Neural Networks as Dynamical Systems

Davide Murari

DAMTP, University of Cambridge

dm2011@cam.ac.uk

Code for today's lecture:
davidemurari.com/mphil/code

Plan of the lecture

1. ResNets Based on Dynamical Systems
2. Overview of Structure-Preserving Deep Learning
3. 1-Lipschitz Neural Networks

This lecture is partially based on

Matthias J Ehrhardt, Davide Murari, and Ferdia Sherry. "Stable neural networks and connections to continuous dynamical systems". In: *arXiv preprint arXiv:2510.22299* (2025),

where you can find more details on the experiments and the theory behind them.

ResNets Based on Dynamical Systems

What are Residual Neural Networks (ResNets)?

- One of the building blocks behind several of the modern architectures, such as Transformers, is the so-called **residual layer** or **skip-connection**.
- This building block was introduced for the first time in the ResNet architecture[1].

[1] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

What are Residual Neural Networks (ResNets)?

- One of the building blocks behind several of the modern architectures, such as Transformers, is the so-called **residual layer** or **skip-connection**.
- This building block was introduced for the first time in the ResNet architecture[1].
- The skip-connection amounts to layers of the form

$$\mathbf{x}_{n+1} = F_{\theta_n}(\mathbf{x}_n) = \mathbf{x}_n + \mathcal{F}_{\theta_n}(\mathbf{x}_n),\ \mathcal{F}_{\theta_n} : \mathbb{R}^d \to \mathbb{R}^d,\ \theta_n \in \Theta,$$
$$\text{e.g. } \mathcal{F}_{\theta_n}(\mathbf{x}) = A_n^\top \mathrm{ReLU}(B_n\mathbf{x} + b_n),\ \theta_n = (A_n, B_n, b_n).$$

[1] He et al., "Deep Residual Learning for Image Recognition".

What are Residual Neural Networks (ResNets)?

- One of the building blocks behind several of the modern architectures, such as Transformers, is the so-called **residual layer** or **skip-connection**.
- This building block was introduced for the first time in the ResNet architecture[1].
- The skip-connection amounts to layers of the form

$$\mathbf{x}_{n+1} = F_{\theta_n}(\mathbf{x}_n) = \mathbf{x}_n + \mathcal{F}_{\theta_n}(\mathbf{x}_n),\ \mathcal{F}_{\theta_n} : \mathbb{R}^d \to \mathbb{R}^d,\ \theta_n \in \Theta,$$
$$\text{e.g. } \mathcal{F}_{\theta_n}(\mathbf{x}) = A_n^\top \mathrm{ReLU}(B_n\mathbf{x} + b_n),\ \theta_n = (A_n, B_n, b_n).$$

- ResNets have other types of layers, but the residual ones are their core. The other layers can be used to change the input dimension, which is left unchanged by residual layers.

[1] He et al., "Deep Residual Learning for Image Recognition".

One-Step Numerical Methods for ODEs

- Let us consider a (regular enough) vector field $\mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d$. Fix $\mathbf{x}_0 \in \mathbb{R}^d$. Solving the initial value problem (IVP)

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)), & \text{notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t) \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases}$$

  exactly is in general impossible. We hence have to approximate it numerically.

One-Step Numerical Methods for ODEs

- Let us consider a (regular enough) vector field $\mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d$. Fix $\mathbf{x}_0 \in \mathbb{R}^d$. Solving the initial value problem (IVP)

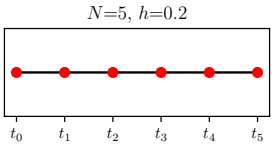
$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)), & \text{notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t) \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases}$$

  exactly is in general impossible. We hence have to approximate it numerically.

- Fix $T > 0$, $N \in \mathbb{N}$, and $h = T/N$. A one-step numerical method $\varphi^h_{\mathcal{F}} : \mathbb{R}^d \to \mathbb{R}^d$ is a map

$$\mathbf{y}_{n+1} = \varphi^h_{\mathcal{F}}(\mathbf{y}_n),\ n = 0, ..., N-1,$$

  such that $\mathbf{y}_0 = \mathbf{x}_0$ and $\mathbf{y}_n \approx \mathbf{x}(nh)$, $n = 1, ..., N$, for any (regular enough) vector field $\mathcal{F}$.



The Explicit Euler Method

- There are several one-step methods. Runge–Kutta methods are a very rich family. In our lecture today, we will mostly need the simplest of them: the explicit Euler method.

The Explicit Euler Method

- There are several one-step methods. Runge–Kutta methods are a very rich family. In our lecture today, we will mostly need the simplest of them: the explicit Euler method.
- For this method, the update map is defined as follows:

$$\mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^{h}(\mathbf{y}_n) := \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_n),\ n = 0, ..., N-1,$$

and it provides a first-order accurate approximation of the exact solution: $\|\mathbf{x}(nh) - \mathbf{x}_n\| \leq C_n h$.

The Explicit Euler Method

- There are several one-step methods. Runge–Kutta methods are a very rich family. In our lecture today, we will mostly need the simplest of them: the explicit Euler method.
- For this method, the update map is defined as follows:

$$\mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^{h}(\mathbf{y}_n) := \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_n),\ n = 0, ..., N-1,$$

  and it provides a first-order accurate approximation of the exact solution: $\|\mathbf{x}(nh) - \mathbf{x}_n\| \leq C_n h$.
- **Example**: Let $\mathcal{F}(\mathbf{x}) = A\mathbf{x}$, for a matrix $A \in \mathbb{R}^{d\times d}$. The exact solution with initial condition $\mathbf{x}(0) = \mathbf{x}_0$ is $\mathbf{x}(t) = \exp(At)\mathbf{x}_0$, whereas the explicit Euler approximation is

$$\mathbf{y}_{n+1} = \mathbf{y}_n + hA\mathbf{y}_n = (I_d + hA)\mathbf{y}_n = (I_d + hA)^{n+1}\mathbf{y}_0,\ n = 0, ..., N-1.$$

ResNet Layers as Explicit Euler Steps

- Why introduce the explicit Euler method in a lecture on neural networks?

ResNet Layers as Explicit Euler Steps

- Why introduce the explicit Euler method in a lecture on neural networks?
- Let's put side by side the definition of a ResNet layer, and the explicit Euler update $\varphi_{\mathcal{F}}^{h}$:

$$\text{ResNet layer : } \mathbf{x}_{n+1} = \mathbf{x}_n + \mathcal{F}_\theta(\mathbf{x}_n), \quad \text{Explicit Euler : } \mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^{h}(\mathbf{y}_n) = \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_n).$$

- We see that if $\mathcal{F}(\mathbf{x}) = \frac{1}{h}\mathcal{F}_\theta(\mathbf{x})$ for every $\mathbf{x} \in \mathbb{R}^d$, then the two maps coincide.

ResNet Layers as Explicit Euler Steps

- Why introduce the explicit Euler method in a lecture on neural networks?
- Let's put side by side the definition of a ResNet layer, and the explicit Euler update $\varphi_{\mathcal{F}}^{h}$:

$$\text{ResNet layer : } \mathbf{x}_{n+1} = \mathbf{x}_n + \mathcal{F}_\theta(\mathbf{x}_n), \quad \text{Explicit Euler : } \mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^{h}(\mathbf{y}_n) = \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_n).$$

- We see that if $\mathcal{F}(\mathbf{x}) = \frac{1}{h}\mathcal{F}_\theta(\mathbf{x})$ for every $\mathbf{x} \in \mathbb{R}^d$, then the two maps coincide.
- **Important remark**: There is no true dynamics behind a ResNet layer. However, we have freedom when designing it and we could hence interpret it as a single Euler step of size 1 applied to the differential equation $\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(\mathbf{x}(t))$.

ResNet Layers as Explicit Euler Steps

- Why introduce the explicit Euler method in a lecture on neural networks?
- Let's put side by side the definition of a ResNet layer, and the explicit Euler update $\varphi_{\mathcal{F}}^{h}$:

$$\text{ResNet layer : } \mathbf{x}_{n+1} = \mathbf{x}_n + \mathcal{F}_\theta(\mathbf{x}_n), \quad \text{Explicit Euler : } \mathbf{y}_{n+1} = \varphi_{\mathcal{F}}^{h}(\mathbf{y}_n) = \mathbf{y}_n + h\mathcal{F}(\mathbf{y}_n).$$

- We see that if $\mathcal{F}(\mathbf{x}) = \frac{1}{h}\mathcal{F}_\theta(\mathbf{x})$ for every $\mathbf{x} \in \mathbb{R}^d$, then the two maps coincide.
- **Important remark**: There is no true dynamics behind a ResNet layer. However, we have freedom when designing it and we could hence interpret it as a single Euler step of size 1 applied to the differential equation $\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(\mathbf{x}(t))$.
- What does this analogy buy us?

