Scientific Machine Learning Workshop

Lecture 3: Physics-Informed Neural Networks

Ulisses Braga Neto

Department of Electrical and Computer Engineering
Scientific Machine Learning Lab (SciML Lab)
Texas A&M Institute of Data Science (TAMIDS)
Texas A&M University

Cenpes
August 2025

Physics-Informed Neural Networks (PINNs)

- Perhaps the most basic technique in physics-informed machine learning is constraining the output of a deep neural network by adding a term to the loss function that penalizes a lack of agreement with a PDE.
- This is a "soft" constraint, in the sense that the PDE is only approximately satisfied, even after training.
- This is known as a *physics-informed neural network* (PINN), which we describe in detail next, for both forward and inverse problems.
- We also discuss a few ways to impose hard physics constraints on a neural network (which, strictly speaking, should not be called a PINN).

Partial Differential Equation Models

- Let the open set $\Omega \subset R^d$ represent a physical domain, and let $u : \Omega \cup \partial\Omega \to R$ be a physical quantity, where $\partial\Omega$ denotes the boundary of Ω.
- Let $D^\alpha u(\boldsymbol{x})$ be the set of all partial derivatives of u of order α:

$$D^\alpha u(\boldsymbol{x}) = \left\{ \frac{\partial^\alpha u}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}} \;\middle|\; \alpha_1 + \cdots + \alpha_d = \alpha \right\}.$$

- A PDE of order k is an equation of the kind:

$$F(u(\boldsymbol{x}), \boldsymbol{x}, Du(\boldsymbol{x}), D^2 u(\boldsymbol{x}), \ldots, D^k u(\boldsymbol{x}); \lambda) = f(\boldsymbol{x}), \qquad (1)$$

  which must be satisfied at all $\boldsymbol{x} \in \Omega$, for a given function F, parameter λ (this could be a scalar, a vector, or a function), and a *source* or *forcing* function f (the *unforced* case $f \equiv 0$ being common).

Partial Differential Equation Models

- PDEs generally admit an infinite number of solutions. In order to obtain a unique solution, additional information on the unknown u or its derivatives must be provided.
- It is usually the case that physical considerations provide this information on all or part of the boundary $\partial\Omega$ of the physical domain.
- These *boundary conditions* can be written as

$$G(u(\boldsymbol{x}), \boldsymbol{x}, Du(\boldsymbol{x}), D^2u(\boldsymbol{x}), \ldots, D^m u(\boldsymbol{x})) = g(\boldsymbol{x}), \qquad (2)$$

  which must be satisfied at all $\boldsymbol{x} \in \Gamma \subseteq \delta\Omega$, for a given function G and boundary data g.
- In a time evolution problem, data on the boundary $t = 0$ is called an *initial condition*.

PINN - Forward Problem

- In the forward problem, all the parameters, including the boundary conditions, are known.
- This is therefore a pure PDE solving problem, in which the PINN is used as a traditional numerical method.
- The problem where there is incomplete information, which must be estimated from sample data, is called the inverse problem; this will be discussed later in the lecture.
- Let the unknown $u(\boldsymbol{x})$ be approximated by an *ansatz* $u(\boldsymbol{x}; \boldsymbol{w})$, consisting of a deep neural network with inputs $\boldsymbol{x} = (x_1, \ldots, x_d)$ and network weights $\boldsymbol{w}$.
- There is only one output, since here u is a scalar function; in the case of a PDE system, there would be multiple outputs, one for each component.

PINN - Forward Problem

- The ansatz is substituted in (1) and (2), and the task is to find network weights $\boldsymbol{w}$ such that

$$F(u(\boldsymbol{x};\boldsymbol{w}),\boldsymbol{x},Du(\boldsymbol{x};\boldsymbol{w}),\ldots,D^k u(\boldsymbol{x};\boldsymbol{w});\lambda) = f(\boldsymbol{x}), \quad \boldsymbol{x}\in\Omega,$$
$$G(u(\boldsymbol{x};\boldsymbol{w}),\boldsymbol{x},Du(\boldsymbol{x};\boldsymbol{w}),\ldots,D^m u(\boldsymbol{x};\boldsymbol{w})) = g(\boldsymbol{x}), \quad \boldsymbol{x}\in\Gamma.$$

- The "secret sauce" of PINNs is that the values of F and G at given values of $\boldsymbol{x}$ and $\boldsymbol{w}$ can be computed accurately and efficiently by *automatic differentiation* of the neural network.
- Moreover, applying AD on neural network produces another neural network (another computational graph). Hence, F is a "residual neural network" and G is a "boundary neural network".
- Now, consider the following loss functions

$$\mathcal{L}_r(\boldsymbol{w}) = \int_\Omega |F(u(\boldsymbol{x};\boldsymbol{w}),\boldsymbol{x},Du(\boldsymbol{x};\boldsymbol{w}),\ldots,D^k u(\boldsymbol{x};\boldsymbol{w});\lambda) - f(\boldsymbol{x})|^2 d\boldsymbol{x},$$
$$\mathcal{L}_b(\boldsymbol{w}) = \int_\Gamma |G(u(\boldsymbol{x};\boldsymbol{w}),\boldsymbol{x},Du(\boldsymbol{x};\boldsymbol{w}),\ldots,D^m u(\boldsymbol{x};\boldsymbol{w})) - g(\boldsymbol{x})|^2 d\boldsymbol{x}.$$

PINN - Forward Problem

- It is clear that $\mathcal{L}_r(\boldsymbol{w}), \mathcal{L}_b(\boldsymbol{w}) \geq 0$. Hence, $u(\boldsymbol{x}; \boldsymbol{w})$ satisfies the PDE and the boundary conditions everywhere (outside a set of measure zero) *if and only if* $\mathcal{L}_r(\boldsymbol{w}) = \mathcal{L}_b(\boldsymbol{w}) = 0$ or, equivalently, $\mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) = 0$.

