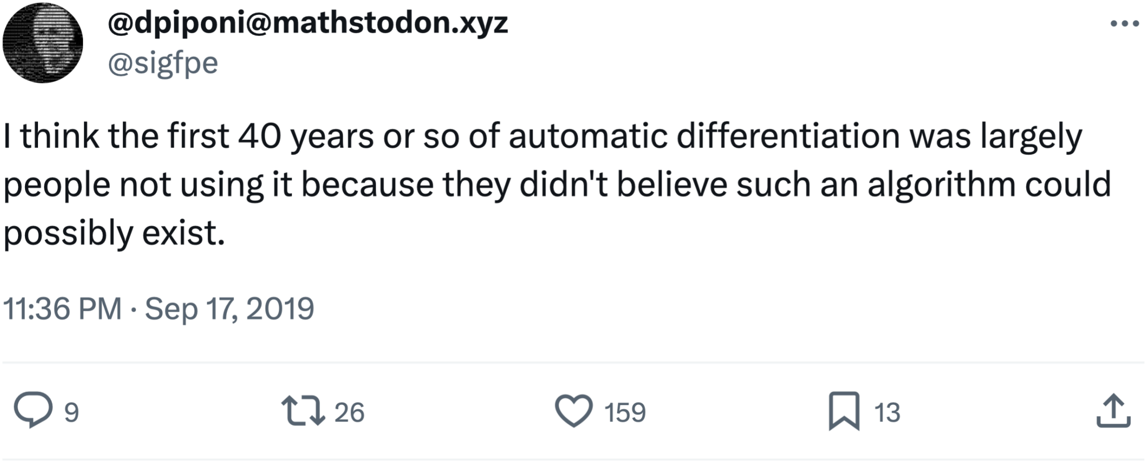
Automatic differentiation.

Daniil Merkulov

Optimization for ML. Faculty of Computer Science. HSE University


@dpiponi@mathstodon.xyz
@sigfpe

I think the first 40 years or so of automatic differentiation was largely people not using it because they didn't believe such an algorithm could possibly exist.

11:36 PM · Sep 17, 2019

9 26 159 13

Figure 1: When you got the idea

Figure 2: This is not autograd

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

- Such problems typically arise in machine learning, when you need to find optimal hyperparameters w of an ML model (i.e. train a neural network).

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

- Such problems typically arise in machine learning, when you need to find optimal hyperparameters w of an ML model (i.e. train a neural network).
- You may use a lot of algorithms to approach this problem, but given the modern size of the problem, where d could be dozens of billions it is very challenging to solve this problem without information about the gradients using zero-order optimization algorithms.

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

- Such problems typically arise in machine learning, when you need to find optimal hyperparameters w of an ML model (i.e. train a neural network).
- You may use a lot of algorithms to approach this problem, but given the modern size of the problem, where d could be dozens of billions it is very challenging to solve this problem without information about the gradients using zero-order optimization algorithms.
- That is why it would be beneficial to be able to calculate the gradient vector $\nabla_w L = \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_d} \right)^T$.

Problem

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

- Such problems typically arise in machine learning, when you need to find optimal hyperparameters w of an ML model (i.e. train a neural network).
- You may use a lot of algorithms to approach this problem, but given the modern size of the problem, where d could be dozens of billions it is very challenging to solve this problem without information about the gradients using zero-order optimization algorithms.
- That is why it would be beneficial to be able to calculate the gradient vector $\nabla_w L = \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_d} \right)^T$.
- Typically, first-order methods perform much better in huge-scale optimization, while second-order methods require too much memory.

Example: multidimensional scaling

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d, \; i = 1, \ldots, N$.

Example: multidimensional scaling

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d, \; i = 1, \ldots, N$.

$$L(W) = \sum_{i,j=1}^{N} \left(\|W_i - W_j\|_2^2 - D_{i,j} \right)^2 \rightarrow \min_{W \in \mathbb{R}^{N \times d}}$$

Example: multidimensional scaling

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d$, $i = 1, \ldots, N$.

$$L(W) = \sum_{i,j=1}^{N} \left(\|W_i - W_j\|_2^2 - D_{i,j} \right)^2 \to \min_{W \in \mathbb{R}^{N \times d}}$$

Link to a nice visualization ♣, where one can see, that gradient-free methods handle this problem much slower, especially in higher dimensions.

Question

Is it somehow connected with PCA?

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

with the Gradient Descent (GD) algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w L(w_k)$$

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

with the Gradient Descent (GD) algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w L(w_k)$$

Is it possible to replace $\nabla_w L(w_k)$ using only zero-order information?

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

with the Gradient Descent (GD) algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w L(w_k)$$

Is it possible to replace $\nabla_w L(w_k)$ using only zero-order information?

Yes, but at a cost.

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

with the Gradient Descent (GD) algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w L(w_k)$$

Is it possible to replace $\nabla_w L(w_k)$ using only zero-order information?
Yes, but at a cost.
One can consider 2-point gradient estimator[a] G:

$$G = d\frac{L(w + \varepsilon v) - L(w - \varepsilon v)}{2\varepsilon} v,$$

where v is spherically symmetric.

[a] I suggest a nice presentation about gradient-free methods

Example: Gradient Descent without gradient

Suppose we need to solve the following problem:

$$L(w) \to \min_{w \in \mathbb{R}^d}$$

with the Gradient Descent (GD) algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w L(w_k)$$

Is it possible to replace $\nabla_w L(w_k)$ using only zero-order information?

Yes, but at a cost.

One can consider 2-point gradient estimator[a] G:

$$G = d\frac{L(w + \varepsilon v) - L(w - \varepsilon v)}{2\varepsilon} v,$$

where v is spherically symmetric.

[a] I suggest a nice presentation about gradient-free methods

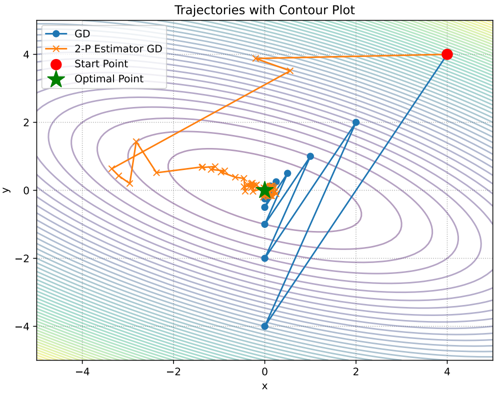


Figure 3: "Illustration of two-point estimator of Gradient Descent"

Example: Gradient Descent without gradient

$$w_{k+1} = w_k - \alpha_k G$$

Example: Gradient Descent without gradient

$$w_{k+1} = w_k - \alpha_k G$$

One can also consider the idea of finite differences:

$$G = \sum_{i=1}^{d} \frac{L(w + \varepsilon e_i) - L(w - \varepsilon e_i)}{2\varepsilon} e_i$$

Open In Colab ♣

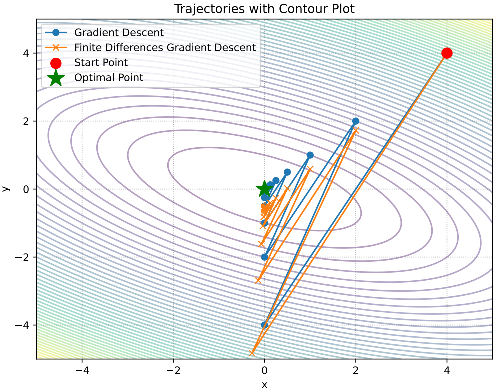


Figure 4: "Illustration of finite differences estimator of Gradient Descent"

The curse of dimensionality for zero-order methods

$$
\min_{x \in \mathbb{R}^n} f(x)
$$

The curse of dimensionality for zero-order methods

$$
\min_{x \in \mathbb{R}^n} f(x)
$$

GD: $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$ Zero order GD: $x_{k+1} = x_k - \alpha_k G,$

where G is a 2-point or multi-point estimator of the gradient.

The curse of dimensionality for zero-order methods

$$\min_{x \in \mathbb{R}^n} f(x)$$

GD: $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$ Zero order GD: $x_{k+1} = x_k - \alpha_k G,$

where G is a 2-point or multi-point estimator of the gradient.

	$f(x)$ - smooth	$f(x)$ - smooth and convex	$f(x)$ - smooth and strongly convex
GD	$\|\nabla f(x_k)\|^2 \approx \mathcal{O}\left(\frac{1}{k}\right)$	$f(x_k) - f^* \approx \mathcal{O}\left(\frac{1}{k}\right)$	$\|x_k - x^*\|^2 \approx \mathcal{O}\left(\left(1 - \frac{\mu}{L}\right)^k\right)$
Zero order GD	$\|\nabla f(x_k)\|^2 \approx \mathcal{O}\left(\frac{n}{k}\right)$	$f(x_k) - f^* \approx \mathcal{O}\left(\frac{n}{k}\right)$	$\|x_k - x^*\|^2 \approx \mathcal{O}\left(\left(1 - \frac{\mu}{nL}\right)^k\right)$

Finite differences

The naive approach to get approximate values of gradients is **Finite differences** approach. For each coordinate, one can calculate the partial derivative approximation:

$$\frac{\partial L}{\partial w_k}(w) \approx \frac{L(w + \varepsilon e_k) - L(w)}{\varepsilon}, \quad e_k = (0, \ldots, \underset{k}{1}, \ldots, 0)$$

[1] Linnainmaa S. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

Finite differences

The naive approach to get approximate values of gradients is **Finite differences** approach. For each coordinate, one can calculate the partial derivative approximation:

$$\frac{\partial L}{\partial w_k}(w) \approx \frac{L(w + \varepsilon e_k) - L(w)}{\varepsilon}, \quad e_k = (0, \dots, \underset{k}{1}, \dots, 0)$$

Question

If the time needed for one calculation of $L(w)$ is T, what is the time needed for calculating $\nabla_w L$ with this approach?

