SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 13

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, October 13th 2022

LAST TIME

- Beyond the basics:
  - Computing derivatives in higher dimensions using the forward mode
  - The Jacobian in forward mode
  - What the forward mode actually computes
- Review of complex numbers
- Dual numbers in forward mode AD

TODAY

Main topics: ***Automatic differentiation: forward mode, dual numbers, implementation, operator overloading, automatic differentiation: reverse mode***

Details:

- Dual numbers exercise
- Implementation approaches for automatic differentiation
- Operator overloading
- Reverse mode AD
- Examples for application and project extensions

DUAL NUMBERS: EXERCISE

Given the function: $f(x) = \frac{\sin(x)}{(\cos(x))^2+1}$

Perform the following tasks:

1. Draw the computational graph for $f(x)$. The last intermediate variable is $v_5 = f(x_1)$, where x_1 is the point where we evaluate f.
2. Show that $D_p v_5$ takes the form

$$D_p v_5 = \frac{1}{v_4^2}(v_4 D_p v_1 - v_1 D_p v_4)$$

   for $v_5 = g(v_1, v_4)$ with g some function (***hint*** → chain rule).
3. Compute the last intermediate state with dual numbers $z_5 = g(z_1, z_4)$. Note that the function g is the same as in item 2 above, we just use dual numbers this time. Depending on how you draw the graph, the arguments to g may have different subscripts.

AUTOMATIC DIFFERENTIATION: FORWARD MODE

General comments on implementation:

- The computational graph we studied earlier identifies the *nodes* associated to intermediate variables v_j. The evaluation of v_j depends on its *parents* in the graph. Node v_i is a ***parent*** of the ***child*** node v_j whenever there is a *directed arc* from i to j. This implies some structure for the data in the algorithm.
- There is ***no need*** to construct the computational graph, break down the problem into its partial ordering or identify intermediate variables *manually*. Automatic (or a better word is *algorithmic*) differentiation software can perform these tasks implicitly via the implemented algorithm (operator precedence) and data structures (e.g. dual numbers).
- Once a child node is evaluated, its parent node(s) are no longer needed (if the parent has no more other children that must be evaluated) and can therefore be overwritten or discarded. *There is no need to store the full graph of v_j and $D_p v_j$ pairs.*

This is a strength of forward mode AD as the computational graph can become very large for non-trivial functions $f(x)$.

IMPLEMENTATION: OPERATOR OVERLOADING

There are different implementation techniques for automatic differentiation. Two techniques often used are:

1. ***Code translation*** on the level of intermediate representation (IR). This happens on the *compiler level* and is therefore very efficient.
2. ***Operator-overloading*** on the *software level*. We have already touched this topic when we studied the Python data model.

Code translation is not in the scope of this class. We will focus on ***operator-overloading*** for which you need to apply what you have learned about the Python data model and its relation to the special (dunder) methods.

IMPLEMENTATION: OPERATOR OVERLOADING

What is "operator overloading"?

- *An operator is essentially a function*. Recall the "+" operator is a function call in the Python data model.
- Consider for example $\sin(x)$: here the sine is a *mathematical operator* that *acts* on an argument x.
- Your intuition is that the operator "$\sin$" acts on a *real number* $x \in \mathbb{R}$.
- How should the "$\sin$" operator act if its argument was a dual number z? ***In order to define this behavior, the operator must be overloaded***.
- Operator overloading is a ***form of polymorphism*** where an operator may have different implementations (therefore different behavior) depending on the argument it acts on.
- ***Example:*** `np.sin` is overloaded to act on arguments that are NumPy arrays.

IMPLEMENTATION: OPERATOR OVERLOADING

- Let us revisit the `Complex` class type from lecture 7:

```
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag
```

- Back then we did not talk about why adding two complex numbers together like that does not work:

```
>>> z1 = Complex(1, 1)
>>> z2 = Complex(2, 2)
>>> z3 = z1 + z2
Traceback (most recent call last):
  File "/tmp/complex.py", line 14, in <module>
    z3 = z1 + z2
TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

- As you learned about the Python data model you know how to fix this 

IMPLEMENTATION: OPERATOR OVERLOADING

- As you learned about the Python data model you know how to fix this 

```
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag

    def __add__(self, other):
        """Adding complex numbers together"""
        return Complex(self.real + other.real, self.imag + other.imag)
```