- In practice, the integrals in (6) must be numerically approximated. The simplest way to do so, but not necessarily the only way, is to form Monte-Carlo estimates using a uniformly distributed random samples $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r} \subset \Omega$ and $\{\boldsymbol{x}_b^i\}_{i=1}^{N_b} \subset \Gamma$:

$$\mathcal{L}_r(\boldsymbol{w}) \doteq \frac{1}{N_r} \sum_{i=1}^{N_r} |F(u(\boldsymbol{x}_r^i; \boldsymbol{w}), \boldsymbol{x}_r^i, Du(\boldsymbol{x}_r^i; \boldsymbol{w}), \ldots, D^k u(\boldsymbol{x}_r^i; \boldsymbol{w}); \lambda) - f(\boldsymbol{x}_r^i)|^2,$$

$$\mathcal{L}_b(\boldsymbol{w}) \doteq \frac{1}{N_b} \sum_{i=1}^{N_b} |G(u(\boldsymbol{x}_b^i; \boldsymbol{w}), \boldsymbol{x}_b^i, Du(\boldsymbol{x}_b^i; \boldsymbol{w}), \ldots, D^m u(\boldsymbol{x}_b^i; \boldsymbol{w})) - g(\boldsymbol{x}_b^i)|^2.$$

PINN - Forward Problem

- The neural network is trained by minimizing the combined loss

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w})$$

  using any of the optimization methods described in Lecture 2.

- PINN is a *meshless* method, as the *collocation points* $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r}$ and boundary points $\{\boldsymbol{x}_b^i\}_{i=1}^{N_b}$ do not have to be located on regular meshes.

- This makes PINN more suitable for high-dimensional problems and problems with irregular boundaries than traditional mesh-based numerical methods.

PINN - Forward Problem

- On the other hand, the discretization of the loss functions means that, even if $\mathcal{L}(\boldsymbol{w}) = 0$, the PDE and the boundary conditions are guaranteed to be satisfied only at the collocation points and boundary points, respectively.
- Using ever larger numbers of points would appear at first to be recommended to obtain more accurate results; however, one will eventually run into computational problems and difficulty in training the neural network.
- The use of *stochastic gradient descent* can be very effective in this regard: instead of sampling a large batch of collocation and boundary points at the start of training, one can sample a smaller batch and sample new batches of the same size on a regular schedule. Not only does this reduce the computational problem, but it has also been found to produce much more accurate PINNs.

PINN - Forward Problem

- The preferred accuracy metric for PINNs is the *relative L2 error*:

$$\text{relative L2 error} = \frac{\sqrt{\int_\Omega |u(\boldsymbol{x};\boldsymbol{w}) - u(\boldsymbol{x})|^2 \, d\boldsymbol{x}}}{\sqrt{\int_\Omega |u(\boldsymbol{x})|^2 \, d\boldsymbol{x}}} .$$

- The normalization is desirable to remove a dependency on the units of the solution; it is a form of *nondimensionalization*.

- In practice, the integrals have to be approximated, by generating a uniformly distributed random test sample $\{\boldsymbol{x}_t^i\}_{i=1}^{N_t}$ and defining

$$\text{relative L2 error} \doteq \frac{\sum_{i=1}^{N_t} |u(\boldsymbol{x}_t^i;\boldsymbol{w}) - u(\boldsymbol{x}_t^i)|^2}{\sum_{i=1}^{N_t} |u(\boldsymbol{x}_t^i)|^2} .$$

- A relative L2 error of 1% or less is generally considered very good.

Example: Viscous Burgers - Forward Problem

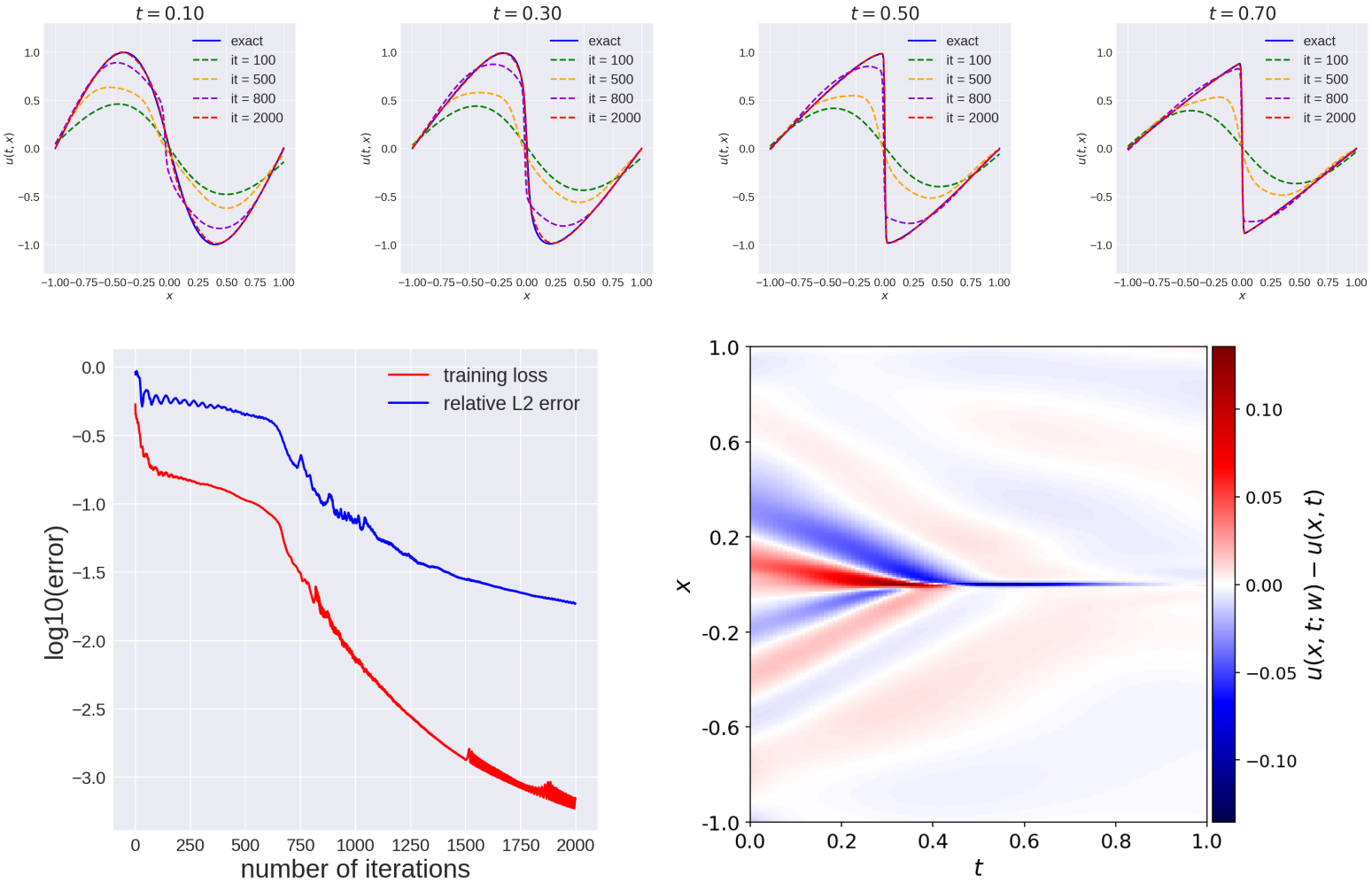

Figure: PINN results for the viscous Burger's equation problem (plots generated by `c12_BurgersPINN.ipynb`).