Finite differences

The naive approach to get approximate values of gradients is **Finite differences** approach. For each coordinate, one can calculate the partial derivative approximation:

$$\frac{\partial L}{\partial w_k}(w) \approx \frac{L(w + \varepsilon e_k) - L(w)}{\varepsilon}, \quad e_k = (0, \ldots, \underset{k}{1}, \ldots, 0)$$

Question

If the time needed for one calculation of $L(w)$ is T, what is the time needed for calculating $\nabla_w L$ with this approach?
Answer $2dT$, which is extremely long for the huge scale optimization. Moreover, this exact scheme is unstable, which means that you will have to choose between accuracy and stability.

Finite differences

The naive approach to get approximate values of gradients is **Finite differences** approach. For each coordinate, one can calculate the partial derivative approximation:

$$\frac{\partial L}{\partial w_k}(w) \approx \frac{L(w + \varepsilon e_k) - L(w)}{\varepsilon}, \quad e_k = (0, \ldots, \underset{k}{1}, \ldots, 0)$$

Question

If the time needed for one calculation of $L(w)$ is T, what is the time needed for calculating $\nabla_w L$ with this approach?

Answer $2dT$, which is extremely long for the huge scale optimization. Moreover, this exact scheme is unstable, which means that you will have to choose between accuracy and stability.

Theorem

There is an algorithm to compute $\nabla_w L$ in $\mathcal{O}(T)$ operations. [1]

[1] Linnainmaa S. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

Forward mode automatic differentiation

To dive deep into the idea of automatic differentiation we will consider a simple function for calculating derivatives:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

Forward mode automatic differentiation

To dive deep into the idea of automatic differentiation we will consider a simple function for calculating derivatives:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

Let's draw a *computational graph* of this function:

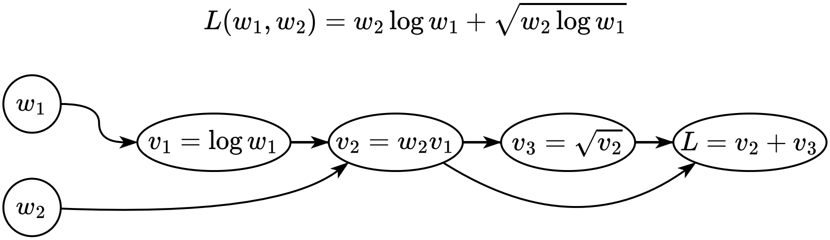


Figure 5: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Forward mode automatic differentiation

To dive deep into the idea of automatic differentiation we will consider a simple function for calculating derivatives:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

Let's draw a *computational graph* of this function:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

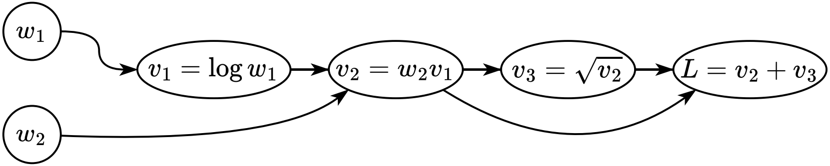


Figure 5: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Let's go from the beginning of the graph to the end and calculate the derivative $\dfrac{\partial L}{\partial w_1}$.

Forward mode automatic differentiation

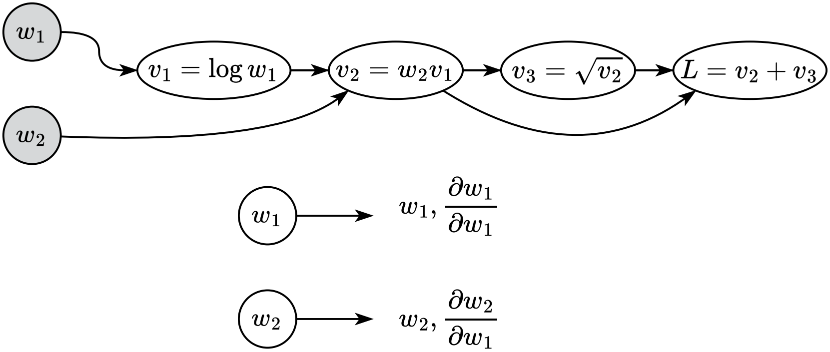


Figure 6: Illustration of forward mode automatic differentiation

Function

$w_1 = w_1, w_2 = w_2$

Forward mode automatic differentiation

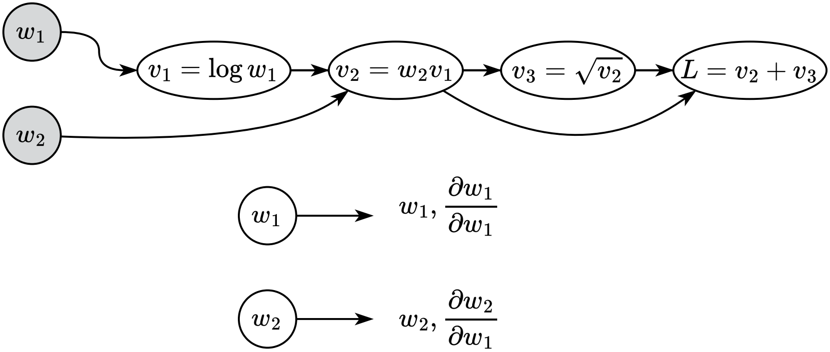


Figure 6: Illustration of forward mode automatic differentiation

Function

$w_1 = w_1, w_2 = w_2$

Derivative

$\frac{\partial w_1}{\partial w_1} = 1, \frac{\partial w_2}{\partial w_1} = 0$

Forward mode automatic differentiation

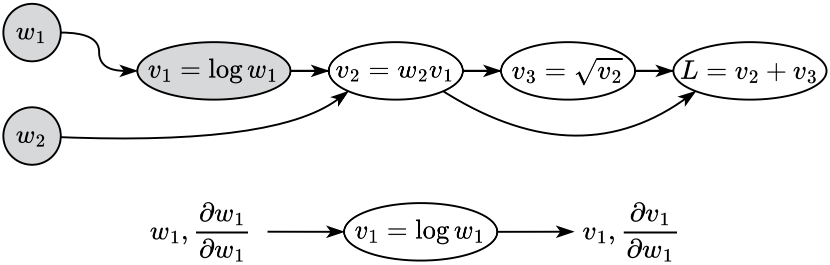


Figure 7: Illustration of forward mode automatic differentiation

Forward mode automatic differentiation

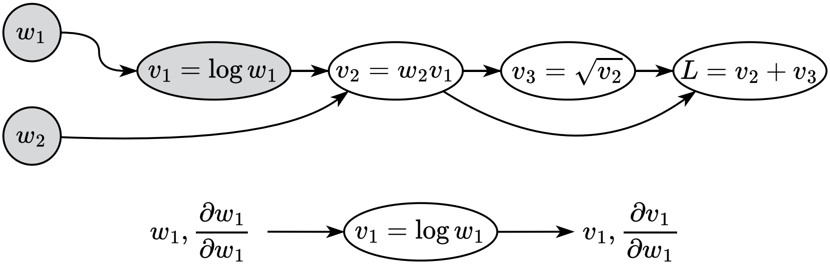


Figure 7: Illustration of forward mode automatic differentiation

Function

$v_1 = \log w_1$

Forward mode automatic differentiation

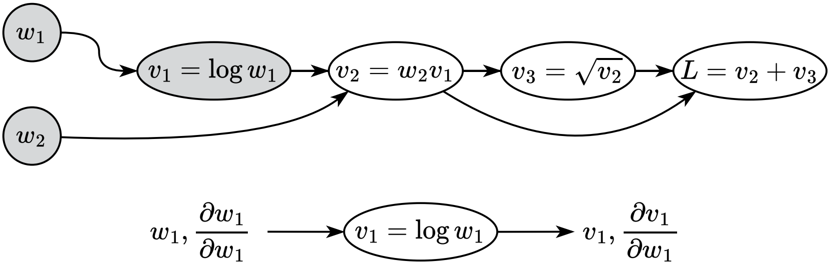


Figure 7: Illustration of forward mode automatic differentiation

Function

$v_1 = \log w_1$

Derivative

$\frac{\partial v_1}{\partial w_1} = \frac{\partial v_1}{\partial w_1} \frac{\partial w_1}{\partial w_1} = \frac{1}{w_1} 1$

