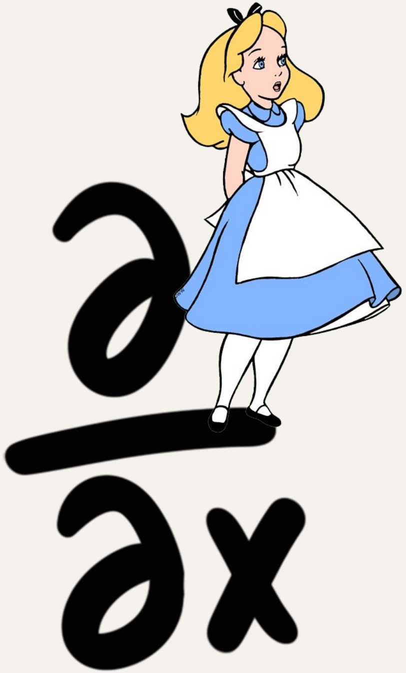
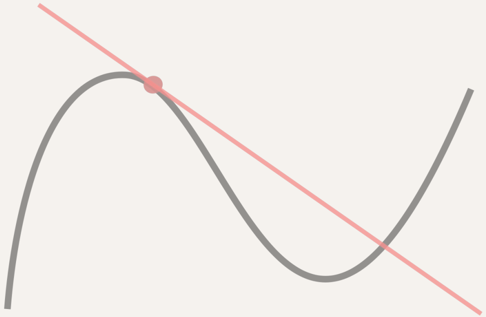
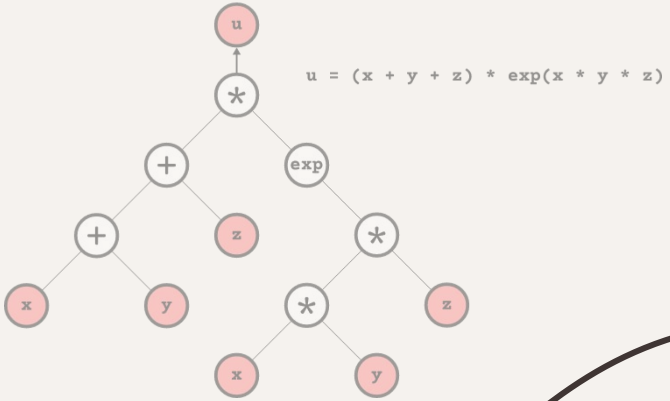
Alice's Adventures in a differentiable Wonderland



Presenter: Simone Scardapane





01

Differentiability: the key ingredient of AI?

"For, you see, so many out-of-the-way things had happened lately, that Alice had begun to think that very few things indeed were really impossible."

—Chapter 1, Down the Rabbit-Hole

A Path Towards Autonomous Machine Intelligence

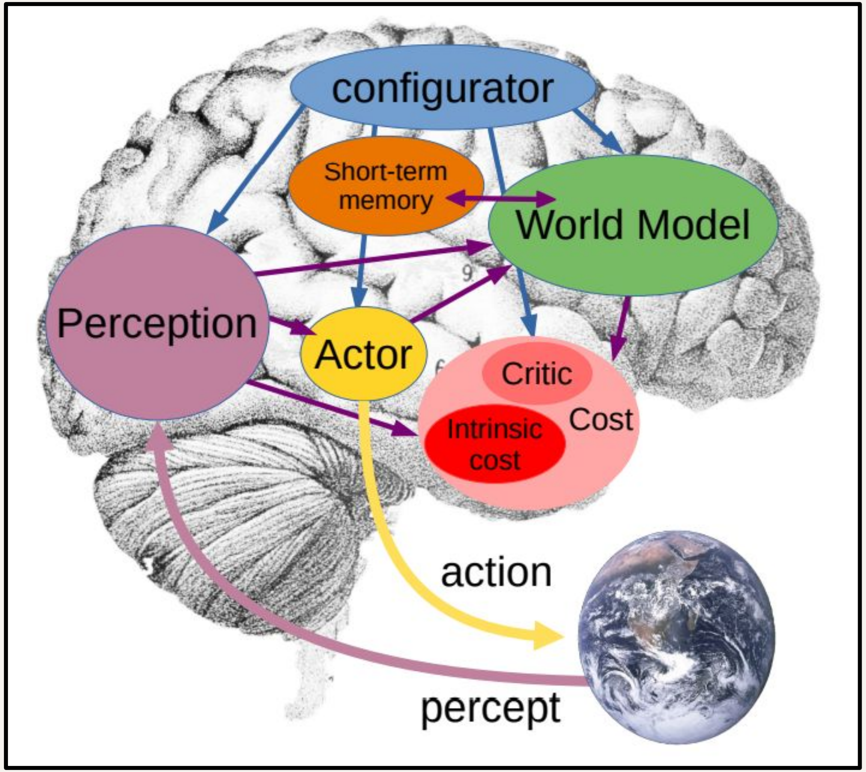
Yann LeCun

27 Jun 2022 OpenReview Archive Direct Upload Readers: Everyone

Abstract: How could machines learn as efficiently as humans and animals? How could machines learn to reason and plan? How could machines learn representations of percepts and action plans at multiple levels of abstraction, enabling them to reason, predict, and plan at multiple time horizons? This position paper proposes an architecture and training paradigms with which to construct autonomous intelligent agents. It combines concepts such as configurable predictive world model, behavior driven through intrinsic motivation, and hierarchical joint embedding architectures trained with self-supervised learning.

Add **Comment**

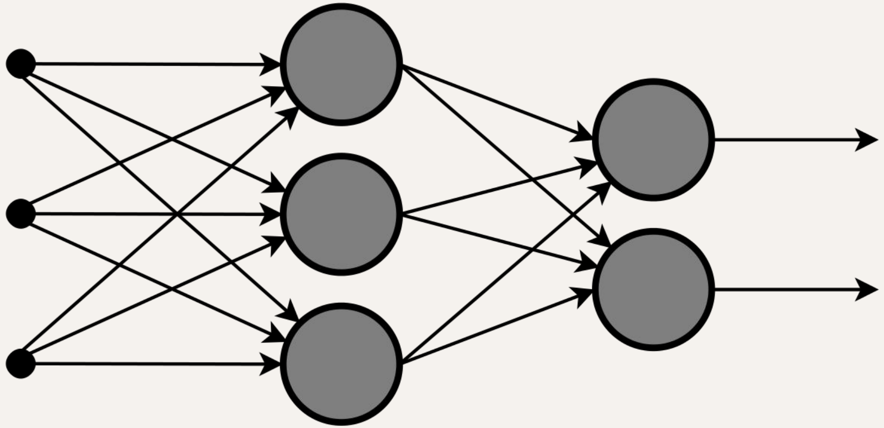
"This position paper proposes an architecture and training paradigms with which to construct autonomous intelligent agents."



“A system architecture for autonomous intelligence. All modules in this model are assumed to be **differentiable**.”

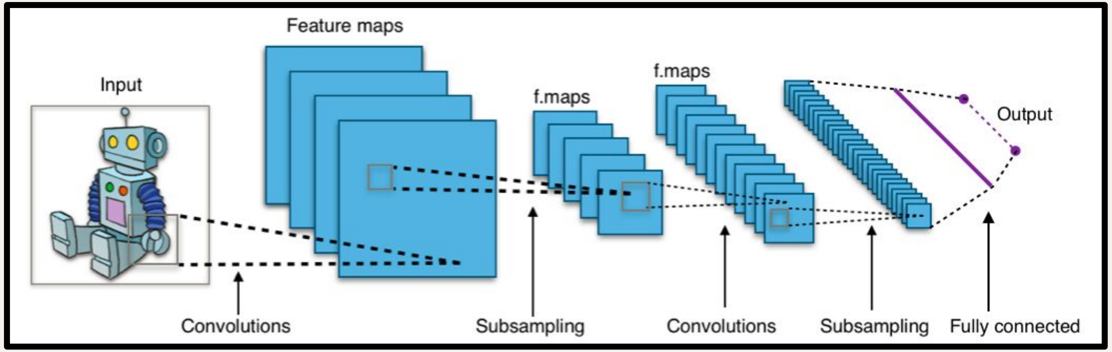
What is an “artificial neural network”?

No biology, please.