PINN - Forward Problem

- PINN training can be difficult because it is in fact a *multi-objective optimization* problem.
- On the *Pareto front*, some of the losses can be small, while others can be large. Hence, the loss components may be in conflict with each other.
- To address this, it is common to consider the weighted loss

$$\mathcal{L}(\boldsymbol{w}) = \lambda_r \mathcal{L}_r(\boldsymbol{w}) + \lambda_b \mathcal{L}_b(\boldsymbol{w}),$$

  where the weights λ_r and λ_b are carefully selected to maintain training balance between the losses components.
- This indeed corresponds to a *linear scalarization* of the original multi-objective problem into a single-objective one.

PINN - Forward Problem

- These weights can be fixed or change during training, via a variety of methods. The following is a small smaple of the most well-known methods:
  - Ad-hoc fixed weights (e.g., setting one of the weights to a large value to prioritize that component). (Wight and Zhao, 2000).
  - Change weights during training using gradient information ("learning rate annealing", Wang and Perdikaris, 2020).
  - Change weights according to the eigenvalues of the the empirical Neural Tangent Kernel matrix (Wang et al, 2020).
  - Redistribute the weights according to the loss, putting more weight in the components that do not decrease (Liu and Wang, 2021).

PINN - Forward Problem

- Alternatively, *self-adaptive PINNs* weight each individual point in the loss criteria:

$$\mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) =$$

$$\frac{1}{N_r} \sum_{i=1}^{N_r} g(\lambda_r^i) |F(u(\boldsymbol{x}_r^i; \boldsymbol{w}), \boldsymbol{x}_r^i, Du(\boldsymbol{x}_r^i; \boldsymbol{w}), \ldots, D^k u(\boldsymbol{x}_r^i; \boldsymbol{w}); \lambda) - f(\boldsymbol{x}_r^i)|^2,$$

$$\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) =$$

$$\frac{1}{N_b} \sum_{i=1}^{N_b} g(\lambda_b^i) |G(u(\boldsymbol{x}_b^i; \boldsymbol{w}), \boldsymbol{x}_b^i, Du(\boldsymbol{x}_b^i; \boldsymbol{w}), \ldots, D^m u(\boldsymbol{x}_b^i; \boldsymbol{w})) - g(\boldsymbol{x}_b^i)|^2.$$

where $\boldsymbol{\lambda}_r = (\lambda_r^1, \ldots, \lambda_r^{N_r})$ and $\boldsymbol{\lambda}_b = (\lambda_b^1, \ldots, \lambda_b^{N_b})$ are the vectors of *self-adaptive weights* and the *self-adaptation mask function* $g : [0, \infty) \to R$ is a nonnegative, differentiable, strictly increasing function (a common example is $g(\lambda) = \lambda^2$).

PINN - Forward Problem

- The key feature of self-adaptive PINNs is that the loss

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b) = \mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) + \mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b)$$

  is minimized with respect to the network weights $\boldsymbol{w}$ by gradient descent, as usual, but is *maximized* with respect to the self-adaptative weights $\boldsymbol{\lambda}_r$ and $\boldsymbol{\lambda}_b$ by gradient *ascent*.

- It is easy to check that the gradient with respect to a self-adaptive weight is large where the loss at the corresponding point is large.

- This makes the self-adaptive weights increase at points where the neural network fitting is poor, forcing the error at stubborn points of the solution to decrease.

- This accelerates training and modifies the loss landscape in a way that allows more accurate solutions to be reached.

Discrete-Time PINN

- Consider the time-evolution PDE:

$$
\begin{aligned}
u_t &= \mathcal{N}_{\boldsymbol{x}}[u(\boldsymbol{x},t)]\,, \quad \boldsymbol{x}\in\Omega\,,\ t\in(0,T]\,,\\
u(\boldsymbol{x},t) &= g(\boldsymbol{x},t)\,, \quad \boldsymbol{x}\in\partial\Omega,\ t\in(0,T]\,,\\
u(\boldsymbol{x},0) &= h(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\overline{\Omega}\,.
\end{aligned}
$$

- Here, the domain $\Omega\subset R^d$ in a open set, $\overline{\Omega}$ is its closure, $u:\overline{\Omega}\times[0,T]\to R$ is the desired solution, $\boldsymbol{x}\in\Omega$ is a spatial vector variable, t is time, and $\mathcal{N}_{\boldsymbol{x}}$ is a **spatial** differential operator.

- Given $\boldsymbol{x}\in\Omega$, the basic backward Euler iteration is

$$
u^{i+1}(\boldsymbol{x}) = u^i(\boldsymbol{x}) + h\,\mathcal{N}_{\boldsymbol{x}}[u^{i+1}(\boldsymbol{x})]
$$

  where h is the time step. This can be rewritten as

$$
u^{i+1}(\boldsymbol{x}) - u^i(\boldsymbol{x}) - h\,\mathcal{N}_{\boldsymbol{x}}[u^{i+1}(\boldsymbol{x})] = 0\,.
$$

Euler-PINN

- Hence, at the $(i+1)$-th iteration, an "Euler-PINN" $u^{i+1}(\boldsymbol{x};\boldsymbol{w})$ can be trained by minimizing the composite loss function

$$\mathcal{L}^{i+1}(\boldsymbol{w}) = \mathcal{L}_r^{i+1}(\boldsymbol{w}) + \mathcal{L}_b^{i+1}(\boldsymbol{w})$$

with

$$\mathcal{L}_r^{i+1}(\boldsymbol{w}) = \frac{1}{N_r}\sum_{j=1}^{N_r} |u^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w}) - u^{i,j} - h\mathcal{N}_{\boldsymbol{x}}[u^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w})]|^2,$$

$$\mathcal{L}_b^{i+1}(\boldsymbol{w}) = \frac{1}{N_b}\sum_{j=1}^{N_b} |u^{i+1}(\boldsymbol{x}_b^j, t_b^j;\boldsymbol{w}) - g(\boldsymbol{x}_b^j, t_b^{i+1,i})|^2,$$

where $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r}$ are collocation points, $\{u^{n,i}\}_{i=1}^{N_r}$ is the data at the previous time step (either initial data or the output of the previous step), and $\{\boldsymbol{x}_b^i, t_b^{n,i}\}_{i=1}^{N_b}$ are boundary points.

