Implicit Models

and Test-Time Compute

Jay Bear and Jonathon Hare

How deep should a network be?

About Me

Hi, I'm Jay!

I'm a PhD student in Vision, Learning, and Control.

I research recurrent and **implicit models** in deep learning.

My supervisors are Adam Prügel-Bennett and Jonathon Hare.

I love math and enjoy programming in Haskell.

Explicit vs Implicit

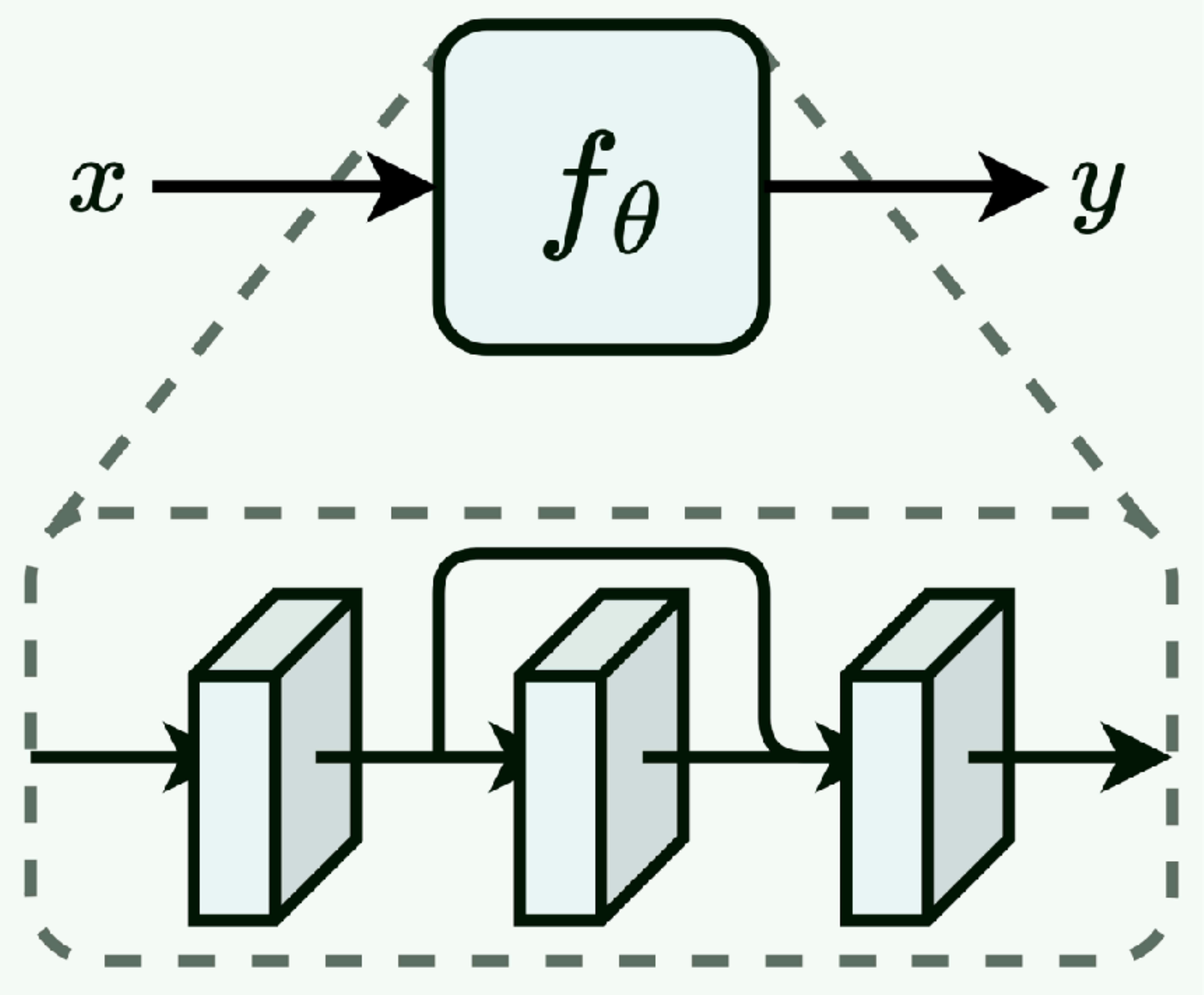
Explicit Models

Basically all models.

Models are often made up of **layers** or **blocks**.

These components can be defined as **explicit functions**:

- Generally $y = f_\theta(x)$, where $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$.
- Linear, convolution, multi-headed attention, residual, recurrent, etc.



Explicit Models

Why anything else?

Surely deep learning is just composing explicit functions?

Clearly explicit models do well:

- ResNets and vision transformers achieve human-level image recognition.
- LLMs produce human-like natural language.
- Cancer detection and radiology diagnostics made easier.
- Deep reinforcement learning can beat professionals in games.
- Near-human transcription accuracy with transformers.

Explicit Models

Why anything else?

There's still problems. They...

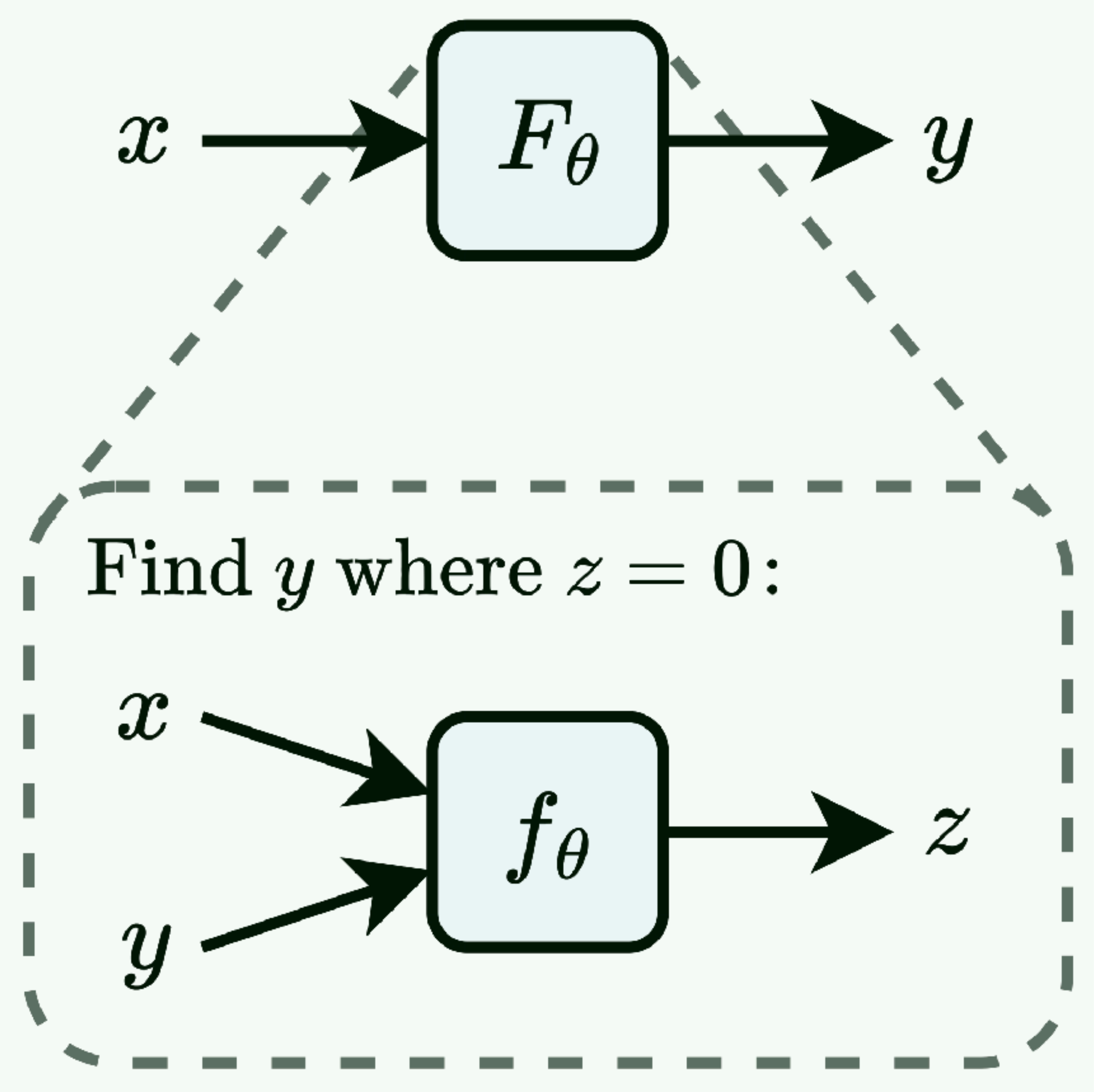
...require massive amounts of data, compute, and energy.