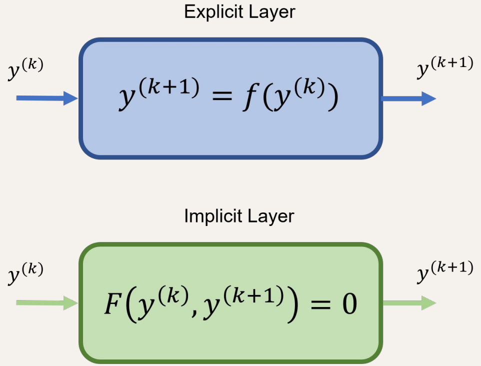


« **computing systems** vaguely inspired by the **biological neural networks** that constitute animal **brains** »
— Wikipedia

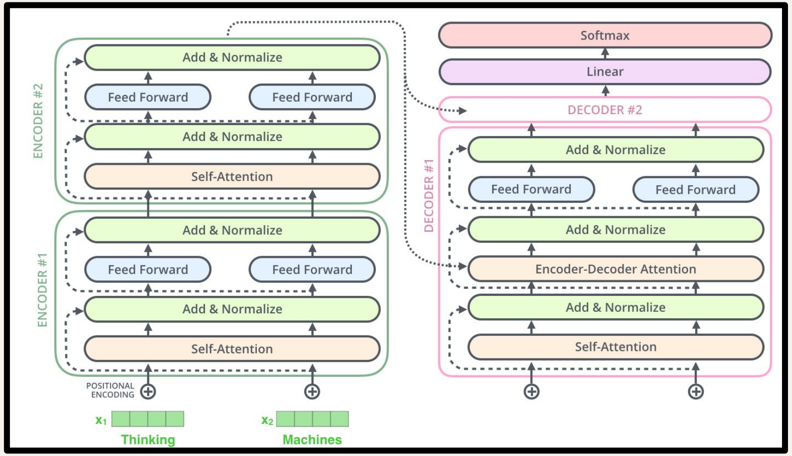
Convolutional neural networks



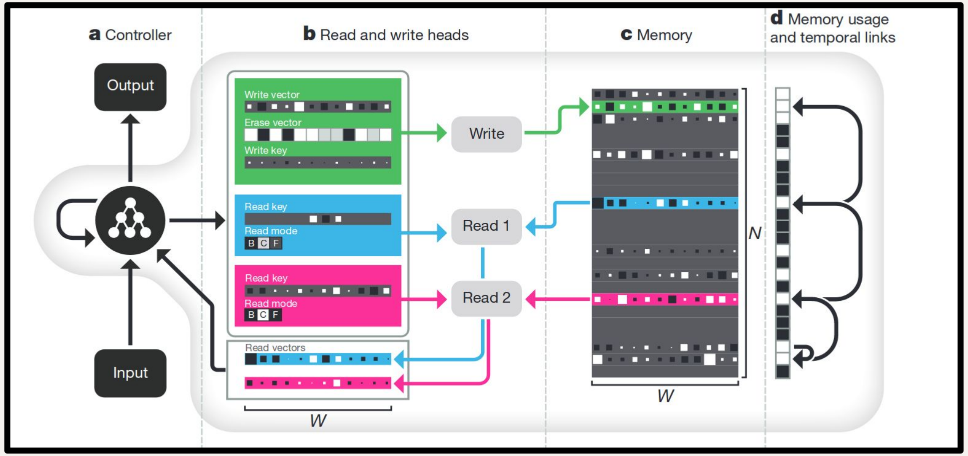
Implicit layers



Transformers blocks



Neural computers



A deep network is a **differentiable function** …

```
def my_network(x: tensor) -> tensor:
    ...
    ...
    ...
    return y
```

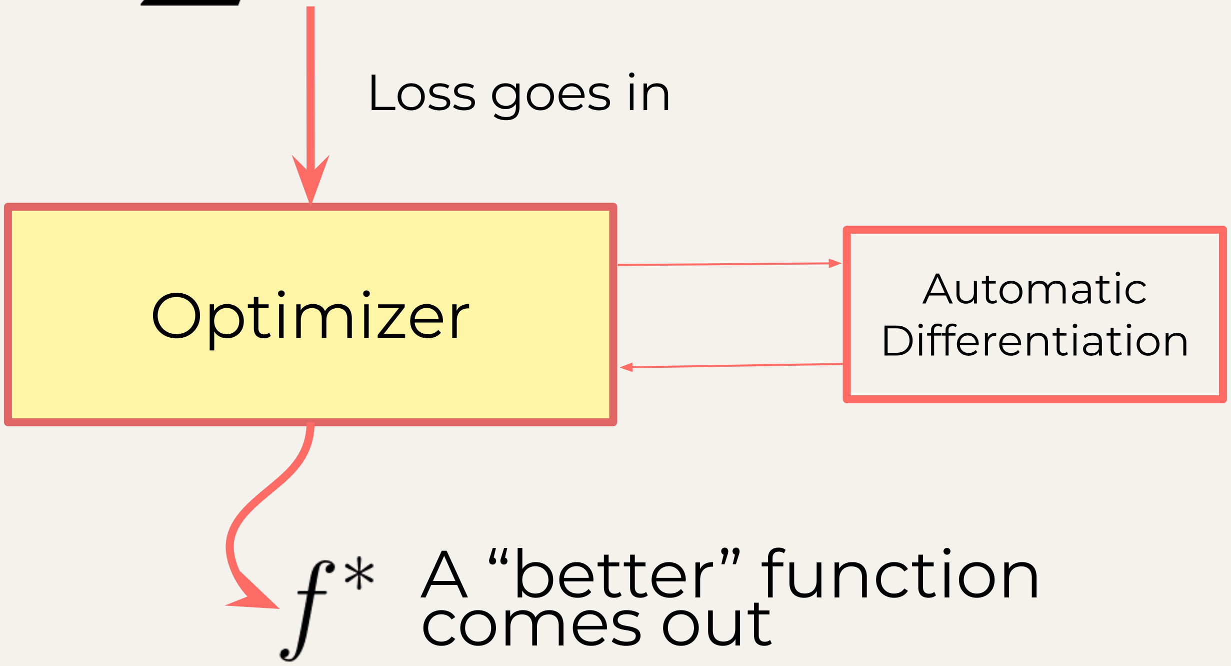
Input type

Output type

Sequence of **differentiable** primitives

… that can be **optimized** from data.

$$\min \sum l\left(y_i, f(x_i)\right)$$



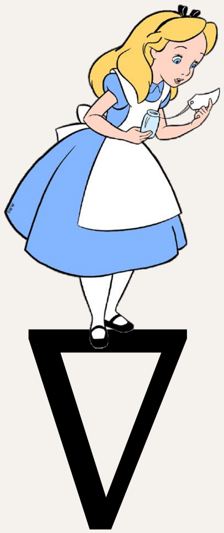
Corollary: deep networks are **composable**

```
def f(x):
    y = my_network(x)
    y = another_network(y)
    y = yet_another_network(y)
    return y
```

$\rightarrow$ **Automatic Differentiation** $\rightarrow$ $\nabla f(x)$

02

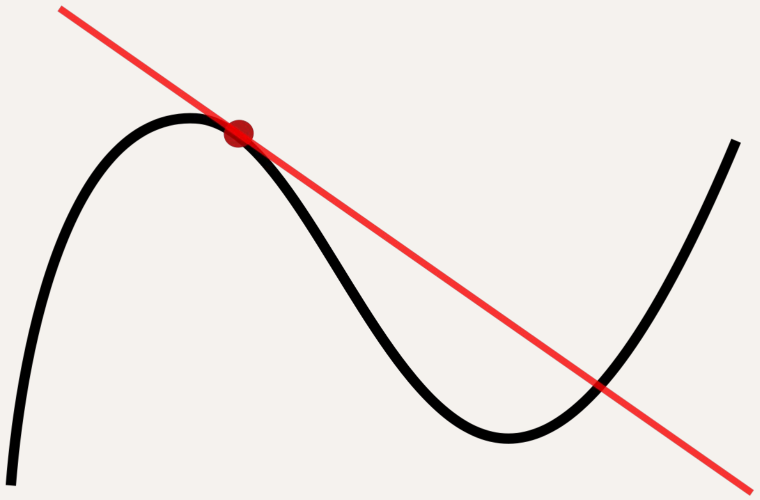
The how: automatic differentiation



“Would you tell me, please, which way I ought to go from here?”
“That depends a good deal on where you want to get to,” said the Cat.
“I don’t much care where” said Alice.
“Then it doesn’t matter which way you go,” said the Cat.
—Chapter 6, Pig and Pepper

Preliminaries: Derivative(s)

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



For a scalar function, the rate of change for an infinitesimally small displacement.

Preliminaries: Gradient(s)

For a function with an n-dimensional vector in input, we can use partial derivatives:

i-th basis vector

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

The vector of all partial derivatives is called the **gradient**: $\nabla f(\mathbf{x})$

Preliminaries: Jacobian(s)

Consider now a function with an *n*-dimensional input and a *m*-dimensional output, its **Jacobian** is defined as:

$$\mathrm{J}_f(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \cdots \\ \nabla f_m(\mathbf{x})^\top \end{bmatrix}$$

m rows (one for each output)

n columns (one for each input)

The Jacobian wrt what?