ResNets as Discrete Dynamical Systems

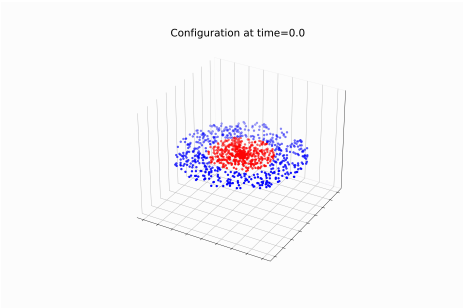
- Having drawn this connection between dynamical systems/ODEs and ResNets, we open up several possibilities:
  1. We are not tied to the use of the explicit Euler method to design the network layers: we could design a suitable parametric family of vector fields $\mathcal{F} = \{\mathcal{F}_\theta : \mathbb{R}^d \to \mathbb{R}^d : \ \theta \in \Theta\}$ and define ResNet-like layers as $\mathbf{x} \mapsto \varphi^h_{\mathcal{F}_\theta}(\mathbf{x})$ where $\varphi^h_{\mathcal{F}}$ is another one-step method (e.g. geometric integrators davidemurari.com/graduateCourseNotes.pdf).
  2. We can look into the theory of numerical analysis, differential equations, and dynamical systems to design new ResNets that behave well, or understand the behaviour of already existing ones.
- In this lecture, we will focus on how to use these ideas to design **stable/robust neural networks** and **symplectic neural networks** (appendix).

A Visual Understanding

- Dataset of two-dimensional points $\{((p_i^1, p_i^2), y_i)\}_{i=1,\dots,N}$. We train a NN to classify them.
- The considered NN is a ResNet based on Euler steps applied to the differential equation

$$\begin{cases} \dot{\mathbf{x}}(t) = B(t)^\top \tanh\left(A(t)\mathbf{x}(t) + \mathbf{b}(t)\right), \ B(t), A(t) \in \mathbb{R}^{3\times 3}, \mathbf{b} \in \mathbb{R}^3, \\ \mathbf{x}(0) = [p_i^1, p_i^2, 0] \in \mathbb{R}^3. \end{cases}$$

- We assume the weight functions $t \mapsto A(t)$, $t \mapsto B(t)$, $t \mapsto \mathbf{b}(t)$ to be piecewise constant.



Overview of Structure-Preserving Deep Learning

What do we mean with structure preservation?

- Sometimes when approximating a target function we are not just looking for an accurate approximation, but we care about interpretability, reliability, and qualitative compatibility with the true function.

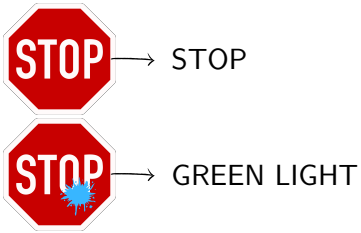


Figure 1: Misclassification of an image that could harm self-driving cars.

What do we mean with structure preservation?

- Sometimes when approximating a target function we are not just looking for an accurate approximation, but we care about interpretability, reliability, and qualitative compatibility with the true function.

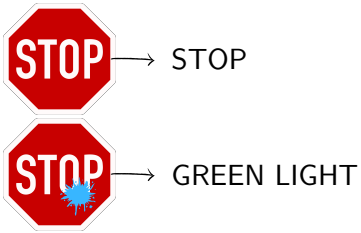
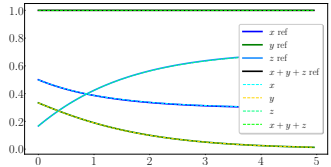


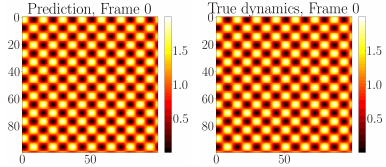
Figure 1: Misclassification of an image that could harm self-driving cars.

- Such properties can be encouraged by losses, but are reliably enforced only when built into the architecture or parametrisation. We call the area of deep learning that focuses on constraining neural networks **structure-preserving deep learning**.

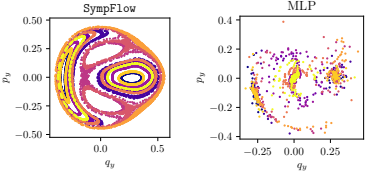
Some learning problems with a structure worth preserving



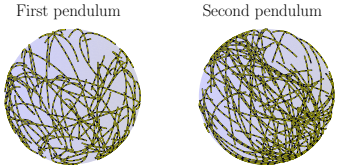
(a) Learning the mass preserving flow map of the SIR model.



(b) Learning the norm-preserving flow map of the linear advection PDE.



(c) Approximating the solutions of unknown Hamiltonian systems.



(d) Approximating the solutions of unknown constrained systems.

Imposing structure over a neural network

- To build networks satisfying a desired property, we can either restrict the parametrisation $\mathcal{N}_\theta$ or modify the loss function.

Imposing structure over a neural network

- To build networks satisfying a desired property, we can either restrict the parametrisation $\mathcal{N}_\theta$ or modify the loss function.

- **Restrict the architecture**:

$$\mathcal{N}_\theta(\mathbf{x}) = \frac{\widetilde{\mathcal{N}}_\theta(\mathbf{x})}{\left\|\widetilde{\mathcal{N}}_\theta(\mathbf{x})\right\|_2} \|\mathbf{x}\|_2 .$$

- **Modify the loss function**:

$$\widetilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \|\mathcal{N}_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2 + \underbrace{\frac{1}{N}\sum_{i=1}^{N} \left(\|\mathbf{x}_i\|_2 - \|\mathcal{N}_\theta(\mathbf{x}_i)\|_2\right)^2}_{\text{regulariser}} .$$

Imposing structure over a neural network

- To build networks satisfying a desired property, we can either restrict the parametrisation $\mathcal{N}_\theta$ or modify the loss function.
- **Restrict the architecture**:

$$\mathcal{N}_\theta(\mathbf{x}) = \frac{\widetilde{\mathcal{N}}_\theta(\mathbf{x})}{\left\|\widetilde{\mathcal{N}}_\theta(\mathbf{x})\right\|_2} \|\mathbf{x}\|_2 .$$

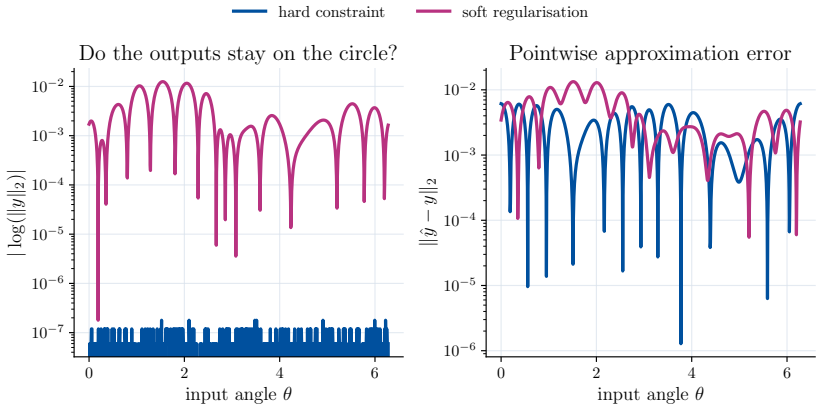
- **Modify the loss function**:

$$\widetilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \|\mathcal{N}_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2 + \underbrace{\frac{1}{N}\sum_{i=1}^{N} \left(\|\mathbf{x}_i\|_2 - \|\mathcal{N}_\theta(\mathbf{x}_i)\|_2\right)^2}_{\text{regulariser}} .$$

- Not all restrictions are equally effective, e.g. $\mathcal{N}_R(\mathbf{x}) = R\mathbf{x}$, $R^\top R = I_d$, is norm-preserving but probably not expressive enough.

A Simple Comparison

$$z(\theta) = \big(\cos(\varphi(\theta)), \sin(\varphi(\theta))\big) \in \mathcal{S}^1, \qquad \varphi(\theta) = \theta + 0.6\sin(2\theta) - 0.15\cos(3\theta).$$



davidemurari.com/mphil/structure

Structured networks based on dynamical systems

RECIPE:

- Choose a property (closed under composition) $\mathcal{P}$ that the network has to satisfy.

Structured networks based on dynamical systems

RECIPE:

- Choose a property (closed under composition) $\mathcal{P}$ that the network has to satisfy.
- Choose a family of parametric vector fields $\mathcal{S}_\Theta$ whose solutions satisfy $\mathcal{P}$.

Structured networks based on dynamical systems

RECIPE:

- Choose a property (closed under composition) $\mathcal{P}$ that the network has to satisfy.
- Choose a family of parametric vector fields $\mathcal{S}_\Theta$ whose solutions satisfy $\mathcal{P}$.
- Choose a numerical method $\Psi^h_{\mathcal{F}_\theta}$, $\mathcal{F}_\theta \in \mathcal{S}_\Theta$, that preserves $\mathcal{P}$ at a discrete level.

The resulting network $\mathcal{N}_\theta = \Psi^{h_L}_{\mathcal{F}_{\theta_L}} \circ \cdots \circ \Psi^{h_1}_{\mathcal{F}_{\theta_1}}$ will preserve $\mathcal{P}$.

