AI504: Programming for Artificial Intelligence

Week 14: Neural Ordinary Differential Equations

Edward Choi
Grad School of AI
edwardchoi@kaist.ac.kr

Index

- Ordinary Differential Equations
  - First order ODE
  - Initial value problem
  - How to solve ODE
- Neural ODE
  - Adjoint sensitivity method

Ordinary Differential Equations

Differential Equation

- 1st order DE: any problem formulation that follows

$$\frac{dy}{dt} = f(y, t)$$

Differential Equation

- 1st order DE: any problem formulation that follows

$$\frac{dy}{dt} = f(y,t)$$ ➡

You don’t know how to calculate y

You know how to calculate the change of y

Differential Equation

- 1st order DE: any problem formulation that follows

$$\frac{dy}{dt} = f(y,t)$$ ➡

You don't know how to calculate y

You know how to calculate the change of y

- Many types of 1st-order differential equations
  - 1st-order, linear DE ➔ $y' + p(t)y = g(t)$
  - Separable DE ➔ $p(y)y' = g(t)$
  - Bernoullie DE ➔ $y' + p(t)y = y^n$

Differential Equation

- 1st order DE: any problem formulation that follows

$$\frac{dy}{dt} = f(y,t)$$ ➡

You don't know how to calculate y

You know how to calculate the change of y

- Many types of 1st-order differential equations
  - 1st-order, linear DE ➔ $y' + p(t)y = g(t)$ ➔ $y' = g(t) - p(t)y$
  - Separable DE ➔ $p(y)y' = g(t)$ ➔ $y' = g(t) / p(y)$
  - Bernoullie DE ➔ $y' + p(t)y = y^n$ ➔ $y' = y^n - p(t)y$

ODE Example: Free-falling Object

- We want to know **the velocity** of a falling object **at time *t*.**

- We don't know how to calculate velocity at time *t* (v(*t*))
- Do we know how the velocity changes over time (v' = dv/dt)?

ODE Example: Free-falling Object

- We want to know the velocity of a falling object at time *t*.

- We don't know how to calculate velocity at time *t* (v(*t*))
- Do we know how the velocity changes over time (v' = dv/dt)?

  ➔ Newton's law of motion

ODE Example: Free-falling Object

- Newton's law of motion

$$F = ma$$

- Acceleration can be written as

$$a = \frac{dv}{dt}$$

- Force is a function of velocity and time

$$m\frac{dv}{dt} = F(t, v)$$

ODE Example: Free-falling Object

- Newton's law of motion

$$m\frac{dv}{dt} = F(t, v)$$

- Without air friction

$$m\frac{dv}{dt} = mg \quad (g = 9.8\, m/s^2)$$

- With air firction

$$m\frac{dv}{dt} = mg - \gamma v \Rightarrow \frac{dv}{dt} = g - \frac{\gamma v}{m} \Rightarrow \frac{dv}{dt} = 9.8 - 0.196v$$

ODE Example: Free-falling Object

- 1st-order differential equation

$$\frac{dv}{dt} = 9.8 - 0.196v$$

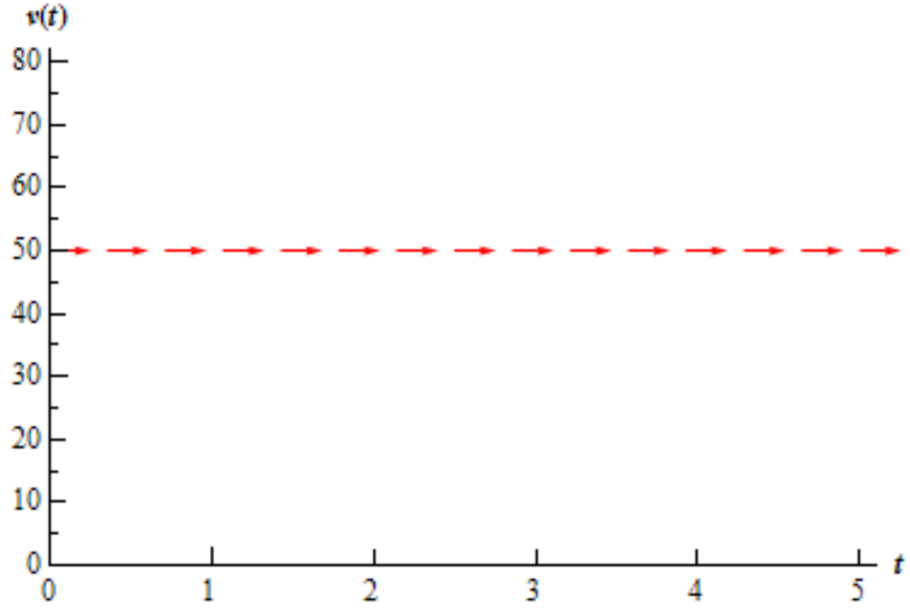
- What if v(t) = 50 m/s?

