UNIVERSITÀ DEGLI STUDI FIRENZE

DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB
DISTRIBUTED DATA INTELLIGENCE AND TECHNOLOGIES LAB

https://www.snap4City.org

SNAP4CITY

KM4CITY

NeuroSymbolic Artificial Intelligence at Scale

Marco Fanfani, marco.fanfani@unifi.it

https://www.disit.org/

Parte: 2.3 (2025-26) Physics-Informed Neural Networks
Automatic Differentiation

Physics-Informed Neural Networks (PINNs)

Solving Partial Differential Equations through Scientific Machine Learning

AUTOMATIC DIFFERENTIATION

Neural Networks as Function Approximators

The Core Engine of PINNs:

- Neural networks (NNs) are designed to recognize patterns and process vast amounts of data.
- Universal Approximation Theorem: A deep neural network can represent **any complex non-linear mapping**.
- PINN Context: The network is specifically trained to approximate the hidden state $\mathbf{u}(t, \mathbf{x})$ of a physical system.

Structure of a Multilayer Perceptron (MLP)

- **Architecture:** Consists of an input layer, an output layer, and multiple hidden layers.
- **Dense Connectivity:** In a fully connected architecture, every neuron in one layer links to every neuron in the next.
- **Recursive Learning:** Allows the network to capture increasingly complex relationships within the data.

Mathematical Building Blocks: Weights and Biases

- Defining the Parameters (θ):
  - **Weight Matrix $\mathbf{W}_h$** indicates a neuron's sensitivity to each input term from the previous layer.
  - **Bias Vector $\mathbf{b}_h$ r**epresents a measure of a neuron's potential activity or threshold.

Linear Combination: $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$

Information Flow: Layer-by-Layer Computation

- **Input Layer** G_1 receives spatial and temporal coordinates (x, t).
- **Hidden Layers** G_h applies weights/biases followed by a non-linear transformation:

$$G_h(\mathbf{x}) = \sigma(\mathbf{W}_h\, G_{h-1}(\mathbf{x}) + \mathbf{b}_h)$$

- **Output Layer** G_H produces final physical variables, i.e. (v_x, v_y, v_z, p).

The Role of Non-Linear Activation Functions (σ)

- Enabling Complexity:
  - Without σ, a multi-layer network collapses into a simple linear model.
  - **Sigmoid** & **Tanh**: Provide smooth, differentiable transitions.
  - **ReLU**: Popular but problematic for PINNs due to zero-derivatives.
  - **Adaptive Functions** (**GELU**, **Swish**): Accelerate convergence in SciML tasks.

Introducing Automatic Differentiation (AD)

What is AD?

- Techniques to **efficiently evaluate derivatives of numeric functions** expressed as computer programs.
- It is **neither numerical** (finite differences) **nor purely symbolic** differentiation.
- Key Concept: AD redefines code operators to propagate derivatives according to the **Chain Rule**.

AD vs. Numerical Differentiation

- The Limitations of Finite Differences:
  - **Approximation Errors:** Suffers from truncation (large h) and round-off (small h) errors.
  - **Computational Cost:** Requires n+1 evaluations for n inputs; prohibitive for millions of parameters.
- **AD Advantage: Exact derivatives at machine precision without numerical instability.**

AD vs. Symbolic Differentiation

- **The 'Expression Expansion' Problem:**
  - Symbolic methods lead to exponentially large formulas for nested functions, making evaluation extremely slow.
  - AD Solution: Stores **intermediate numerical results** (the 'Wengert list') instead of full formulas, maintaining the same complexity as the original function.

How AD Works: Divide and Differentiate

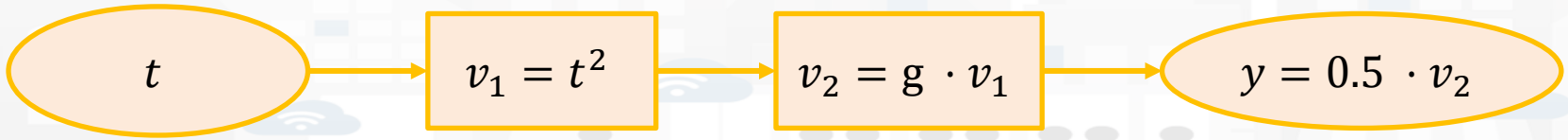
- **Atomization:** Complex expressions are broken into elementary operations (sin, exp, +, *).
- **Computational Graph:** Operations are organized into a Directed Acyclic Graph (DAG).
- **Local Differentiation:** Derivatives for each atom are well-known and easy to compute.
- **Chain Rule:** Local derivatives are combined to compute the total derivative.

