Neural ODE

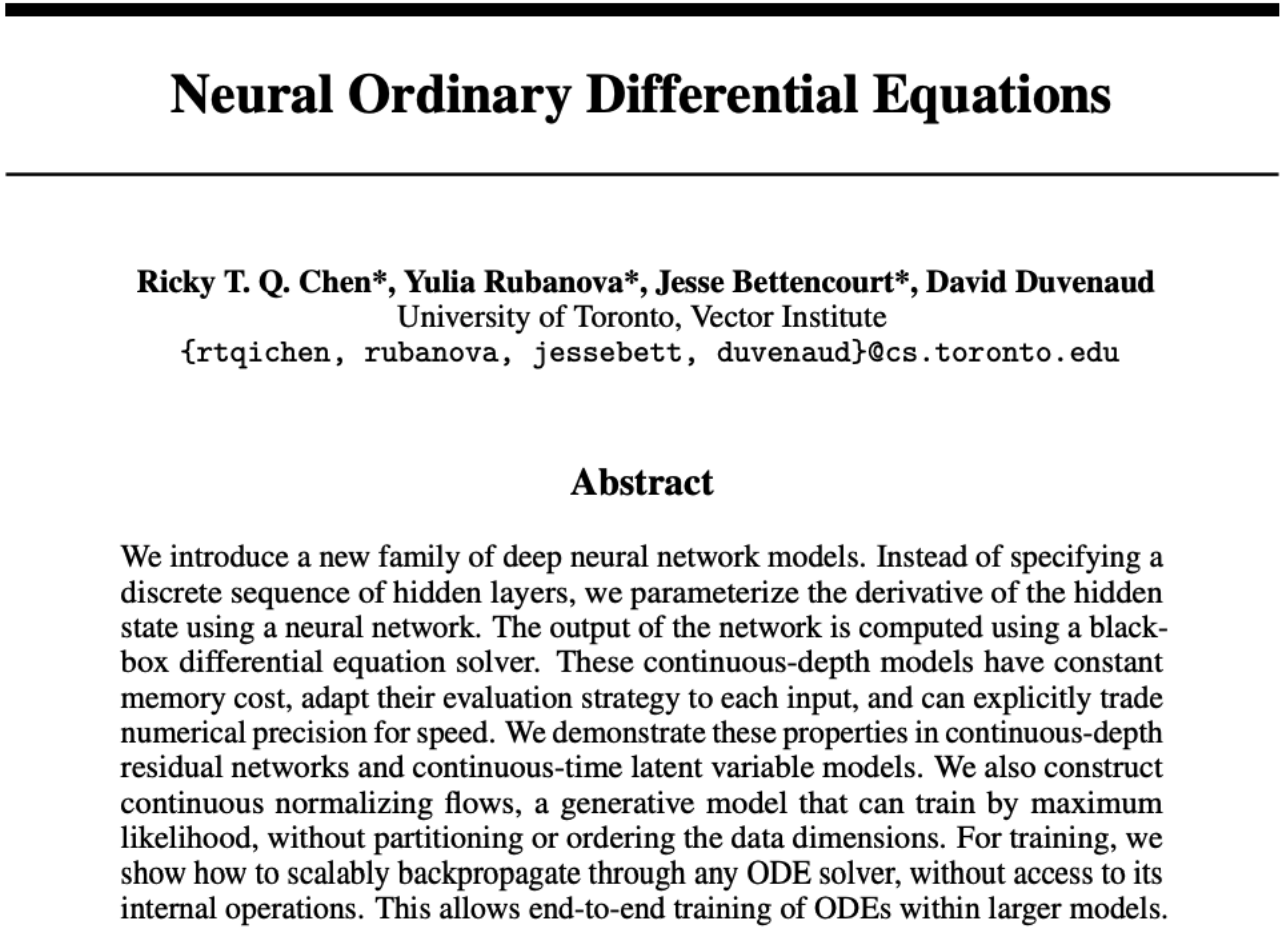
Fundamentals

Oualid Missaoui

Agenda

- Historical Note
- ResNet
- From ResNet to Neural ODE
- Neural ODE
  - Parametrization
  - Explicit vs Implicit layers
  - Standard Training
  - Adjoint Method Training
- References

Historical note



Neural Ordinary Differential Equations

Ricky T. Q. Chen*, Yulia Rubanova*, Jesse Bettencourt*, David Duvenaud
University of Toronto, Vector Institute
{rtqichen, rubanova, jessebett, duvenaud}@cs.toronto.edu

Abstract

We introduce a new family of deep neural network models. Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network. The output of the network is computed using a black-box differential equation solver. These continuous-depth models have constant memory cost, adapt their evaluation strategy to each input, and can explicitly trade numerical precision for speed. We demonstrate these properties in continuous-depth residual networks and continuous-time latent variable models. We also construct continuous normalizing flows, a generative model that can train by maximum likelihood, without partitioning or ordering the data dimensions. For training, we show how to scalably backpropagate through any ODE solver, without access to its internal operations. This allows end-to-end training of ODEs within larger models.

Best paper NIPS 2018

Ricky Chen et al , Neural Ordinary Differential Equations

ResNet

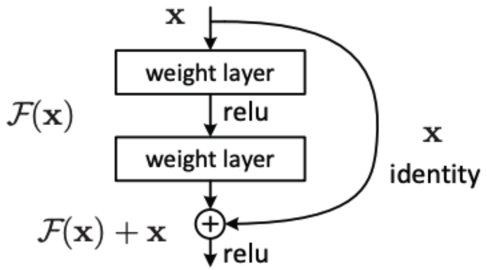

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2. Residual learning: a building block.

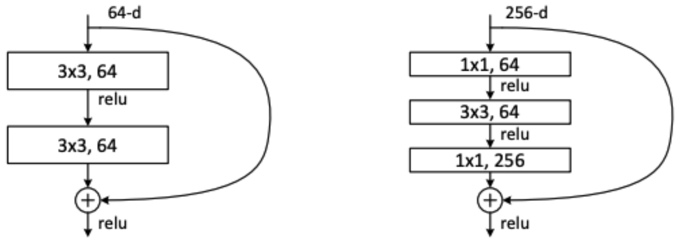


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

$$x_{l+1} = x_l + \text{ResBlock}(x_l, \theta)$$

$$x_{l+1} = x_l + \mathbf{g}(x_l, l, \theta)$$

where $\mathbf{g}(x_l, l, \theta)$ is the l-th residual block. (The parameters of all blocks are concatenated together into θ)

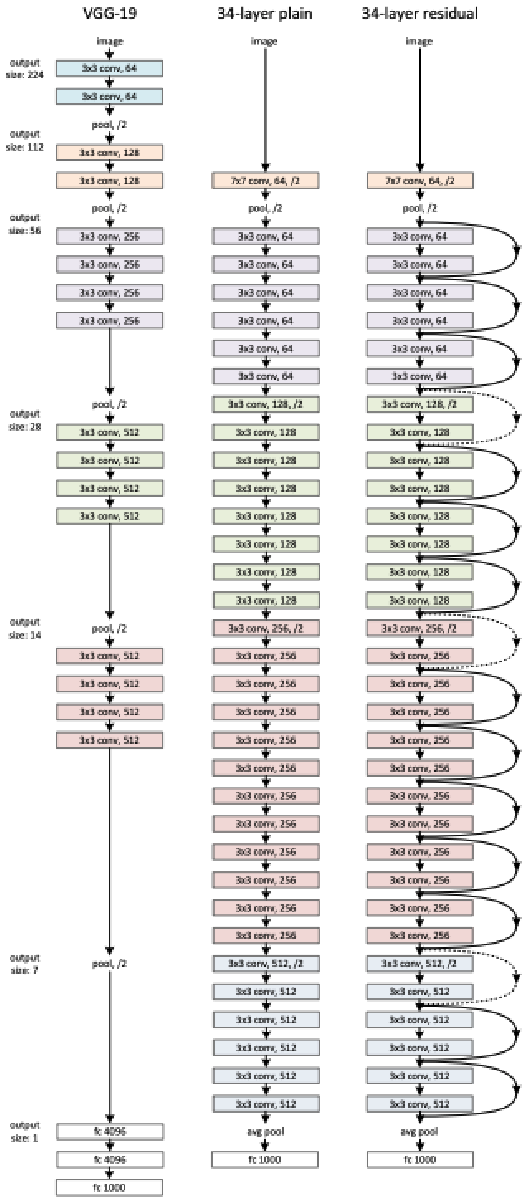
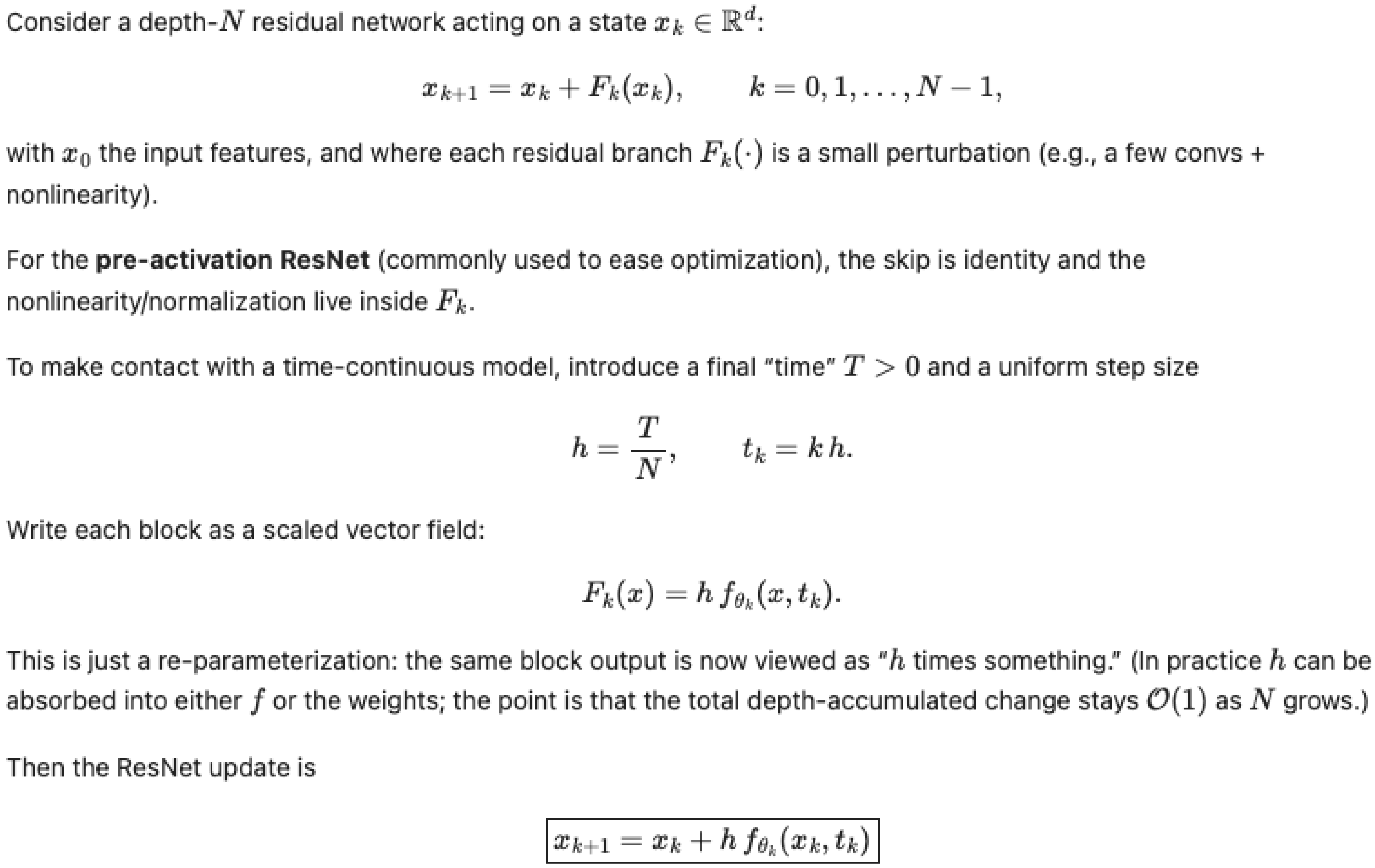


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

Source: Kaiming He et al (2015), Deep Residual Learning for image recognition, Microsoft Research

From ResNet to Neural ODE



Consider a depth-N residual network acting on a state $x_k \in \mathbb{R}^d$:

$$x_{k+1} = x_k + F_k(x_k), \qquad k = 0, 1, \dots, N-1,$$

with x_0 the input features, and where each residual branch $F_k(\cdot)$ is a small perturbation (e.g., a few convs + nonlinearity).