...struggle with out-of-distribution generalization.

...often lack robustness and interpretability.

...are vulnerable to adversarial attacks and subtle errors.

To move forward, we must not only refine the tools we know, but also seek out the tools we don't yet understand.



Implicit Models

All models... Plus more.

Instead, define components as solving **implicit functions**:

$$F_\theta(x) = y \quad \text{where} \quad f_\theta\big(x, y\big) = 0$$

f_θ is often a 'regular' architecture.

An iterative algorithm (a **solver**) is used to obtain y by finding **zeros**.

Can be stacked or used with other components.

Implicit Models

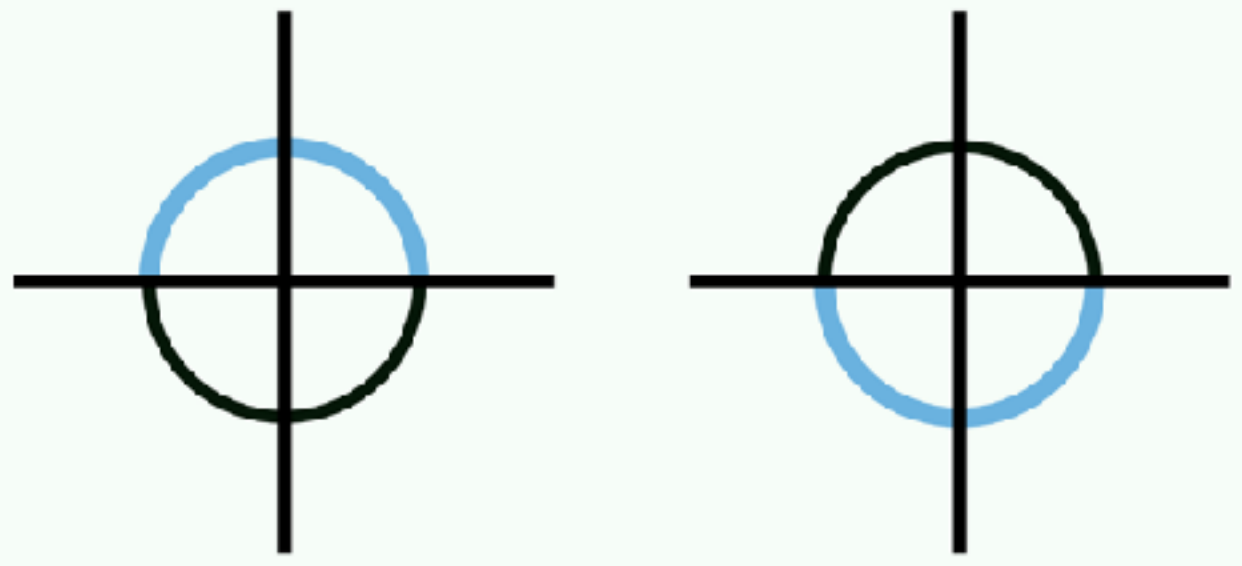
What can they do that explicit models can't?

Any explicit function $y = f(x)$ can be written implicitly:

$$F\left(x, y\right) = y - f(x) = 0$$

Not every implicit function can be written explicitly:

$F\left(x, y\right) = x^2 + y^2 - 1 = 0$, the unit circle, cannot be globally explicit.



Differentiating Implicit Models

Backpropagation?

Don't do it all the way.

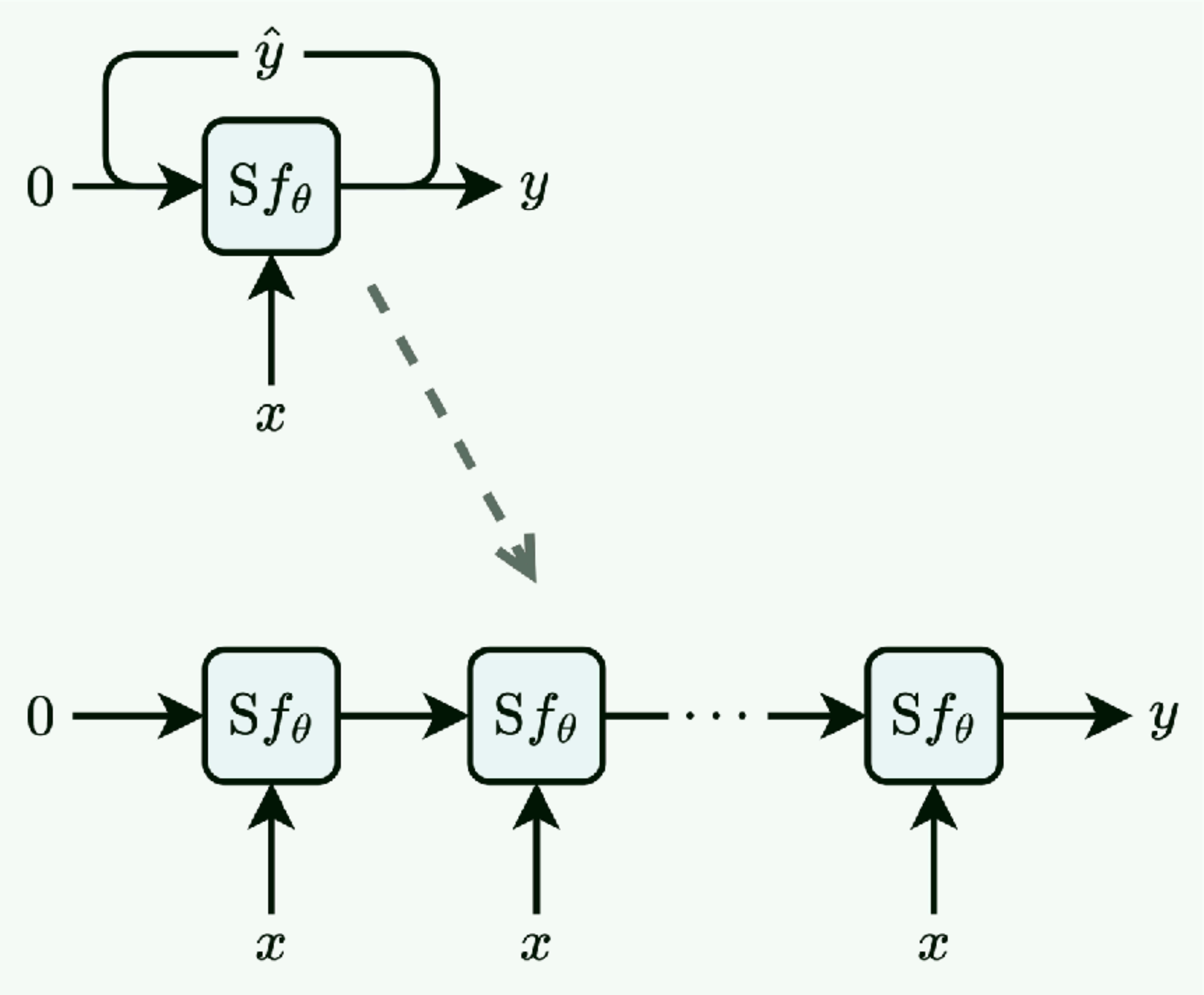
Can't we just **backpropagate** through F_θ ?

We can...

...but calculating $F_\theta(x)$ varies in uncontrollable complexity...

...and the solver often requires too many iterations.

The solution is the **implicit function theorem.**



Implicit Function Theorem

Implicit zeros are locally explicit function graphs.

Let $f: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be a continuously differentiable function and let $a \in \mathbb{R}^n, b \in \mathbb{R}^m$ such that $f\left(a, b\right) = 0$.

If Jacobian matrix $J_{f,y}\left(a, b\right)$ is invertible, then there exists an open set $U \subset \mathbb{R}^n$, with $a \in U$, such that there exists a unique function $g : U \to \mathbb{R}^m$, where $g(a) = b$ and $\forall x \in \mathbb{R}^n : f\left(x, g(x)\right) = 0$.