1-Lipschitz Neural Networks

Why 1-Lipschitz neural networks? $\|F(\mathbf{y}) - F(\mathbf{x})\|_2 \leq \|\mathbf{y} - \mathbf{x}\|_2$

Adversarial robustness

1-Lipschitz $\implies$ small input perturbations can only produce small output changes

Why 1-Lipschitz neural networks? $\|F(\mathbf{y}) - F(\mathbf{x})\|_2 \leq \|\mathbf{y} - \mathbf{x}\|_2$

Adversarial robustness

1-Lipschitz $\implies$ small input perturbations can only produce small output changes

Wasserstein Generative Adversarial Networks (Kantorovich-Rubinstein duality)

$$W_1(\mu, \nu) = \sup_{\substack{f:\mathcal{X}\to\mathbb{R} \\ f\ 1-\text{Lipschitz}}} \mathbb{E}_{X\sim\mu}[f(X)] - \mathbb{E}_{Y\sim\nu}[f(Y)].$$

Why 1-Lipschitz neural networks? $\|F(\mathbf{y}) - F(\mathbf{x})\|_2 \leq \|\mathbf{y} - \mathbf{x}\|_2$

Adversarial robustness

1-Lipschitz $\implies$ small input perturbations can only produce small output changes

Wasserstein Generative Adversarial Networks (Kantorovich-Rubinstein duality)

$$W_1(\mu, \nu) = \sup_{\substack{f:\mathcal{X}\to\mathbb{R} \\ f \text{ 1-Lipschitz}}} \mathbb{E}_{X\sim\mu}[f(X)] - \mathbb{E}_{Y\sim\nu}[f(Y)].$$

Convergent fixed point iterations

$T_\alpha(\mathbf{x}) = (1-\alpha)\mathbf{x} + \alpha F(\mathbf{x})$, $\alpha \in (0,1)$, F 1-Lipschitz $\implies$ $\mathbf{x}_{k+1} = T_\alpha(\mathbf{x}_k) \to \mathrm{Fix}(F)$.

1-Lipschitz MLPs

- Given two Lipschitz-continuous functions $F : \mathbb{R}^h \to \mathbb{R}^c$, $G : \mathbb{R}^d \to \mathbb{R}^h$, with Lipschitz constants $\mathrm{Lip}(F)$ and $\mathrm{Lip}(G)$, respectively, the composition $H = F \circ G : \mathbb{R}^d \to \mathbb{R}^c$ is Lipschitz continuous as well, with $\mathrm{Lip}(H) \leq \mathrm{Lip}(F)\mathrm{Lip}(G)$:

$$\begin{aligned} \|H(\mathbf{y}) - H(\mathbf{x})\|_2 &= \|F(G(\mathbf{y})) - F(G(\mathbf{x}))\|_2 \leq \mathrm{Lip}(F)\|G(\mathbf{y}) - G(\mathbf{x})\|_2 \\ &\leq \mathrm{Lip}(F)\mathrm{Lip}(G)\|\mathbf{y} - \mathbf{x}\|_2, \ \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d. \end{aligned}$$

1-Lipschitz MLPs

- Given two Lipschitz-continuous functions $F : \mathbb{R}^h \to \mathbb{R}^c$, $G : \mathbb{R}^d \to \mathbb{R}^h$, with Lipschitz constants $\mathrm{Lip}(F)$ and $\mathrm{Lip}(G)$, respectively, the composition $H = F \circ G : \mathbb{R}^d \to \mathbb{R}^c$ is Lipschitz continuous as well, with $\mathrm{Lip}(H) \leq \mathrm{Lip}(F)\mathrm{Lip}(G)$:

$$\begin{aligned}\|H(\mathbf{y}) - H(\mathbf{x})\|_2 = \|F(G(\mathbf{y})) - F(G(\mathbf{x}))\|_2 &\leq \mathrm{Lip}(F)\|G(\mathbf{y}) - G(\mathbf{x})\|_2 \\ &\leq \mathrm{Lip}(F)\mathrm{Lip}(G)\|\mathbf{y} - \mathbf{x}\|_2, \ \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d.\end{aligned}$$

- We can get a 1-Lipschitz feedforward network (MLP) composing 1-Lipschitz layers:

$$\mathcal{N}_\theta = A_L \circ \sigma \circ A_{L-1} \circ ... \circ \sigma \circ A_1 : \mathbb{R}^d \to \mathbb{R}^c,$$

where we need $|\sigma(s) - \sigma(t)| \leq |s - t|$, and $\|A_i\|_2 \leq 1$ for $j = 1, ..., L$. Most activation functions, such as $\tanh, \mathrm{ReLU}, \mathrm{LeakyReLU}$, sigmoid, sin are 1-Lipschitz.

1-Lipschitz ResNets are more challenging to obtain

For ResNets, it is more challenging, since the basic layers are of the form

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \mathbf{x} + \tau \mathcal{F}_{\theta_i}(\mathbf{x}) = \varphi^\tau_{\theta_i}(\mathbf{x}) \in \mathbb{R}^d, \ \tau > 0,$$

and, for a generic $\mathcal{F}_{\theta_i} : \mathbb{R}^d \to \mathbb{R}^d$, it is hard to get better bounds than

$$\|\varphi^\tau_{\theta_i}(\mathbf{y}) - \varphi^\tau_{\theta_i}(\mathbf{x})\|_2 \leq (1 + \tau \mathrm{Lip}(\mathcal{F}_{\theta_i})) \, \|\mathbf{y} - \mathbf{x}\|_2, \ \mathbf{x}, \mathbf{y} \in \mathbb{R}^d.$$

We hence need to modify them slightly, or properly choose the residual map $\mathcal{F}_{\theta_i}$.

Negative gradient flows

Let $V : \mathbb{R}^d \to \mathbb{R}$ be a continuously differentiable convex function. We consider vector fields of the form

$$\mathcal{F}(\mathbf{x}) = -\nabla V(\mathbf{x}).$$

Given two solution curves, $\dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t))$ and $\dot{\mathbf{y}}(t) = \mathcal{F}(\mathbf{y}(t))$, we see that

$$\frac{d}{dt}\frac{1}{2}\|\mathbf{x}(t) - \mathbf{y}(t)\|_2^2 = -\left(\nabla V(\mathbf{x}(t)) - \nabla V(\mathbf{y}(t))\right)^\top (\mathbf{x}(t) - \mathbf{y}(t)) \leq 0.$$

Thus, the flow map $\phi_{\mathcal{F}}^t : \mathbb{R}^d \to \mathbb{R}^d$ defined by $\phi_{\mathcal{F}}^t(\mathbf{x}(0)) = \mathbf{x}(t)$ is 1-Lipschitz.

Non-expansive gradient flows

Gradient flows on $\mathbb{R}^d$

Consider the convex scalar function[a] $V_\theta(x) = \|\mathrm{ReLU}(W\mathbf{x} + \mathbf{b})\|_2^2/2$. Define

$$\mathcal{F}_\theta(\mathbf{x}) = -\nabla V_\theta(\mathbf{x}) = -W^\top \mathrm{ReLU}(W\mathbf{x} + \mathbf{b}).$$

[a] $W \in \mathbb{R}^{h\times d}$, $\mathbf{b} \in \mathbb{R}^h$, $h \in \mathbb{N}$, $\theta = (W, \mathbf{b})$.

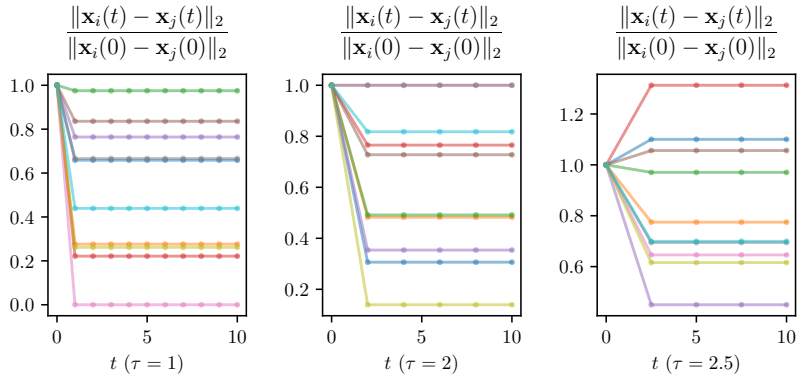
Euler step (1-Lipschitz)

If $\tau \in [0, 2/\|W\|_2^2]$, the explicit Euler map $\varphi_\theta^\tau(\mathbf{x}) = \mathbf{x} + \tau\mathcal{F}_\theta(\mathbf{x})$ is 1-Lipschitz, i.e.,

$$\|\varphi_\theta^\tau(\mathbf{y}) - \varphi_\theta^\tau(\mathbf{x})\|_2 \leq \|\mathbf{y} - \mathbf{x}\|_2, \ \mathbf{x}, \mathbf{y} \in \mathbb{R}^d.$$

ODEs with 1-Lipschitz solution and Euler maps

$$\dot{\mathbf{x}}(t) = -W^\top \mathrm{ReLU}(W\mathbf{x}(t)),\ W = \frac{1}{2}\begin{bmatrix}\sqrt{2} & -\sqrt{2}\\ \sqrt{2} & \sqrt{2}\end{bmatrix},\ \mathrm{ReLU}(s) = \max\{s, 0\}.$$



Neural networks based on gradient flows

We consider neural networks of the form

$$\mathcal{N}_\theta = \pi \circ \varphi_{\theta_L} \circ ... \circ \varphi_{\theta_1} \circ Q : \mathbb{R}^d \to \mathbb{R}^c,\ \varphi_{\theta_\ell} \in \mathcal{E}_h,$$

Neural networks based on gradient flows

We consider neural networks of the form

$$\mathcal{N}_\theta = \pi \circ \varphi_{\theta_L} \circ ... \circ \varphi_{\theta_1} \circ Q : \mathbb{R}^d \to \mathbb{R}^c,\ \varphi_{\theta_\ell} \in \mathcal{E}_h,$$

$$\mathcal{E}_h := \Big\{ \varphi : \mathbb{R}^h \to \mathbb{R}^h \ \Big|\ \varphi(\mathbf{x}) = x - \tau W^\top \mathrm{ReLU}(W\mathbf{x} + \mathbf{b}),\ W \in \mathbb{R}^{h' \times h}, \mathbf{b} \in \mathbb{R}^{h'},$$
$$h' \in \mathbb{N}, \tau \in [0, 2/\|W\|_2^2] \Big\},$$

where $Q : \mathbb{R}^d \to \mathbb{R}^h$ and $\pi : \mathbb{R}^h \to \mathbb{R}^c$ are 1-Lipschitz affine maps.

