Neural Networks as Dynamical Systems

Davide Murari

Department of Applied Mathematics and Theoretical Physics
University of Cambridge

davidemurari.com/cism

dm2011@cam.ac.uk

Outline

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**.

[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].

[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

$$\mathbf{x}_{n+1} = F_{\theta_n}(\mathbf{x}_n) = \mathbf{x}_n + \mathcal{F}_{\theta_n}(\mathbf{x}_n),\ \mathcal{F}_{\theta_n} : \mathbb{R}^d \to \mathbb{R}^d,\ \theta_n \in \Theta.$$

[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

$$\mathbf{x}_{n+1} = F_{\theta_n}(\mathbf{x}_n) = \mathbf{x}_n + \mathcal{F}_{\theta_n}(\mathbf{x}_n),\ \mathcal{F}_{\theta_n} : \mathbb{R}^d \to \mathbb{R}^d,\ \theta_n \in \Theta.$$

- The term residual refers to the map $\mathcal{F}_{\theta_n} = F_{\theta_n} - \mathrm{id}$; we parametrise F_{θ_n} by parametrising the residual.
- 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.

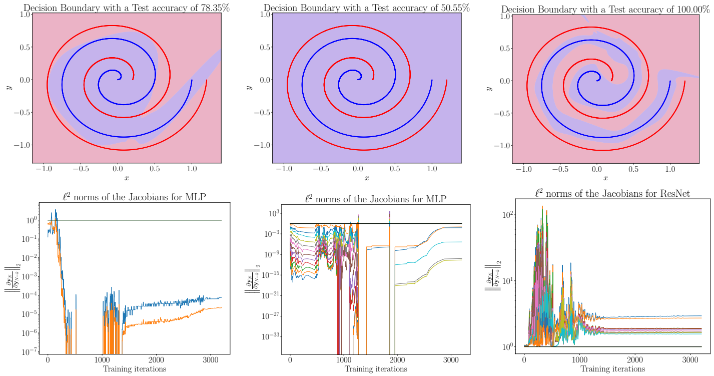
[1] He et al., "Deep Residual Learning for Image Recognition".

Why ResNets?

Recall that to minimise the loss function $\mathcal{L}(\theta)$ we have to use some numerical method, such as gradient descent

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

If $\|\nabla \mathcal{L}(\theta_k)\|_2$ is very large or very small, we will struggle to find a meaningful set of weights.



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)

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

  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)

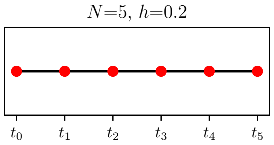
$$\begin{cases} \dot{\mathbf{x}}(t) = \mathcal{F}(\mathbf{x}(t)), \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases} \qquad \text{notation: } \dot{\mathbf{x}}(t) = \frac{d\mathbf{x}}{dt}(t)$$

  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 $\varphi_{\mathcal{F}}^h : \mathbb{R}^d \to \mathbb{R}^d$ is a map

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

  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 of them. In our lectures 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 of them. In our lectures we will mostly need the simplest of them: the explicit Euler method.
- For this method, the update map is defined as follows:

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

  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 of them. In our lectures we will mostly need the simplest of them: the explicit Euler method.

- For this method, the update map is defined as follows:

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

  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

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

ResNet Layers as Explicit Euler Steps

- Why introducing the explicit Euler method in a lecture on Neural Networks?

ResNet Layers as Explicit Euler Steps

- Why introducing 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 introducing 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 one applied to the differential equation $\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(\mathbf{x}(t))$.

ResNet Layers as Explicit Euler Steps

- Why introducing 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 one 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

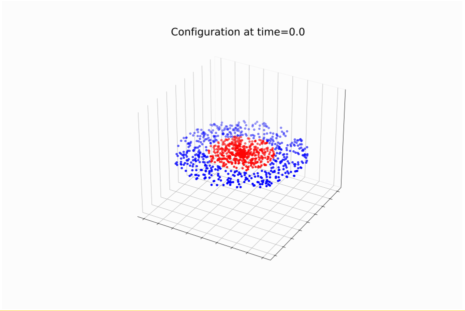
- Having drawn this connection between dynamical systems/ODEs and ResNets, we open up several possibilities:
  1. We are not tied to the use of the explicit Euler method to design the network layers: we could design a suitable parametric family of vector fields $\mathcal{F} = \{\mathcal{F}_\theta : \mathbb{R}^d \to \mathbb{R}^d : \ \theta \in \Theta\}$ 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 solely focus on the second point.

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

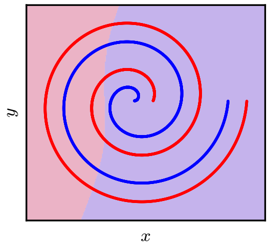
$$\begin{cases} \dot{\mathbf{x}}(t) = B(t)^\top \tanh\left(A(t)\mathbf{x}(t) + \mathbf{b}(t)\right), \ B(t), A(t) \in \mathbb{R}^{3\times 3},\ \mathbf{b} \in \mathbb{R}^3, \\ \mathbf{x}(0) = [p_i^1, p_i^2, 0] \in \mathbb{R}^3. \end{cases}$$

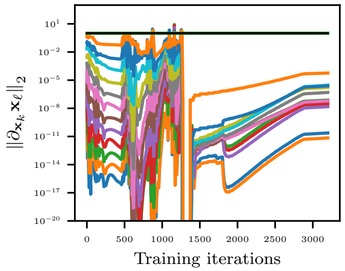
- We assume the weight functions $t \mapsto A(t)$, $t \mapsto B(t)$, $t \mapsto \mathbf{b}(t)$ to be piecewise constant.