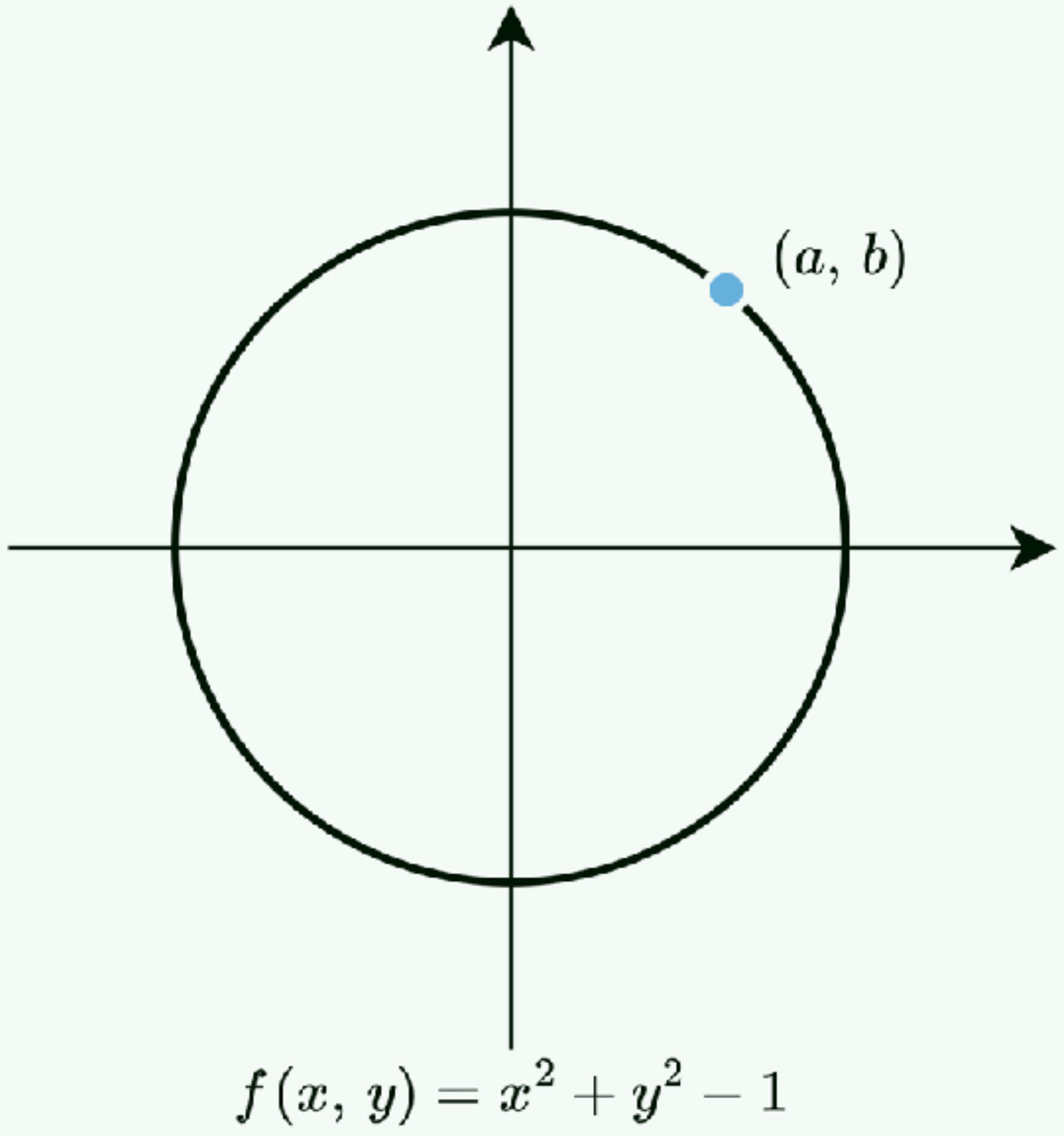
g is then continuously differentiable with Jacobian

$$J_g(x) = -\left[J_{f,y}\left(x, g(x)\right)\right]^{-1} J_{f,x}\left(x, g(x)\right)$$

Implicit Function Theorem

A unit circle example.

Let $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be a continuously differentiable function and let $a \in \mathbb{R}^n, b \in \mathbb{R}^m$ such that $f(a, b) = 0.$



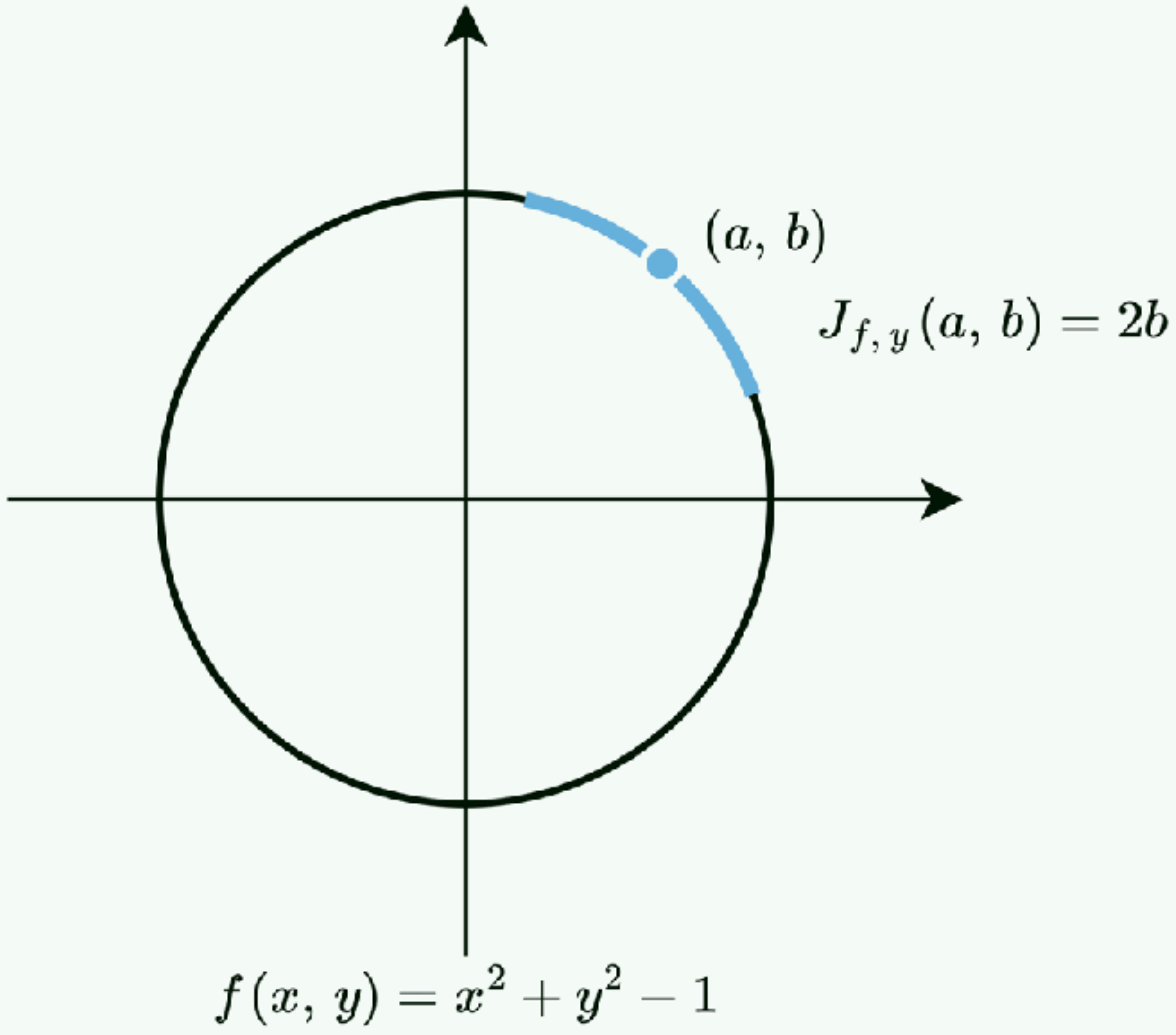
Unit circle $f(x, y) = x^2 + y^2 - 1$ with point (a, b) on its graph.

Implicit Function Theorem

A unit circle example.