```
>>> z1 = Complex(1, 1)
>>> z2 = Complex(2, 2)
>>> z3 = z1 + z2
>>> vars(z3)
{'real': 3, 'imag': 3}
```

IMPLEMENTATION: OPERATOR OVERLOADING

- As you learned about the Python data model you know how to fix this 

```python
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag

    def __add__(self, other):
        """Adding complex numbers together"""
        return Complex(self.real + other.real, self.imag + other.imag)
```

- ***The interface is of course very incomplete.*** What about multiplication, division, subtraction and other operators? All the other big libraries like NumPy for example, must ***overload*** this behavior for their own types.
- Another common (and valid) operation for complex numbers we have not talked about yet could be `z4 = 1 + z3` with `1` being a floating point number or integer (a real number). ***What happens in this case?***

IMPLEMENTATION: OPERATOR OVERLOADING

- Another common (and valid) operation for complex numbers we have not talked about yet could be `z4 = 1 + z3` with `1` being a floating point number or integer (a real number). ***What happens in this case?***
- Python evaluates the right-hand side from left to right which results in this function call: `z4 = 1.__add__(z3)`.
- It is highly unlikely that Python integer objects support your custom `Complex` type. *This operation will fail with a* `NotImplementedError`.
- In that case Python checks if the other object implements the `__radd__` special method which will then be called instead. The "r" stands for ***reflected*** or swapped. The argument to `__radd__` would be the integer `1` in this case.
- You can simply call `__add__` from within `__radd__` ***if the operator is commutative***, but be careful to handle the type of `other` correctly. Here `other` is an *integer* and not a `Complex` type.

IMPLEMENTATION: OPERATOR OVERLOADING

Example: dual number

- Walk through a simple dual number implementation with support for addition.
- The implementation should also support addition with scalars.
- Eventually, your project AD library must be able to handle overloaded elementary transcendental functions like $\sin$, $\cos$, $\exp$, $\ln$, computing powers as well as other elementary functions → https://harvard-iacs.github.io/2022-CS107/project/M2/#M2-functions

AUTOMATIC DIFFERENTIATION: REVERSE MODE

- The reverse mode of automatic differentiation is a ***two-pass*** process as opposed to the m-pass forward mode (when computing *full gradient*).
- Reverse mode does not evaluate v_j and $D_p v_j$ simultaneously!
- For this reason the useful properties of dual numbers in forward mode *are not* useful in reverse mode.
- Reverse mode recovers the *partial derivatives of the i-th output f_i with respect to the n variables v_{j-m}* with $j = 1, 2, \ldots, n$ by traversing the computational graph ***backwards***. These partial derivatives describe the *sensitivity* of the output with respect to the intermediate variable v_{j-m}:

$$\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}.$$

We call $\bar{v}_{j-m}$ the ***adjoint*** of v_{j-m}.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

- Reverse mode recovers the *partial derivatives of the i-th output f_i with respect to the n variables v_{j-m}* with $j = 1, 2, \ldots, n$ by traversing the computational graph ***backwards***. These partial derivatives describe the *sensitivity* of the output with respect to the intermediate variable v_{j-m}:

$$\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}.$$

  We call $\bar{v}_{j-m}$ the ***adjoint*** of v_{j-m}.

- ***Recall:*** we have defined *(the first m intermediate results are the independent coordinates)*:

$$v_{j-m} = x_j \text{ for } j = 1, 2, \ldots, m$$

So the first m adjoints in the reverse mode are the m components of the gradient ∇f_i!

AUTOMATIC DIFFERENTIATION: REVERSE MODE

What is the difference between forward and reverse mode?

- ***Forward mode*** computes the gradient ∇f with respect to the *independent* variable x.
- ***Reverse mode*** computes the sensitivity $\bar{v}_{j-m}$ of f with respect to the independent ***and*** intermediate variables v_{j-m}. We therefore recover the gradient ∇f in reverse mode as well.
- Compared to the forward mode, the reverse mode has a *significantly smaller arithmetic operation count* for mappings of the form $f(x) : \mathbb{R}^m \mapsto \mathbb{R}$ if m is very large. *Artificial neural networks have exactly this property.*
- ***No free lunch:*** we have to store the full computational graph in reverse mode. This can become a limitation for complicated functions $f(x)$!

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Forward pass

- Computes the primal trace v_j and the partial derivatives $\frac{\partial v_j}{\partial v_i}$ *with respect to v_j's parent node(s) v_i.*
- ***Note:*** the partial derivatives here are *the **outer derivatives** that show up in the chain rule*, not the chain rule itself.
- ***We do not need to apply the chain rule explicitly in reverse mode, we will "build it up" in the reverse pass instead!*** That is why we *do not* compute $D_p v_j$ in the forward pass of the reverse mode.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Forward pass

- ***We do not need to apply the chain rule explicitly in reverse mode, we will "build it up" in the reverse pass instead!*** That is why we *do not* compute $D_p v_j$ in the forward pass of the reverse mode.
- ***Compare what is being computed:***
  - ***Forward mode:*** $v_j = \sin(v_i)$ *and* $D_p v_j = \frac{\partial v_j}{\partial v_i} D_p v_i = \cos(v_i) D_p v_i$ (chain rule)
  - ***Forward pass in reverse mode:*** $v_j = \sin(v_i)$ *and* $\frac{\partial v_j}{\partial v_i} = \cos(v_i)$ (***not*** the chain rule)
- In reverse mode, we must know the relationship between ***parent*** and ***child***:

The partial derivative $\frac{\partial v_j}{\partial v_i}$ describes the change in a *child* node with respect to its *parent* node v_i. ***This is not the chain rule.***

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Reverse pass

- In the reverse pass we ***reconstruct*** the chain rule that we *ignored in the forward pass*.
- The goal is to compute the following quantity for each node v_i:

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j \text{ a child of } i} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \text{ a child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Reverse pass

- The goal is to compute the following quantity for each node v_i:

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j \text{ a child of } i} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \text{ a child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

- The partial derivatives $\frac{\partial v_j}{\partial v_i}$ are computed during the forward pass.

  At the start of the reverse pass, we initialize $\bar{v}_i = 0$ for all i and ***accumulate*** the values with