For the **pre-activation ResNet** (commonly used to ease optimization), the skip is identity and the nonlinearity/normalization live inside F_k.

To make contact with a time-continuous model, introduce a final "time" $T > 0$ and a uniform step size

$$h = \frac{T}{N}, \qquad t_k = k\,h.$$

Write each block as a scaled vector field:

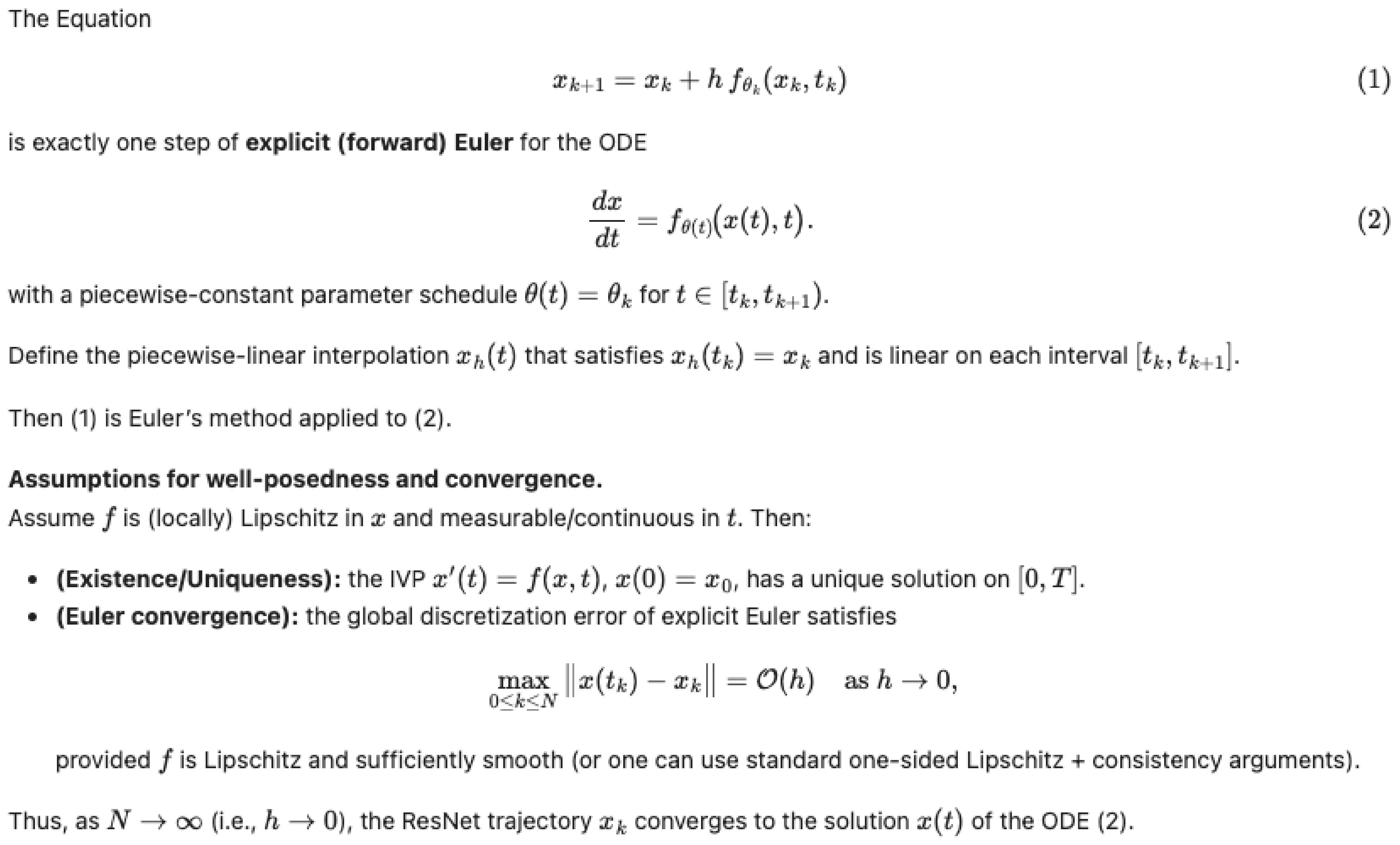
$$F_k(x) = h\, f_{\theta_k}(x, t_k).$$

This is just a re-parameterization: the same block output is now viewed as "h times something." (In practice h can be absorbed into either f or the weights; the point is that the total depth-accumulated change stays $\mathcal{O}(1)$ as N grows.)

Then the ResNet update is

$$\boxed{x_{k+1} = x_k + h\, f_{\theta_k}(x_k, t_k)}$$

Neural ODE



The Equation

$$x_{k+1} = x_k + h\, f_{\theta_k}(x_k, t_k) \tag{1}$$

is exactly one step of **explicit (forward) Euler** for the ODE

$$\frac{dx}{dt} = f_{\theta(t)}(x(t), t). \tag{2}$$

with a piecewise-constant parameter schedule $\theta(t) = \theta_k$ for $t \in [t_k, t_{k+1})$.

Define the piecewise-linear interpolation $x_h(t)$ that satisfies $x_h(t_k) = x_k$ and is linear on each interval $[t_k, t_{k+1}]$.

Then (1) is Euler's method applied to (2).

Assumptions for well-posedness and convergence.
Assume f is (locally) Lipschitz in x and measurable/continuous in t. Then:

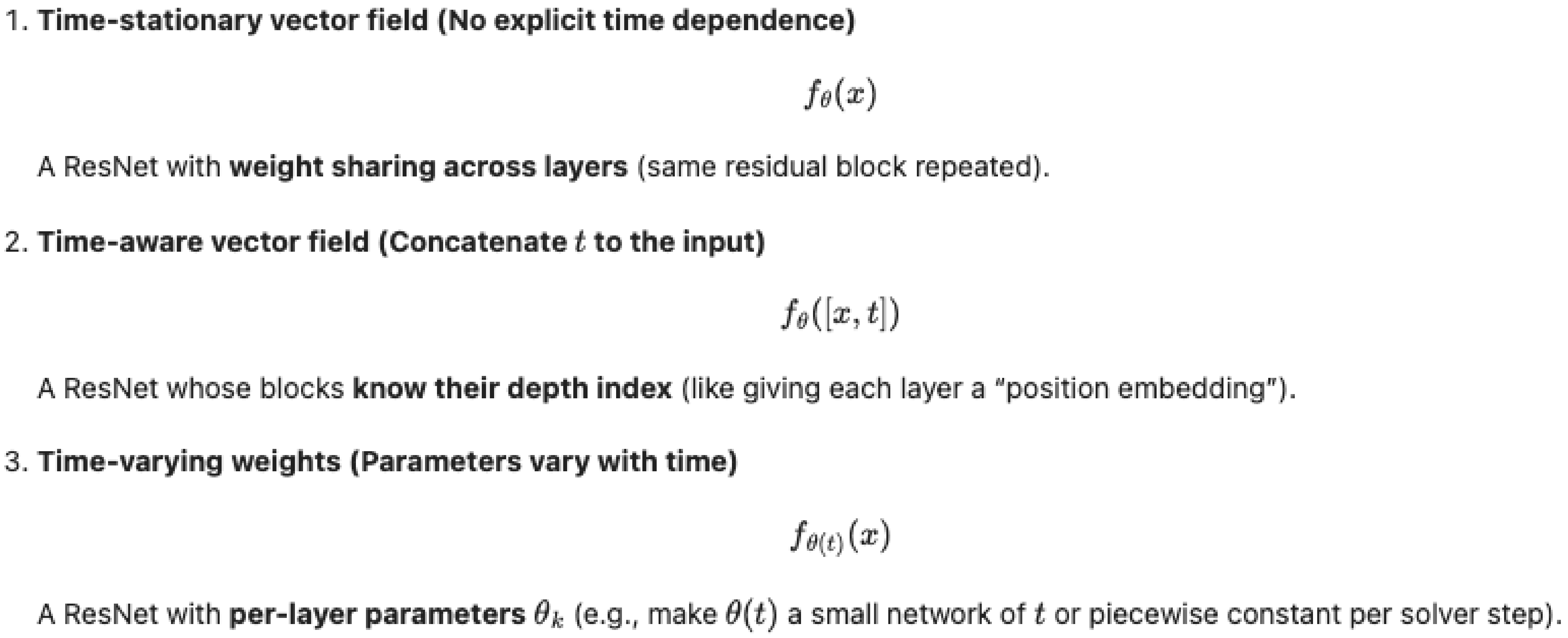
- **(Existence/Uniqueness):** the IVP $x'(t) = f(x, t)$, $x(0) = x_0$, has a unique solution on $[0, T]$.
- **(Euler convergence):** the global discretization error of explicit Euler satisfies

$$\max_{0 \le k \le N} \left\| x(t_k) - x_k \right\| = \mathcal{O}(h) \quad \text{as } h \to 0,$$

provided f is Lipschitz and sufficiently smooth (or one can use standard one-sided Lipschitz + consistency arguments).

Thus, as $N \to \infty$ (i.e., $h \to 0$), the ResNet trajectory x_k converges to the solution $x(t)$ of the ODE (2).

Neural ODE Parameterizations



1. **Time-stationary vector field (No explicit time dependence)**

   $$f_\theta(x)$$

   A ResNet with **weight sharing across layers** (same residual block repeated).