Hamiltonian Neural Networks

Recap on the Vanishing Gradient Problem





We have seen that the gradient of the loss function with respect to the network weights satisfies

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

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

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 linear map $F(\mathbf{x}) = A\mathbf{x}$, $A \in \mathbb{R}^{2n\times 2n}$, is symplectic if the matrix A satisfies
$$A^\top \mathbb{J} A = \mathbb{J}.$$
Equivalently, it means that the map F preserves the bilinear form
$$\Omega(\mathbf{u},\mathbf{v}) := \mathbf{u}^\top \mathbb{J}\mathbf{v} \iff \Omega(A\mathbf{u}, A\mathbf{v}) = \Omega(\mathbf{u},\mathbf{v}),\ \forall \mathbf{u},\mathbf{v} \in \mathbb{R}^{2n}.$$

The Symplecticity Condition

- A linear map $F(\mathbf{x}) = A\mathbf{x}$, $A \in \mathbb{R}^{2n\times 2n}$, is symplectic if the matrix A satisfies

$$A^\top \mathbb{J} A = \mathbb{J}.$$

  Equivalently, it means that the map F preserves the bilinear form

$$\Omega(\mathbf{u},\mathbf{v}) := \mathbf{u}^\top \mathbb{J}\mathbf{v} \iff \Omega(A\mathbf{u}, A\mathbf{v}) = \Omega(\mathbf{u},\mathbf{v}),\ \forall \mathbf{u},\mathbf{v} \in \mathbb{R}^{2n}.$$

- A (non-linear) continuously differentiable function $F : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if it infinitesimally preserves Ω, i.e. $\partial_{\mathbf{x}} F(\mathbf{x}) \in \mathbb{R}^{2n\times 2n}$ is symplectic for every $\mathbf{x}$:

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

The Symplecticity Condition

- A linear map $F(\mathbf{x}) = A\mathbf{x}$, $A \in \mathbb{R}^{2n\times 2n}$, is symplectic if the matrix A satisfies
$$A^\top \mathbb{J} A = \mathbb{J}.$$
Equivalently, it means that the map F preserves the bilinear form
$$\Omega(\mathbf{u},\mathbf{v}) := \mathbf{u}^\top \mathbb{J}\mathbf{v} \iff \Omega(A\mathbf{u}, A\mathbf{v}) = \Omega(\mathbf{u},\mathbf{v}),\ \forall \mathbf{u},\mathbf{v} \in \mathbb{R}^{2n}.$$

- A (non-linear) continuously differentiable function $F : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if it infinitesimally preserves Ω, i.e. $\partial_{\mathbf{x}} F(\mathbf{x}) \in \mathbb{R}^{2n\times 2n}$ is symplectic for every $\mathbf{x}$:
$$\left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right)^\top \mathbb{J} \left(\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}}\right) = \mathbb{J}.$$

- **Exercise:** Show that the composition of continuously differentiable symplectic maps is symplectic.

Why do we care about symplectic maps?

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 not contribute to vanishing gradients!

Hamiltonian NNs (HNNs) / Symplectic NN

- A Hamiltonian or Symplectic NN is a network which is symplectic. This typically means that all its layers are symplectic maps.

Hamiltonian NNs (HNNs) / Symplectic NN

- A Hamiltonian or Symplectic NN is a network which is symplectic. This typically means that all its layers are symplectic maps.

- A common way to define them is by composing 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

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

  Exercise: Show that $\phi_{H^1_\theta,t}$ and $\phi_{H^2_\theta,t}$ are symplectic maps.

Example of a Hamiltonian/Symplectic NN

- A common strategy is to 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 of a Hamiltonian/Symplectic NN

- A common strategy is to set

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

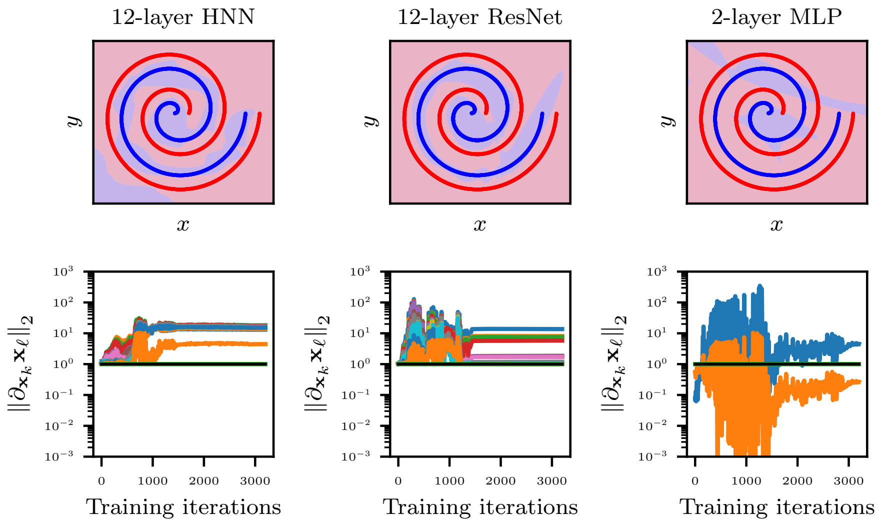
  so that

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

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

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

Gradient Stability in HNNs



1-Lipschitz Neural Networks

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

Adversarial robustness

Constraining the Lipschitz constant leads to a reduced sensitivity to input perturbations.

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

Adversarial robustness

Constraining the Lipschitz constant leads to a reduced sensitivity to input perturbations.

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

Constraining the Lipschitz constant leads to a reduced sensitivity to input perturbations.

Wasserstein Generative Adversarial Networks (Kantorovich-Rubinstein duality)

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

Convergent fixed point iterations

If $\|f(\mathbf{y}) - f(\mathbf{x})\|_2 < \|\mathbf{y} - \mathbf{x}\|_2$ for every $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, then $\mathbf{x}_{k+1} = f(\mathbf{x}_k)$ admits a unique and attractive fixed point. If $T_\alpha(\mathbf{x}) = (1-\alpha)\mathbf{x} + \alpha g(\mathbf{x})$, $\alpha \in (0,1)$ and g 1-Lipschitz, then whenever $\mathbf{x}_{k+1} = T_\alpha(\mathbf{x}_k)$ has a fixed point, the sequence converges.

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:

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

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

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

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

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

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

Non-expansive gradient flows

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

Consider the scalar function[a] $V_\theta(x) = 1^\top \mathrm{ReLU}^2(W\mathbf{x}+\mathbf{b})/2$. Define

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

If $\dot{\mathbf{x}} = \mathcal{F}_\theta(\mathbf{x})$ and $\dot{\mathbf{y}} = \mathcal{F}_\theta(\mathbf{y})$, we have $\|\mathbf{y}(t)-\mathbf{x}(t)\|_2 \leq \|\mathbf{y}(0)-\mathbf{x}(0)\|_2$ for every $t \geq 0$.

[a] $W \in \mathbb{R}^{h\times d}$, $\mathbf{b} \in \mathbb{R}^h$, $h \in \mathbb{N}$, $\theta = (W, \mathbf{b})$, and $1 \in \mathbb{R}^h$ a vector of ones.

