Automatic Differentiation

Mark van der Wilk

Department of Computing
Imperial College London

@markvanderwilk
m.vdwilk@imperial.ac.uk

October 17, 2022

Coursework

- Check out the repo for the notebook to get started.
- We recommend to run this on the **lab machines**.
- Soon, a GitLab repo will be created for you.
- Submit code to CATE for grading in LabTS.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.
- Can always work derivatives out with index notation.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.
- Can always work derivatives out with index notation.
  - All functions of vectors/matrices/arrays are just multivariate functions, just with reshaped inputs.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.
- Can always work derivatives out with index notation.
  - All functions of vectors/matrices/arrays are just multivariate functions, just with reshaped inputs.
  - Regardless of shape, e.g. deriv of matrix by vector, matrix by matrix, or weirder ones!

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.
- Can always work derivatives out with index notation.
  - All functions of vectors/matrices/arrays are just multivariate functions, just with reshaped inputs.
  - Regardless of shape, e.g. deriv of matrix by vector, matrix by matrix, or weirder ones!
- Vector chain rule leads to matrix multiplication if we only take derivative of vector w.r.t. vector.

Multivariate Differentiation Summary

- Directional derivatives motivate partial derivatives.
- Conditions for minimum: $\mathrm{d}f/\mathrm{d}\mathbf{x} = 0$, $\lambda_i(\boldsymbol{H}) > 0, \forall i$.
- Can always work derivatives out with index notation.
  - All functions of vectors/matrices/arrays are just multivariate functions, just with reshaped inputs.
  - Regardless of shape, e.g. deriv of matrix by vector, matrix by matrix, or weirder ones!
- Vector chain rule leads to matrix multiplication if we only take derivative of vector w.r.t. vector.
- We can still use chain rule notation when dealing with matrix derivatives, but we need to separately keep track what summation is meant with this.

Overview

Introduction

Forward Mode Automatic Differentiation

Reverse Mode Automatic Differentiation

Backpropagation in Neural Networks

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Automatic Differentiation

Today: Wishlist

In the last lectures, you learned how to differentiate anything, which is helpful for **optimisation**.

Today we will discuss a method with the following wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Automatic Differentiation

Will roughly be following the review article by Baydin et al. (2018).

Symbolic Differentiation

What we studied so far:

- Define a function $f(\mathbf{x})$, that we can evaluate for any $\mathbf{x}$

Symbolic Differentiation

What we studied so far:

- Define a function $f(\mathbf{x})$, that we can evaluate for any $\mathbf{x}$
- Find a new function $\mathrm{d}f/\mathrm{d}\mathbf{x}$, that we can evaluate for any $\mathbf{x}$

Symbolic Differentiation

What we studied so far:

- Define a function $f(\mathbf{x})$, that we can evaluate for any $\mathbf{x}$
- Find a new function $\mathrm{d}f/\mathrm{d}\mathbf{x}$, that we can evaluate for any $\mathbf{x}$
- Many ways to differentiate a function

Symbolic Differentiation

What we studied so far:

- Define a function $f(\mathbf{x})$, that we can evaluate for any $\mathbf{x}$
- Find a new function $\mathrm{d}f/\mathrm{d}\mathbf{x}$, that we can evaluate for any $\mathbf{x}$
- Many ways to differentiate a function
- Can be very inefficient, if done carelessly

Example: Inefficient Symbolic Differentiation

$$L(\theta) = f(\boldsymbol{K}(\boldsymbol{D}(\theta)))\,,$$

$$f : \mathbb{R}^{N\times N} \to \mathbb{R}\,, \qquad \boldsymbol{K} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}\,, \qquad \boldsymbol{D} : \mathbb{R}^{P} \to \mathbb{R}^{N\times N}\,.$$

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \underbrace{\frac{\partial L}{\partial \boldsymbol{K}}}_{1\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}}_{(N\times N)\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}}_{(N\times N)\times P}$$

Procedure: **1)** Compute each array. Computational cost?

Example: Inefficient Symbolic Differentiation

$$L(\theta) = f(\boldsymbol{K}(\boldsymbol{D}(\theta)))\,,$$

$$f : \mathbb{R}^{N\times N} \to \mathbb{R}\,, \qquad \boldsymbol{K} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}\,, \qquad \boldsymbol{D} : \mathbb{R}^{P} \to \mathbb{R}^{N\times N}\,.$$

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \underbrace{\frac{\partial L}{\partial \boldsymbol{K}}}_{1\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}}_{(N\times N)\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}}_{(N\times N)\times P}$$

Procedure: **1)** Compute each array. Computational cost? Scales with elements, so **at least** $N^2 + N^4 + N^2 P$.