2. **Time-aware vector field (Concatenate t to the input)**

   $$f_\theta([x, t])$$

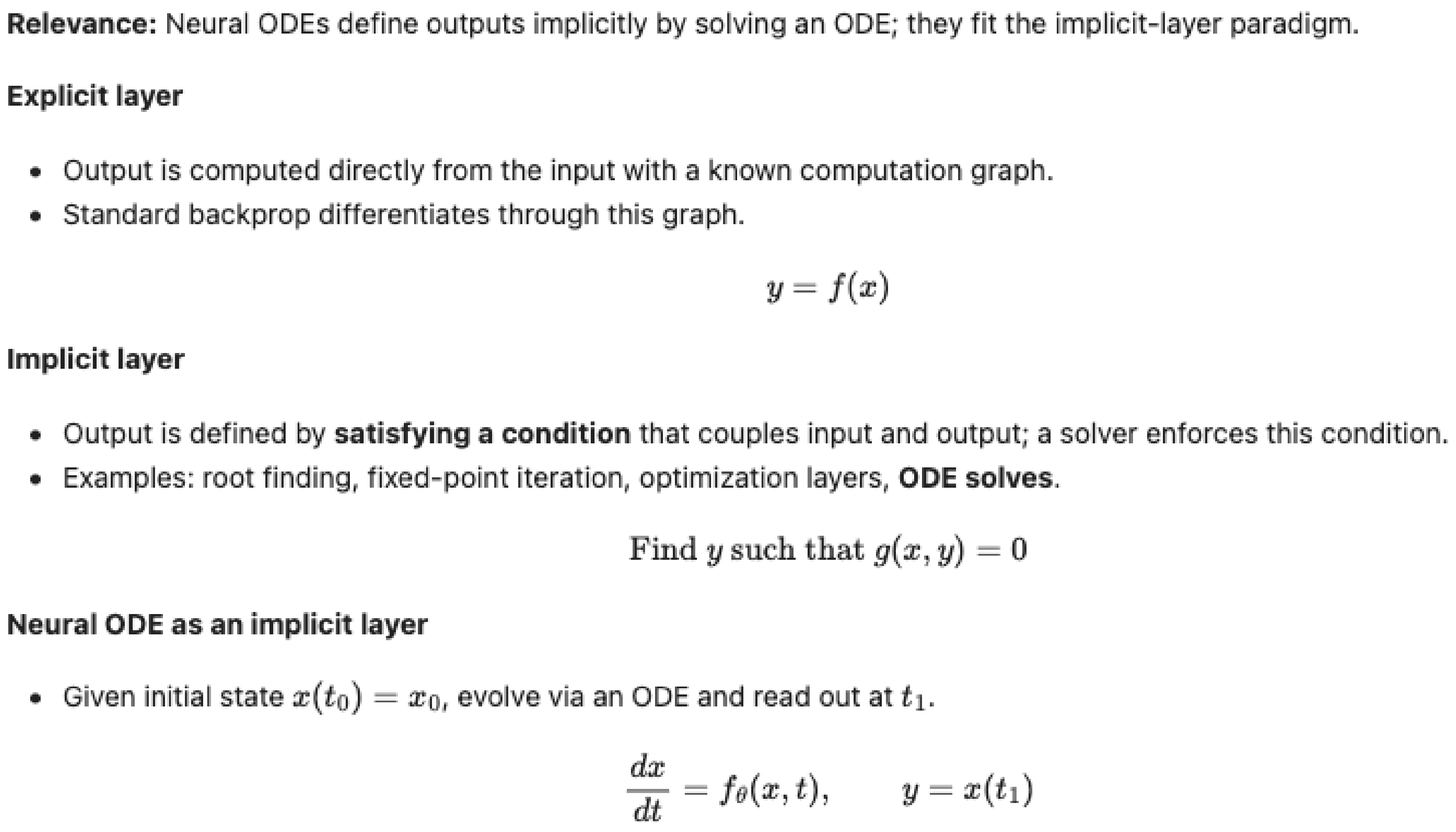
   A ResNet whose blocks **know their depth index** (like giving each layer a "position embedding").

3. **Time-varying weights (Parameters vary with time)**

   $$f_{\theta(t)}(x)$$

   A ResNet with **per-layer parameters** θ_k (e.g., make $\theta(t)$ a small network of t or piecewise constant per solver step).

Explicit vs Implicit layers (and Why Neural ODEs Are Implicit)



Relevance: Neural ODEs define outputs implicitly by solving an ODE; they fit the implicit-layer paradigm.

Explicit layer

- Output is computed directly from the input with a known computation graph.
- Standard backprop differentiates through this graph.

$$y = f(x)$$

Implicit layer

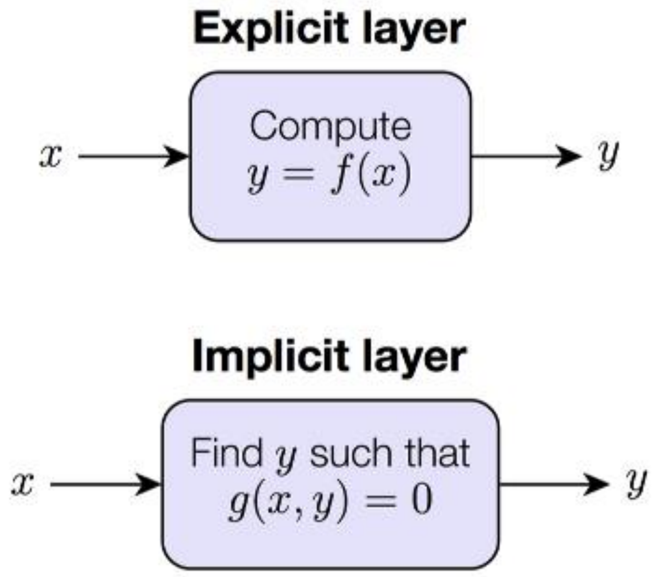
- Output is defined by **satisfying a condition** that couples input and output; a solver enforces this condition.
- Examples: root finding, fixed-point iteration, optimization layers, **ODE solves**.

$$\text{Find } y \text{ such that } g(x, y) = 0$$

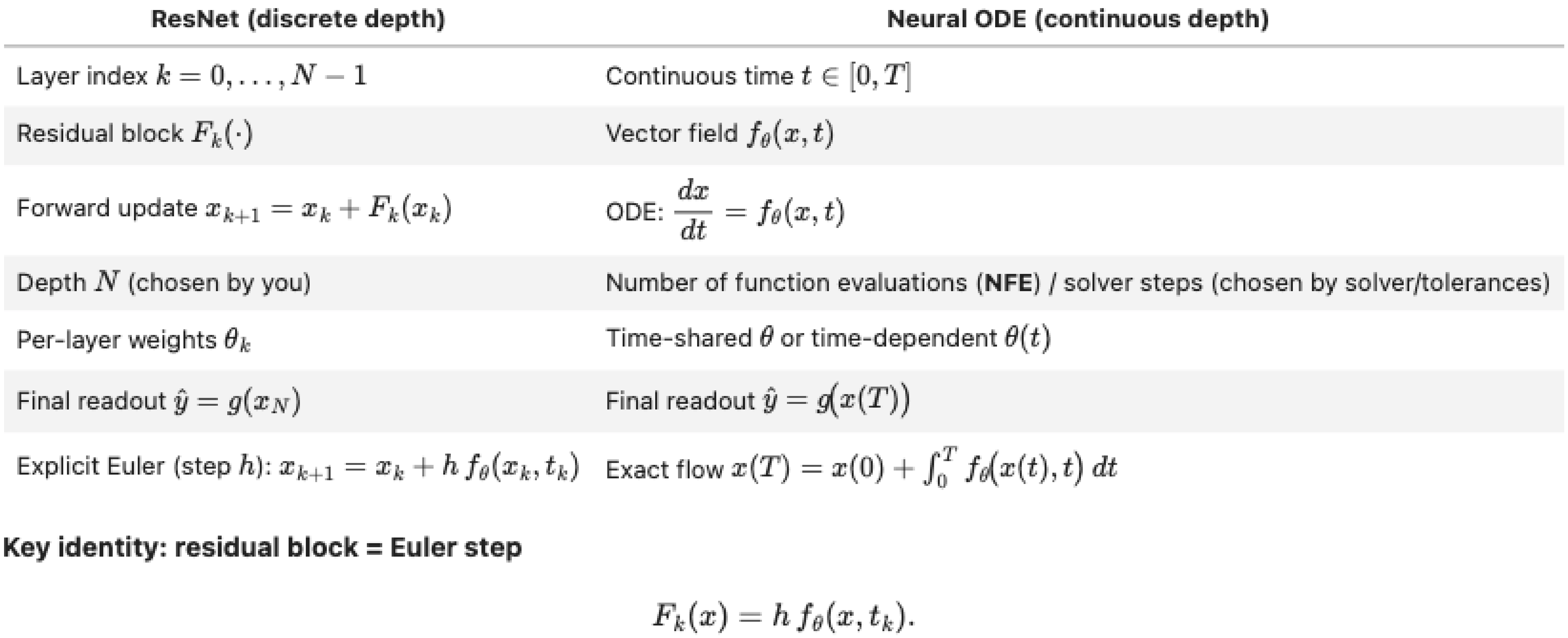
Neural ODE as an implicit layer

- Given initial state $x(t_0) = x_0$, evolve via an ODE and read out at t_1.

$$\frac{dx}{dt} = f_\theta(x, t), \qquad y = x(t_1)$$



ResNet vs Neural ODE: A Discrete vs Continuous Dictionary

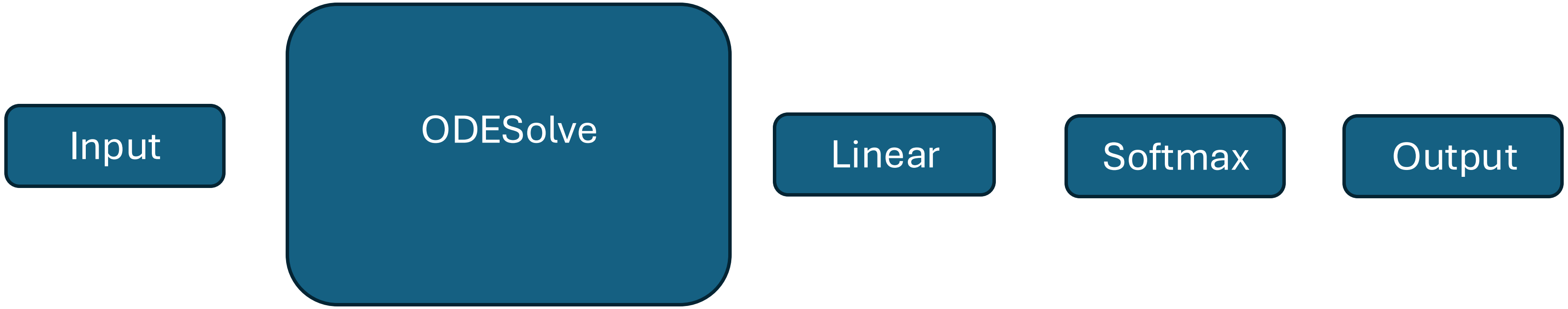


ResNet (discrete depth)	Neural ODE (continuous depth)
Layer index $k = 0, \ldots, N-1$	Continuous time $t \in [0, T]$
Residual block $F_k(\cdot)$	Vector field $f_\theta(x, t)$
Forward update $x_{k+1} = x_k + F_k(x_k)$	ODE: $\dfrac{dx}{dt} = f_\theta(x, t)$
Depth N (chosen by you)	Number of function evaluations (**NFE**) / solver steps (chosen by solver/tolerances)
Per-layer weights θ_k	Time-shared θ or time-dependent $\theta(t)$
Final readout $\hat{y} = g(x_N)$	Final readout $\hat{y} = g\big(x(T)\big)$
Explicit Euler (step h): $x_{k+1} = x_k + h\, f_\theta(x_k, t_k)$	Exact flow $x(T) = x(0) + \int_0^T f_\theta\big(x(t), t\big)\, dt$

