Topics:
• Backpropagation / Automatic Differentiation
• Jacobians

CS 4803-DL / 7643-A
ZSOLT KIRA
To develop a general algorithm for this, we will view the function as a **computation graph**

Graph can be any **directed acyclic graph** (DAG)

- Modules must be differentiable to support gradient computations for gradient descent

A **training algorithm** will then process this graph, **one module at a time**

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*
Directed Acyclic Graphs (DAGs)

• Exactly what the name suggests
  – Directed edges
  – No (directed) cycles
  – Underlying undirected cycles okay

![Directed Acyclic Graphs (DAGs) Diagram]
Directed Acyclic Graphs (DAGs)

• Concept
  – Topological Ordering
Directed Acyclic Graphs (DAGs)
\[ f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2) \]
\(- \log \left( \frac{1}{1 + e^{-w \cdot x}} \right) \)
Backpropagation
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Layer 1 → **Layer 2** → **Layer 3**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Layer 1 → Layer 2 → Layer 3

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Note that we must store the **intermediate outputs of all layers**!

This is because we will need them to **compute the gradients** (the gradient equations will have terms with the output values in them)

---

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

**Neural Network Training**
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1 → Layer 2 → (Green Arrow)

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**
Step 1: Compute Loss on Mini-Batch: **Forward Pass**
Step 2: Compute Gradients wrt parameters: **Backward Pass**
Step 3: Use *gradient* to update **all parameters** at the end

Backpropagation is the application of gradient descent to a computation graph via the chain rule!

$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
We want to compute: \[ \frac{\partial L}{\partial h^\ell}, \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \]

We will use the chain rule to do this:

Chain Rule: \[ \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \]
We will use the chain rule to compute: \( \{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \} \)

- **Gradient of loss w.r.t. inputs:** \( \frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial h^{\ell-1}} \)
- **Gradient of loss w.r.t. weights:** \( \frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial W} \)

Given by upstream module (upstream gradient)
Calculated Analytically

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Backpropagation: a simple example

\[
f(x, y, z) = (x + y)z
\]

e.g. \(x = -2, \ y = 5, \ z = -4\)

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \ \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q
\]

Want:

\[
\frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z}
\]

Chain rule:

\[
\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}
\]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Gradients add at branches
Duality in Fprop and Bprop

\[ \text{FPROP} \quad \begin{array}{c}
\text{SUM} \\
\begin{array}{c}
+ \\
\end{array}
\end{array} \quad \text{BPROP} \]

\[ \text{COPY} \quad \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array}
\end{array} \quad \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\end{array}
\end{array}
\end{array}
\end{array} \]

(C) Dhruv Batra
Caffe Sigmoid Layer

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\[ (1 - \sigma(x)) \sigma(x) \] * top_diff (chain rule)
Linear Algebra View: Vector and Matrix Sizes
Closer Look at a Linear Classifier

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ w_{31} & w_{32} & \cdots & w_{3m} & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{bmatrix}$$

**Sizes:** $[c \times (d + 1)]$  $[(d + 1) \times 1]$

Where $c$ is number of classes

$d$ is dimensionality of input
Conventions:
- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar \( s \in \mathbb{R}^1 \), vector \( \mathbf{v} \in \mathbb{R}^m \), i.e. \( \mathbf{v} = [v_1, v_2, \ldots, v_m]^T \)
  and matrix \( \mathbf{M} \in \mathbb{R}^{k \times \ell} \)

<table>
<thead>
<tr>
<th></th>
<th>( S ) [ ]</th>
<th>( V ) [ ]</th>
<th>( M ) [ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( \frac{\partial s_1}{\partial s_2} ) [ ]</td>
<td>( \frac{\partial s}{\partial \mathbf{v}} ) [ ]</td>
<td>( \frac{\partial s}{\partial \mathbf{M}} ) [ ]</td>
</tr>
<tr>
<td>( V )</td>
<td>( \frac{\partial \mathbf{v}}{\partial s} ) [ ]</td>
<td>( \frac{\partial \mathbf{v}_1}{\partial \mathbf{v}_2} ) [ ]</td>
<td></td>
</tr>
<tr>
<td>( M )</td>
<td>( \frac{\partial \mathbf{M}}{\partial s} ) [ ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tensors
Conventions:

- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar $s \in \mathbb{R}^1$, vector $v \in \mathbb{R}^m$, i.e. $v = [v_1, v_2, ..., v_m]^T$ and matrix $M \in \mathbb{R}^{k \times \ell}$