ODE Example: Free-falling Object

- 1st-order differential equation

$$\frac{dv}{dt} = 9.8 - 0.196v$$

- What if v(t) = 50 m/s ➔ v’ = 0 ➔ No change to velocity

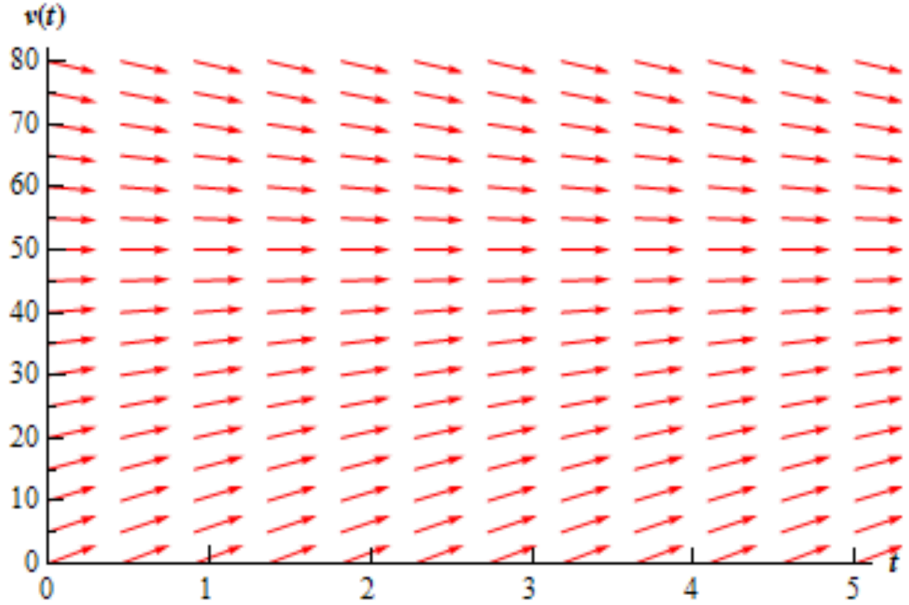


ODE Example: Free-falling Object

- 1st-order differential equation

$$\frac{dv}{dt} = 9.8 - 0.196v$$

- We can draw the vector field of DE

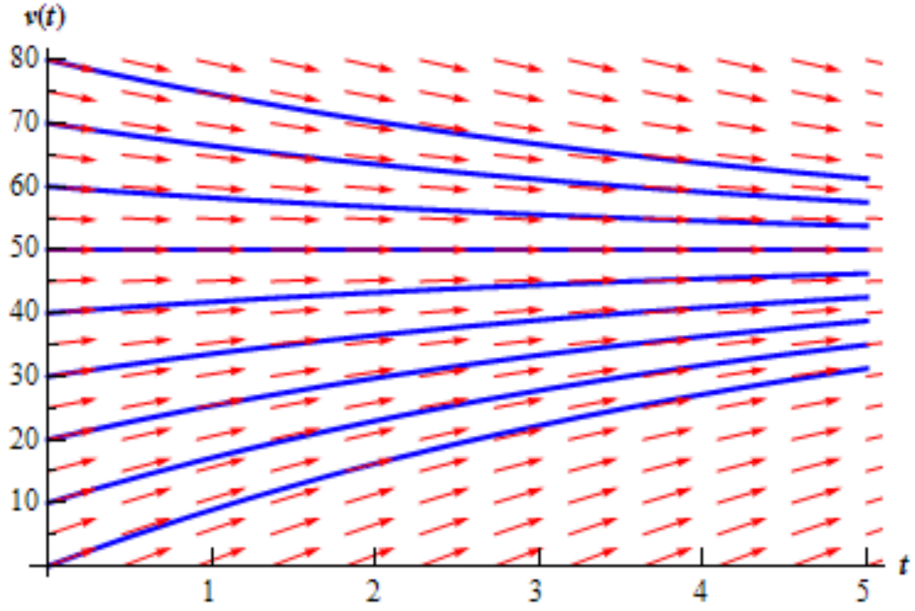


ODE Example: Free-falling Object

- 1st-order differential equation

$$\frac{dv}{dt} = 9.8 - 0.196v$$

- Velocity trends according to different initial conditions:



ODE Example: Free-falling Object

- 1st-order differential equation

$$\frac{dv}{dt} = 9.8 - 0.196v$$

- We can analytically solve this:

$$v(t) = 50 + c\mathbf{e}^{-0.196t}$$

- The general solution contains an unknown constant *c*
  - We need an initial condition
- Differential Equation + Initial Condition ➔ **Initial Value Problem (IVP)**

ODE Example: Free-falling Object

- Initial value problem

$$\frac{dv}{dt} = 9.8 - 0.196v \qquad v(0) = 48$$

- Final analytical solution:

$$v = 50 - 2\mathbf{e}^{-0.196t}$$

Numerical Solution

- Initial value problem

$$\frac{dv}{dt} = 9.8 - 0.196v \qquad v(0) = 48$$

- What if we can't solve analytically?

  ➔ We can solve numerically

Euler's Method

- Numerical solution to IVPs
- Initial value problem

$$\frac{dy}{dt} = f(t, y) \qquad y(t_0) = y_0$$

- Slope of the solution at $t = t_0$

$$\left.\frac{dy}{dt}\right|_{t=t_0} = f(t_0, y_0)$$

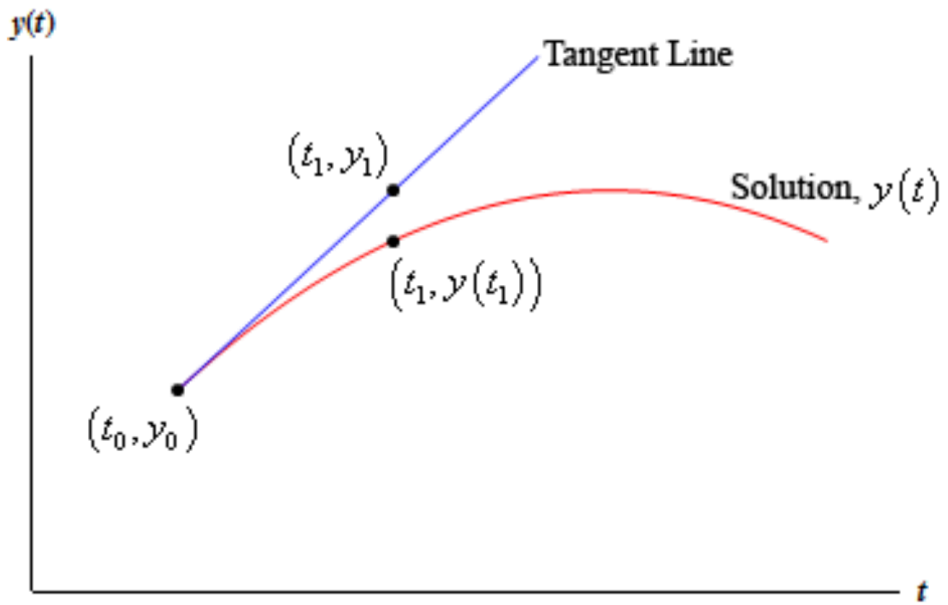
- Tangent line to the solution at $t = t_0$

$$y = y_0 + f(t_0, y_0)(t - t_0)$$

Euler's Method

- Initial value problem $\dfrac{dy}{dt} = f(t, y) \qquad y(t_0) = y_0$
- Tangent line to the solution at $t = t_0$