2) Then we have two options:

- $\frac{\partial L}{\partial \boldsymbol{K}}\left(\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}\right)$: $N^4 P + N^2 P$

Example: Inefficient Symbolic Differentiation

$$L(\theta) = f(\boldsymbol{K}(\boldsymbol{D}(\theta)))\,,$$

$$f : \mathbb{R}^{N\times N} \to \mathbb{R}\,, \qquad \boldsymbol{K} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}\,, \qquad \boldsymbol{D} : \mathbb{R}^{P} \to \mathbb{R}^{N\times N}\,.$$

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \underbrace{\frac{\partial L}{\partial \boldsymbol{K}}}_{1\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}}_{(N\times N)\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}}_{(N\times N)\times P}$$

Procedure: **1)** Compute each array. Computational cost?
Scales with elements, so **at least** $N^2 + N^4 + N^2P$.

2) Then we have two options:

- $\frac{\partial L}{\partial \boldsymbol{K}}\left(\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}\right)$: $N^4P + N^2P$
- $\left(\frac{\partial L}{\partial \boldsymbol{K}}\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\right)\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}$: $N^4 + N^2P$

Example: Inefficient Symbolic Differentiation

$$L(\theta) = f(\boldsymbol{K}(\boldsymbol{D}(\theta)))\,,$$

$$f : \mathbb{R}^{N\times N} \to \mathbb{R}\,, \qquad \boldsymbol{K} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}\,, \qquad \boldsymbol{D} : \mathbb{R}^{P} \to \mathbb{R}^{N\times N}\,.$$

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \underbrace{\frac{\partial L}{\partial \boldsymbol{K}}}_{1\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}}_{(N\times N)\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}}_{(N\times N)\times P}$$

Procedure: **1)** Compute each array. Computational cost?
Scales with elements, so **at least** $N^2 + N^4 + N^2 P$.

2) Then we have two options:

- $\frac{\partial L}{\partial \boldsymbol{K}}\left(\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}\right)$: $N^4 P + N^2 P$
- $\left(\frac{\partial L}{\partial \boldsymbol{K}}\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\right)\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}$: $N^4 + N^2 P$

Problems:

- Cannot take advantage of structure (e.g. zero elements)

Example: Inefficient Symbolic Differentiation

$$L(\theta) = f(\boldsymbol{K}(\boldsymbol{D}(\theta)))\,,$$

$$f : \mathbb{R}^{N\times N} \to \mathbb{R}\,, \qquad \boldsymbol{K} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}\,, \qquad \boldsymbol{D} : \mathbb{R}^{P} \to \mathbb{R}^{N\times N}\,.$$

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = \underbrace{\frac{\partial L}{\partial \boldsymbol{K}}}_{1\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}}_{(N\times N)\times(N\times N)} \quad \underbrace{\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}}_{(N\times N)\times P}$$

Procedure: **1)** Compute each array. Computational cost? Scales with elements, so **at least** $N^2 + N^4 + N^2 P$.

2) Then we have two options:

- $\frac{\partial L}{\partial \boldsymbol{K}}\left(\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}\right)$: $N^4 P + N^2 P$
- $\left(\frac{\partial L}{\partial \boldsymbol{K}}\frac{\partial \boldsymbol{K}}{\partial \boldsymbol{D}}\right)\frac{\partial \boldsymbol{D}}{\partial \boldsymbol{\theta}}$: $N^4 + N^2 P$

Problems:

- Cannot take advantage of structure (e.g. zero elements)
- Not clear which order to compute in to be efficient.

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the “best” computational complexity.

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Autodiff provides:

- a data structure for specifying mathematical functions,

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Autodiff provides:

- a data structure for specifying mathematical functions,
- different methods for automatically finding gradients,

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Autodiff provides:

- a data structure for specifying mathematical functions,
- different methods for automatically finding gradients,
- provides guarantees of the computational complexity,

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the "best" computational complexity.

Autodiff provides:

- a data structure for specifying mathematical functions,
- different methods for automatically finding gradients,
- provides guarantees of the computational complexity,
- rules of thumb for when to use each method.