Let us go back to a classical fully-connected layer:

$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

The Jacobian wrt **x** is *(m,n)*, but the Jacobian w.r.t. **W** is rank-3 *(m,m,n)*. In general, full Jacobians in real layers can be quite cumbersome.

We will return to this point later on.

Chain rule of Jacobians

Like classical derivatives, Jacobians also have a **chain rule**:

$$f(\mathbf{x}) = g(h(\mathbf{x})) = (g \circ h)(\mathbf{x})$$

$$\underbrace{\mathrm{J}_f(\mathbf{x})}_{o \times n} = \underbrace{\mathrm{J}_g(h(\mathbf{x}))}_{o \times m} \underbrace{\mathrm{J}_h(\mathbf{x})}_{m \times n}$$

The gradient of function composition is the multiplication of the corresponding Jacobians.

Neural network primitives

Neural networks are composed of simple **differentiable** primitives:

$$f(\mathbf{x}, \mathbf{w})$$

Input

(Trainable) parameters

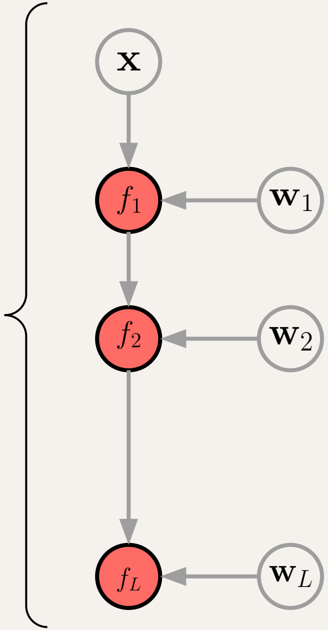
For each primitive, we know how to compute the input Jacobian and the weight Jacobian.

Neural networks

Simple NNs are a sequence of primitive operations:

$$
\begin{aligned}
\mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1) \\
\mathbf{h}_2 &= f_2(\mathbf{h}_1, \mathbf{w}_2) \\
\cdots &= \cdots \\
o &= f_L(\mathbf{h}_{L-1}, \mathbf{w}_L)
\end{aligned}
$$

Computational graph



In the more general case, we can have a DAG (not a sequence), and also parameter sharing between layers.

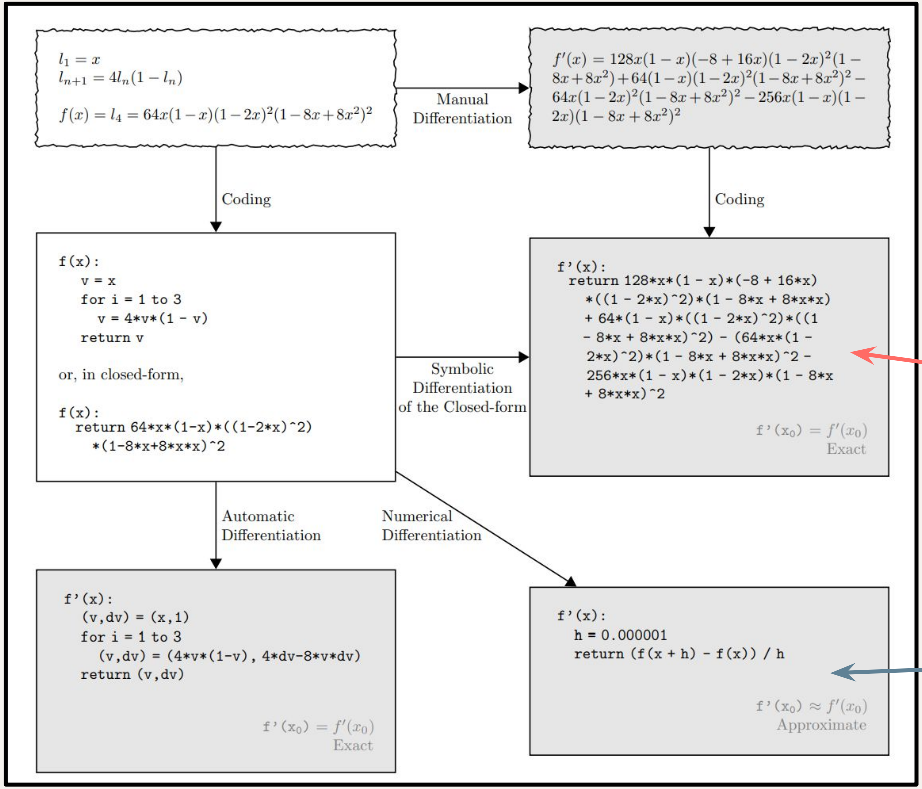
The goal of autodiff

Note that our last output, almost always, is **scalar** (e.g., sum of the per-element losses).

What we need is a way to *efficiently, simultaneously* compute all weight Jacobians (up to numerical precision):

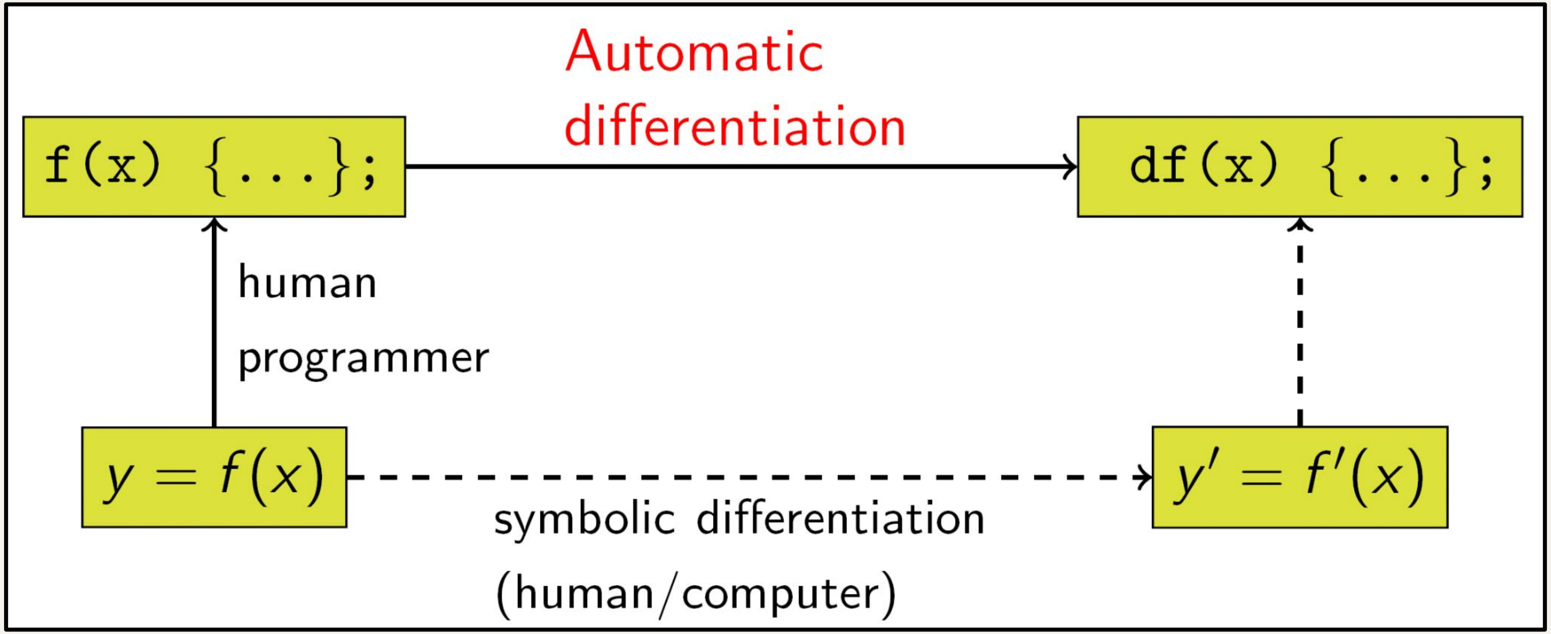
$$\left\{ \frac{\partial o}{\partial \mathbf{w}_i} \right\}_i$$

[1502.05767] Automatic differentiation in machine learning: a survey



Symbolic: build a symbolic formula for the gradients from the original *symbolic* program.

Numeric: numerically evaluate the derivatives using the definition.



Automatic differentiation - Wikipedia

One worked-out example

Consider a very simple example:

$$\begin{aligned}\mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1)\\ \mathbf{h}_2 &= f_2(\mathbf{h}_1, \mathbf{w}_2)\\ \mathbf{h}_3 &= f_3(\mathbf{h}_2, \mathbf{w}_3)\\ o &= \sum \mathbf{h_3} = \mathbf{1}^\top \mathbf{h}_3\end{aligned}$$

For example, this could be a one-layer neural network, cross-entropy loss, and final sum.

