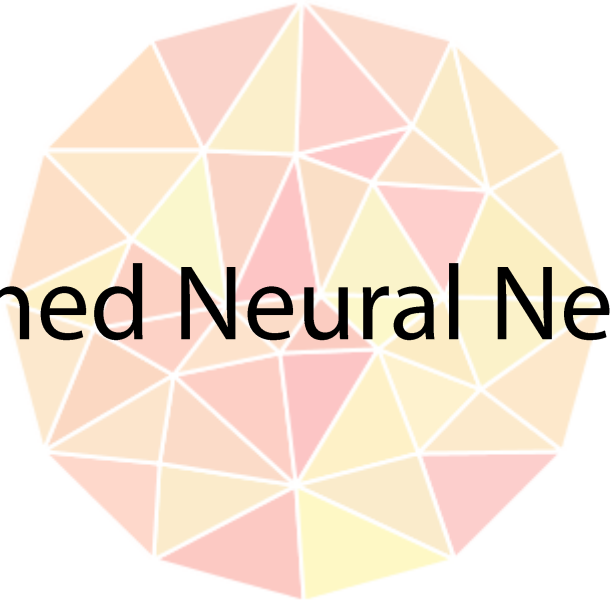
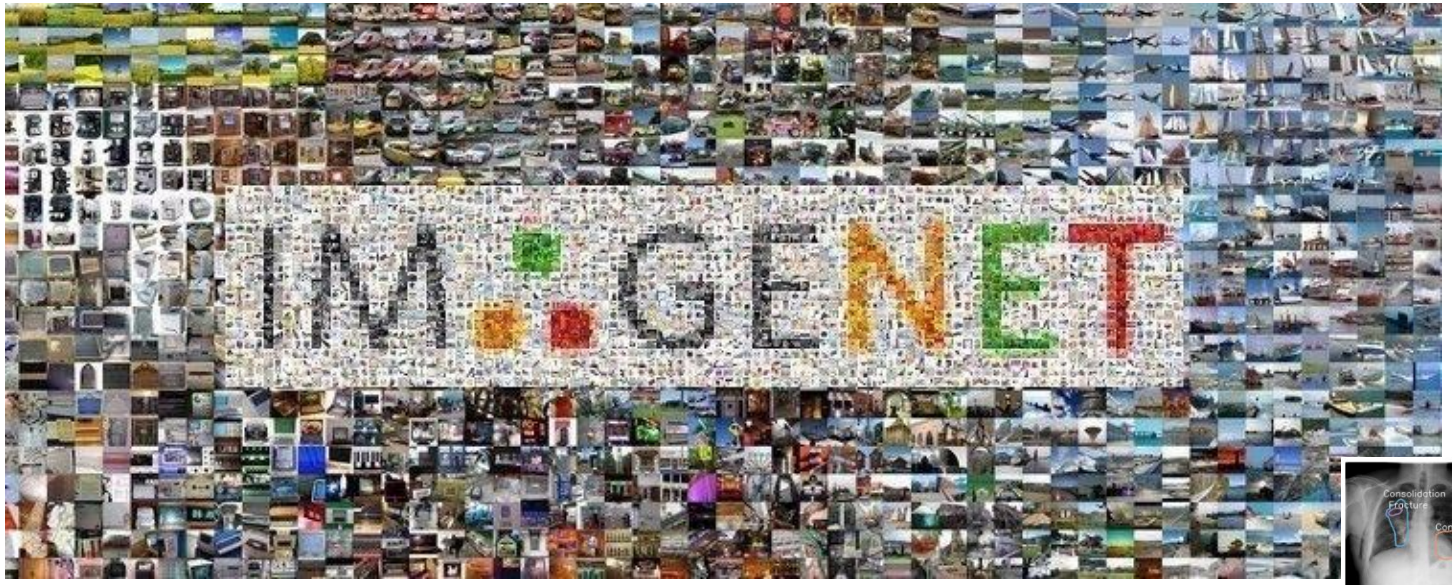
Physics-Informed Neural Networks (PINN)

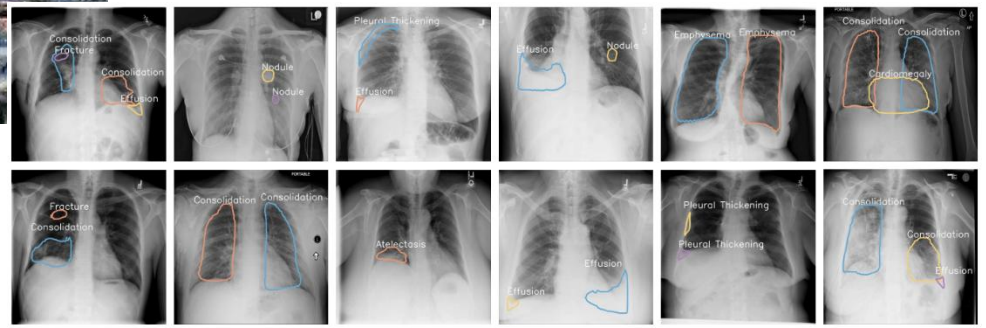


Stephen Baek

Fitting Neural Networks on Physics Data

- Neural networks are data hungry (deep networks, especially)





Fitting Neural Networks on Physics Data

- Neural networks are data hungry (deep networks, especially)
- In scientific applications, however, large data sets are not always available. For instance:
  - A complex fluid simulation (complex boundary conditions, fast flow speed, etc.) can take days on a supercomputer.
  - A material experiment may take hours to produce a specimen, configure experiment parameters, and perform an experiment.

  → These are just for one training sample!

Fitting Neural Networks on Physics Data

- Furthermore, in physics applications, we are not interested in simply just an input-output mapping. The input-output relationships must comply with e.g.:
  - Governing laws (e.g., conservation of mass, momentum, energy, etc.)
  - Initial/boundary conditions
  - Other constraints

Physics-Informed Neural Networks

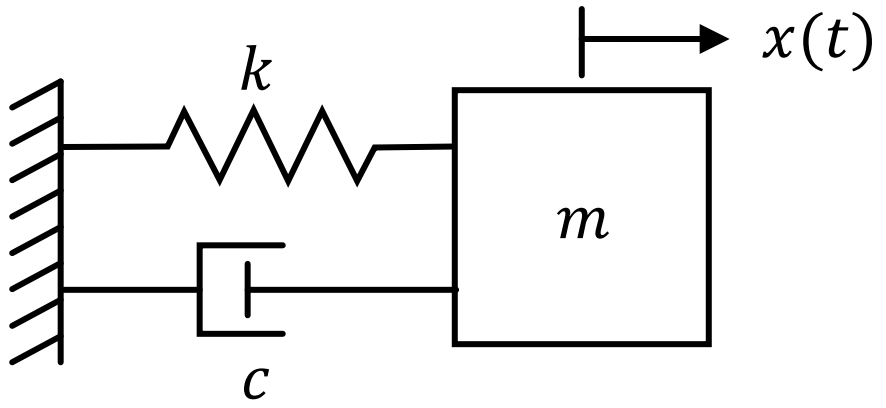
- Idea—Fit a neural network to the solution function of governing differential equations with the residuals of differential equations as loss functions:
  - For example: harmonic oscillator

$$m\ddot{x} + c\dot{x} + kx = 0$$

with initial conditions:

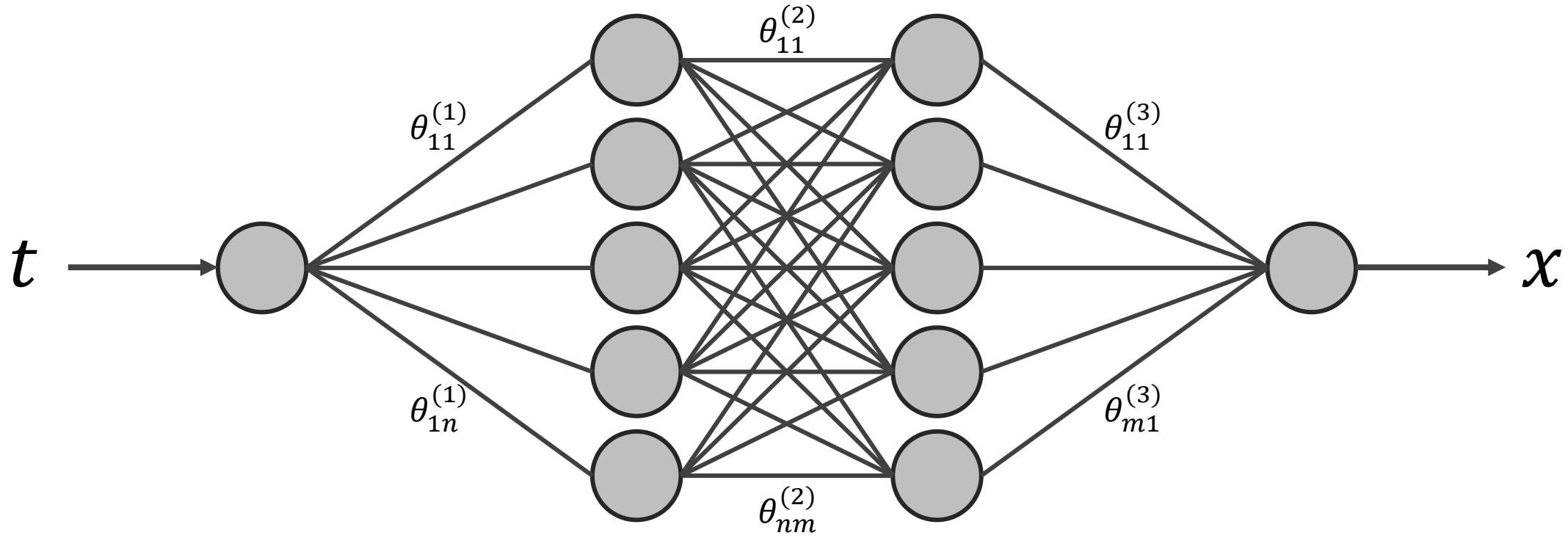
$$x(0) = x_0$$

$$\dot{x}(0) = v_0$$



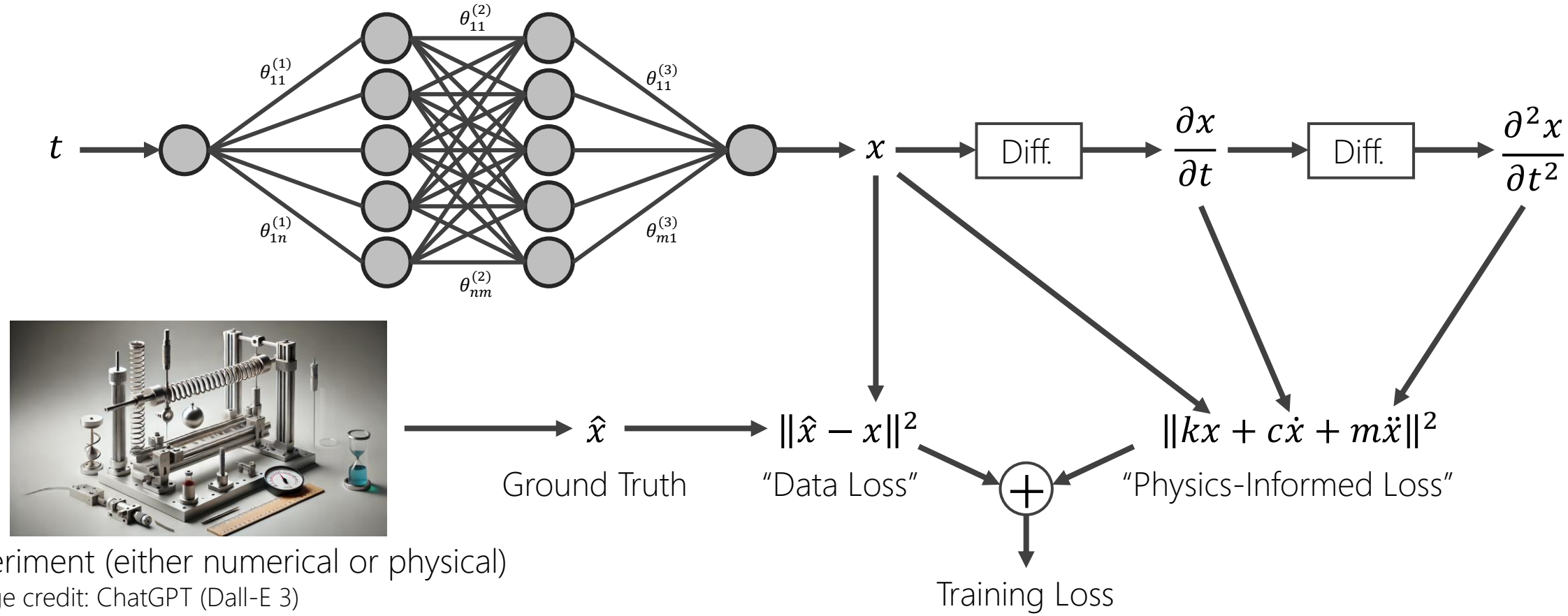
Physics-Informed Neural Networks

- Model $x(t)$ as a neural network $x_\theta(t)$:



Physics-Informed Neural Networks

- Model $x(t)$ as a neural network $x_\theta(t)$:

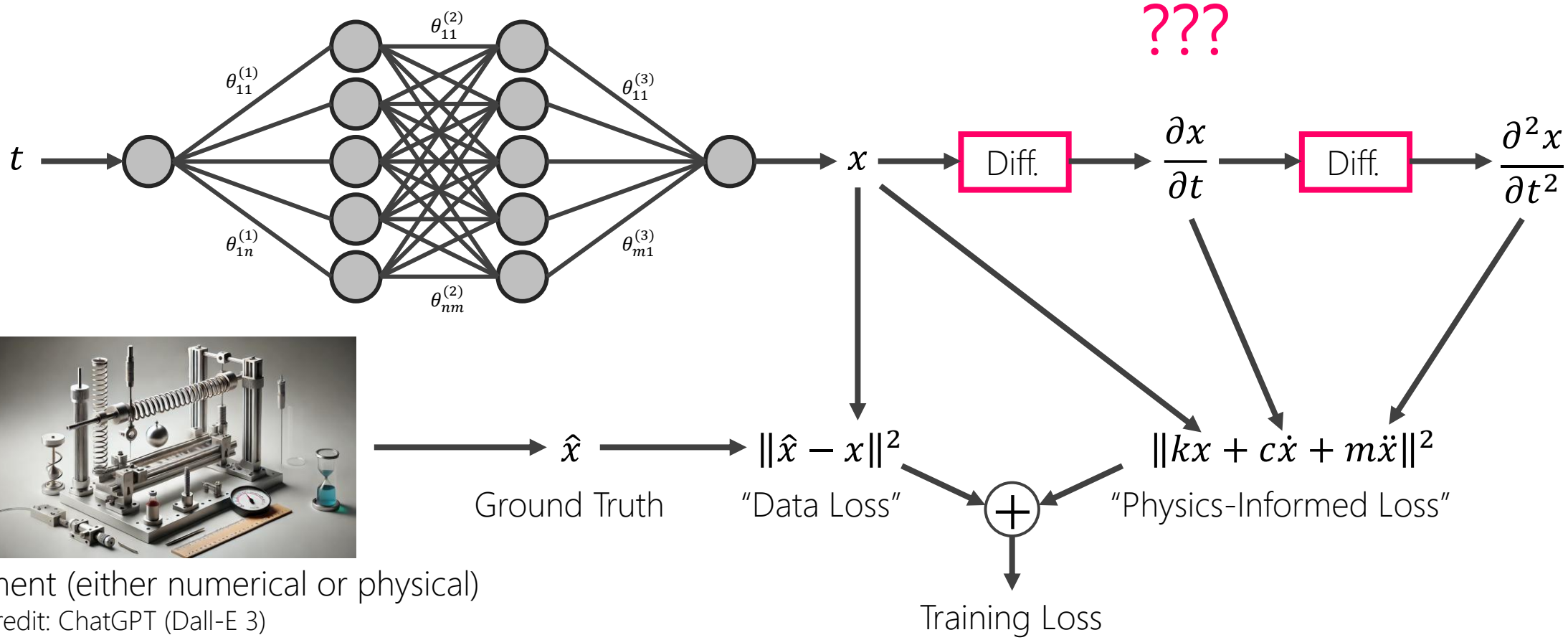


Experiment (either numerical or physical)

*image credit: ChatGPT (Dall-E 3)

Physics-Informed Neural Networks

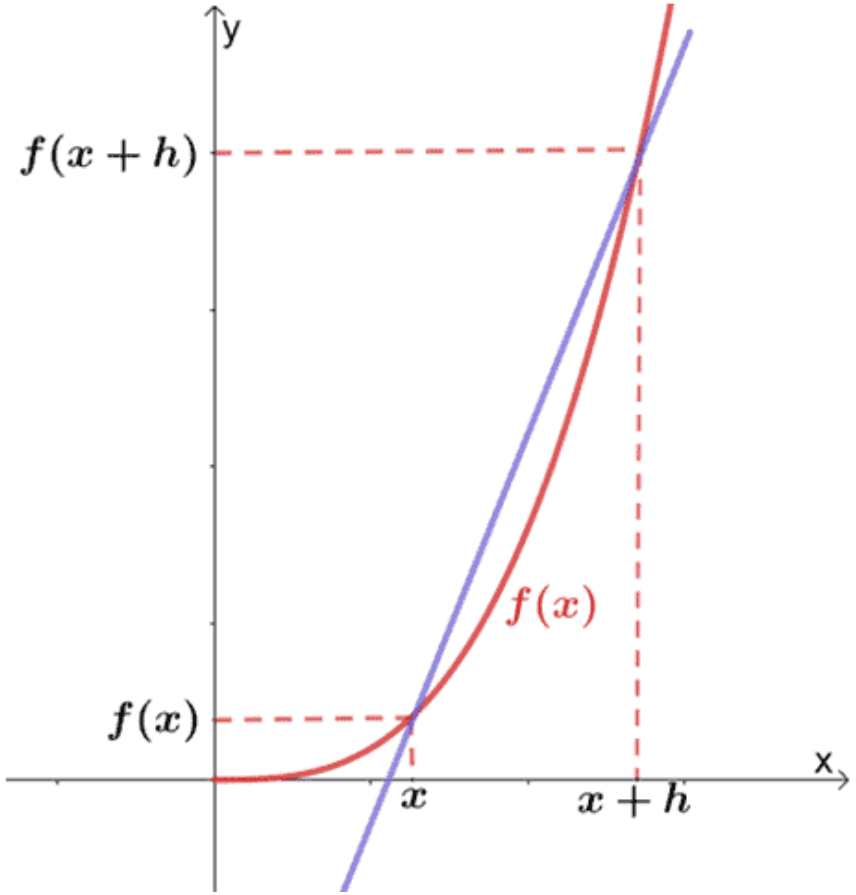
- Model $x(t)$ as a neural network $x_\theta(t)$:



Experiment (either numerical or physical)

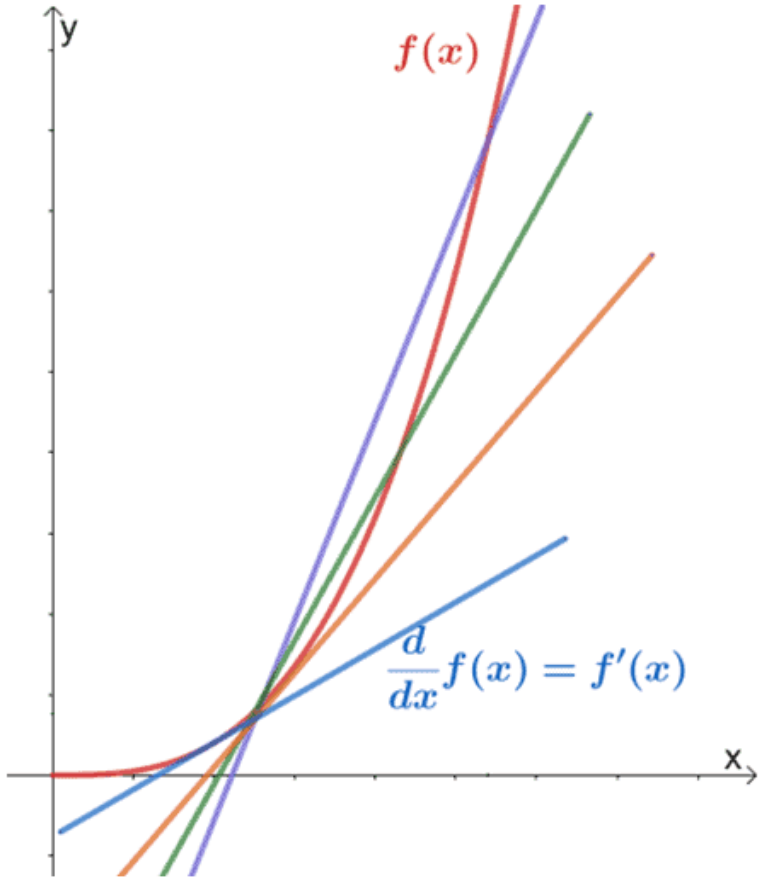
*image credit: ChatGPT (Dall-E 3)

Derivatives



$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x\ +h) - f(x)}{h}$$

Derivatives



$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

Numerical Derivatives

- Can you numerically compute the derivatives in the rigorous sense? Why?

