence guarantees in stochastic training regimes remain important open problems.

11 Conclusion and outlook

In this chapter, we reviewed optimization techniques for training SciML models, emphasizing that efficient training requires not only a suitable optimizer, but also problem-aware design choices in the loss formulation, model architecture, and sampling strategy. These two aspects were addressed in complementary parts of the chapter. Foundational optimization tools were covered in Sections 2–6, while problem-aware design choices were discussed in Sections 7–9.

The preconditioning viewpoint introduced in Section 3.1 provides a unifying framework for organizing the existing algorithms and connecting them to ideas from classical scientific computing. The large-scale approaches reviewed in Section 10 illustrate how modern SciML training increasingly draws on principles from numerical PDE solvers while retaining the flexibility of DNNs. Together, data-space, parameter-space, and function-space preconditioning strategies tackle complementary challenges of the underlying optimization problem, including spectral bias, ill-conditioning, and the multiscale nature of PDE solutions, offering a unified perspective for improving the efficiency and reliability of large-scale SciML training.

Nevertheless, many open questions remain, particularly in nonconvex and

stochastic settings, and the relationships between these three preconditioning viewpoints are still only partially understood. As a result, the design of efficient optimization techniques remains an active area of research, with continued progress needed to develop scalable and reliable methods for increasingly complex SciML applications.

Acknowledgements

The work of A.K. benefited from Artificial and Natural Intelligence Toulouse Institute (ANITI), funded by the France 2030 program under Grant Agreement No. ANR-23-IACL-0002. The work of E.R. was supported by the ANR project MEPHISTO (ANR-24-CE23-7039), the PEPR IA SHARP and the Fondation Simone et Cino Del Duca.

References

[1] S. Ahamed, N. Zakariaei, E. Haber, and M. Eliasof, *Multiscale training of convolutional neural networks*, arXiv preprint arXiv:2501.12739, (2025).

[2] E. L. Allgower and K. Georg, *Introduction to numerical continuation methods*, SIAM, 2003.

[3] S.-i. Amari, *Natural gradient works efficiently in learning*, Neural Computation, 10 (1998), pp. 251–276.

[4] E. Amid, R. Anil, and M. Warmuth, *Locoprop: Enhancing backprop via local loss optimization*, in International conference on artificial intelligence and statistics, PMLR, 2022, pp. 9626–9642.

[5] H. Anagnostopoulos, M. Papachristou, D. Giovanis, and P. Koumoutsakos, *Residual-based adaptive reweighting for physics-informed neural networks*, Journal of Computational Physics, 478 (2023), p. 111959.

[6] S. J. Anagnostopoulos, J. D. Toscano, N. Stergiopulos, and G. E. Karniadakis, *Residual-based attention in physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 421 (2024), p. 116805.

[7] S. Arora, S. S. Du, W. Hu, Z. Li, R. R. Salakhutdinov, and R. Wang, *On exact computation with an infinitely wide neural net*, Advances in neural information processing systems, 32 (2019).

[8] F. Bach, *Breaking the curse of dimensionality with convex neural networks*, Journal of Machine Learning Research, 18 (2017), pp. 1–53.

[9] A. Beck, *First-order methods in optimization*, SIAM, 2017.

[10] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, *Curriculum learning*, in ICML, 2009, pp. 41–48.

[11] Y. Bengio, N. Roux, P. Vincent, O. Delalleau, and P. Marcotte, *Convex neural networks*, Advances in neural information processing systems, 18 (2005).

[12] L. Bottou, F. E. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning*, Siam Review, 60 (2018), pp. 223–311.

[13] H. Calandra, S. Gratton, E. Riccietti, and X. Vasseur, *On a multilevel Levenberg–Marquardt method for the training of artificial neural networks and its application to the solution of partial differential equations*, Optimization Methods and Software, 37 (2022), pp. 361–386.

[14] B. Chang, L. Meng, E. Haber, F. Tung, and D. Begert, *Multi-level residual networks from dynamical systems view*, arXiv preprint arXiv:1710.10348, (2017).