Non-expansive gradient flows

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

Consider the scalar function[a] $V_\theta(x) = 1^\top \mathrm{ReLU}^2(W\mathbf{x}+\mathbf{b})/2$. Define

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

If $\dot{\mathbf{x}} = \mathcal{F}_\theta(\mathbf{x})$ and $\dot{\mathbf{y}} = \mathcal{F}_\theta(\mathbf{y})$, we have $\|\mathbf{y}(t)-\mathbf{x}(t)\|_2 \leq \|\mathbf{y}(0)-\mathbf{x}(0)\|_2$ for every $t \geq 0$.

[a] $W \in \mathbb{R}^{h\times d}$, $\mathbf{b} \in \mathbb{R}^h$, $h \in \mathbb{N}$, $\theta = (W, \mathbf{b})$, and $1 \in \mathbb{R}^h$ a vector of ones.

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.,

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

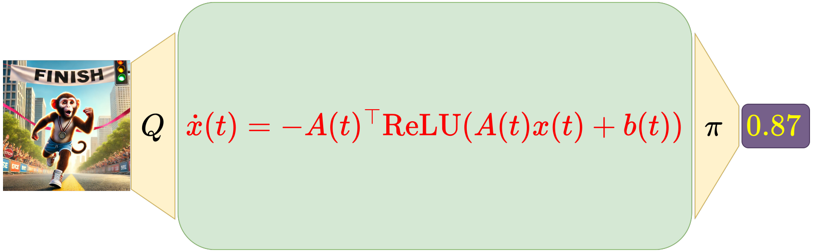
Neural networks based on gradient flows

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 affine maps.



1-Lipschitz Networks for Robust Classification

The problem of robust classification

Classification problem

Let $\Omega \subset \mathbb{R}^d$ be a set whose points are known to belong to C classes. Given part of their labels, we want to label the remaining points using $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^C$ where we set

$$\text{predicted class of } \mathbf{x} = \arg\max_{c=1,\dots,C} \left(\mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_c \right).$$

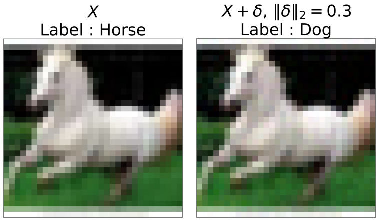
The problem of robust classification

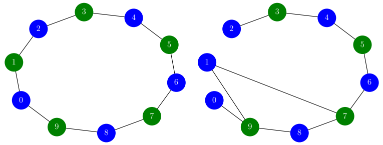
Classification problem

Let $\Omega \subset \mathbb{R}^d$ be a set whose points are known to belong to C classes. Given part of their labels, we want to label the remaining points using $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^C$ where we set

$$\text{predicted class of } \mathbf{x} = \arg\max_{c=1,\dots,C} \left(\mathcal{N}_\theta(\mathbf{x})^\top \boldsymbol{e}_c \right).$$

Adversarial examples





How to have guaranteed robustness

- Not all correct predictions are equivalent.
- Let $\ell(\mathbf{x}) = 2$ be the correct label for the point $\mathbf{x} \in \Omega$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ is not so certain as a prediction.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ there is a higher gap here.

How to have guaranteed robustness

- Not all correct predictions are equivalent.
- Let $\ell(\mathbf{x}) = 2$ be the correct label for the point $\mathbf{x} \in \Omega$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ is not so certain as a prediction.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ there is a higher gap here.

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 equivalent.
- Let $\ell(\mathbf{x}) = 2$ be the correct label for the point $\mathbf{x} \in \Omega$.
- $\mathcal{N}_{\theta_1}(\mathbf{x}) = \begin{bmatrix} 0.49 & 0.51 & 0 \end{bmatrix}$ is not so certain as a prediction.
- $\mathcal{N}_{\theta_2}(\mathbf{x}) = \begin{bmatrix} 0.05 & 0.9 & 0.05 \end{bmatrix}$ there is a higher gap here.

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

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

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

- We constrain the Lipschitz constant of $\mathcal{N}_\theta$ (and train the network so it maximises the margin).

Adapting Gradient Flows to Convolutional Neural Networks

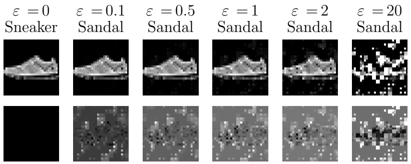
Code Snippet 1: Fully-connected

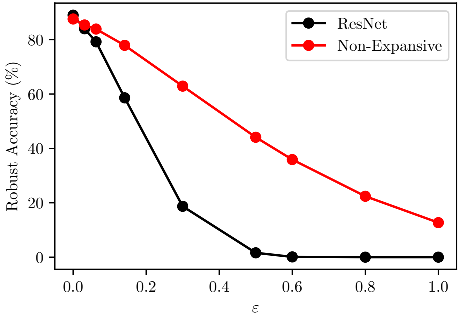
```
A = nn.Parameter(torch.randn(h,h))
b = nn.Parameter(torch.randn(h))
tau = nn.Parameter(torch.tensor([2.]))
x = x - tau * act(x @ A.T + b) @ A
```

Code Snippet 2: Convolutional

```
A = nn.Conv2d(in_channels=h,out_channels=h,kernel_size=3,padding=1)
tau = nn.Parameter(torch.tensor([2.]))
K = A.weight
V = nn.functional.conv_transpose2d(input=act(A(x)), weight=K, padding=1)
x = x - tau * V
```

Robustness to adversarial attacks





1-Lipschitz Networks for Inverse Problems

The Proximal Gradient Descent Method

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

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:

$$\mathbf{x}_{k+1} = \mathrm{prox}_{\gamma g,\tau}\left(\mathbf{x}_k - \tau\nabla f(\mathbf{x}_k)\right), \ \tau > 0,$$
$$\mathrm{prox}_{\gamma g,\tau}(\mathbf{x}) = \arg\min_{\mathbf{z}\in\mathbb{R}^d}\left(\frac{1}{2\tau}\|\mathbf{x}-\mathbf{z}\|_2^2 + \gamma g(\mathbf{z})\right).$$

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.