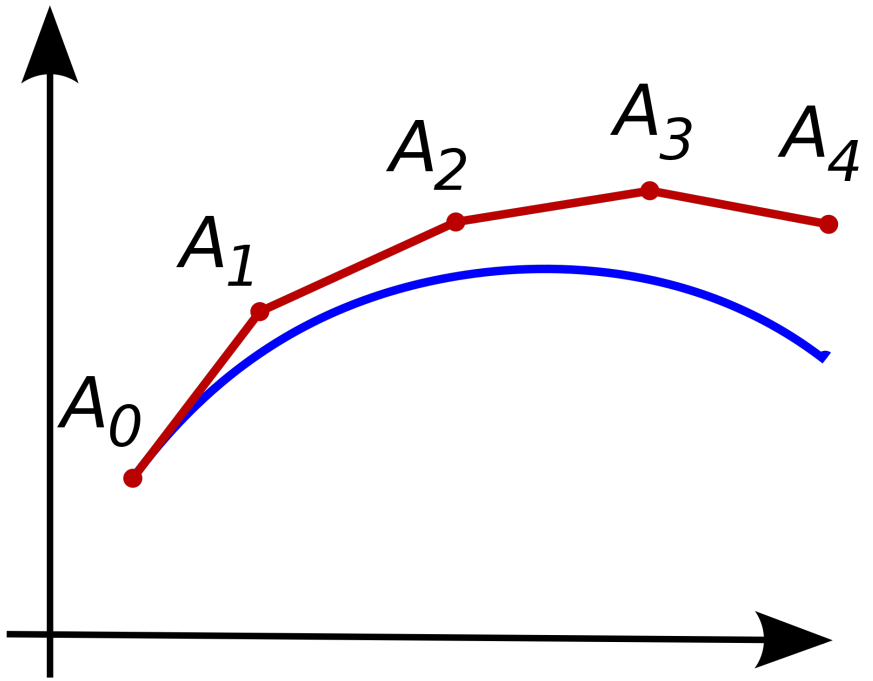
$$y = y_0 + f(t_0, y_0)(t - t_0)$$



Euler's Method

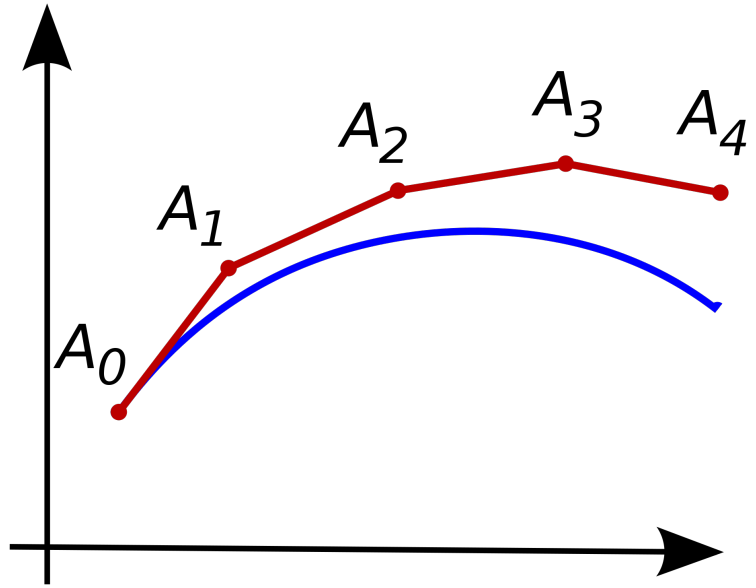
- Fixed-step algorithm

1. **define** $f(t, y)$.
2. **input** t_0 and y_0.
3. **input** step size, h and the number of steps, n.
4. **for** j from 1 to n **do**
   - **(a)** $m = f(t_0, y_0)$
   - **(b)** $y_1 = y_0 + h * m$
   - **(c)** $t_1 = t_0 + h$
   - **(d)** Print t_1 and y_1
   - **(e)** $t_0 = t_1$
   - **(f)** $y_0 = y_1$
5. **end**



Euler's Method

- Numerical integration

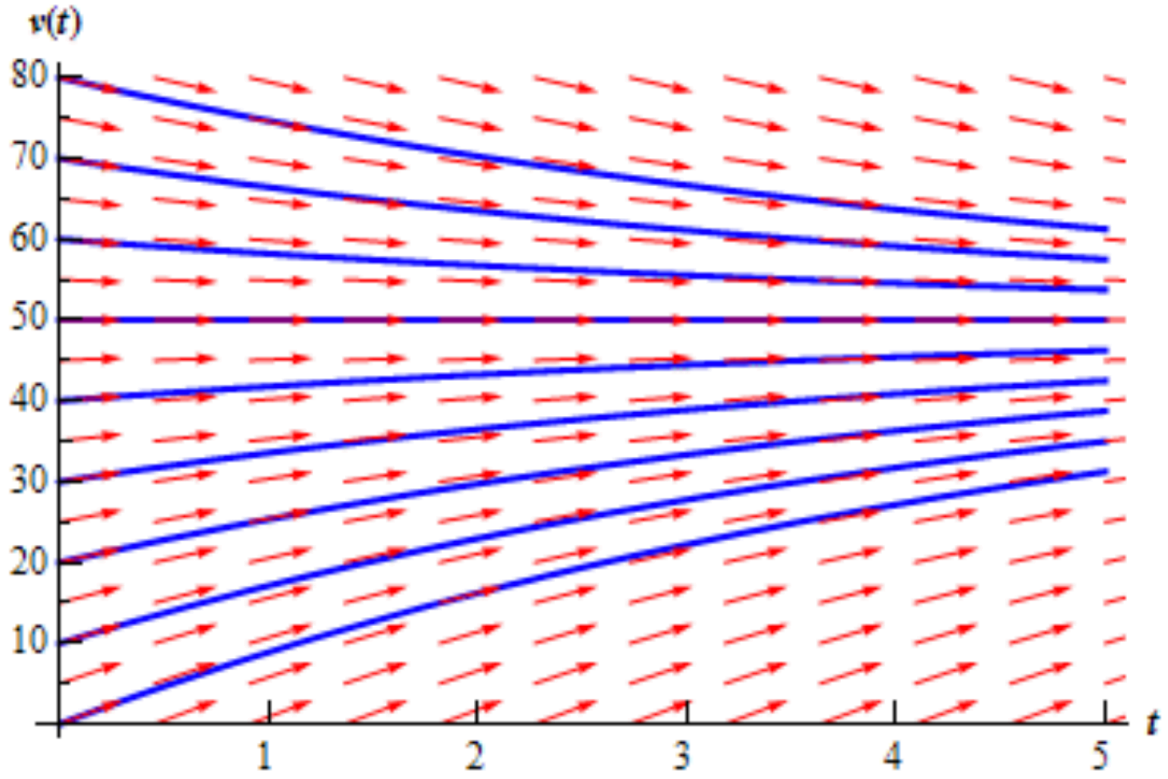


$$y_T = y_0 + \int_0^T f(y_t, t)\, dt = \text{ODESolve}(y_0, f, [0, T])$$

Euler's Method

- If we have a vector field and an initial condition,

  ➔ We can numerically solve the problem with a computer



Runge-Kutta Method

- Better precision than Euler's method
- Given IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

Euler's Method

$$y_{n+1} = y_n + hf(t_n, y_n)$$

Runge-Kutta (4th Order)

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\ k_4 &= f\left(t_n + h, y_n + hk_3\right). \end{aligned}$$

Runge-Kutta Method

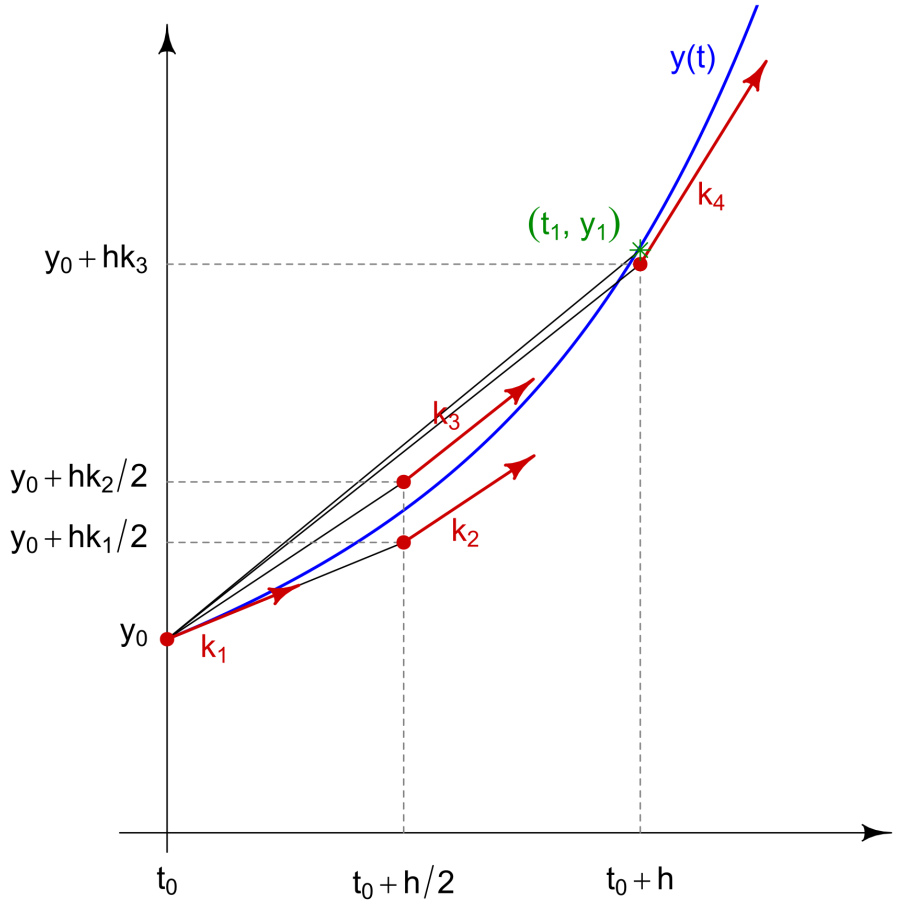
- Better precision than Euler's method
- Given IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