Key identity: residual block = Euler step

$$F_k(x) = h\, f_\theta(x, t_k).$$

Neural ODE



Input

ODESolve

Linear

Softmax

Output

- **Neural ODE core:** evolve hidden state by solving

$$\frac{dx}{dt} = f(x, t; \theta), \qquad x(t_0) = x_0.$$

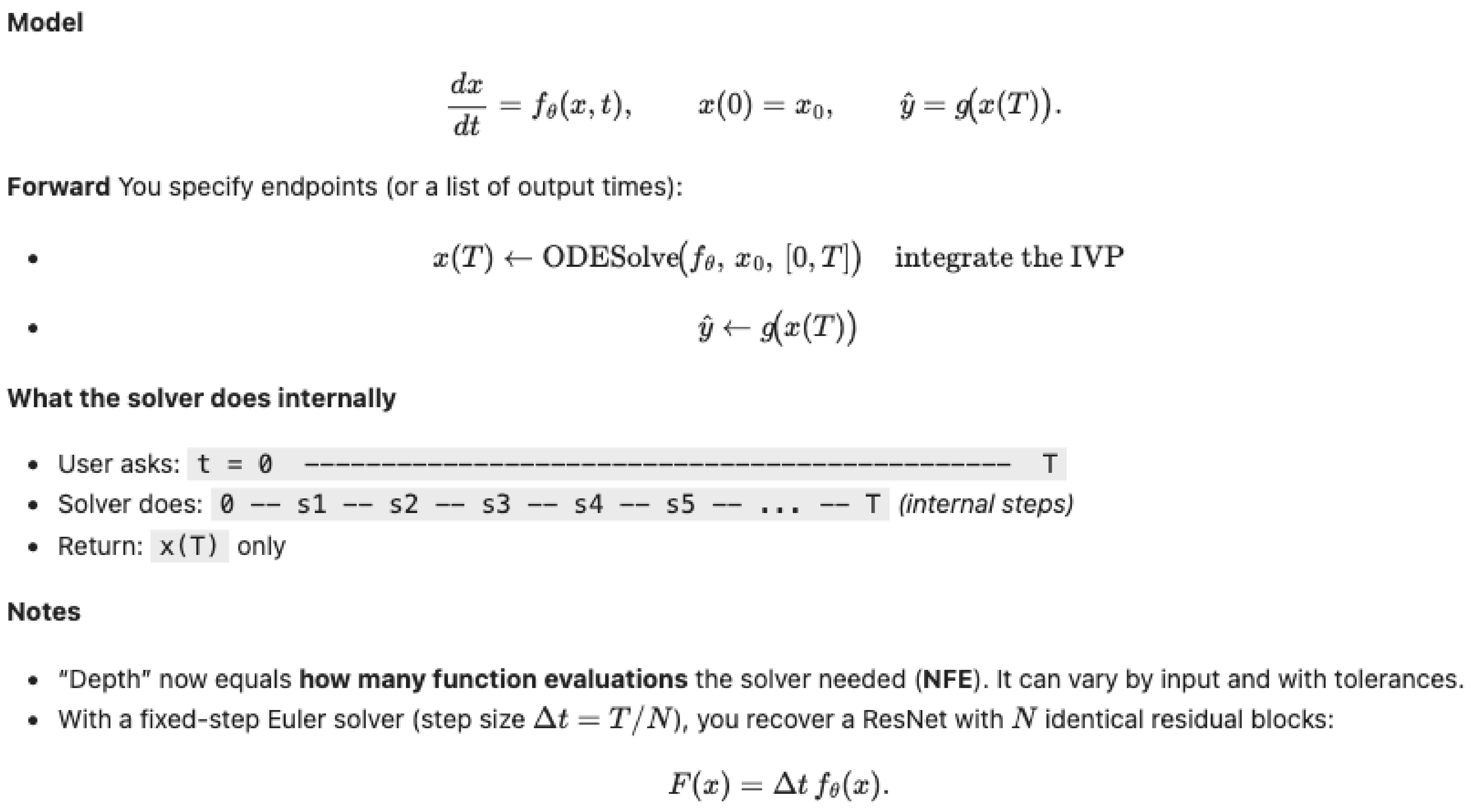
- **Forward pass:**

$$x(t_1) = \mathrm{ODESolve}\big(f,\, x_0,\, t_0 \to t_1; \theta\big), \quad z = W\, x(t_1) + b, \quad \hat{y} = \mathrm{softmax}(z).$$

- **Training is standard:** minimize the loss $\mathcal{L}(\hat{y}, y)$ and update parameters by gradient descent / Adam.

$$\Theta = \{\theta, W, b\}, \qquad \Theta \leftarrow \Theta - \eta \nabla_\Theta \mathcal{L}(\hat{y}, y).$$

Forward Pass



Model

$$\frac{dx}{dt} = f_\theta(x,t), \qquad x(0) = x_0, \qquad \hat{y} = g\big(x(T)\big).$$

Forward You specify endpoints (or a list of output times):

- $$x(T) \leftarrow \text{ODESolve}\big(f_\theta,\ x_0,\ [0,T]\big) \quad \text{integrate the IVP}$$
- $$\hat{y} \leftarrow g\big(x(T)\big)$$

What the solver does internally

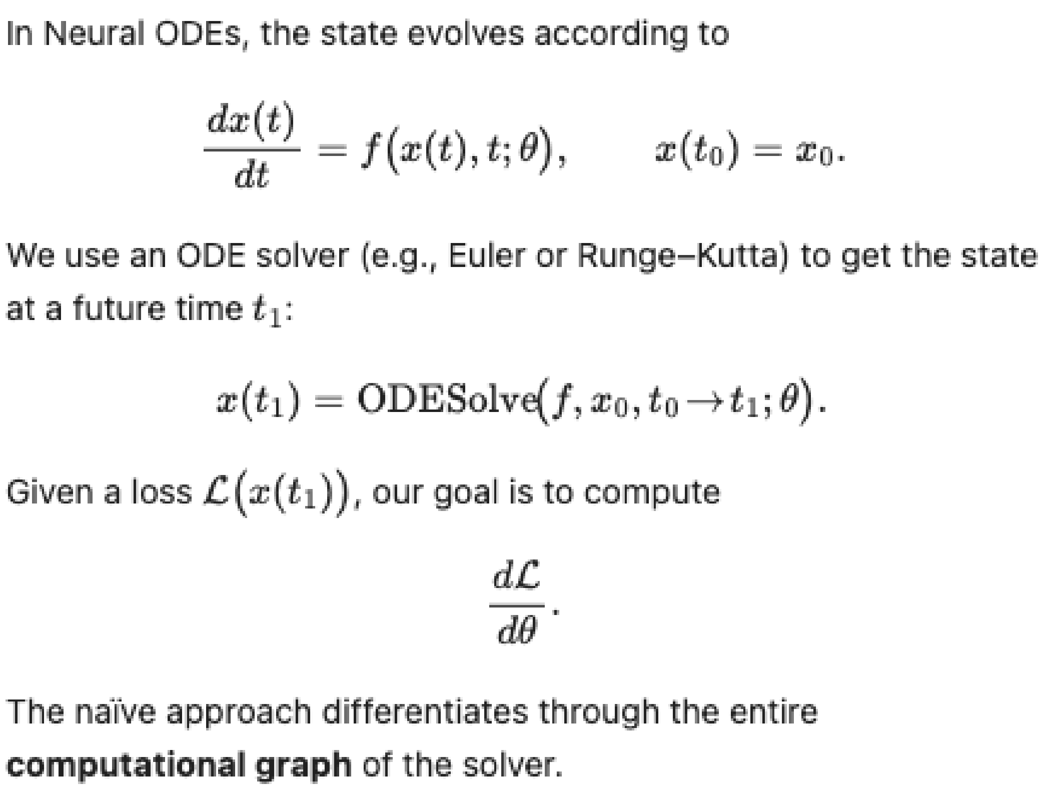
- User asks: `t = 0  ----------------------------------------------  T`
- Solver does: `0 -- s1 -- s2 -- s3 -- s4 -- s5 -- ... -- T` *(internal steps)*
- Return: `x(T)` only

Notes

- "Depth" now equals **how many function evaluations** the solver needed (**NFE**). It can vary by input and with tolerances.
- With a fixed-step Euler solver (step size $\Delta t = T/N$), you recover a ResNet with N identical residual blocks:

$$F(x) = \Delta t\, f_\theta(x).$$

Naïve Backpropagation through the ODE solver (1)



In Neural ODEs, the state evolves according to

$$\frac{dx(t)}{dt} = f\big(x(t), t; \theta\big), \qquad x(t_0) = x_0.$$

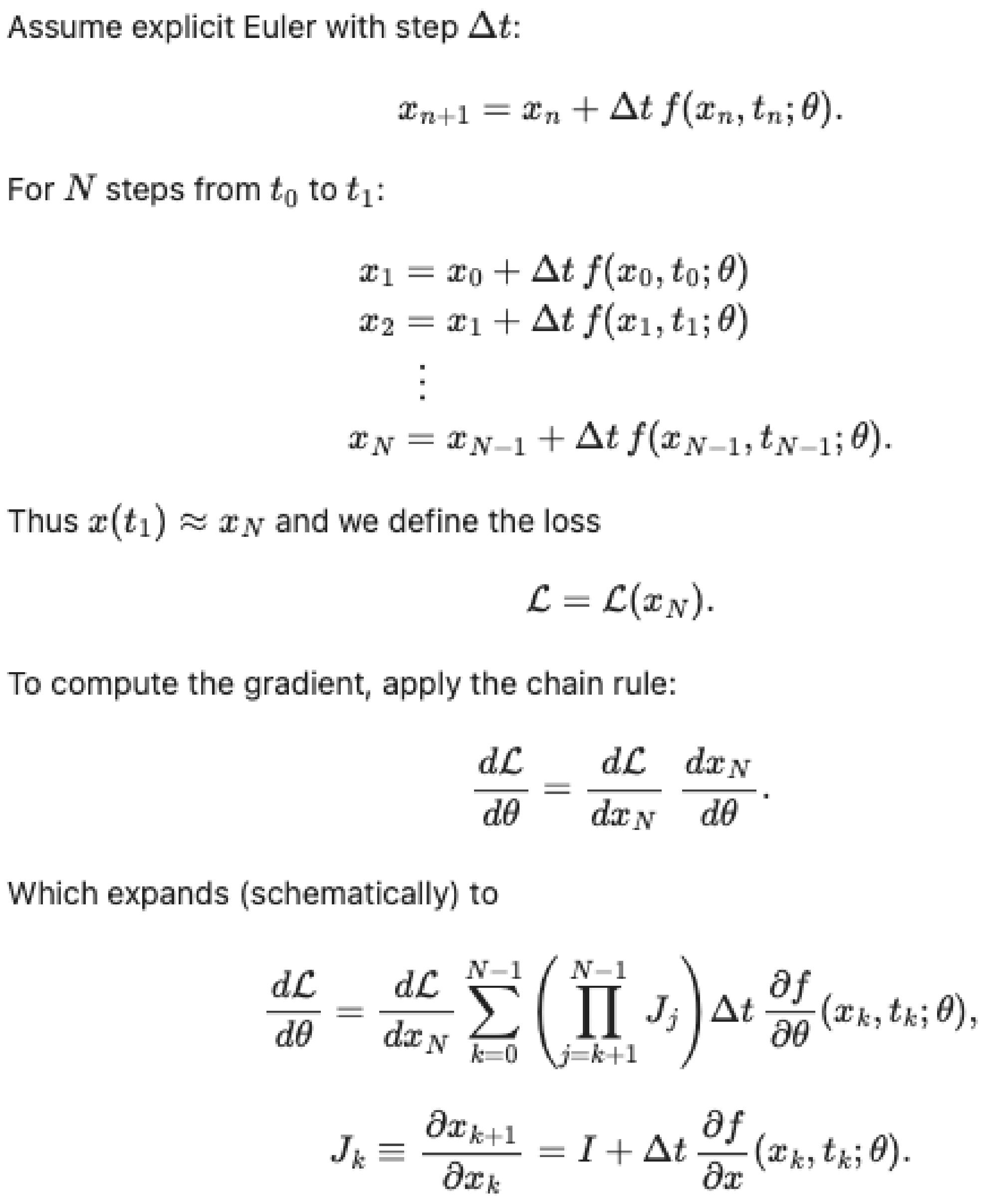
We use an ODE solver (e.g., Euler or Runge–Kutta) to get the state at a future time t_1:

$$x(t_1) = \mathrm{ODESolve}\big(f, x_0, t_0 \to t_1; \theta\big).$$

Given a loss $\mathcal{L}\big(x(t_1)\big)$, our goal is to compute

$$\frac{d\mathcal{L}}{d\theta}.$$

The naïve approach differentiates through the entire **computational graph** of the solver.