How AD Works: Divide and Differentiate

Toy example: suppose wanting to compute the derivative of

$$y = 0.5gt^2$$

1. **Atomization**: the equation is reduced to elementary operations:
   - $v_1 = t^2$
   - $v_2 = g \cdot v_1$
   - $y = 0.5 \cdot v_2$



How AD Works: Divide and Differentiate

Toy example: suppose wanting to compute the derivative of

$$y = 0.5gt^2$$

2. **Forward pass**: inputs flow though the operations to compute the output:
   - $t = 2 \Rightarrow v_1 = 2^2 = 4$
   - $g = 9.81 \Rightarrow v_2 = 9{,}81 \cdot 4 = 39.24$
   - $y = 0.5 \cdot 39.24 = 19.62$

How AD Works: Divide and Differentiate

Toy example: suppose wanting to compute the derivative of

$$y = 0.5gt^2$$

3. Local Differentiation & Chain Rule (Backward Pass):

- To obtain the derivative, the system runs the graph in reverse (Reverse Mode).
- Each node calculates its local derivative and multiplies it by the value it receives from the next node (Chain Rule)

How AD Works: Divide and Differentiate

Toy example: suppose wanting to compute the derivative of

$$y = 0.5gt^2 \Rightarrow \frac{\partial y}{\partial t} = 0.5 \cdot g \cdot 2 \cdot t = gt$$

Node	Local Operation	Local Derivative	Adjoint Calculation (Chain Rule)	Accumulated Value
y	Output finale	1	1	**1.0** (Partenza)
v_2	$0.5 \cdot v_2$	0.5	$1.0 \cdot 0.5$	**0.5**
v_1	$g \cdot v_1$	$g\ (= 9.81)$	$0.5 \cdot 9.81$	**4.905**
t	t^2	$2t\ (= 2 \cdot 2 = 4)$	$4.905 \cdot 4$	**19.62**

Adjoint Variables and Sensitivity

Adjoint:

$$\bar{v}_i = \frac{\partial L}{\partial v_i}$$

- The adjoint represents the sensitivity of the final Loss (L) to changes in the intermediate value v_i
- By calculating adjoints, we *pull back* the error from the output layer to the input parameters.

The Backward Pass: Reverse Topological Order

- **Initialize:** Start at the output node and set its adjoint to 1, i.e.

$$\frac{\partial L}{\partial L} = 1$$

- **Traverse:** Move backward through the graph in reverse topological order.
- **Accumulate:** Sum contributions from all successors using the chain rule:

$$\bar{v}_i = \sum_{j \in successor(i)} v_j \left(\frac{\partial v_j}{\partial v_i}\right)$$

How AD Works: Divide and Differentiate

```python
import torch

# 1. We define the input and enable gradient tracing
# t represents time (2.0s), g the gravity constant
t = torch.tensor(2.0, requires_grad=True)
g = torch.tensor(9.81)

# 2. Forward Pass: We define the physical function (y = 1/2 * g * t^2)
# PyTorch builds the computational graph "under the hood"
y = 0.5 * g * t**2

# 3. Backward Pass: Automatic Gradient Calculation (Reverse Mode)
# Trigger the Chain Rule from output (y) to input (t)
y.backward()

# 4. Risultato: La derivata dy/dt è ora memorizzata in t.grad
print(f"Position (y): {y.item()} m")     # Output: 19.62
print(f"Velocity (dy/dt): {t.grad} m/s") # Output: 19.62
```

How AD Works: Divid ntiate

```python
import torch

# 1. We define the input and enable gr
# t represents time (2.0s), g the gravity  ant

t = torch.tensor(2.0, requires_grad=True)
g = torch.tensor(9.81)

# 2. Forward Pass: We define the physical function (y = 1/2 * g * t^2)
# PyTorch builds the computational graph "under the hood"
y = 0.5 * g * t**2

# 3. Backward Pass: Automatic Gradient Calculation (Reverse Mode)
# Trigger the Chain Rule from output (y) to input (t)
y.backward()

# 4. Risultato: La derivata dy/dt è ora memorizzata in t.grad
print(f"Position (y): {y.item()} m")     # Output: 19.62
print(f"Velocity (dy/dt): {t.grad} m/s") # Output: 19.62
```

It instructs PyTorch to monitor every elementary operation (atom) applied to that tensor to build the dynamic graph.