What autodiff provides

Our wishlist:

- specify how to compute an objective function *in code*,
- automatically get the gradient vector for a particular parameter,
- understand the computational complexity of doing so,
- ideally have the “best” computational complexity.

Autodiff provides:

- a data structure for specifying mathematical functions,
- different methods for automatically finding gradients,
- provides guarantees of the computational complexity,
- rules of thumb for when to use each method.

Although unfortunately finding the optimal gradient in general (optimal jacobian accumulation problem) is NP-complete :(

Today: Answers / Topics

- Symbolic differentiation, and its problem
- Computational graphs (describing computation)
- Forward mode autodiff
- Reverse mode autodiff (backpropagation)
- Computational considerations

Computational Graphs

- A graph is a **data structure** that can be used to represent a computation.
- Each intermediate result is a node.
- Edges indicate a dependency in a computation.

Computational Graphs

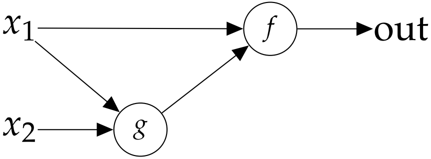
- A graph is a **data structure** that can be used to represent a computation.
- Each intermediate result is a node.
- Edges indicate a dependency in a computation.

Example: $f(x_1, x_2) = f(x_1, g(x_1, x_2))$

Computational Graphs

- A graph is a **data structure** that can be used to represent a computation.
- Each intermediate result is a node.
- Edges indicate a dependency in a computation.

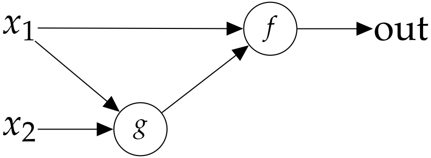
Example: $f(x_1, x_2) = f(x_1, g(x_1, x_2))$



Computational Graphs

- A graph is a **data structure** that can be used to represent a computation.
- Each intermediate result is a node.
- Edges indicate a dependency in a computation.

Example: $f(x_1, x_2) = f(x_1, g(x_1, x_2))$



- To find the output, **traverse** the graph from the inputs.
- Gradient computation traverses the graph in various ways.

Overview

Introduction

Forward Mode Automatic Differentiation

Reverse Mode Automatic Differentiation

Backpropagation in Neural Networks

Forward mode Autodiff

We want to compute the gradient w.r.t. x_i.

- Initialise each input node j with $\frac{\partial x_j}{\partial x_i}$.

Forward mode Autodiff

We want to compute the gradient w.r.t. x_i.

- Initialise each input node j with $\frac{\partial x_j}{\partial x_i}$.
- Traverse the nodes of the graph, indexed by j, in the same way as computing the output.

Forward mode Autodiff

We want to compute the gradient w.r.t. x_i.

- Initialise each input node j with $\frac{\partial x_j}{\partial x_i}$.
- Traverse the nodes of the graph, indexed by j, in the same way as computing the output.
  - For the input value $\mathbf{x} = \mathbf{a}$, compute the numerical value of

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{1}$$

Forward mode Autodiff

We want to compute the gradient w.r.t. x_i.

- Initialise each input node j with $\frac{\partial x_j}{\partial x_i}$.
- Traverse the nodes of the graph, indexed by j, in the same way as computing the output.
    - For the input value $\mathbf{x} = \mathbf{a}$, compute the numerical value of

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{1}$$

- We end up with $\partial \text{out} / \partial x_i$.

Forward mode Autodiff

We want to compute the gradient w.r.t. x_i.

- Initialise each input node j with $\frac{\partial x_j}{\partial x_i}$.
- Traverse the nodes of the graph, indexed by j, in the same way as computing the output.
  - For the input value $\mathbf{x} = \mathbf{a}$, compute the numerical value of

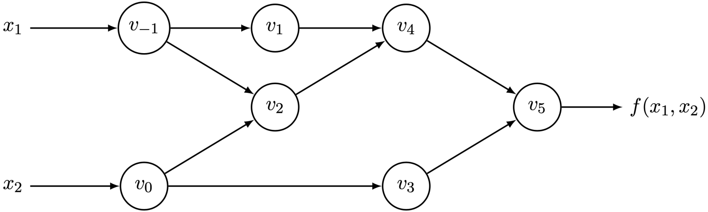
$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{1}$$

- We end up with $\partial \text{out} / \partial x_i$.