Assume explicit Euler with step Δt:

$$x_{n+1} = x_n + \Delta t\, f(x_n, t_n; \theta).$$

For N steps from t_0 to t_1:

$$\begin{aligned} x_1 &= x_0 + \Delta t\, f(x_0, t_0; \theta) \\ x_2 &= x_1 + \Delta t\, f(x_1, t_1; \theta) \\ &\vdots \\ x_N &= x_{N-1} + \Delta t\, f(x_{N-1}, t_{N-1}; \theta). \end{aligned}$$

Thus $x(t_1) \approx x_N$ and we define the loss

$$\mathcal{L} = \mathcal{L}(x_N).$$

To compute the gradient, apply the chain rule:

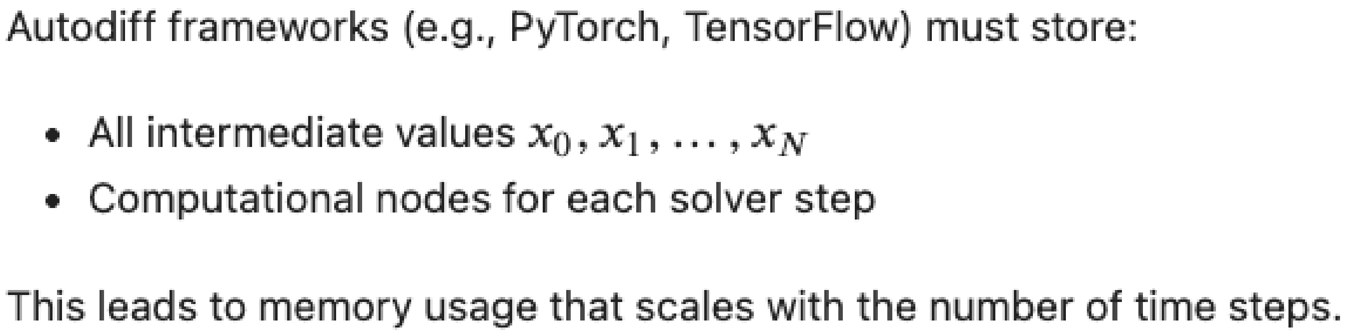
$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{dx_N}\,\frac{dx_N}{d\theta}.$$

Which expands (schematically) to

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{dx_N}\sum_{k=0}^{N-1}\left(\prod_{j=k+1}^{N-1} J_j\right)\Delta t\,\frac{\partial f}{\partial \theta}(x_k, t_k; \theta),$$

$$J_k \equiv \frac{\partial x_{k+1}}{\partial x_k} = I + \Delta t\,\frac{\partial f}{\partial x}(x_k, t_k; \theta).$$

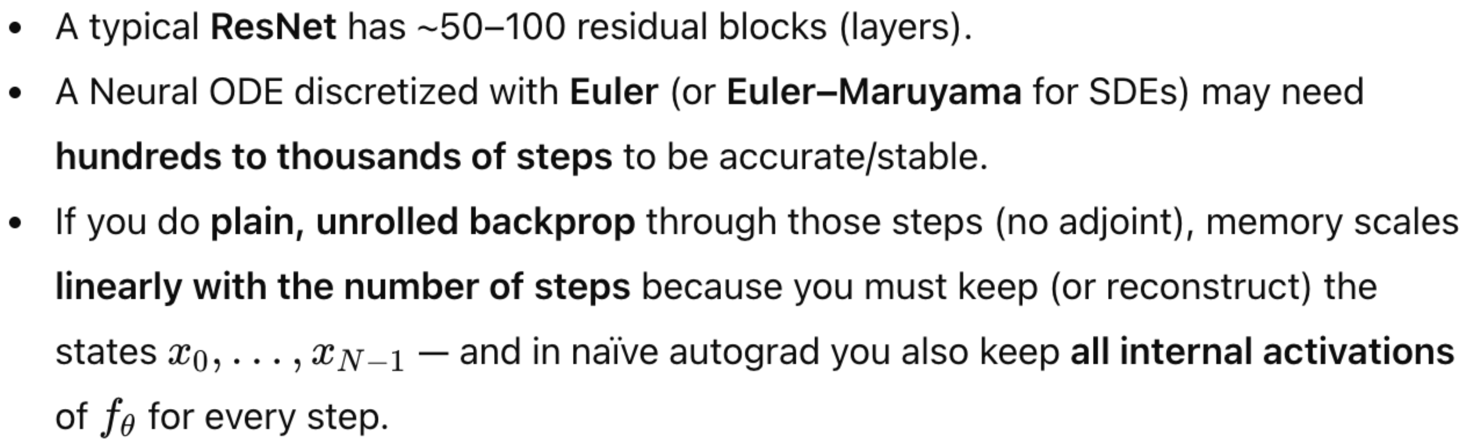
Naïve Backpropagation through the ODE solver (2)



Autodiff frameworks (e.g., PyTorch, TensorFlow) must store:

- All intermediate values $x_0, x_1, \ldots, x_N$
- Computational nodes for each solver step

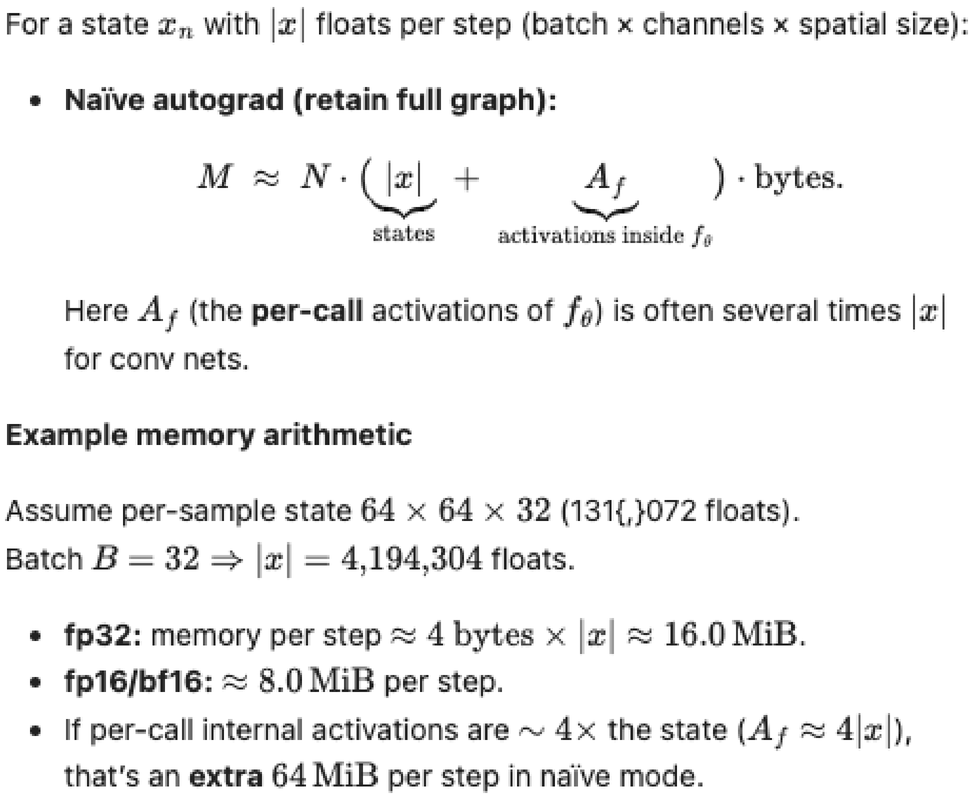
This leads to memory usage that scales with the number of time steps.



- A typical **ResNet** has ~50–100 residual blocks (layers).
- A Neural ODE discretized with **Euler** (or **Euler–Maruyama** for SDEs) may need **hundreds to thousands of steps** to be accurate/stable.
- If you do **plain, unrolled backprop** through those steps (no adjoint), memory scales **linearly with the number of steps** because you must keep (or reconstruct) the states $x_0, \ldots, x_{N-1}$ — and in naïve autograd you also keep **all internal activations** of f_θ for every step.

Aspect	Standard NN	Neural ODE (Naïve BP)
Intermediate storage	Per layer	Per time step (can be many !)
Backprop path	Finite depth	Dense ODE solver steps
Memory usage	Scales with # layers	Scales with # solver steps
Differentiability	Built-in	Solver-dependent

Naïve Backpropagation through the ODE solver (3)



For a state x_n with $|x|$ floats per step (batch × channels × spatial size):