- What is the size of $\frac{\partial v}{\partial s}$? $\mathbb{R}^{m \times 1}$ (column vector of size $m$)

- What is the size of $\frac{\partial s}{\partial v}$? $\mathbb{R}^{1 \times m}$ (row vector of size $m$)

\[
\frac{\partial s}{\partial v} = \begin{bmatrix}
\frac{\partial s}{\partial v_1} & \frac{\partial s}{\partial v_2} & \cdots & \frac{\partial s}{\partial v_m}
\end{bmatrix}
\]
Conventions:

- What is the size of $\frac{\partial v_1}{\partial v_2}$? A matrix:

\[
\begin{bmatrix}
\frac{\partial v_1^1}{\partial v_1} & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial v_1^2}{\partial v_1} & \cdots & \cdots & \cdots & \cdots \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
\frac{\partial v_i^1}{\partial v_1} & \cdots & \frac{\partial v_i^1}{\partial v_j} & \cdots & \frac{\partial v_i^1}{\partial v_{m_2}} \\
\frac{\partial v_i^2}{\partial v_1} & \cdots & \frac{\partial v_i^2}{\partial v_j} & \cdots & \frac{\partial v_i^2}{\partial v_{m_2}} \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}
\]

- This matrix of partial derivatives is called a Jacobian

(Note this is slightly different convention than on Wikipedia). Also, computationally other conventions are used.
Conventions:

What is the size of $\frac{\partial s}{\partial M}$? A matrix:

$$
\begin{pmatrix}
\frac{\partial s}{\partial m_{[1,1]}} & \cdots & \cdots & \cdots & \cdots \\
\vdots & \ddots & \cdots & \cdots & \vdots \\
\vdots & \cdots & \frac{\partial s}{\partial m_{[i,j]}} & \cdots & \vdots \\
\vdots & \cdots & \cdots & \ddots & \vdots \\
\vdots & \cdots & \cdots & \cdots & \ddots
\end{pmatrix}
$$
Example 1:
\[ y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x \\ x^2 \end{bmatrix} \]
\[ \frac{\partial y}{\partial x} = \begin{bmatrix} 1 \\ 2x \end{bmatrix} \]

Example 2:
\[ y = w^T x = \sum_k w_k x_k \]
\[ \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix} \]
\[ = [w_1, \ldots, w_m] \quad \text{because} \quad \frac{\partial (\sum_k w_k x_k)}{\partial x_i} = w_i \]
\[ = w^T \]
Example 3:

\[ \frac{\partial(wAw)}{\partial w} = 2w^T A \text{ (assuming } A \text{ is symmetric)} \]

Example 4:

\[ y = Wx \]
\[ \frac{\partial y}{\partial x} = W \]

Row \(i\)

\[
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1} \\
\vdots \\
\frac{\partial y_i}{\partial x_j} \\
\vdots \\
\frac{\partial y_n}{\partial x_j}
\end{bmatrix}
\]

Col \(j\)

\[
\begin{bmatrix}
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

\[ y_i = \sum_j w_{ij}x_j \]
What is the size of $\frac{\partial L}{\partial W}$?

Remember that loss is a **scalar** and $W$ is a matrix:

\[
W = \begin{bmatrix}
w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\
w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\
w_{31} & w_{32} & \cdots & w_{3m} & b_3 \\
\end{bmatrix}
\]

Jacobian is also a matrix:

\[
\begin{bmatrix}
\frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial w_{1m}} & \frac{\partial L}{\partial b_1} \\
\frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \cdots & \frac{\partial L}{\partial w_{2m}} & \frac{\partial L}{\partial b_2} \\
\frac{\partial L}{\partial w_{31}} & \frac{\partial L}{\partial w_{32}} & \cdots & \frac{\partial L}{\partial w_{3m}} & \frac{\partial L}{\partial b_3} \\
\end{bmatrix}
\]
Batches of data are matrices or tensors (multi-dimensional matrices)

Examples:
- Each instance is a vector of size $m$, our batch is of size $[B \times m]$
- Each instance is a matrix (e.g. grayscale image) of size $W \times H$, our batch is $[B \times W \times H]$
- Each instance is a multi-channel matrix (e.g. color image with R,B,G channels) of size $C \times W \times H$, our batch is $[B \times C \times W \times H]$