Forward mode automatic differentiation

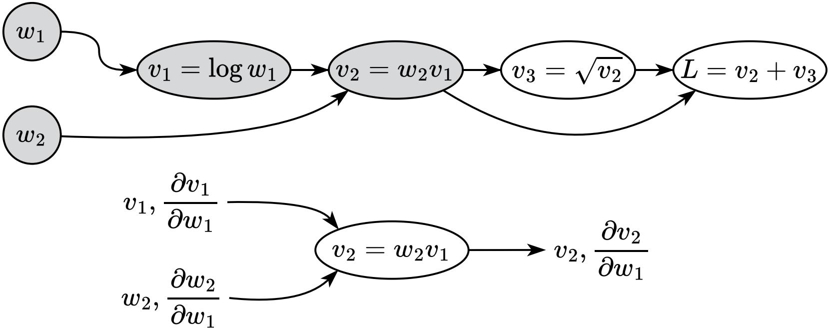


Figure 8: Illustration of forward mode automatic differentiation

Forward mode automatic differentiation

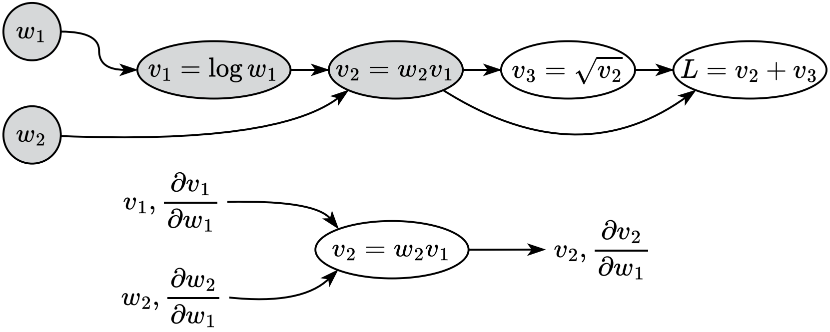


Figure 8: Illustration of forward mode automatic differentiation

Function

$v_2 = w_2 v_1$

Forward mode automatic differentiation

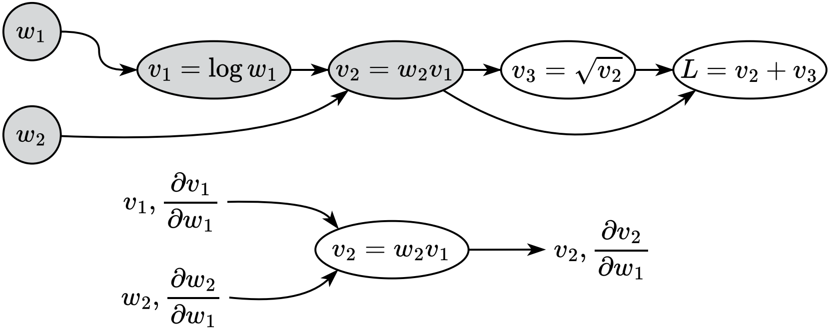


Figure 8: Illustration of forward mode automatic differentiation

Function

$v_2 = w_2 v_1$

Derivative

$\frac{\partial v_2}{\partial w_1} = \frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial w_1} + \frac{\partial v_2}{\partial w_2}\frac{\partial w_2}{\partial w_1} = w_2\frac{\partial v_1}{\partial w_1} + v_1\frac{\partial w_2}{\partial w_1}$

Forward mode automatic differentiation

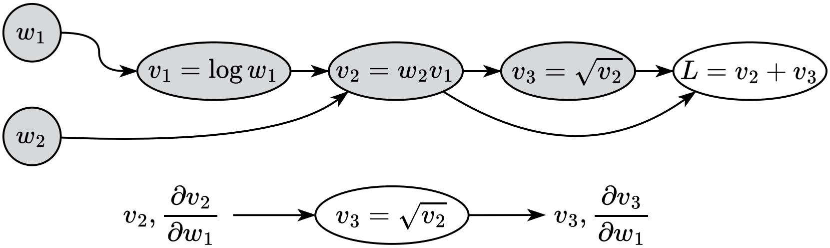


Figure 9: Illustration of forward mode automatic differentiation

Forward mode automatic differentiation

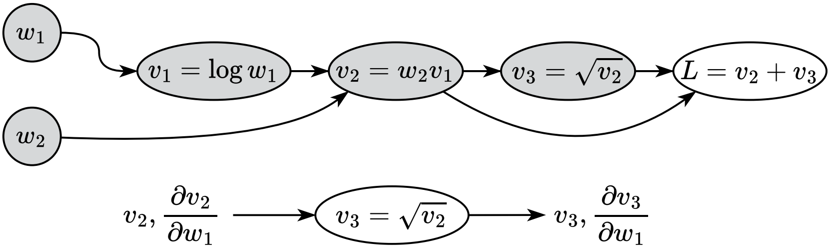


Figure 9: Illustration of forward mode automatic differentiation

Function

$v_3 = \sqrt{v_2}$

Forward mode automatic differentiation

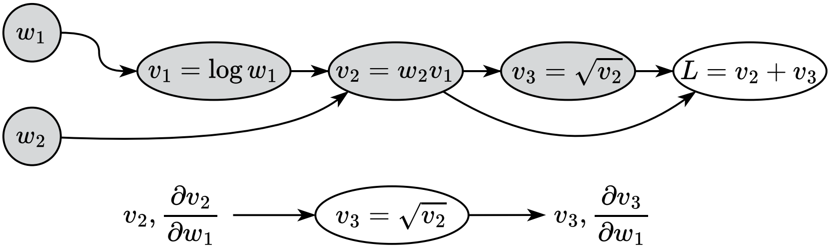


Figure 9: Illustration of forward mode automatic differentiation

Function

$$v_3 = \sqrt{v_2}$$

Derivative

$$\frac{\partial v_3}{\partial w_1} = \frac{\partial v_3}{\partial v_2}\frac{\partial v_2}{\partial w_1} = \frac{1}{2\sqrt{v_2}}\frac{\partial v_2}{\partial w_1}$$

Forward mode automatic differentiation

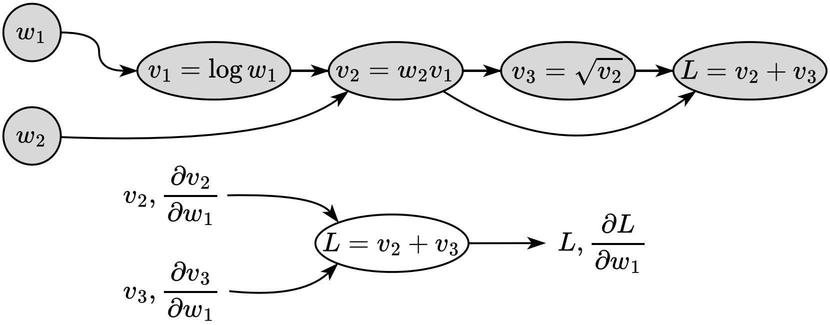


Figure 10: Illustration of forward mode automatic differentiation

Forward mode automatic differentiation

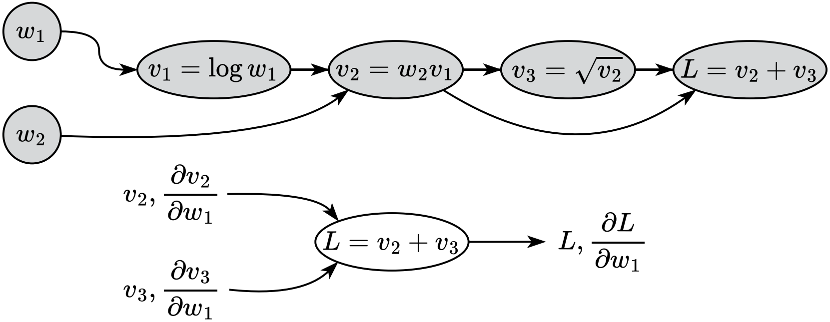


Figure 10: Illustration of forward mode automatic differentiation

Function

$L = v_2 + v_3$

Forward mode automatic differentiation

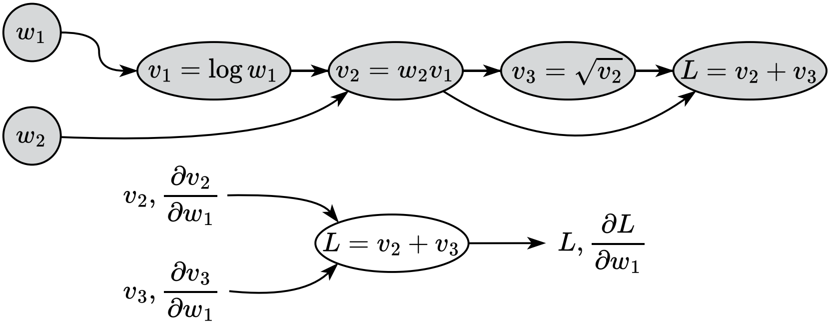


Figure 10: Illustration of forward mode automatic differentiation

Function

$L = v_2 + v_3$

Derivative

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial w_1} + \frac{\partial L}{\partial v_3}\frac{\partial v_3}{\partial w_1} = 1\frac{\partial v_2}{\partial w_1} + 1\frac{\partial v_3}{\partial w_1}$$

Make the similar computations for $\frac{\partial L}{\partial w_2}$

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

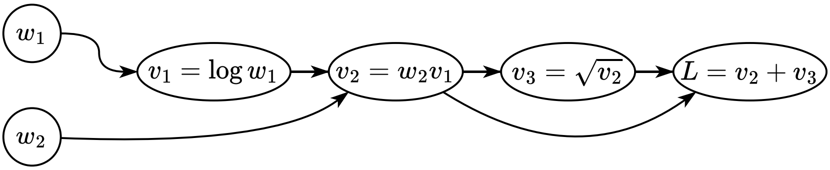


Figure 11: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Forward mode automatic differentiation example

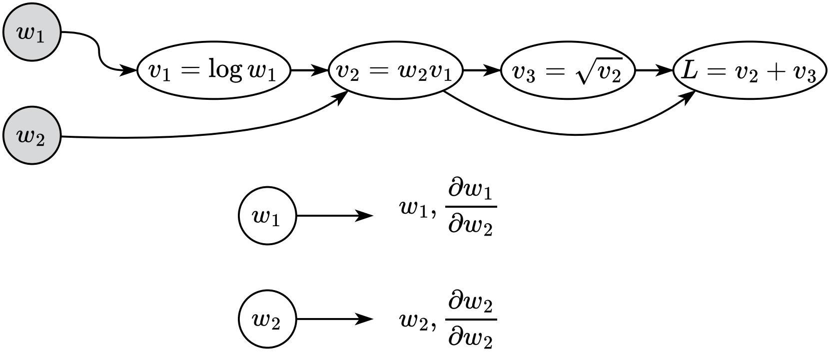


Figure 12: Illustration of forward mode automatic differentiation

Function

$w_1 = w_1, w_2 = w_2$

Derivative

$$\frac{\partial w_1}{\partial w_2} = 0, \frac{\partial w_2}{\partial w_2} = 1$$

Forward mode automatic differentiation example

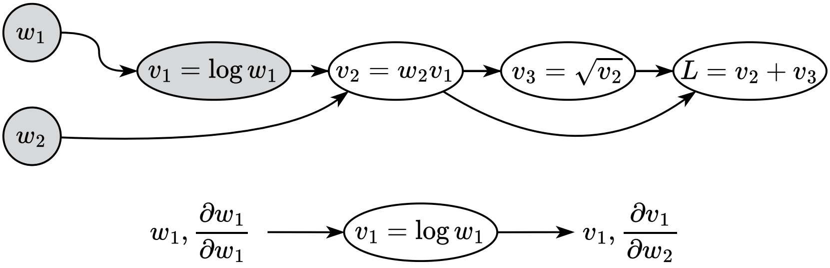


Figure 13: Illustration of forward mode automatic differentiation

Function

$v_1 = \log w_1$

Derivative

$\frac{\partial v_1}{\partial w_2} = \frac{\partial v_1}{\partial w_2} \frac{\partial w_2}{\partial w_2} = 0 \cdot 1$

