PROBABILISTIC AND DIFFERENTIABLE PROGRAMMING

V7: Automatic Differentiation (AD)

Özgür L. Özçep
Universität zu Lübeck
Institut für Informationssysteme
Today’s Agenda

\[
\frac{\partial}{\partial x}(\text{HTML code})
\]
WHY YOU NEED AD
**Reminder:** Backprop = AD in reverse mode

<table>
<thead>
<tr>
<th>Sample labeled data (batch)</th>
<th><strong>Forward</strong> it through the network, get predictions</th>
<th><strong>Back-propagate</strong> the errors</th>
<th><strong>Update</strong> the network weights</th>
</tr>
</thead>
</table>

Backpropagation idea
- Generate **error signal** that measures difference between predictions and target values
- Use error signal to change the weights and get more accurate predictions backwards
- Underlying mathematics: chain rule

**Chain rule (1-dim)**

\[
\frac{dh}{dx} = \frac{df}{dg} \frac{dg}{dx}
\]

(for \( h(x) = f(g(x)) \))

---

**Figure 1: Overview of backpropagation.**

(a) **Forward pass**

(b) **Backward pass**

\[
\frac{dE}{dw} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial w}
\]

\( E(y_3, t) \)
**Reminder: Computational graph perspective**

**Function f**

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

for \( q = x + y \)

**Partial Derivatives**

\[
\frac{\partial f}{\partial z} = q \\
\frac{\partial f}{\partial q} = z \\
\frac{\partial f}{\partial x} = 1 \\
\frac{\partial f}{\partial y} = 1
\]

**Chain rule applied**

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \\
\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z
\]

**Gradient**

\[ \nabla_{x,y,z}f = (z, z, q) \]

(In particular: \( \nabla_{x,y,z}f(-2,5,-4) = (-4,-4,3) \))

**Diagram**

Forward pass:
function values and local gradients
Backward: chain rule
To solve optimisation problems using gradient methods we need to compute the gradients (derivatives) of the objective with respect to the parameters.

- In neural nets we’re talking about the gradients of the loss function, $L$ with respect to the parameters $\theta$
- AD is at the heart of ”Differentiable Programming“ (the next big thing after deep learning)
  - AD is a topic on its own
  - But has been come into focus with Differentiable Programming and lead to many developments in the intersection of programming languages, numerical computing, and ML
• Symbolically differentiate the function with respect to its parameters
  – by hand
  – using a CAS

• Make estimates using finite differences
  \[ f'(a) \approx \frac{f(a+he_i)-f(a)}{h} \]

• Use Automatic Differentiation

• Problem: Static, expression swell. Can’t differentiate algorithms

• Problem: Numerical errors (such as rounding and truncation errors)
Problem with symbolic computation

\[
\frac{d(f(x) \cdot g(x))}{dx} = \frac{d f(x)}{dx} g(x) + \frac{d g(x)}{dx} f(x) \quad \text{(Product rule)}
\]

- \( h(x) : = g(x) \cdot f(x) \)
- \( \frac{dh(x)}{dx} \) and \( h \) have two components in common
- This may also be the case for \( f \).
- Symbolically calculating \( f \) won’t profit from common parts of \( f \) and \( \frac{df(x)}{dx} \)
Problems with numerical calculation

Truncation error:
Approximation error due to not sufficiently small $h$
  - tends to 0 for $h \to 0$

Rounding error: due to limited precision in computation
  - Increases for $h \to 0$

Can be mitigated partly by using centered approximation

$$f'(a) \approx \frac{f(a+he_i) - f(a-he_i)}{h}$$

(error shift from $O(h)$ to $O(h^2)$)
- **Automatic Differentiation** is a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.
Automatic Differentiation in Machine Learning: a Survey

\[ l_1 = x \]
\[ l_{n+1} = 4l_n(1 - l_n) \]
\[ f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2 \]

\[ f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)^2 + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2 \]

\[ f'(x_0) = f'(x_0) \]

Manual Differentiation

\[ f(x) : \]
\[ v = x \]
\[ \text{for } i = 1 \text{ to } 3 \]
\[ v = 4v*(1 - v) \]
\[ \text{return } v \]

or, in closed-form,

\[ f(x) : \]
\[ \text{return } 64x(1-x)((1-2x)^2) * (1-8x+8x^2)^2 \]