$$\bar{v}_i = \bar{v}_i + \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

  as we iterate over the children j of node i.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Reverse pass

- The partial derivatives $\frac{\partial v_j}{\partial v_i}$ are computed during the forward pass. At the start of the reverse pass, we initialize $\bar{v}_i = 0$ for all i and ***accumulate*** the values with

$$\bar{v}_i = \bar{v}_i + \frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

  as we iterate over the children j of node i.

- Once ***all*** contributions from child nodes are ***accumulated*** in node i, we can proceed with updating its parent node(s). If $\bar{v}_j$ for a particular child node ***is not complete*** we can not proceed with $\bar{v}_i$ and must continue with another node instead.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Reverse pass

- The very last intermediate state is $v_{n-m} = f(x)$ with $x \in \mathbb{R}^m$ and ***this last node obviously has no children***.
- ***Recall:*** n is the sum of the *independent* variables (the number m) and *dependent* variables (nodes) in the graph with $i > 0$.
- We therefore know the initial value of the last adjoint $\bar{v}_{n-m}$:

$$\bar{v}_{n-m} = \frac{\partial f}{\partial v_{n-m}} = \frac{\partial v_{n-m}}{\partial v_{n-m}} = 1$$

- The initial state is required to start the reverse pass. The computational graph is ***traversed backwards***, from the right (outputs) to the left (inputs). In the reverse pass we *only* work with ***numerical values***, no operator overloading needed.

REVERSE MODE: EXAMPLE

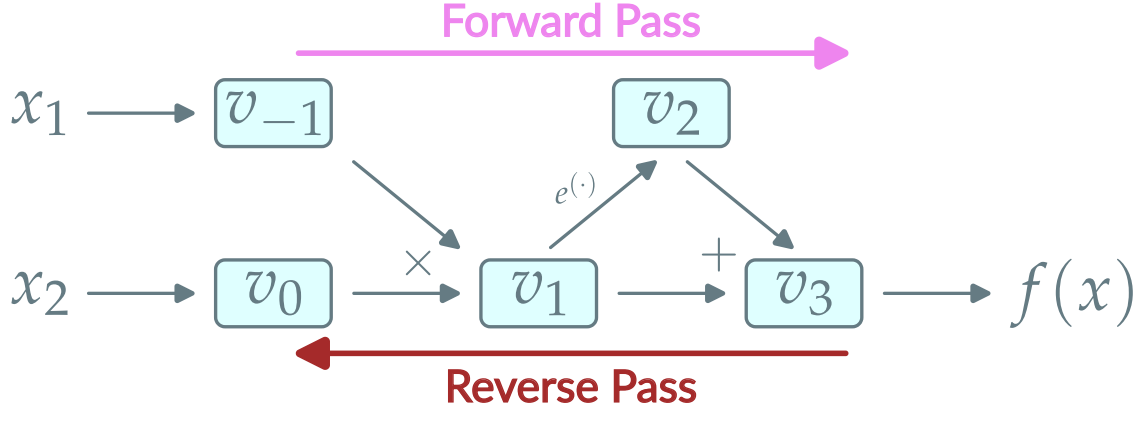
- Consider the following function $f(x) : \mathbb{R}^2 \mapsto \mathbb{R}$

$$f(x) = x_1 x_2 + e^{x_1 x_2}$$

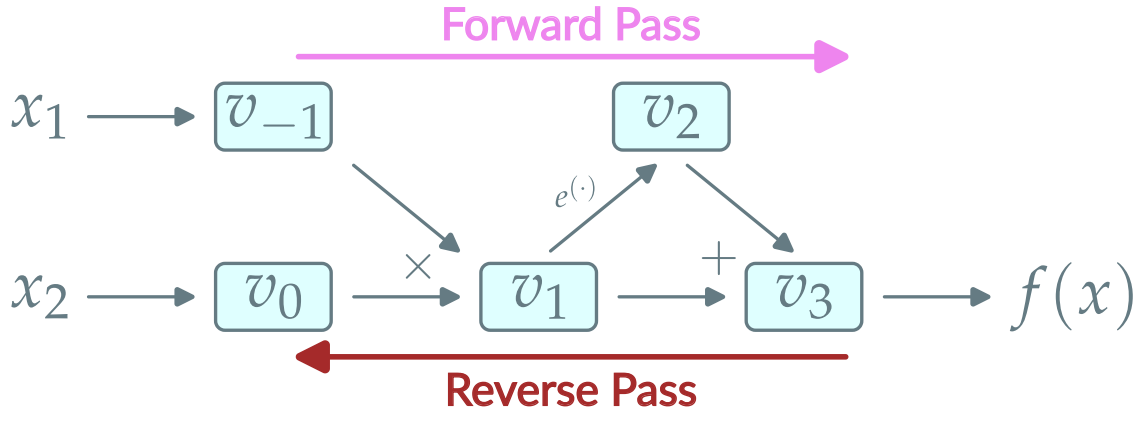
- We want to evaluate the gradient ∇f at the point $x = [1, 2]^\intercal$. Computing the gradient by hand is easy:

$$\nabla f = (1 + e^{x_1 x_2}) \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = (1 + e^2) \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

- Its computational graph is given by:



REVERSE MODE: EXAMPLE



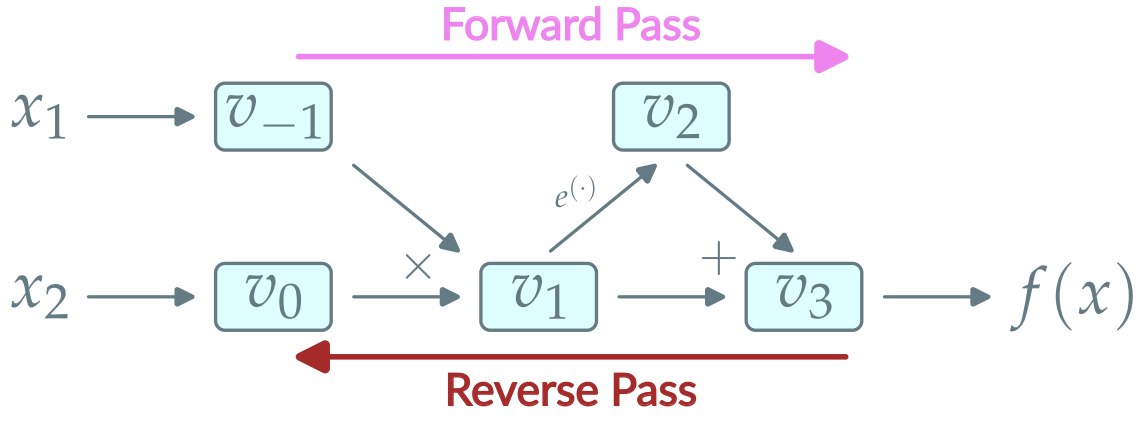
- ***Forward mode:*** *(what we did up to here, this is not forward pass)*

Forward primal trace	Forward tangent trace	Pass with $p = [1, 0]^\intercal$	Pass with $p = [0, 1]^\intercal$
$v_{-1} = x_1 = 1$	$D_p v_{-1} = p_1$	$D_p v_{-1} = 1$	$D_p v_{-1} = 0$
$v_0 = x_2 = 2$	$D_p v_0 = p_2$	$D_p v_0 = 0$	$D_p v_0 = 1$
$v_1 = v_{-1} v_0 = 2$	$D_p v_1 = v_0 D_p v_{-1} + v_{-1} D_p v_0$	$D_p v_1 = 2$	$D_p v_1 = 1$
$v_2 = e^{v_1} = e^2$	$D_p v_2 = e^{v_1} D_p v_1$	$D_p v_2 = 2e^2$	$D_p v_2 = e^2$
$v_3 = v_1 + v_2$	$D_p v_3 = D_p v_1 + D_p v_2$	$D_p v_3 = 2 + 2e^2$	$D_p v_3 = 1 + e^2$

independent variables | *dependent* variables

Note that we need $m = 2$ passes in ***forward mode*** to compute the gradient ∇f

REVERSE MODE: EXAMPLE



- ***Reverse mode:***

Forward Pass:

Intermediate	Partial Derivatives
$v_{-1} = x_1 = 1$	
$v_0 = x_2 = 2$	
$v_1 = v_{-1} v_0 = 2$	$\frac{\partial v_1}{\partial v_{-1}} = v_0 = 2$ $\frac{\partial v_1}{\partial v_0} = v_{-1} = 1$
$v_2 = e^{v1} = e^2$	$\frac{\partial v_2}{\partial v_1} = e^{v_1} = e^2$
$v_3 = v_1 + v_2 = 2 + e^2$	$\frac{\partial v_3}{\partial v_1} = 1$ $\frac{\partial v_3}{\partial v_2} = 1$

independent variables *dependent* variables

Reverse Pass:

Adjoint
$\bar{v}_{-1} = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = (1 + e^2) \cdot 2 = \frac{\partial f}{\partial x_1}$