Forward mode automatic differentiation example

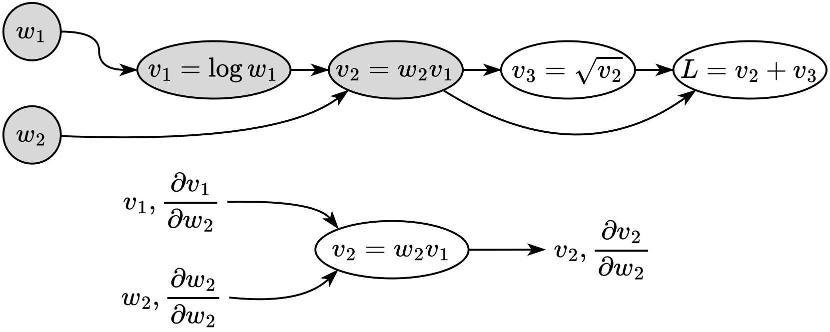


Figure 14: Illustration of forward mode automatic differentiation

Function

$v_2 = w_2 v_1$

Derivative

$$\frac{\partial v_2}{\partial w_2} = \frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial w_2} + \frac{\partial v_2}{\partial w_2}\frac{\partial w_2}{\partial w_2} = w_2 \frac{\partial v_1}{\partial w_2} + v_1 \frac{\partial w_2}{\partial w_2}$$

Forward mode automatic differentiation example

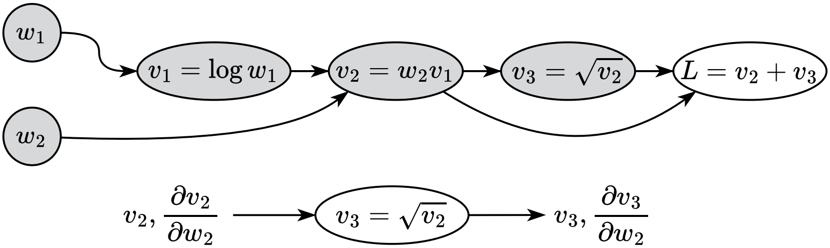


Figure 15: Illustration of forward mode automatic differentiation

Function

$v_3 = \sqrt{v_2}$

Derivative

$\frac{\partial v_3}{\partial w_2} = \frac{\partial v_3}{\partial v_2} \frac{\partial v_2}{\partial w_2} = \frac{1}{2\sqrt{v_2}} \frac{\partial v_2}{\partial w_2}$

Forward mode automatic differentiation example

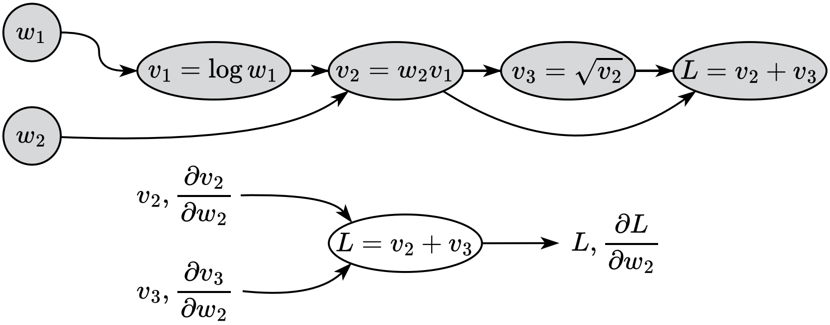


Figure 16: Illustration of forward mode automatic differentiation

Function

$L = v_2 + v_3$

Derivative

$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial w_2} + \frac{\partial L}{\partial v_3}\frac{\partial v_3}{\partial w_2} = 1\frac{\partial v_2}{\partial w_2} + 1\frac{\partial v_3}{\partial w_2}$

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$
\overline{v_i} = \frac{\partial v_i}{\partial w_k}
$$

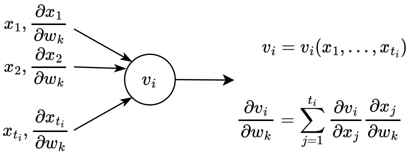


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$

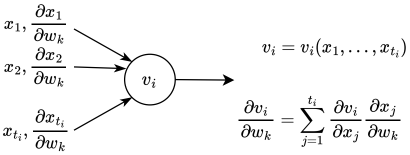


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

- For $i = 1, \ldots, N$:

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$

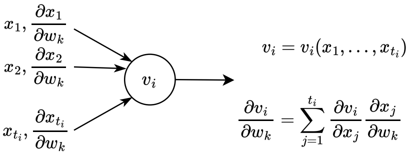


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \dfrac{\partial v_i}{\partial w_k}$$

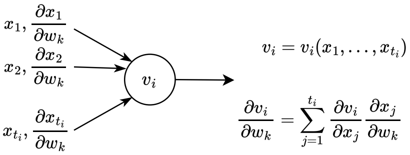


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

$$\overline{v_i} = \sum_{j=1}^{t_i} \dfrac{\partial v_i}{\partial x_j} \dfrac{\partial x_j}{\partial w_k}$$

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$

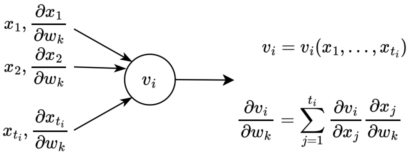


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

$$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

$$\overline{v_i} = \sum_{j=1}^{t_i} \frac{\partial v_i}{\partial x_j} \frac{\partial x_j}{\partial w_k}$$

Forward mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to some input variable w_k, i.e. $\dfrac{\partial v_N}{\partial w_k}$. This idea implies propagation of the gradient with respect to the input variable from start to end, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial v_i}{\partial w_k}$$

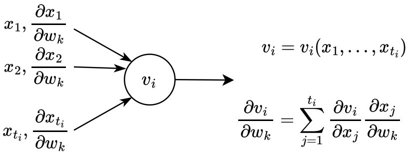


Figure 17: Illustration of forward chain rule to calculate the derivative of the function L with respect to w_k.

- For $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents (inputs) $x_1, \ldots, x_{t_i}$:

    $$v_i = v_i(x_1, \ldots, x_{t_i})$$

  - Compute the derivative $\overline{v_i}$ using the forward chain rule:

    $$\overline{v_i} = \sum_{j=1}^{t_i} \frac{\partial v_i}{\partial x_j} \frac{\partial x_j}{\partial w_k}$$

Note, that this approach does not require storing all intermediate computations, but one can see, that for calculating the derivative $\dfrac{\partial L}{\partial w_k}$ we need $\mathcal{O}(T)$ operations. This means, that for the whole gradient, we need $d\mathcal{O}(T)$ operations, which is the same as for finite differences, but we do not have stability issues, or inaccuracies now (the formulas above are exact).


There is another

Backward mode automatic differentiation

We will consider the same function with a computational graph:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

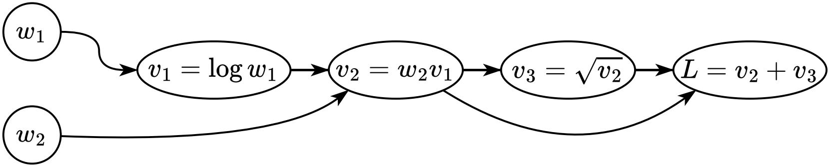


Figure 18: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Backward mode automatic differentiation

We will consider the same function with a computational graph:

$$L(w_1, w_2) = w_2 \log w_1 + \sqrt{w_2 \log w_1}$$

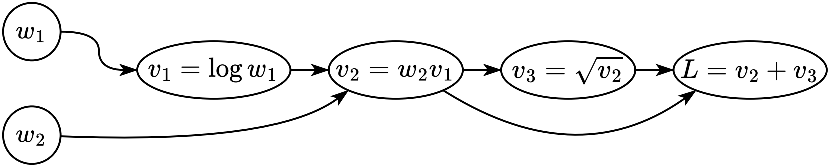


Figure 18: Illustration of computation graph of primitive arithmetic operations for the function $L(w_1, w_2)$

Assume, that we have some values of the parameters w_1, w_2 and we have already performed a forward pass (i.e. single propagation through the computational graph from left to right). Suppose, also, that we somehow saved all intermediate values of v_i. Let's go from the end of the graph to the beginning and calculate the derivatives $\dfrac{\partial L}{\partial w_1}, \dfrac{\partial L}{\partial w_2}$:

Backward mode automatic differentiation example

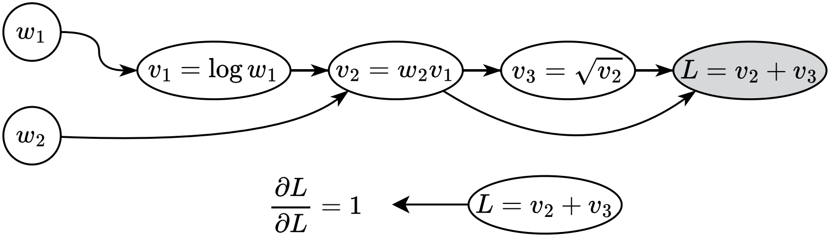


Figure 19: Illustration of backward mode automatic differentiation

Backward mode automatic differentiation example

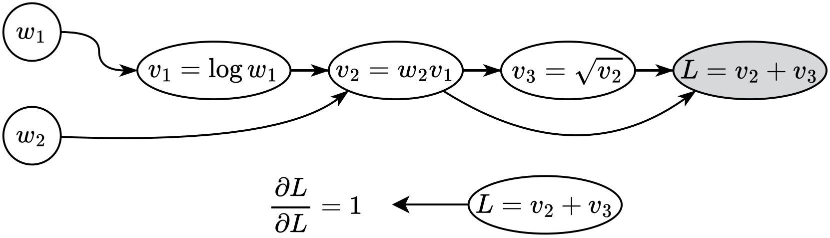


Figure 19: Illustration of backward mode automatic differentiation

Derivatives

Backward mode automatic differentiation example

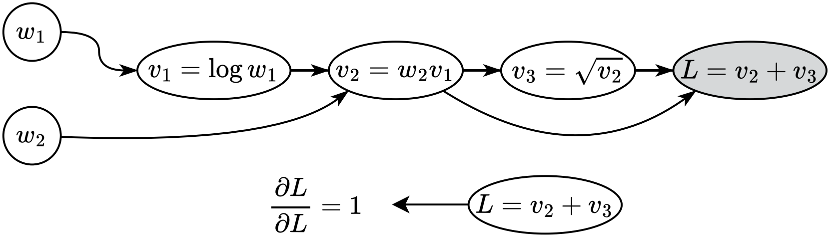


Figure 19: Illustration of backward mode automatic differentiation

Derivatives

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

Backward mode automatic differentiation example

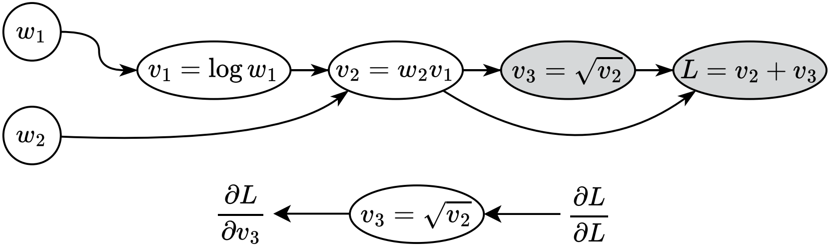


Figure 20: Illustration of backward mode automatic differentiation

Backward mode automatic differentiation example

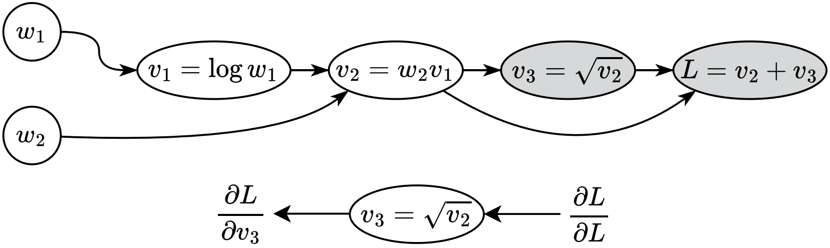


Figure 20: Illustration of backward mode automatic differentiation

Derivatives

Backward mode automatic differentiation example

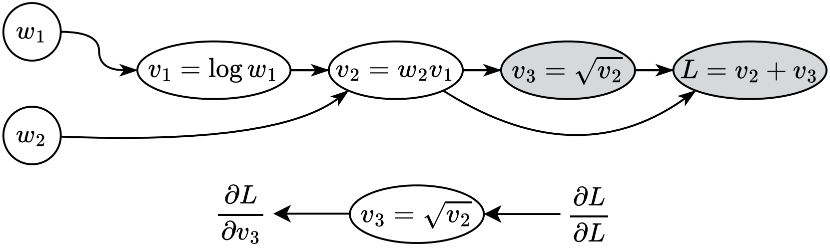


Figure 20: Illustration of backward mode automatic differentiation

Derivatives

$$
\begin{aligned}
\frac{\partial L}{\partial v_3} &= \frac{\partial L}{\partial L}\frac{\partial L}{\partial v_3} \\
&= \frac{\partial L}{\partial L}1
\end{aligned}
$$

Backward mode automatic differentiation example

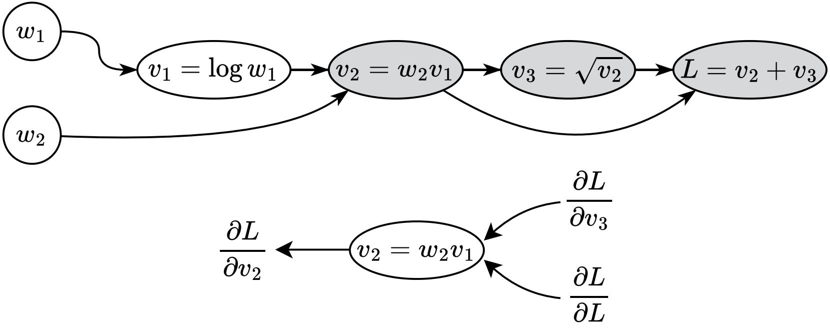


Figure 21: Illustration of backward mode automatic differentiation

Backward mode automatic differentiation example

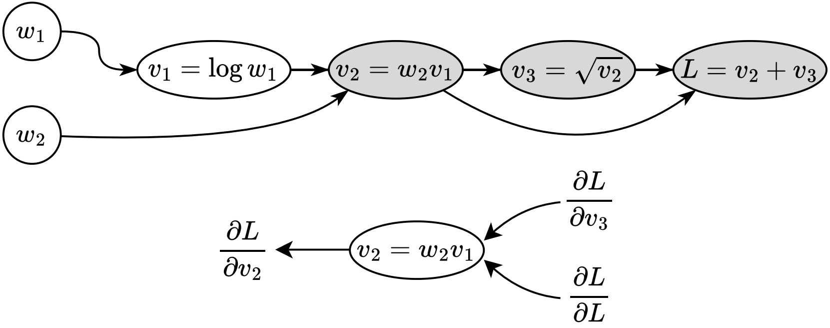


Figure 21: Illustration of backward mode automatic differentiation

Derivatives

Backward mode automatic differentiation example

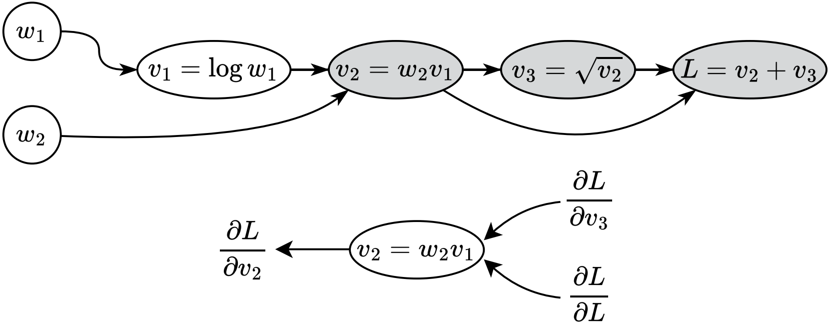


Figure 21: Illustration of backward mode automatic differentiation

Derivatives

$$
\begin{aligned}
\frac{\partial L}{\partial v_2} &= \frac{\partial L}{\partial v_3}\frac{\partial v_3}{\partial v_2} + \frac{\partial L}{\partial L}\frac{\partial L}{\partial v_2} \\
&= \frac{\partial L}{\partial v_3}\frac{1}{2\sqrt{v_2}} + \frac{\partial L}{\partial L}1
\end{aligned}
$$

Backward mode automatic differentiation example

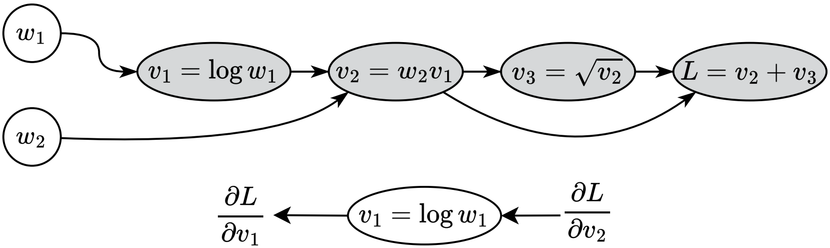


Figure 22: Illustration of backward mode automatic differentiation

Backward mode automatic differentiation example

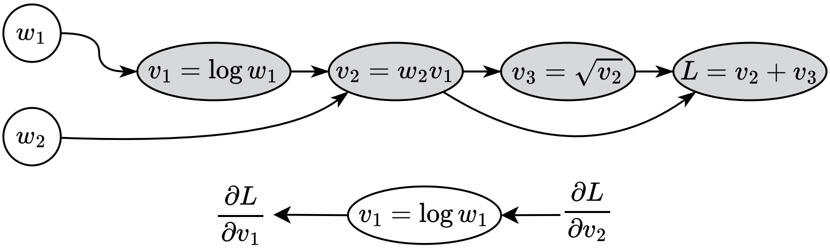


Figure 22: Illustration of backward mode automatic differentiation

Derivatives

Backward mode automatic differentiation example

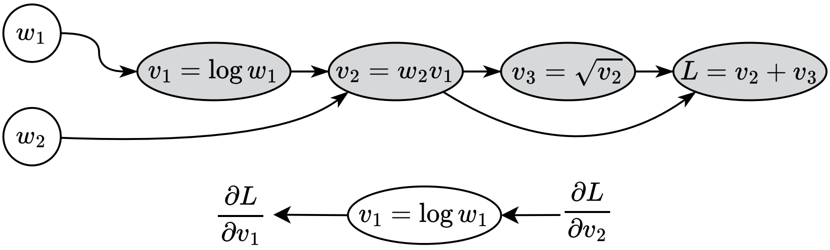


Figure 22: Illustration of backward mode automatic differentiation

Derivatives

$$
\begin{aligned}
\frac{\partial L}{\partial v_1} &= \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial v_1} \\
&= \frac{\partial L}{\partial v_2} w_2
\end{aligned}
$$