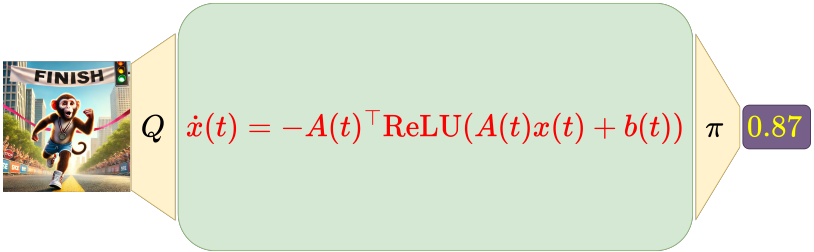
Neural networks based on gradient flows

We consider neural networks of the form

$$\mathcal{N}_\theta = \pi \circ \varphi_{\theta_L} \circ \ldots \circ \varphi_{\theta_1} \circ Q : \mathbb{R}^d \to \mathbb{R}^c,\ \varphi_{\theta_\ell} \in \mathcal{E}_h,$$

$$\mathcal{E}_h := \Big\{ \varphi : \mathbb{R}^h \to \mathbb{R}^h \ \Big|\ \varphi(\mathbf{x}) = x - \tau W^\top \mathrm{ReLU}(W\mathbf{x} + \mathbf{b}),\ W \in \mathbb{R}^{h' \times h}, \mathbf{b} \in \mathbb{R}^{h'},$$
$$h' \in \mathbb{N}, \tau \in [0, 2/\|W\|_2^2] \Big\},$$

where $Q : \mathbb{R}^d \to \mathbb{R}^h$ and $\pi : \mathbb{R}^h \to \mathbb{R}^c$ are 1-Lipschitz affine maps.



Robust Classification with 1-Lipschitz Networks

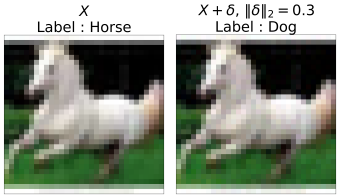
The problem of robust classification

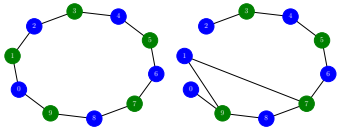
- We train a classifier that maps an input $\mathbf{x}$ to one of a finite number of classes.
- A robust classifier should not change its prediction when $\mathbf{x}$ is modified by a tiny, semantically irrelevant perturbation.

The problem of robust classification

- We train a classifier that maps an input $\mathbf{x}$ to one of a finite number of classes.
- A robust classifier should not change its prediction when $\mathbf{x}$ is modified by a tiny, semantically irrelevant perturbation.

Adversarial examples





How to have guaranteed robustness

- Not all correct predictions are equally safe.
- Suppose the correct label is $\ell(\mathbf{x}) = 2$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ barely wins, so a tiny perturbation may flip the class.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ has a much larger safety gap.

How to have guaranteed robustness

- Not all correct predictions are equally safe.
- Suppose the correct label is $\ell(\mathbf{x}) = 2$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ barely wins, so a tiny perturbation may flip the class.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ has a much larger safety gap.

Margin: $\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) := \mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_{\ell(\mathbf{x})} - \max_{j \neq \ell(\mathbf{x})} \mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_j.$

$$\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) > 0 \implies \mathcal{N}_\theta \text{ correctly classifies } \mathbf{x}.$$

$$\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) > \sqrt{2}\mathrm{Lip}(\mathcal{N}_\theta)\varepsilon \implies \mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x} + \boldsymbol{\eta}) > 0 \, \forall \|\boldsymbol{\eta}\|_2 \leq \varepsilon.$$

How to have guaranteed robustness

- Not all correct predictions are equally safe.
- Suppose the correct label is $\ell(\mathbf{x}) = 2$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ barely wins, so a tiny perturbation may flip the class.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ has a much larger safety gap.

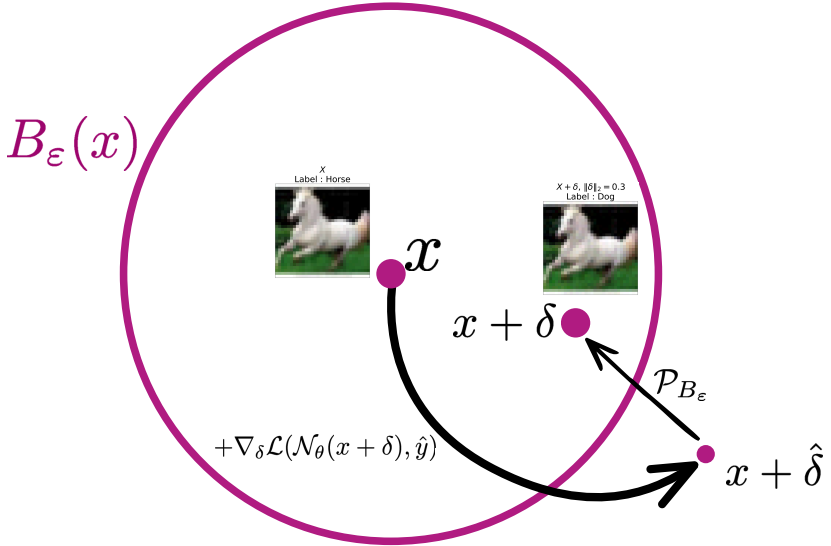
$$\textbf{Margin: } \mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) := \mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_{\ell(\mathbf{x})} - \max_{j \neq \ell(\mathbf{x})} \mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_j.$$

$$\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) > 0 \implies \mathcal{N}_\theta \text{ correctly classifies } \mathbf{x}.$$

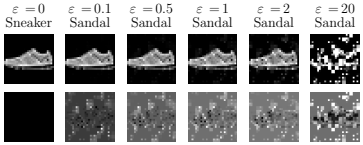
$$\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x}) > \sqrt{2}\mathrm{Lip}(\mathcal{N}_\theta)\varepsilon \implies \mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x} + \boldsymbol{\eta}) > 0 \, \forall \|\boldsymbol{\eta}\|_2 \leq \varepsilon.$$

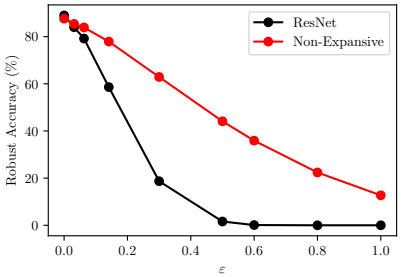
- Smaller $\mathrm{Lip}(\mathcal{N}_\theta)$ and larger margin $\mathcal{M}_{\mathcal{N}_\theta}(\mathbf{x})$ both enlarge the certified robust radius ε.

How we build the adversarial attacks: Projected Gradient Descent (PGD)



Robustness to adversarial attacks





davidemurari.com/mphil/classification

Some extensions

The dynamical-systems lens beyond ResNets

- ResNets are the simplest discrete-dynamics example, but the same viewpoint appears in many other architectures.
- It can be used to think about:
  - **continuous-depth models**: neural ODEs and flow-based models,
  - **sequence models**: RNNs and neural CDEs,
  - **graph neural networks**: diffusion- and PDE-inspired dynamics,
  - **transformers**: residual dynamics and transport viewpoints.

Further reading

Continuous-depth models

- E, Weinan, "'A Proposal on Machine Learning via Dynamical Systems'", *Communications in Mathematics and Statistics*, 2017.
- Chen, Ricky T. Q. et al., "'Neural Ordinary Differential Equations'", *NeurIPS*, 2018.

RNNs / sequence models

- Chang, Bo et al., "'AntisymmetricRNN: A Dynamical System View on Recurrent Neural Networks'", *ICLR*, 2019.

Stability-aware architectures

- Haber, Eldad and Lars Ruthotto, "'Stable Architectures for Deep Neural Networks'", *Inverse Problems*, 2017.
- Ruthotto, Lars and Eldad Haber, "'Deep Neural Networks Motivated by Partial Differential Equations'", *Journal of Mathematical Imaging and Vision*, 2020.

Graph neural networks

- Chamberlain, Benjamin Paul et al., "'GRAND: Graph Neural Diffusion'", *ICML*, 2021.
- Eliasof, Moshe, Eldad Haber, and Eran Treister, "'PDE-GCN: Novel architectures for graph neural networks motivated by partial differential equations'", *NeurIPS*, 2021.

Transformers

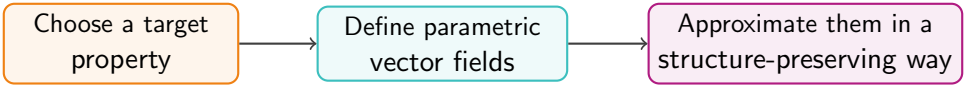
- Castin, Victor et al., "'A unified perspective on the dynamics of deep transformers'", *arXiv preprint arXiv:2501.18322*, 2025.
- Geshkovski, Borjan et al., "'A Mathematical Perspective on Transformers'", *Bulletin of the American Mathematical Society*, 2025.