[15] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, *Neural ordinary differential equations*, Advances in neural information processing systems, 31 (2018).

[16] T. Chen, L. Lu, and G. Karniadakis, *BRDR: Boundary residual decay reweighting for physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 418 (2024), p. 116731.

[17] Z. Chen, V. Badrinarayanan, C.-Y. Lee, and A. Rabinovich, *Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks*, in International conference on machine learning, PMLR, 2018, pp. 794–803.

[18] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust Region Methods*, MOS-SIAM Series on Optimization, SIAM, 2000.

[19] W. M. Czarnecki, S. Osindero, M. Jaderberg, G. Swirszcz, and R. Pascanu, *Sobolev training for neural networks*, Advances in neural information processing systems, 30 (2017).

[20] A. Daw, J. Bu, S. Wang, P. Perdikaris, and A. Karpatne, *Rethinking the importance of sampling in physics-informed neural networks*, arXiv preprint arXiv:2207.02338, (2022).

[21] T. De Ryck, F. Bonnet, S. Mishra, and E. de Bézenac, *An operator preconditioning perspective on training in physics-informed machine learning*, arXiv preprint arXiv:2310.05801, (2023).

[22] T. De Ryck, S. Mishra, and R. Molinaro, *wpinns: Weak physics informed neural networks for approximating entropy solutions of hyperbolic conservation laws*, SIAM Journal on Numerical Analysis, 62 (2024), pp. 811–841.

[23] A. Defazio, F. Bach, and S. Lacoste-Julien, *Saga: A fast incremental gradient method with support for non-strongly convex composite objectives*, Advances in neural information processing systems, 27 (2014).

[24] A. Défossez, L. Bottou, F. Bach, and N. Usunier, *A simple convergence proof of adam and adagrad*, arXiv preprint arXiv:2003.02395, (2020).

[25] R. S. Dembo, S. C. Eisenstat, and T. Steihaug, *Inexact newton methods*, SIAM Journal on Numerical analysis, 19 (1982), pp. 400–408.

[26] J. Dick, F. Y. Kuo, and I. H. Sloan, *High-dimensional integration: the quasi-Monte Carlo way*, Acta Numerica, 22 (2013), pp. 133–288.

[27] V. Dolean, A. Heinlein, S. Mishra, and B. Moseley, *Multilevel domain decomposition-based architectures for physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 429 (2024), p. 117116.

[28] J. Duchi, E. Hazan, and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization.*, Journal of machine learning research, 12 (2011).

[29] S. C. Eisenstat and H. F. Walker, *Choosing the forcing terms in an inexact newton method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 16–32.

[30] L. Gaedke-Merzhäuser, A. Kopaničáková, and R. Krause, *Multilevel minimization for deep residual networks*, in Proceedings of French-German-Swiss Optimization Conference (FGS'2019), 2021.

[31] Z. Gao, L. Yan, and T. Zhou, *Failure-informed adaptive sampling for pinns*, SIAM Journal on Scientific Computing, 45 (2023), pp. A1971–A1994.

[32] G. Garrigos and R. M. Gower, *Handbook of convergence theorems for (stochastic) gradient methods*, arXiv preprint arXiv:2301.11235, (2023).

[33] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 2010, pp. 249–256.

[34] S. Gratton, A. Kopaničáková, and P. Toint, *Recursive bound-constrained AdaGrad with applications to multilevel and domain decomposition minimization*, arXiv preprint arXiv:2507.11513, (2025).

[35] S. Gratton, A. Kopaničáková, and P. L. Toint, *Multilevel objective-function-free optimization with an application to neural networks training*, SIAM Journal on Optimization, 33 (2023), pp. 2772–2800.

[36] S. Gratton, V. Mercier, E. Riccietti, and P. L. Toint, *A block-coordinate approach of multi-level optimization with an application to physics-informed neural networks*, Computational Optimization and Applications, 89 (2024), pp. 385–417.

[37] L. Gu, W. Zhang, J. Liu, and X.-C. Cai, *Decomposition and composition of deep convolutional neural networks and training acceleration via sub-network transfer learning*, (2022).