Coding

Ex: Baydin et al. 2017

No one has time for manual computation

Gives you smell of the expression swell

Can directly reuse program with for-loop - no need for closed-form

Small h (as we have seen) does not help w.r.t. rounding errors

Automatic Differentiation

\[ f'(x) : \]
\[ (v, dv) = (x, 1) \]
\[ \text{for } i = 1 \text{ to } 3 \]
\[ (v, dv) = (4*v*(1-v), 4*dv-8*v*dv) \]
\[ \text{return } (v, dv) \]

\[ f'(x_0) = f'(x_0) \]

Exact

Numerical Differentiation

\[ f'(x) : \]
\[ h = 0.000001 \]
\[ \text{return } (f(x + h) - f(x)) / h \]

\[ f'(x_0) \approx f'(x_0) \]

Approximate

Symbolic Differentiation

\[ f(x) : \]
\[ \text{return } 66*x*(1-x)*((1-2*x)^2) * (1-8*x+8*x^2)^2 \]

Exact

Edition 2022 | LÜBECK UNIVERSITY
AUTOMATIZATION
From Differentiation to Programming

• Example (Math)

\[
\begin{align*}
x &= ? \\
y &= ? \\
a &= xy \\
b &= \sin(x) \\
z &= a + b
\end{align*}
\]

• Example (code)

```python
x= ?
Y= ?
a = x * y 
b = \sin(x)
z = a + b
```
The chain rule for vectors

Given functions $f, g$ with

- $\mathbb{R}^m \xrightarrow{g} \mathbb{R}^n \xrightarrow{f} \mathbb{R}$
- $x \mapsto y = g(x) \mapsto z = f(y)$

the chain rule leads to the partial derivatives

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

(in short form: $\nabla_x Z = \left( \frac{\partial y}{\partial x} \right)^\top \nabla_y Z$

where $\left( \frac{\partial y}{\partial x} \right)$ is the $n \times m$ Jacobian matrix of $g$)
Let us rename for the following

Given functions $f, g$ with

- $\mathbb{R}^m \xrightarrow{g} \mathbb{R}^n \xrightarrow{f} \mathbb{R}$
- $t \mapsto u = g(x) \mapsto w = f(u)$

the chain rule leads to the partial derivatives

$$\frac{\partial w}{\partial t} = \sum_j \frac{\partial w}{\partial u_j} \frac{\partial u_j}{\partial t}$$

$w$ is some output variable from a family of outputs $\{w_i\}$ and $u_j$ are the inputs variables $w$ depends on.
Applying the chain rule

Example expression

\[ x = ? \]
\[ y = ? \]
\[ a = xy \]
\[ b = \sin(x) \]
\[ z = a + b \]

Derivatives w.r.t. some yet to be given variable \( t \)

\[ \frac{\partial x}{\partial t} = ? \]
\[ \frac{\partial y}{\partial t} = ? \]
\[ \frac{\partial a}{\partial t} = x \frac{\partial y}{\partial t} + y \frac{\partial x}{\partial t} \]
\[ \frac{\partial b}{\partial t} = \cos x \frac{\partial x}{\partial t} \]
\[ \frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t} \]

- If we substitute \( t = x \) we get an algorithm for computing \( \frac{\partial z}{\partial x} \).
- Choosing \( t = y \) similarly gives \( \frac{\partial z}{\partial y} \).
Translating to code

<table>
<thead>
<tr>
<th>Derivatives as programs</th>
<th>Substituting $t = x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dx=\ ?$</td>
<td>$dx = 1$</td>
</tr>
<tr>
<td>$dY= \ ?$</td>
<td>$dy = 0$</td>
</tr>
<tr>
<td>$da = y \ast dx + x \ast dy$</td>
<td>$da = y \ast dx + x \ast dy$</td>
</tr>
<tr>
<td>$db = \cos(x) \ast dx$</td>
<td>$db = \cos(x) \ast dx$</td>
</tr>
<tr>
<td>$dz = da + db$</td>
<td>$dz = da + db$</td>
</tr>
</tbody>
</table>