- RK4

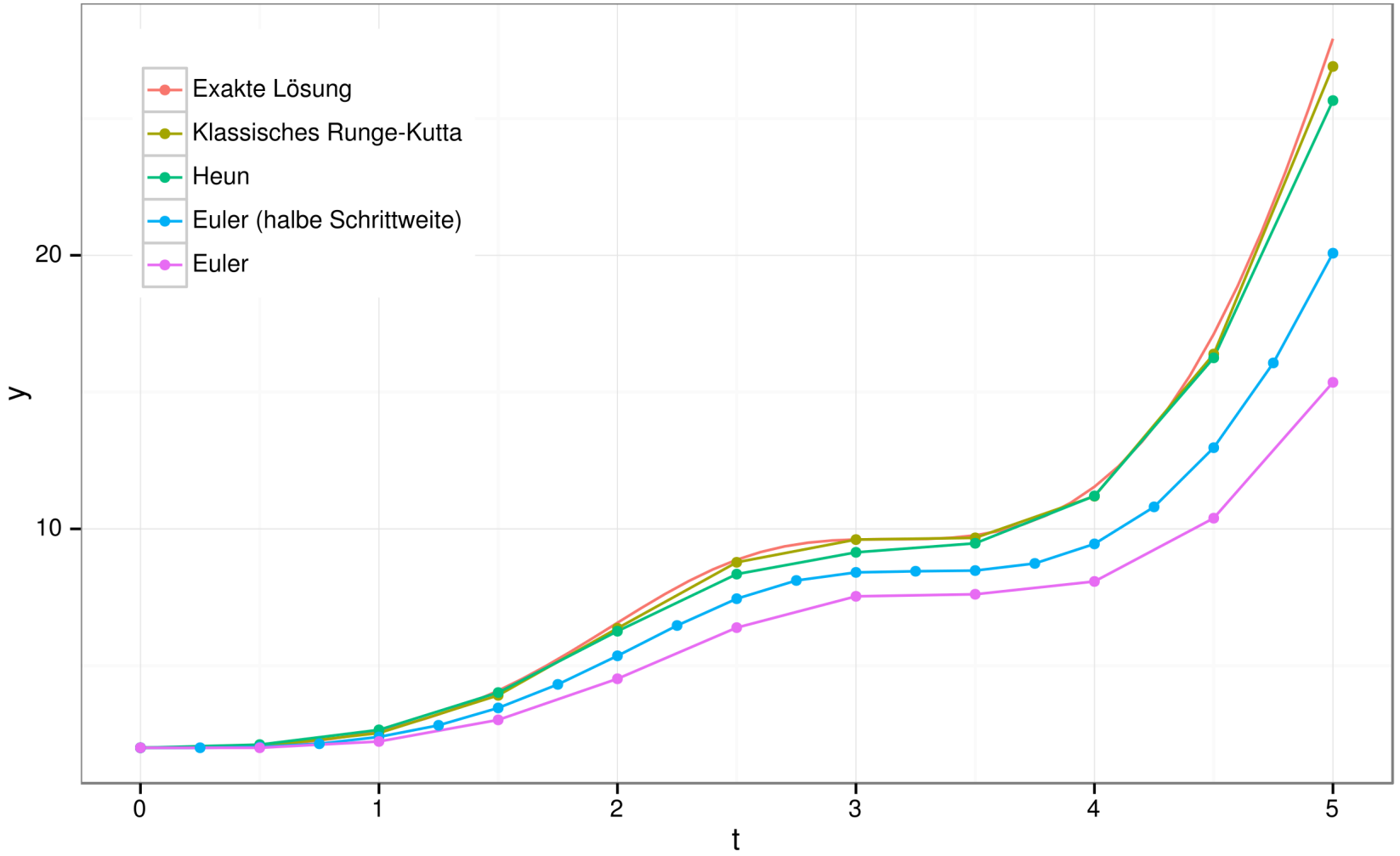
$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\ k_4 &= f\left(t_n + h, y_n + hk_3\right). \end{aligned}$$



RK4 vs Euler's Method

- $y' = y\sin(t)^2$
  - Orange line



ODE Solvers

- Long history in mathematics & physics
- Fixed step size solvers
  - Euler
  - Midpoint
  - Runge-Kutta
  - Adams-Bashforth
- Adaptive step size solvers
  - Dormand-Prince
  - Dormand-Prince-Shampine
  - Bogacki-Shampine

ODE Solvers

- Long history in mathematics & physics
- Fixed step size solvers
  - Euler
  - Midpoint
  - Runge-Kutta
  - Adams-Bashforth

User decides when to evaluate f() and take a step.
(Need to explicitly tell ODESolve Δt)

- Adaptive step size solvers
  - Dormand-Prince
  - Dormand-Prince-Shampine
  - Bogacki-Shampine

Algorithm decides when to evaluate f() and take a step.
(Smooth dynamics ➔ large steps
Varying dynamics ➔ small steps)

2nd-Order Differential Equation

- General form 2nd-order DE
  - $p(t)y'' + q(t)y' + r(t)y = g(t)$
- Usually we deal with constant coefficients
  - $ay'' + by' + cy = 0$
  - $ay'' + by' + cy = d$
  - $ay'' + by' + cy = g(t)$
- Can also be solved numerically

Neural ODE

ResNet

- $\mathbf{h}_{t+1} = \text{ReLU}(W_t\mathbf{h}_t + b_t) + \mathbf{h}_t$

ResNet

- $\mathbf{h}_{t+1} = f(\mathbf{h}_t, \theta_t) + \mathbf{h}_t$

ResNet

- $\mathbf{h}_{t+1} = f(\mathbf{h}_t, \theta_t) + \mathbf{h}_t$

➔ $\mathbf{h}_{t+1} - \mathbf{h}_t = f(\mathbf{h}_t, \theta_t)$

➔ $\frac{\mathbf{h}_{t+1} - \mathbf{h}_t}{1} = f(\mathbf{h}_t, \theta_t)$

➔ $\left.\frac{\mathbf{h}_{t+\Delta} - \mathbf{h}_t}{\Delta}\right|_{\Delta=1} = f(\mathbf{h}_t, \theta_t)$