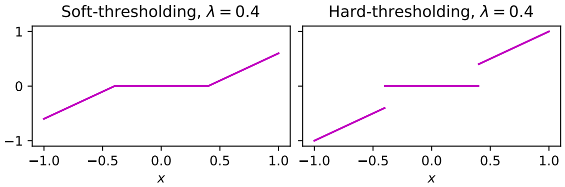
Example: ISTA (cfr. SINDy)

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

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

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



The Plug-and-Play Method

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

1. It is extremely 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 Method

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

1. It is extremely 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 $\mathrm{prox}_{\gamma g,\alpha}$ 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 Method

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

1. It is extremely 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 $\mathrm{prox}_{\gamma g,\alpha}$ 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:

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

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 in (2) converge to a unique fixed point.

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 $\mathcal{N}_\theta$.

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

PSNR (Peak Signal-to-Noise Ratio)

$$\mathrm{PSNR}(\hat{\mathbf{x}}, \mathbf{x}^*) = 10\log_{10}\left(\frac{\max_{i,j,k}|x^*_{i,j,k}|^2}{\frac{1}{3\cdot 321\cdot 481}\sum_{i,j,k}|x^*_{i,j,k} - \hat{x}_{i,j,k}|^2}\right).$$

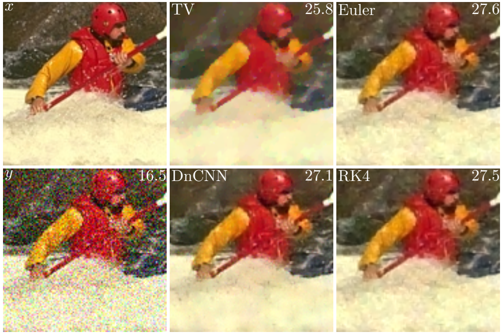


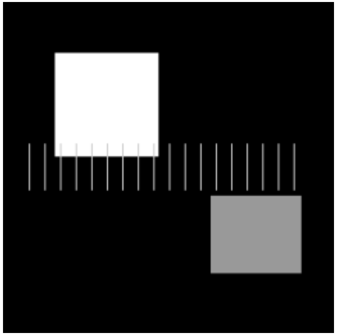
Figure 1: Image from BSDS500 dataset, composed of 500 natural colour images of size 321 × 481.

What do we mean with Deblurring?

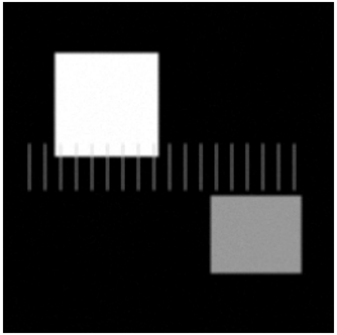
- Let us consider the **inverse problem of deblurring**: we assume that we are given measurements $y = Kx + \varepsilon$, where $K\mathbf{x} = k * \mathbf{x}$ is a convolution operation representing a motion blur.
- The ill-posedness of this problem is manifested in the instability of the inverse of 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

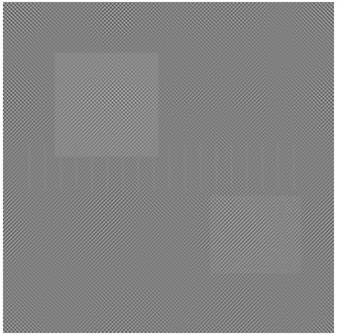
Original, x



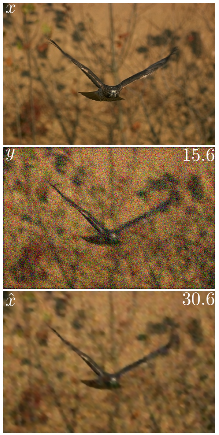
Blur + noise, $y = Kx + \varepsilon$



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



Use in a Deblurring Task



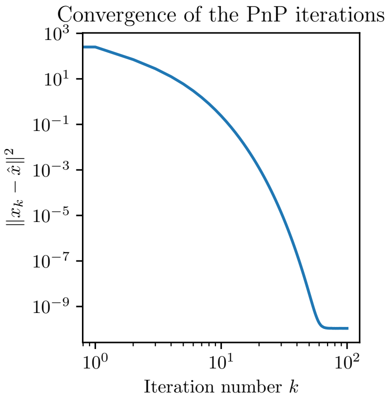


Figure 2: Using the learned Euler denoiser to solve an ill-posed inverse problem (deblurring) in a PnP fashion, with convergence guarantee.

APPENDIX

Neural ODEs

Neural ODEs: The Continuous-Depth Limit

- ResNet layers can be interpreted as discretisations of parametric ODEs.
- If we go to the limit as the time step goes to zero, we can recover a dynamical system

$$\dot{\mathbf{x}}(t) = \mathcal{F}_\theta(t, \mathbf{x}(t)),\ \theta \in \Theta,$$

where $\mathcal{F}_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is parametrised by a neural network.

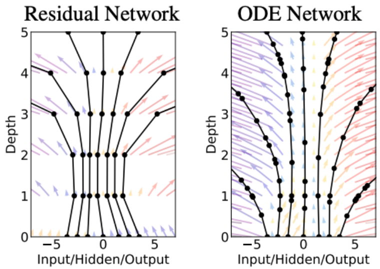


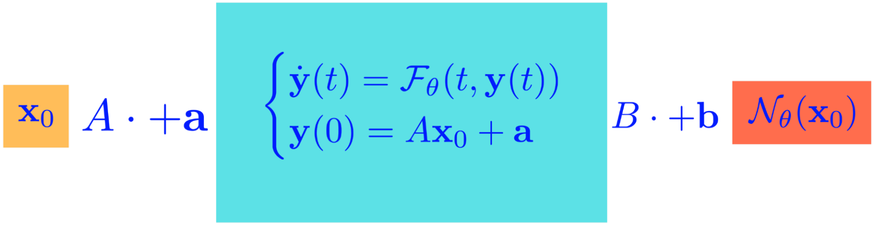
Figure 3: Source: Chen et al., "Neural Ordinary Differential Equations".