If Jacobian matrix $J_{f,y}(a,b)$ is invertible, then there exists an open set $U \subset \mathbb{R}^n$, with $a \in U$, such that there exists a unique function $g : U \to \mathbb{R}^m$, where $g(a) = b$ and $\forall x \in \mathbb{R}^n : f\big(x, g(x)\big) = 0$.

Partial derivative $J_{f,y}(a,b) = 2b$ is invertible when $b \neq 0$.
g approximately represented in blue.



Implicit Function Theorem

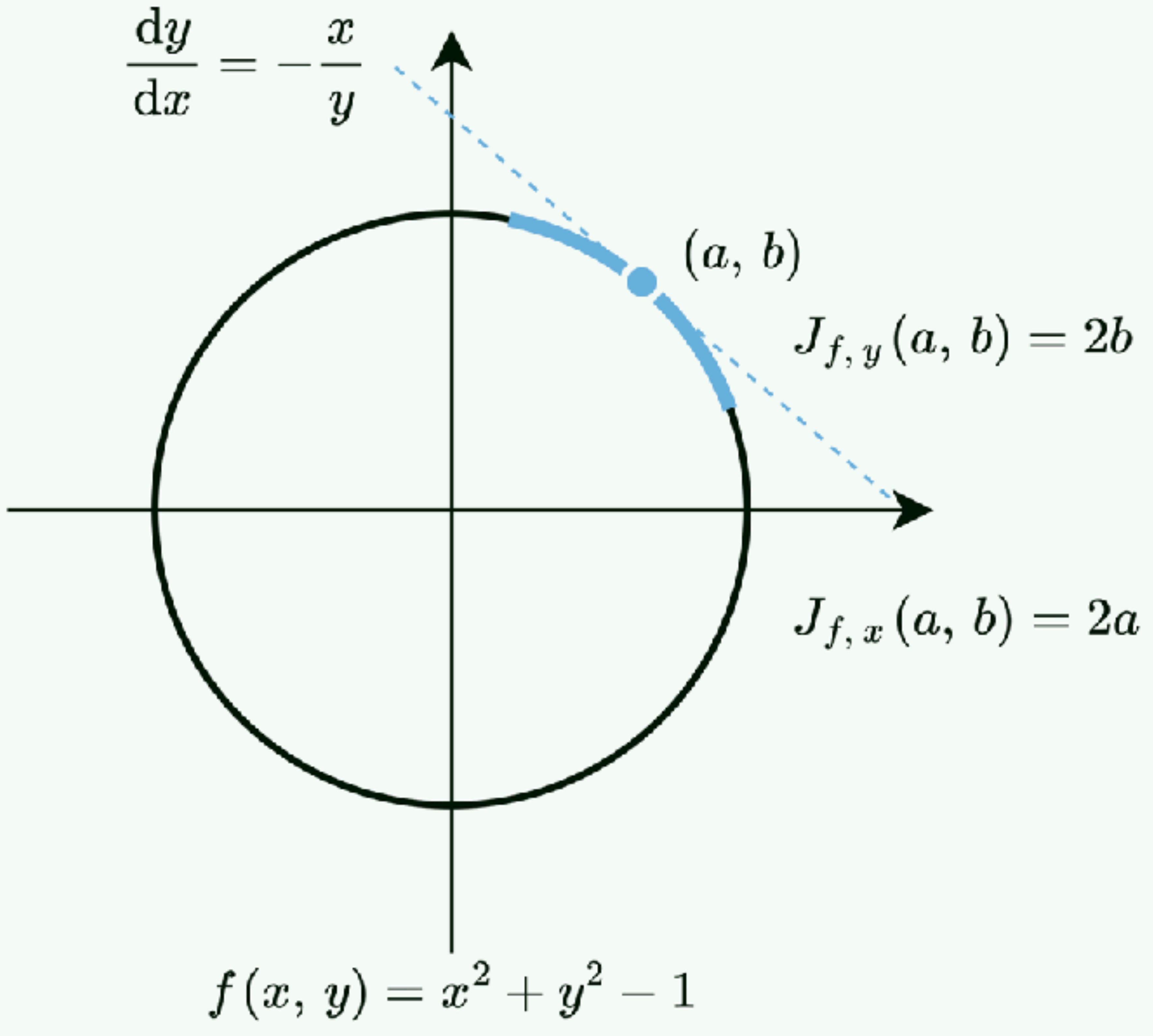
A unit circle example.

g is then continuously differentiable with Jacobian

$$J_g(x) = -\left[J_{f,y}\left(x, g(x)\right)\right]^{-1} J_{f,x}\left(x, g(x)\right)$$

Partial derivative $J_{f,x}\left(a, b\right) = 2a$.
Since $y = g(x)$;

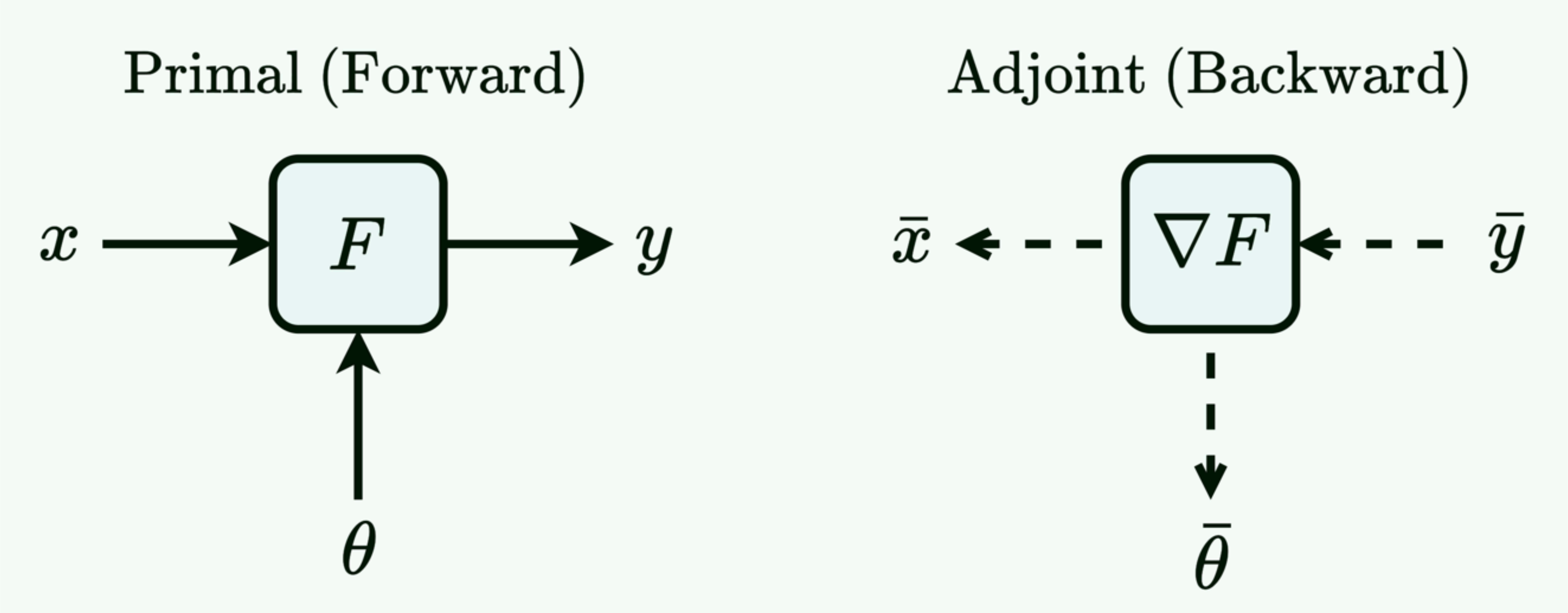
$$J_g(x) = -\frac{2x}{2g(x)} = -\frac{x}{y} = \frac{\mathrm{d}y}{\mathrm{d}x}$$



Autograd with Implicit Functions

Backpropagating with the implicit function theorem.