[38] V. Gupta, T. Koren, and Y. Singer, *Shampoo: Preconditioned stochastic tensor optimization*, in International Conference on Machine Learning, PMLR, 2018, pp. 1842–1850.

[39] N. Halko, P.-G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM review, 53 (2011), pp. 217–288.

[40] Z. Hao, C. Ying, H. Su, J. Zhu, J. Song, and Z. Cheng, *Bi-level physics-informed neural networks for pde constrained optimization using broyden's hypergradients*, arXiv preprint arXiv:2209.07075, (2022).

[41] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1026–1034.

[42] W. He, J. Li, X. Kong, and L. Deng, *Multi-level physics informed deep learning for solving partial differential equations in computational structural mechanics*, Communications Engineering, 3 (2024), p. 151.

[43] M. Heusel et al., *GANs trained by a two time-scale update rule converge to a local nash equilibrium*, in NeurIPS, 2017.

[44] R. A. Horn and C. R. Johnson, *Matrix analysis*, Cambridge university press, 2012.

[45] Z. Hu, A. D. Jagtap, G. E. Karniadakis, and K. Kawaguchi, *Augmented Physics-Informed Neural Networks (APINNs): A gating network-based soft domain decomposition methodology*, Engineering Applications of Artificial Intelligence, 126 (2023), p. 107183.

[46] B. Irwin and E. Haber, *Secant penalized BFGS: a noise robust quasi-Newton method via penalizing the secant condition*, Computational Optimization and Applications, 84.3 (2023), pp. 651–702.

[47] A. Jacot, F. Gabriel, and C. Hongler, *Neural tangent kernel: Convergence and generalization in neural networks*, NeurIPS, (2018).

[48] A. D. Jagtap and G. E. Karniadakis, *Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations*, Communications in Computational Physics, 28 (2020).

[49] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis, *Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems*, Computer Methods in Applied Mechanics and Engineering, 365 (2020), p. 113028.

[50] R. Johnson and T. Zhang, *Accelerating stochastic gradient descent using predictive variance reduction*, Advances in neural information processing systems, 26 (2013).

[51] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, *Physics-informed machine learning*, Nature Reviews Physics, 3 (2021), pp. 422–440.

[52] A. Kendall, Y. Gal, and R. Cipolla, *Multi-task learning using uncertainty to weigh losses for scene geometry and semantics*, CVPR, (2018), pp. 7482–7491.

[53] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, *On large-batch training for deep learning: Generalization gap and sharp minima*, ICLR, (2017).

[54] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[55] E. Kiyani, K. Shukla, J. F. Urbán, J. Darbon, and G. E. Karniadakis, *Optimizing the optimizer for physics-informed neural networks and Kolmogorov-Arnold networks*, Computer Methods in Applied Mechanics and Engineering, 446 (2025), p. 118308.

[56] A. Klawonn, M. Lanser, and J. Weber, *Machine learning and domain decomposition methods-a survey*, Computational Science and Engineering, 1 (2024), p. 2.

[57] D. A. Knoll and D. E. Keyes, *Jacobian-free Newton–Krylov methods: a survey of approaches and applications*, Journal of Computational Physics, 193 (2004), pp. 357–397.

[58] A. Kopaničáková, H. Kothari, G. E. Karniadakis, and R. Krause, *Enhancing training of physics-informed neural networks using domain decomposition–based preconditioning strategies*, SIAM Journal on Scientific Computing, 46 (2024), pp. S46–S67.

[59] A. Kopaničáková and R. Krause, *Globally Convergent Multilevel Training of Deep Residual Networks*, SIAM Journal on Scientific Computing, 45 (2023), pp. S254–S280.

[60] A. Krishnapriyan, A. Gholami, S. Zhe, R. Kirby, and M. Mahoney, *Characterizing possible failure modes in physics-informed neural networks*, Advances in neural information processing systems, 34 (2021), pp. 26548–26560.