- **Naïve autograd (retain full graph):**

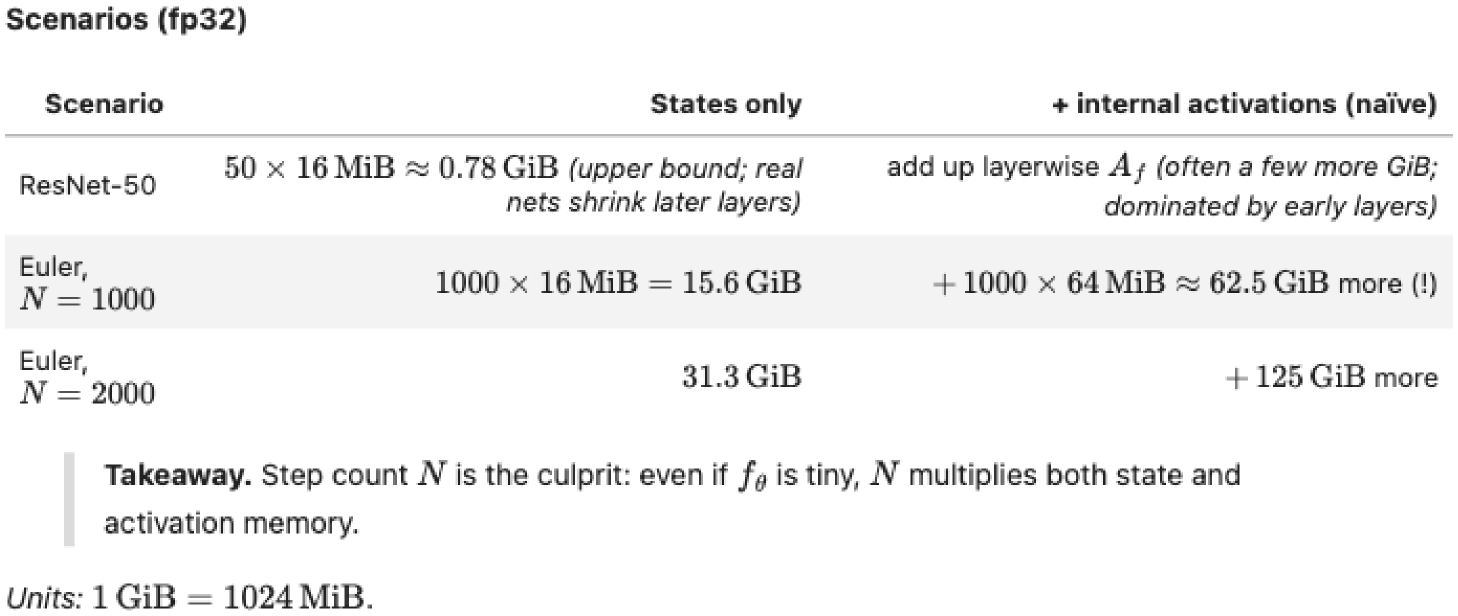
$$M \;\approx\; N\cdot\big(\underbrace{|x|}_{\text{states}} \;+\; \underbrace{A_f}_{\text{activations inside } f_\theta}\big)\cdot\text{bytes.}$$

  Here A_f (the **per-call** activations of f_θ) is often several times $|x|$ for conv nets.

Example memory arithmetic

Assume per-sample state $64\times64\times32$ (131{,}072 floats).
Batch $B=32 \Rightarrow |x| = 4{,}194{,}304$ floats.

- **fp32:** memory per step $\approx 4\ \text{bytes}\times|x| \approx 16.0\,\text{MiB}$.
- **fp16/bf16:** $\approx 8.0\,\text{MiB}$ per step.
- If per-call internal activations are $\sim 4\times$ the state ($A_f \approx 4|x|$), that's an **extra** $64\,\text{MiB}$ per step in naïve mode.



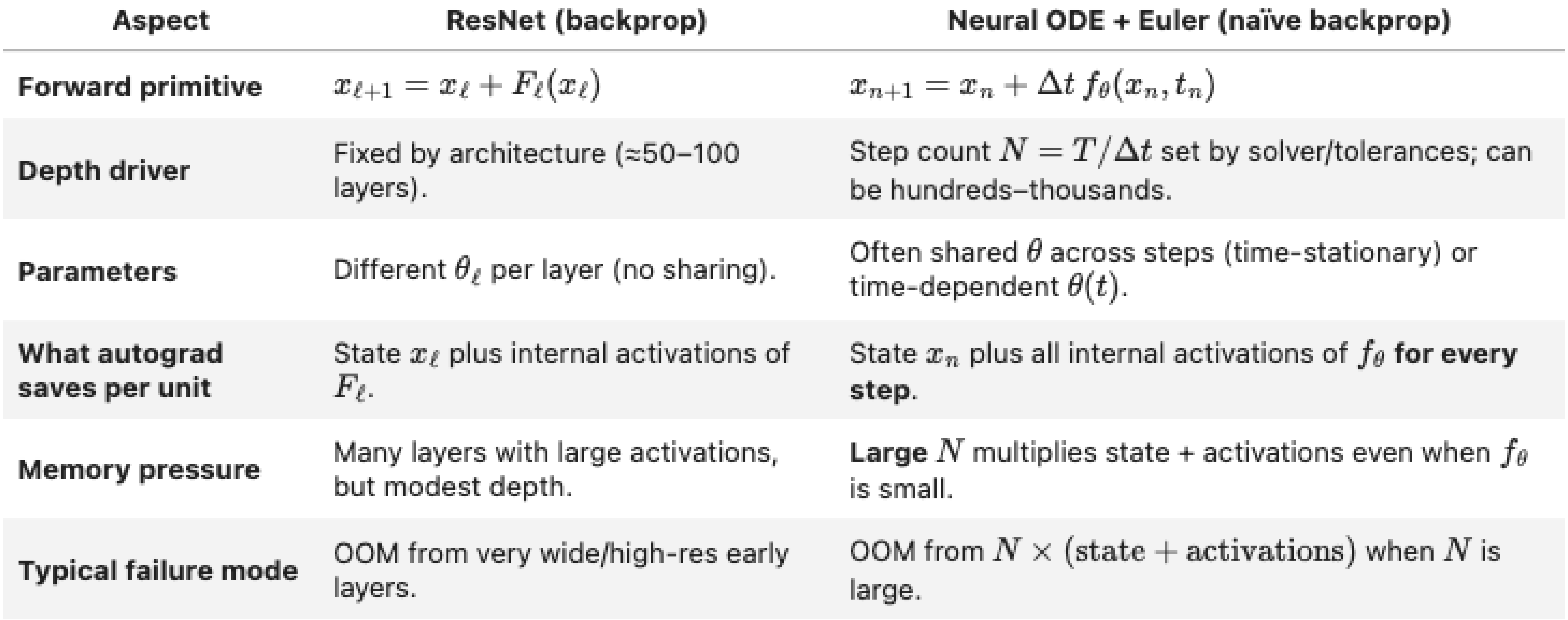
Scenarios (fp32)

Scenario	States only	+ internal activations (naïve)
ResNet-50	$50\times16\,\text{MiB}\approx0.78\,\text{GiB}$ *(upper bound; real nets shrink later layers)*	add up layerwise A_f *(often a few more GiB; dominated by early layers)*
Euler, $N=1000$	$1000\times16\,\text{MiB}=15.6\,\text{GiB}$	$+1000\times64\,\text{MiB}\approx62.5\,\text{GiB}$ more (!)
Euler, $N=2000$	$31.3\,\text{GiB}$	$+125\,\text{GiB}$ more

> **Takeaway.** Step count N is the culprit: even if f_θ is tiny, N multiplies both state and activation memory.

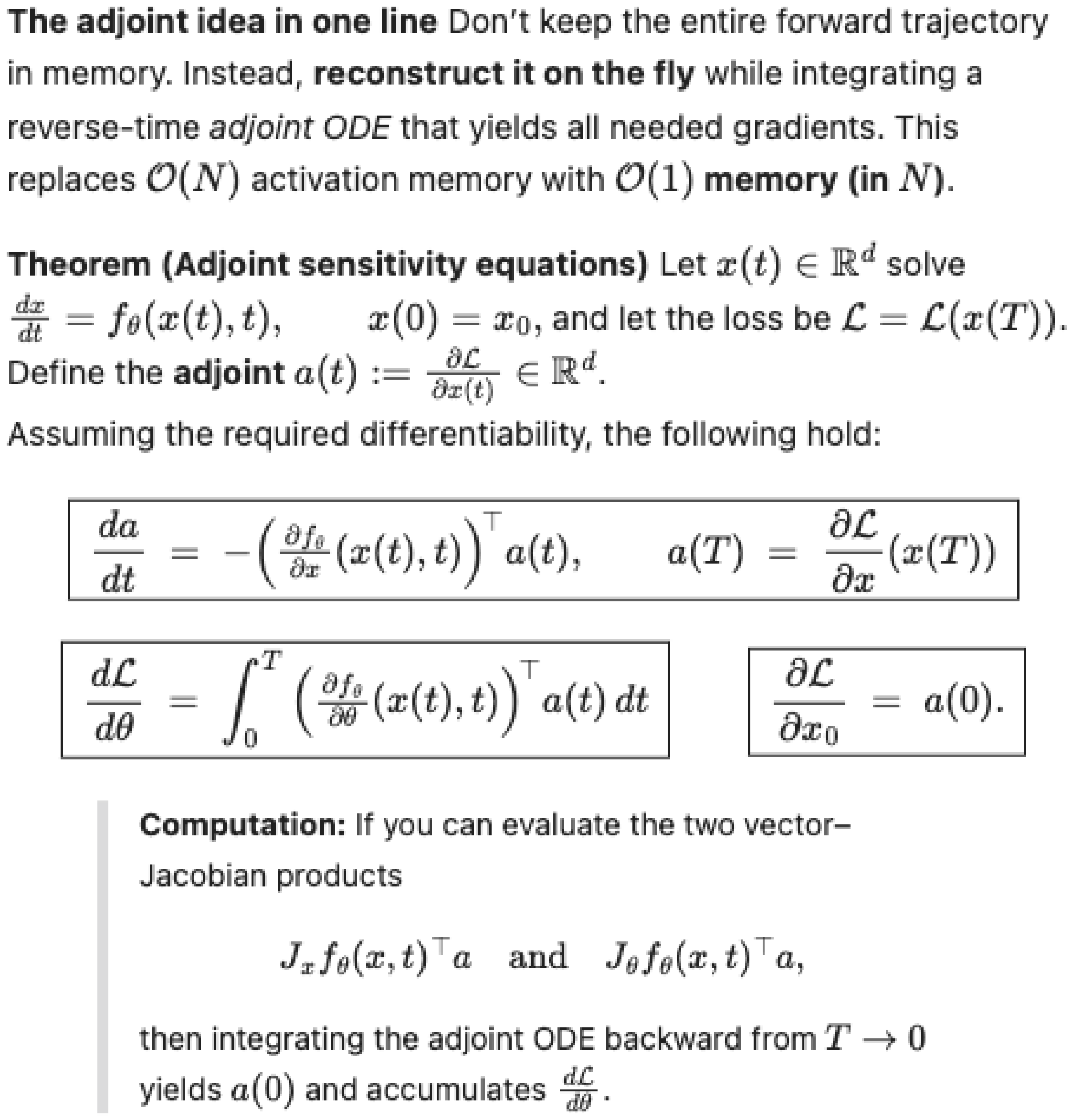
Units: $1\,\text{GiB}=1024\,\text{MiB}$.

Backprop: ResNet vs Neural ODE (Euler), naive/autograd



Aspect	ResNet (backprop)	Neural ODE + Euler (naïve backprop)
Forward primitive	$x_{\ell+1} = x_\ell + F_\ell(x_\ell)$	$x_{n+1} = x_n + \Delta t\, f_\theta(x_n, t_n)$
Depth driver	Fixed by architecture (≈50–100 layers).	Step count $N = T/\Delta t$ set by solver/tolerances; can be hundreds–thousands.
Parameters	Different θ_ℓ per layer (no sharing).	Often shared θ across steps (time-stationary) or time-dependent $\theta(t)$.
What autograd saves per unit	State x_ℓ plus internal activations of F_ℓ.	State x_n plus all internal activations of f_θ **for every step**.
Memory pressure	Many layers with large activations, but modest depth.	**Large** N multiplies state + activations even when f_θ is small.
Typical failure mode	OOM from very wide/high-res early layers.	OOM from $N \times (\text{state} + \text{activations})$ when N is large.