Neural ODEs

More explicitly, a Neural ODE is a parametric map $\mathcal{N}_\theta : \mathbb{R}^d \to \mathbb{R}^c$ of the form

$$\mathcal{N}_\theta(\mathbf{x}_0) = B\mathbf{y}(T) + \mathbf{b} \in \mathbb{R}^c, \quad \begin{cases} \dot{\mathbf{y}}(t) = \mathcal{F}_\theta(t, \mathbf{y}(t)), \ \mathcal{F}_\theta : \mathbb{R} \times \mathbb{R}^h \to \mathbb{R}^h, \\ \mathbf{y}(0) = A\mathbf{x}(0) + \mathbf{a} \in \mathbb{R}^h, \end{cases}$$

for an $h \in \mathbb{N}$. Here, $A \in \mathbb{R}^{h\times d}$, $\mathbf{a} \in \mathbb{R}^h$, $B \in \mathbb{R}^{c\times h}$, and $\mathbf{b} \in \mathbb{R}^c$.


$\mathbf{x}_0$
$A \cdot + \mathbf{a}$
$\begin{cases} \dot{\mathbf{y}}(t) = \mathcal{F}_\theta(t, \mathbf{y}(t)) \\ \mathbf{y}(0) = A\mathbf{x}_0 + \mathbf{a} \end{cases}$
$B \cdot + \mathbf{b}$
$\mathcal{N}_\theta(\mathbf{x}_0)$

How to train them: discrete backpropagation vs. adjoint method

- There are two main strategies to train Neural ODEs:

  1. Discretise backpropagation, and
  2. Adjoint method.

- The first, corresponds to the conventional backpropagation algorithm, where the forward pass is defined through a numerical method:

$$\begin{aligned}\mathbf{y}_0 &= A\mathbf{x}_0 + \mathbf{a}\\ \mathbf{y}_{k+1} &= \varphi^{h_k}_{\mathcal{F}_\theta}(t_k, \mathbf{y}_k),\ t_{k+1} = t_k + h_{k+1},\ k = 0, \dots, K-1,\\ \mathcal{N}_\theta(\mathbf{x}_0) &= B\mathbf{y}_K + \mathbf{b}.\end{aligned}$$

  As long as the numerical method φ is differentiable, we can backpropagate through it and minimise the loss function to find a good set of weights.

The adjoint sensitivity method

- For simplicity, fix $d = c$, and consider Neural ODEs of the form

$$\mathcal{N}_\theta(\mathbf{x}_0) = \mathbf{x}(T) = \mathbf{x}_0 + \int_0^t \mathcal{F}_\theta(t, \mathbf{x}(t))dt, \; A = B = I_d, \, \mathbf{a} = \mathbf{b} = 0.$$

- Let us introduce a loss function $L : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$, and study the gradient $\nabla_\theta L(\mathcal{N}_\theta(\mathbf{x}_0), \mathbf{y})$.

- First, we introduce the so-called adjoint variable

$$\mathbf{a}(t) = \frac{\partial L(\mathbf{x}(T), \mathbf{y})}{\partial \mathbf{x}(t)} \in \mathbb{R}^d.$$

Assuming to know $\mathbf{x}(T)$, we see that $\mathbf{a}(T)$ is known as well. What about a generic $\mathbf{a}(t)$ for $t \in [0, T)$?.

The adjoint sensitivity method

- For any $t \in \mathbb{R}$ and $\varepsilon > 0$, we see that

$$\mathbf{x}(t+\varepsilon) = \mathbf{x}(t) + \int_t^{t+\varepsilon} \mathcal{F}_\theta(s, \mathbf{x}(s))ds.$$

  Furthermore, by the chain rule we get

$$\frac{dL(\mathbf{x}(T), \mathbf{y})}{d\mathbf{x}(t)} = \left(\frac{d\mathbf{x}(t+\varepsilon)}{d\mathbf{x}(t)}\right)^\top \frac{dL(\mathbf{x}(T), \mathbf{y})}{d\mathbf{x}(t+\varepsilon)}, \text{ i.e., } \mathbf{a}(t) = \left(\frac{d\mathbf{x}(t+\varepsilon)}{d\mathbf{x}(t)}\right)^\top \mathbf{a}(t+\varepsilon).$$

- This allows us to obtain that

$$\frac{d}{dt}\mathbf{a}(t) = \lim_{\varepsilon \to 0} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} = \ldots = -\left(\frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t).$$

- It follows that, for $t \in [0, T)$:

$$\mathbf{a}(t) = \mathbf{a}(T) - \int_T^t \left(\frac{\partial \mathcal{F}_\theta(s, \mathbf{x}(s))}{\partial \mathbf{x}(s)}\right)^\top \mathbf{a}(s)ds.$$

The adjoint sensitivity method

- Let $\theta \in \mathbb{R}^p$. Call $J_\theta(t) = \frac{\partial \mathbf{x}(t)}{\partial \theta} \in \mathbb{R}^{d\times p}$, a matrix which satisfies the ODE

$$\frac{d}{dt} J_\theta(t) = \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \theta} + \frac{\partial \mathcal{F}_\theta(t, \mathbf{x}(t))}{\partial \mathbf{x}(t)} J_\theta(t), \;\; J_\theta(0) = \frac{\partial \mathbf{x}_0}{\partial \theta} = 0_{d\times p}.$$

- We see that

$$\mathbb{R}^p \ni \nabla_\theta L(\mathbf{x}(T), \mathbf{y}) = (J_\theta(T))^\top \mathbf{a}(T) = (J_\theta(0))^\top \mathbf{a}(0) + \int_0^T \frac{d}{dt}\Big((J_\theta(t))^\top \mathbf{a}(t)\Big)\, dt.$$

The desired expression follows from the derivation below

$$\frac{d}{dt}((J_\theta(t))^\top \mathbf{a}(t)) = -(J_\theta(t))^\top \left(\partial_{\mathbf{x}(t)} \mathcal{F}_\theta(t, \mathbf{x}(t))\right)^\top \mathbf{a}(t)$$
$$+ \left(\partial_\theta \mathcal{F}_\theta(t, \mathbf{x}(t)) + \partial_{\mathbf{x}(t)} \mathcal{F}_\theta(t, \mathbf{x}(t)) J_\theta(t)\right)^\top \mathbf{a}(t) = \left(\partial_\theta \mathcal{F}_\theta(t, \mathbf{x}(t))\right)^\top \mathbf{a}(t).$$
$$\Longrightarrow \;\; \nabla_\theta L(\mathbf{x}(T), \mathbf{y}) = \int_0^T \left(\partial_\theta \mathcal{F}_\theta(t, \mathbf{x}(t))\right)^\top \mathbf{a}(t) dt.$$

Variations of the conventional Neural ODE

There are a lot of research papers considering alternative design strategies for Neural ODEs. We include here a couple:

- In Krzysztof M Choromanski et al. "Ode to an ODE". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 3338–3350, the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

$$\begin{cases} \dot{\mathbf{x}}(t) = \sigma(W(t)\mathbf{x}(t)) \in \mathbb{R}^d, \ \text{(e.g. } \sigma(x) = |x|) \\ \dot{W}(t) = W(t)\Omega(t, W(t)) \in \mathbb{R}^{d\times d}, \ \Omega(t, W) \in \mathrm{Skew}(d) \\ \mathbf{x}(0) = \mathbf{x}_0, W(0) = W_0 \in \mathcal{O}(d). \end{cases}$$

Variations of the conventional Neural ODE

There are a lot of research papers considering alternative design strategies for Neural ODEs. We include here a couple:

- In Choromanski et al., "Ode to an ODE", the authors augment the Neural ODE with an ODE for the network weights, which hence become time-dependent:

$$\begin{cases} \dot{\mathbf{x}}(t) = \sigma(W(t)\mathbf{x}(t)) \in \mathbb{R}^d, \ \text{(e.g. } \sigma(x) = |x|) \\ \dot{W}(t) = W(t)\Omega(t, W(t)) \in \mathbb{R}^{d\times d}, \ \Omega(t, W) \in \text{Skew}(d) \\ \mathbf{x}(0) = \mathbf{x}_0, W(0) = W_0 \in \mathcal{O}(d). \end{cases}$$

- In Norcliffe et al., "On Second Order Behaviour in Augmented Neural ODEs", the authors consider second order Neural ODEs

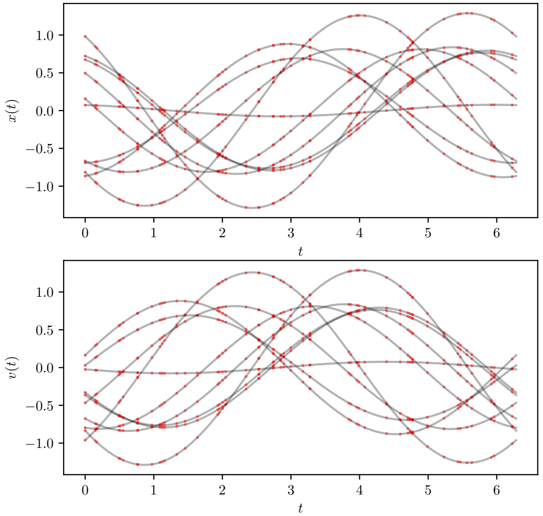
$$\ddot{\mathbf{x}}(t) = \mathcal{F}_\theta(\mathbf{x}(t), \dot{\mathbf{x}}(t), t, \theta) \in \mathbb{R}^d \iff \begin{cases} \dot{\mathbf{x}}(t) = \mathbf{v}(t) \\ \dot{\mathbf{v}}(t) = \mathcal{F}_\theta(\mathbf{x}(t), \mathbf{v}(t), t, \theta). \end{cases}$$

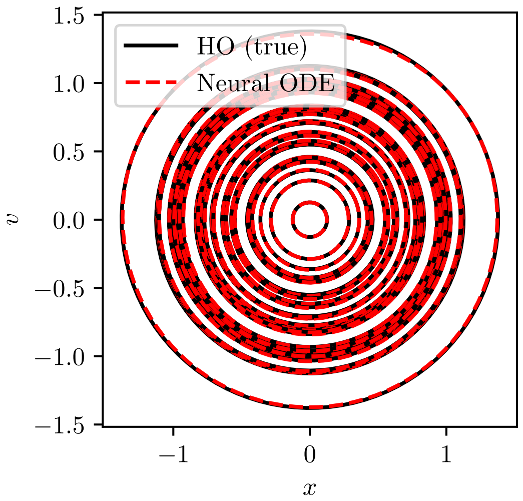
Implementation with PyTorch

- There are several libraries that allow for the quick implementation of these models.
- An example is https://github.com/rtqichen/torchdiffeq:

```
import numpy as np; from scipy.integrate import odeint as sp_odeint
import torch, torch.nn as nn, torch.optim as optim; from torchdiffeq import odeint_adjoint as odeint

simpleHO = lambda y, t: [y[1], -y[0]]

T = np.linspace(0., 2*np.pi, 50); Y0_np = np.random.randn(1000, 2)
Y_star_np = np.stack([sp_odeint(simpleHO, y0, T) for y0 in Y0_np], axis=1)

T_t = torch.from_numpy(T).float(); Y0  = torch.from_numpy(Y0_np).float(); Y_star = torch.from_numpy(Y_star_np).float()

class ODEFunc(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(2, 32), nn.Tanh(), nn.Linear(32, 2))
    def forward(self, t, y):
        return self.net(y)

f = ODEFunc(); opt = optim.Adam(f.parameters(), lr=1e-2)

for _ in range(1000):
    Y = odeint(f, Y0, T_t)
    loss = (Y - Y_star).pow(2).mean()
    opt.zero_grad(); loss.backward(); opt.step()
```

Simulation with irregular time-sampling





What do we mean by generative modelling?

A generative model is a machine learning model designed to create new data that is similar to its training data. Generative models learn the distribution of the training data, then apply those understandings to generate new content in response to new input data.

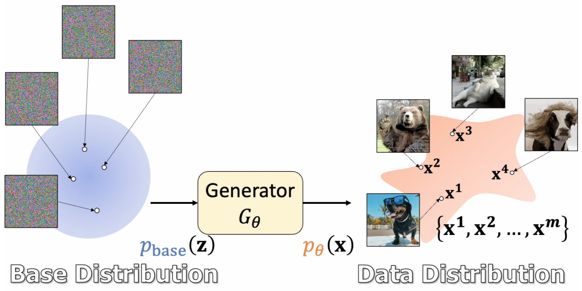


Figure 5: Source: `https://www.youtube.com/watch?v=DDq_pIfHqLs&ab_channel=Jia-BinHuang`.

Second application of Neural ODEs: Generative Modelling

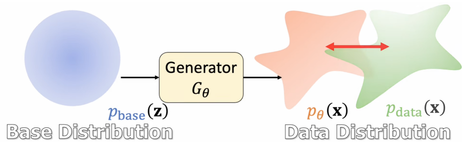


Figure 6: Source: `https://www.youtube.com/watch?v=DDq_pIfHqLs&ab_channel=Jia-BinHuang`.