Considering each instruction in isolation, we have 4 input Jacobians and 3 weight Jacobians:

$$\{\underbrace{\mathbf{J_x}, \mathbf{J_{w_1}},}_{\text{Jacobians of } f_1} \underbrace{\mathbf{J_{h_1}}, \mathbf{J_{w_2}},}_{\text{Jacobians of } f_2} \underbrace{\mathbf{J_{h_2}}, \mathbf{J_{w_3}},}_{\text{Jacobians of } f_3} \mathbf{1}\}$$

Then, we can use the chain rule to “stitch” them together.

Performing multiplications left-to-right: **forward-mode autodiff**

$$
\begin{aligned}
\frac{\partial o}{\partial \mathbf{w}_1} &= \left(\mathbf{J}_{\mathbf{w}_1}\right)^\top \left(\mathbf{J}_{\mathbf{h}_1}\right)^\top \left(\mathbf{J}_{\mathbf{h}_2}\right)^\top \mathbf{1} \\
\frac{\partial o}{\partial \mathbf{w}_2} &= \left(\mathbf{J}_{\mathbf{w}_2}\right)^\top \left(\mathbf{J}_{\mathbf{h}_2}\right)^\top \mathbf{1} \\
\frac{\partial o}{\partial \mathbf{w}_3} &= \left(\mathbf{J}_{\mathbf{w}_3}\right)^\top \mathbf{1}
\end{aligned}
$$

Repeated!

Performing multiplications right-to-left: **reverse-mode autodiff**

Forward-mode autodiff

Forward-model autodiff can be implemented easily: all operations can be performed *in parallel* to the original program (i.e., we can devise a new program returning the original outputs and the gradients).

However, all operations will scale linearly w.r.t. number of parameters, which is impractical for today's neural networks. On the good side, it requires little memory because previous operations can be discarded.

Forward-mode autodiff

Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\mathbf{= 5.5}$

[1502.05767] Automatic differentiation in machine learning: a survey

Reverse-mode autodiff

Reverse-mode autodiff collects all intermediate operations of the program (**tracing**), and then “unrolls” all the gradient operations from right-to-left.

It is significantly more efficient (only vector-matrix operations above), but it requires to store all intermediate outputs, making it highly memory consuming.

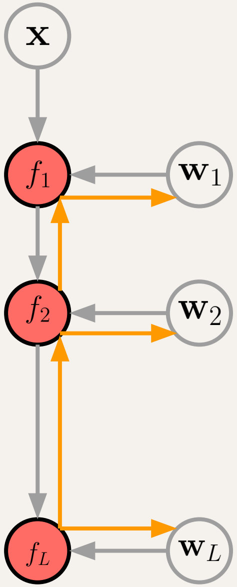
Autodiff in ML is almost always in reverse-mode (**backpropagation**).

Reverse-mode autodiff

Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Reverse Adjoint (Derivative) Trace			
$\boldsymbol{\bar{x}_1}$	$\boldsymbol{= \bar{v}_{-1}}$		$\mathbf{= 5.5}$
$\boldsymbol{\bar{x}_2}$	$\boldsymbol{= \bar{v}_0}$		$\mathbf{= 1.716}$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	



[1502.05767] Automatic differentiation in machine learning: a survey

Quick-note: autodiff or backprop?

A few commonly accepted milestones:

- **Wengert (1964)** is credited as the first description of forward-mode AD, which became popular in the 80’ mostly with the work of **Griewank**.
- **Linnainmaa (1976)** is considered the first description of modern reverse-mode AD, with the first major implementation in **Speelpenning (1980)**.
- **Werbos (1982)** is the first concrete application to NNs, before being popularized (as backpropagation) by **Rumelhart et al. (1986)**.

Who Invented Backpropagation? Who Invented the Reverse Mode of Differentiation?

Vector-Jacobian products

Importantly, we do not need to know how to compute the Jacobians of the primitives, but only their **vector-Jacobian products** (VJP):

$$\mathbf{v}^\top \mathbf{J}_{\mathbf{x}} \,, \quad \mathbf{u}^\top \mathbf{J}_{\mathbf{w}}$$

(To see this, transpose all the equations before!)

Forward-mode autodiff can equivalently be written with Jacobian-vector products (JVPs), by computing the final gradients one value at a time.

VJPs vs. Jacobians

The previous result is important, because sometimes VJPs are easier than the full Jacobians:

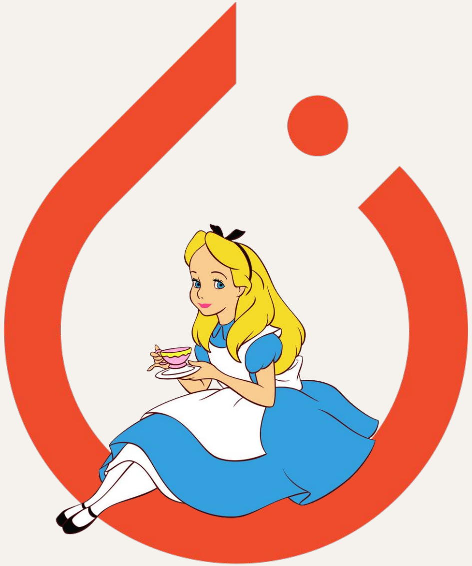
$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

The input Jacobian is a rank-3 tensor, while:

$$\begin{aligned} \mathbf{v}^\top \mathrm{J}_{\mathbf{x}} &= \mathbf{W}^\top \mathbf{v} \\ \mathbf{u}^\top \mathrm{J}_{\mathbf{w}} &= \mathbf{x}\mathbf{u}^\top \end{aligned}$$

03

The nitty-gritty details



"The best way to explain it is to do it."

—Chapter 3, A Caucus-Race and a Long Tale

Deep learning frameworks

Ecosystem
(hubs, libraries, extensions, ...)

Layer 2
High-level constructs (layers, optimizers, losses, metrics, ...)

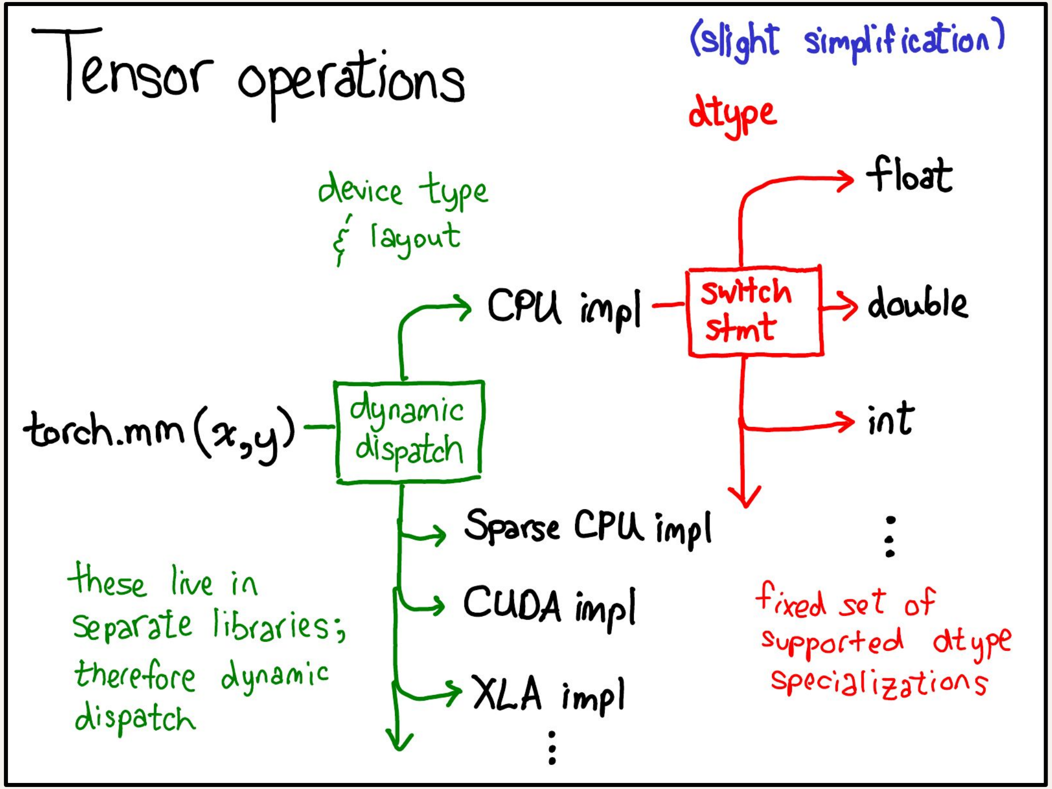
Layer 1
Autodiff module

Layer 0
Tensor primitives + their VJPs (matrix-multiplication, etc.).