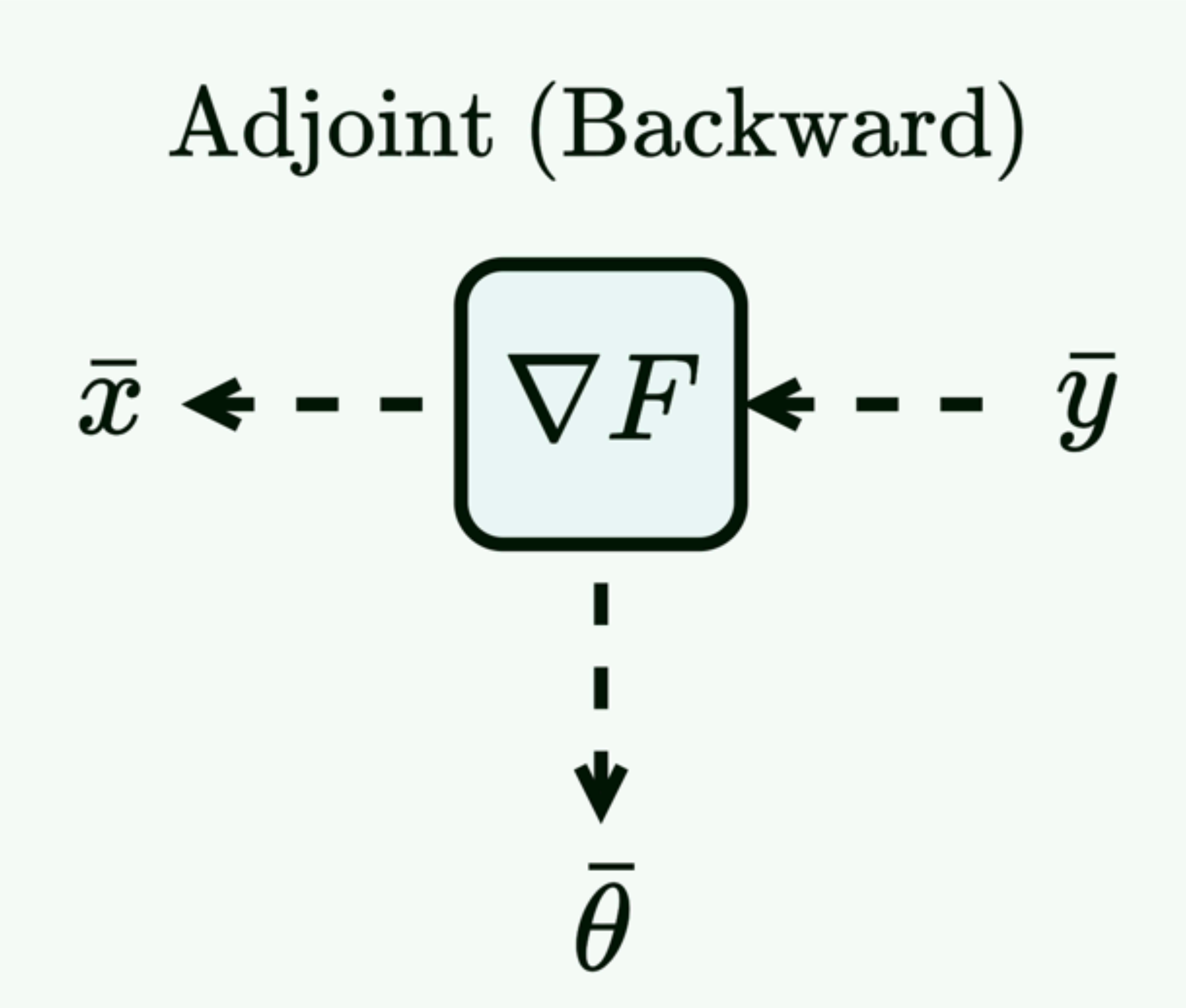
With F viewed as a function on input x and parameters θ:



Autograd with Implicit Functions

Backpropagating with the implicit function theorem.

x and θ can be viewed equivalently as $(\cdot)$ due to the implicit function theorem;



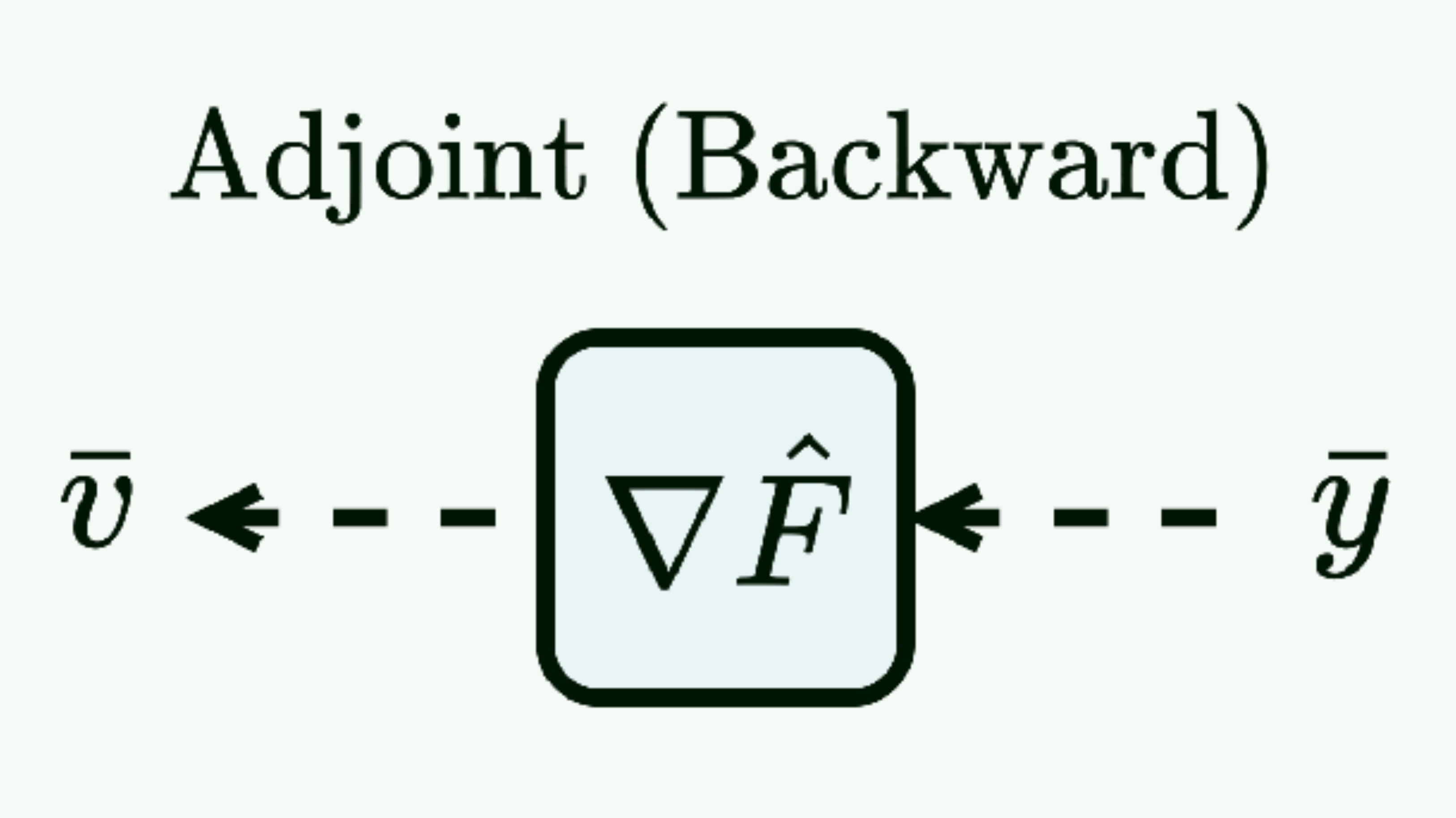
$$\bar{(\cdot)} = \left[-\left(\frac{\partial f}{\partial y}\right)^{-1}\left(\frac{\partial f}{\partial(\cdot)}\right)\right]^{\top}\bar{y} = -\left(\frac{\partial f}{\partial(\cdot)}\right)^{\top}\left(\frac{\partial f}{\partial y}\right)^{-\top}\bar{y}$$

but with $\bar{v} = -\left(\frac{\partial f}{\partial y}\right)^{-\top}\bar{y}$ such that $\bar{(\cdot)} = \left(\frac{\partial f}{\partial(\cdot)}\right)^{\top}\bar{v}$;

$$\left(\frac{\partial f}{\partial y}\right)^{\top}\bar{v} + \bar{y} = 0$$

Autograd with Implicit Functions

Backpropagating with the implicit function theorem.



Calculating $\bar{v}$ now just requires solving

$$\left(\frac{\partial f}{\partial y}\right)^{\top} \bar{v} + \bar{y} = 0$$

which is an implicit function!