(Using the notation $dx = \frac{\partial x}{\partial t}, dy = \frac{\partial y}{\partial t}, \ldots$)

So, to compute $\frac{\partial z}{\partial x}$ just seed algorithm with $dx = 1$, $dy = 0$
Translating to code

### Derivatives as programs

<table>
<thead>
<tr>
<th>dx=?</th>
<th>dY= ?</th>
<th>da = y * dx + x * dy</th>
<th>db = cos(x)*dx</th>
<th>dz = da + db</th>
</tr>
</thead>
</table>

(Using the notation

\[ dx = \frac{\partial x}{\partial t}, \quad dy = \frac{\partial y}{\partial t}, \ldots \] )

### Substituting \( t = y \)

| dx= 0 | dy= 1 | da = y * dx + x * dy | db = cos(x)*dx | dz = da + db |

So, to compute \( \frac{\partial z}{\partial y} \) just seed algorithm with

\[ dx = 0, \quad dy = 1 \]
Making Rules

- Idea of the examples can be generalized to arbitrary functions
- Need to describe rules for translation
  program evaluating expression => program evaluating derivates
- These are just rules known from mathematics for calculating derivates, e.g.
  
  \[ \begin{align*}
  - c &= a + b & \Rightarrow & & dc &= da + db \\
  - c &= a \times b & \Rightarrow & & dc &= b \times da + a \times db \\
  - c &= \sin(a) & \Rightarrow & & dc &= \cos(a) \times da
  \end{align*} \]
- Note: These rules are used on number-level (not for symbolic computation of derivatives)
Further Rules

\[ c = a - b \quad \Rightarrow \quad dc = da - db \]
\[ c = a / b \quad \Rightarrow \quad dc = da/b - a*db/b**2 \]
\[ c = a**b \quad \Rightarrow \quad dc = b*a**(b-1)*da + \log(a)*a**b*db \]
\[ c = \cos(a) \quad \Rightarrow \quad dc = -\sin(a) * da \]
\[ c = \tan(a) \quad \Rightarrow \quad dc = da/cos(a)**2 \]

(\(a**b\) stands for \(a^b\))
FORWARD MODE
Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.
- The order of computation remains unchanged: if a statement $K$ is evaluated before another statement $L$, then the differential analogue of $K$ is evaluated before the analogue statement of $L$.
- This is **Forward-mode Automatic Differentiation**.
  - Nice feature: Interleaving (function evaluation and derivatives) is possible
  - Bad feature: Need to rerun program to compute derivative for each input (in particular for gradient)
Interleave computing expression and derivatives

- Can keep track of value and gradient at the same time
- Can be mathematically founded using “dual numbers”
- Leads to direct simple implementation of AD

\[
\begin{align*}
x &= ? \\
dx &= ? \\
y &= ? \\
dy &= ? \\
a &= x \times y \\
da &= y \times dx + x \times dy \\
b &= \sin(x) \\
db &= \cos(x) \times dx \\
z &= a + b \\
dz &= da + db
\end{align*}
\]
The Jacobian in Forward Mode AD

- \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m ; x \mapsto z \)
- Calculate derivatives w.r.t. \( i \)th variable \( x_i \) for all outputs \( z_i \) in one pass

\[
\frac{\partial z}{\partial x} = \begin{pmatrix}
\frac{\partial z_1}{\partial x_1}(a) \\
\vdots \\
\frac{\partial z_m}{\partial x_1}(a) \\
\frac{\partial z_1}{\partial x_i}(a) \\
\vdots \\
\frac{\partial z_m}{\partial x_i}(a) \\
\vdots \\
\frac{\partial z_1}{\partial x_n}(a) \\
\vdots \\
\frac{\partial z_m}{\partial x_n}(a)
\end{pmatrix}
\]

- Efficient calculating product w.r.t. vector \( r \)
  - \( \frac{\partial z}{\partial x} \cdot r \)
  - Just seed with \( \text{dx}_1 = r_1, \ldots, \text{dx}_n = r_n \)

- Special case
  \( f : \mathbb{R}^n \rightarrow \mathbb{R} ; x \mapsto z \)
  Calculate directional derivate in direction \( r \).
  - \( \nabla f \cdot r \)
Another view on AD