One primitive may be implemented with multiple kernels depending on the supported hardware (CPU, GPU, TPU, IPU, ...).

Dispatchers

PyTorch internals : ezyang’s blog



Some history and terminology

The revival of AD in neural networks started with Theano (2008), to which followed a Cambrian explosion of frameworks (TensorFlow 1.0, PyTorch, Caffe, JAX, ...).

Theano and TF 1.0 focused heavily on performance. The user implemented the computational graph with a small domain specific language (DSL), and execution was decoupled from the definition (**define-then-run**).

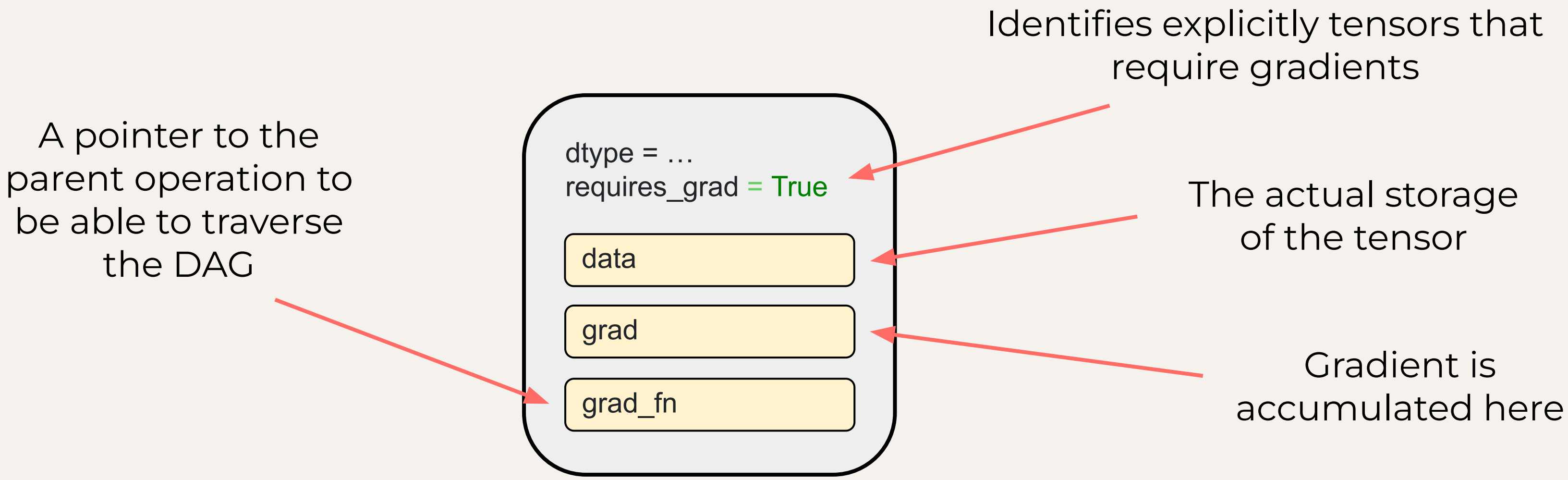
Most frameworks today are more dynamic (**define-by-run**), leaving compilation as a separate, optional step (JIT in JAX / PyTorch, tf.function in TF).

Flavours of implementations

While most frameworks implement similar things, the *way* they implement them can make some use cases considerably faster or easier.

1. Having an external context manager to store operations (e.g., the GradientTape of TF, technically a Wengert list) vs. building the DAG dynamically.
2. Being able to easily differentiate w.r.t. any sort of object (e.g., the PyTrees of JAX)
3. The flexibility of the autodiff framework (e.g., to compute full Hessians).
4. Having only a functional interface (e.g., JAX).
5. Supporting sparse and/or complex-valued data.

Anatomy of a PyTorch tensor



Note: PyTorch also has a functional variant, very similar to the JAX implementation.

High-level APIs

Most frameworks (TensorFlow, PyTorch) implement an object-oriented API:

```
class MyModule(nn.Module):
    def __init__(self):
        self.params = Parameter( torch.tensor ( … ) )

    def forward(self, x):
        return self.params @ x
```

Simple polymorphism allows for module compositionality

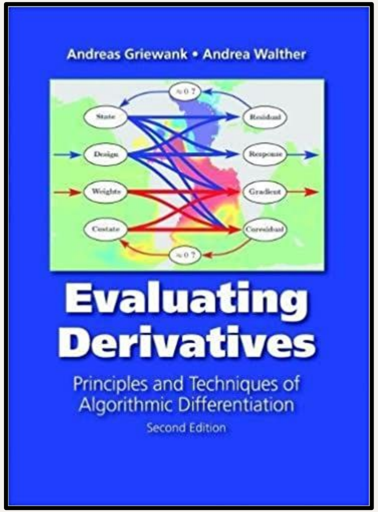
Parameters are properties of the object

Forward logic is a method

Materials to learn more

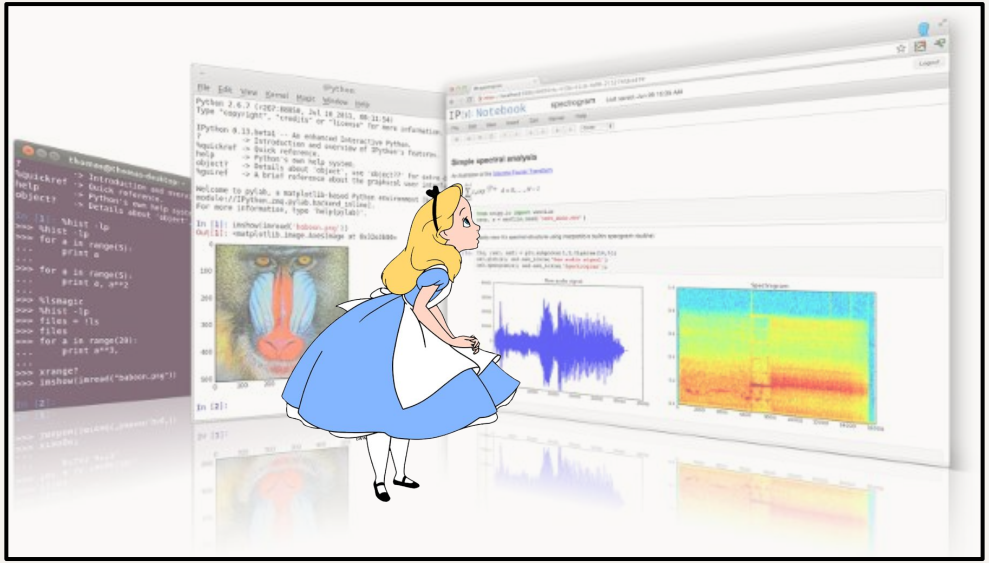
PyTorch internals : ezyang's blog (slightly outdated)
Automatic differentiation (Matthieu Blondel, 2020)
GitHub - mattjj/autodidact: A pedagogical implementation of Autograd
GitHub - karpathy/micrograd: A tiny scalar-valued autograd engine and a neural net library on top of it with PyTorch-like API
GitHub - geohot/tinygrad: You like pytorch? You like micrograd? You love tinygrad! ❤️
GitHub - MINI-PYTORCH/MINI-TORCH: Mini-pytorch implemented from scratch using Python
The spelled-out intro to neural networks and backpropagation: building micrograd

Our discussion will be highly simplified, leaving out many important topics (e.g., tracing and distributed implementations).



Notebook time!

Notebook 1: simple experiments with PyTorch and JAX, side-by-side.
Notebook 2: building a toy autodiff framework, PyTorch-style.
https://colab.research.google.com/drive/1AbNRRjL0DMoj4VPnul7QIYJC5nLU3Fn2?usp=sharing



Limits of OOP

By default, JAX takes a fully functional paradigm: *everything* (layers, losses) is a function.

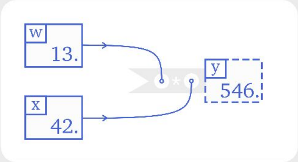
Sometimes, this makes it easier to explicitly manipulate parameters and to compose different transformations.

Most high-level frameworks in JAX define a layer by a pair of init/apply functions (with some exceptions, see Equinox).