A way to get p_θ as close as possible to the correct distribution p_{data} is to maximise the log-likelihood:

$$\arg\max_\theta \mathbb{E}_{x\sim p_{\text{data}}}[\log(p_\theta(x))] = \arg\min_\theta D_{\text{KL}}(p_{\text{data}}||p_\theta),\ D_{\text{KL}}(P||Q) = \mathbb{E}_{x\sim P}\left[\log\frac{P(x)}{Q(x)}\right].$$

$$\text{Empirically: } \arg\min_\theta -\frac{1}{N}\sum_{i=1}^N \log(p_\theta(x_i)),\ x_1,\dots,x_N \sim p_{\text{data}},\ \text{iid}.$$

Generative modelling with continuous normalising flows

Consider a neural ODE

$$\begin{cases} \frac{d}{dt}\phi_{t,\theta}(z) = \mathcal{F}_t^\theta(\phi_{t,\theta}(z)),\ \mathcal{F}^\theta : [0,1] \times \mathbb{R}^d \to \mathbb{R}^d, \\ \phi_0(z) = z, \end{cases}$$

and an easy-to-sample probability measure with density p_{init}. We then set $x = \phi_1(z)$.

Continuous normalising flows define $p_t^\theta = (\phi_{t,\theta})_* p_{\text{init}}$, $t \in [0,1]$, as

$$p_t^\theta(x) = (\phi_{t,\theta})_* p_{\text{init}}(x) = p_{\text{init}}(\phi_{t,\theta}^{-1}(x)) \left| \det \partial_x \left(\phi_{t,\theta}^{-1}(x) \right) \right|.$$

This leads to $\log p_1^\theta(x) = \log(p_{\text{init}}(\phi_{1,\theta}^{-1}(x))) - \int_0^1 \mathrm{div}(\mathcal{F}_s^\theta)(\phi_{s,\theta}^{-1}(x))ds$.

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.

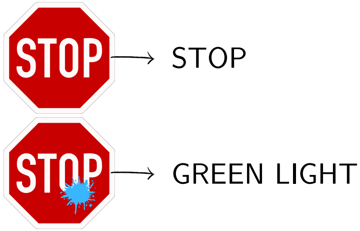


Figure 7: Misclassification of an image that could harm self-driving cars.

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.

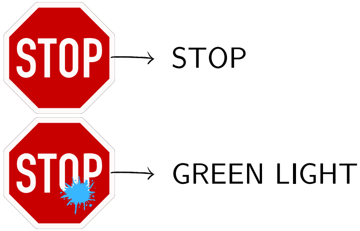
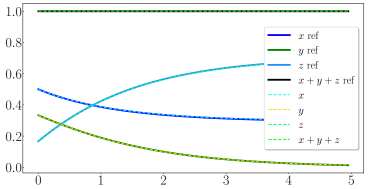


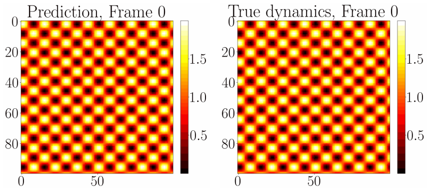
Figure 7: Misclassification of an image that could harm self-driving cars.

- Such properties are achievable only by constraining the neural networks we construct so that they behave as desired. We call the area of Deep Learning interested in constraining neural networks **structure-preserving deep learning**.

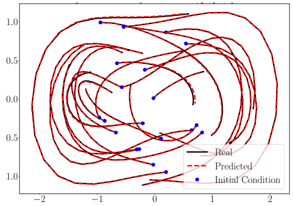
Some learning problems with a structure worth preserving



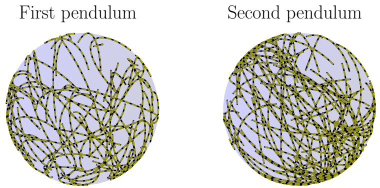
(a) Learning the mass preserving flow map of the SIR model.



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



(c) Learning the Hamiltonian of unconstrained systems.



(d) Learning the Hamiltonian of 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.

Structured networks based on dynamical systems

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

Structured networks based on dynamical systems

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

- Choose a family of parametric vector fields $\mathcal{S}_\Theta$ whose solutions satisfy $\mathcal{P}$, e.g.

$$\mathcal{F}_\theta(\mathbf{x}) = \begin{bmatrix} \sigma\left(A_1\mathbf{x}_2 + \mathbf{b}_1\right) \\ \sigma\left(A_2\mathbf{x}_1 + \mathbf{b}_2\right) \end{bmatrix} = \begin{bmatrix} \sigma\left(A_1\mathbf{x}_2 + \mathbf{b}_1\right) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma\left(A_2\mathbf{x}_1 + \mathbf{b}_2\right) \end{bmatrix}, \ \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix},$$

with $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{x}_1 \in \mathbb{R}^{d_1}$, $\mathbf{x}_2 \in \mathbb{R}^{d_2}$, and $d = d_1 + d_2$.

Structured networks based on dynamical systems

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

$$\mathcal{F}_\theta(\mathbf{x}) = \begin{bmatrix} \sigma\left(A_1\mathbf{x}_2 + \mathbf{b}_1\right) \\ \sigma\left(A_2\mathbf{x}_1 + \mathbf{b}_2\right) \end{bmatrix} = \begin{bmatrix} \sigma\left(A_1\mathbf{x}_2 + \mathbf{b}_1\right) \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma\left(A_2\mathbf{x}_1 + \mathbf{b}_2\right) \end{bmatrix}, \ \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix},$$

  with $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{x}_1 \in \mathbb{R}^{d_1}$, $\mathbf{x}_2 \in \mathbb{R}^{d_2}$, and $d = d_1 + d_2$.
- Choose a numerical method $\Psi^h_{\mathcal{F}_\theta}$ that preserves the property $\mathcal{P}$ at a discrete level, e.g.

$$\Psi^h_{\mathcal{F}_\theta}(\mathbf{x}) = \begin{bmatrix} \mathbf{x}_1 + h\sigma\left(A_1\mathbf{x}_2 + b_1\right) =: \widetilde{\mathbf{x}}_1 \\ \mathbf{x}_2 + h\sigma\left(A_2\widetilde{\mathbf{x}}_1 + \mathbf{b}_2\right) \end{bmatrix}.$$

- 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 $\mathcal{P}$.