- $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$
  - $v_{i-n} = x_i, i = 1, \ldots, n$  
    input variables
  - $v_i, \quad i = 1, \ldots, l$  
    intermediate variables
  - $y_{m-i} = v_{l-i}, i = m - 1, \ldots, 0$  
    output variables
  - $v_i = \phi_i(v_j)_{j<i}, \phi_i: \mathbb{R}^{nj} \rightarrow \mathbb{R}$  
    (elemental functions)
    - where $<$ is precedence relation ($j < i$ iff $v_i$ directly depends on $v_j$)
    - $n_j$ number of elements preceding $v_j$
  - $u_i = (v_j)_{j<i}$
Forward mode AD = Tangents mapping

- Assume you have time-dependent paths \( x(t), y(t) \)
- Forward mode AD is mapping function evaluation \((F: x \mapsto y)\) plus tangents mapping \( \hat{F}: \dot{x} \mapsto \dot{y} \)

\[
\begin{align*}
[v_{i-n}, \dot{v}_{i-n}] &= [x_i, \dot{x}_i] & \text{for } i = 1, \ldots, n \\
[v_i, \dot{v}_i] &= [\phi_i(u_i), \dot{\phi}_i(u_i, u_i)] & \text{for } i = 1, \ldots, l \\
[y_{m-i}, \dot{y}_{m-i}] &= [v_{l-i}, \dot{v}_{l-i}] & \text{for } i = 0, \ldots, m - 1
\end{align*}
\]
Dual numbers (Clifford 1873)

- Want to mathematize parallel evaluation of $f, f'$
- Dual numbers have the form $(v + \dot{v}\epsilon)$ where
  - $v, \dot{v} \in \mathbb{R}$
  - $\epsilon$ is a nilpotent element ($\epsilon^2 = 0, \epsilon \neq 0$)
  - compare with complex numbers $x + yi$ where $i^2 = -1$, which can be considered as pairs in $\mathbb{R}^2$ (more general: quaterions)
- Gives intended behaviour mirroring symbolic derivation
  - $(v + \dot{v}\epsilon) + (u + \dot{u}\epsilon) = (v + u) + (\dot{v} + \dot{u})\epsilon$
  - $(v + \dot{v}\epsilon)(u + \dot{u}\epsilon) = (vu) + (v\dot{u} + \dot{v}u)\epsilon$
Dual numbers (Clifford 1873)

- Can define functions $f$ on dual numbers by
  \[ f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon \]
  (results from Taylor series application)

- Then: Chain rule works as expected:
  \[ f(g(v + \dot{v}\epsilon)) = f(g(v) + g'(v)\dot{v}\epsilon) \]
  \[ = f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon \]

- Can extract derivative
  \[ \frac{df}{dx}(v) = \epsilon - \text{coeff (dual} - \text{version}(f)(v + 1\epsilon)) \]
REVERSE MODE
Reverse Mode AD

• Whilst Forward-mode AD is easy to implement, it comes with a very big disadvantage. . .
• For every variable we wish to compute the gradient with respect to, we have to run the complete program again.
• This is obviously going to be a problem if we’re talking about the gradients of a function with very many parameters (e.g. a deep network).
• A solution is Reverse Mode Automatic Differentiation.
Reversing the Chain Rule

- Conceptually, chain rule doesn’t care about role of enumerator and denominator – can turn it upside down

  \[- \frac{\partial w}{\partial t} \text{ becomes } \frac{\partial t}{\partial w} \]

  becomes by renaming (s for t and u for w)

  \[- \frac{\partial s}{\partial w} \text{ is by applying chain} \]

  \[- \frac{\partial s}{\partial u} = \sum_j \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i} \]

  - u is some input variable
  - w_i s are output variables depending on u
  - s is the yet-to-be-given variable

Now can compute in 1-pass in parallel: \( \frac{\partial s}{\partial x}, \frac{\partial s}{\partial y}, \ldots \)
Example

\[ \frac{\partial s}{\partial u} = \sum_j \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i} \]

\[ \frac{\partial s}{\partial z} = ? \]

\[ \frac{\partial s}{\partial b} = \frac{\partial s}{\partial z} \frac{\partial z}{\partial b} \]

\[ x = ? \]
\[ y = ? \]
\[ a = xy \]
\[ b = \sin(x) \]
\[ z = a + b \]
Visualising dependencies

• Differentiating in reverse can be quite mind-bending: instead of asking what input variables an output depends on, we have to ask what output variables a given input variable can affect.