Repeat for all i to find all gradients.

Forward mode Autodiff: Example

Computational graph for $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$



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	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\mathbf{= 5.5}$

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \dots)$.

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

You can differentiate anything you implement!

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \dots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!
- Implementation can take advantage of any sparsity in $\partial v_j / \partial v_k$.

Things to notice

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (1).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!
- Implementation can take advantage of any sparsity in $\partial v_j / \partial v_k$.

The procedure above is efficient for vector inputs too!

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \mathrm{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- No memory overhead.

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \mathrm{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- No memory overhead.
- **However**,

Forward mode: Computational complexity

Remember the key equation:

$$\frac{\partial v_j}{\partial x_i} = \sum_{k \in \text{inputs}(i)} \frac{\partial v_j}{\partial v_k} \frac{\partial v_k}{\partial x_i} \tag{2}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- No memory overhead.
- **However**, cost scales linearly with the number of gradients!

Fun exercise

Prove the product rule using forward mode autodiff.

Board

Overview

Introduction

Forward Mode Automatic Differentiation

Reverse Mode Automatic Differentiation

Backpropagation in Neural Networks

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.
- Initialise the output node with $\partial \text{out}/\partial v_{\text{final}} = 1$.

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.
- Initialise the output node with $\partial \text{out} / \partial v_{\text{final}} = 1$.
- Traverse the nodes of the graph, indexed by j, *backwards*, starting from the output.

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.
- Initialise the output node with $\partial \text{out}/\partial v_{\text{final}} = 1$.
- Traverse the nodes of the graph, indexed by j, *backwards*, starting from the output.
    - At the numerically computed values of v_j, compute the numerical value of

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{3}$$

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.
- Initialise the output node with $\partial \text{out}/\partial v_{\text{final}} = 1$.
- Traverse the nodes of the graph, indexed by j, *backwards*, starting from the output.
  - At the numerically computed values of v_j, compute the numerical value of

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{3}$$

- We end up with $\partial \text{out}/\partial x_i$.

Reverse mode autodiff

We want to compute the gradient w.r.t. x_i.

- Traverse the graph to compute the value of all the nodes, *and store them*.
- Initialise the output node with $\partial \text{out}/\partial v_{\text{final}} = 1$.
- Traverse the nodes of the graph, indexed by j, *backwards*, starting from the output.
    - At the numerically computed values of v_j, compute the numerical value of

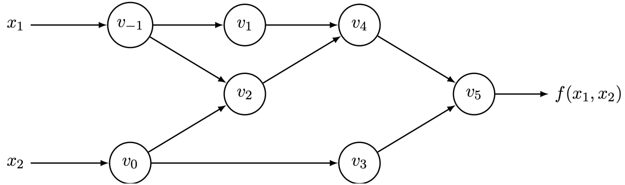
$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{3}$$

- We end up with $\partial \text{out}/\partial x_i$.

Repeat for all i to find all gradients.

Reverse mode Autodiff: Example

Computational graph for $f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$



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

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (3).

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (3).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (3).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

You can differentiate anything you implement!

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (3).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \dots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k\in\text{inputs(j)}})$, you need to implement the equation eq. (3).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \dots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!
- Implementation can take advantage of any sparsity in $\partial v_j / \partial v_k$.

Things to notice (again)

- For each intermediate function $v_j(\{v_k\}_{k \in \text{inputs(j)}})$, you need to implement the equation eq. (3).
- **All** algorithms are composed of simple functions $(+, *, \text{pow}, \ldots)$.

You can differentiate anything you implement!

- v_j and x_i can be vectors: the vector chain rule holds!
- Implementation can take advantage of any sparsity in $\partial v_j / \partial v_k$.

The procedure above is efficient for vector inputs too!

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \mathrm{out}}{\partial v_j} = \sum_{k \in \mathrm{outputs}(j)} \frac{\partial \mathrm{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- Need to store **all** intermediate results (or recompute).

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- Need to store **all** intermediate results (or recompute).
- **However**,

Reverse mode: Computational complexity

Remember the key equation:

$$\frac{\partial \text{out}}{\partial v_j} = \sum_{k \in \text{outputs}(j)} \frac{\partial \text{out}}{\partial v_k} \frac{\partial v_k}{\partial v_j} \tag{4}$$

Consider the scalar case (remember, vector funcs are a special case):

- The derivative of all elementary functions has the same cost as the computation itself. E.g. $+, \times, \sin, \text{pow}, \ldots$.
- Only a constant difference in cost between the deriv and func.
- $\implies$ same computational complexity.
- Need to store **all** intermediate results (or recompute).
- **However**, cost of computing all derivatives is same as fwd pass.