Adjoint Sensitivity Method



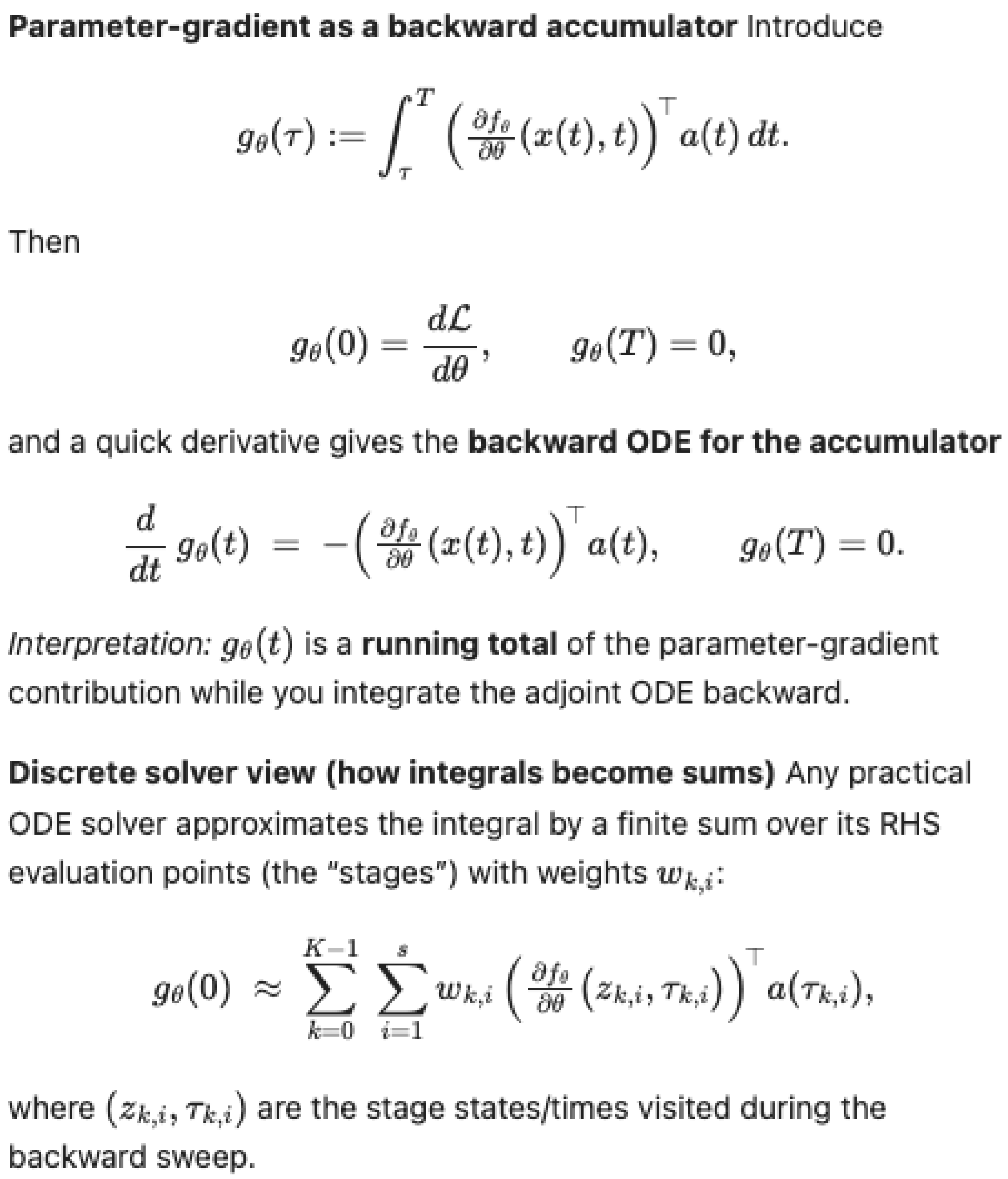
The adjoint idea in one line Don't keep the entire forward trajectory in memory. Instead, **reconstruct it on the fly** while integrating a reverse-time *adjoint ODE* that yields all needed gradients. This replaces $\mathcal{O}(N)$ activation memory with $\mathcal{O}(1)$ **memory (in N)**.

Theorem (Adjoint sensitivity equations) Let $x(t) \in \mathbb{R}^d$ solve $\frac{dx}{dt} = f_\theta(x(t), t), \qquad x(0) = x_0$, and let the loss be $\mathcal{L} = \mathcal{L}(x(T))$.
Define the **adjoint** $a(t) := \frac{\partial \mathcal{L}}{\partial x(t)} \in \mathbb{R}^d$.
Assuming the required differentiability, the following hold:

$$\boxed{\frac{da}{dt} \;=\; -\Big(\tfrac{\partial f_\theta}{\partial x}(x(t), t)\Big)^{\top} a(t), \qquad a(T) \;=\; \frac{\partial \mathcal{L}}{\partial x}(x(T))}$$

$$\boxed{\frac{d\mathcal{L}}{d\theta} \;=\; \int_0^T \Big(\tfrac{\partial f_\theta}{\partial \theta}(x(t), t)\Big)^{\top} a(t)\, dt} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial x_0} \;=\; a(0).}$$

> **Computation:** If you can evaluate the two vector–Jacobian products
>
> $$J_x f_\theta(x, t)^\top a \quad \text{and} \quad J_\theta f_\theta(x, t)^\top a,$$
>
> then integrating the adjoint ODE backward from $T \to 0$ yields $a(0)$ and accumulates $\frac{d\mathcal{L}}{d\theta}$.



Parameter-gradient as a backward accumulator Introduce

$$g_\theta(\tau) := \int_\tau^T \Big(\tfrac{\partial f_\theta}{\partial \theta}(x(t), t)\Big)^{\top} a(t)\, dt.$$

Then

$$g_\theta(0) = \frac{d\mathcal{L}}{d\theta}, \qquad g_\theta(T) = 0,$$

and a quick derivative gives the **backward ODE for the accumulator**

$$\frac{d}{dt}\, g_\theta(t) \;=\; -\Big(\tfrac{\partial f_\theta}{\partial \theta}(x(t), t)\Big)^{\top} a(t), \qquad g_\theta(T) = 0.$$

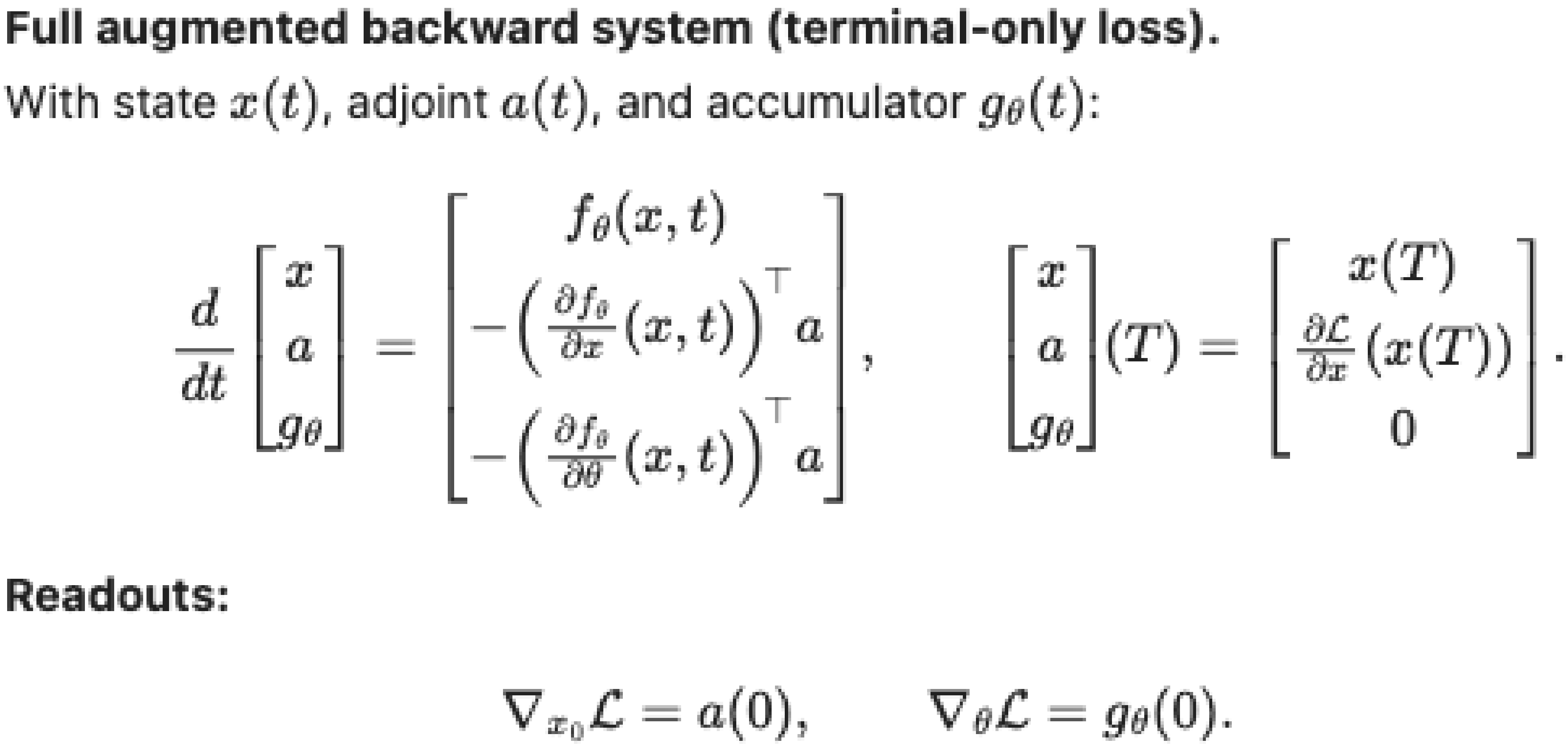
Interpretation: $g_\theta(t)$ is a **running total** of the parameter-gradient contribution while you integrate the adjoint ODE backward.

Discrete solver view (how integrals become sums) Any practical ODE solver approximates the integral by a finite sum over its RHS evaluation points (the "stages") with weights $w_{k,i}$:

$$g_\theta(0) \;\approx\; \sum_{k=0}^{K-1} \sum_{i=1}^{s} w_{k,i} \Big(\tfrac{\partial f_\theta}{\partial \theta}\,(z_{k,i}, \tau_{k,i})\Big)^{\top} a(\tau_{k,i}),$$

where $(z_{k,i}, \tau_{k,i})$ are the stage states/times visited during the backward sweep.

Augmented Adjoint based ODE



Full augmented backward system (terminal-only loss).