The same solver we use for finding y in $f(x, y) = 0$ can also be used to find $\bar{v}$ in

$$\nabla \hat{f}(\bar{y}, \bar{v}) = \left(\frac{\partial f}{\partial y}\right)^{\top} \bar{v} + \bar{y} = 0$$

Autograd with Implicit Functions

Backpropagating with the implicit function theorem.

In PyTorch, this is relatively simple.

Use the solver to calculate y.

Clone, detach, and re-engage gradients with function call.

Use the solver on PyTorch's `autograd.grad` function to find $\bar{v}$.

```
# Forward:
y = solver(f, x)
# Backward:
y_in   = y.clone().detach().requires_grad_()
z_out  = f(x, y_in)
v_grad = solver(
  lambda g: torch.autograd.grad(
    outputs      = z_out,
    inputs       = y_in,
    grad_outputs = g,
    retain_graph = True
  )[0] + y_grad,
  y_grad
)
```

Types of Implicit Model

Common Types

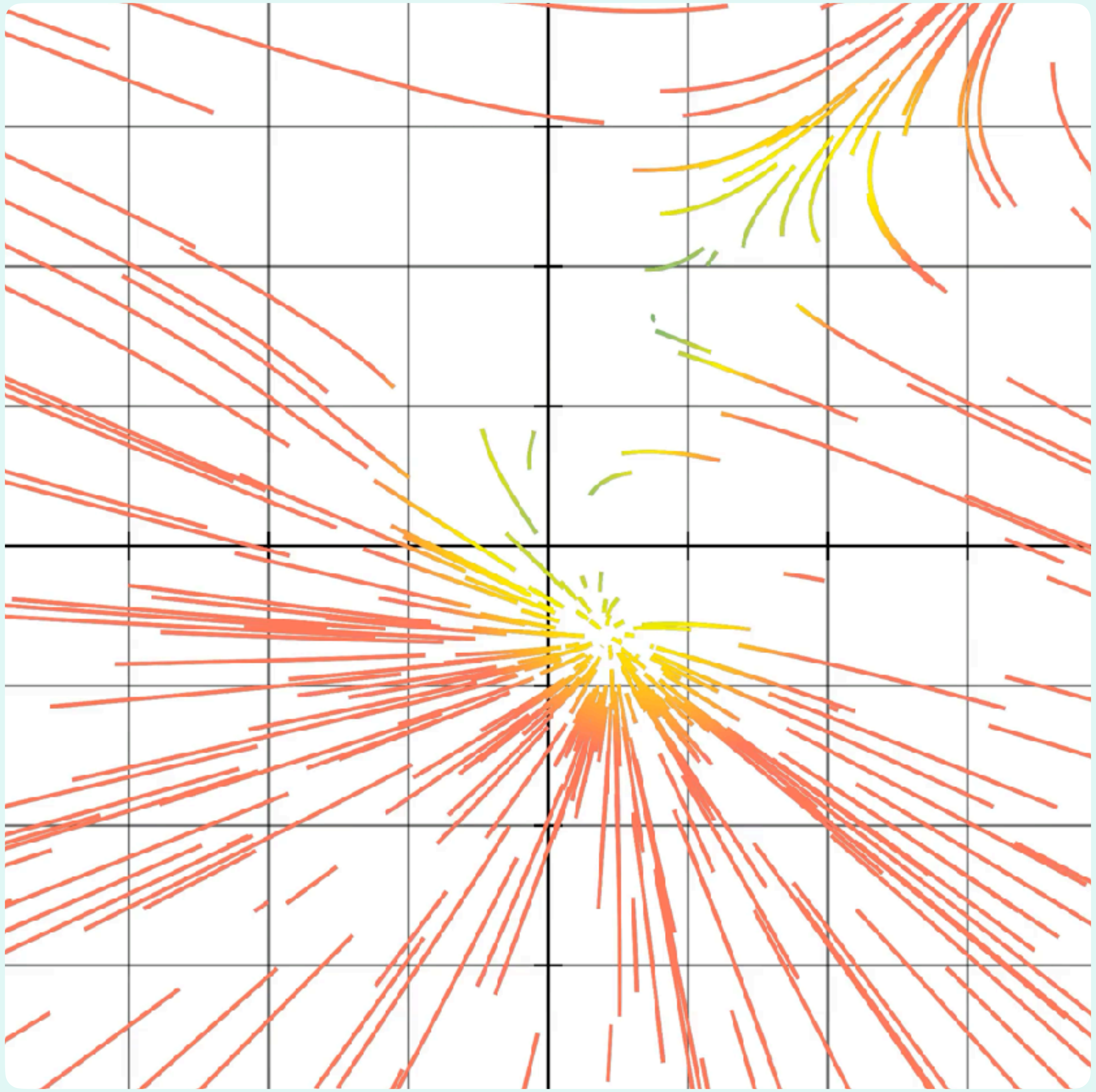
They're all basically the same.

- Root-finding implicit models:
  - Locate zeros.
- Neural ODEs:
  - Solve differential equations.
- Optimization networks:
  - Solve optimization problems.
- Deep equilibrium networks (DEQs):
  - Find fixed-points.

Common Types

They're all basically the same.

- Root-finding implicit models:
  - Locate zeros.
- Neural ODEs:
  - Solve differential equations.
- Optimization networks:
  - Solve optimization problems.
- Deep equilibrium networks (DEQs):
  - Find fixed-points.



Root-Finding Implicit Models

Locate zeros.

- Define some layer/block $f_\theta : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ with parameters θ, then

$$F_\theta(x) = \operatorname{sel}\left\{ y \in \mathbb{R}^m \;\middle|\; f_\theta\left(x, y\right) = 0 \right\}$$

$F_\theta(x)$ is some y where $f_\theta\left(x, y\right) = 0$

- Certain constraints and designs learn different processes:
  - $f_\theta = \nabla g_\theta \implies F_\theta$ is doing optimization. (optimization layer)
  - $f_\theta\left(x, y\right) = y - h_\theta\left(x, y\right) \implies F_\theta$ is locating fixed-points of h_θ. (DEQ layer)
  - $f_\theta\left(x, y\right) < 0 \Leftrightarrow y \in \Omega \implies F_\theta$ locates boundaries. (mostly)