Jacobians become tensors which is complicated
- Instead, flatten input to a vector and get a vector of derivatives!
- This can also be done for partial derivatives between two vectors, two matrices, or two tensors
Vectorization and Jacobians of Simple Layers
Composition of Functions: $f(g(x)) = (f \circ g)(x)$

A complex function (e.g. defined by a neural network):

$$f(x) = g_\ell \ (g_{\ell-1} (\ldots g_1(x)))$$

$$f(x) = g_\ell \circ g_{\ell-1} \ldots \circ g_1$$

(Many of these will be parameterized)

(Note you might find the opposite notation as well!)
Jacobian View of Chain Rule
Chain Rule: Cascaded
Input: \( x \in \mathbb{R}^D \)

Binary label: \( y \in \{-1, +1\} \)

Parameters: \( w \in \mathbb{R}^D \)

Output prediction: \( p(y = 1|x) = \frac{1}{1+e^{-w^T x}} \)

Loss: \( L = \frac{1}{2} \|w\|^2 - \lambda \log(p(y|x)) \)

Adapted from slide by Marc'Aurelio Ranzato
We have discussed **computation graphs for generic functions**

Machine Learning functions (input -> model -> loss function) is also a computation graph

We can use the **computed gradients from backprop/automatic differentiation** to update the weights!

\[
-T + u - \log \left( \frac{1}{1 + e^{-w^T x}} \right)
\]

\[
\begin{align*}
\text{Neural Network Computation Graph} \\
\end{align*}
\]
Automatic differentiation:
- Carries out this procedure for us on arbitrary graphs
- Knows derivatives of primitive functions
- As a result, we just define these (forward) functions **and don’t even need to specify the gradient (backward) functions!**

Example Gradient Computations

\[
\bar{L} = 1 \\
\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}
\]

where \( p = \sigma(w^T x) \) and \( \sigma(x) = \frac{1}{1+e^{-x}} \)

\[
\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p} \sigma(1 - \sigma)
\]

\[
\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial w} = \bar{u} x^T
\]

We can do this in a combined way to see all terms together:

\[
\bar{w} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x)) x^T \\
= -\left(1 - \sigma(w^T x)\right) x^T
\]

This effectively shows gradient flow along path from \( L \) to \( w \)
The chain rule can be computed as a series of scalar, vector, and matrix linear algebra operations. Extremely efficient in graphics processing units (GPUs).

\[ w^Tx \rightarrow \frac{1}{1 + e^{-u}} \rightarrow p \rightarrow -\log(p) \]

\[ \bar{w} = -\frac{1}{\sigma(w^Tx)} \sigma(w^Tx)(1 - \sigma(w^Tx))x^T \]
Define:
\[ h_i = w_i^T h^{\ell-1} \]

\[ h^\ell = W h^{\ell-1} \]
Define:

\[ h_i = w_i^T h^{\ell-1} \]

\[ \frac{\partial h_i^\ell}{\partial w_i} = h^{(\ell-1),T} \]

\[ \frac{\partial h^\ell}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^\ell} \cdot \frac{\partial h^\ell}{\partial h^{\ell-1}} \]

\[ 1 \times |h^{\ell-1}| \quad 1 \times |h^\ell| \quad |h^\ell| \times |h^{\ell-1}| \]
\[ h^\ell = W h^{\ell-1} \]

\[ \frac{\partial h^\ell}{\partial h^{\ell-1}} = W \]

Define:

\[ h_i = w_i^T h^{\ell-1} \]

\[ \frac{\partial h_i^\ell}{\partial w_i} = h^{(\ell-1),T} \]

Note doing this on full \( W \) matrix would result in Jacobian tensor!

But it is sparse – each output only affected by corresponding weight row

\[ \frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_i^\ell} \cdot \frac{\partial h_i^\ell}{\partial w_i} \]

\[ 1 \times |h^{\ell-1}| 1 \times |h^\ell| \quad |h^\ell| \times |h^{\ell-1}| \]
We can employ any differentiable (or piecewise differentiable) function.

A common choice is the Rectified Linear Unit:
- Provides non-linearity but better gradient flow than sigmoid
- Performed element-wise

How many parameters for this layer?

\[ h^\ell = \max(0, h^{\ell-1}) \]
Full Jacobian of ReLU layer is large (output dim x input dim)

- But again it is sparse
- Only diagonal values non-zero because it is element-wise
- An output value affected only by corresponding input value

Max function funnels gradients through selected max
- Gradient will be zero if input <= 0

\[ h^\ell = \max(0, h^{\ell-1}) \]

\[ \frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial h^{\ell-1}} \]

\[ \frac{\partial L}{\partial h^{\ell-1}} = \begin{cases} 1 & \text{if } h^{\ell-1} > 0 \\ 0 & \text{otherwise} \end{cases} \]
Backpropagation and Automatic Differentiation
Backpropagation does not really spell out how to efficiently carry out the necessary computations.