Take-home message

- We need structure when accuracy alone is not enough and we also care about robustness, stability, or qualitative correctness.

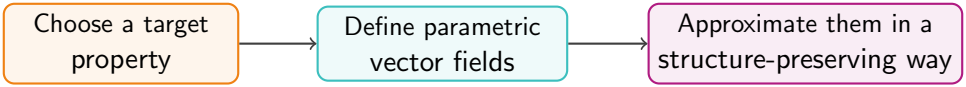
Take-home message

- We need structure when accuracy alone is not enough and we also care about robustness, stability, or qualitative correctness.



Take-home message

- We need structure when accuracy alone is not enough and we also care about robustness, stability, or qualitative correctness.



Examples

1-Lip $\Longrightarrow$ negative gradient flows $\Longrightarrow$ Euler with restricted step size

Symplectic $\Longrightarrow$ Hamiltonian systems $\Longrightarrow$ symplectic Euler *(appendix)*

THANK YOU FOR THE ATTENTION!

Any questions?

APPENDIX

1-Lipschitz Networks for Inverse Problems

The Proximal Gradient Descent Method

$$\min_{\mathbf{x}\in\mathbb{R}^d} \left(f(\mathbf{x}) + \gamma g(\mathbf{x})\right), \quad f : \mathbb{R}^d \to \mathbb{R},\ g : \mathbb{R}^d \to \mathbb{R} \cup \{\pm\infty\}, \tag{1}$$

where f is a data-fidelity term, g is a regularisation term, and $\gamma > 0$.

The Proximal Gradient Descent Method

$$\min_{\mathbf{x}\in\mathbb{R}^d} \left(f(\mathbf{x}) + \gamma g(\mathbf{x})\right), \; f : \mathbb{R}^d \to \mathbb{R}, \; g : \mathbb{R}^d \to \mathbb{R} \cup \{\pm\infty\}, \tag{1}$$

where f is a data-fidelity term, g is a regularisation term, and $\gamma > 0$.

Example:

$$f(\mathbf{x}) = \frac{1}{2}\|K\mathbf{x} - \mathbf{y}\|_2^2, \; g(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|_2^2 \quad \text{(Ridge Regression)}.$$

The Proximal Gradient Descent Method

$$\min_{\mathbf{x}\in\mathbb{R}^d} \left(f(\mathbf{x}) + \gamma g(\mathbf{x})\right), \ f : \mathbb{R}^d \to \mathbb{R}, \ g : \mathbb{R}^d \to \mathbb{R} \cup \{\pm\infty\}, \tag{1}$$

where f is a data-fidelity term, g is a regularisation term, and $\gamma > 0$.

Example:

$$f(\mathbf{x}) = \frac{1}{2}\|K\mathbf{x} - \mathbf{y}\|_2^2, \ g(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|_2^2 \quad \text{(Ridge Regression)}.$$

Assume $f : \mathbb{R}^d \to \mathbb{R}$ and $g : \mathbb{R}^d \to \mathbb{R} \cup \{\pm\infty\}$ convex, f continuously differentiable, g continuous and proper. A method to solve (1) is the Proximal Gradient Descent Method:

$$\mathbf{x}_{k+1} = \operatorname{prox}_{\gamma g,\tau}\left(\mathbf{x}_k - \tau\nabla f(\mathbf{x}_k)\right), \ \tau > 0,$$

$$\operatorname{prox}_{\gamma g,\tau}(\mathbf{x}) = \arg\min_{\mathbf{z}\in\mathbb{R}^d}\left(\frac{1}{2\tau}\|\mathbf{x} - \mathbf{z}\|_2^2 + \gamma g(\mathbf{z})\right).$$

Example: Projected Gradient Descent

$\Omega \subset \mathbb{R}^d$ non-empty, closed, convex set. $f : \mathbb{R}^d \to \mathbb{R}$ convex and continuously differentiable.

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}) \iff \min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) + i_\Omega(\mathbf{x}), \; i_\Omega(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} \in \Omega, \\ +\infty, & \mathbf{x} \notin \Omega. \end{cases}$$

Example: Projected Gradient Descent

$\Omega \subset \mathbb{R}^d$ non-empty, closed, convex set. $f : \mathbb{R}^d \to \mathbb{R}$ convex and continuously differentiable.

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}) \iff \min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) + i_\Omega(\mathbf{x}), \; i_\Omega(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} \in \Omega, \\ +\infty, & \mathbf{x} \notin \Omega. \end{cases}$$

Here, we have that if $i_\Omega =: g$, the proximal operator is an orthogonal projection operator:

$$\mathrm{prox}_{\gamma g,\tau}(\mathbf{x}) = \arg\min_{\mathbf{z}\in\mathbb{R}^d} \left(\frac{1}{2\tau}\|\mathbf{x}-\mathbf{z}\|_2^2 + \gamma i_\Omega(\mathbf{z}) \right) = \arg\min_{\mathbf{z}\in\Omega} \|\mathbf{x}-\mathbf{z}\|_2^2 = \mathrm{proj}_\Omega(\mathbf{x}).$$

Example: Projected Gradient Descent

$\Omega \subset \mathbb{R}^d$ non-empty, closed, convex set. $f : \mathbb{R}^d \to \mathbb{R}$ convex and continuously differentiable.

$$\min_{\mathbf{x}\in\Omega} f(\mathbf{x}) \iff \min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) + i_\Omega(\mathbf{x}), \; i_\Omega(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} \in \Omega, \\ +\infty, & \mathbf{x} \notin \Omega. \end{cases}$$

Here, we have that if $i_\Omega =: g$, the proximal operator is an orthogonal projection operator:

$$\mathrm{prox}_{\gamma g,\tau}(\mathbf{x}) = \arg\min_{\mathbf{z}\in\mathbb{R}^d} \left(\frac{1}{2\tau}\|\mathbf{x}-\mathbf{z}\|_2^2 + \gamma i_\Omega(\mathbf{z}) \right) = \arg\min_{\mathbf{z}\in\Omega} \|\mathbf{x}-\mathbf{z}\|_2^2 = \mathrm{proj}_\Omega(\mathbf{x}).$$

The proximal gradient method then becomes the projected gradient descent method:

$$\mathbf{x}_{k+1} = \mathrm{proj}_\Omega(\mathbf{x}_k - \tau \nabla f(\mathbf{x}_k)).$$

Example: ISTA (cfr. SINDy)

Let $f(\mathbf{x}) = \frac{1}{2}\|K\mathbf{x} - \mathbf{y}\|_2^2$, $g(\mathbf{x}) = \|\mathbf{x}\|_1 = \sum_{i=1}^{d} |x_i|$, and $\gamma > 0$ the regularisation parameter.

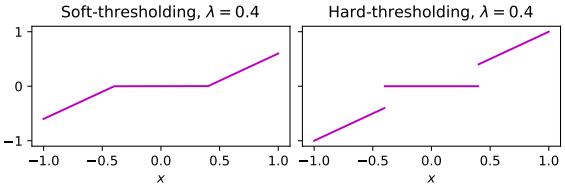
Example: ISTA (cfr. SINDy)

Let $f(\mathbf{x}) = \frac{1}{2}\|K\mathbf{x} - \mathbf{y}\|_2^2$, $g(\mathbf{x}) = \|\mathbf{x}\|_1 = \sum_{i=1}^{d} |x_i|$, and $\gamma > 0$ the regularisation parameter.

The Proximal Gradient Descent then writes

$$\mathbf{x}_{k+1} = \mathrm{prox}_{\gamma g, \tau}\left(\mathbf{x}_k - \tau K^\top (K\mathbf{x} - \mathbf{y})\right) = S_{\tau\gamma}\left(\mathbf{x}_k - \tau K^\top (K\mathbf{x} - \mathbf{y})\right),$$

$$(S_\lambda(\mathbf{x}))_i = \begin{cases} x_i - \lambda, & x_i > \lambda, \\ 0, & |x_i| \leq \lambda, \\ x_i + \lambda, & x_i < -\lambda, \end{cases} \quad \lambda > 0,\ i = 1, ..., d.$$



The Plug-and-Play (PnP) Method

There are two problems with what we saw in the previous slides:

1. It is very hard to define a **good regulariser** for any given task,
2. The **proximal operator** of a generic regulariser g is **not easy to compute**.

The Plug-and-Play (PnP) Method

There are two problems with what we saw in the previous slides:

1. It is very hard to define a **good regulariser** for any given task,
2. The **proximal operator** of a generic regulariser g is **not easy to compute**.

Solution: The Plug-and-Play method is defined by replacing $\text{prox}_{\gamma g,\alpha}$ with a neural network:

$$\text{Plug-and-Play:} \quad \mathbf{x}_{k+1} = \mathcal{N}_\theta(\mathbf{x}_k - \tau \nabla f(\mathbf{x}_k)), \ \mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^d. \tag{2}$$

The network $\mathcal{N}_\theta$ is typically trained offline to denoise images:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \|\mathcal{N}_\theta(\mathbf{x}_i + \delta_i) - \mathbf{x}_i\|_2^2, \quad \delta_1, ..., \delta_N \sim \mathcal{D}.$$

The Plug-and-Play (PnP) Method

There are two problems with what we saw in the previous slides:

1. It is very hard to define a **good regulariser** for any given task,
2. The **proximal operator** of a generic regulariser g is **not easy to compute**.