RK-PINN

- Higher-order time-stepping methods can be easily implemented.
- For example, given $\boldsymbol{x} \in \Omega$, a general (explicit or implicit) Runge-Kutta method with ν stages can be written as

$$\xi_j(\boldsymbol{x}) = u^i(\boldsymbol{x}) + h \sum_{k=1}^{\nu-1} a_{j,k} \mathcal{N}_{\boldsymbol{x}}[\xi_k(\boldsymbol{x})], \quad j = 1, \ldots, \nu$$

$$u^{i+1}(\boldsymbol{x}) = u^i(\boldsymbol{x}) + h \sum_{j=1}^{\nu} b_j \mathcal{N}_{\boldsymbol{x}}[\xi_j(\boldsymbol{x})]$$

- To implement this with a PINN, define a neural network $\mathbf{u}^{i+1}(\boldsymbol{x}; \boldsymbol{w})$ with *multiple outputs* $u_1^{i+1}(\boldsymbol{x}; \boldsymbol{w}), \ldots, u_{\nu+1}^{i+1}(\boldsymbol{x}; \boldsymbol{w})$.
- The first ν output approximate the intermediate steps $\xi_j(\boldsymbol{x})$, while the $(\nu+1)$-th output approximates the RK solution $u^{i+1}(\boldsymbol{x})$ at the next time step.

RK-PINN

- The residual loss to enforce the RK method is

$$\mathcal{L}_r^{i+1}(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1}^{N_r}\sum_{k=1}^{\nu}\left| u_k^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w}) - u^{i,j} - h\sum_{\ell=1}^{\nu-1} a_{k,\ell}\,\mathcal{N}_{\boldsymbol{x}}[u_\ell^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w})]\right|^2$$
$$+ \frac{1}{2}\sum_{j=1}^{N_r}\left| u_{\nu+1}^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w}) - u^{i,j} - h\sum_{k=1}^{\nu} b_k\,\mathcal{N}_{\boldsymbol{x}}[u_k^{i+1}(\boldsymbol{x}_r^j;\boldsymbol{w})]\right|^2$$

  where $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r}$ are collocation points, $\{u^{n,i}\}_{i=1}^{N_r}$ is the data at the previous time step (either initial data or the output of the previous step).

- Notice that this can implement implicit RK methods with a very large number of stages with ease.

PINN - Inverse Problem

- Recall that, given a PDE, the forward problem consists of obtaining $u(\boldsymbol{x})$ from the parameters and boundary conditions.
- In contrast, the *inverse problem* consists of obtaining the value of unknown parameters from noisy data on $u(\boldsymbol{x})$.
- The forward problem must be *well posed*, i.e., the following conditions must be true:
  1. $u(\boldsymbol{x})$ exists;
  2. $u(\boldsymbol{x})$ is unique;
  3. $u(\boldsymbol{x})$ is stable; i.e., it depends continuously on the data (parameters and boundary conditions).
- In contrast, the inverse problem is generally *ill-posed*, i.e., its solution is nonunique and not continuous on the data.
- In addition, the available data in applications are generally sparse and noisy, further complicating the issue.

PINN - Inverse Problem

- A classical example of an ill-posed inverse problem is the *backward diffusion equation*:

$$u_t(x,t) - \kappa u_{xx}(x,t) = 0\,, \quad x \in \Omega \times [0,T]\,,$$
$$\text{s.t.} \;\; u(x,T) = g(x)\,, \quad x \in \Omega\,.$$

- The problem is to determine the value of $u(x,t)$ at time $t = 0$.

- This is an ill-posed problem, as there is an infinite number of initial states that lead to the same final state (this is similar to trying to find the initial shape of an ice cube after it has melted).

- In addition, the solution does not vary continuously with the data: it can be shown that the backward diffusion equation is equivalent to a forward diffusion equation with the final state as the initial condition, but with a *negative* diffusion coefficient, in which case any minute high-frequency noise in the initial condition (i.e., final state) immediately dominates the solution.

PINN - Inverse Problem

- *Regularization* is a process that seeks to cure ill-posedness by conferring existence, uniqueness, and stability — the last two properties being the challenging ones.
- Uniqueness can be enforced by restricting the space of solutions in a suitable manner.
- This is illustrated in the next slide: In the original solution space $\tilde{H}$, the inverse problem is ill-posed due to nonuniqueness, but if one restricts the space to H, the inverse problem is well posed.
- Stability can be enforced by replacing the original unstable problem with a "nearby" stable one; that is, a stable problem that produces a reasonable solution to the original problem.

PINN - Inverse Problem

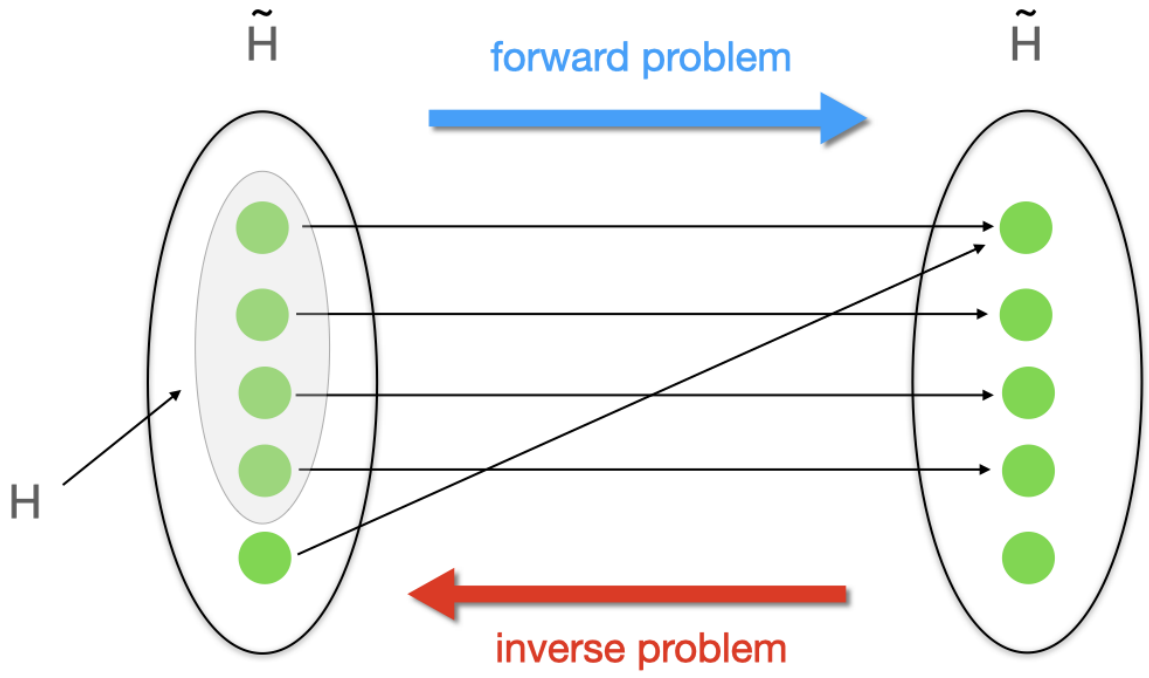

Figure: Regularization by restricting the function space where a problem is defined. In the original solution space $\tilde{H}$, the inverse problem is ill-posed due to nonuniqueness, but in the restricted solution space H, uniqueness is restored and the inverse problem is well-posed.