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

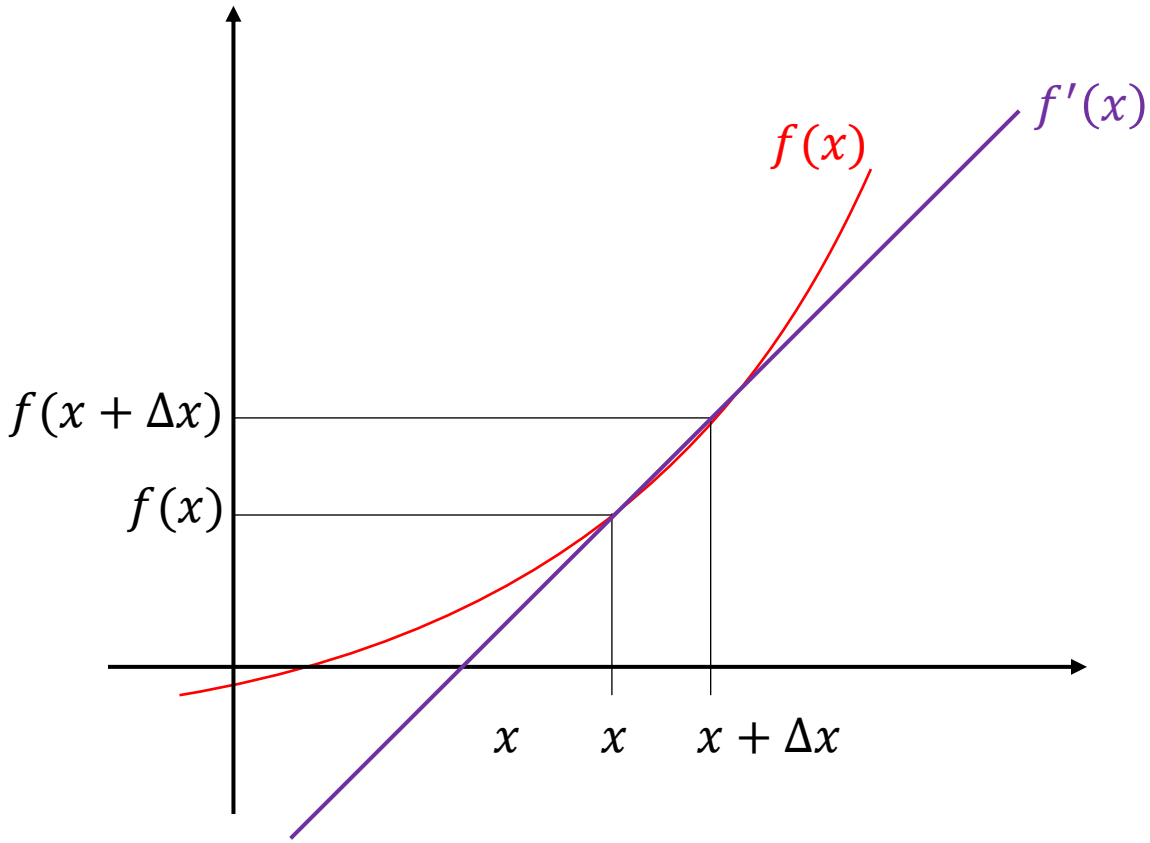
  - No. Because computers cannot represent the infinitesimally small quantity h.
  - Even if we use high enough numerical precision, all sorts of cancellation/round-off errors may occur (not to mention the issues of computational efficiency)

Numerical Derivatives

- A numerical approximate:

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

$$\cong \frac{f(x + \Delta x) - f(x)}{\Delta x}$$



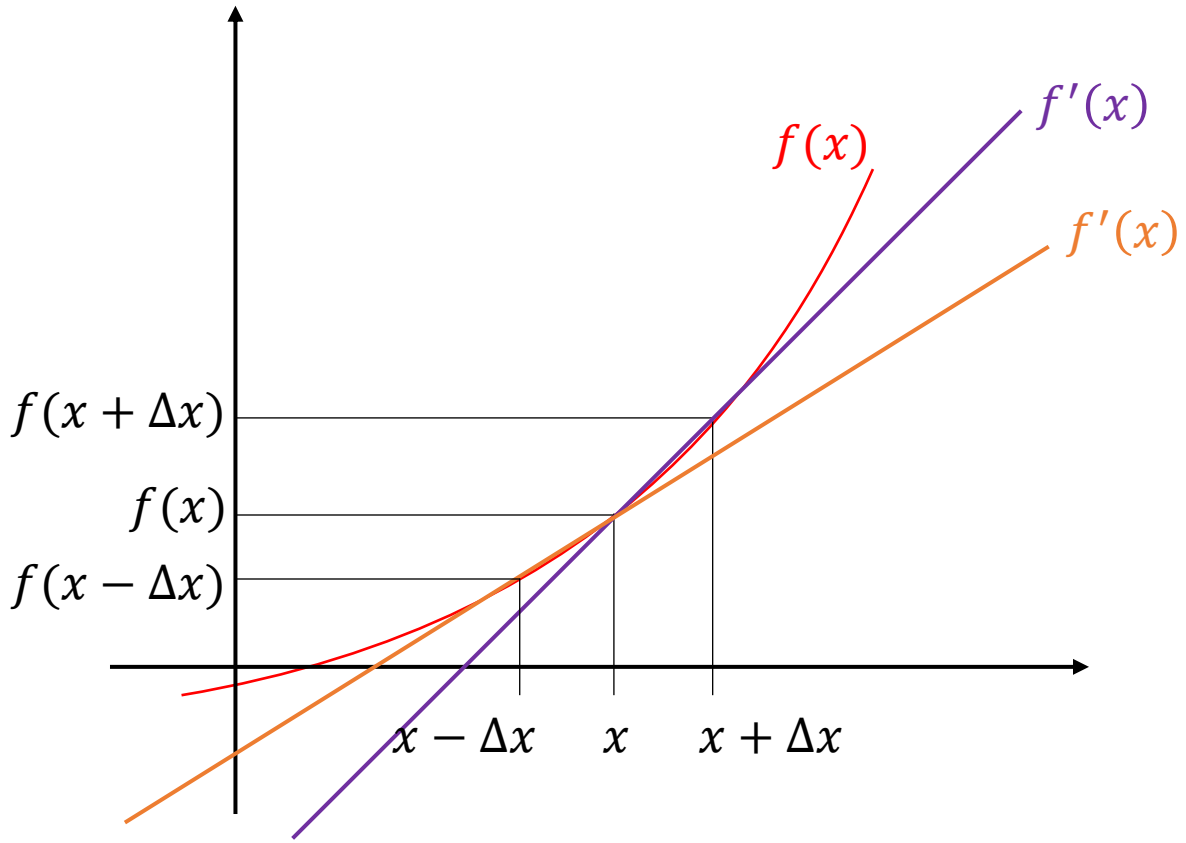
Forward and Backward Differences

- Depending on how you view it,
  - Forward difference:

$$\frac{dy}{dx} \cong \frac{f(x\ +\Delta x) - f(x)}{\Delta x}$$

  - Backward difference:

$$\frac{dy}{dx} \cong \frac{f(x) - f(x\ -\Delta x)}{\Delta x}$$




Forward Difference
Backward Difference
JAKE-CLARK.TUMBLR

Central Difference

- Compute the mean of forward and backward differences:

$$\frac{1}{2}\left(\frac{dy}{dx}+\frac{dy}{dx}\right)=\frac{1}{2}\left(\frac{f(x\ +\Delta x)-f(x)}{\Delta x}+\frac{f(x)-f(x\ -\Delta x)}{\Delta x}\right)$$

$$=\frac{1}{2}\left(\frac{f(x\ +\Delta x)-f(x)+f(x)-f(x\ -\Delta x)}{\Delta x}\right)$$

$$=\frac{f(x\ +\Delta x)-f(x\ -\Delta x)}{2\Delta x}$$

$$\Leftrightarrow\ \frac{f\left(x\ +\frac{\Delta x}{2}\right)-f\left(x\ -\frac{\Delta x}{2}\right)}{\Delta x}$$

→ Central difference

Finite Difference Methods

- Forward Difference

$$\frac{dy}{dx} \cong \frac{f(x\ + \Delta x) - f(x)}{\Delta x}$$

- Backward Difference

$$\frac{dy}{dx} \cong \frac{f(x) - f(x\ - \Delta x)}{\Delta x}$$

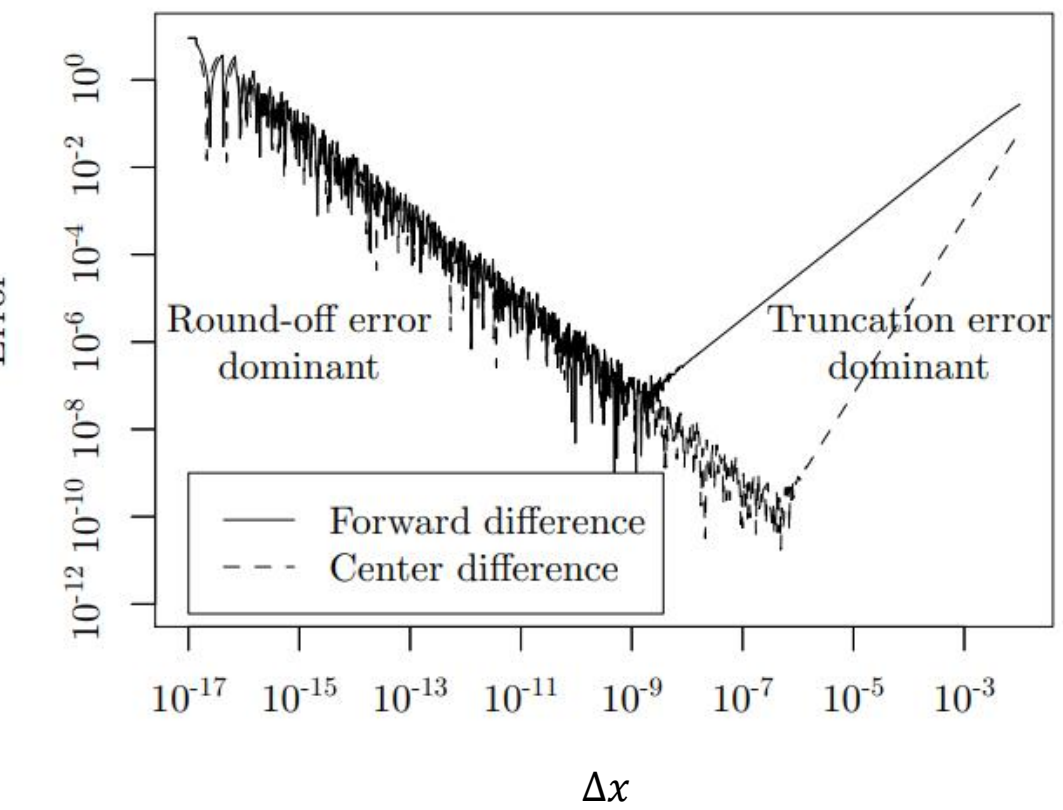
- Central Difference

$$\frac{dy}{dx} \cong \frac{f\left(x\ + \frac{\Delta x}{2}\right) - f\left(x\ - \frac{\Delta x}{2}\right)}{\Delta x}$$

Limitations?

- Truncation vs Rounding Error
- "Δx" can never be actually "zero"
  - Large Δx: Approximation error
  - Small Δx: Numerical error
    - The numerator terms ($f(x + \Delta x)$ and $f($ denominator Δx is tiny.
    - What happens to the insignificant mantissa bits

Baydin et al. (2018). Automatic differentiation in machine learning: a survey

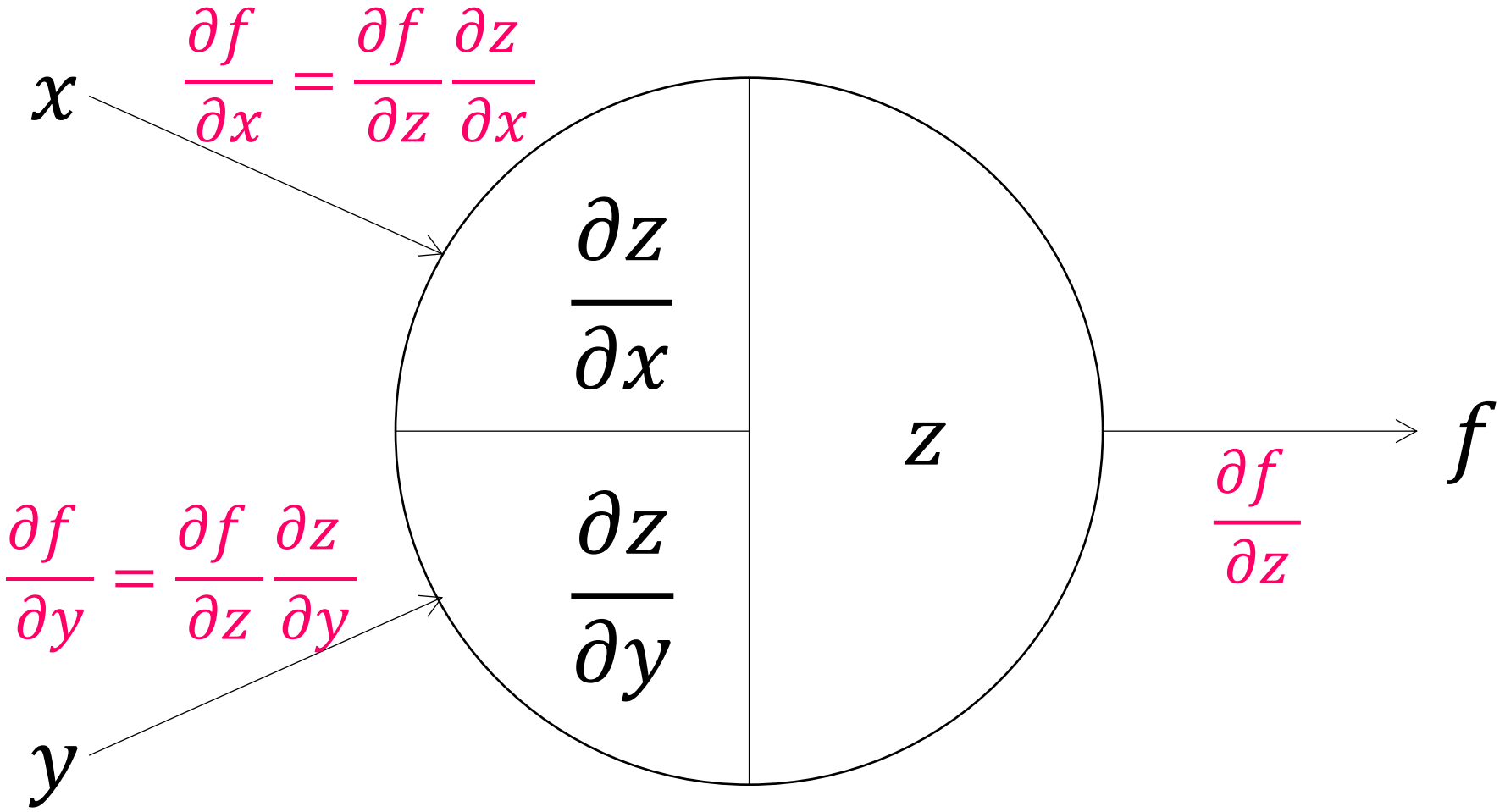


What about n-Dimensions?