Backward mode automatic differentiation example

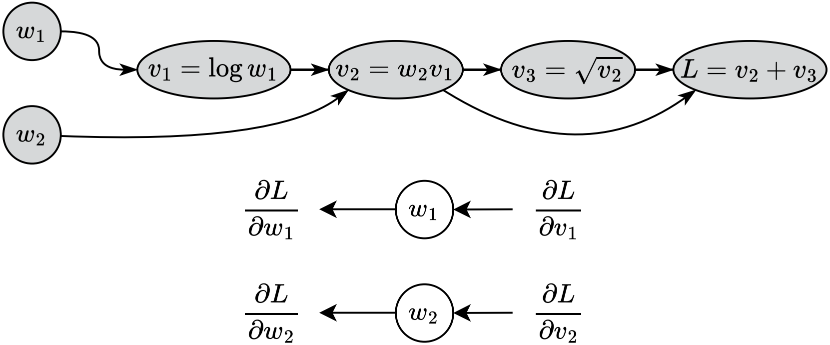


Figure 23: Illustration of backward mode automatic differentiation

Backward mode automatic differentiation example

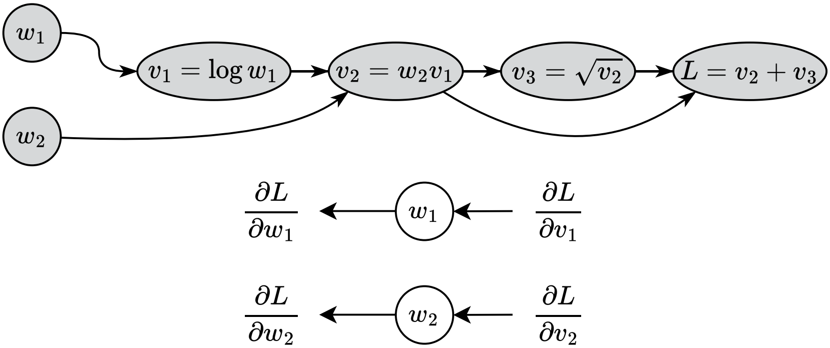


Figure 23: Illustration of backward mode automatic differentiation

Derivatives

Backward mode automatic differentiation example

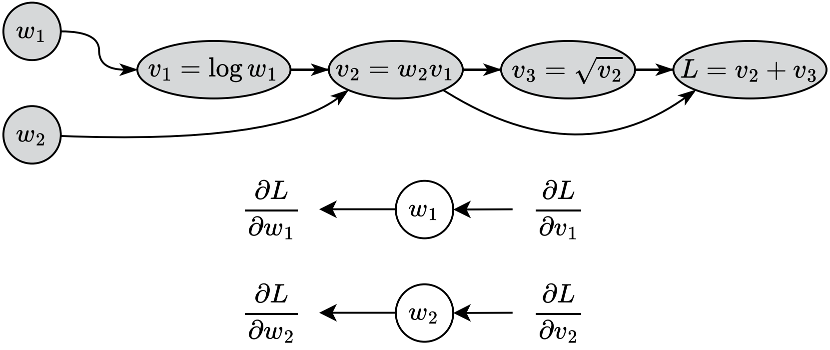


Figure 23: Illustration of backward mode automatic differentiation

Derivatives

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial v_1}\frac{\partial v_1}{\partial w_1} = \frac{\partial L}{\partial v_1}\frac{1}{w_1} \qquad \frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial v_2}\frac{\partial v_2}{\partial w_2} = \frac{\partial L}{\partial v_1} v_1$$

Backward (reverse) mode automatic differentiation

Question

Note, that for the same price of computations as it was in the forward mode we have the full vector of gradient $\nabla_w L$. Is it a free lunch? What is the cost of acceleration?

Backward (reverse) mode automatic differentiation

Question

Note, that for the same price of computations as it was in the forward mode we have the full vector of gradient $\nabla_w L$. Is it a free lunch? What is the cost of acceleration?

Answer Note, that for using the reverse mode AD you need to store all intermediate computations from the forward pass. This problem could be somehow mitigated with the gradient checkpointing approach, which involves necessary recomputations of some intermediate values. This could significantly reduce the memory footprint of the large machine-learning model.

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \ldots, \frac{\partial v_N}{\partial w_d} \right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$

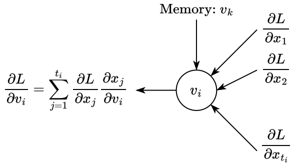


Figure 24: Illustration of reverse chain rule to calculate the derivative of the function L with respect to the node v_i.

- **FORWARD PASS**

  For $i = 1, \ldots, N$:

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \ldots, \frac{\partial v_N}{\partial w_d}\right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$

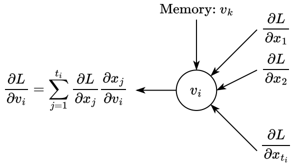


Figure 24: Illustration of reverse chain rule to calculate the derivative of the function L with respect to the node v_i.

- **FORWARD PASS**

  For $i = 1, \ldots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \ldots, \frac{\partial v_N}{\partial w_d} \right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}$$

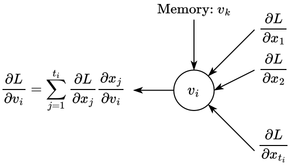


Figure 24: Illustration of reverse chain rule to calculate the derivative of the function L with respect to the node v_i.

- **FORWARD PASS**

  For $i = 1, \ldots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)
- **BACKWARD PASS**

  For $i = N, \ldots, 1$:

Reverse mode automatic differentiation algorithm

Suppose, we have a computational graph $v_i, i \in [1; N]$. Our goal is to calculate the derivative of the output of this graph with respect to all inputs variable w, i.e. $\nabla_w v_N = \left(\frac{\partial v_N}{\partial w_1}, \ldots, \frac{\partial v_N}{\partial w_d} \right)^T$. This idea implies propagation of the gradient of the function with respect to the intermediate variables from the end to the origin, that is why we can introduce the notation:

$$
\overline{v_i} = \frac{\partial L}{\partial v_i} = \frac{\partial v_N}{\partial v_i}
$$

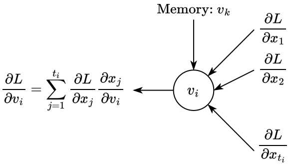


Figure 24: Illustration of reverse chain rule to calculate the derivative of the function L with respect to the node v_i.

- **FORWARD PASS**

  For $i = 1, \ldots, N$:
  - Compute and store the values of v_i as a function of its parents (inputs)
- **BACKWARD PASS**

  For $i = N, \ldots, 1$:
  - Compute the derivative $\overline{v_i}$ using the backward chain rule and information from all of its children (outputs) $(x_1, \ldots, x_{t_i})$:

$$
\overline{v_i} = \frac{\partial L}{\partial v_i} = \sum_{j=1}^{t_i} \frac{\partial L}{\partial x_j} \frac{\partial x_j}{\partial v_i}
$$

Choose your fighter

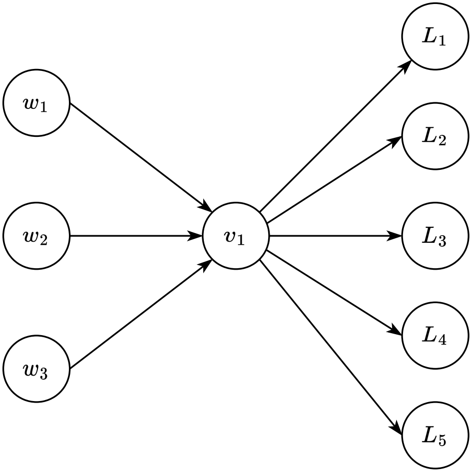


Figure 25: Which mode would you choose for calculating gradients there?

Question

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \frac{\partial L_i}{\partial w_j} \right\}_{i,j}$

Choose your fighter

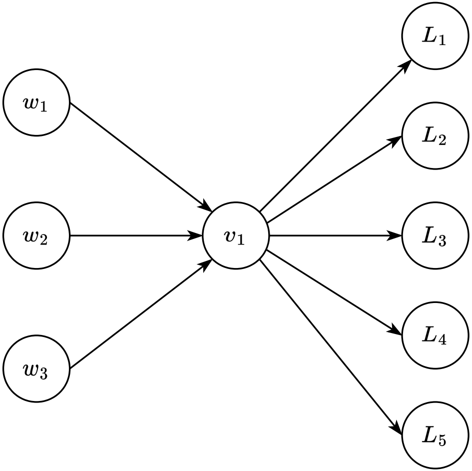


Figure 25: Which mode would you choose for calculating gradients there?

Question

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \frac{\partial L_i}{\partial w_j} \right\}_{i,j}$

Answer Note, that the reverse mode computational time is proportional to the number of outputs here, while the forward mode works proportionally to the number of inputs there. This is why it would be a good idea to consider the forward mode AD.