PINN - Inverse Problem

- PINNs offer a simple and effective solution to inverse problems.
- Consider the problem of obtaining the parameter λ in the PDE

$$F(u(\boldsymbol{x}), \boldsymbol{x}, Du(\boldsymbol{x}), \ldots, D^k u(\boldsymbol{x}); \lambda) = f(\boldsymbol{x}),$$

  from data $\{(\boldsymbol{x}_s^i, u^\delta(\boldsymbol{x}_s^i))\}_{i=1}^{N_s} \subseteq \Omega \times R$, where $u^\delta(\boldsymbol{x}_s^i)$ is a noisy version of $u(\boldsymbol{x}_s^i)$, $i = 1, \ldots, N_s$.
- We will first consider the case where λ is a scalar or vector.
- Let $u(\boldsymbol{x}; \boldsymbol{w})$ be an ansatz deep neural network and let $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r} \subset \Omega$ be the set of collocation points.

PINN - Inverse Problem

- Consider the losses

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{N_s}\sum_{i=1}^{N_s} |u(\boldsymbol{x}_i^s;\boldsymbol{w}) - u^\delta(\boldsymbol{x}_i^s)|\,,$$

$$\mathcal{L}_r(\boldsymbol{w},\lambda) =$$

$$\frac{1}{N_r}\sum_{i=1}^{N_r} |F(u(\boldsymbol{x}_r^i;\boldsymbol{w}), \boldsymbol{x}_r^i, Du(\boldsymbol{x}_r^i;\boldsymbol{w}), \ldots, D^k u(\boldsymbol{x}_r^i;\boldsymbol{w});\lambda) - f(\boldsymbol{x}_r^i)|^2\,.$$

- The neural network weights $\boldsymbol{w}$ and the parameter λ are simultaneously obtained by minimizing the combined loss

$$\mathcal{L}(\boldsymbol{w},\lambda) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w},\lambda)\,,$$

using any of the optimization methods described in Lecture 2.

PINN - Inverse Problem

- When $\lambda : \Omega \to R$ is a function, the previous approach needs to be modified, by introducing a *parameter neural network* $\lambda(\boldsymbol{x}; \boldsymbol{\theta})$, and modifying the residual loss to

$$\mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\theta}) =$$

$$\frac{1}{N_r} \sum_{i=1}^{N_r} |F(u(\boldsymbol{x}_r^i; \boldsymbol{w}), \boldsymbol{x}_r^i, Du(\boldsymbol{x}_r^i; \boldsymbol{w}), \ldots, D^k u(\boldsymbol{x}_r^i; \boldsymbol{w}); \lambda(\boldsymbol{x}_r^i; \boldsymbol{\theta})) - f(\boldsymbol{x}_r^i)|^2$$

- The two neural networks are trained simultaneously by minimizing the combined loss

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{\theta}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\theta}) .$$

- Once trained, the neural network $\lambda(\boldsymbol{x}; \boldsymbol{\theta})$ can be used to compute the estimated value of the parameter over any desired set of points in the domain.

Example: PINN for Burgers Inverse Problem

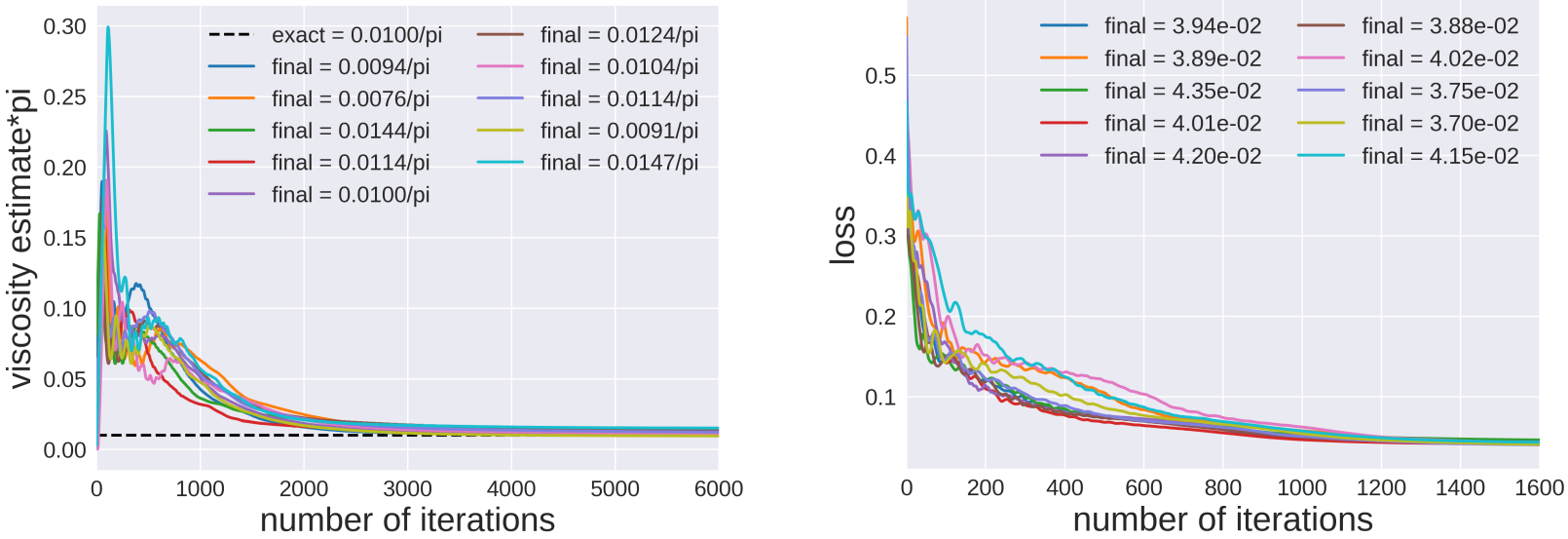

Figure: Inverse problem with a PINN for the viscous Burger's equation. Left: evolution of the viscosity parameter estimate during the first 6000 ADAM iterations for ten independent runs with different randomly sampled training data, collocation points, and randomly initialized neural network weights. The mean parameter estimate at 10,000 ADAM iterations was $(0.0111 \pm 0.0022)/\pi$. Right: evolution of the training loss during the first 1600 ADAM iterations (plots generated by `c12_BurgersPINN_Inverse.ipynb`).