PyTorch

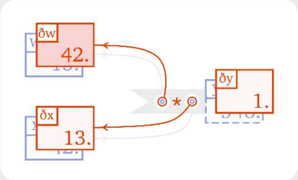
Executing code produces graph/tape

```
w = torch.tensor(13.)
x = torch.tensor(42.)
y = w * x
```



Backprop/reverse-mode autodiff by following the graph/tape

```
y.backward()
```

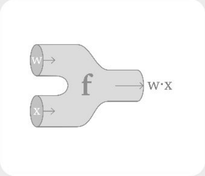



```
grad_w = w.grad
```

JAX

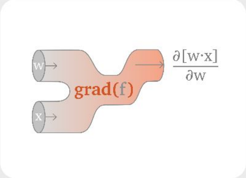
Define pure function

```
def f(w, x):
  return w * x
```



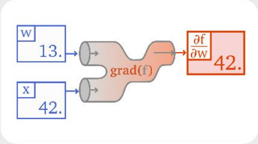
JAX creates gradient function

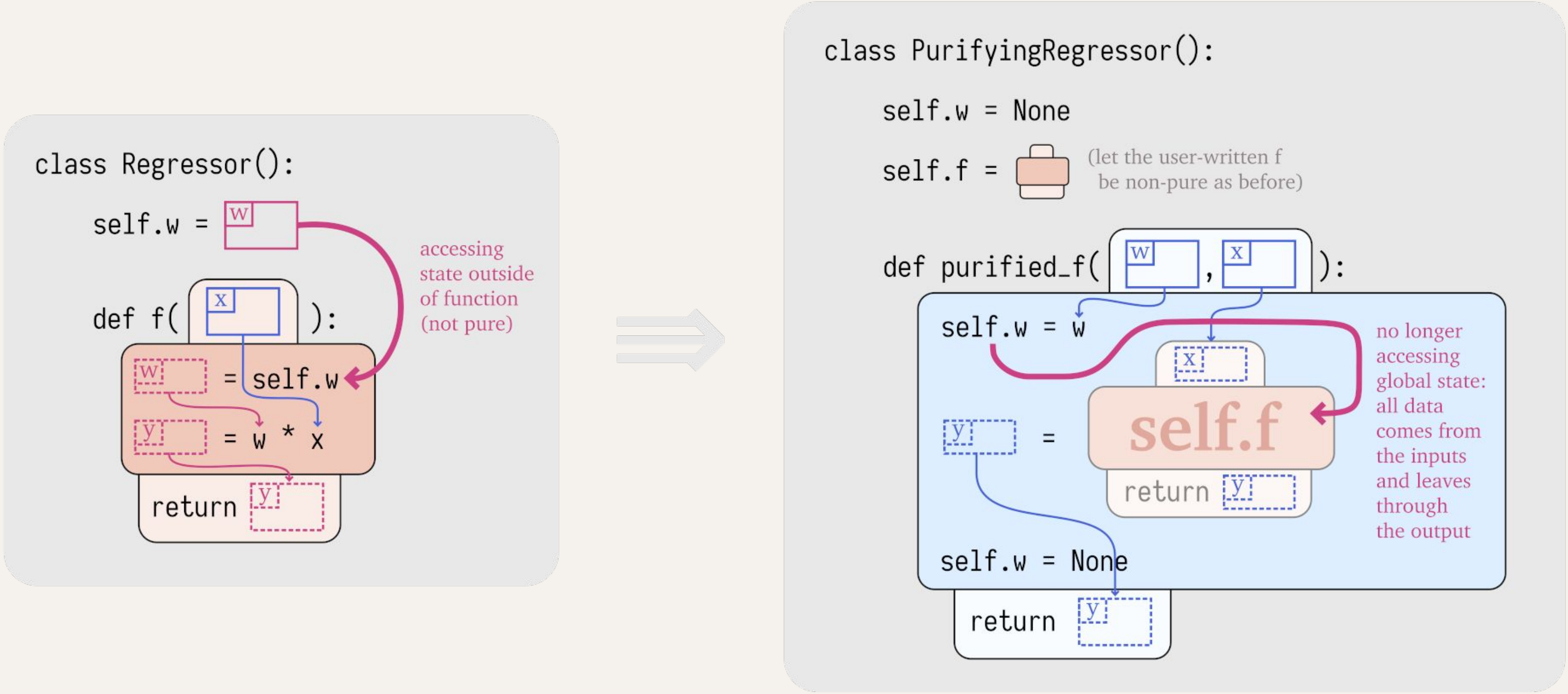
```
df_dw = jax.grad(f)
# ⇒ df_dw(w, x) == x
```



Evaluate that to get gradients

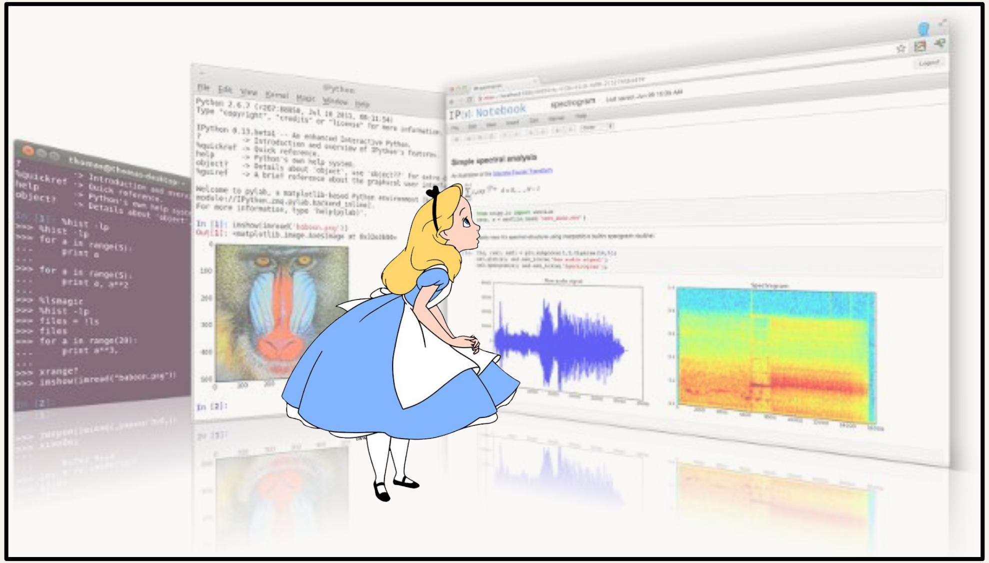
```
w = jnp.array(13.)
x = jnp.array(42.)
grad_w = df_dw(w, x)
```




class Regressor():
self.w =
w
def f(
x
):
w
= self.w
y
= w * x
return
y
accessing state outside of function (not pure)
class PurifyingRegressor():
self.w = None
self.f =
(let the user-written f be non-pure as before)
def purified_f(
w
,
x
):
self.w = w
x
y
=
self.f
return
y
self.w = None
return
y
no longer accessing global state: all data comes from the inputs and leaves through the output

Notebook time!

Notebook 3: moving from PyTorch to JAX and .vice versa
https://colab.research.google.com/drive/1AbNRRjL0DMoj4VPnul7QIYJC5nLU3Fn2?usp=sharing