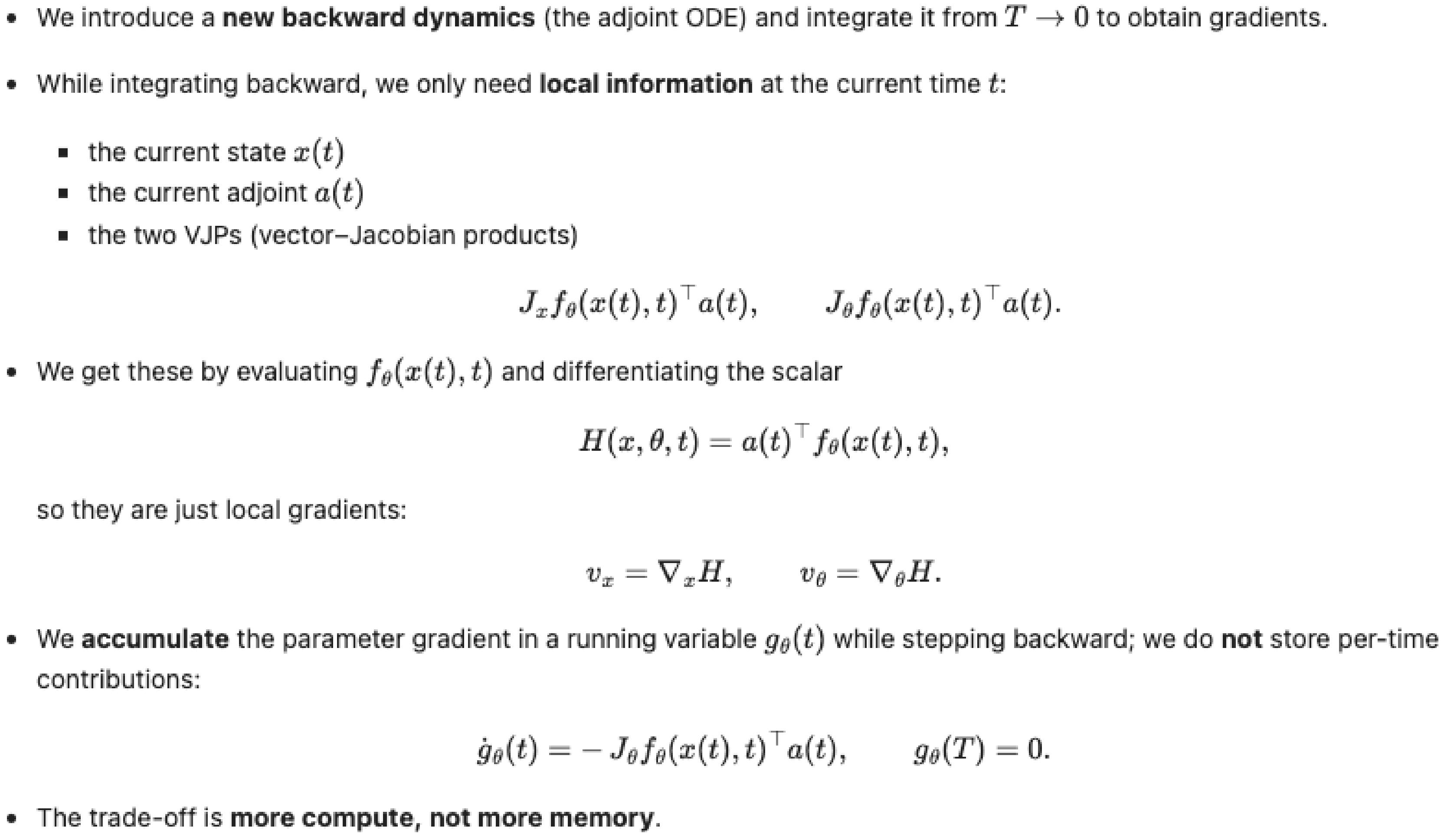
With state $x(t)$, adjoint $a(t)$, and accumulator $g_\theta(t)$:

$$\frac{d}{dt}\begin{bmatrix} x \\ a \\ g_\theta \end{bmatrix} = \begin{bmatrix} f_\theta(x,t) \\ -\left(\frac{\partial f_\theta}{\partial x}(x,t)\right)^\top a \\ -\left(\frac{\partial f_\theta}{\partial \theta}(x,t)\right)^\top a \end{bmatrix}, \qquad \begin{bmatrix} x \\ a \\ g_\theta \end{bmatrix}(T) = \begin{bmatrix} x(T) \\ \frac{\partial \mathcal{L}}{\partial x}(x(T)) \\ 0 \end{bmatrix}.$$

Readouts:

$$\nabla_{x_0}\mathcal{L} = a(0), \qquad \nabla_\theta \mathcal{L} = g_\theta(0).$$

Why the adjoint method is memory efficient?



- We introduce a **new backward dynamics** (the adjoint ODE) and integrate it from $T \to 0$ to obtain gradients.

- While integrating backward, we only need **local information** at the current time t:

  - the current state $x(t)$
  - the current adjoint $a(t)$
  - the two VJPs (vector–Jacobian products)

$$J_x f_\theta(x(t),t)^\top a(t), \qquad J_\theta f_\theta(x(t),t)^\top a(t).$$

- We get these by evaluating $f_\theta(x(t),t)$ and differentiating the scalar

$$H(x,\theta,t) = a(t)^\top f_\theta(x(t),t),$$

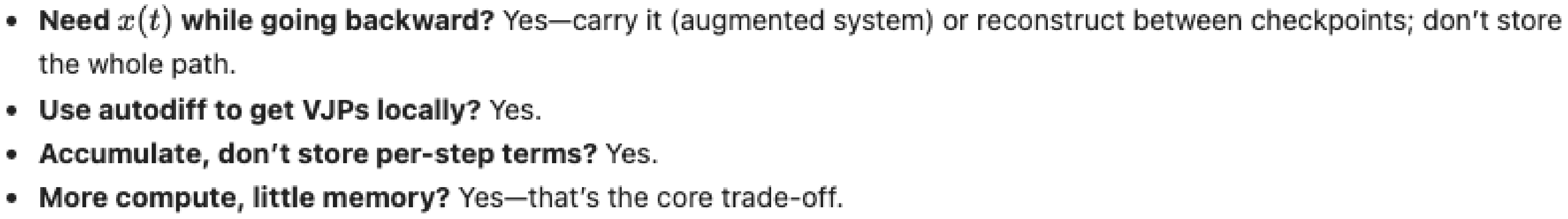
  so they are just local gradients:

$$v_x = \nabla_x H, \qquad v_\theta = \nabla_\theta H.$$

- We **accumulate** the parameter gradient in a running variable $g_\theta(t)$ while stepping backward; we do **not** store per-time contributions:

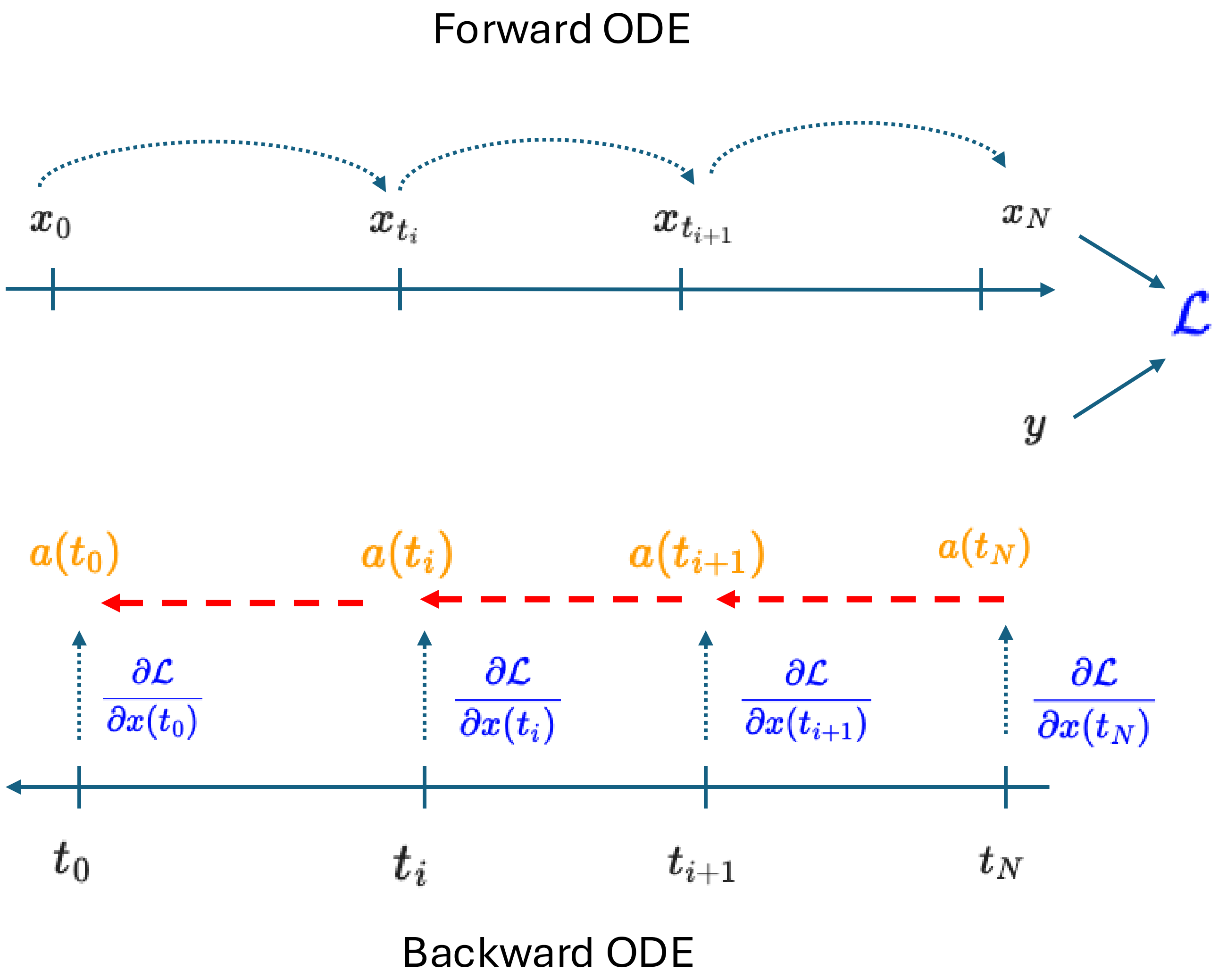
$$\dot{g}_\theta(t) = -\,J_\theta f_\theta(x(t),t)^\top a(t), \qquad g_\theta(T) = 0.$$

- The trade-off is **more compute, not more memory**.



- **Need $x(t)$ while going backward?** Yes—carry it (augmented system) or reconstruct between checkpoints; don't store the whole path.
- **Use autodiff to get VJPs locally?** Yes.
- **Accumulate, don't store per-step terms?** Yes.
- **More compute, little memory?** Yes—that's the core trade-off.

Illustration


Forward ODE

Backward ODE

x_0
x_{t_i}
$x_{t_{i+1}}$
x_N
$\mathcal{L}$
y
$a(t_0)$
$a(t_i)$
$a(t_{i+1})$
$a(t_N)$
$\frac{\partial \mathcal{L}}{\partial x(t_0)}$
$\frac{\partial \mathcal{L}}{\partial x(t_i)}$
$\frac{\partial \mathcal{L}}{\partial x(t_{i+1})}$
$\frac{\partial \mathcal{L}}{\partial x(t_N)}$
t_0
t_i
t_{i+1}
t_N

Memory vs Compute tradeoff

Aspect	Adjoint Method	Naïve Backprop
X(t) access	Recomputed via solver	Stored during forward pass
Memory cost	Constant (w.r.t time steps)	Linear in solver steps
Compute cost	Higher (re-solving ODE)	Lower
Accuracy	May suffer from numerical error	High (exact gradient via graph)

- So, yes, we do need x(t) – but we get it by recomputing it, not by storing it.
- This what makes the adjoint method so clever:
    - It solves the memory issue by turning it into a compute problem – recompute instead of remember.

References

- Source: Kaiming He et al (2015), Deep Residual Learning for image recognition, Microsoft Research
- Ricky Chen et al , Neural Ordinary Differential Equations