Choose your fighter

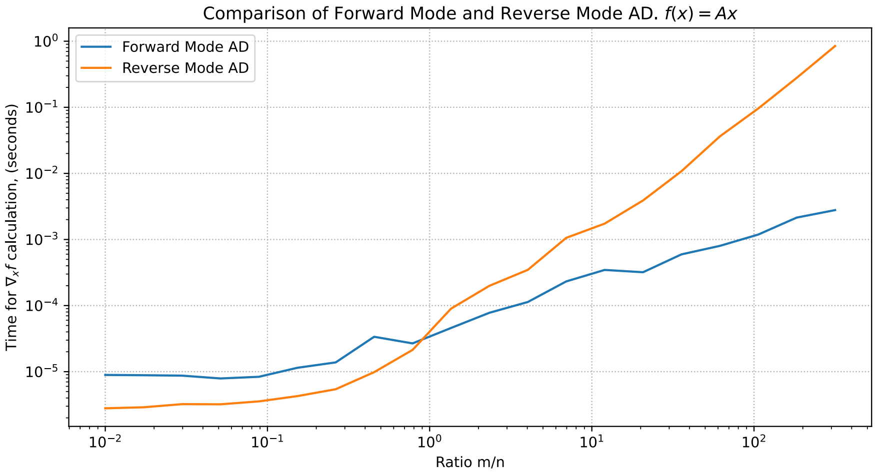


Figure 26: ♣ This graph nicely illustrates the idea of choice between the modes. The $n = 100$ dimension is fixed and the graph presents the time needed for Jacobian calculation w.r.t. x for $f(x) = Ax$

Choose your fighter

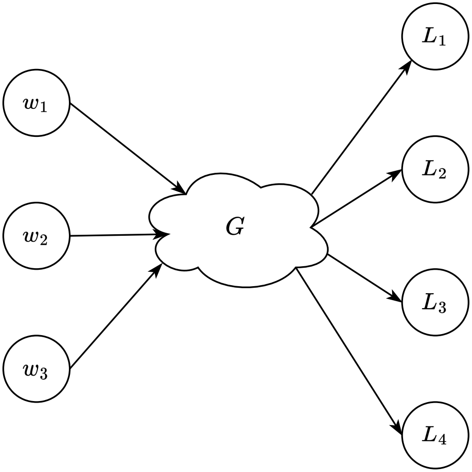


Figure 27: Which mode would you choose for calculating gradients there?

Question

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \dfrac{\partial L_i}{\partial w_j} \right\}_{i,j}$. Note, that G is an arbitrary computational graph

Choose your fighter

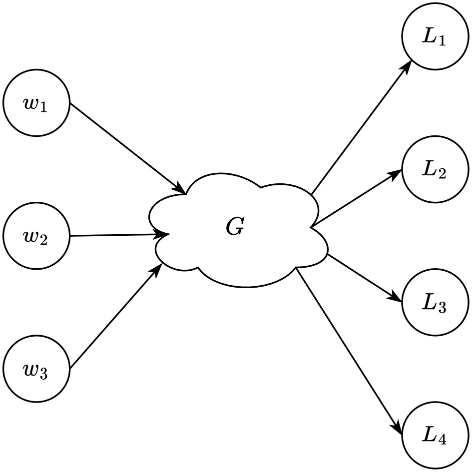


Figure 27: Which mode would you choose for calculating gradients there?

Question

Which of the AD modes would you choose (forward/ reverse) for the following computational graph of primitive arithmetic operations? Suppose, you are needed to compute the jacobian $J = \left\{ \dfrac{\partial L_i}{\partial w_j} \right\}_{i,j}$. Note, that G is an arbitrary computational graph

Answer It is generally impossible to say it without some knowledge about the specific structure of the graph G. Note, that there are also plenty of advanced approaches to mix forward and reverse mode AD, based on the specific G structure.

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.

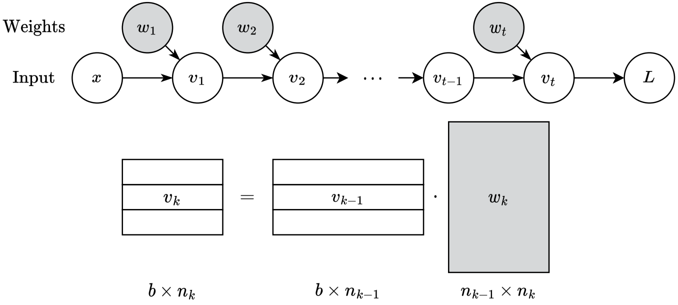


Figure 28: Feedforward neural network architecture

BACKWARD

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \dots, t-1, t$:

BACKWARD

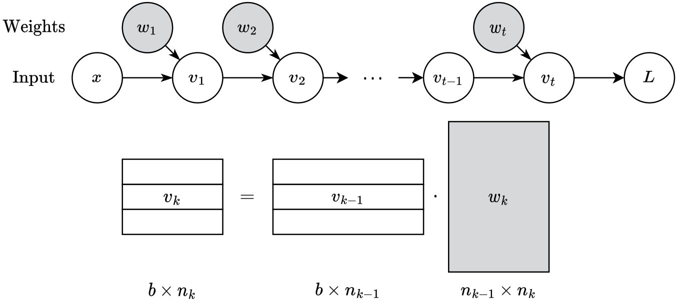


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.

BACKWARD

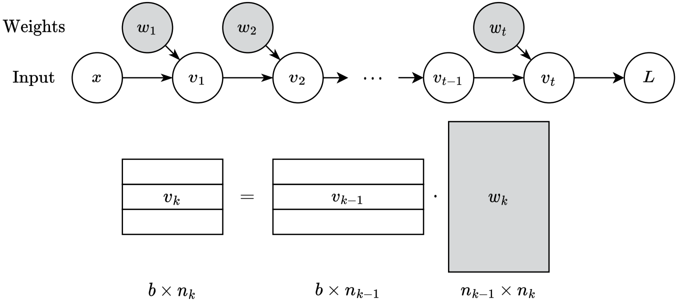


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \dots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

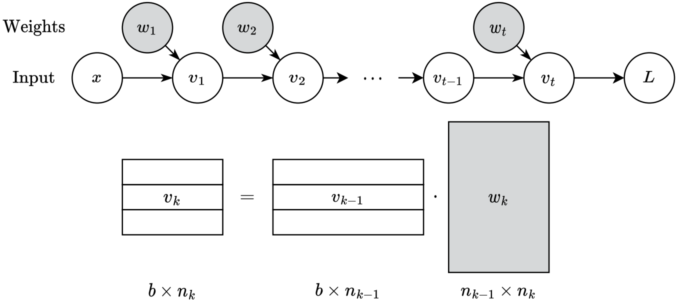


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1}w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$

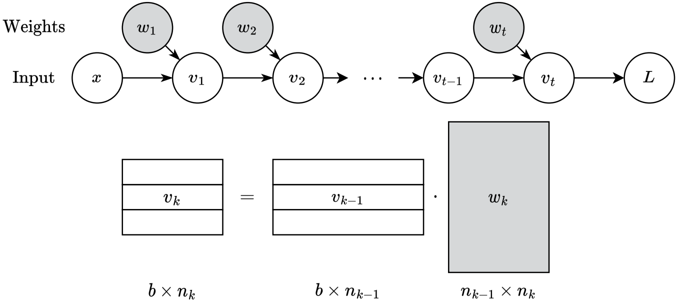


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \dots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \dots, 1$:

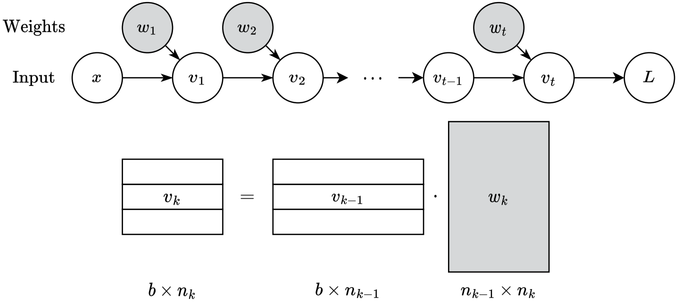


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
  - $v_k = \sigma(v_{k-1} w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \ldots, 1$:
  - $$\underset{b \times n_k}{\frac{\partial L}{\partial v_k}} = \underset{b \times n_{k+1}}{\frac{\partial L}{\partial v_{k+1}}} \underset{n_{k+1} \times n_k}{\frac{\partial v_{k+1}}{\partial v_k}}$$

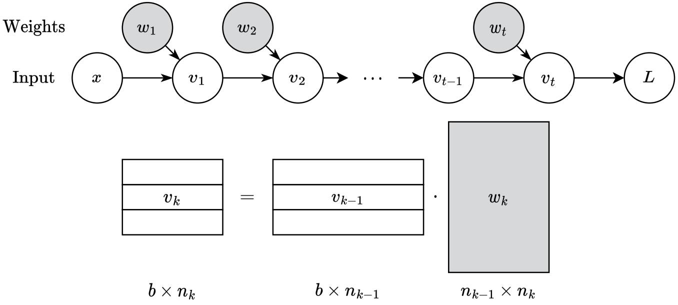


Figure 28: Feedforward neural network architecture

Feedforward Architecture

FORWARD

- $v_0 = x$ typically we have a batch of data x here as an input.
- For $k = 1, \ldots, t-1, t$:
    - $v_k = \sigma(v_{k-1}w_k)$. Note, that practically speaking the data has dimension $x \in \mathbb{R}^{b \times d}$, where b is the batch size (for the single data point $b = 1$). While the weight matrix w_k of a k layer has a shape $n_{k-1} \times n_k$, where n_k is the dimension of an inner representation of the data.
- $L = L(v_t)$ - calculate the loss function.