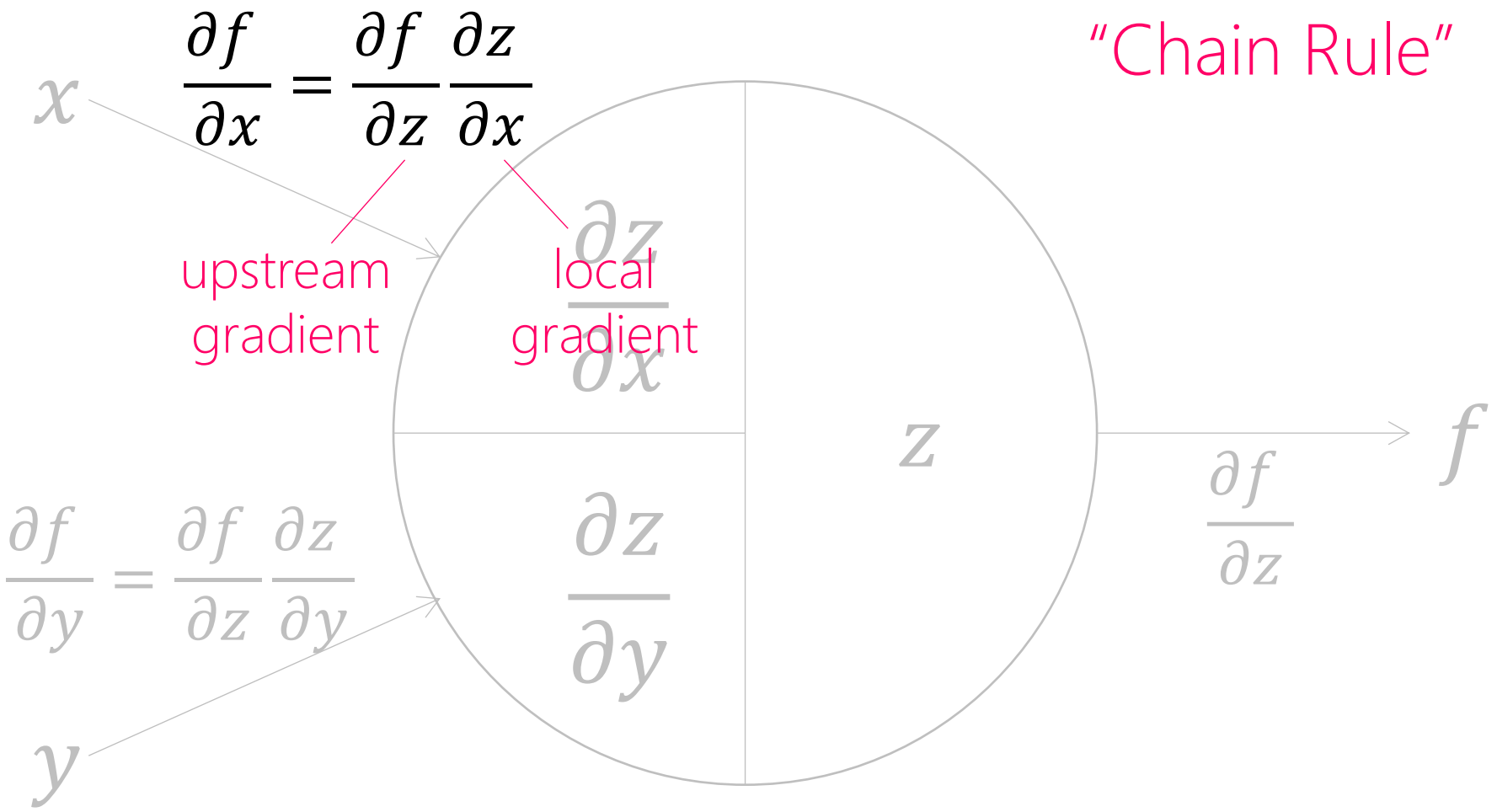
- How many times do you need to evaluate function $f()$ for a multivariate case (i.e. $x \in \mathbb{R}^n$ for $n > 1$)?

→ Requires $O(n)$ evaluations

Recap: Backpropagation

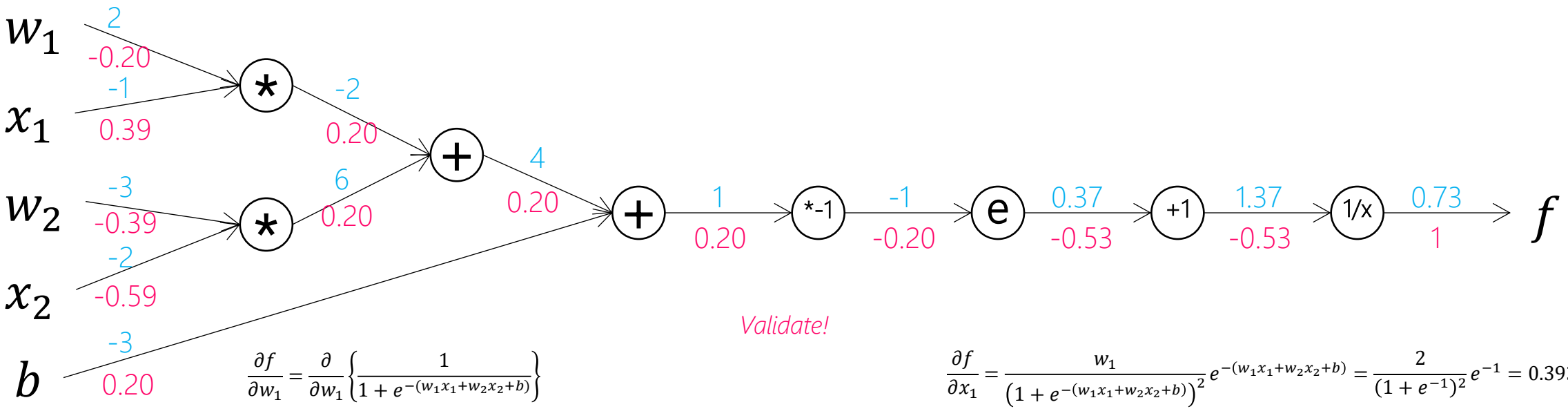


Recap: Backpropagation



Example

$$f(w,x) = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + b)}}$$



Validate!

Cheat Sheet:

$$\frac{\partial}{\partial x}\left\{\frac{1}{x}\right\} = -\frac{1}{x^2}$$

$$\frac{\partial}{\partial x}e^x = e^x$$

$$\frac{\partial f}{\partial w_1} = \frac{\partial}{\partial w_1}\left\{\frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + b)}}\right\}$$

$$= -\frac{1}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}\frac{\partial}{\partial w_1}\left\{1 + e^{-(w_1x_1 + w_2x_2 + b)}\right\}$$

$$= -\frac{1}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)}\frac{\partial}{\partial w_1}\{-(w_1x_1 + w_2x_2 + b)\}$$

$$= \frac{x_1}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)}$$

$$= \frac{-1}{\left(1 + e^{-(-2+6-3)}\right)^2}e^{-(-2+6-3)} = -0.1966$$

$$\frac{\partial f}{\partial x_1} = \frac{w_1}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)} = \frac{2}{(1 + e^{-1})^2}e^{-1} = 0.3932$$

$$\frac{\partial f}{\partial w_2} = \frac{x_2}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)} = \frac{-2}{(1 + e^{-1})^2}e^{-1} = -0.3932$$

$$\frac{\partial f}{\partial x_2} = \frac{w_2}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)} = \frac{-3}{(1 + e^{-1})^2}e^{-1} = -0.5898$$

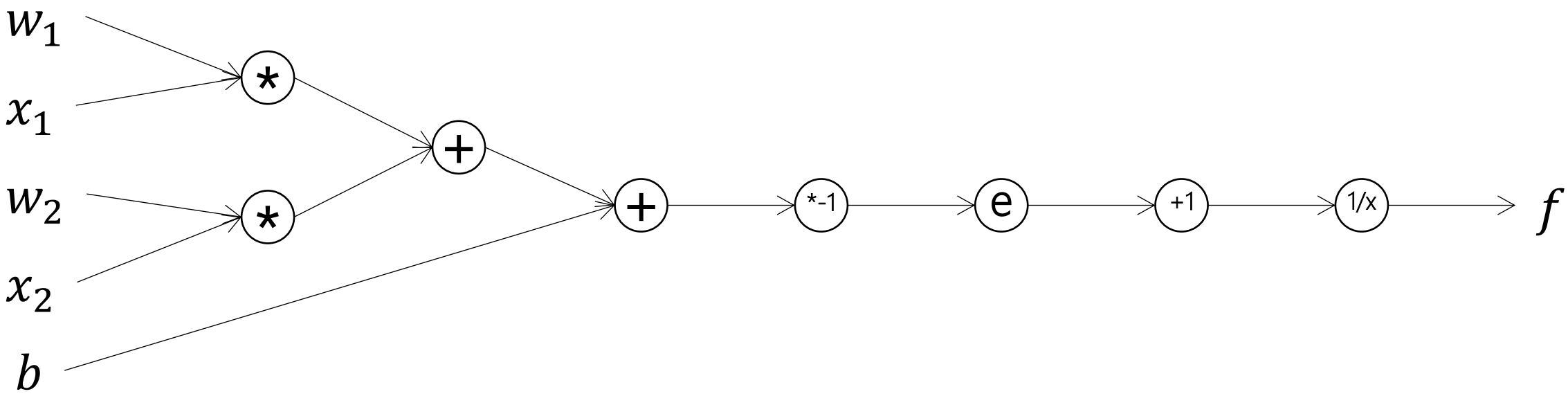
$$\frac{\partial f}{\partial b} = \frac{1}{\left(1 + e^{-(w_1x_1 + w_2x_2 + b)}\right)^2}e^{-(w_1x_1 + w_2x_2 + b)} = \frac{1}{(1 + e^{-1})^2}e^{-1} = 0.1966$$

Can we do better?

- What are the problems of backpropagation?
  - You always need to keep intermediate data in the memory during the forward pass in case it will be used in the backpropagation.
  - Only limited to the first order derivatives. (e.g. gradient of gradient?)

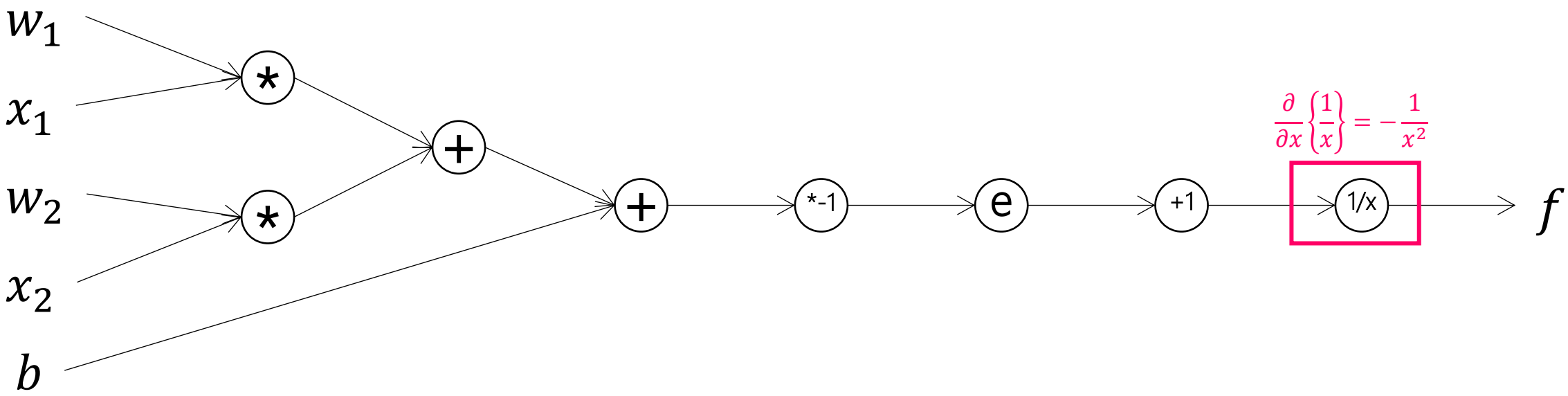
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



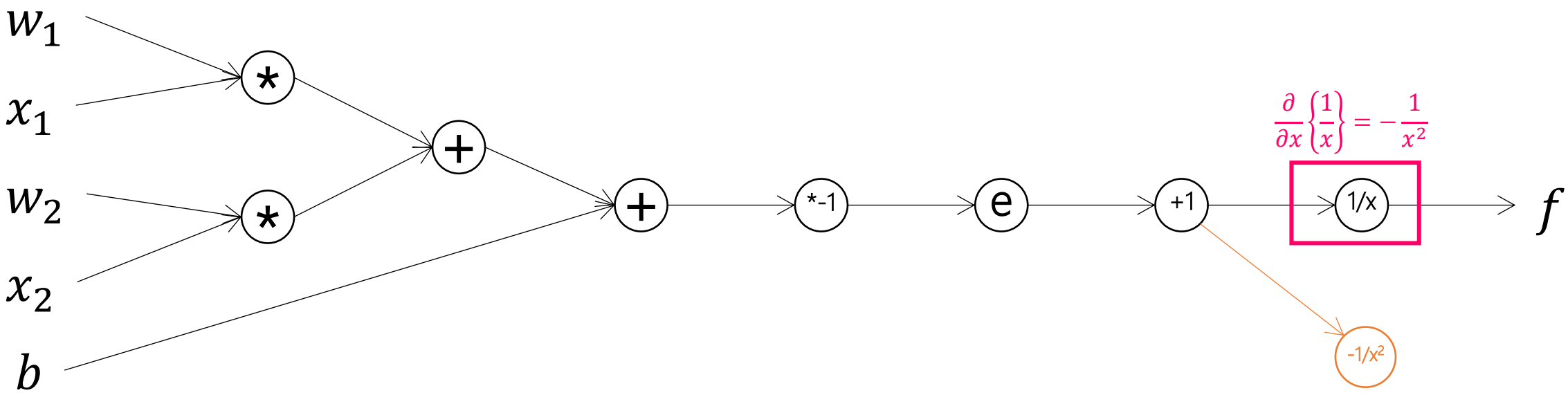
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



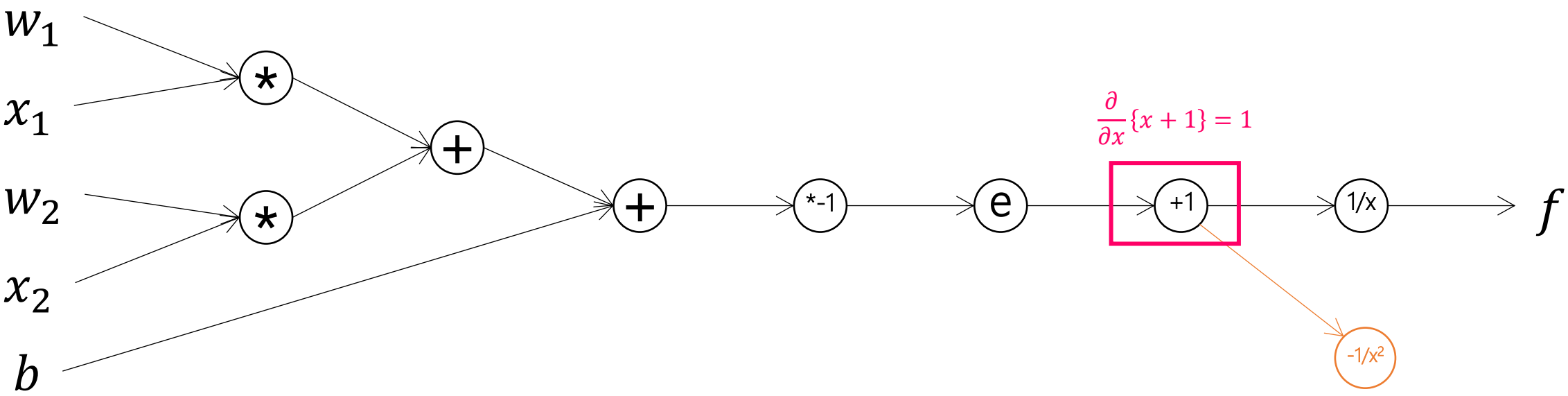
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



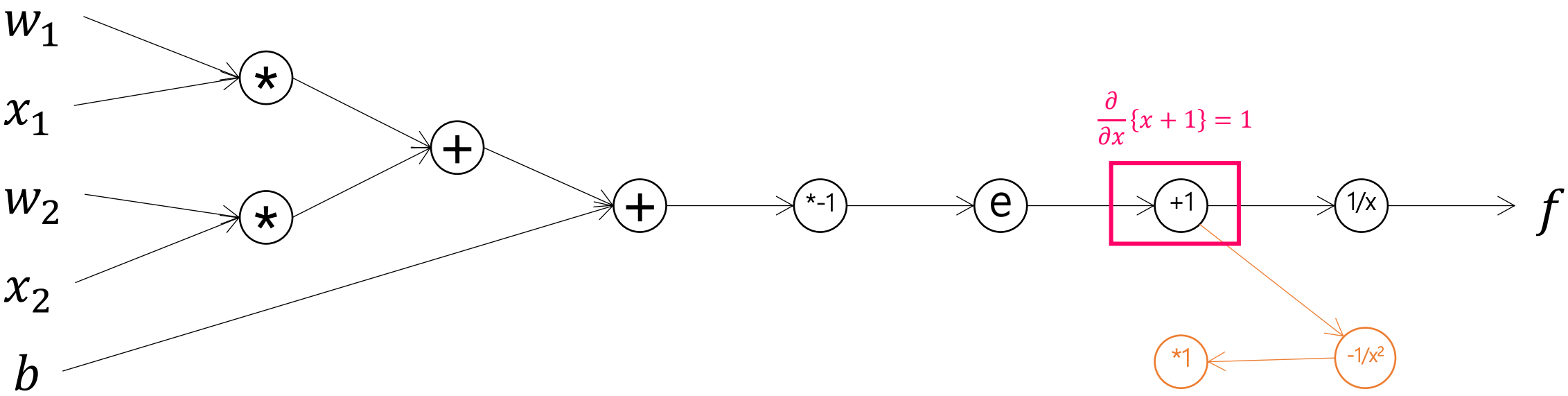
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



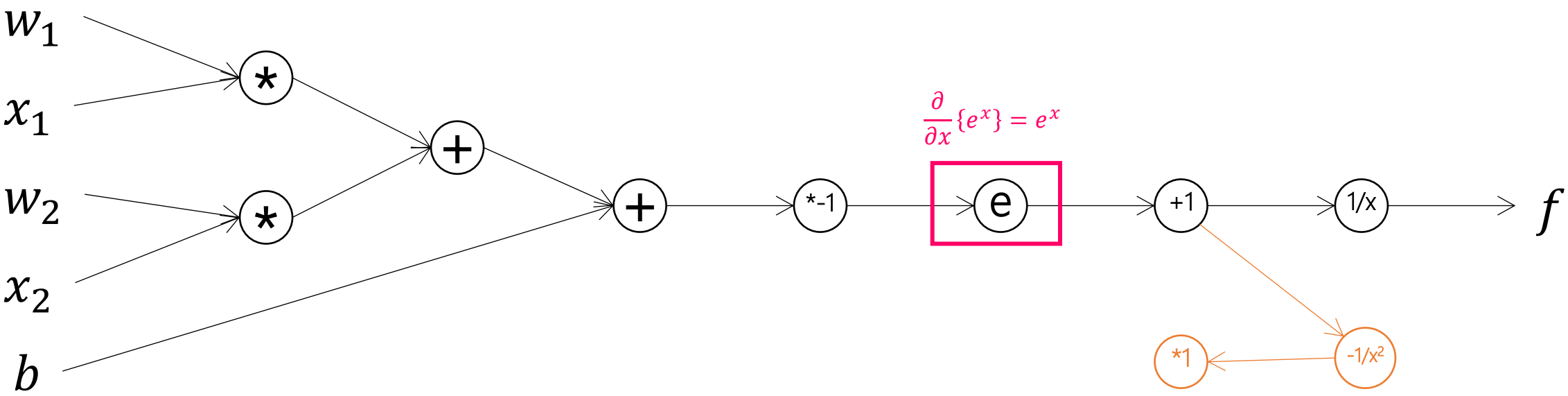
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