Overview

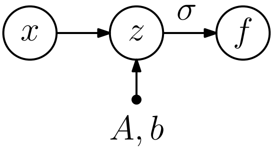
Introduction

Forward Mode Automatic Differentiation

Reverse Mode Automatic Differentiation

Backpropagation in Neural Networks

Gradients of a Single-Layer Neural Network



$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z}\in\mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M\times N}, \boldsymbol{b} \in \mathbb{R}^M$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z}\in\mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M\times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} =$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} =$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z}\in\mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M\times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M\times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M\times M} \in \mathbb{R}^{M\times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} =$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z}\in\mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M\times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M\times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M\times M} \in \mathbb{R}^{M\times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M\times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M\times (M\times N)} \in \mathbb{R}^{M\times (M\times N)}$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z} \in \mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M \times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M \times M} \in \mathbb{R}^{M \times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M \times (M \times N)} \in \mathbb{R}^{M \times (M \times N)}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\mathrm{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in \mathbb{R}^{M \times M}}$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z} \in \mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M \times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M \times M} \in \mathbb{R}^{M \times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M \times (M \times N)} \in \mathbb{R}^{M \times (M \times N)}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\mathrm{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in \mathbb{R}^{M \times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \underbrace{\boldsymbol{I}}_{\in \mathbb{R}^{M \times M}}$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z}\in\mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M\times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M\times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M\times M} \in \mathbb{R}^{M\times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M\times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M\times(M\times N)} \in \mathbb{R}^{M\times(M\times N)}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\operatorname{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in\mathbb{R}^{M\times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \underbrace{\boldsymbol{I}}_{\in\mathbb{R}^{M\times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}} = \underbrace{\begin{bmatrix} \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top \end{bmatrix}}_{\in\mathbb{R}^{M\times(M\times N)}}$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}}_{=:\boldsymbol{z} \in \mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M \times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M \times M} \in \mathbb{R}^{M \times M}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}}[i, j] = \sum_{l=1}^{M} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}[i, l] \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}[l, j]$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M \times (M \times N)} \in \mathbb{R}^{M \times (M \times N)}$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\operatorname{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in \mathbb{R}^{M \times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \underbrace{\boldsymbol{I}}_{\in \mathbb{R}^{M \times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}} = \underbrace{\begin{bmatrix} \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top \end{bmatrix}}_{\in \mathbb{R}^{M \times (M \times N)}}$$

Gradients of a Single-Layer Neural Network

$$\boldsymbol{f} = \tanh(\underbrace{\boldsymbol{Ax} + \boldsymbol{b}}_{=:\boldsymbol{z} \in \mathbb{R}^M}) \in \mathbb{R}^M, \quad \boldsymbol{x} \in \mathbb{R}^N, \boldsymbol{A} \in \mathbb{R}^{M \times N}, \boldsymbol{b} \in \mathbb{R}^M$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}_{M \times M} \in \mathbb{R}^{M \times M} \qquad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}}[i,j] = \sum_{l=1}^{M} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}[i,l] \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}[l,j]$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}} = \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}}_{M \times M} \underbrace{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}}_{M \times (M \times N)} \in \mathbb{R}^{M \times (M \times N)} \qquad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{A}}[i,j,k] = \sum_{l=1}^{M} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}[i,l] \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}[l,j,k]$$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\operatorname{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in \mathbb{R}^{M \times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \underbrace{\boldsymbol{I}}_{\in \mathbb{R}^{M \times M}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}} = \underbrace{\begin{bmatrix} \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top \end{bmatrix}}_{\in \mathbb{R}^{M \times (M \times N)}}$$

Putting Things Together

- Inputs $\boldsymbol{x} \in \mathbb{R}^N$

Putting Things Together

- Inputs $\boldsymbol{x} \in \mathbb{R}^N$
- Observed outputs $\boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{\epsilon} \in \mathbb{R}^M, \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$

Putting Things Together

- Inputs $\boldsymbol{x} \in \mathbb{R}^N$
- Observed outputs $\boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{\epsilon} \in \mathbb{R}^M, \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$
- Train single-layer neural network with