[61] M. Lazzara, M. Chevalier, M. Colombo, J. G. Garcia, C. Lapeyre, and O. Teste, *Surrogate modelling for an aircraft dynamic landing loads simulation using an lstm autoencoder-based dimensionality reduction approach*, Aerospace Science and Technology, 126 (2022), p. 107629.

[62] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, *Wide neural networks of any depth evolve as linear models under gradient descent*, Advances in neural information processing systems, 32 (2019).

[63] Y. Lee, A. Kopaničáková, and G. E. Karniadakis, *Two-level overlapping additive Schwarz preconditioner for training scientific machine learning applications*, Computer Methods in Applied Mechanics and Engineering, 448 (2026), p. 118400.

[64] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, *Visualizing the loss landscape of neural nets*, Advances in neural information processing systems, 31 (2018).

[65] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Mathematical Programming, 45 (1989), pp. 503–528.

[66] J. Liu, J. Su, X. Yao, Z. Jiang, G. Lai, Y. Du, Y. Qin, W. Xu, E. Lu, J. Yan, et al., *Muon is scalable for llm training*, arXiv preprint arXiv:2502.16982, (2025).

[67] I. Loshchilov and F. Hutter, *SGDR: Stochastic gradient descent with warm restarts*, in ICLR, 2017.

[68] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*, Nature Machine Intelligence, 3 (2021), pp. 218–229.

[69] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, *DeepXDE: A deep learning library for solving differential equations*, SIAM review, 63 (2021), pp. 208–228.

[70] K. O. Lye, S. Mishra, and R. Molinaro, *A multi-level procedure for enhancing accuracy of machine learning algorithms*, European Journal of Applied Mathematics, 32 (2021), pp. 436–469.

[71] K. Maharana, S. Mondal, and B. Nemade, *A review: Data pre-processing and data augmentation techniques*, Global Transitions Proceedings, 3 (2022), pp. 91–99.

[72] I. Makarov, D. Kiselev, N. Nikitinsky, and L. Subelj, *Survey on graph embeddings and their applications to machine learning problems on graphs*, PeerJ Computer Science, 7 (2021), p. e357.

[73] F. Marini, M. Porcelli, and E. Riccietti, *A multilevel stochastic regularized first-order method with application to finite sum minimization*, arXiv preprint arXiv: arXiv:2412.11630, (2024).

[74] J. Martens and R. Grosse, *Optimizing neural networks with Kronecker-factored approximate curvature*, in International conference on machine learning, PMLR, 2015, pp. 2408–2417.

[75] R. Mattey and S. Ghosh, *A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations*, Computer Methods in Applied Mechanics and Engineering, 390 (2022), p. 114474.

[76] L. McClenny and U. Braga-Neto, *Self-adaptive physics-informed neural networks using a residual-based adaptive activation*, arXiv:2009.04544, (2020).

[77] L. D. McClenny and U. M. Braga-Neto, *Self-adaptive physics-informed neural networks*, Journal of Computational Physics, 474 (2023), p. 111722.

[78] X. Meng and G. E. Karniadakis, *A composite neural network that learns from multi-fidelity data: Application to function approximation and inverse pde problems*, Journal of Computational Physics, 401 (2020), p. 109020.

[79] K. Mishchenko, A. Khaled, and P. Richtárik, *Random reshuffling: Simple analysis with vast improvements*, Advances in Neural Information Processing Systems, 33 (2020), pp. 17309–17320.

[80] S. Mishra, *A machine learning framework for data driven acceleration of computations of differential equations*, arXiv preprint arXiv:1807.09519, (2018).

[81] S. Mishra and R. Molinaro, *Estimates on the generalization error of physics-informed neural networks for approximating pdes*, IMA Journal of Numerical Analysis, 43 (2023), pp. 1–43.

[82] B. Moseley, A. Markham, and T. Nissen-Meyer, *Finite basis physics-informed neural networks (FBPINNs): a scalable domain decomposition approach for solving differential equations*, Advances in Computational Mathematics, 49 (2023), p. 62.