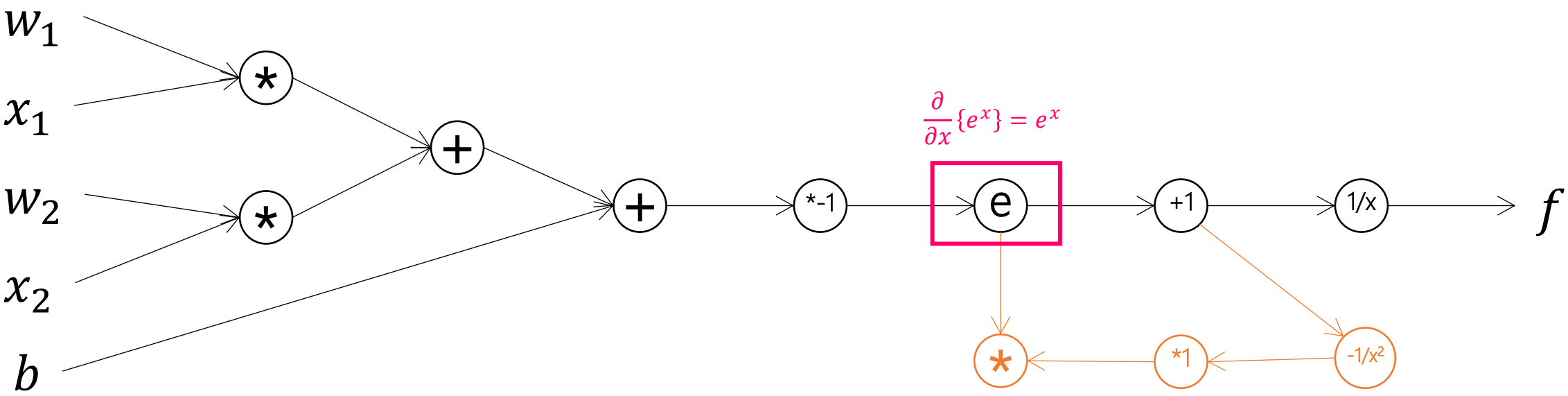
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



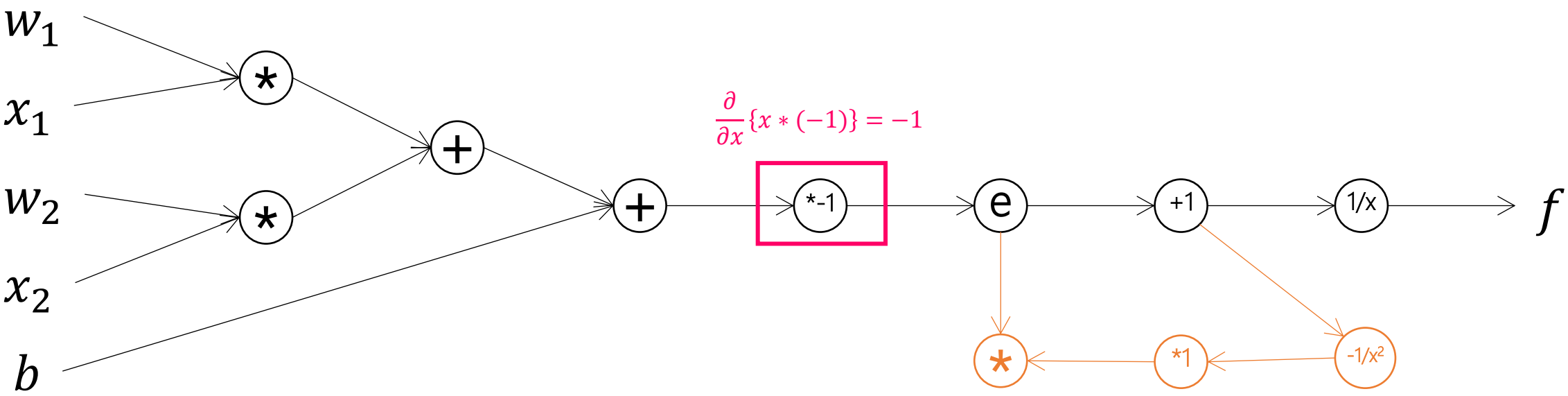
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



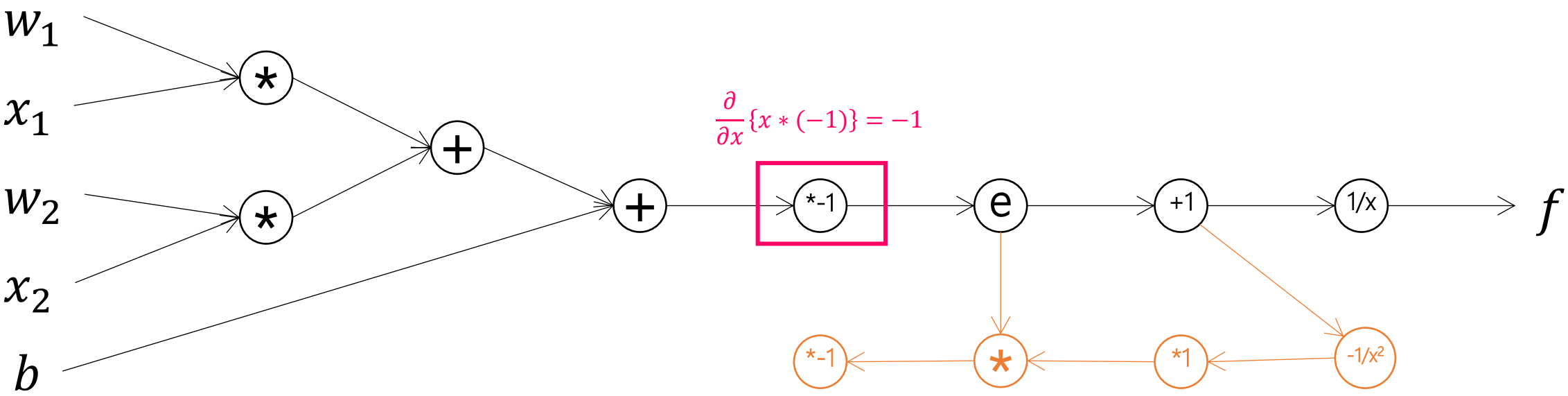
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



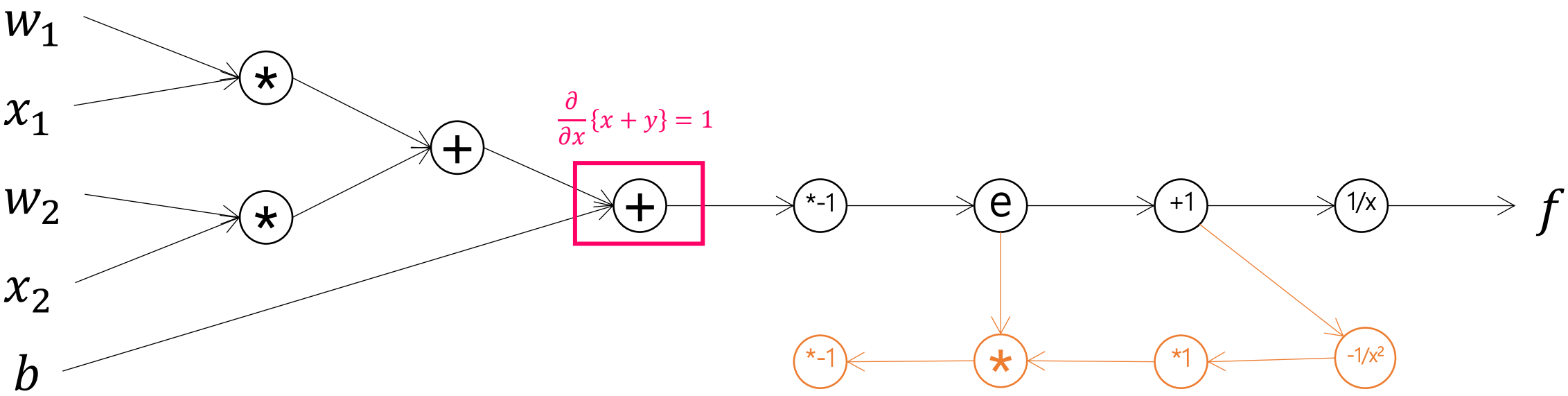
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



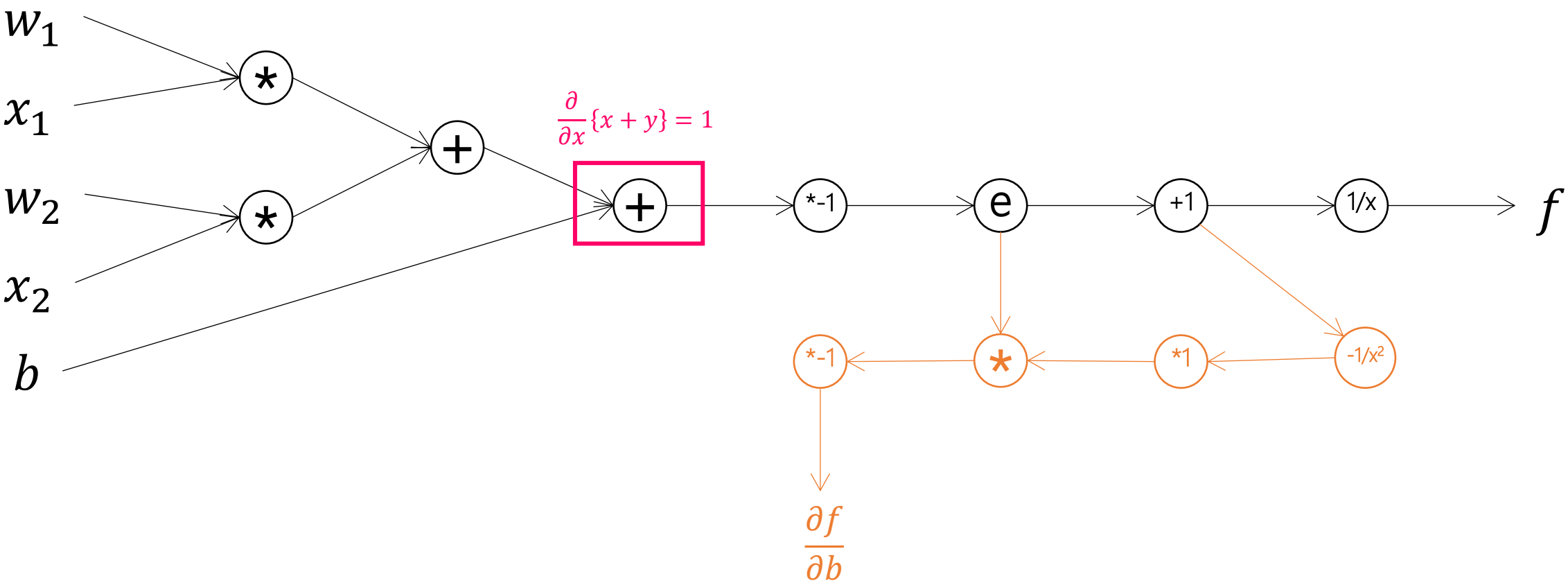
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



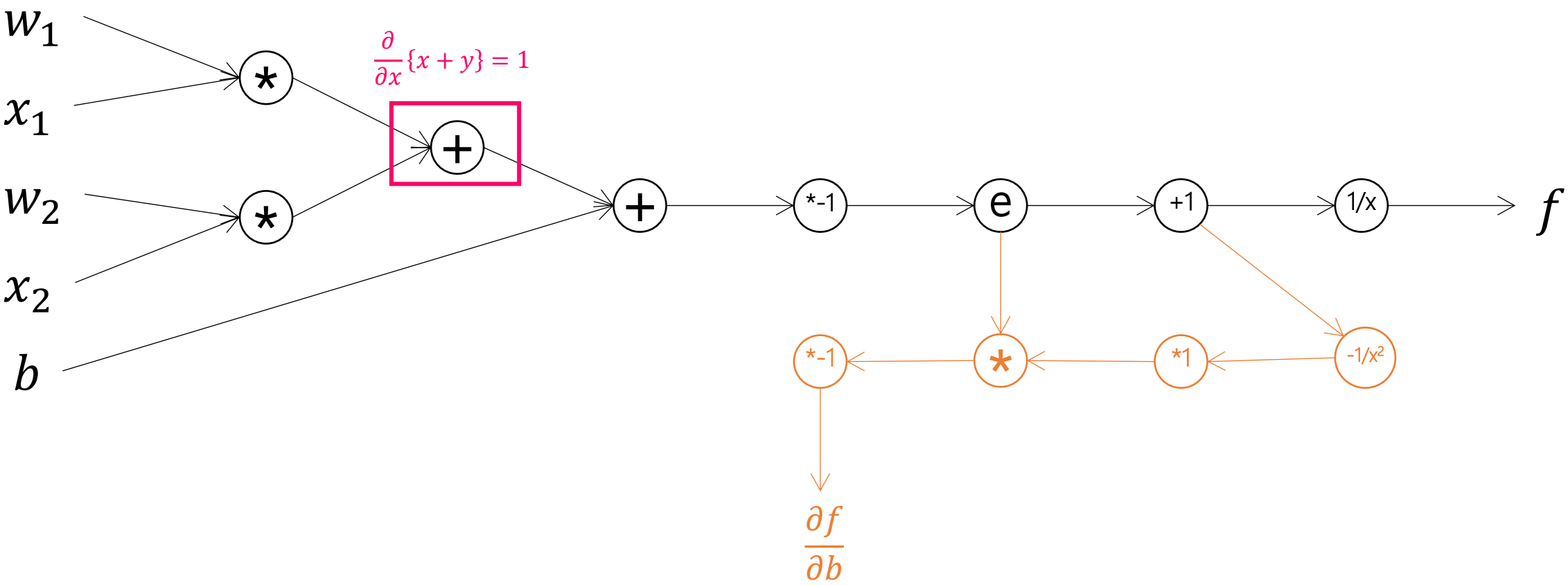
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



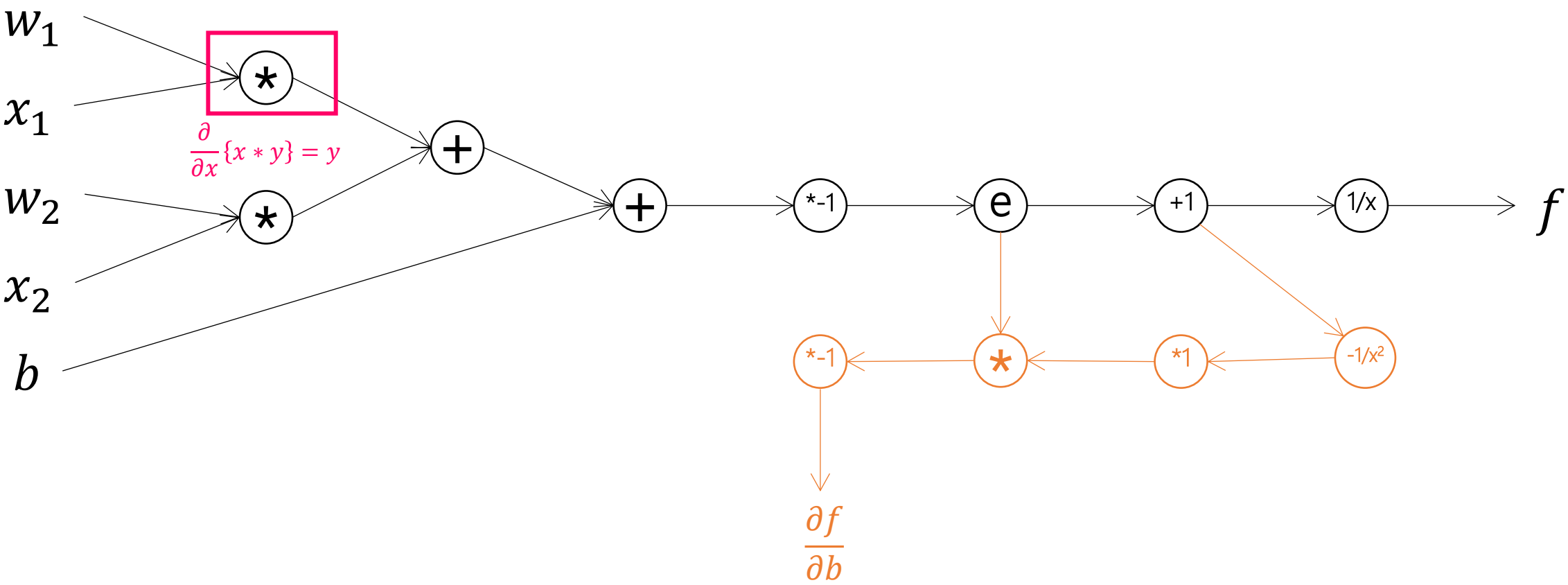
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



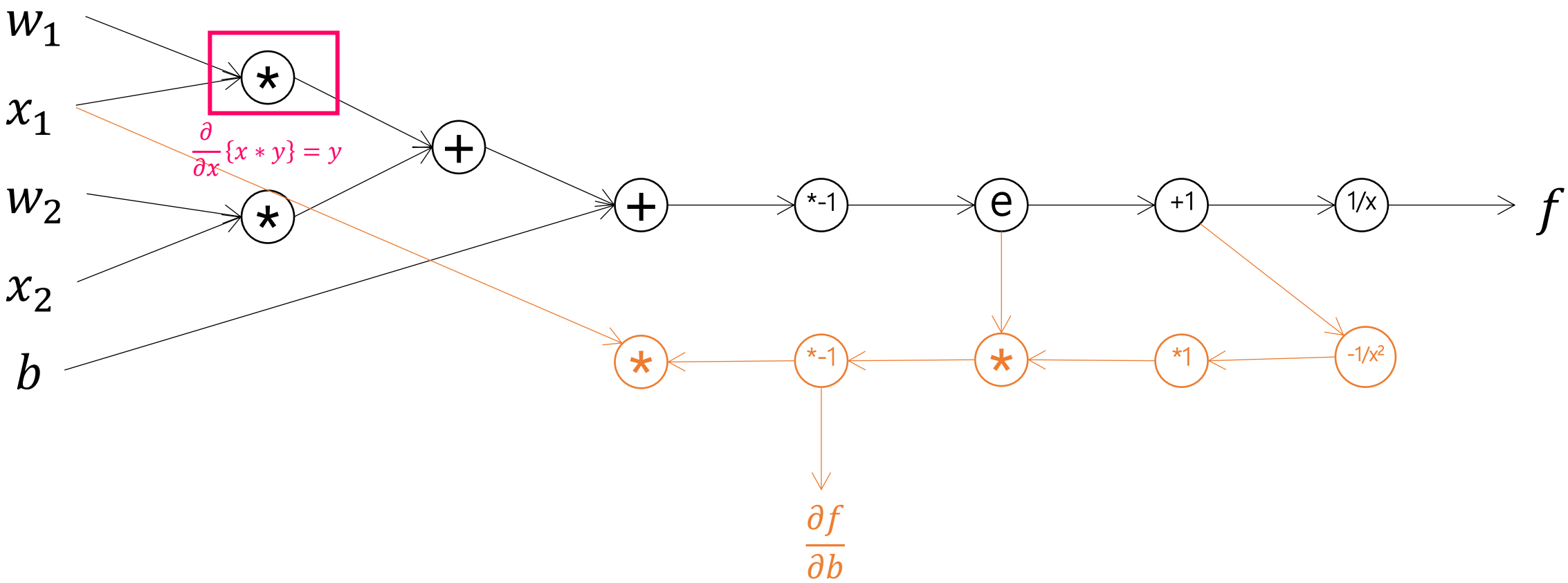
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