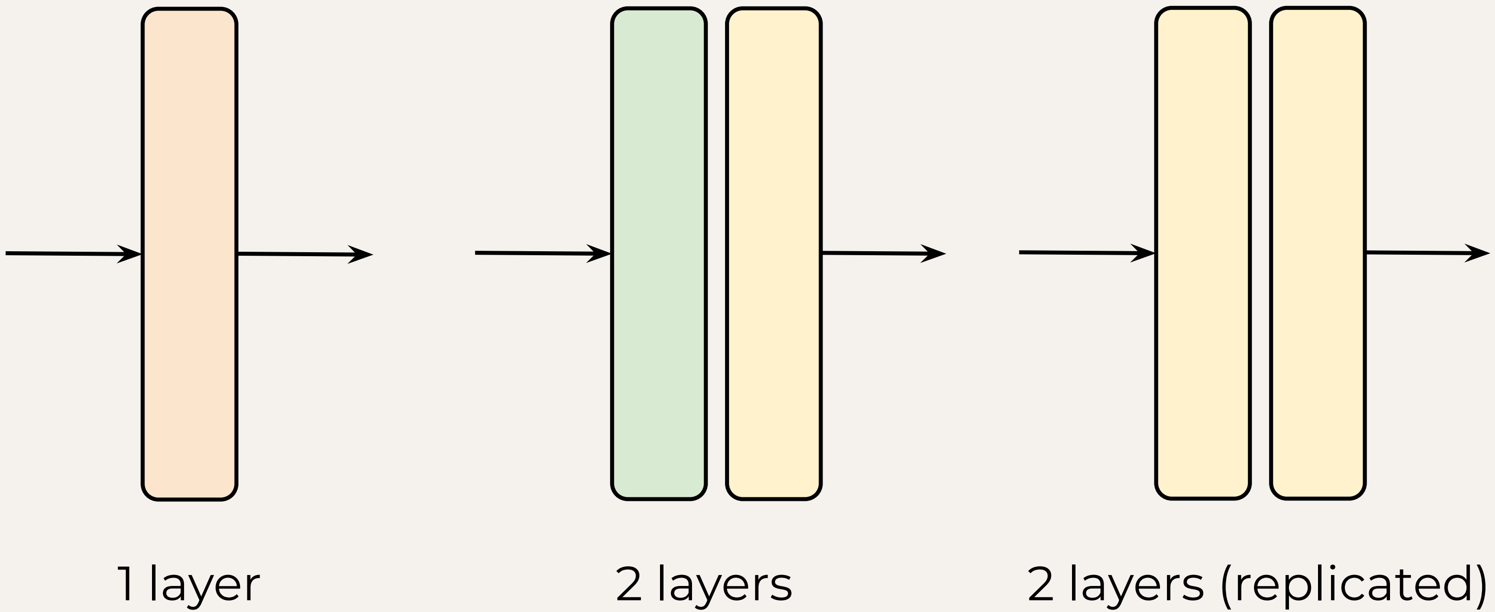
04

Advanced topics

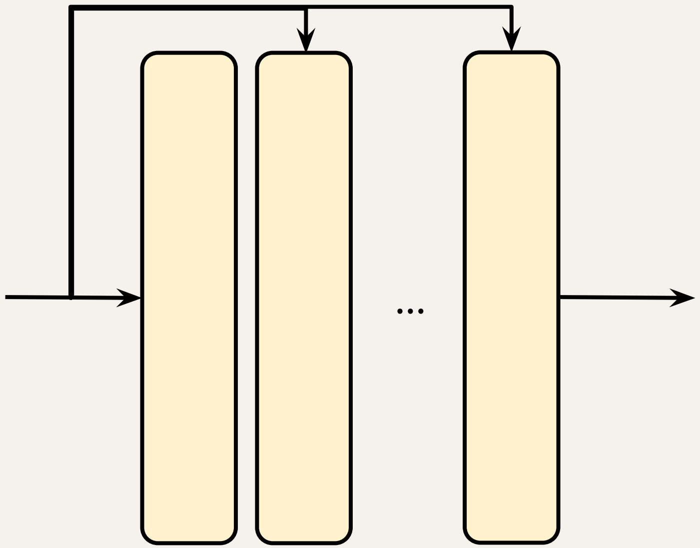
““Curiouser and curiouser!” cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English).

—Chapter 2, The Pool of Tears

Layer sharing



Deep equilibrium layers



What happens if we replicate the layer *infinite* times?

The output of the layer is now *implicitly* defined by a fixed-point equation:

$$f(\mathbf{x}, \mathbf{z}) = \mathbf{z}$$

Input

Output (can be initialized to zero)

Deep Equilibrium Models

Solving fixed-point equations

Writing a fixed-point layer is easy (of course, there are faster alternatives):

```
class FixedPointLayer(nn.Module):
    def __init__(self):
        self.w = ...

    def forward(self, x):
        z = torch.zeros_like(x)
        while self.check_convergence():
            z = f(self.w, z, x)
        return z
```

By default, however, AD requires to store all intermediate steps of the while loop and backpropagate through them, which is expensive.

Implicit function theorem

By the **implicit function theorem**, there exists a continuous function z*
such that:

$$f(\mathbf{x}, z(\mathbf{x})) = z(\mathbf{x})$$

Differentiating everything:

$$\frac{\partial z(\mathbf{x})}{\partial \mathbf{x}} = \left(\mathbf{I} - \frac{\partial f(\mathbf{x}, \mathbf{z})}{\mathbf{z}}\right)^{-1} \frac{\partial f(\mathbf{x}, \mathbf{z})}{\partial \mathbf{x}}$$

We can differentiate the layer by computing two gradients at the optimum **z** (no need to store any intermediate layers).

VJPs of implicit functions

In order to implement the layer as a primitive in a framework, we need its VJP with some vector **u**. Fascinatingly, this can be expressed as another fixed-point equation:

$$\mathbf{g} = \left(\frac{\partial f(\mathbf{x}, \mathbf{z})}{\partial \mathbf{z}} \right)^{\top} \mathbf{g} + \mathbf{u}$$

Custom derivative rules for JAX-transformable Python functions

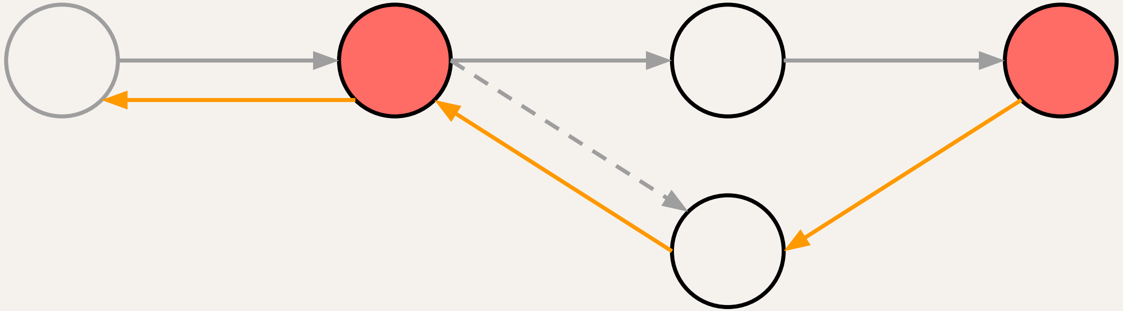
Read more

Implicit differentiation is the key for several topics:

- Defining layers in terms of convex optimization problems;
- Relaxing combinatorial problems inside layers;
- Neural ordinary differential equations (Neural ODEs);
- And so on...

Check out this tutorial for more: Deep Implicit Layers

Gradient checkpointing



To save memory, gradient checkpointing is now popular. Outputs of red nodes are stored (checkpoints). When back-propagating through a non-checkpointed node, its output is recomputed starting from the previous checkpoint in memory.

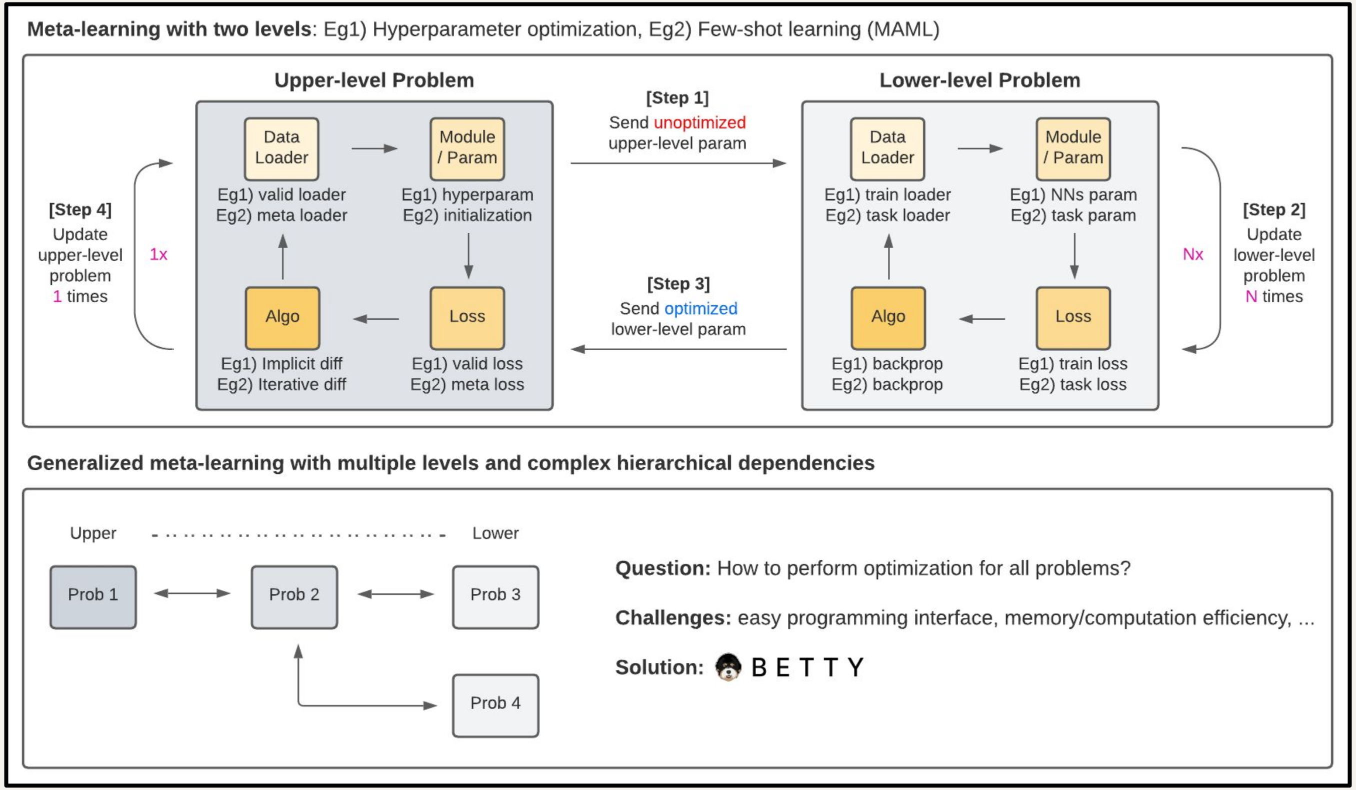
Bilevel optimization

$$\underbrace{\arg\min_{\alpha} \; F(\alpha, \beta^*)}_{\text{Outer problem}} \;\; \text{s.t.} \;\; \underbrace{\beta^* = \arg\min_{\beta} \; G(\alpha, \beta)}_{\text{Inner problem}}$$