Remark on Properties not Closed Under Composition

Not all the properties a function might satisfy are closed under composition. This makes their imposition over NNs more challenging. An example is given by the set of gradient vector fields

$$\mathcal{G} = \left\{ \mathcal{F} : \mathbb{R}^d \to \mathbb{R}^d \mid \text{ exists } V : \mathbb{R}^d \to \mathbb{R}, \mathcal{F} = \nabla V \right\}.$$

- A way to model neural networks which are gradients is: $V_\theta(\mathbf{x}) = \mathrm{MLP}_\theta(\mathbf{x})$, $\mathcal{F}_\theta = \nabla V_\theta$ (hard to train in high dimensions)
- Or rely on modified architectures, such as

$$\mathcal{F}_\theta(\mathbf{x}) = \mathcal{N}_\theta(\mathbf{x}) - (\partial_{\mathbf{x}} \mathcal{N}_\theta(\mathbf{x}))^\top \mathbf{x}.$$

We will not go into further details on this property.

Symplectic Numerical Methods

Symplectic numerical methods

A one-step numerical method $\varphi^h : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if and only if when applied to a Hamiltonian system the map φ^h is symplectic, i.e.,

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

Symplectic numerical methods

A one-step numerical method $\varphi^h : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if and only if when applied to a Hamiltonian system the map φ^h is symplectic, i.e.,

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

Symplectic and energy preserving methods

Let $\dot{\mathbf{x}} = \mathbb{J}\nabla H(\mathbf{x})$ be a Hamiltonian system with Hamiltonian H and no conserved quantities other than H. Let φ^h be a symplectic and energy-preserving method for the Hamiltonian system. Then φ^h reproduces the exact solution up to a time re-parametrisation.

Symplectic numerical methods

A one-step numerical method $\varphi^h : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is symplectic if and only if when applied to a Hamiltonian system the map φ^h is symplectic, i.e.,

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

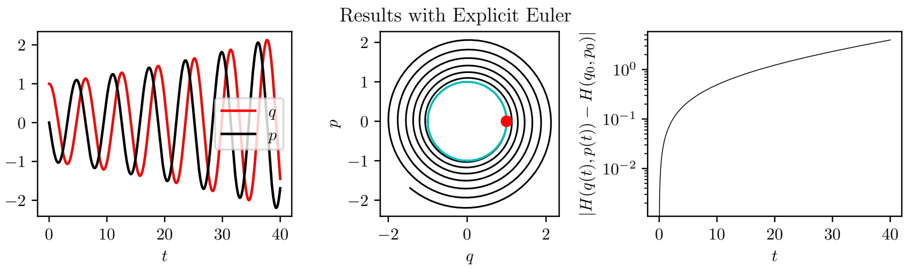
Symplectic and energy preserving methods

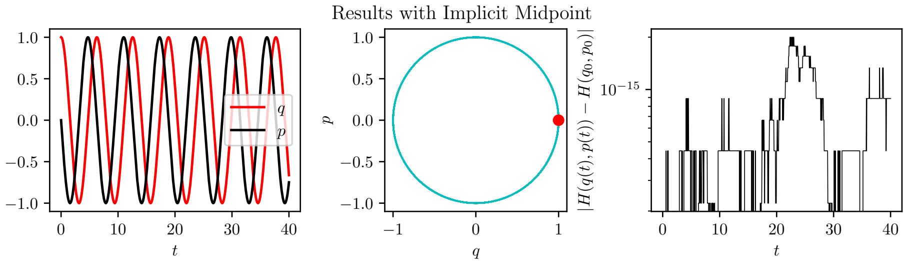
Let $\dot{\mathbf{x}} = \mathbb{J}\nabla H(\mathbf{x})$ be a Hamiltonian system with Hamiltonian H and no conserved quantities other than H. Let φ^h be a symplectic and energy-preserving method for the Hamiltonian system. Then φ^h reproduces the exact solution up to a time re-parametrisation.

Informal theorem

A symplectic method almost conserves the Hamiltonian for an exponentially long time.

Example: simple harmonic oscillator

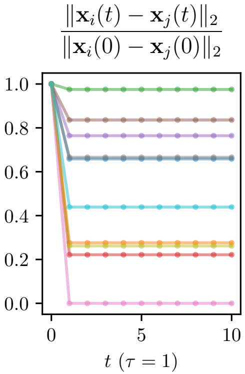


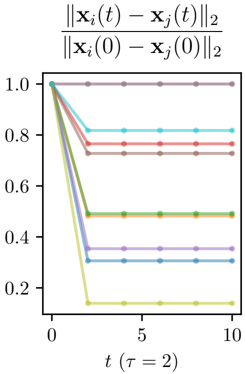


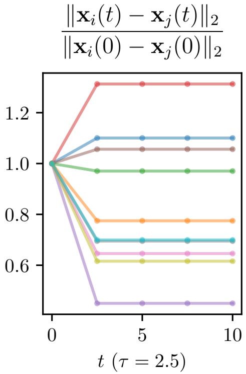
Additional material for 1-Lipschitz networks

ODEs with 1-Lipschitz solution and Euler maps

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



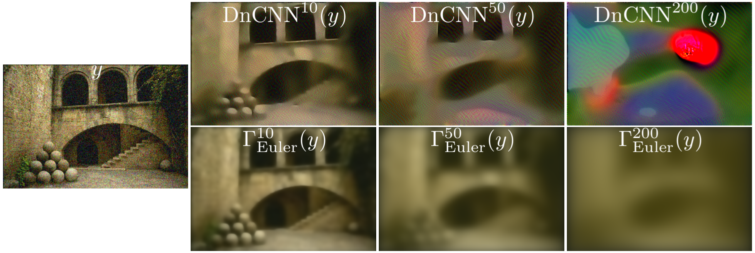




Denoising Performance

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

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



y
$\mathrm{DnCNN}^{10}(y)$
$\mathrm{DnCNN}^{50}(y)$
$\mathrm{DnCNN}^{200}(y)$
$\Gamma^{10}_{\mathrm{Euler}}(y)$
$\Gamma^{50}_{\mathrm{Euler}}(y)$
$\Gamma^{200}_{\mathrm{Euler}}(y)$

Figure 9: Repeated application of the unconstrained denoiser DnCNN[2] and the constrained denoiser Γ_{Euler} to a given input image.

[2] 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.