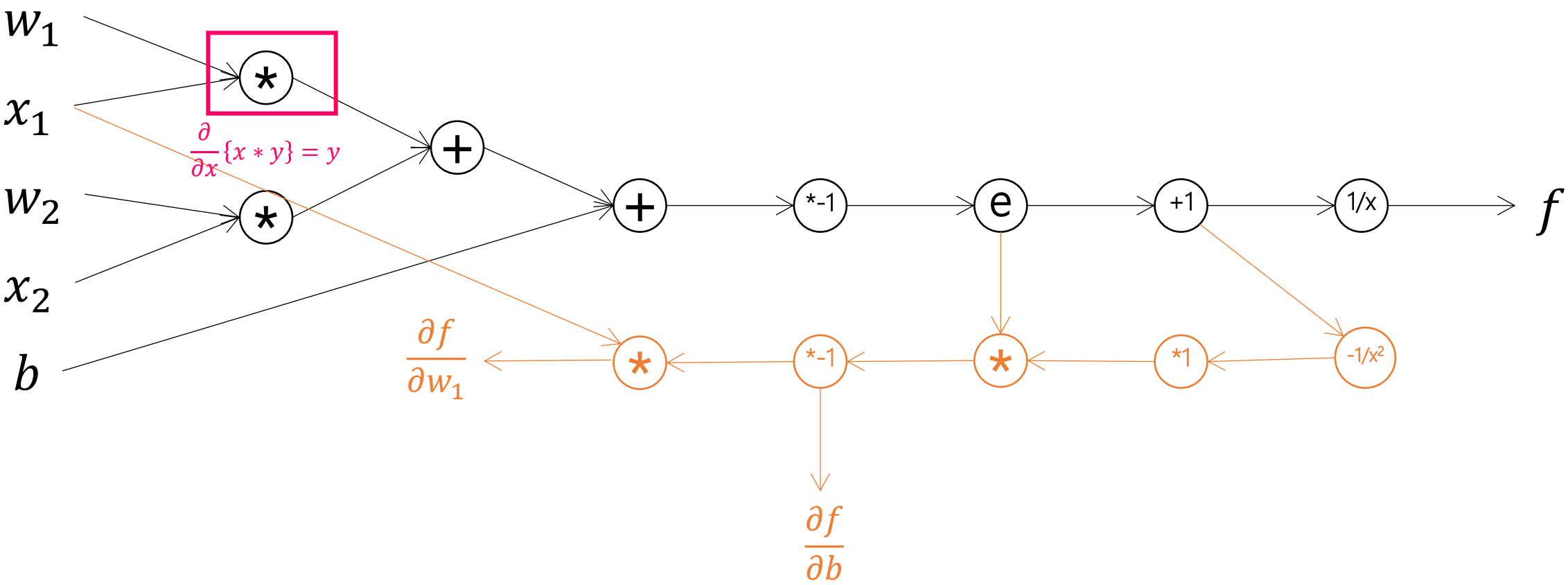
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



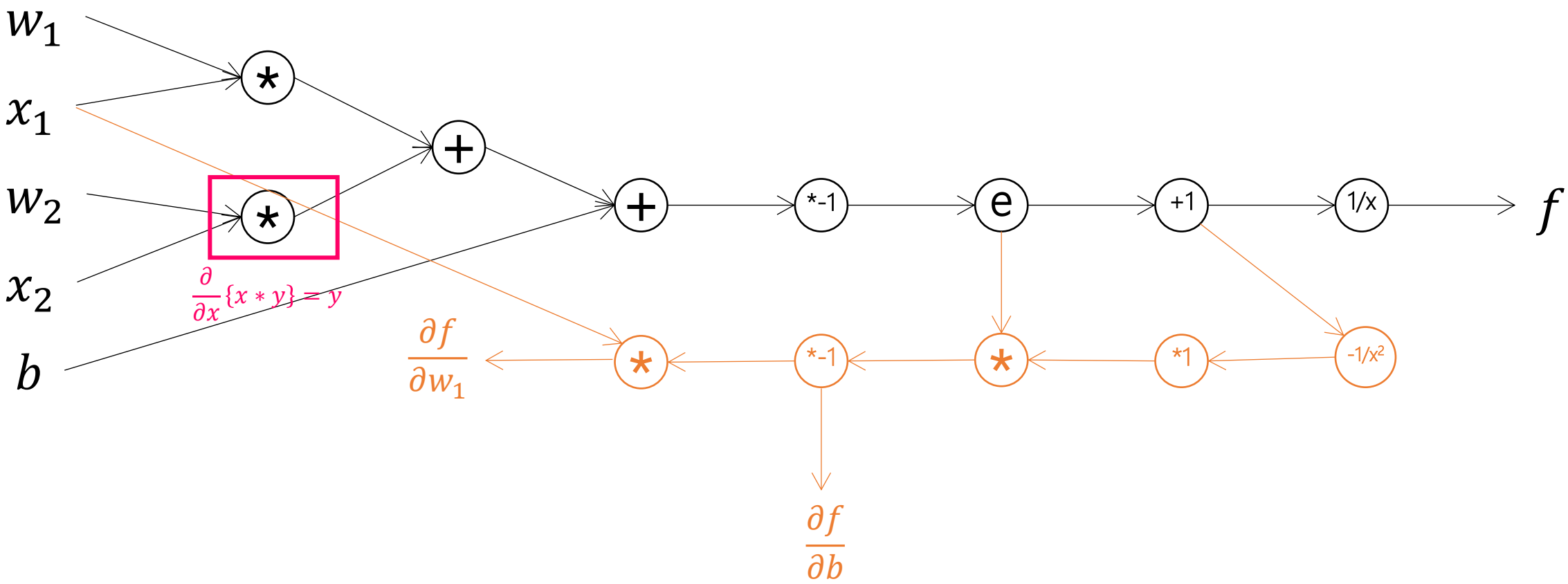
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



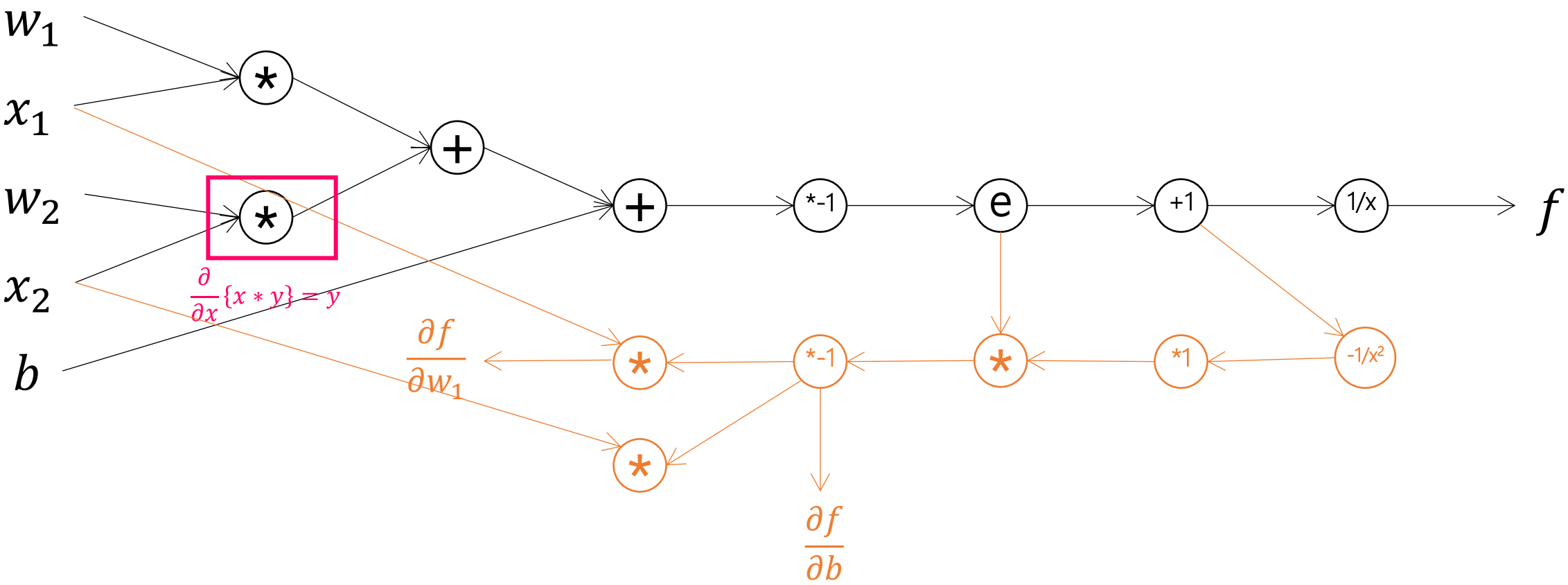
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



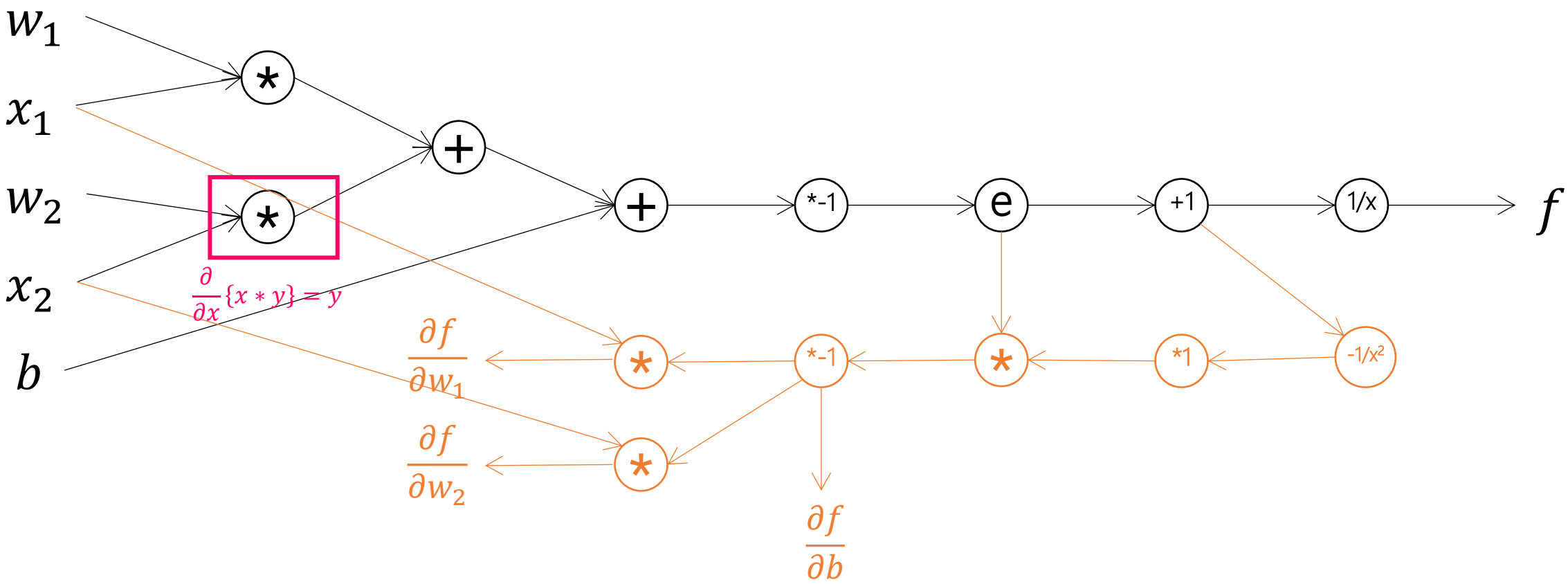
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



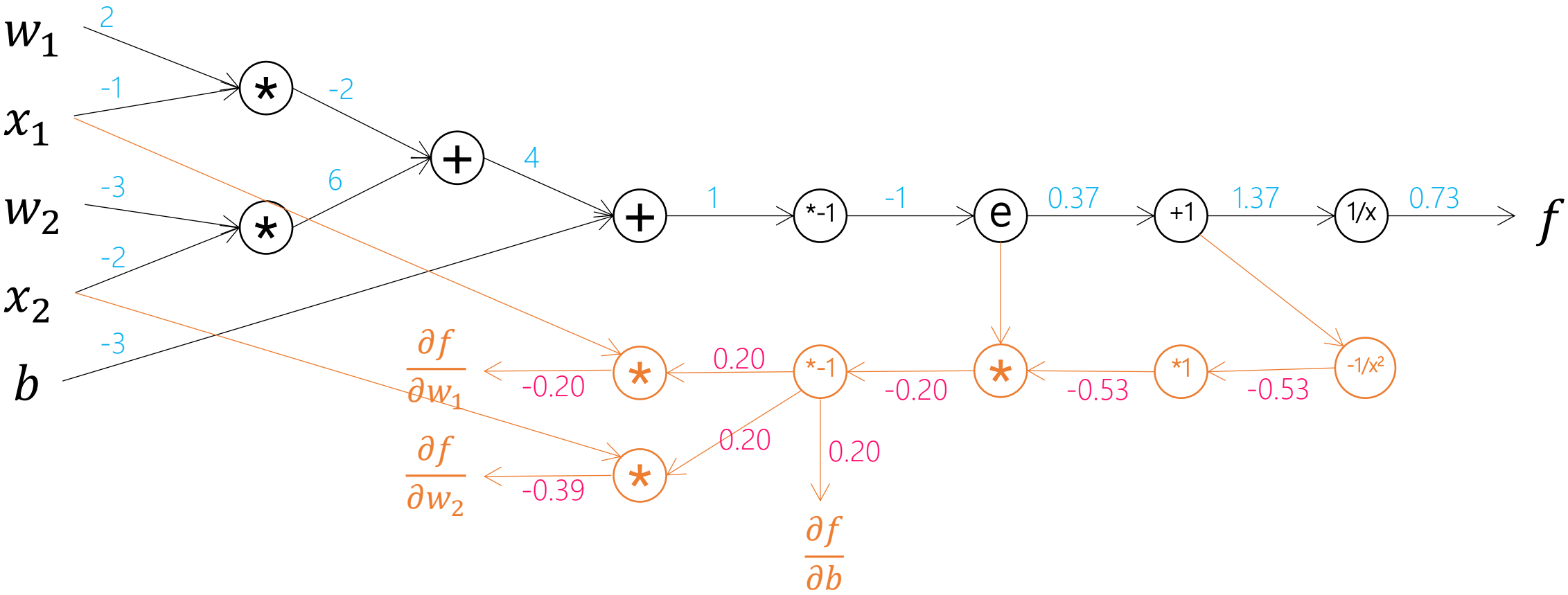
Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



Automatic Differentiation (Autodiff)

- Create a computational graph for the gradient computation too.



Automatic Differentiation (Autodiff)

- Autodiff is not finite differences.
  - Finite differences are expensive
    - At least two forward passes for each derivative.
  - Prone to numerical error
- Autodiff is efficient.
  - Linear computing cost.
- Autodiff is accurate and numerically stable.

Automatic Differentiation (Autodiff)

- Autodiff is not symbolic differentiation
  - No complex and redundant expressions
  - The goal of autodiff is not a formula, but a procedure for computing derivatives.

For vector-valued functions?

- Jacobian!

$$J = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

For vector-valued functions?

- Jacobian!

$$J = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- Chain rule in Jacobian?

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y_1}\frac{\partial y_1}{\partial x} + \cdots + \frac{\partial f}{\partial y_d}\frac{\partial y_d}{\partial x} = \sum_k \frac{\partial f}{\partial y_k}\frac{\partial y_k}{\partial x}$$

$$\mathbf{f}' = \begin{bmatrix} \frac{\partial f}{\partial y_1} & \cdots & \frac{\partial f}{\partial y_d} \end{bmatrix}^T \mathbf{y}'$$

For vector-valued functions?

- Jacobian!

$$J = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- Chain rule in Jacobian?

$$\frac{\partial f_i}{\partial x_j} = \frac{\partial f_i}{\partial y_1}\frac{\partial y_1}{\partial x_j} + \cdots + \frac{\partial f_i}{\partial y_d}\frac{\partial y_d}{\partial x_j} = \sum_k \frac{\partial f_i}{\partial y_k}\frac{\partial y_k}{\partial x_j}$$

$$\mathbf{f}' = \mathbf{J}^T \mathbf{y}'$$

Vector-Jacobian Products

- Does that mean that we need to compute Jacobian for each node?
  - No. We never explicitly construct the Jacobian.
  - Instead, it is simpler and more efficient to compute VJP directly.

Vector-Jacobian Products

- Consider a primitive operation (aka a simple scalar-valued function that can be used as a building block to construct more complicated vector valued functions) → e.g. `f = np.exp(g(x))`

- Jacobian:

See the inefficiency here?

$$J = \frac{\partial f}{\partial g} = \begin{bmatrix} e^{g_1} & & 0 \\ & \ddots & \\ 0 & & e^{g_n} \end{bmatrix}$$

Vector-Jacobian Products

- Instead, VJP:

$$J = \frac{\partial f}{\partial g} = \begin{bmatrix} e^{g_1} & & 0 \\ & \ddots & \\ 0 & & e^{g_n} \end{bmatrix}$$

$$f' = J^T g' = \begin{bmatrix} e^{g_1} & & 0 \\ & \ddots & \\ 0 & & e^{g_n} \end{bmatrix} \begin{bmatrix} g_1' \\ \vdots \\ g_n' \end{bmatrix} = \begin{bmatrix} g_1' e^{g_1} & \cdots & g_n' e^{g_n} \end{bmatrix}$$

Autodiff in Python – Autograd

- https://github.com/HIPS/autograd or '`pip install autograd`'

Very sneaky! Look and feel like NumPy functions, but secretly build the computation graph.

```
>>> import autograd.numpy as np  # Thinly-wrapped numpy
>>> from autograd import grad    # The only autograd function you may ever need
>>>
>>> def tanh(x):                 # Define a function
...     y = np.exp(-2.0 * x)
...     return (1.0 - y) / (1.0 + y)
...
>>> grad_tanh = grad(tanh)       # Obtain its gradient function
>>> grad_tanh(1.0)               # Evaluate the gradient at x = 1.0
0.41997434161402603
>>> (tanh(1.0001) - tanh(0.9999)) / 0.0002  # Compare to finite differences
0.41997434264973155
```

Or in PyTorch…

- Coding time! See '`03_autograd.ipynb`'

Comparison

- Numerical differentiation
  - Tool to check the correctness of implementation
- Backpropagation
  - Easy to understand and implement
  - Bad for memory use and schedule optimization
- Automatic differentiation
  - Generate gradient computation to entire computation graph
  - Better for system optimization

Back to PINN…

More Formal Problem Setup

- Nonlinear PDEs with parameter λ:

$$u_t + \mathcal{N}[u;\lambda] = 0, x \in \Omega \subset \mathbb{R}^D, t \in [0,T] \subset \mathbb{R}$$

  where…
  - $u(t,x)$: the solution function
  - $u_t \coloneqq \frac{\partial u}{\partial t}$: the time-derivative
  - $\mathcal{N}[u;\lambda]$: a nonlinear operator parameterized by λ
  - Ω : physical domain with boundary $\partial\Omega$

Example: Linear convection-diffusion equation

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0$$

where $\mathcal{N}[u;\lambda] = cu_x - \nu u_{xx}$ and $\lambda = (c,\nu) > 0$ (convection and diffusion coefficients)

More Formal Problem Setup

$$u_t + \mathcal{N}[u; \lambda] = 0, x \in \Omega \subset \mathbb{R}^D, t \in [0, T] \subset \mathbb{R}$$

Raissi et al. (2018):

- Data driven solutions of PDEs:

  "Given $\boldsymbol{\lambda}$, what is $\boldsymbol{u(t, x)}$?"