Solution: The Plug-and-Play method is defined by replacing $\text{prox}_{\gamma g,\alpha}$ with a neural network:

$$\text{Plug-and-Play:} \quad \mathbf{x}_{k+1} = \mathcal{N}_\theta(\mathbf{x}_k - \tau \nabla f(\mathbf{x}_k)), \ \mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^d. \tag{2}$$

The network $\mathcal{N}_\theta$ is typically trained offline to denoise images:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \|\mathcal{N}_\theta(\mathbf{x}_i + \delta_i) - \mathbf{x}_i\|_2^2, \quad \delta_1, ..., \delta_N \sim \mathcal{D}.$$

Convergence guarantees

Assume f is μ-strongly convex, L-smooth, and $\tau \in (0, 2/L)$. Then if $\mathcal{N}_\theta$ is 1-Lipschitz, the iterates of the PnP method converge to a unique fixed point of the PnP operator.

The Network $\mathcal{N}_\theta$ Trained as Denoiser[2]

$$\Gamma_{\text{Euler}} := \mathcal{P} \circ \mathcal{N}_\theta \circ \mathcal{L},$$

$$\mathcal{L}(x_1, x_2, x_3) = (x_1, x_2, x_3, 0, ..., 0) \in \mathbb{R}^{64}, \ \mathcal{P}(x_1, ..., x_{64}) = (x_1, x_2, x_3) \in \mathbb{R}^3.$$

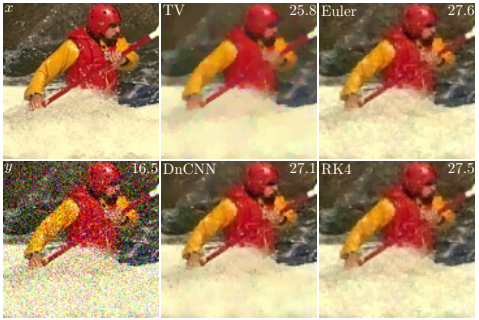


Figure 3: Image from BSDS500 dataset, composed of 500 natural colour images of size 321×481.

[2] Ferdia Sherry et al. "Designing stable neural networks using convex analysis and odes". In: *Physica D: Nonlinear Phenomena* 463 (2024), p. 134159.

Stability of the trained denoisers

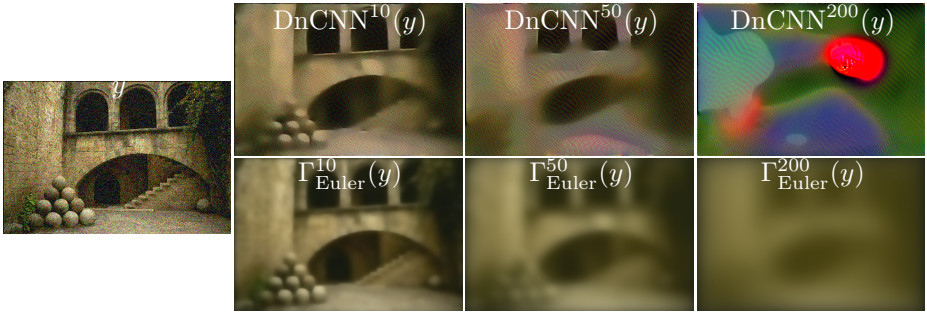


Figure 4: Repeated application of the unconstrained denoiser DnCNN^3 and the constrained denoiser Γ_{Euler} to a given input image.

[3] Kai Zhang et al. "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising". In: *IEEE transactions on image processing* 26.7 (2017), pp. 3142–3155.

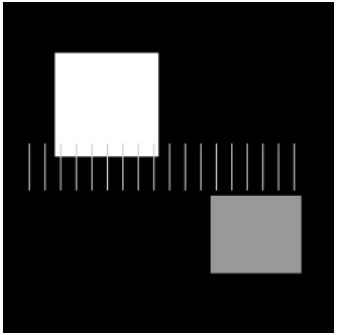
What do we mean with Deblurring?

- Let us consider the **inverse problem of deblurring**: we assume that we are given measurements $y = Kx + \varepsilon$, where $K\mathbf{x} = k * \mathbf{x}$ is a convolution operation representing a motion blur.
- The ill-posedness of this problem is manifested in the instability of the inverse of the convolution; as a consequence of this, a naive inversion of the measurements will blow up the noise in the measurements.
- The data-fidelity term is

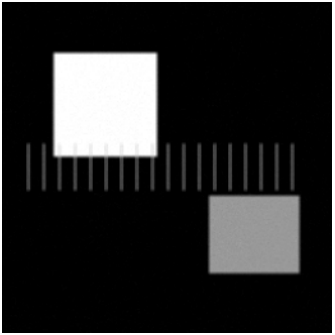
$$f(\mathbf{x}) = \frac{1}{2}\|K\mathbf{x} - \mathbf{y}\|_2^2.$$

Visualisation of the ill-posedness

Original, x



Blur + noise, $y = Kx + \varepsilon$



Naive inverse $K^{-1}y = \hat{x}$

Use in a Deblurring Task

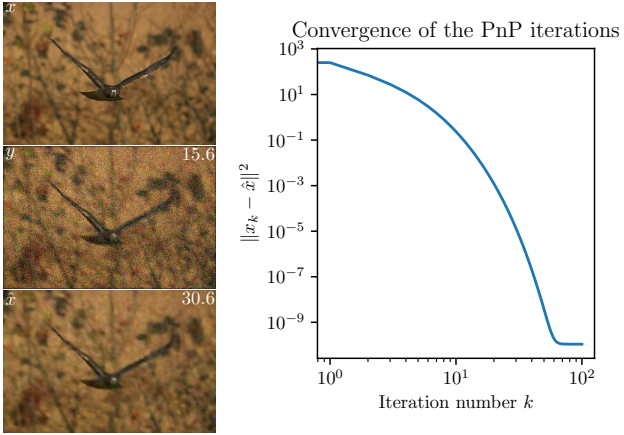


Figure 5: Using the learned Euler denoiser to solve an ill-posed inverse problem (deblurring) in a PnP fashion, with convergence guarantee.

Relaxed convergence theory for Plug-and-Play

Averaged maps

α-averaged map

The map $T : \mathbb{R}^d \to \mathbb{R}^d$ is averaged if there exists $\alpha \in (0, 1)$ and a 1-Lipschitz map $F : \mathbb{R}^d \to \mathbb{R}^d$ such that $T = (1 - \alpha)\mathrm{id} + \alpha F$. The composition of averaged maps is again averaged. Patrick L Combettes and Isao Yamada. "Compositions and Convex Combinations of Averaged Nonexpansive Operators". In: *Journal of Mathematical Analysis and Applications* 425.1 (2015), pp. 55–70, Proposition 2.4

Let $f : \mathbb{R}^d \to \mathbb{R}$ be convex, continuously-differentiable, and L-smooth. Then if $\tau \in (0, 2/L)$ the map $T(\mathbf{x}) = \mathbf{x} - \tau \nabla f(\mathbf{x})$ is averaged with $\alpha = \tau L/2$.

Convergence under convexity

Convergence Theorem

Let $f : \mathbb{R}^d \to \mathbb{R}$ be continuously differentiable, convex, and L-smooth. Assume $\tau \in (0, 2/L)$. Then $G = \mathrm{id} - \tau \nabla f$ is $\tau L/2$ averaged. Further assume that $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^d$ is averaged. Let $T = \mathcal{N}_\theta \circ G$. Assuming that $\mathrm{Fix}(T) \neq \emptyset$, the Plug-and-Play iterates $\mathbf{x}_{k+1} = T(\mathbf{x}_k)$ will converge to a fixed point.

Convergence under convexity

Convergence Theorem

Let $f : \mathbb{R}^d \to \mathbb{R}$ be continuously differentiable, convex, and L-smooth. Assume $\tau \in (0, 2/L)$. Then $G = \mathrm{id} - \tau \nabla f$ is $\tau L/2$ averaged. Further assume that $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^d$ is averaged. Let $T = \mathcal{N}_\theta \circ G$. Assuming that $\mathrm{Fix}(T) \neq \emptyset$, the Plug-and-Play iterates $\mathbf{x}_{k+1} = T(\mathbf{x}_k)$ will converge to a fixed point.

Our networks are explicit Euler steps for the gradient of $f(x) = 1^\top \mathrm{ReLU}^2(Ax + b)/2$, which is convex and its gradient is

$$\nabla f(x) = A^\top \mathrm{ReLU}(Ax + b),$$

which is $\|A\|_2^2$-Lipschitz. This means that the layers of our 1-Lipschitz network are averaged if $0 < \tau_i < 2/\|A_i\|_2^2$, and hence so is the full network $\mathcal{N}_\theta$.

Hamiltonian/Symplectic Neural Networks

What is a Canonical Hamiltonian System?

- The equations of motion of canonical Hamiltonian systems write

$$\begin{cases} \dot{\mathbf{x}} = \mathbb{J}\nabla H(\mathbf{x}) = X_H(\mathbf{x}) \in \mathbb{R}^{2n} \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases}, \quad \mathbb{J} = \begin{bmatrix} 0_n & I_n \\ -I_n & 0_n \end{bmatrix} \in \mathbb{R}^{2n \times 2n}.$$

What is a Canonical Hamiltonian System?