PINN - Hard Constraints

- PINNs use a soft penalty approach via the residual loss function, which does not guarantee that the output of the neural network will indeed be a solution of the PDE.
- We review next a few methods that can include physical constraints directly into the architecture of neural networks, thereby forcing the output to satisfy exactly such constraints.
- The downside of such procedures is that the more constraints that are put in the neural network, the harder it may become to achieve a good local minimum from a random initialization of the weights.

PINN - Hard Constraints

- First, it is possible to make boundary conditions be automatically satisfied. For example, with a Dirichlet boundary condition, this can be accomplished by defining a conditioned neural network:

$$\tilde{u}(\boldsymbol{x}; \boldsymbol{w}) = a(\boldsymbol{x}) u(\boldsymbol{x}; \boldsymbol{w}) + h(\boldsymbol{x}) , \tag{3}$$

  where $u(\boldsymbol{x}; \boldsymbol{w})$ is a regular neural network, $h(\boldsymbol{x})$ satisfies the boundary condition, and $a(\boldsymbol{x})$ is an interpolating function that is zero if and only if $\boldsymbol{x}$ is on the boundary.

- For example, in a time evolution problem with initial condition $u(\boldsymbol{x}, 0) = g(\boldsymbol{x})$, then $\tilde{u}(\boldsymbol{x}, t; \boldsymbol{w}) = u(\boldsymbol{x}, t; \boldsymbol{w}) t + g(x) e^{-t}$ satisfies $\tilde{u}(\boldsymbol{x}, 0; \boldsymbol{w}) = g(x)$, for any value of the weights $\boldsymbol{w}$.

- The conditioned neural network $\tilde{u}(\boldsymbol{x}, t; \boldsymbol{w})$ is trained with the usual physics-informed loss, where derivatives are computed by reverse-mode automatic differentiation.

PINN - Hard Constraints

- A neural network can be constrained to produce values in a set S by adding a "nonlinearity" $\rho : R \to S$ at the output:

$$\tilde{u}(\boldsymbol{x}; \boldsymbol{w}) = \rho(u(\boldsymbol{x}; \boldsymbol{w})),$$

  where $u(\boldsymbol{x}; \boldsymbol{w})$ is a traditional neural network.

- For example, a hyperbolic tangent (tanh) nonlinearity ρ forces the output of $\tilde{u}(\boldsymbol{x}; \boldsymbol{w})$ to be in the interval $[-1, 1]$. As long as ρ is differentiable, $\tilde{u}(\boldsymbol{x}; \boldsymbol{w})$ can be trained as before.

- Alternatively, one can modify the last layer of the network. For example, if the last layer contains nonnegative nonlinearities (e.g., ReLUs) and the output weights are constrained to be nonnegative, then the network output must be nonnegative.

- Adding a differentiable numerical integrator at the output of a nonegative neural network produces a monotone nondecreasing output; this network can be trained end-to-end as usual.

PINN - Hard Constraints

- Odd/even symmetry can be enforced by using the fact that $f(\boldsymbol{x}) + f(-\boldsymbol{x})$ and $f(\boldsymbol{x}) - f(-\boldsymbol{x})$ are even and odd functions, respectively, regardless of the function f.
- For example, the output of the conditioned neural network

$$\tilde{u}(\boldsymbol{x}; \boldsymbol{w}) = u(\boldsymbol{x}; \boldsymbol{w}) + u(-\boldsymbol{x}; \boldsymbol{w}),$$

  is even, regardless of the weights $\boldsymbol{w}$.
- Note that this neural network can be trained as usual, as long as the weights $\boldsymbol{w}$ are "tied" between the two copies of $u(\cdot\,; \boldsymbol{w})$ (i.e., corresponding weights are updated together).
- Symmetry can be imposed only on a subset of the inputs. For example, the output of

$$\tilde{u}(x, t; \boldsymbol{w}) = u(x, t; \boldsymbol{w}) + u(-x, t; \boldsymbol{w})$$

  is even only in x.

PINN - Hard Constraints

- In many physical processes, a vectorial quantity $\mathbf{u}(\boldsymbol{x})$ must satisfy a hard constraint $\mathcal{G}[\mathbf{u}] = 0$, where $\mathcal{G}$ is an operator.
- Suppose that another operator $\mathcal{G}'$ can be found such that $\mathcal{G}[\mathcal{G}'[\mathbf{g}]] = 0$, for every vector field $\mathbf{g}(\boldsymbol{x})$. Then, given a multi-output neural network $\mathbf{u}(\boldsymbol{x}; \boldsymbol{w})$, the neural network

$$\tilde{\mathbf{u}}(\boldsymbol{x}; \boldsymbol{w}) = \mathcal{G}'[\mathbf{u}(\boldsymbol{x}; \boldsymbol{w})]\,.$$

  satisfies the constraint $\mathcal{G}[\tilde{\mathbf{u}}(\boldsymbol{x}; \boldsymbol{w})] = 0$ for any value of $\boldsymbol{w}$.
- As an example, the velocity field $\mathbf{u}(\boldsymbol{x}) = (u(\boldsymbol{x}), v(\boldsymbol{x}), w(\boldsymbol{x}))$ of an incompressible fluid, where $\boldsymbol{x} = (x, y, z) \in R^3$ is the vector of spatial coordinates, must be divergence-free:

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0\,.$$

PINN - Hard Constraints

- One can take advantage here of the vectorial calculus identity

$$\nabla \cdot (\nabla \times \mathbf{g}) = 0\,,$$

where $\nabla \times \mathbf{g}$ is the curl of $\mathbf{g}$, which holds for any (differentiable) vector field $\mathbf{g}(\boldsymbol{x}) \in R^3$.