[83] M. A. Nabian, R. J. Gladstone, and H. Meidani, *Efficient training of physics-informed neural networks via importance sampling*, Computer-Aided Civil and Infrastructure Engineering, 36 (2021), pp. 962–977.

[84] H. Niederreiter, *Random Number Generation and Quasi–Monte Carlo Methods*, SIAM, 1992.

[85] J. Nocedal, *Updating quasi-newton matrices with limited storage*, Mathematics of computation, 35 (1980), pp. 773–782.

[86] J. Nocedal and S. Wright, *Numerical optimization*, Springer Science & Business Media, 2006.

[87] R. Novak, J. Sohl-Dickstein, and S. S. Schoenholz, *Fast finite width neural tangent kernel*, in Proceedings of the 39th International Conference on Machine Learning, PMLR, 2022, pp. 17018–17044.

[88] R. Novak, L. Xiao, J. Hron, J. Lee, A. A. Alemi, J. Sohl-Dickstein, and S. S. Schoenholz, *Neural tangents: Fast and easy infinite neural networks in python*, arXiv preprint arXiv:1912.02803, (2019).

[89] M. Penwarden, A. D. Jagtap, S. Zhe, G. E. Karniadakis, and R. M. Kirby, *A unified scalable framework for causal sweeping strategies for physics-informed neural networks (pinns) and their temporal decompositions*, Journal of Computational Physics, 493 (2023), p. 112464.

[90] B. T. Polyak, *Some methods of speeding up the convergence of iteration methods*, Ussr computational mathematics and mathematical physics, 4 (1964), pp. 1–17.

[91] C. Ponce, R. Li, C. Mao, and P. Vassilevski, *Multilevel-in-width training for deep neural network regression*, Numerical Linear Algebra with Applications, 30 (2023), p. e2501.

[92] A. Quarteroni, P. Gervasio, and F. Regazzoni, *Combining physics-based and data-driven models: advancing the frontiers of research with scientific machine learning*, arXiv preprint arXiv:2501.18708, (2025).

[93] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville, *On the spectral bias of neural networks*, in International conference on machine learning, PMLR, 2019, pp. 5301–5310.

[94] A. Rahimi and B. Recht, *Random features for large-scale kernel machines*, in Advances in neural information processing systems, vol. 20, 2007.

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

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

[97] S. J. Reddi, S. Kale, and S. Kumar, *On the convergence of adam and beyond*, arXiv preprint arXiv:1904.09237, (2019).

[98] E. Riccietti, V. Mercier, S. Gratton, and P. Boudier, *Multilevel physics informed neural networks (mpinns)*, in The International Conference on Learning Representations (ICLR), 2022, pp. 1–15.

[99] H. Robbins and S. Monro, *A stochastic approximation method*, The annals of mathematical statistics, (1951), pp. 400–407.

[100] F. Roosta-Khorasani and M. W. Mahoney, *Sub-sampled Newton methods*, Mathematical Programming, 174 (2019), pp. 293–326.

[101] A. Ruiz, C. Osorio, J. Sanchez, C. E. Pachon, and A. Özisari, *When physics meets deep learning: Training dynamics and diagnostic tools for physics-informed neural networks*, Journal of Computational Physics, 463 (2022), p. 111262.

[102] M. Scott, T. Xu, Z. Tang, A. Pichette-Emmons, Q. Ye, Y. Saad, and Y. Xi, *Designing preconditioners for sgd: Local conditioning, noise floors, and basin stability*, arXiv preprint arXiv:2511.19716, (2025).

[103] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.

[104] C. J. Shallue, J. Lee, J. Antognini, et al., *Measuring the effects of data parallelism on neural network training*, JMLR, (2019).

[105] H.-J. M. Shi, Y. Xie, R. Byrd, and J. Nocedal, *A noise-tolerant quasi-Newton algorithm for unconstrained optimization*, SIAM Journal on Optimization, 32 (2022), pp. 29–55.

[106] J. W. Siegel, Q. Hong, X. Jin, W. Hao, and J. Xu, *Greedy training algorithms for neural networks and applications to pdes*, Journal of Computational Physics, 484 (2023), p. 112084.