• We can see this visually by drawing a dependency graph of the expression (e.g. \( x \) effects \( a \) and \( b \)): 

\[
\begin{align*}
\text{y} & \quad \text{a} \\
\times & \quad 
\begin{align*}
\text{x} & \quad \text{a} \\
\ast & \quad 
\begin{align*}
\sin & \quad 
\begin{align*}
\text{b} & \quad 
\begin{align*}
\text{z} & \quad 
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]
Translating to Code

- As before we replace the derivatives \((\partial s/\partial z, \partial s/\partial b, \ldots)\) with variables \((gz, gb, \ldots)\) which we call adjoint variables:

  - \(gz = ?\)
  - \(gb = gz\)
  - \(ga = gz\)
  - \(gy = x \times ga\)
  - \(gx = y \times ga + \cos(x) \times gb\)

- Substituting \(s = z\) in equations gives both gradients \(\frac{\partial z}{\partial x}\) and \(\frac{\partial z}{\partial y}\) in last two lines

- Equivalently set \(gz = 1\)
Reverse mode AD = Co-tangents mapping

- Reverse mode AD: function evaluation, $F: x \mapsto y$, plus co-tangents mapping by adjoint $\bar{F}: \bar{y} \mapsto \bar{x}$

\[
\bar{F}(x, \bar{y}) := \bar{y}^T F'(x) = \bar{x}
\]

\[
\begin{align*}
v_i &= 0 & \text{for } i &= 1, \ldots, l \\
[v_{i-n}, \bar{v}_{i-1}] &= [x_i, \bar{x}_i] & \text{for } i &= 1, \ldots, n \\
\text{Push}(v_i) & & \\
v_i &= \phi_i(u_i) & \text{for } i &= 1, \ldots, l \\
y_{m-1} &= v_{l-1} & \text{for } i &= 0, \ldots, m - 1 \\
\bar{v}_{l-i} &= \bar{y}_{m-1} & \text{for } i &= 0, \ldots, m - 1 \\
v_i &\leftarrow \text{pop}(\quad) \\
\tilde{u}_i &\leftarrow \bar{v}_{i} * \nabla \phi_i(u_i) & \text{for } l, \ldots, 1 \\
v_i &= 0 & \text{for } i &= 1, \ldots, 1 \\
\bar{x}_i &= \bar{v}_{i-n} & \text{for } i &= 1, \ldots, n
\end{align*}
\]

(simple algorithm without sophisticated memory management: just using stack)
But wait... Limitations of Reverse Mode AD

- We have a problem dual to that of forward AD: Now have to run the program for each output variable one is interested in differentiating

- Example
  - $z = 2x + \sin x$
  - $v = 4x + \cos x$

  Calculating $\frac{\partial z}{\partial x}$ and $\frac{\partial v}{\partial x}$ each requires running the program.

  Cannot interleave the calculations as they appear to be in reverse mode. => Recent research on automatization

- So: Reverse AD has advantage only if number of output variables much smaller than number of input variables
Implementing Reverse Mode AD

There are two ways to implement Reverse AD:

1. We can parse the original program and generate the \textit{adjoint} program that calculates the derivatives.
   - Potentially hard to do.
   - Static, so can only be used to differentiate algorithms that have parameters predefined.
   - But, efficient (lots of opportunities for optimisation)

2. We can make a \textit{dynamic} implementation by constructing a graph that represents the original expression as the program runs.
Constructing an expression graph (in Python)

- **Goal:** get a graph as

  ```python
  class Var:
  def __init__(self, value):
      self.value = value
      self.children = []

  x = Var(0.5)
  y = Var(4.2)
  ...
  ```

- **Root of the graph are independent variables** \(x, y\)