$\bar{v}_0 = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_0} = \bar{v}_1 \frac{\partial v_1}{\partial v_0} = 1 + e^2 = \frac{\partial f}{\partial x_2}$
$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1} = \bar{v}_1 + \bar{v}_2 \frac{\partial v_2}{\partial v_1} = 1 + e^2$ (second child update)
$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \bar{v}_1 + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 1$ (first child)
$\bar{v}_2 = \bar{v}_2 + \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_2} = \bar{v}_2 + \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
$\bar{v}_3 = \frac{\partial f}{\partial v_3} = \frac{\partial v_3}{\partial v_3} = 1$

REVERSE MODE: EXAMPLE

- ***Reverse mode:***

Forward Pass: ***Reverse Pass:***

Intermediate	Partial Derivatives	Adjoint
$v_{-1} = x_1 = 1$		$\bar{v}_{-1} = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} = \bar{v}_1\frac{\partial v_1}{\partial v_{-1}} = (1+e^2)\cdot 2 = \frac{\partial f}{\partial x_1}$
$v_0 = x_2 = 2$		$\bar{v}_0 = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_0} = \bar{v}_1\frac{\partial v_1}{\partial v_0} = 1+e^2 = \frac{\partial f}{\partial x_2}$
$v_1 = v_{-1}v_0 = 2$	$\frac{\partial v_1}{\partial v_{-1}} = v_0 = 2$ $\frac{\partial v_1}{\partial v_0} = v_{-1} = 1$	$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1} = \bar{v}_1 + \bar{v}_2\frac{\partial v_2}{\partial v_1} = 1+e^2$ (second child update)
$v_2 = e^{v1} = e^2$	$\frac{\partial v_2}{\partial v_1} = e^{v_1} = e^2$	$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \bar{v}_1 + \bar{v}_3\frac{\partial v_3}{\partial v_1} = 1$ (first child)
$v_3 = v_1 + v_2 = 2 + e^2$	$\frac{\partial v_3}{\partial v_1} = 1$ $\frac{\partial v_3}{\partial v_2} = 1$	$\bar{v}_2 = \bar{v}_2 + \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_2} = \bar{v}_2 + \bar{v}_3\frac{\partial v_3}{\partial v_2} = 1$
		$\bar{v}_3 = \frac{\partial f}{\partial v_3} = \frac{\partial v_3}{\partial v_3} = 1$
independent variables	*dependent* variables	

We only need ***1 reverse mode pass*** to compute the gradient ∇f (forward + reverse pass is considered *one* reverse mode pass).
→ ***compare this to forward mode if*** $m \gg 1$***!***

REVERSE MODE: EXAMPLE

Observations:

- ***Forward mode*** computes the gradient with respect to the *independent* variables: $\nabla_x f$.
- ***Reverse mode*** computes the gradient with respect to the coordinates v: $\nabla_v f$. Because we have chosen $v_{j-m} = x_j$ for $j = 1, 2, \ldots, m$, the gradient $\nabla_x f$ *is a subset of* $\nabla_v f$*!*
- The ***computational cost*** of ***forward mode*** depends on the number of *independent* variables m.
- The computational cost of ***reverse mode*** is independent of m.
  This is why reverse mode (back-propagation) is the algorithm of choice in machine learning because m is usually very large.

REVERSE MODE: EXAMPLE

Summary:

- In machine learning, the objective function is a *scalar* function with possibly a very large number m of input arguments.
- The *gradient* of the objective function is needed to train the model. A popular and efficient algorithm for this task is called *back-propagation*, which is a special case of reverse mode AD. Special in the sense that the function is scalar and it represents an error between the computed output (hence we must compute the primal trace v_j in the forward pass too) and an expected output.