- The equations of motion of canonical Hamiltonian systems write

$$\begin{cases} \dot{\mathbf{x}} = \mathbb{J}\nabla H(\mathbf{x}) = X_H(\mathbf{x}) \in \mathbb{R}^{2n} \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases}, \quad \mathbb{J} = \begin{bmatrix} 0_n & I_n \\ -I_n & 0_n \end{bmatrix} \in \mathbb{R}^{2n \times 2n}.$$

- Denoted with $\phi_{H,t} : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ the exact flow, $\phi_{H,t}(\mathbf{x}_0) = \mathbf{x}(t)$, we see that

$$\frac{d}{dt} H(\phi_{H,t}(\mathbf{x}_0)) = \nabla H(\phi_{H,t}(\mathbf{x}_0))^\top \mathbb{J} \nabla H(\phi_{H,t}(\mathbf{x}_0)) = 0,$$

  which means that the Hamiltonian is constant along the solutions.

The Symplecticity Condition

- A continuously differentiable function $F : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if

$$\left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right)^{\top} \mathbb{J} \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right) = \mathbb{J}. \tag{3}$$

- The composition of symplectic maps is symplectic (chain rule + (3)).
- The flow map $\phi_{H,t}$ of a Hamiltonian system is symplectic for every t.

Why do we care about symplectic networks?

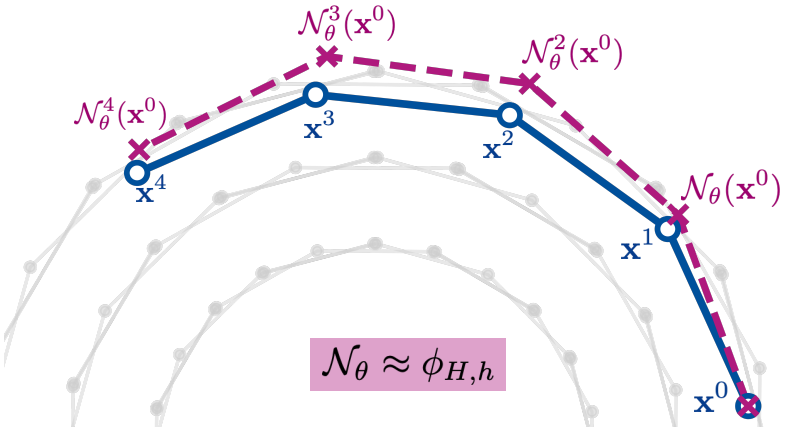
There are two main reasons why symplectic/Hamiltonian networks are studied:

1. **To approximate the solutions of (unknown) Hamiltonian systems**.
   - Pengzhan Jin et al. "SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems". In: *Neural Networks* 132 (2020), pp. 166–179
   - Priscilla Canizares et al. "Symplectic neural flows for modeling and discovery". In: *SIAM Journal on Scientific Computing (in print)* (2024)

2. **For gradient stability**, (see appendix slides),
   - Clara Lucía Galimberti et al. "Hamiltonian deep neural networks guaranteeing nonvanishing gradients by design". In: *IEEE Transactions on Automatic Control* 68.5 (2023), pp. 3155–3162

Approximating the solution map of Hamiltonian systems

Definition of the problem



$$\mathcal{L}(\theta) = \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} \left\| \left(\mathcal{N}_\theta\right)^j \left(\mathbf{x}_i^0\right) - \mathbf{x}_i^j \right\|_2^2 \to \min .$$

Hamiltonian NNs (HNNs) / Symplectic NNs

IDEA[4]

Compose exact flows of Hamiltonian systems with Hamiltonian functions

$$H^1_\theta(\mathbf{q}, \mathbf{p}) = K_\theta(\mathbf{p}), \;\; H^2_\theta(\mathbf{q}, \mathbf{p}) = U_\theta(\mathbf{q}).$$

[4] Pengzhan Jin et al. "SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems". In: *Neural Networks* 132 (2020), pp. 166–179.

Hamiltonian NNs (HNNs) / Symplectic NNs

IDEA[4]

Compose exact flows of Hamiltonian systems with Hamiltonian functions

$$H^1_\theta(\mathbf{q},\mathbf{p}) = K_\theta(\mathbf{p}),\ H^2_\theta(\mathbf{q},\mathbf{p}) = U_\theta(\mathbf{q}).$$

The ODEs they define are

$$\begin{bmatrix}\dot{\mathbf{q}}\\ \dot{\mathbf{p}}\end{bmatrix} = \begin{bmatrix}\nabla K_\theta(\mathbf{p})\\ 0\end{bmatrix},\ \begin{bmatrix}\dot{\mathbf{q}}\\ \dot{\mathbf{p}}\end{bmatrix} = \begin{bmatrix}0\\ -\nabla U_\theta(\mathbf{q})\end{bmatrix},$$

and they have solutions

$$\phi_{H^1_\theta,t}(\mathbf{q},\mathbf{p}) = \begin{bmatrix}\mathbf{q} + t\nabla K_\theta(\mathbf{p})\\ \mathbf{p}\end{bmatrix},\ \phi_{H^2_\theta,t}(\mathbf{q},\mathbf{p}) = \begin{bmatrix}\mathbf{q}\\ \mathbf{p} - t\nabla U_\theta(\mathbf{q})\end{bmatrix}.$$

[4] Jin et al., "SympNets: Intrinsic structure-preserving symplectic networks for identifying Hamiltonian systems".

Example: Gradient Modules-type SympNets

- Set

$$K_\theta(\mathbf{p}) = \mathbf{u}^\top \gamma(A\mathbf{p} + \mathbf{a}),\ U_\theta(\mathbf{q}) = \mathbf{v}^\top \gamma(B\mathbf{q} + \mathbf{b}),$$

  so that

$$\nabla K_\theta(\mathbf{p}) = A^\top \mathrm{diag}(\mathbf{u})\sigma(A\mathbf{p} + \mathbf{a}),\ \nabla U_\theta(\mathbf{q}) = B^\top \mathrm{diag}(\mathbf{v})\sigma(B\mathbf{q} + \mathbf{b}),\ \sigma = \gamma'.$$

Example: Gradient Modules-type SympNets

- Set

$$K_\theta(\mathbf{p}) = \mathbf{u}^\top \gamma(A\mathbf{p} + \mathbf{a}),\ U_\theta(\mathbf{q}) = \mathbf{v}^\top \gamma(B\mathbf{q} + \mathbf{b}),$$

  so that

$$\nabla K_\theta(\mathbf{p}) = A^\top \mathrm{diag}(\mathbf{u})\sigma(A\mathbf{p} + \mathbf{a}),\ \nabla U_\theta(\mathbf{q}) = B^\top \mathrm{diag}(\mathbf{v})\sigma(B\mathbf{q} + \mathbf{b}),\ \sigma = \gamma'.$$

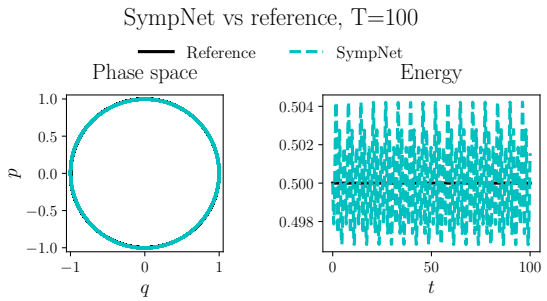
- The Symplectic/Hamiltonian NN that we obtain then has layers of the form

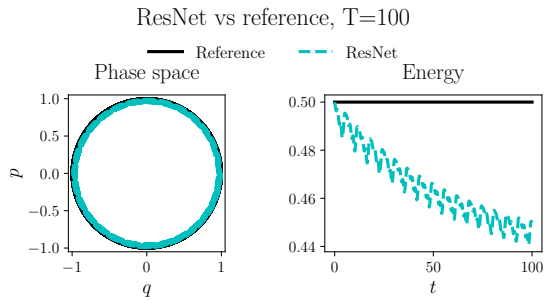
$$F_{\theta_{2i}}(\mathbf{q}, \mathbf{p}) = \begin{bmatrix} \mathbf{q} + h_{2i} A_i^\top \mathrm{diag}(\mathbf{u}_i)\sigma(A_i\mathbf{p} + \mathbf{a}_i) \\ \mathbf{p} \end{bmatrix},\ h_{2i}, h_{2i+1} \in \mathbb{R},$$

$$F_{\theta_{2i+1}}(\mathbf{q}, \mathbf{p}) = \begin{bmatrix} \mathbf{q} \\ \mathbf{p} - h_{2i+1} B_i^\top \mathrm{diag}(\mathbf{v}_i)\sigma(B_i\mathbf{q} + \mathbf{b}_i) \end{bmatrix},\ i = 1, ..., L.$$

Numerical experiment: Simple Harmonic Oscillator

$$\dot{q} = p, \;\; \dot{p} = -q, \;\; H(q,p) = \frac{1}{2}(p^2 + q^2) \text{ (energy)}.$$





SympNet preserves the qualitative structure over long horizons much better than a generic ResNet.

davidemurari.com/mphil/sympnet

Gradient stability of Hamiltonian networks

How do we train neural networks?

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

How do we train neural networks?

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D},\ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...),\ k = 1, ..., \text{epochs}.$$

How do we train neural networks?