- Data driven discovery of PDEs:

  "Find $\boldsymbol{\lambda}$ that best describes observations $u\left(t^{(i)}, x^{(i)}\right)|_{i=1,\cdots,N}$"

Data-Driven Solutions

- Let $u_\theta(t, x)$ be a neural network with network parameters θ that approximates the unknown PDE solution $u(t, x) \approx u_\theta(t, x)$.
- Given:
  - Observations : $t^{(i)}; x^{(i)}; u^{(i)} = u\left(t^{(i)}, x^{(i)}\right)$ for $i = 1, \ldots, N_u$
    (i.e., data collected from either physical or numerical experiments)
  - Governing equation: $u_t + \mathcal{N}[u; \lambda] = 0, x \in \Omega \subset \mathbb{R}^D, t \in [0, T] \subset \mathbb{R}$
    with some known parameters λ.
- Objective: train the neural network u_θ with the following objectives

$$\mathcal{L} = \mathcal{L}_u + \mathcal{L}_f$$

  - "*Data loss*" (mean-square error): $\mathcal{L}_u = \frac{1}{N_u} \sum_{i=1}^{N_u} \left\| u^{(i)} - u_\theta\left(t^{(i)}, x^{(i)}\right) \right\|^2$
  - "*Physics loss*" (PDE residual): $\mathcal{L}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left\| f\left(t_f^{(i)}, x_f^{(i)}\right) \right\|^2$ where $f(u; t, x) = u_t + \mathcal{N}[u; \lambda]$

More on the Physics-Informed Loss

- "Physics loss" (PDE residual):

$$\mathcal{L}_f = \frac{1}{N_f}\sum_{i=1}^{N_f}\left\|f\left(t_f^{(i)}, x_f^{(i)}\right)\right\|^2$$

  where $f(u;t,x) = u_t + \mathcal{N}[u;\lambda]$.

- $\left\{t_f^{(i)}, x_f^{(i)}\right\}_{i=1}^{N_f}$ are called "*collocation points*" that are usually sampled from…
  - $(t_f, x_f) \sim \mathcal{U}([0,T]\times\Omega)$: uniform distribution in space-time (for now).
  - $(t_f, x_f) \sim P(t,x)$: some fancier sampling method (we'll see in the later slides).

Data-Driven Solutions: Example – Heat Eqn.

- Heat Equation:
  - Describes the diffusion of heat through a given region.

- 1D heat diffusion process:

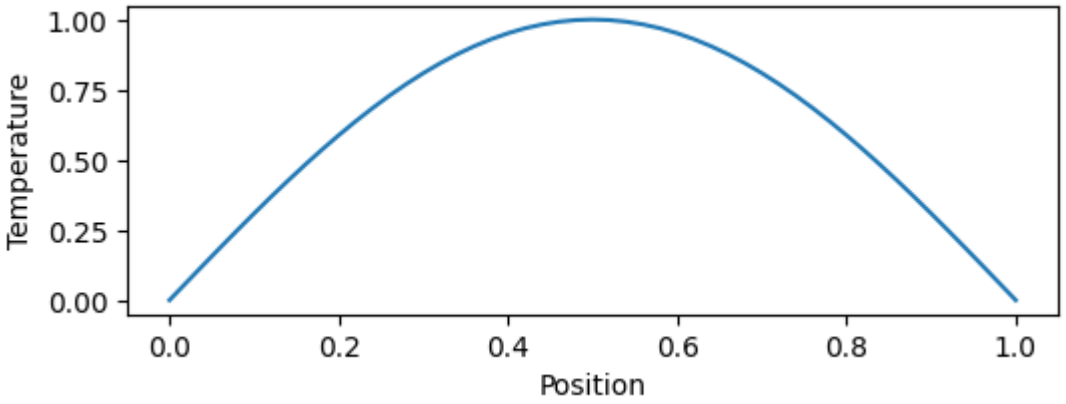
$$\frac{\partial u}{\partial t} = \lambda u_{xx}$$

  - … where $u(t, x)$ is the temperature at time t and location $x \in \Omega \subset \mathbb{R}$ in the physical domain Ω, and λ is a positive coefficient representing the thermal diffusivity of the medium.

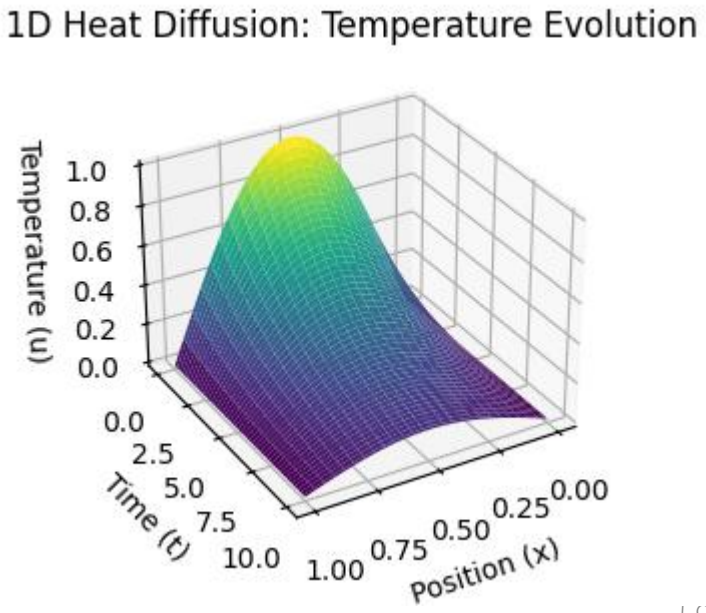
- Our goal: Find u, given λ and the initial condition $u_0 \coloneqq u(0, x)$

Heat Equation (cont'd)

- Consider $u(0, x) = \sin(\pi x)$



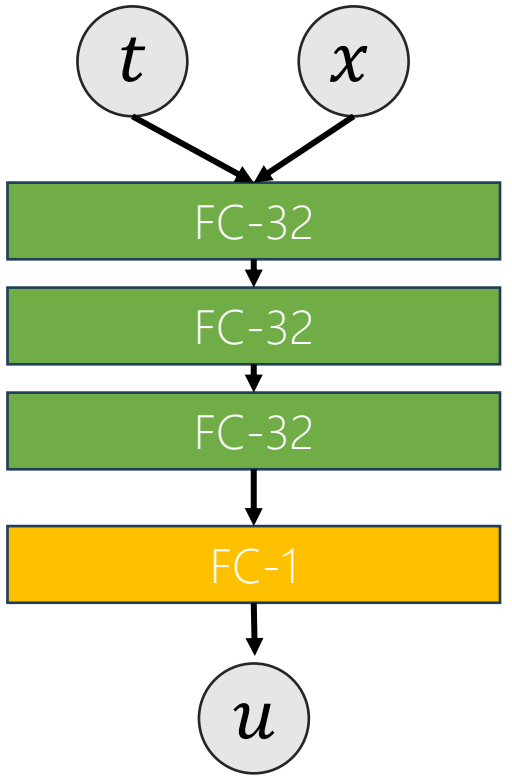
- Analytical solution for this IVP is known as $u(t, x) = e^{-\lambda\pi^2 t}\sin(\pi x)$:



Heat Equation (cont'd)

- … of course, we will "pretend" we don't know the solution yet.
- Design of the neural network $u_\theta(t, x)$:

```
class Backbone(nn.Module):
    def __init__(self, dtype=torch.float32):
        super().__init__()

        self.fc1 = nn.Linear(2, 32, dtype=dtype)  # input dim = 2 (t,x)
        self.fc2 = nn.Linear(32, 32, dtype=dtype)  # hidden dims = 32, 32, 32
        self.fc3 = nn.Linear(32, 32, dtype=dtype)  #
        self.out = nn.Linear(32, 1, dtype=dtype)  # output dim = 1 (u)

        self.dtype = dtype

    def forward(self, x):
        x = self.fc1(x)
        x = nn.SiLU()(x)
        x = self.fc2(x)
        x = nn.SiLU()(x)
        x = self.fc3(x)
        x = nn.SiLU()(x)
        return self.out(x)
```



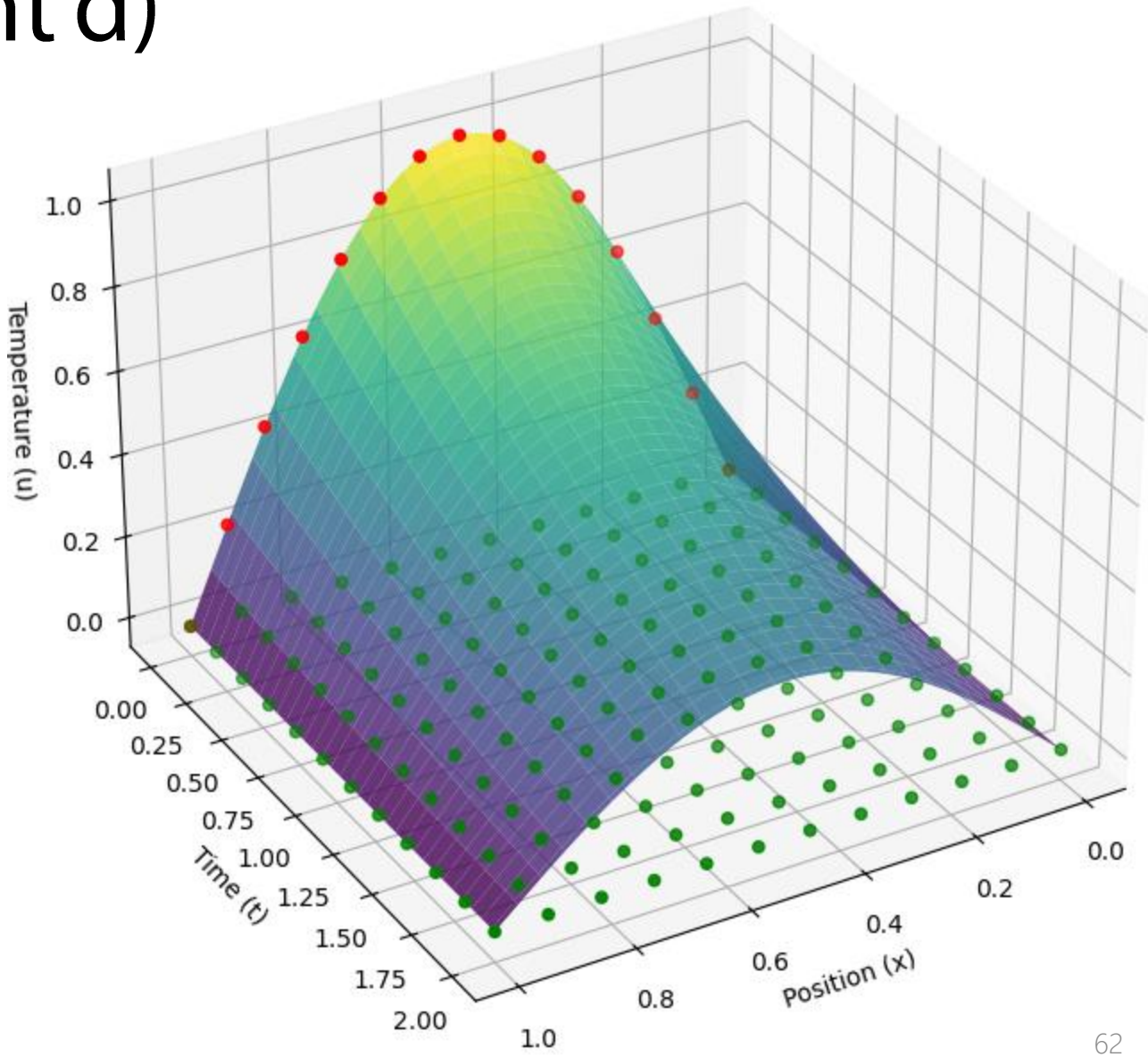
Heat Equation (cont'd)

- Physics-informed Loss

```
input, output = ground_truth

prediction = model(input)
data_loss = torch.mean((output-prediction)**2)

colloc_pred = model(colloc)

deriv = torch.autograd.grad(
    colloc_pred, colloc, torch.ones_like(colloc_pred),
    retain_graph=True, create_graph=True)[0]

dt = deriv[:,0]
dx = deriv[:,1]

deriv2 = torch.autograd.grad(dx, colloc, torch.ones_like(dx), create_graph=True)[0]
ddx = deriv2[:,1]

physics_loss = torch.mean((dt-alpha*ddx)**2)

loss = data_loss + physics_loss
loss.backward()

optimizer.step()
```

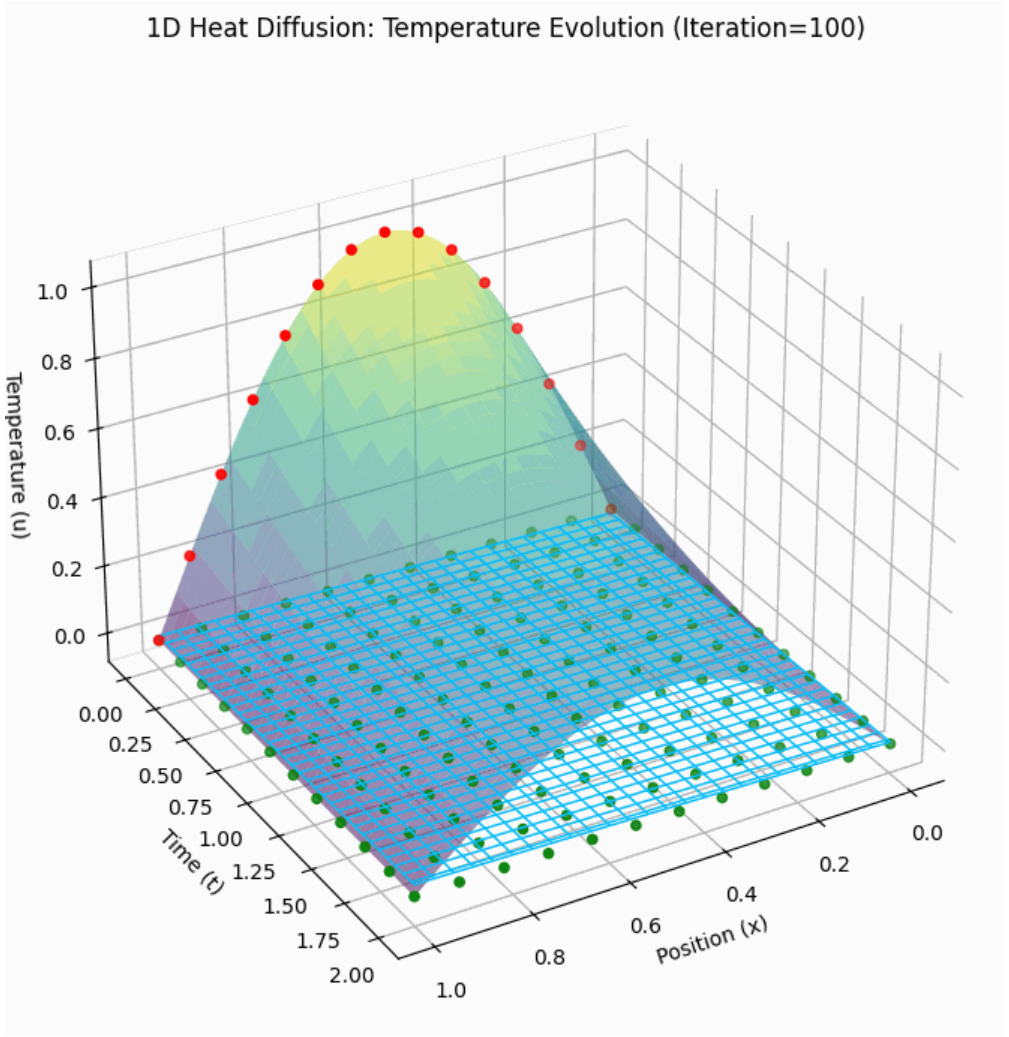
Heat Equation (cont'd)

- Data sampling
  - Initial condition
    - $u(0, x) = \sin(\pi x)$
  - Collocation points
    - Mesh grid with 12 grid points in each of the t and x directions



Heat Equation (cont'd)

- Training



Data-Driven Discovery: Example – NS Eqn.

- Navier-Stokes Equations:
  - Describe the physics of many fluid phenomena, such as water flow around an object, vehicle aerodynamics, ocean currents, weather, etc.

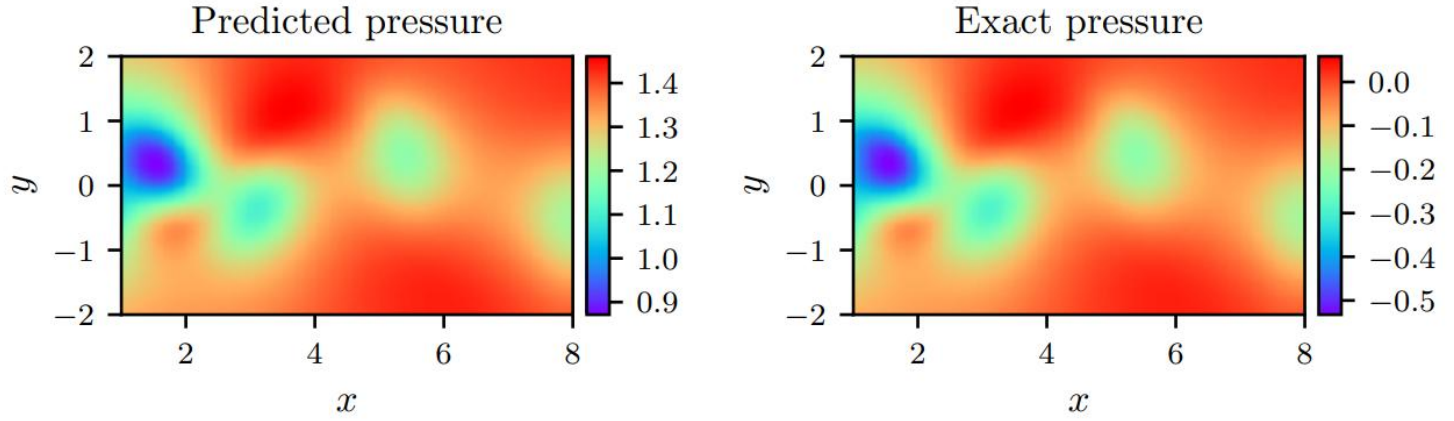
- N-S equation for 2D Incompressible Flows: $\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u}$
  - … or, equivalently:

$$\rho\left(u_t + uu_x + vu_y\right) = -p_x + \mu\left(u_{xx} + u_{yy}\right)$$
$$\rho\left(v_t + uv_x + vv_y\right) = -p_y + \mu\left(v_{xx} + v_{yy}\right)$$

  - where $u(t,x,y)$ and $v(t,x,y)$ denotes the x- and y-component of the evolving velocity field, $p(t,x,y)$ denotes the evolving pressure field, ρ is the fluid density, and μ is viscosity.
  - Conservation of mass: $u_x + v_y = 0$

Navier-Stokes Equations (cont'd)

- Approach: Set fluid density ρ and viscosity μ as learnable parameters, alongside other model parameters.