➔ $\lim_{\Delta \to 0} \frac{\mathbf{h}_{t+\Delta} - \mathbf{h}_t}{\Delta} = f(\mathbf{h}_t, t, \theta)$

➔ $\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$

ResNet: Discrete vs Continuous

$$\mathbf{h}_{t+1} = f(\mathbf{h}_t, \theta_t) + \mathbf{h}_t$$

- *L* discrete layers
- Latent state changes discretely
- Latent state dynamics controlled by *L* functions

VS

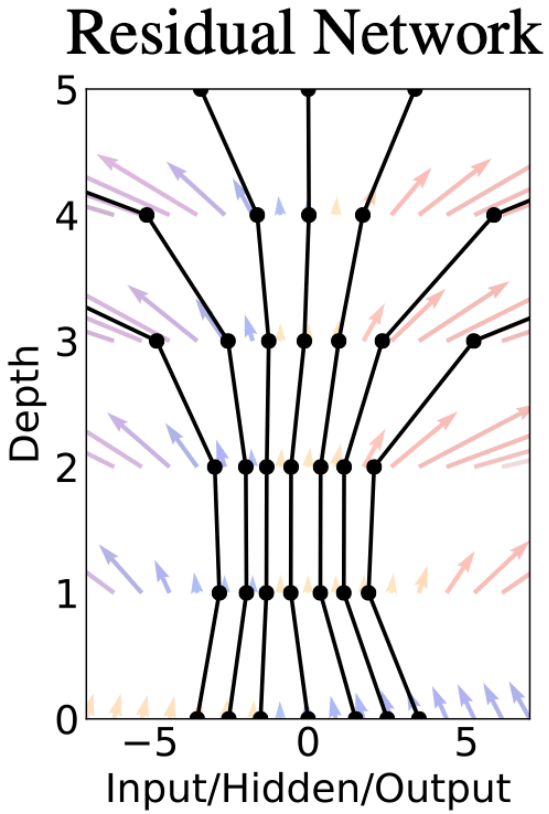
$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$$

- Inifinte layers
- Latent state changes continuously
- Latent state dynamics controlled by one function

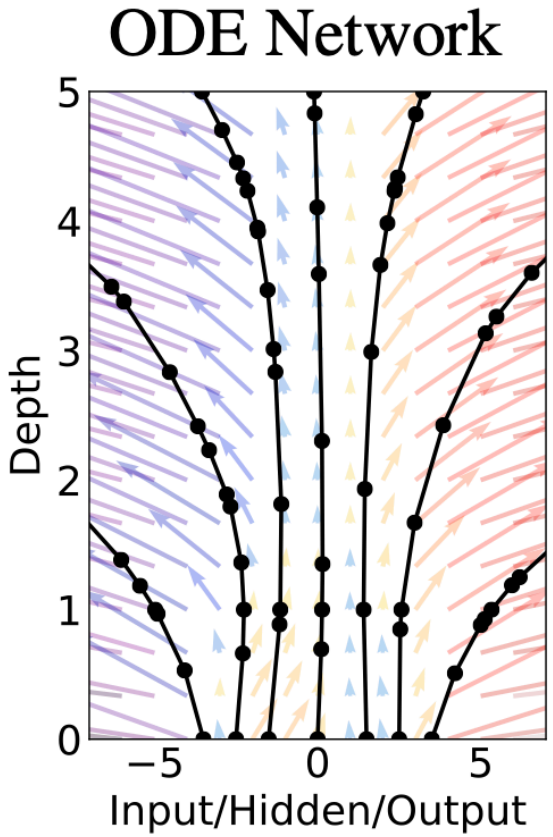
ResNet: Vector Field

$$\mathbf{h}_{t+1} = f(\mathbf{h}_t, \theta_t) + \mathbf{h}_t$$

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$$



VS



Recurrent Neural Network

- $\mathbf{h}_{t+1} = f(\mathbf{U}\mathbf{h}_t + \mathbf{W}\mathbf{x}_{t+1} + \mathbf{b})$

➔ $\mathbf{h}_{t+1} = \mathbf{h}_t + \mathbf{W}\mathbf{x}_{t+1} + \mathbf{b} \quad (\mathbf{U} = \mathbf{I}, f = \mathbf{1})$

➔ $\mathbf{h}_{t+1} = \mathbf{W}\mathbf{x}_{t+1} + \mathbf{b} + \mathbf{h}_t$

➔ $\mathbf{h}_{t+1} = f(\mathbf{h}_t, \mathbf{x}_{t+1}, t, \theta) + \mathbf{h}_t$

Neural ODE

- $\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$
- $f(\mathbf{h}(t), t, \theta)$ ➔ Neural Network

- What is the meaning of this in terms of ODE?

Neural ODE

- $\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$
- $f(\mathbf{h}(t), t, \theta)$ ➔ Neural Network

- What is the meaning of this in terms of ODE?