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D},\ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...),\ k = 1, ..., \text{epochs}.$$

- When dealing with neural network training, we generally need to solve very high-dimensional problems, since $\theta \in \mathbb{R}^p$ with p usually large. For example, GPT-1 has 117 million parameters.

How do we train neural networks?

- To find the network weights θ, we need to minimise a loss function $\mathcal{L}(\theta)$. This cannot generally be done analytically, and we therefore need a numerical method to approximate such a solution.

- Most of these methods are iterative, in the sense that they start from a hopefully good initial guess θ_0, and define an iteration that aims to improve on it until a stopping criterion is met:

$$\theta_0 \sim \mathcal{D},\ \theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k), ...),\ k = 1, ..., \text{epochs}.$$

- When dealing with neural network training, we generally need to solve very high-dimensional problems, since $\theta \in \mathbb{R}^p$ with p usually large. For example, GPT-1 has 117 million parameters.

- This implies that we need to use first-order algorithms, i.e., methods where T only depends on the gradient of $\mathcal{L}$, and not on its higher-order derivatives.

The gradient descent algorithm

$$\theta_{k+1} = T(\theta_k, \nabla\mathcal{L}(\theta_k)) := \theta_k - \tau\nabla\mathcal{L}(\theta_k)$$

Backpropagation: how do we compute the gradients?

Let us focus on a data point $(\mathbf{x}_n, \mathbf{y}_n) \in \mathbb{R}^d \times \mathbb{R}^c$, and consider the network $\mathcal{N}_\theta = F_{\theta_L} \circ \cdots \circ F_{\theta_1}$. Define

$$\mathbf{x}^1 = \mathbf{x}_n, \qquad \mathbf{x}^{j+1} = F_{\theta_j}(\mathbf{x}^j),\ j = 1, \dots, L, \qquad \widehat{\mathbf{y}}_n := \mathbf{x}^{L+1}.$$

Backpropagation: how do we compute the gradients?

Let us focus on a data point $(\mathbf{x}_n, \mathbf{y}_n) \in \mathbb{R}^d \times \mathbb{R}^c$, and consider the network $\mathcal{N}_\theta = F_{\theta_L} \circ \cdots \circ F_{\theta_1}$. Define

$$\mathbf{x}^1 = \mathbf{x}_n, \qquad \mathbf{x}^{j+1} = F_{\theta_j}(\mathbf{x}^j),\ j = 1, \dots, L, \qquad \widehat{\mathbf{y}}_n := \mathbf{x}^{L+1}.$$

Define $\mathcal{L}_n := \|\widehat{\mathbf{y}}_n - \mathbf{y}_n\|_2^2/2$. Assume for simplicity that all the weights $\theta_1, ..., \theta_L$ are vectors. Set

$$\mathbf{g}^{L+1} := \nabla_{\mathbf{x}^{L+1}} \mathcal{L}_n = \mathbf{x}^{L+1} - \mathbf{y}_n, \qquad \mathbf{g}^j := \nabla_{\mathbf{x}^j} \mathcal{L}_n = \left(J_{\mathbf{x}^j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \mathbf{g}^{j+1}, \quad j = L, \dots, 1,$$

where J denotes a Jacobian.

Backpropagation: how do we compute the gradients?

Let us focus on a data point $(\mathbf{x}_n, \mathbf{y}_n) \in \mathbb{R}^d \times \mathbb{R}^c$, and consider the network $\mathcal{N}_\theta = F_{\theta_L} \circ \cdots \circ F_{\theta_1}$. Define

$$\mathbf{x}^1 = \mathbf{x}_n, \qquad \mathbf{x}^{j+1} = F_{\theta_j}(\mathbf{x}^j),\ j = 1, \dots, L, \qquad \widehat{\mathbf{y}}_n := \mathbf{x}^{L+1}.$$

Define $\mathcal{L}_n := \|\widehat{\mathbf{y}}_n - \mathbf{y}_n\|_2^2/2$. Assume for simplicity that all the weights $\theta_1, ..., \theta_L$ are vectors. Set

$$\mathbf{g}^{L+1} := \nabla_{\mathbf{x}^{L+1}} \mathcal{L}_n = \mathbf{x}^{L+1} - \mathbf{y}_n, \qquad \mathbf{g}^j := \nabla_{\mathbf{x}^j} \mathcal{L}_n = \left(J_{\mathbf{x}^j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \mathbf{g}^{j+1}, \quad j = L, \dots, 1,$$

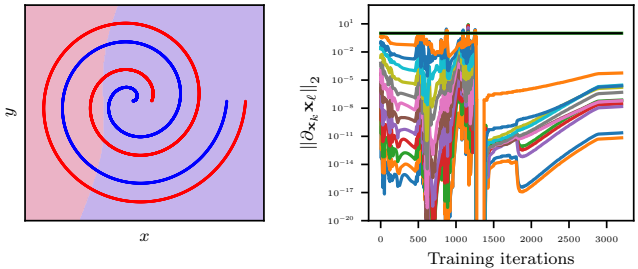
where J denotes a Jacobian.

Gradients (per sample)

$$\nabla_{\theta_j} \mathcal{L}_n = \left(J_{\theta_j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \nabla_{\mathbf{x}^j} \mathcal{L}_n = \left(J_{\theta_j} F_{\theta_j}(\mathbf{x}^j)\right)^\top \mathbf{g}^{j+1}, \qquad j = L, \dots, 1.$$

Thus, the Backpropagation algorithm is just the chain rule organised to reuse Jacobian–vector products.

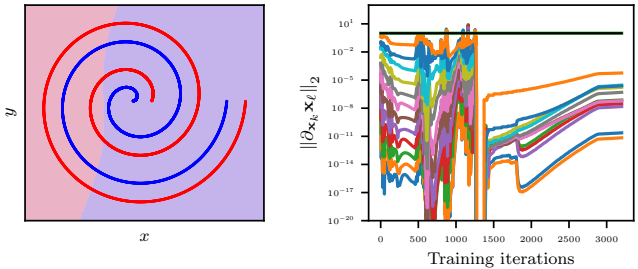
Vanishing gradients



By repeated application of the chain rule, we can see that

$$\|\nabla_{\theta_j}\mathcal{L}_n\|_2 \;\leq\; \|J_{\theta_j}F_{\theta_j}(\mathbf{x}^j)\|_2 \left(\prod_{\ell=j+1}^{L}\left\|J_{\mathbf{x}^\ell}F_{\theta_\ell}(\mathbf{x}^\ell)\right\|_2\right) \|\nabla_{\mathbf{x}^{L+1}}\mathcal{L}_n\|$$

Vanishing gradients



By repeated application of the chain rule, we can see that

$$\|\nabla_{\theta_j}\mathcal{L}_n\|_2 \;\leq\; \|J_{\theta_j}F_{\theta_j}(\mathbf{x}^j)\|_2 \left(\prod_{\ell=j+1}^{L}\left\|J_{\mathbf{x}^\ell}F_{\theta_\ell}(\mathbf{x}^\ell)\right\|_2\right) \|\nabla_{\mathbf{x}^{L+1}}\mathcal{L}_n\|$$

If $\|J_{\mathbf{x}}F_{\theta_\ell}(\mathbf{x})\|_2 \leq \rho < 1$ (e.g. $\mathrm{Lip}(\sigma) \leq 1$ and $\|A_\ell\|_2 \leq \rho$), then $\|\nabla_{\theta_j}\mathcal{L}_n\|_2 \lesssim \rho^{L-j}$
$\Rightarrow$ **vanishing gradients**, and we can not meaningfully update the weights.

Why do symplectic networks improve gradient stability?

Let $F : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ be a continuously differentiable symplectic map, i.e.,

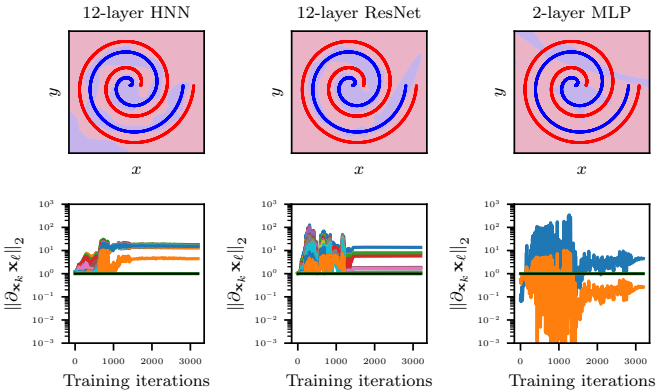
$$\left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \mathbb{J} \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right) = \mathbb{J}.$$

Then we have

$$\begin{aligned} \|\mathbb{J}\|_2 &= \left\| \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \mathbb{J} \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right) \right\|_2 \leq \left\| \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \right\|_2 \|\mathbb{J}\|_2 \left\| \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} \right\|_2 \\ &= \|\mathbb{J}\|_2 \left\| \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} \right\|_2^2 \implies \left\| \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} \right\|_2 \geq 1. \end{aligned}$$

Thus F would likely not contribute to vanishing gradients!

Gradient Stability in HNNs[5]



[5]HNN stands for Hamiltonian Neural Network or Symplectic Neural Network. See more details of this experiment in Matthias J Ehrhardt, Davide Murari, and Ferdia Sherry. "Stable neural networks and connections to continuous dynamical systems". In: *arXiv preprint arXiv:2510.22299* (2025).