Correct PDE	$u_t + (uu_x + vu_y) = -p_x + 0.01(u_{xx} + u_{yy})$ $v_t + (uv_x + vv_y) = -p_y + 0.01(v_{xx} + v_{yy})$
Identified PDE (clean data)	$u_t + 0.999(uu_x + vu_y) = -p_x + 0.01047(u_{xx} + u_{yy})$ $v_t + 0.999(uv_x + vv_y) = -p_y + 0.01047(v_{xx} + v_{yy})$
Identified PDE (1% noise)	$u_t + 0.998(uu_x + vu_y) = -p_x + 0.01057(u_{xx} + u_{yy})$ $v_t + 0.998(uv_x + vv_y) = -p_y + 0.01057(v_{xx} + v_{yy})$

Raissi et al. (2017). Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations

Summary So Far

- PINN: adds the physical constraints as soft penalty terms in the training loss.

Pros:

- Enforcing physical constraints to NN predictions (to some extent)
- Easy to implement.

Challenges/Opportunities for PINN

- 1D Advection Equation: $\frac{\partial u}{\partial t} + \beta \frac{\partial u}{\partial x} = 0,\ x \in \Omega \subset \mathbb{R},\ t \in [0, T]$
  - Initial condition: $u(0, x) = \sin(x)$
  - Periodic BC: $u(t, 0) = u(t, 2\pi)$

$$\mathcal{L} = \frac{1}{N_f} \sum_{(t_f, x_f)} \|u_t + \beta u_x\|^2 \qquad \text{PDE Residual}$$

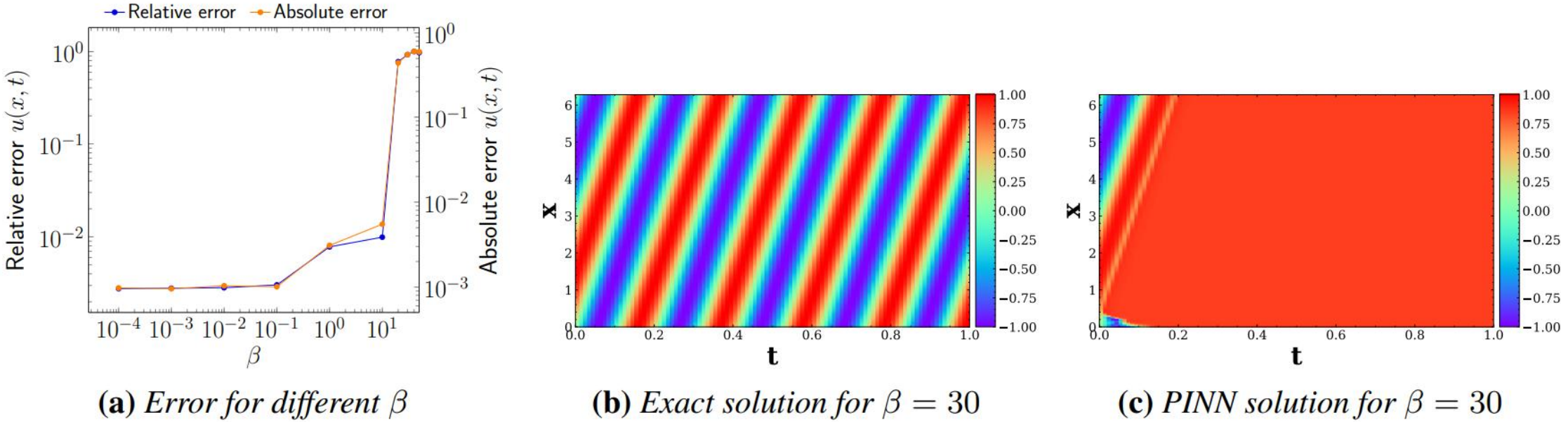
$$+ \frac{1}{N_i} \sum_{x_i} \|u(0, x) - \sin(x)\|^2 \qquad \text{Initial Condition}$$

$$+ \frac{1}{N_b} \sum_{t_b} \|u(t, 2\pi) - u(t, 0)\|^2 \qquad \text{Boundary Condition}$$

Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

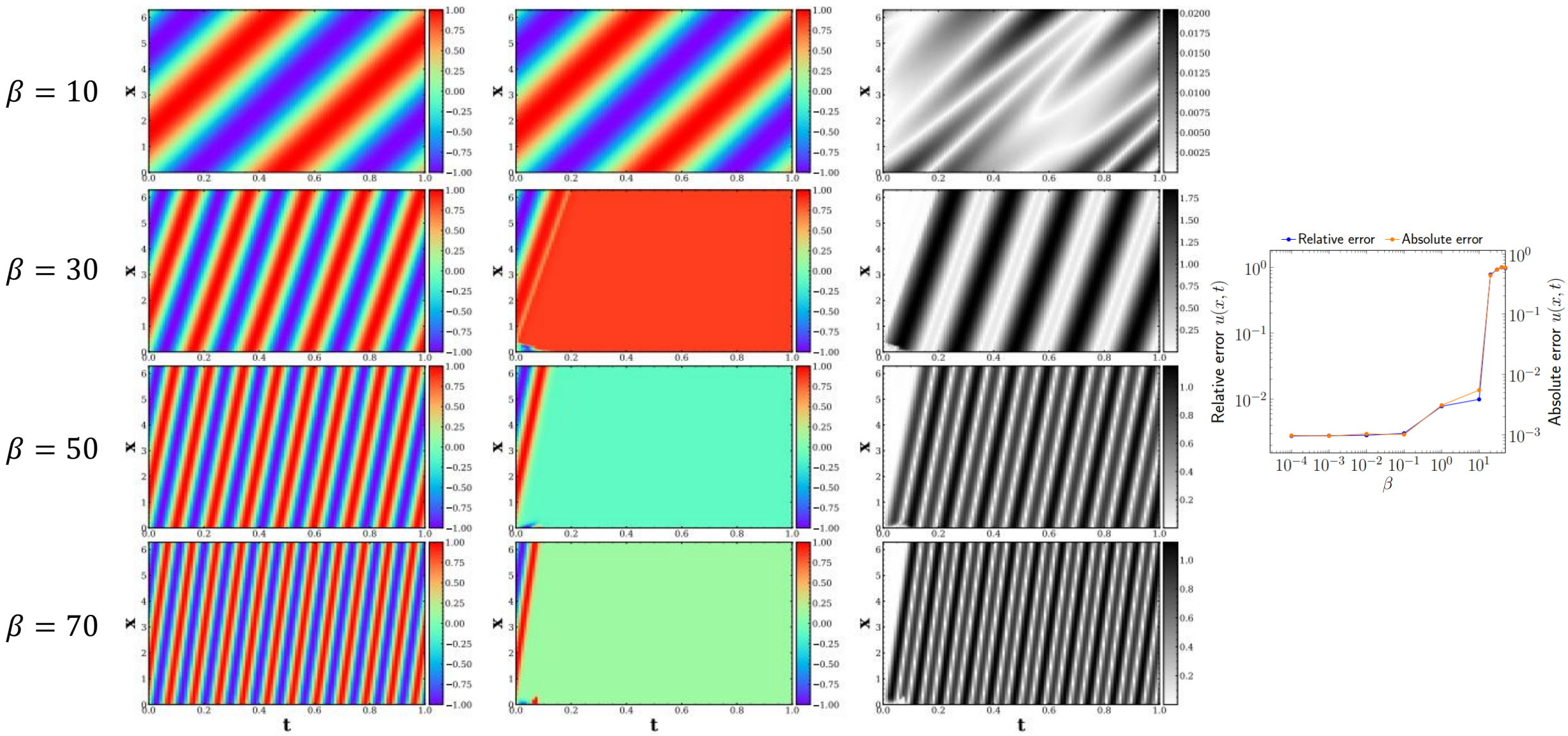
PINN on Advection

- 1D Advection Equation: $\frac{\partial u}{\partial t} + \beta \frac{\partial u}{\partial x} = 0,\ x \in \Omega \subset \mathbb{R},\ t \in [0, T]$
  - Initial condition: $u(0, x) = \sin(x)$
  - Periodic BC: $u(t, 0) = u(t, 2\pi)$



(a) *Error for different* β **(b)** *Exact solution for* $\beta = 30$ **(c)** *PINN solution for* $\beta = 30$

Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

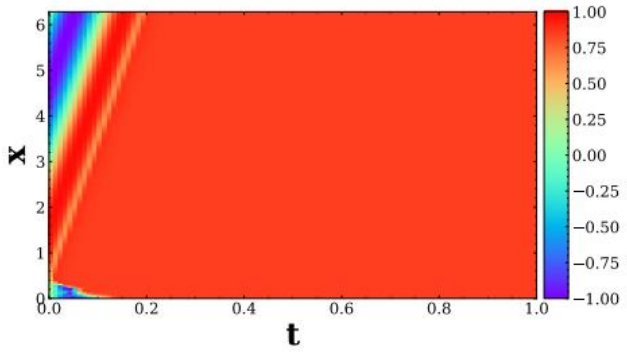


Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

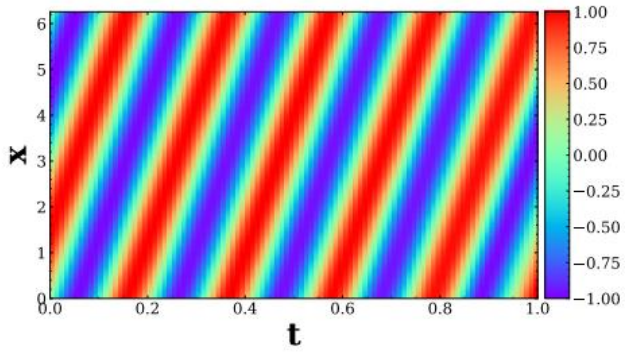
PINN on Advection

- Potential Solution: Curriculum Training

  "start by training the PINN on lower β (easier for the PINN to learn) and then gradually move to training the PINN on higher β."



(b) *Regular training PINN solution for* $\beta = 30$



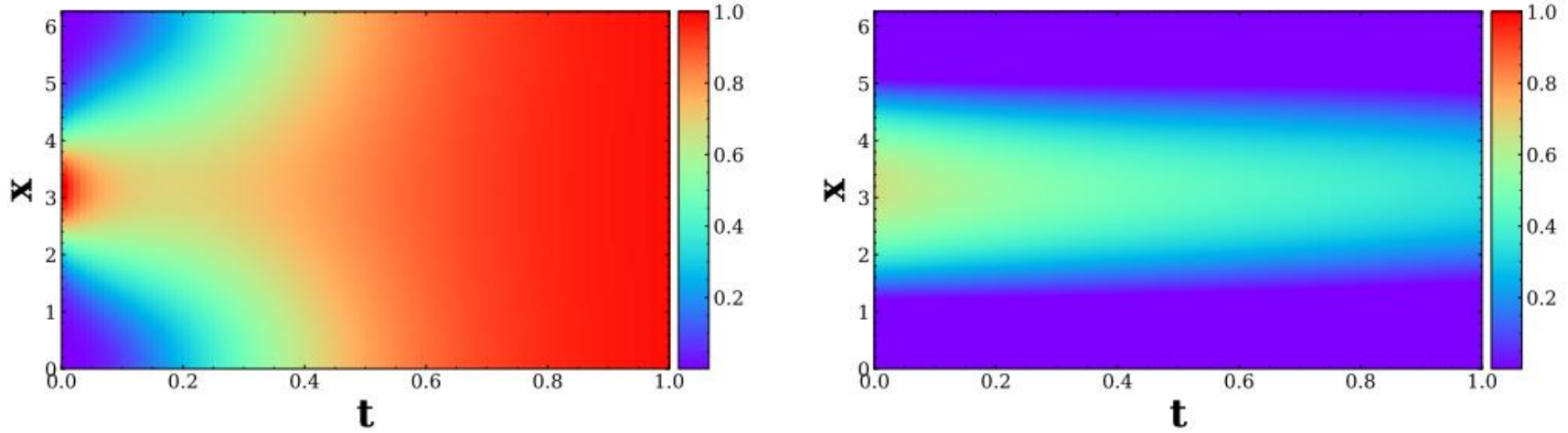
(c) *Curriculum training PINN solution for* $\beta = 30$

		Regular PINN	Curriculum training
$\beta = 20$	Relative error	7.50×10^{-1}	$\mathbf{9.84 \times 10^{-3}}$
	Absolute error	4.32×10^{-1}	$\mathbf{5.42 \times 10^{-3}}$
$\beta = 30$	Relative error	8.97×10^{-1}	$\mathbf{2.02 \times 10^{-2}}$
	Absolute error	5.42×10^{-1}	$\mathbf{1.10 \times 10^{-2}}$
$\beta = 40$	Relative error	9.61×10^{-1}	$\mathbf{5.33 \times 10^{-2}}$
	Absolute error	5.82×10^{-1}	$\mathbf{2.69 \times 10^{-2}}$

Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

Challenges/Opportunities for PINN

- 1D Reaction-Diffusion Equation: $\frac{\partial u}{\partial t} - \nu \frac{\partial^2 u}{\partial x^2} - \rho u(1-u) = 0,\ x \in \Omega,\ t \in [0, T]$
  - Initial condition: $u(0, x) = \exp\left(-\frac{(x-\pi)^2}{2(\pi/4)^2}\right)$
  - Periodic BC: $u(t, 0) = u(t, 2\pi)$



(a) *Exact solution for* $\rho = 5, \nu = 5$ **(b)** *PINN solution for* $\rho = 5, \nu = 5$

Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

Diffusion Coefficient

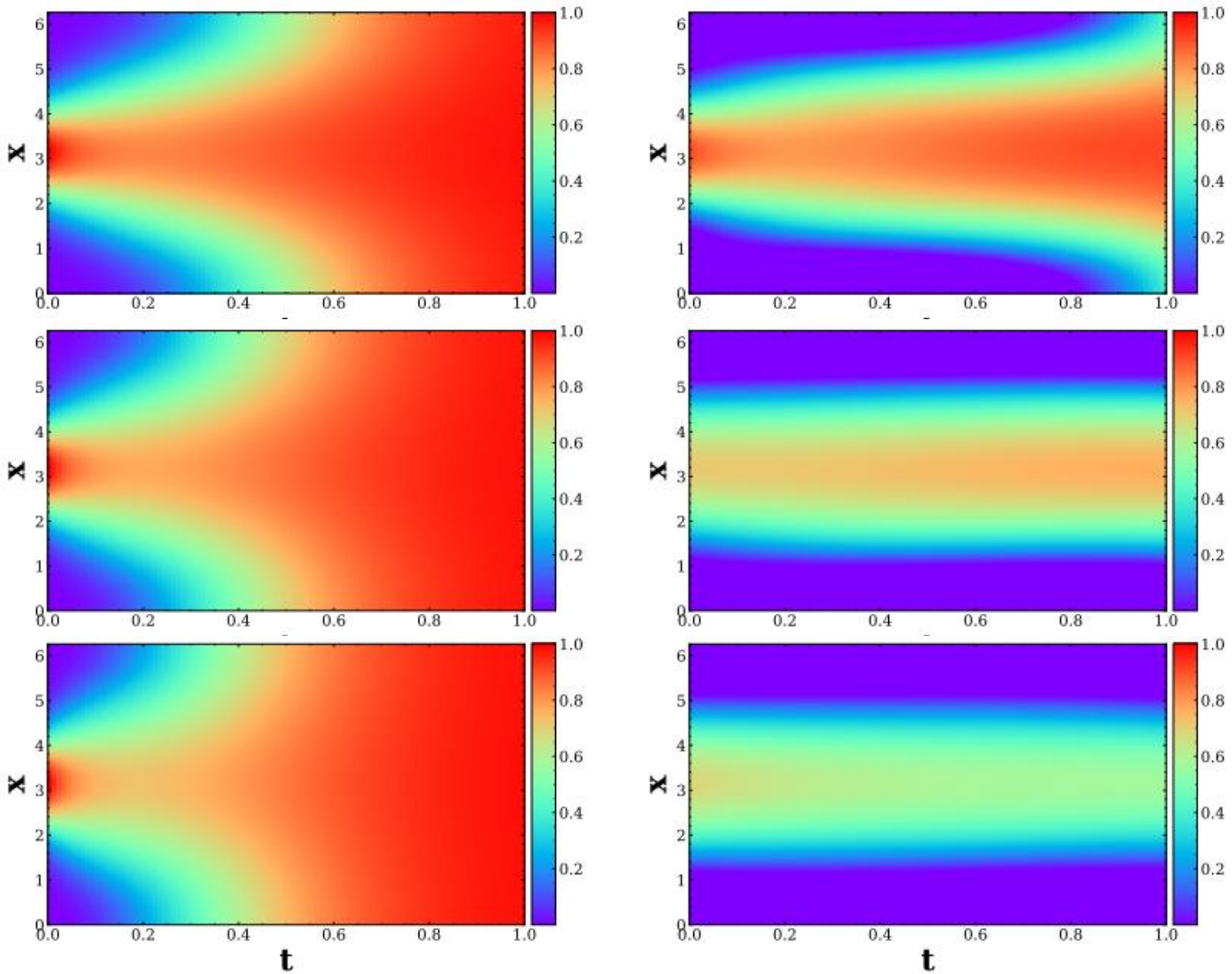
$\rho = 5, \nu = 2$

$\rho = 5, \nu = 3$

$\rho = 5, \nu = 4$

Exact Solution

PINN Solution

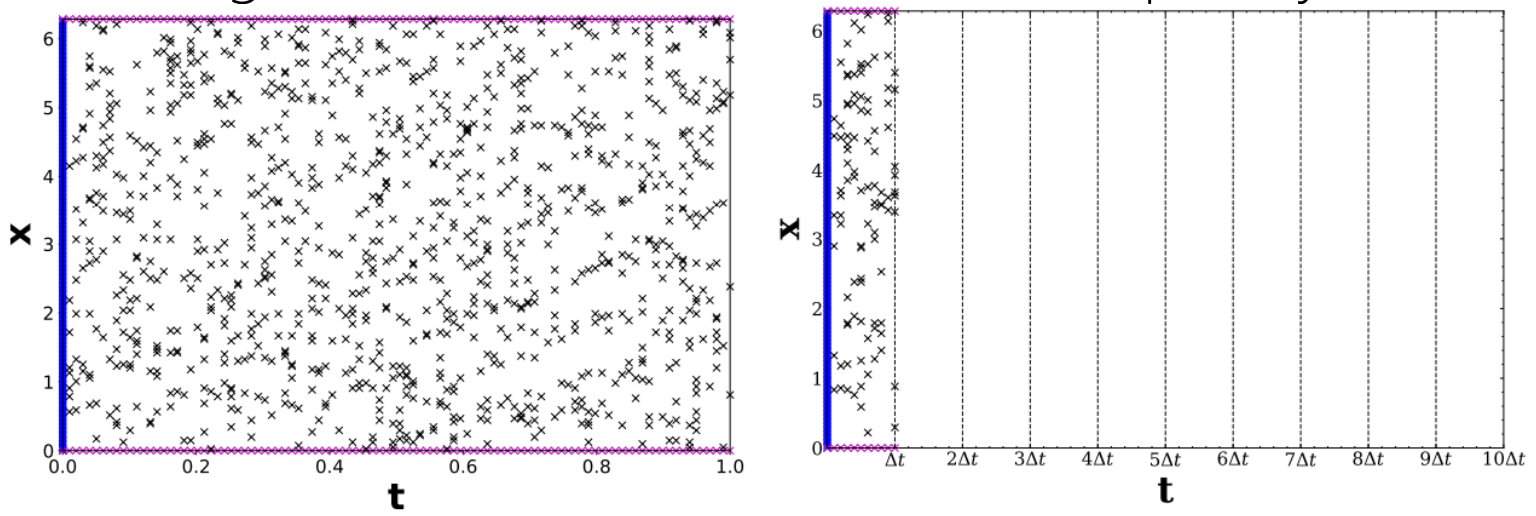


Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

PINN on Sharp Features

- Potential Solution: Sequence-to-Sequence Training

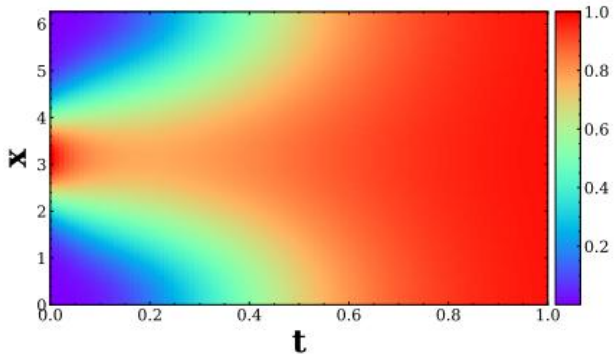
  "Predict(ing) the entire space-time at once (…) can be more difficult to learn." Instead, "predict(ing) the solution at the next time step" may be better.