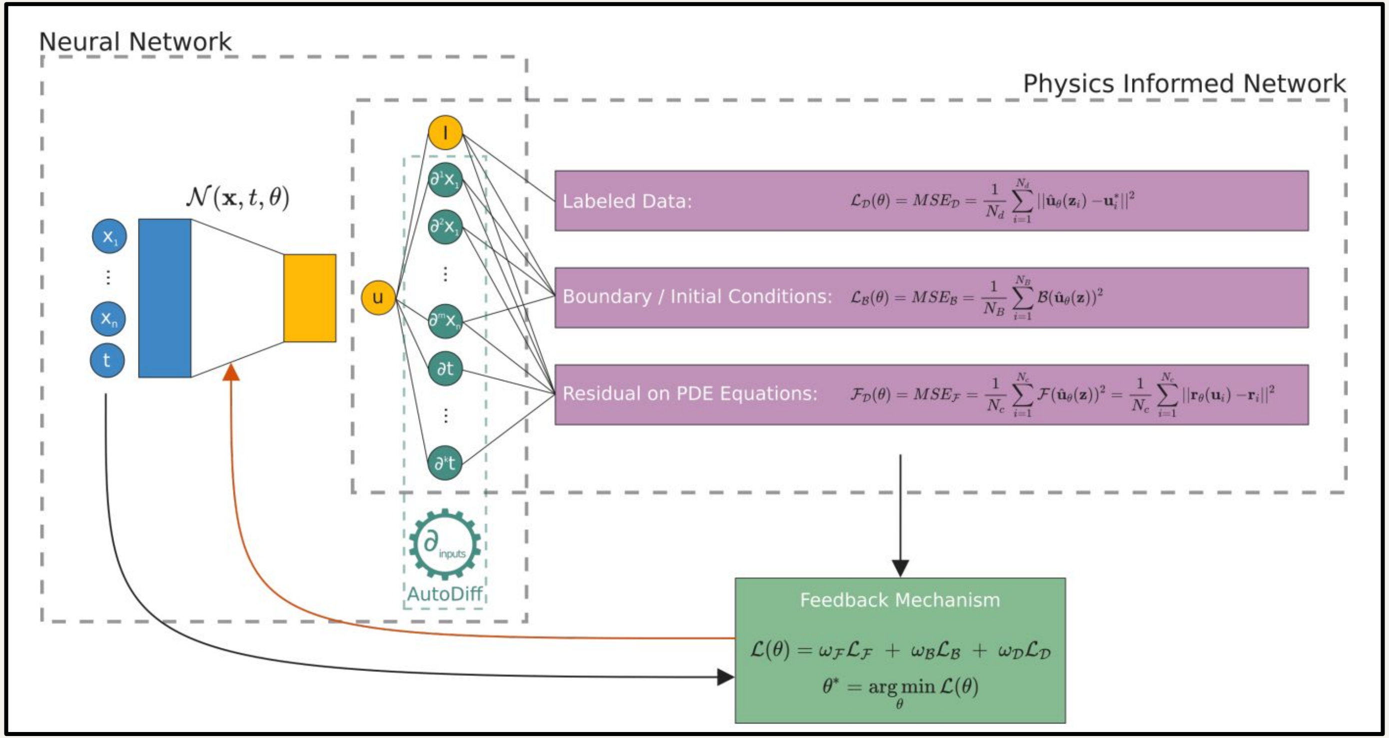
Examples:

1. Hyper-parameter optimization (outer loop is the validation accuracy;
2. Few-shot learning (inner loop is the training step on the few-shot dataset).

[GitHub - leopard-ai/betty: Betty: an automatic differentiation library for generalized meta-learning and multilevel optimization]

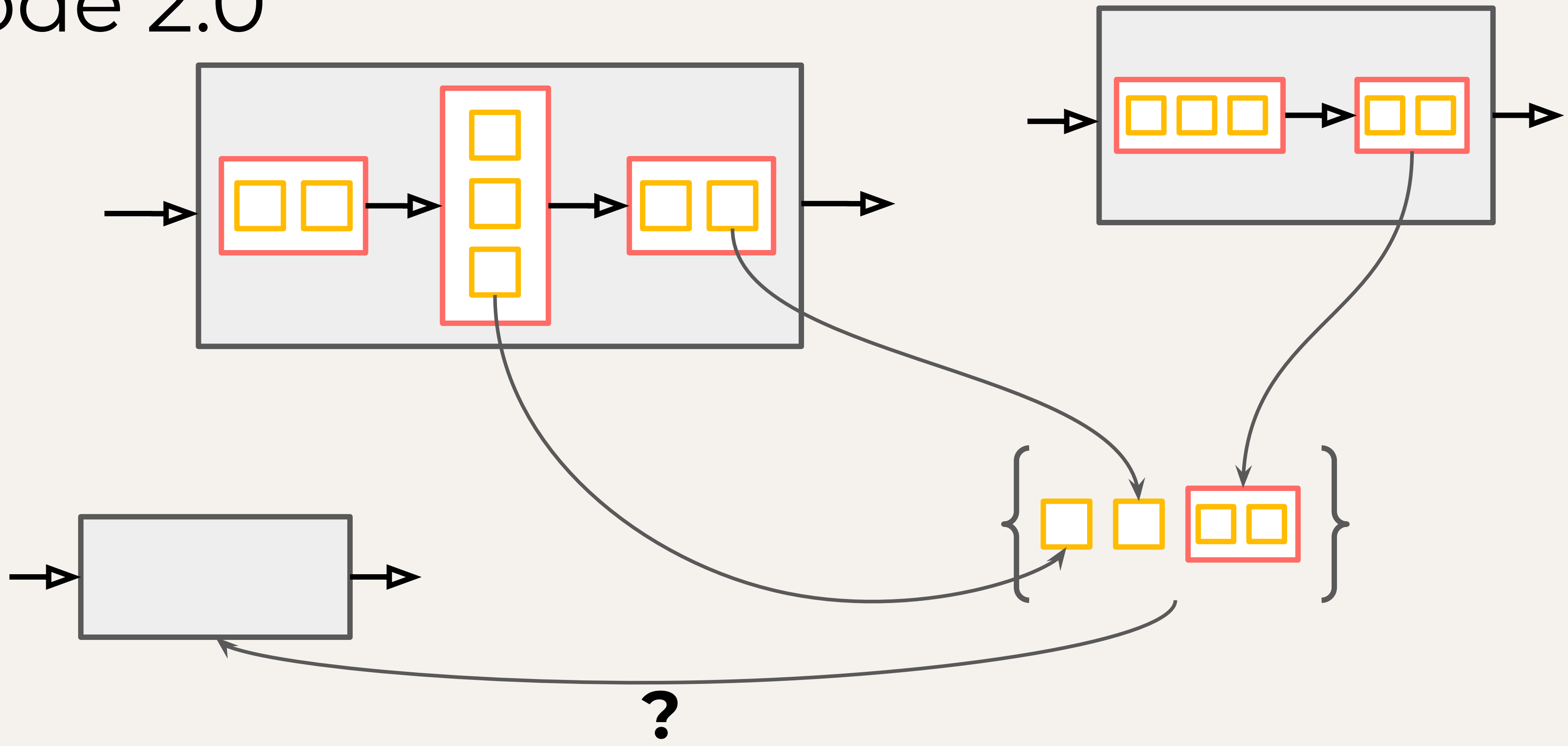


Physics-informed NNs



[2201.05624] Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next

Code 2.0



"Tut, tut, child!" said the Duchess. "Everything's got a moral, if only you can find it."
—Chapter 9, The Mock Turtle's Story

https://www.sscardapane.it/

https://twitter.com/s_scardapane

Thanks for listening!