[107] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, *Implicit neural representations with periodic activation functions*, in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020, pp. 7462–7473.

[108] S. L. Smith, P.-J. Kindermans, and Q. V. Le, *Don't decay the learning rate, increase the batch size*, in ICLR, 2018.

[109] H. Son, J. W. Jang, W. J. Han, and H. J. Hwang, *Sobolev training for physics informed neural networks*, arXiv preprint arXiv:2101.08932, (2021).

[110] S. Subramanian, R. M. Kirby, M. W. Mahoney, and A. Gholami, *Adaptive self-supervision algorithms for physics-informed neural networks*, arXiv preprint arXiv:2207.04084, (2022).

[111] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng, *Fourier features let networks learn high frequency functions in low dimensional domains*, Advances in neural information processing systems, 33 (2020), pp. 7537–7547.

[112] T. Tieleman, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*, COURSERA: Neural networks for machine learning, 4 (2012), p. 26.

[113] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Elsevier, 2000.

[114] V. Vapnik, *Statistical Learning Theory*, Wiley, 1998.

[115] R. Verfürth, *A posteriori error estimation and adaptive mesh-refinement techniques*, Journal of Computational and Applied Mathematics, 50 (1994), pp. 67–83.

[116] C. Visser, A. Heinlein, and B. Giovanardi, *Pacmann: Point adaptive collocation method for artificial neural networks*, Computer Methods in Applied Mechanics and Engineering, 452 (2026), p. 118723.

[117] N. Vyas, D. Morwani, R. Zhao, M. Kwun, I. Shapira, D. Brandfonbrener, L. Janson, and S. Kakade, *Soap: Improving and stabilizing Shampoo using Adam*, arXiv preprint arXiv:2409.11321, (2024).

[118] S. Wang, S. Sankaran, and P. Perdikaris, *An expert's guide to training physics-informed neural networks*, Journal of Computational Physics, 473 (2023), p. 111742.

[119] ——, *Respecting causality is all you need for training physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 421 (2024), p. 116813.

[120] S. Wang, Y. Teng, and P. Perdikaris, *Understanding and mitigating gradient flow pathologies in physics-informed neural networks*, SIAM Journal on Scientific Computing, 43 (2021), pp. A3055–A3081.

[121] S. Wang, Y. Teng, and P. Perdikaris, *When and why PINNs fail to train: A neural tangent kernel perspective*, Journal of Computational Physics, 449 (2022), p. 110768.

[122] S. Wang, H. Wang, and P. Perdikaris, *On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 384 (2021), p. 113938.

[123] Z. Wang and A. Yurtsever, *Generalized stochastic gradient descent with momentum methods for smooth optimization*, arXiv preprint arXiv:2602.23444, (2026).

[124] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, *The marginal value of adaptive gradient methods in machine learning*, NeurIPS, (2017).

[125] C. Wu, M. Zhu, Q. Tan, Y. Kartha, and L. Lu, *A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks*, Computer Methods in Applied Mechanics and Engineering, 403 (2023), p. 115671.

[126] L. Xiang, L. Peng, X. Chen, and G. Lin, *lbPINN: A likelihood-based physics-informed neural network*, Journal of Computational Physics, 451 (2021), p. 110829.

[127] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, *Adaptive methods for nonconvex optimization*, Advances in neural information processing systems, 31 (2018).

[128] S. Zampini, U. Zerbinati, G. Turkyyiah, and D. Keyes, *PETScML: Second-order solvers for training regression problems in Scientific Machine Learning*, in Proceedings of the Platform for Advanced Scientific Computing Conference, 2024, pp. 1–12.

[129] H. Zhongkai, J. Yao, C. Su, H. Su, Z. Wang, F. Lu, Z. Xia, Y. Zhang, S. Liu, L. Lu, et al., *Pinnacle: A comprehensive benchmark of physics-informed neural networks for solving pdes*, Advances in Neural Information Processing Systems, 37 (2024), pp. 76721–76774.