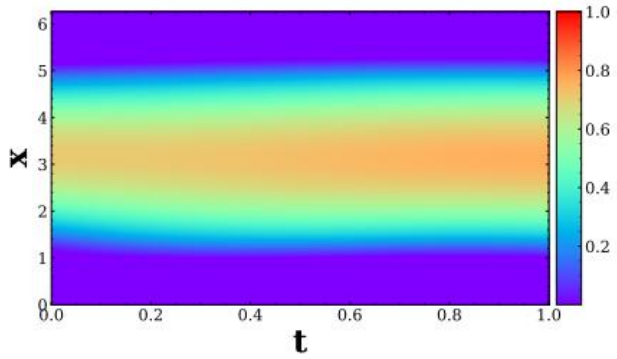
(a) *Regular PINN training*

(b) *Sequence-to-sequence learning (model trained every Δt)*

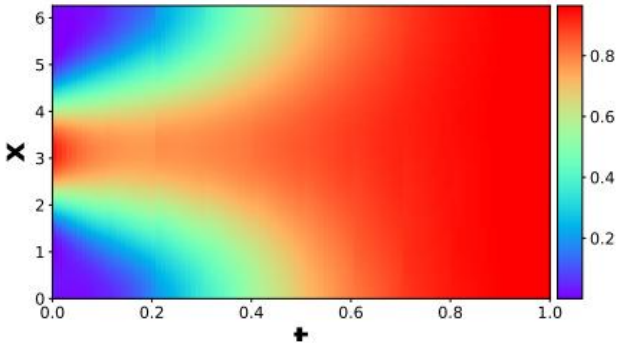
Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.



(a) *Exact solution for* $\rho = 5$, $\nu = 3$



(b) *Regular PINN prediction for* $\rho = 5$, $\nu = 3$



(c) *seq2seq PINN prediction for* $\rho = 5$, $\nu = 3$

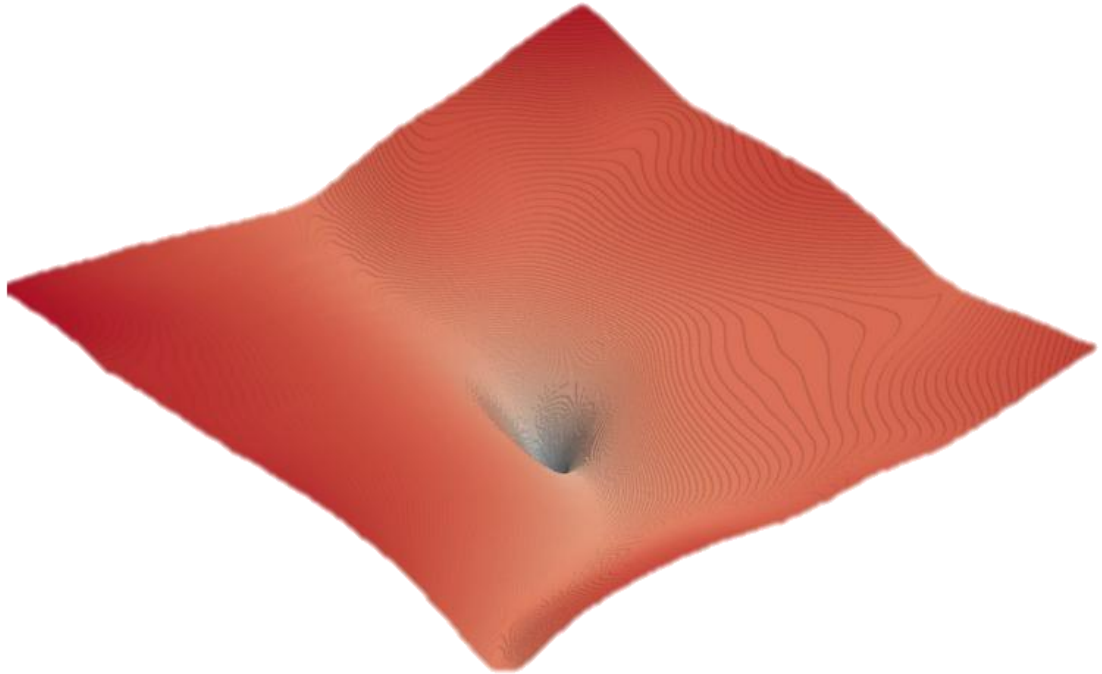
		Entire state space	$\Delta t = 0.05$	$\Delta t = 0.1$
$\nu = 2, \rho = 5$	Relative error	5.07×10^{-1}	2.04×10^{-2}	$\mathbf{1.18 \times 10^{-2}}$
	Absolute error	2.70×10^{-1}	1.06×10^{-2}	$\mathbf{6.41 \times 10^{-3}}$
$\nu = 3, \rho = 5$	Relative error	7.98×10^{-1}	1.92×10^{-2}	$\mathbf{1.56 \times 10^{-2}}$
	Absolute error	4.79×10^{-1}	1.01×10^{-2}	$\mathbf{8.17 \times 10^{-3}}$
$\nu = 4, \rho = 5$	Relative error	8.84×10^{-1}	2.37×10^{-2}	$\mathbf{1.59 \times 10^{-2}}$
	Absolute error	5.74×10^{-1}	1.15×10^{-2}	$\mathbf{8.01 \times 10^{-3}}$
$\nu = 5, \rho = 5$	Relative error	9.35×10^{-1}	$\mathbf{2.36 \times 10^{-2}}$	2.39×10^{-2}
	Absolute error	6.46×10^{-1}	$\mathbf{1.09 \times 10^{-2}}$	1.15×10^{-2}
$\nu = 6, \rho = 5$	Relative error	9.60×10^{-1}	2.81×10^{-2}	$\mathbf{2.69 \times 10^{-2}}$
	Absolute error	6.84×10^{-1}	$\mathbf{1.17 \times 10^{-2}}$	1.28×10^{-2}

Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

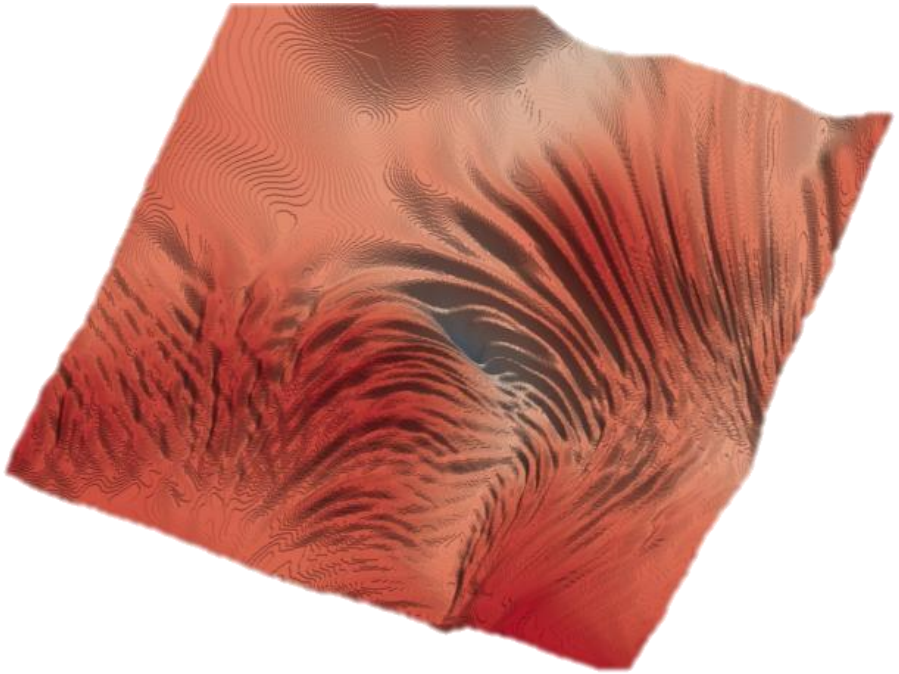
Optimization Challenges

- Change in the loss landscape caused by additional loss terms

Without Physics Loss



With Physics Loss



Images borrowed from Amir Gholami lecture slides (https://jduarte.physics.ucsd.edu/phys139_239/lectures/19_PINNs.pdf)

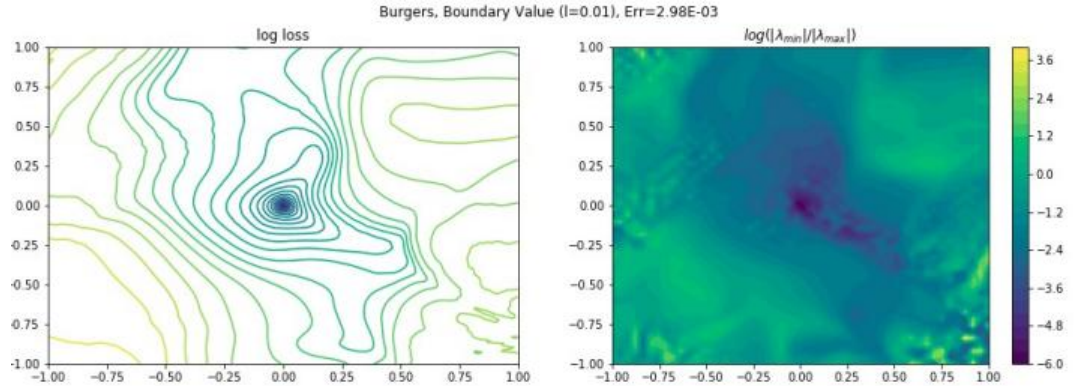
Optimization Challenges

- Change in the loss landscape caused by additional loss terms

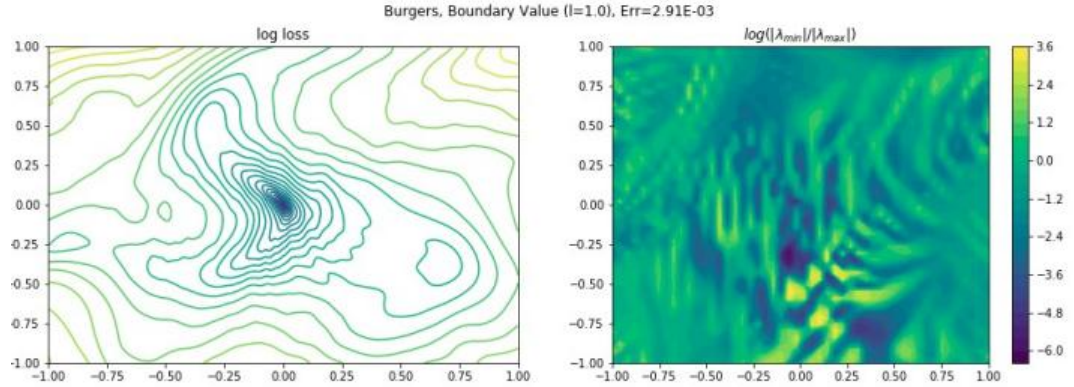
$$\mathcal{L} = \mathcal{L}_u + \alpha \mathcal{L}_f$$

Data Loss PDE Loss

$\alpha = 0.01$



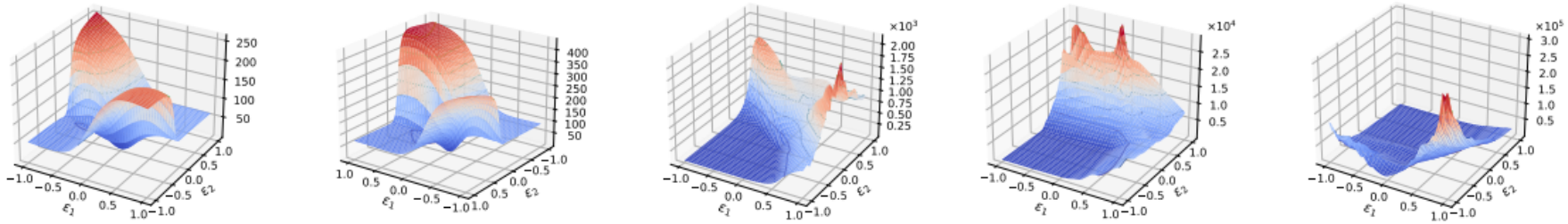
$\alpha = 1.0$



Images borrowed from Amir Gholami lecture slides (https://jduarte.physics.ucsd.edu/phys139_239/lectures/19_PINNs.pdf)

Optimization Challenges

- Adding PDE loss makes it harder to optimize



(a) $\lambda = 1 \times 10^{-6}$ (b) $\lambda = 1 \times 10^{-5}$ (c) $\lambda = 1 \times 10^{-3}$ (d) $\lambda = 1 \times 10^{-1}$ (e) $\lambda = 1 \times 10^{1}$

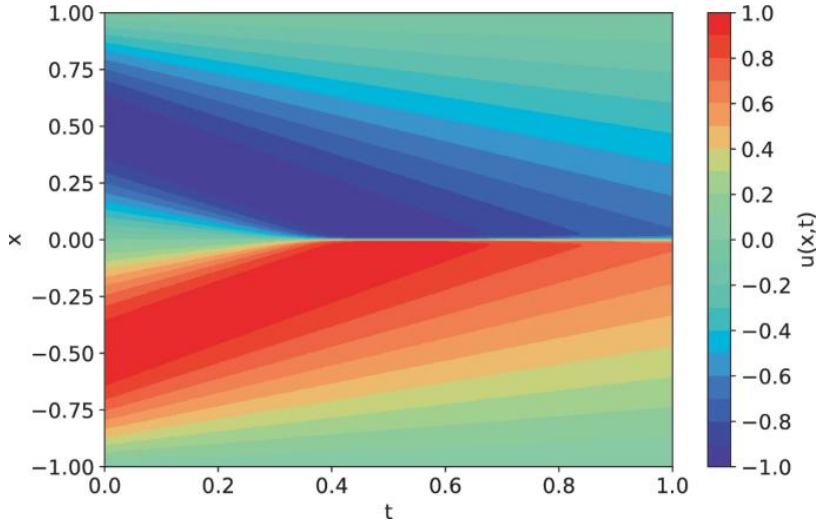
λ	1×10^{-6}	1×10^{-5}	1×10^{-3}	1×10^{-1}	1×10^{1}
Relative error	1.69	1.65	1.00	1.08	0.982
Absolute error	0.987	0.987	0.623	0.647	0.595

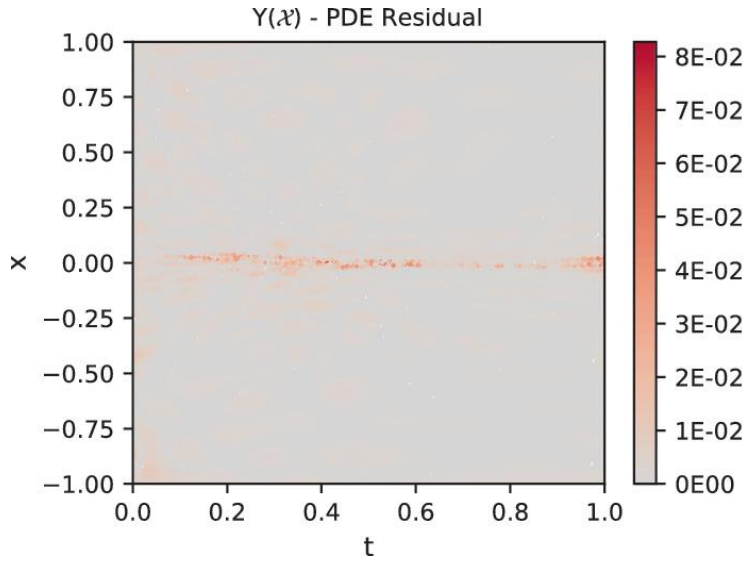
Figure E.1: ***Loss landscapes when varying the λ parameter in $\mathcal{F}$, for the 1D convection equation in §3.1.*** *In this example, $\beta = 30$, which is a point at which the error is high. The loss landscape becomes more complex as λ is increased, i.e., as the regularization term grows. However, error stays consistently high (although it decreases a little as λ is increased).*

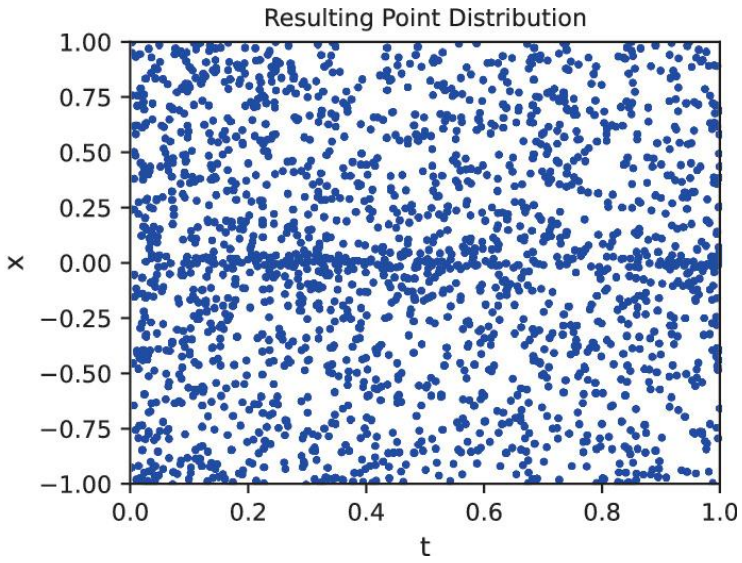
Krishnapriyan et al. (2021). Characterizing possible failure modes in physics-informed neural networks. NeurIPS.

Choice of Collocation Points

- Sampling bias in physics loss can be detrimental
- Potential solution: adaptive sampling (but at the cost of solving another optimization problem on the fly)







Low-Frequency Bias

- Neural nets (not just PINN) are inherently biased toward low-frequency patterns, because high-frequency patterns are viewed as “noise”.