- **Can have children (initially empty):** nodes that depending on parent

![Image of expression graph]

\[ z = a \times (b + \sin x) \]
Building expressions

- Expression creation
- Self-registration of each expression $u$ as a child of each of its dependencies $w_i$
- Also register weight $\frac{\partial w_i}{\partial u}$ (used for gradient calculation)

```python
class Var:
    ...
    def __mul__(self, other):
        z = Var(self.value * other.value)

        # weight = dz/dself = other.value
        self.children.append((other.value, z))

        # weight = dz/dother = self.value
        other.children.append((self.value, z))

    return z
    ...
...

# "a" is a new Var that is a child of both x and y
# a=x*y
Computing gradients

- Propagate Derivatives
- Cache derivatives in grad_value

```python
class Var:
    def __init__(self):
        ...
        self.grad_value = None
    
def grad(self):
        if self.grad_value is None:
            # using chain rule
            self.grad_value =
                sum(weight * var.grad()
                for weight, var in self.children)
        return self.grad_value ...
    ...

a.grad_value = 1.0
print("da/dx_=_{}".format(x.grad()))
```

\[ y = a \times x + \sin(z) \]
Optimising reverse Mode AD

• The outline implementation not very space efficient
  – Instead of children directly store in indices (Wengert list, tape)

• Space efficiency for reverse AD is challenging hence research topic
  – Count-Trailing-Zeros CTZ): trade-off computation for memory of caches (Griewank 92).
  – But, in reality memory is relatively cheap (if managed well)
CTZ example

- Idea: Hierarchical cache storing only \( O(\log(N)) \) of all \( N \) values in expression in forward sweep and maintained during reverse sweep
  - \( \text{Cache}_0 \) store first value
  - \( \text{Cache}_1 \) stores value at \( \frac{1}{2} \) down chain
  - \( \text{Cache}_2 \) stores value at \( \frac{3}{4} \) down the chain ...
  - \( \text{Cache}_{n-1} \) stores value at \( \frac{n}{n+1} \) down the chain
- Assume linear expression of \( N=16 \) Operators
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (value indices)
  - X-------------------X------X--X X (stored value indication)
CTZ example (continued)

• Reverse sweep (with head postion _)
  – 0 1 2 3 4 5 6 7 8 9 a b c d e f _  (can swee over e,f)
  – X------------------X------X---X X

  – 0 1 2 3 4 5 6 7 8 9 a b c d e f  (d not cached, recalculate)
  – X------------------X------X---X X

  – 0 1 2 3 4 5 6 7 8 9 a b c d e f  (d not cached, recalculate)
  – X------------------X------X---X X  from cached c)
  +X
CTZ example (continued)

- Reverse sweep (with head position _)
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (sweep over c, missing b, sweep to 8 and cache a)
  - X-------------------X------X---X X
    +X
  - 0 1 2 3 4 5 6 7 8 9 a b c d e f (recompute 9 from 8, then 7 to be recomputed)
  - X-------------------X------X---X X
    +--X-X+-X
    move to 0, store along 6,4)

  - 0 1 2 3 4 5 6 _ 8 9 a b c d e f (and so on ...)
  - X-------------------X------X---X X
  - +------X--X-X++++-X-X+-X
  - +--X

CTZ example (continued)

- In the end
  - 0 1 2 3 4 5 6 8 9 a b c d e f
  - X-----------------X------X--X X
  - +--------X--X-X+++X-X+-X
  - +---X-X       +X
  - +-X
Uhnh, a lecture with a hopefully useful

APPENDIX
Color Convention in this Course

- Formulae, when occurring inline
- Newly introduced terminology and definitions
- Important results (observations, theorems) as well as emphasizing some aspects
- Examples are given with standard orange with possibly light orange frame
- Comments and notes in nearly opaque post-it
- Algorithms and program code
- Reminders (in the grey fog of your memory)
Today’s lecture is based on the following

- Jonathon Hare: Lecture 5 of course „COMP6248 Differentiable Programming (and some Deep Learning)“
  [http://comp6248.ecs.soton.ac.uk/](http://comp6248.ecs.soton.ac.uk/)

- Blog post by Rufflewind: Reverse-mode automatic differentiation: a tutorial
  [https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation](https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation)


References