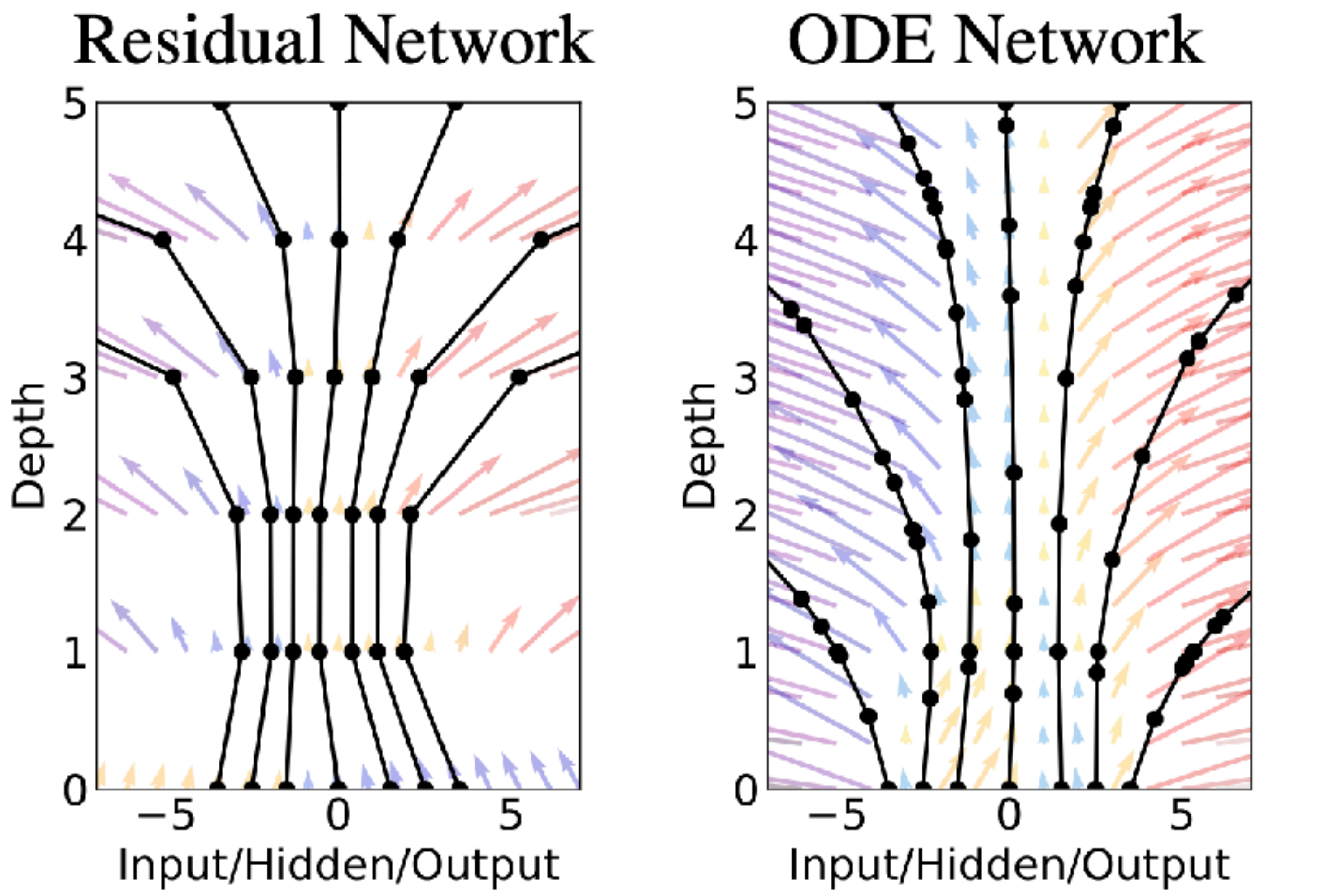


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

https://arxiv.org/pdf/1806.07366

Neural ODEs

Solve differential equations.

Neural networks can be used to specify ODEs;

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = f\big(h(t), t, \theta\big)$$

These can be solved – using **integration** – to find $h(T)$ at some time T.

Its gradient can be computed by integration too.

Opt. Networks

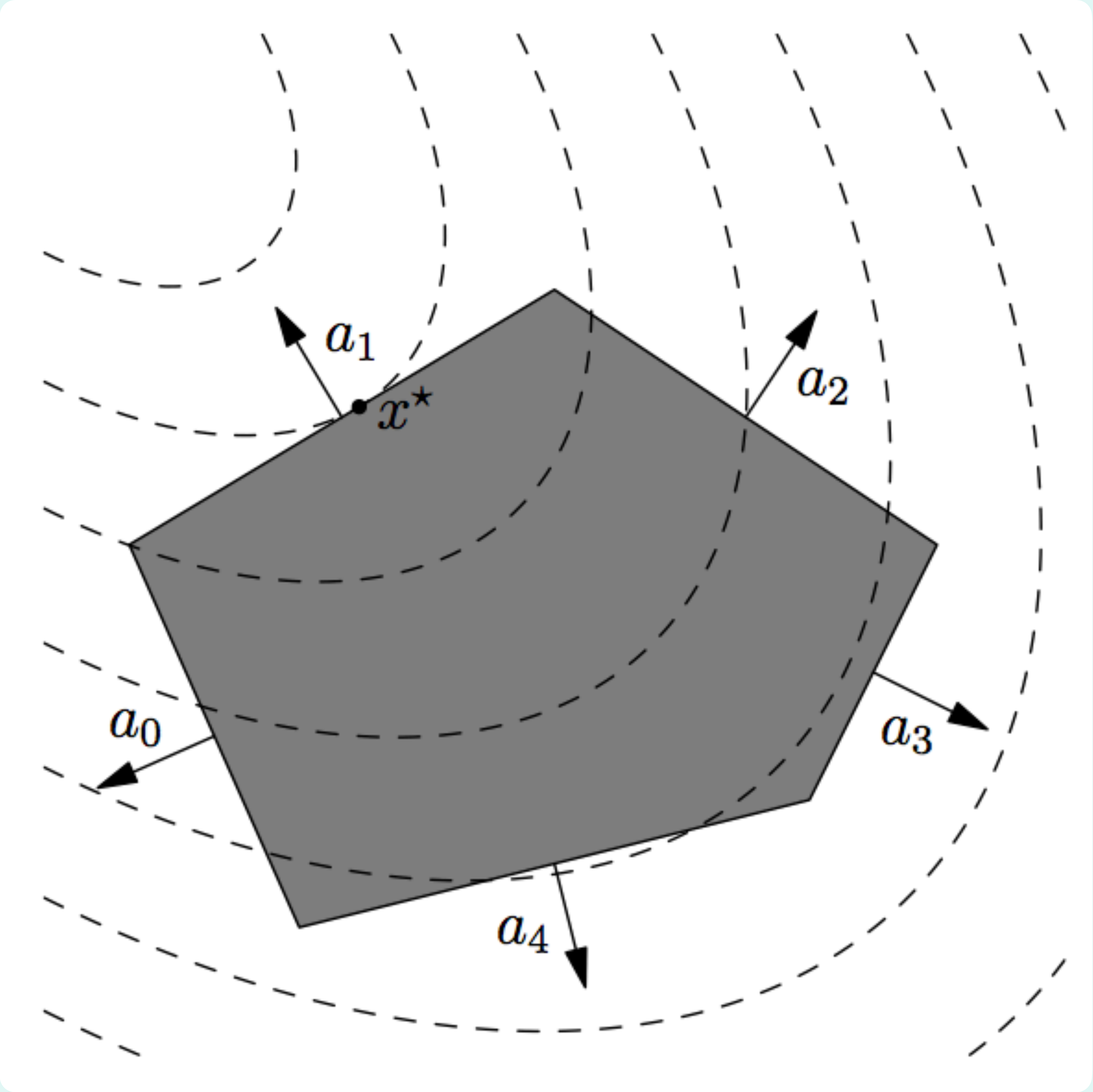
Solve optimization problems.

$f_\theta = \nabla g_\theta$ has zeros at **critical points** of g_θ.

If g_θ is **strictly convex** – either by constraint or design – then $F_\theta(x)$ has a **unique solution**.

Or ensure the Karush-Kuhn-Tucker conditions are met for uniqueness.

Often a **positive-definite** quadratic form with linear constraints.



Mosek ApS

Deep Equilibrium Networks

Find fixed-points.

Instead of finding zeros, find fixed-points where the function doesn't change:

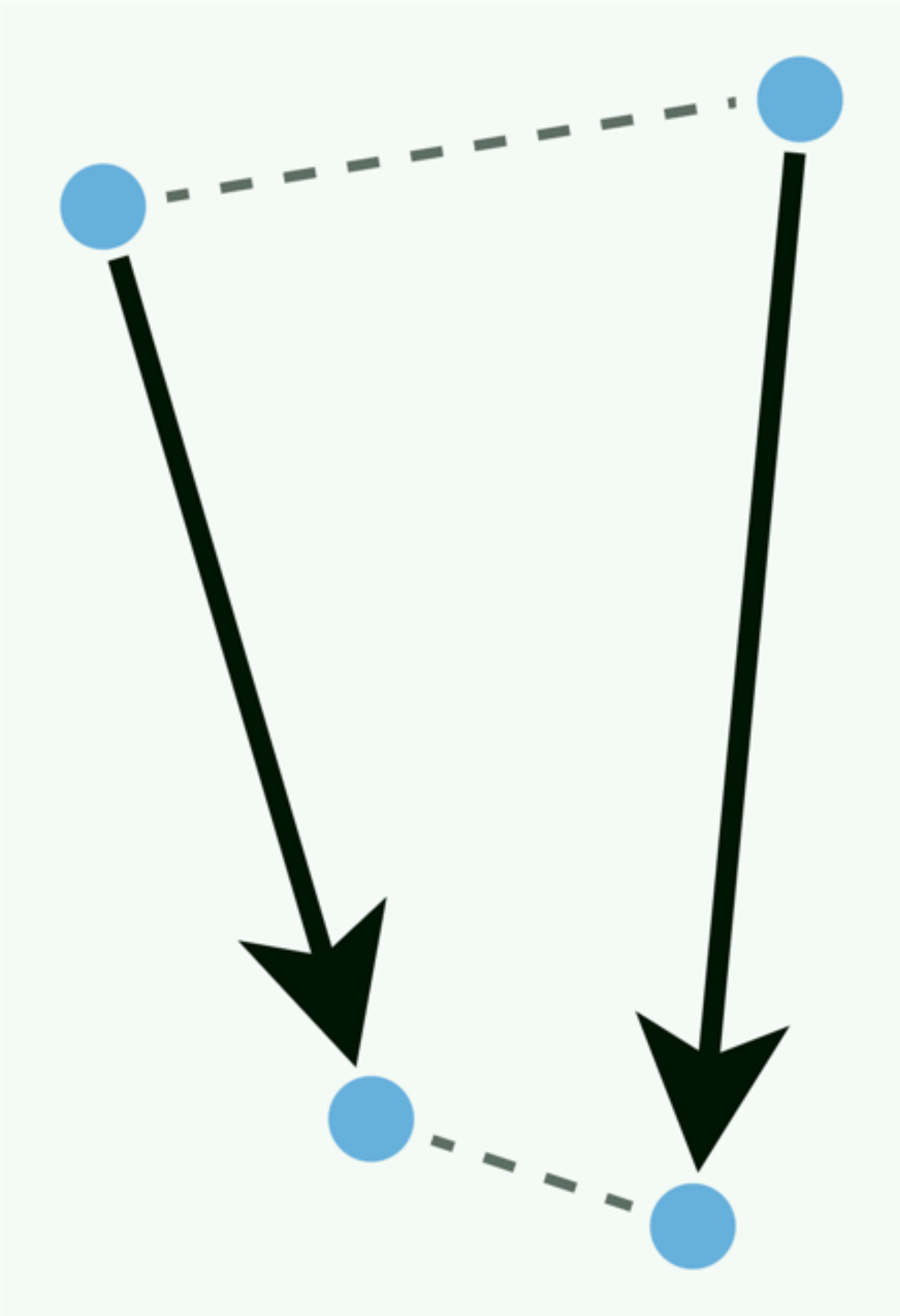
$$y^* = f_\theta\left(x, y^*\right)$$

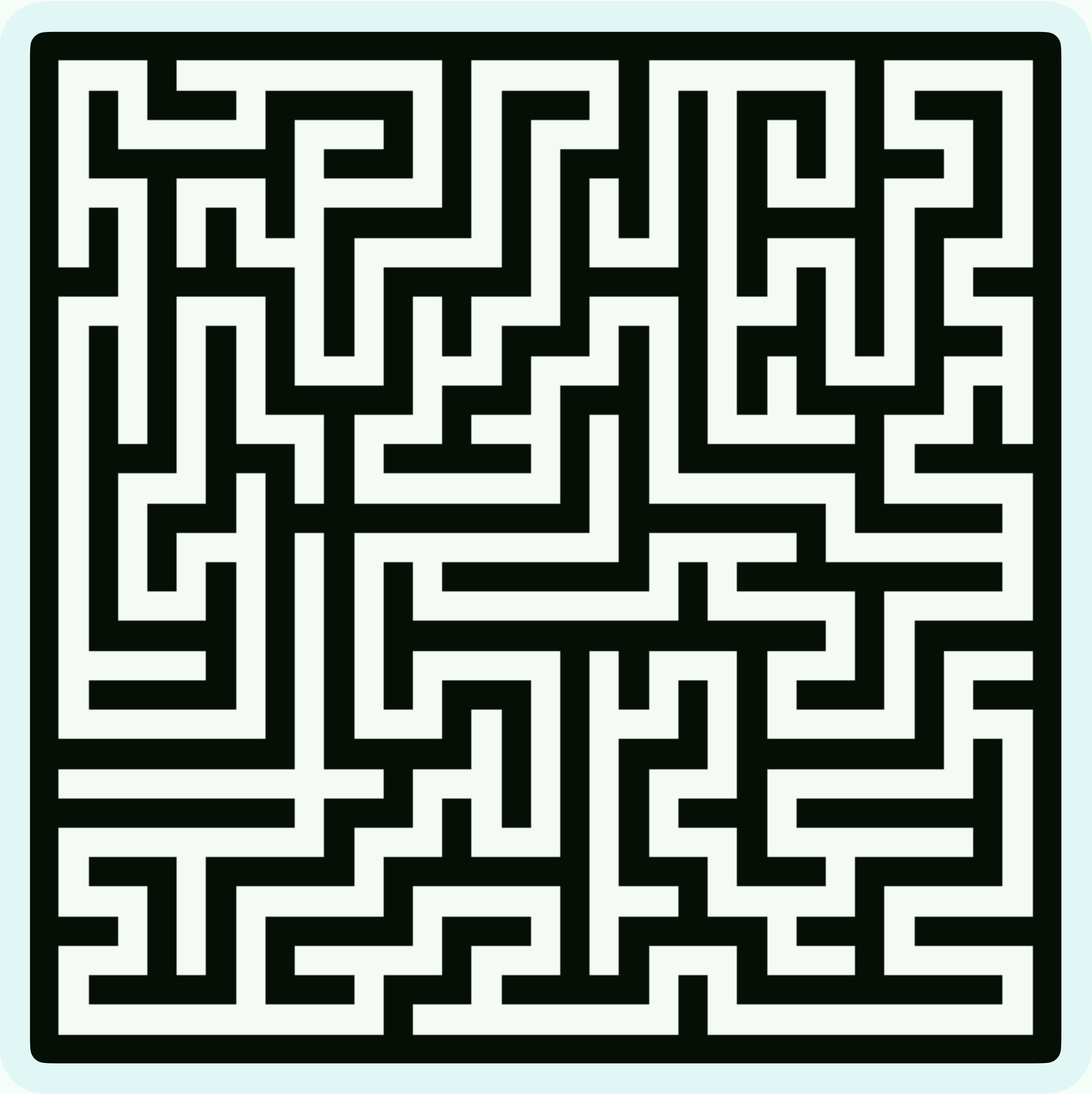
In many cases, y^* can be found through **recurrence**:

$$y^* = \lim_{m \to \infty} y^{(m)} \quad \text{where} \quad y^{(n+1)} = f_\theta\left(x, y^{(n)}\right)$$

This can be **accelerated** under certain constraints, such as f_θ being **contractive**.

The gradient <u>also</u> involves finding a fixed-point.





Problem Solving

Varying sizes and complexity.

What if we wanted to learn to solve mazes by drawing a path?

Such a model needs to handle mazes of different sizes and complexities.

Ideally exact solutions – no approximations.

How deep should a network be?

Learning Iterative Algorithms

Converging to a solution.

A deep equilibrium network can be considered to have **arbitrary depth** with **weight-tied components** when thought of as recurrence.

This is analogous to iteration in algorithms.

Deep equilibrium networks can learn to solve mazes through only examples!

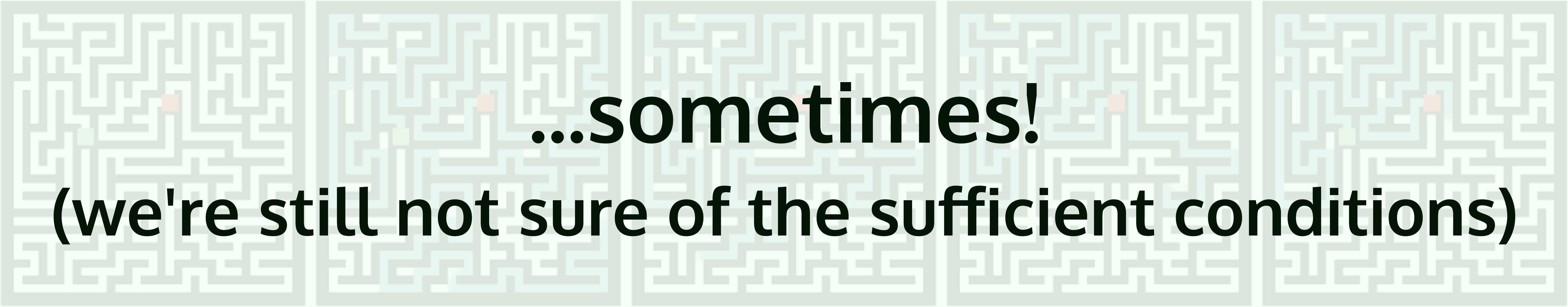
Learning Iterative Algorithms

Converging to a solution.

A deep equilibrium network can be considered to have **arbitrary depth** with **weight-tied components** when thought of as recurrence.

This is analogous to iteration in algorithms.

Deep equilibrium networks can learn to solve mazes through only examples...



...sometimes!

(we're still not sure of the sufficient conditions)

How deep should a network be?