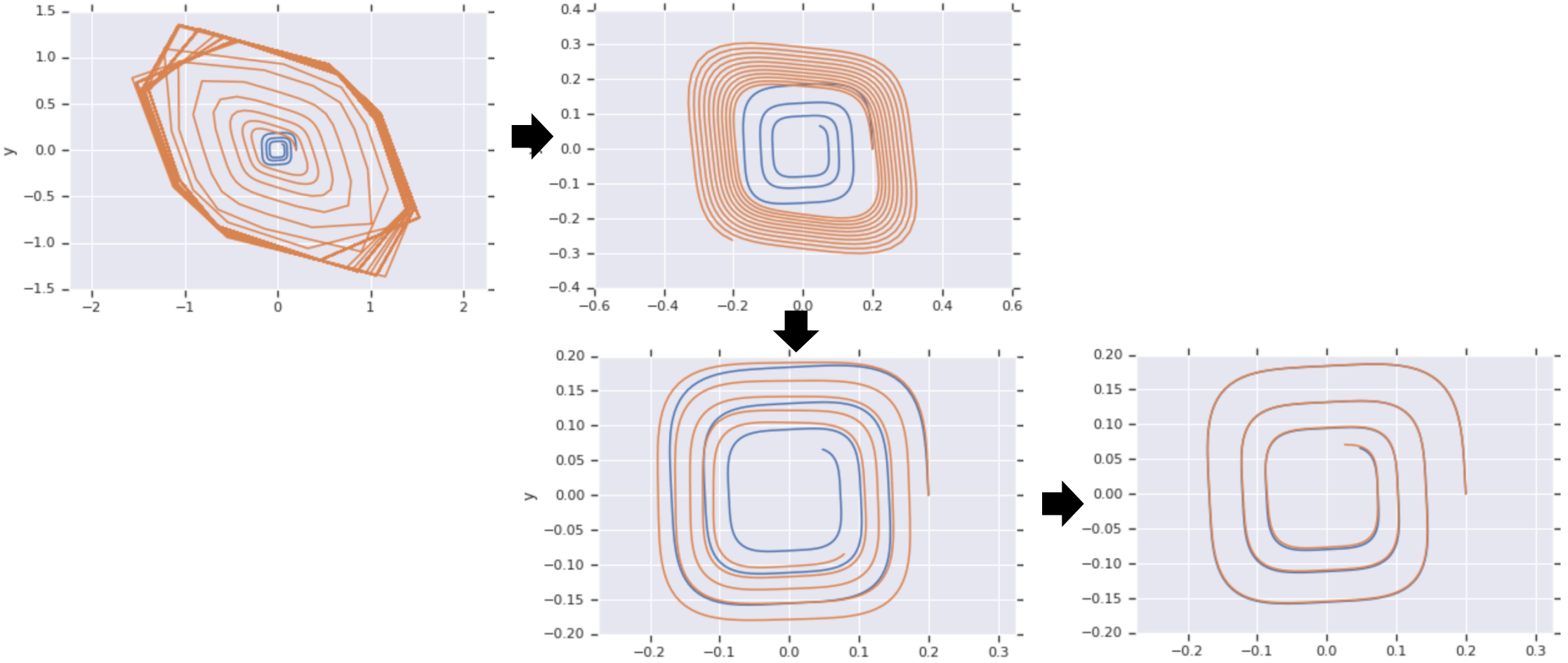
➔ We have an ODE problem

We do not know *y'*

We want to learn *y'* from data via NN + BackProp

Neural ODE

- Spiral example



Neural ODE: Forward Propagation

- Input state: $\mathbf{h}(0)$
- State dynamics

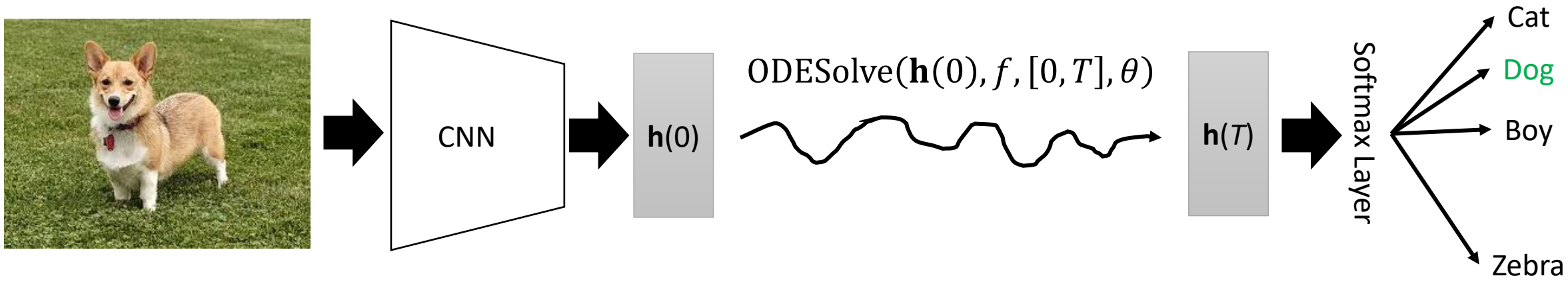
$$\frac{d\mathbf{h}(t)}{dt} = f_\theta(\mathbf{h}(t), t)$$ ➔ Typically just an MLP with some hidden layers

- Output state: $\mathbf{h}(T) = \mathbf{h}(0) + \int_0^T f_\theta(\mathbf{h}(t), t)dt$