- Given a 3-output neural network

$$\mathbf{u}(\boldsymbol{x};\boldsymbol{w}) = [u_1(\boldsymbol{x};\boldsymbol{w}_1), u_2(\boldsymbol{x};\boldsymbol{w}_2), u_3(\boldsymbol{x};\boldsymbol{w}_3)]$$

(with $\boldsymbol{w}_1 \cup \boldsymbol{w}_2 \cup \boldsymbol{w}_3 = \boldsymbol{w}$), the 3-output neural network

$$\tilde{\mathbf{u}}(\boldsymbol{x};\boldsymbol{w}) = \nabla \times \mathbf{u}(\boldsymbol{x};\boldsymbol{w}) = \begin{bmatrix} \frac{\partial u_3(\boldsymbol{x};\boldsymbol{w}_3)}{\partial y} - \frac{\partial u_2(\boldsymbol{x};\boldsymbol{w}_2)}{\partial z} \\ \frac{\partial u_1(\boldsymbol{x};\boldsymbol{w}_1)}{\partial z} - \frac{\partial u_3(\boldsymbol{x};\boldsymbol{w}_3)}{\partial x} \\ \frac{\partial u_2(\boldsymbol{x};\boldsymbol{w}_2)}{\partial x} - \frac{\partial u_1(\boldsymbol{x};\boldsymbol{w}_1)}{\partial y} \end{bmatrix}.$$

is automatically divergence-free.

PINN - Hard Constraints

- Instead of modifying the output of the neural network to impose a hard constraint, the input can be modified instead.
- For example, to ensure that periodic boundary conditions are automatically satisfied, one can exploit the fact that if $g(\boldsymbol{x})$ is periodic, then the composition $f(g(\boldsymbol{x}))$ is also periodic, with the same period(s), no matter what f is. The idea is then to let

$$\tilde{u}(\boldsymbol{x}; \boldsymbol{w}) = u(g(\boldsymbol{x}); \boldsymbol{w}) .$$

  where $u(\cdot\,; \boldsymbol{w})$ is an ordinary neural network. The initial layer $g(\boldsymbol{x})$ forces the output of the neural network to be periodic.
- As with symmetry, periodicity can be imposed only on a subset of the inputs. For example, using the periodic function $g(x) = A\cos(2\pi x/L)$, where L is the period, forces the output of

$$\tilde{u}(x, t; \boldsymbol{w}) = u(g(x), t; \boldsymbol{w})$$

  to be periodic only in x, with period L.

PINN - Hard Constraints

- In all the previous approaches, the conditioned neural network $\tilde{u}(\boldsymbol{x}; \boldsymbol{w})$ still must be trained using a soft PDE-residue constraint.
- We next describe a way to impose a hard constraint on PDE satisfaction, but only on a finite set of collocation points, in the case of a linear PDE.
- The output of a neural network with m neurons in its last layer, and no output bias weight, can be written as

$$u(\boldsymbol{x}; \boldsymbol{w}) = \sum_{i=1}^{m} c_i \, \xi_i(\boldsymbol{x}; \boldsymbol{w}_i) \,,$$

where c_i are the output weights, $\xi_i(\boldsymbol{x}; \boldsymbol{w}_i)$ are the outputs of the neurons in the last layer, and $\boldsymbol{w}_i$ are the weights in the computational graph of the ith output neuron, for $i = 1, \ldots, m$.

PINN - Hard Constraints

- Consider a PDE in operator notation:

$$\mathcal{F}[u(\boldsymbol{x})] = f(\boldsymbol{x}),$$

where $\mathcal{F} = F(I, \boldsymbol{x}, D, D^2, \ldots, D^k; \lambda)$.

- If the PDE is linear, then $\mathcal{F}$ is a linear operator, in which case

$$\mathcal{F}[u(\boldsymbol{x}; \boldsymbol{w})] = \sum_{i=1}^{m} c_i \, \mathcal{F}[\xi_i(\boldsymbol{x}; \boldsymbol{w}_i)] = f(\boldsymbol{x}). \tag{4}$$

- Given a set of k collocation points $\{\boldsymbol{x}_r^i\}_{i=1}^m$, one can write the following linear system of equations:

$$\begin{bmatrix} \mathcal{F}[\xi_1(\boldsymbol{x}_r^1; \boldsymbol{w}_1)] & \cdots & \mathcal{F}[\xi_m(\boldsymbol{x}_r^1; \boldsymbol{w}_m)] \\ \vdots & \ddots & \vdots \\ \mathcal{F}[\xi_1(\boldsymbol{x}_r^m; \boldsymbol{w}_1)] & \cdots & \mathcal{F}[\xi_m(\boldsymbol{x}_r^m; \boldsymbol{w}_m)] \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{x}_r^1) \\ \vdots \\ f(\boldsymbol{x}_r^m) \end{bmatrix}.$$

PINN - Hard Constraints

- The solution of the previous system of equations determines the output weights that satisfy (4), so that the PDE is satisfied on the collocation points.
- When training the network, the output weights must be determined anew at every iteration of gradient descent, in order to maintain the hard constraint, which makes this approach computationally expensive if the PDE is to be satisfied at a large number of collocation points. vsp
- Note that the PDE is generally not satisfied outside of the finite set of collocation points.

PINN - Hard Constraints

- For linear hyperbolic PDEs of the kind

$$\frac{\partial u(\boldsymbol{x}, t)}{\partial t} + \mathbf{v}(\boldsymbol{x}, t) \cdot \nabla u(\boldsymbol{x}, t) = 0 \tag{5}$$

  (which includes the linear advection equation), it is possible to guarantee that the PDE is automatically satisfied over the entire domain, not only on a finite set of collocation points, so that nonphysical approximations cannot occur.

- This approach exploits the fact that equation (5) can be solved along *characteristic curves* in the spatial-temporal domain. By definition, the characteristic curve $\boldsymbol{x}(t)$, for $t \geq 0$, satisfies the ODE system

$$\frac{d\boldsymbol{x}(t)}{dt} = \mathbf{v}(\boldsymbol{x}(t), t)\,. \tag{6}$$

PINN - Hard Constraints

- Along a characteristic curve $\boldsymbol{x}(t)$, one has

$$\begin{aligned}\frac{du(\boldsymbol{x}(t),t)}{dt} &= \frac{\partial u(\boldsymbol{x}(t),t)}{\partial t} + \frac{d\boldsymbol{x}(t)}{dt}\cdot\nabla u(\boldsymbol{x}(t),t)\\ &= \frac{\partial u(\boldsymbol{x}(t),t)}{\partial t} + \mathbf{v}(\boldsymbol{x}(t),t)\cdot\nabla u(\boldsymbol{x}(t),t) = 0\,,\end{aligned}$$

  by virtue of (5) and (6).