How AD Works: Divide and Differentiate

How does PyTorch to know how to compute derivatives of atomic operations?

- PyTorch has a *Table of Derivatives* (`Derivatives.yaml`) in which derivatives have been manually written for each atomic operation (sum, multiplication, sine, exponential, etc.)
- Every time you download PyTorch, you take a huge database of ready-made analytical formulas with you
- PyTorch, each atomic operation is defined as a pair of functions:
  - `forward(ctx, input)`: Calculates the result (e.g. t^2) and saves the necessary inputs in context (`ctx`), i.e. the primals
  - `backward(ctx, grad_output)`: Retrieves the saved inputs and applies the pre-programmed derivative rule

How AD Works: Divide and Differentiate

```python
import torch

# 1. We define the input and enable gradient tracing
# t represents time (2.0s), g the gravity constant
t = torch.tensor(2.0, requires_grad=True)
g = torch.tensor(9.81)

# 2. Forwar                    sical function (y = 1/2 * g * t^2)
# PyTorch b                    graph "under the hood"
y = 0.5 * g

# 3. Backwa                    ent Calculation (Reverse Mode)
# Trigger the                  Rule from output (y) to input (t)
y.backward()

# 4. Risultato: La derivata dy/dt è ora memorizzata in t.grad
print(f"Position (y): {y.item()} m")     # Output: 19.62
print(f"Velocity (dy/dt): {t.grad} m/s") # Output: 19.62
```

It runs the entire Reverse Mode AD algorithm. In a single step, it calculates the sensitivity of the output with respect to all input variables.

How AD Works: Divide and Differentiate

- In real PINNs, instead of a simple formula you will have a Neural Network with millions of parameters.
- The `backward()` function will handle the entire chain of derivations without human error.

Higher order derivatives

- In a PINN, we almost never stop at the first derivative
- E.g.: if we consider Newton's law $F = m \cdot a$, we need acceleration, i.e. the second derivative of the position: $a = \frac{d^2y}{dt^2}$.
- How does Automatic Differentiation handle them?

Higher order derivatives

- AD is recursive!
- Since the result of a derivative operation (the gradient) is itself a function represented by a computational graph, we can reapply the AD on that graph.
  - First Derivative: Generates a graph that calculates $\frac{dy}{dt}$.
  - Second Derivative: "Differentiate the differentiator", calculating the gradient of the gradient.

Higher order derivatives

```python
# 1. We define the input and enable gradient tracing
# t represents time (2.0s), g the gravity constant
t = torch.tensor(2.0, requires_grad=True)
g = torch.tensor(9.81)


# 2. Forward Pass: We define the physical function (y = 1/2 * g * t^2)
# PyTorch builds the computational graph "under the hood"
y = 0.5 * g * t**2


print(y)


# Calculation of the first derivative (Speed)
# create_graph=True allows you to derive this result further
v = torch.autograd.grad(y, t, create_graph=True)[0]


print(v)


# Calculation of the second derivative (Acceleration)
a = torch.autograd.grad(v, t, create_graph=True)[0]


print(a) # Acceleration: a = 9.81 == g
```

Efficiency: The Power of the Scalar Output

- **One-Pass Gradient**: Reverse mode computes the gradient w.r.t. ALL parameters in one backward pass.
- **Computational Cost**: Time required is only a small constant factor of the forward pass.
- **Scalability**: This efficiency makes training models with billions of parameters feasible.

The Space-Time Tradeoff

- **Space Cost:** Requires storing all intermediate numerical values computed during the forward pass.
- **The Tape:** Data structure (usually list or graph) used to store order and values of computed operations for reuse during backward pass.
- **Optimization:** Libraries use *'checkpointing*' to balance speed with memory limits. During the backward pass, if the system needs a value that it has not saved, it recalculates it on the fly starting from the last available checkpoint.