$$\mathbf{h}(T) = \text{ODESolve}(\mathbf{h}(0), f, [0, T], \theta)$$

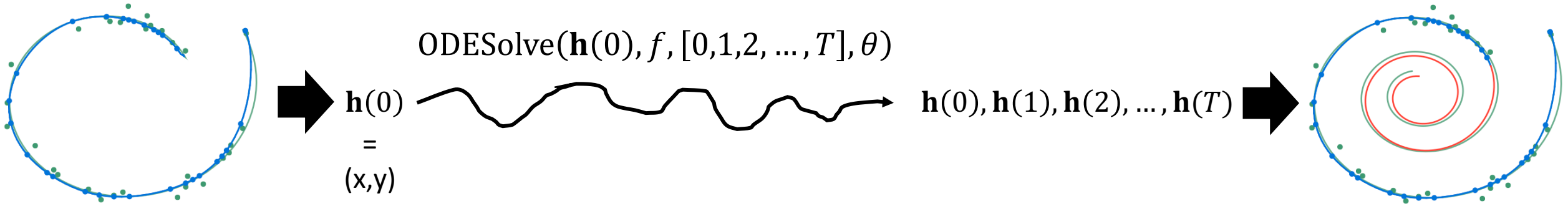
Neural ODE: Forward Propagation

- Image classification with Neural ODE



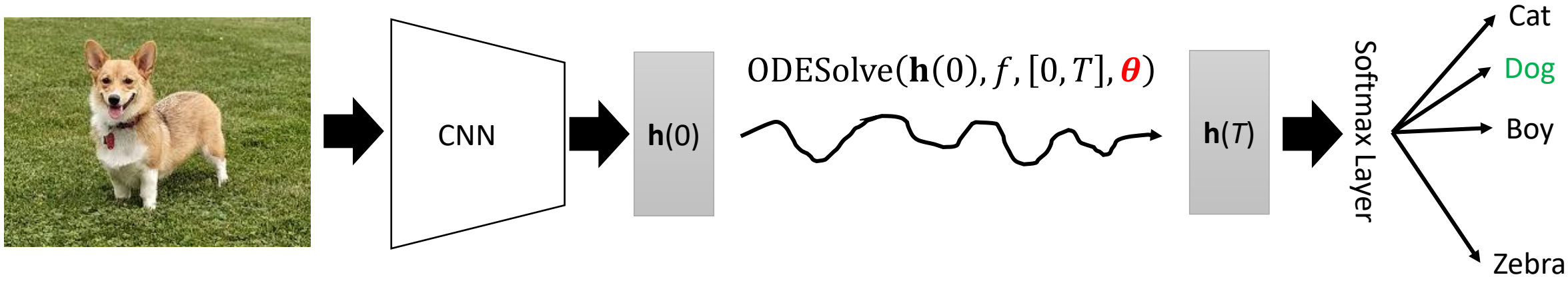
Neural ODE: Forward Propagation

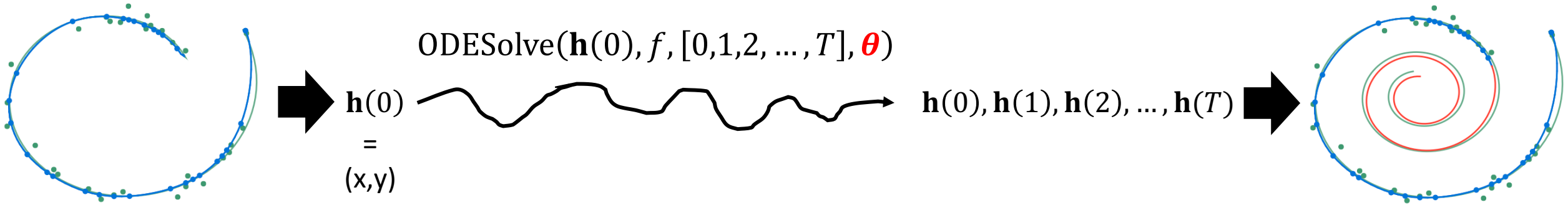
- Time-series Line Fitting



Neural ODE: Parameter Update

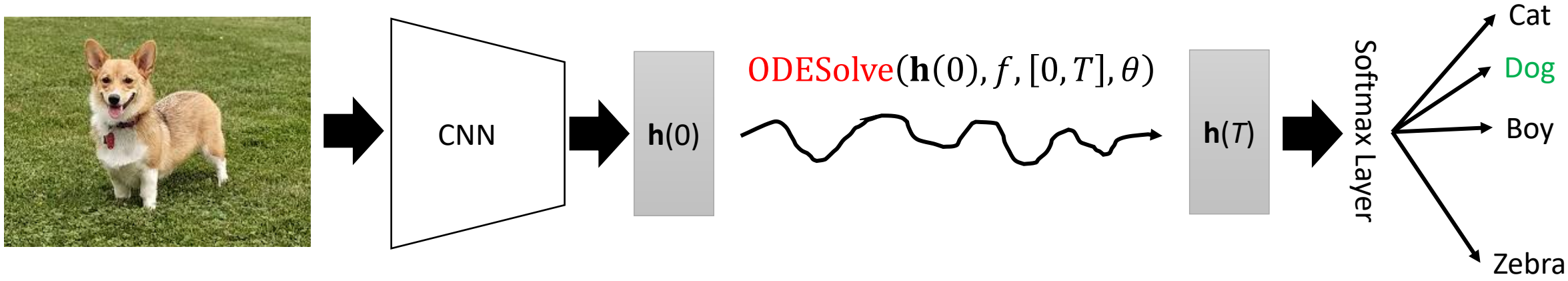
- Need to update $\boldsymbol{\theta}$ so that $f()$ learns the correct dynamics

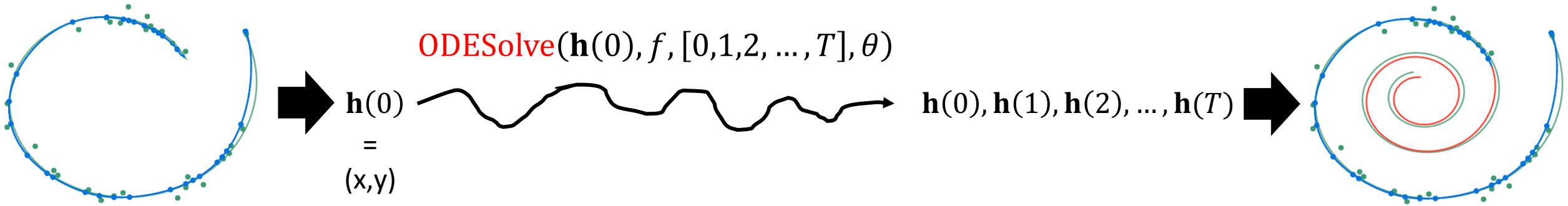




Neural ODE: Parameter Update

- Can we backprop through ODESolve??





Backprop through ODESolve

- For some solvers, we can backprop
  - Euler's Method
  - Runge-Kutta
  - Dopri5 (adaptive-step solver)
- For some solvers, it's probably not a good idea
  - Numerically unstable

Backprop through ODESolve

- For some solvers, we can backprop
  - Euler's Method
  - Runge-Kutta

  } We set the number of function evaluations (NFE)
  - Dopri5 (adaptive-step solver) → Solver adaptively adjusts NFE

- For some solvers, it's probably not a good idea
  - Numerically unstable

Backprop through ODESolve

- For some solvers, we can backprop
  - Euler's Method
  - Runge-Kutta

  } We set the number of function evaluations (NFE)
  - Dopri5 (adaptive-step solver) → Solver adaptively adjusts NFE
- The more NFE, the more accuate solution
  - What if NFE == 1 million?

- For some solvers, it's probably not a good idea
  - Numerically unstable

Backprop through ODESolve

- For some solvers, we can backprop
  - Euler's Method
  - Runge-Kutta

  } We set the number of function evaluations (NFE)
  - Dopri5 (adaptive-step solver) → Solver adaptively adjusts NFE
- The more NFE, the more accuate solution
  - What if NFE == 1 million? ➔ VRAM explode!!

- For some solvers, it's probably not a good idea
  - Numerically unstable

Backprop through ODESolve

- For some solvers, we can backprop
  - Euler's Method
  - Runge-Kutta

  } We set the number of function evaluations (NFE)
  - Dopri5 (adaptive-step solver) → Solver adaptively adjusts NFE
- The more NFE, the more accuate solution
  - What if NFE == 1 million? ➔ VRAM explode!!
  - ResNet is only 50 layers!
  - How can we use Neural ODE without using too much VRAM for backprop?
- For some solvers, it's probably not a good idea
  - Numerically unstable

Adjoint Sensitivity Method

- Necessary gradients
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(0)}$: Gradients of the loss w.r.t input state
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(1)}, \frac{\partial \mathcal{L}}{\partial \mathbf{h}(2)}$, … : (In time-seris) Gradients of the loss w.r.t intermediate states

Adjoint Sensitivity Method

- Necessary gradients
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(0)}$: Gradients of the loss w.r.t input state
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(1)}, \frac{\partial \mathcal{L}}{\partial \mathbf{h}(2)}$, … : (In time-seris) Gradients of the loss w.r.t intermediate states

  ➔Use **adjoint sensitivity method**

  Simply put, solve ODE backwards to obtain $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(t)}$

Adjoint Sensitivity Method

- Necessary gradients
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(0)}$: Gradients of the loss w.r.t input state
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(1)}, \frac{\partial \mathcal{L}}{\partial \mathbf{h}(2)}$, … : (In time-seris) Gradients of the loss w.r.t intermediate states

  ➔Use adjoint sensitivity method

  Simply put, solve ODE backwards to obtain $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(t)}$
  - $\frac{\partial \mathcal{L}}{\partial \theta}$: Gradients of the loss w.r.t dynamics function params

Adjoint Sensitivity Method

- Necessary gradients
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(0)}$: Gradients of the loss w.r.t input state
  - $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(1)}, \frac{\partial \mathcal{L}}{\partial \mathbf{h}(2)}$, … : (In time-seris) Gradients of the loss w.r.t intermediate states

  ➔Use **adjoint sensitivity method**

  Simply put, solve ODE backwards to obtain $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(t)}$
  - $\frac{\partial \mathcal{L}}{\partial \theta}$: Gradients of the loss w.r.t dynamics function params

  ➔ Solve $\frac{\partial \mathcal{L}}{\partial \theta}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(t)}$ **together at the same time!**