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \tanh(\boldsymbol{z}(\mathbf{x})) \in \mathbb{R}^M, \quad \boldsymbol{z} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^M, \quad \boldsymbol{\theta} = \{\boldsymbol{A}, \boldsymbol{b}\}$$

Putting Things Together

- Inputs $\boldsymbol{x} \in \mathbb{R}^N$
- Observed outputs $\boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) + \boldsymbol{\epsilon} \in \mathbb{R}^M, \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$
- Train single-layer neural network with

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \tanh(\boldsymbol{z}(\mathbf{x})) \in \mathbb{R}^M, \quad \boldsymbol{z} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^M, \quad \boldsymbol{\theta} = \{\boldsymbol{A}, \boldsymbol{b}\}$$

- Find $\boldsymbol{A}, \boldsymbol{b}$, such that the squared loss

$$L(\boldsymbol{\theta}) = \tfrac{1}{2}\|\boldsymbol{e}\|^2 \in \mathbb{R}, \quad \boldsymbol{e} = \boldsymbol{y} - \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{z}) \in \mathbb{R}^M$$

is minimized

Putting Things Together

Partial derivatives:

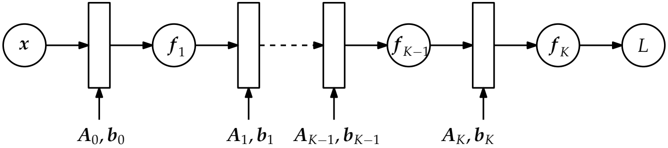
$$\frac{\partial L}{\partial \boldsymbol{A}} = \frac{\partial L}{\partial \boldsymbol{e}} \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}}$$
$$\frac{\partial L}{\partial \boldsymbol{b}} = \frac{\partial L}{\partial \boldsymbol{e}} \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{f}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$\frac{\partial L}{\partial \boldsymbol{e}} = \underbrace{\boldsymbol{e}^\top}_{\in \mathbb{R}^{1\times M}} \qquad \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{f}} = \underbrace{-\boldsymbol{I}}_{\in \mathbb{R}^{M\times M}} \qquad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}} = \underbrace{\mathrm{diag}(1 - \tanh^2(\boldsymbol{z}))}_{\in \mathbb{R}^{M\times M}}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{A}} = \underbrace{\begin{bmatrix} \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top & \cdot & \boldsymbol{0}^\top \\ \cdot & & \cdot & & \cdot \\ \boldsymbol{0}^\top & \cdot & \boldsymbol{0}^\top & \cdot & \boldsymbol{x}^\top \end{bmatrix}}_{\in \mathbb{R}^{M\times (M\times N)}} \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \underbrace{\boldsymbol{I}}_{\in \mathbb{R}^{M\times M}}$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{5}$$



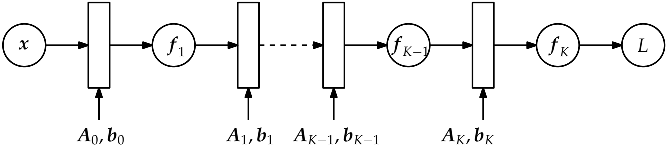
- Inputs $\boldsymbol{x}$, observed outputs $\boldsymbol{y}$
- Train multi-layer neural network with

$$\boldsymbol{f}_0 = \boldsymbol{x}$$

$$\boldsymbol{f}_i = \sigma_i(\boldsymbol{A}_{i-1}\boldsymbol{f}_{i-1} + \boldsymbol{b}_{i-1}), \quad i = 1, \ldots, K$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{5}$$



- Inputs $\boldsymbol{x}$, observed outputs $\boldsymbol{y}$
- Train multi-layer neural network with

$$\boldsymbol{f}_0 = \boldsymbol{x}$$
$$\boldsymbol{f}_i = \sigma_i(\boldsymbol{A}_{i-1}\boldsymbol{f}_{i-1} + \boldsymbol{b}_{i-1}), \quad i = 1, \ldots, K$$

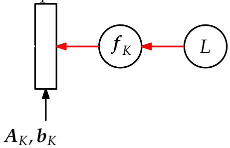
- Find $\boldsymbol{A}_j, \boldsymbol{b}_j$ for $j = 0, \ldots, K-1$, such that the squared loss

$$L(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{f}_{K,\boldsymbol{\theta}}(\boldsymbol{x})\|^2$$

is minimized, where $\boldsymbol{\theta} = \{\boldsymbol{A}_j, \boldsymbol{b}_j\}, \quad j = 0, \ldots, K-1$