But the idea can be applied to any directed acyclic graph (DAG).

- Graph represents an ordering constraining which paths must be calculated first.

Given an ordering, we can then iterate from the last module backwards, applying the chain rule.

- We will store, for each node, its gradient outputs for efficient computation.
- We will do this automatically by computing backwards function for primitives and as you write code, express the function with them.

This is called reverse-mode automatic differentiation.

A General Framework
Computation = Graph

- Input = Data + Parameters
- Output = Loss
- Scheduling = Topological ordering

Auto-Diff

- A family of algorithms for implementing chain-rule on computation graphs

Deep Learning = Differentiable Programming
We want to find the partial derivative of output $f$ (output) with respect to all intermediate variables.

1. Assign intermediate variables
2. Simplify notation: Denote bar as $\overline{a_3} = \frac{\partial f}{\partial a_3}$
3. Start at end and move backward

Example:

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$
\[ f(x_1, x_2) = x_1x_2 + \sin(x_2) \]

\[
\begin{align*}
\overline{a_3} &= \frac{\partial f}{\partial a_3} = 1 \\
\overline{a_1} &= \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = \overline{a_3} \\
\overline{a_2} &= \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = \overline{a_3} \\
\overline{x_2^{P1}} &= \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = \overline{a_1} \cos(x_2) \\
\overline{x_2^{P2}} &= \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1x_2)}{\partial x_2} = \overline{a_2}x_1 \\
\overline{x_1} &= \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2
\end{align*}
\]

Gradients from multiple paths summed
Patterns of Gradient Flow: Addition

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_2) \]

\[
\begin{align*}
\frac{\partial f}{\partial a_1} &= \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{1}{1} = \bar{a}_3 \\
\frac{\partial f}{\partial a_2} &= \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = \bar{a}_3
\end{align*}
\]

Addition operation distributes gradients along all paths!
Multiplication is a gradient switcher (multiplies it by the values of the other term)

\[
\begin{align*}
\overline{x_2} &= \frac{\partial f}{\partial a_2} \cdot \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \cdot \frac{\partial (x_1x_2)}{\partial x_2} = \overline{a_2}x_1 \\
\overline{x_1} &= \frac{\partial f}{\partial a_2} \cdot \frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2
\end{align*}
\]

\(f(x_1, x_2) = x_1x_2 + \sin(x_2)\)
Several other patterns as well, e.g.:

Max operation selects which path to push the gradients through

- Gradient flows along the path that was “selected” to be max
- This information must be recorded in the forward pass

The flow of gradients is one of the most important aspects in deep neural networks

- If gradients do not flow backwards properly, learning slows or stops!
Key idea is to **explicitly store computation graph** in memory and **corresponding gradient functions**

Nodes broken down to **basic primitive computations** (addition, multiplication, log, etc.) for which corresponding derivative is known

\[
\bar{x}_2 = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = \bar{a}_1 \cos(x_2)
\]
Note that we can also do **forward mode** automatic differentiation.

Start from **inputs** and propagate gradients forward.

Complexity is proportional to input size.

- Memory savings (all forward pass, no need to store activations)

- However, in most cases our **inputs** (images) are large and **outputs** (loss) are small.

\[
\begin{align*}
\dot{w}_1 &= \cos(x_1)\dot{x}_1 \\
\dot{w}_2 &= \dot{x}_1 x_2 + x_1 \dot{x}_2 \\
\dot{w}_3 &= \dot{w}_1 + \dot{w}_2
\end{align*}
\]
A graph is created on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```

(Note above)
Computation Graphs in PyTorch

from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))

Back-propagation uses the dynamically built graph

From pytorch.org
Convolutional network (AlexNet)

- Input image
- Weights
- Loss
Neural Turing Machine

input image

loss

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
Computation graphs are **not** limited to mathematical functions!

Can have **control flows** (if statements, loops) and **backpropagate** through algorithms!

Can be done **dynamically** so that **gradients are computed**, then **nodes are added**, repeat

**Differentiable programming**