- Therefore, $u(\boldsymbol{x}(t),t)$ is constant along the characteristic $\boldsymbol{x}(t)$. Now, let a solution of (6) be written in the implicit form

$$\phi(\boldsymbol{x},t) = \boldsymbol{\xi}\,,$$

  where $\boldsymbol{\xi} \in R^d$ is the integration constant of the ODE system.

- The *characteristic transform* $\phi(\boldsymbol{x},t)$ maps the original variables into the characteristic space.

PINN - Hard Constraints

- It can be shown that a solution of (5) must be of the form

$$u(\boldsymbol{x}, t) = f(\phi(\boldsymbol{x}, t)),$$

where $f : R^d \to R$ is a smooth function.

- A *characteristic-informed neural network* (CINN) is given by

$$\tilde{u}(\boldsymbol{x}, t; \boldsymbol{w}) = u(\phi(\boldsymbol{x}, t); \boldsymbol{w}),$$

where $u(\boldsymbol{x}, t; \boldsymbol{w})$ is an ordinary neural network. According to the previous result, $\tilde{u}(\boldsymbol{x}, t; \boldsymbol{w})$ must satisfy the PDE (5) automatically.

- After the characteristic transform layer, the neural network is trained to compute the appropriate value of the solution along the characteristics in order to satisfy any initial/boundary condition data or experimental measurements. No soft penalty terms to enforce the PDE are needed.

PINN - Hard Constraints

- For a simple example, assume that $\mathbf{v}(x,t) \equiv \mathbf{v} = (v_1, v_2, \ldots, v_d)$.
- The characteristic equation (6) in this case reads

$$\frac{d\boldsymbol{x}(t)}{dt} = \mathbf{v}\,,$$

  the solution of which is clearly $\boldsymbol{x}(t) = \mathbf{v}t + \boldsymbol{\xi}$, where $\boldsymbol{\xi} \in R^d$ is the integration constant.

- The characteristic transform in this case is simply

$$\phi(\boldsymbol{x}, t) = \boldsymbol{x} - \mathbf{v}t = \boldsymbol{\xi}\,,$$

  and the general solution of the transport PDE is of the form $u(\boldsymbol{x}, t) = f(\boldsymbol{x} - \mathbf{v}t)$.

PINN - Hard Constraints

- The solution therefore takes constant values on the parallel characteristic lines $\boldsymbol{x} - \mathbf{v}t = \boldsymbol{\xi}$.
- The CINN architecture in this case is depicted on the next slide.
- The characteristic transform layer is a regular neural network layer, except that the weights multiplying the spatial components are always equal to 1.
- The velocity components are the negatives of the weights multiplying the time input.
- In a forward problem, these weights are known and nontrainable, while in an inverse problem, they are trainable, to allow the velocity to be determined from the data.

PINN - Hard Constraints

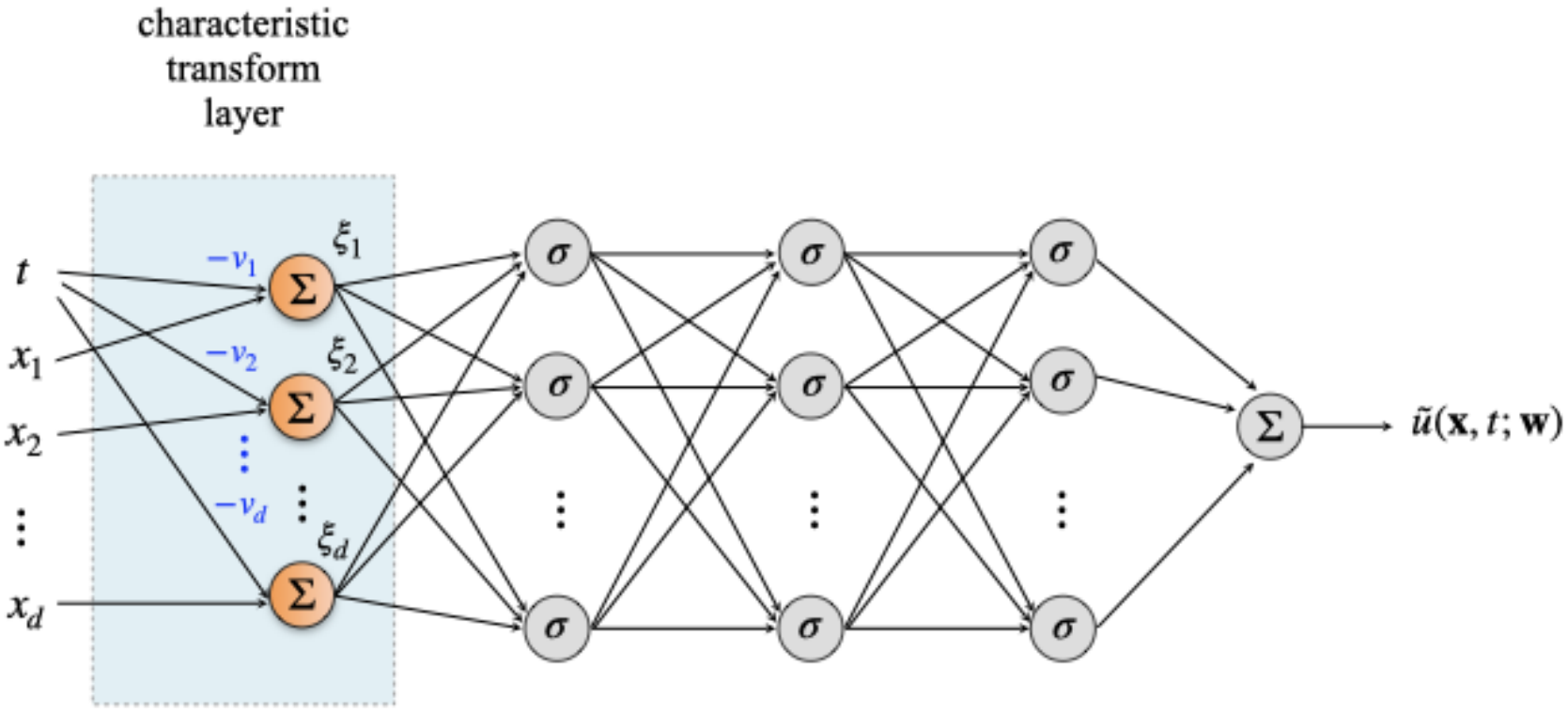

Figure: CINN architecture for the advection equation. The velocity components are weights in the first layer. These weights are fixed in the forward problem, but trainable in the inverse problem.