Gradients of a Multi-Layer Neural Network

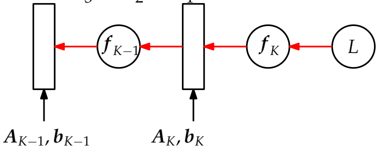
$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{6}$$



$$\frac{\partial L}{\partial \boldsymbol{\theta}_K} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_K}$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{6}$$

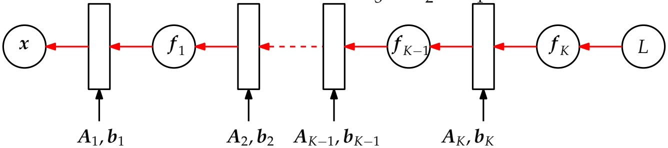


$$\frac{\partial L}{\partial \boldsymbol{\theta}_K} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_K}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \boxed{\frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-1}}}$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{6}$$



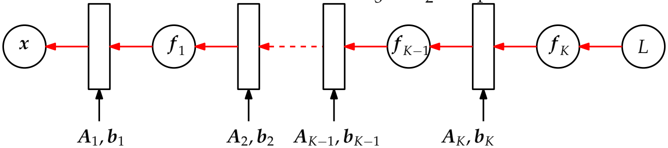
$$\frac{\partial L}{\partial \boldsymbol{\theta}_K} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_K}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \boxed{\frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-1}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \boxed{\frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{f}_{K-2}} \frac{\partial \boldsymbol{f}_{K-2}}{\partial \boldsymbol{\theta}_{K-2}}}$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{6}$$



$$\frac{\partial L}{\partial \boldsymbol{\theta}_K} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_K}$$

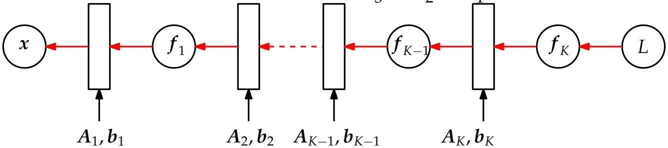
$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \boxed{\frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-1}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \boxed{\frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{f}_{K-2}} \frac{\partial \boldsymbol{f}_{K-2}}{\partial \boldsymbol{\theta}_{K-2}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \cdots \boxed{\frac{\partial \boldsymbol{f}_{i+1}}{\partial \boldsymbol{f}_i} \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{\theta}_i}}$$

Gradients of a Multi-Layer Neural Network

$$L(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3) = ||\mathbf{y} - \boldsymbol{f}_{\boldsymbol{\theta}_3}(\boldsymbol{f}_{\boldsymbol{\theta}_2}(\boldsymbol{f}_{\boldsymbol{\theta}_1}(\mathbf{x})))||^2 \tag{6}$$



$$\frac{\partial L}{\partial \boldsymbol{\theta}_K} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{\theta}_K}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \boxed{\frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-1}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \boxed{\frac{\partial \boldsymbol{f}_{K-1}}{\partial \boldsymbol{f}_{K-2}} \frac{\partial \boldsymbol{f}_{K-2}}{\partial \boldsymbol{\theta}_{K-2}}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \boldsymbol{f}_K} \frac{\partial \boldsymbol{f}_K}{\partial \boldsymbol{f}_{K-1}} \cdots \boxed{\frac{\partial \boldsymbol{f}_{i+1}}{\partial \boldsymbol{f}_i} \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{\theta}_i}}$$

- Intermediate derivatives are stored during the forward pass

Summary: Differentiation

- Computational graphs
- Flavours of automatic differentiation
- Computational cost analysis of automatic differentiation
- Application: Backpropagation in NNs

Summary: Differentiation

- Computational graphs
- Flavours of automatic differentiation
- Computational cost analysis of automatic differentiation
- Application: Backpropagation in NNs

If you have a spare 1.5 hours, and want to see how **minimal** an implementation of this can be, I **highly** recommend Conal Elliott's talk on *The Simple Essence of Automatic Differentiation*: https://www.youtube.com/watch?v=ne99laPUxN4

References I

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. **Journal of Machine Learning Research**, 18(153):1–43, 2018. URL `http://jmlr.org/papers/v18/17-468.html`.