Adjoint Sensitiveity Method

- Final algorithm

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

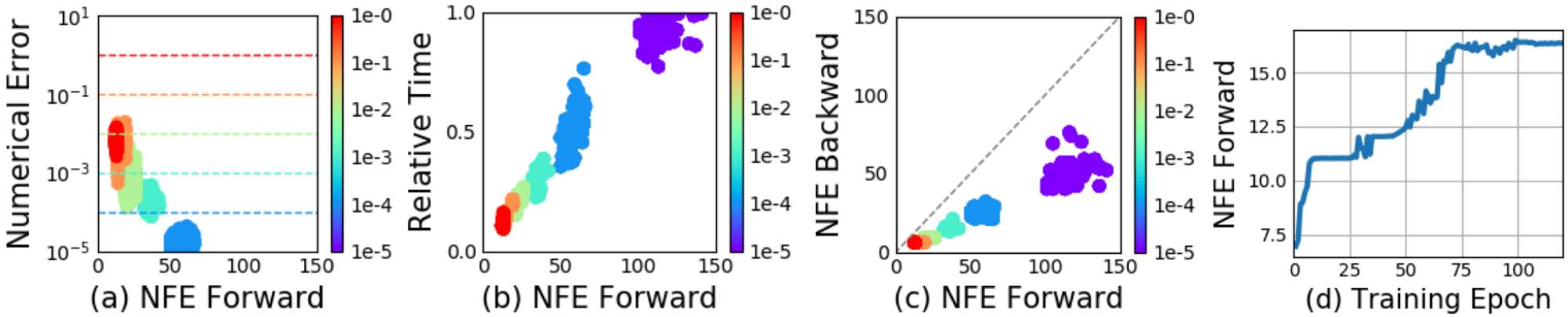
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Experiment: Image

- MNIST Classificiation

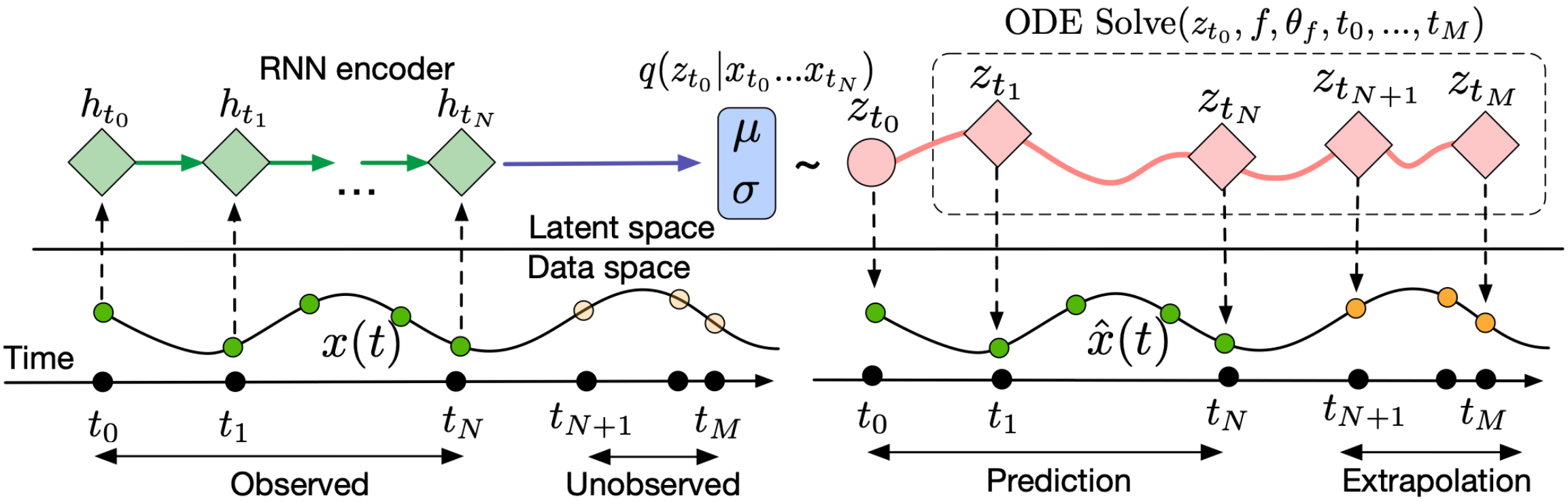
	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(\tilde{L})$

- Statistics of ODE-Net



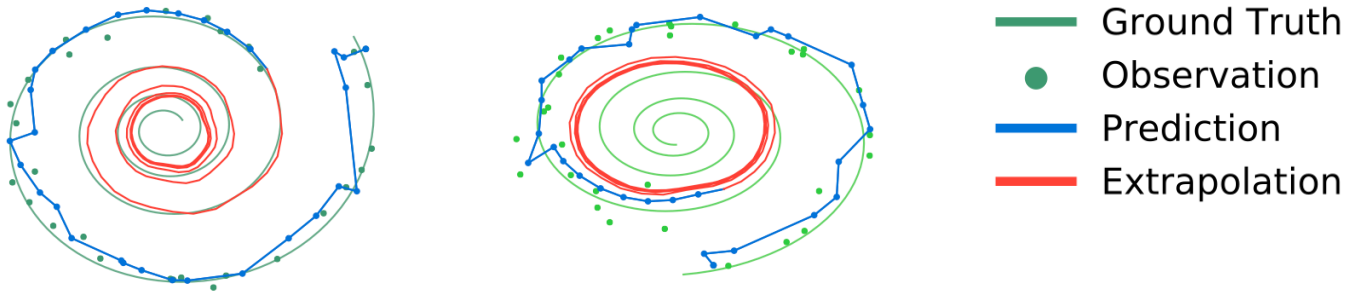
ODE Seq2Seq

- RNN Encoder & ODE Decoder

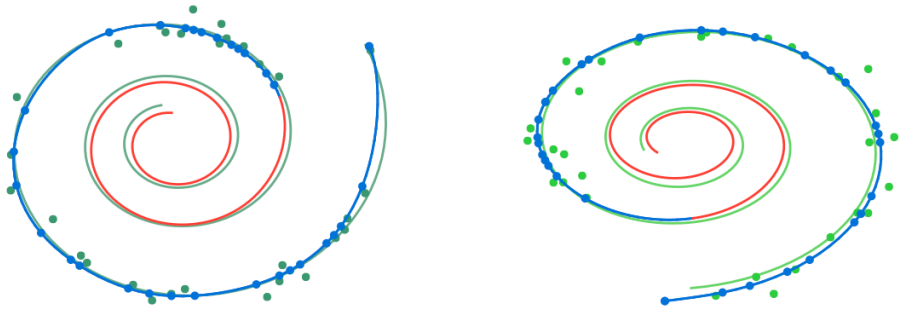


Experiment: Time-Series

- Using Seq2seq, perform interpolation & extrapolation



(a) Recurrent Neural Network



(b) Latent Neural Ordinary Differential Equation

Conclusion

- New deep learning paradigm
  - Great alternative to RNN when handling time
- Can use well-known, guaranteed ODE solvers
  - Modern solvers guarantee error tolerance
- Rapid development these days!

AI504: Programming for Artificial Intelligence

Week 14: Neural Ordinary Differential Equations

Edward Choi
Grad School of AI
edwardchoi@kaist.ac.kr