- If there are many more outputs $n \gg m$ ***forward mode AD is more efficient***.
- If there are many more inputs $m \gg n$ ***reverse mode AD is more efficient***.

AUTOMATIC DIFFERENTIATION: EXERCISE

Given the function $f(x) : \mathbb{R}^5 \mapsto \mathbb{R}$ with

$$f(x) = x_1 x_2 x_3 x_4 x_5,$$

compute the gradient ∇f evaluated at the point $x = [2, 1, 1, 1, 1]^\intercal$.

1. Draw the computational graph.
2. Compute the gradient using forward mode. Note: you need $m = 5$ passes with different seed vectors. Write your solution in a evaluation table similar to what we did earlier.
3. Compute the gradient using reverse mode. Write your results in another evaluation table (with possibly fewer columns than forward mode above).
4. For both, forward and reverse mode, calculate the number of arithmetic operations (addition, subtraction, multiplication, division).

EXAMPLES FOR EXTENSIONS AND APPLICATIONS

- Up to this point we have discussed the math behind automatic differentiation (chain rule and splitting up a function (evaluation) into elementary parts resulting in a computational graph).
- Many extensions and applications exist for an automatic differentiation algorithm.
- A few will be outlined next to give you some ideas for your project.

EXAMPLES FOR EXTENSIONS

- Higher order and mixed derivatives:
  - Laplacian operator $\Delta f = \nabla \cdot (\nabla f)$
  - Mixed derivatives $\frac{\partial^2 f}{\partial x_1 \partial x_2}$
  - Hessian matrix which is the Jacobian of the gradient of a scalar function f, that is $\nabla(\nabla f)$
- Computational optimizations:
  - Efficient graph storage and data structure design/traversal
  - Hybrid graph storage model: writing parts of a large graph to (slow) disks and keeping "hot" graph parts in memory
- Combining forward mode and reverse mode (mixed mode)
- Exploiting sparsity in the Jacobian and/or Hessian matrices (graph coloring)
- Non-differentiable functions

EXAMPLES FOR APPLICATIONS

There are many applications of AD, below are just a few.
See also autodiff.org

- Numerical solution of Ordinary Differential Equations (ODEs):
  - integration of *stiff* ODE systems
  - Newton's method for the solution of non-linear systems of equations (requires Jacobian-vector products)
- Optimization:
  - Optimize an objective function (also know as loss or cost function)
  - These techniques require the gradient of the loss function with respect to its parameters
- Solution of linear systems:
  - Iterative methods are powerful algorithms for solving linear systems
  - Some iterative methods require information obtained through derivatives, for example, steepest gradient descent, conjugate gradient or biconjugate gradient methods.

RECAP

Automatic Differentiation: forward mode and reverse mode

- Implementation approaches for automatic differentiation
- Operator overloading
- Reverse mode AD
- Examples for application and project extensions

Further reading:

- P. H.W. Hoffmann, *A Hitchhiker's Guide to Automatic Differentiation*, Springer 2015, doi:10.1007/s11075-015-0067-6 (You can access this paper through the Harvard network or find it in the class repository)
- Griewank, A. and Walther, A., *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM 2008, Vol. 105
- Nocedal, J. and Wright, S., *Numerical Optimization*, Springer 2006, 2nd Edition
- Baydin, A., Pearlmutter, B., Radul, A. and Siskind, J., *Automatic Differentiation in Machine Learning: A Survey*, Journal of Machine Learning 2017