BACKWARD

- $v_{t+1} = L, \dfrac{\partial L}{\partial L} = 1$
- For $k = t, t-1, \ldots, 1$:
    - $\underset{b \times n_k}{\dfrac{\partial L}{\partial v_k}} = \underset{b \times n_{k+1}}{\dfrac{\partial L}{\partial v_{k+1}}} \underset{n_{k+1} \times n_k}{\dfrac{\partial v_{k+1}}{\partial v_k}}$
    - $\underset{b \times n_{k-1} \cdot n_k}{\dfrac{\partial L}{\partial w_k}} = \underset{b \times n_{k+1}}{\dfrac{\partial L}{\partial v_{k+1}}} \cdot \underset{n_{k+1} \times n_{k-1} \cdot n_k}{\dfrac{\partial v_{k+1}}{\partial w_k}}$

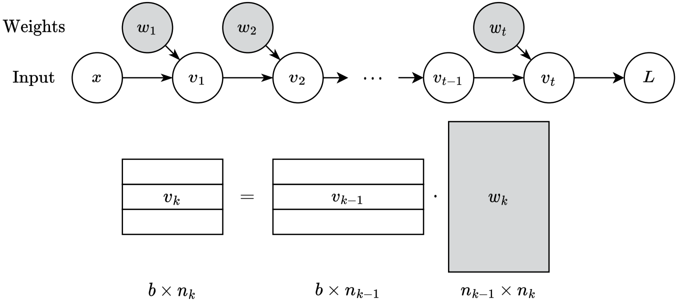


Figure 28: Feedforward neural network architecture

Gradient propagation through the linear least squares

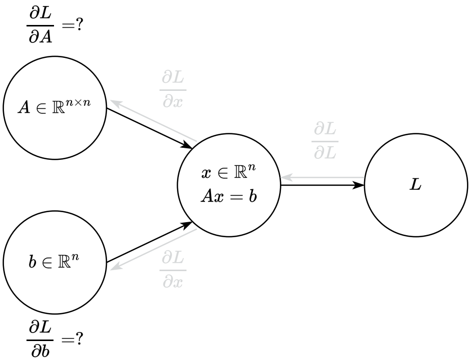


Figure 29: x could be found as a solution of linear system

Suppose, we have an invertible matrix A and a vector b, the vector x is the solution of the linear system $Ax = b$, namely one can write down an analytical solution $x = A^{-1}b$, in this example we will show, that computing all derivatives $\frac{\partial L}{\partial A}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial x}$, i.e. the backward pass, costs approximately the same as the forward pass.

Gradient propagation through the linear least squares

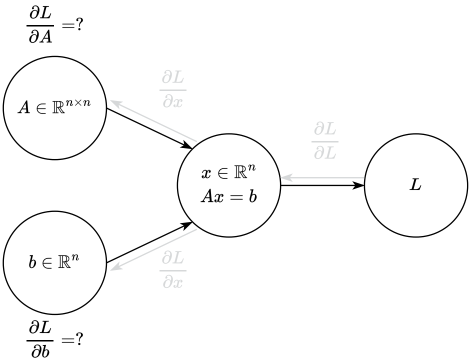


Figure 29: x could be found as a solution of linear system

Suppose, we have an invertible matrix A and a vector b, the vector x is the solution of the linear system $Ax = b$, namely one can write down an analytical solution $x = A^{-1}b$, in this example we will show, that computing all derivatives $\frac{\partial L}{\partial A}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial x}$, i.e. the backward pass, costs approximately the same as the forward pass.
It is known, that the differential of the function does not depend on the parametrization:

$$dL = \left\langle \frac{\partial L}{\partial x}, dx \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Gradient propagation through the linear least squares

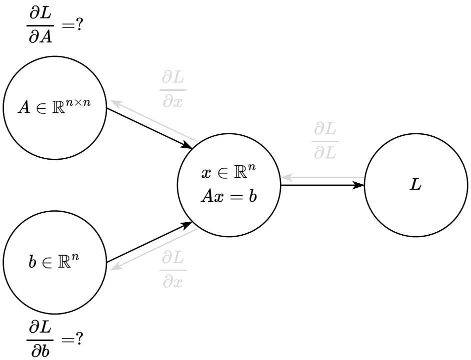


Figure 29: x could be found as a solution of linear system

Suppose, we have an invertible matrix A and a vector b, the vector x is the solution of the linear system $Ax = b$, namely one can write down an analytical solution $x = A^{-1}b$, in this example we will show, that computing all derivatives $\frac{\partial L}{\partial A}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial x}$, i.e. the backward pass, costs approximately the same as the forward pass.

It is known, that the differential of the function does not depend on the parametrization:

$$dL = \left\langle \frac{\partial L}{\partial x}, dx \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Given the linear system, we have:

$$Ax = b$$

$$dAx + Adx = db \to dx = A^{-1}(db - dAx)$$

Gradient propagation through the linear least squares

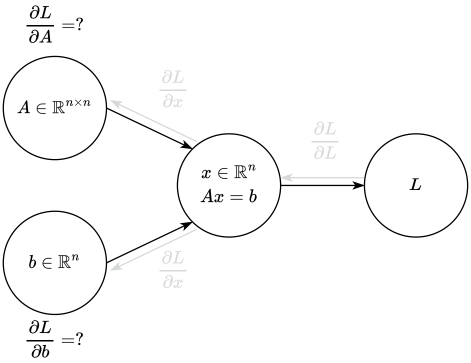


Figure 30: x could be found as a solution of linear system

The straightforward substitution gives us:

$$\left\langle \frac{\partial L}{\partial x}, A^{-1}(db - dAx) \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Gradient propagation through the linear least squares

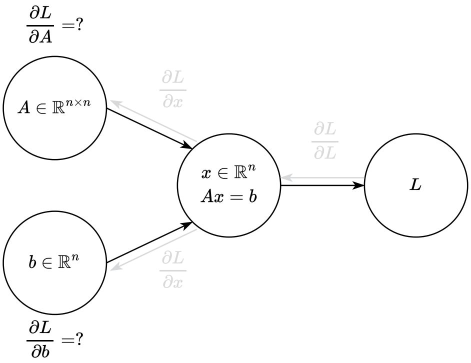


Figure 30: x could be found as a solution of linear system

The straightforward substitution gives us:

$$\left\langle \frac{\partial L}{\partial x}, A^{-1}(db - dAx) \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

$$\left\langle -A^{-T}\frac{\partial L}{\partial x}x^T, dA \right\rangle + \left\langle A^{-T}\frac{\partial L}{\partial x}, db \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Gradient propagation through the linear least squares

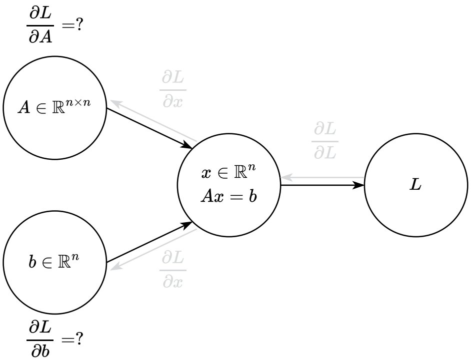


Figure 30: x could be found as a solution of linear system

The straightforward substitution gives us:

$$\left\langle \frac{\partial L}{\partial x}, A^{-1}(db - dAx) \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

$$\left\langle -A^{-T}\frac{\partial L}{\partial x}x^T, dA \right\rangle + \left\langle A^{-T}\frac{\partial L}{\partial x}, db \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Therefore:

$$\frac{\partial L}{\partial A} = -A^{-T}\frac{\partial L}{\partial x}x^T \quad \frac{\partial L}{\partial b} = A^{-T}\frac{\partial L}{\partial x}$$

Gradient propagation through the linear least squares

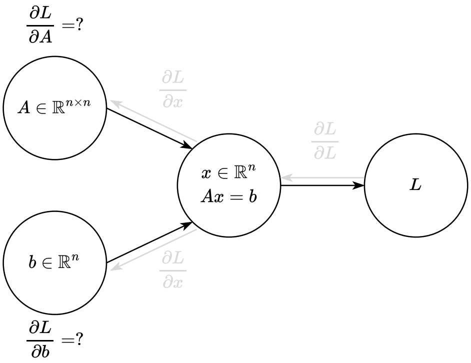


Figure 30: x could be found as a solution of linear system

The straightforward substitution gives us:

$$\left\langle \frac{\partial L}{\partial x}, A^{-1}(db - dAx) \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

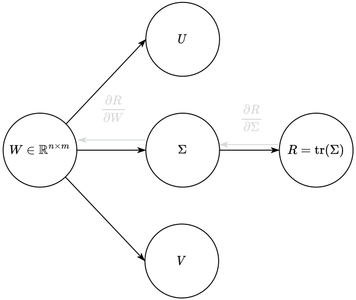
$$\left\langle -A^{-T}\frac{\partial L}{\partial x}x^T, dA \right\rangle + \left\langle A^{-T}\frac{\partial L}{\partial x}, db \right\rangle = \left\langle \frac{\partial L}{\partial A}, dA \right\rangle + \left\langle \frac{\partial L}{\partial b}, db \right\rangle$$

Therefore:

$$\frac{\partial L}{\partial A} = -A^{-T}\frac{\partial L}{\partial x}x^T \quad \frac{\partial L}{\partial b} = A^{-T}\frac{\partial L}{\partial x}$$

It is interesting, that the most computationally intensive part here is the matrix inverse, which is the same as for the forward pass. Sometimes it is even possible to store the result itself, which makes the backward pass even cheaper.

Gradient propagation through the SVD



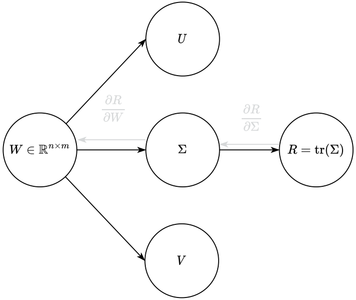
Suppose, we have the rectangular matrix $W \in \mathbb{R}^{m \times n}$, which has a singular value decomposition:

$$W = U\Sigma V^T, \quad U^TU = I, \quad V^TV = I, \quad \Sigma = \text{diag}(\sigma_1, \ldots, \sigma_{\min(m,n)})$$

1. Similarly to the previous example:

$$\begin{aligned}
W &= U\Sigma V^T \\
dW &= dU\Sigma V^T + Ud\Sigma V^T + U\Sigma dV^T \\
U^T dW V &= U^T dU\Sigma V^T V + U^T U d\Sigma V^T V + U^T U\Sigma dV^T V \\
U^T dW V &= U^T dU\Sigma + d\Sigma + \Sigma dV^T V
\end{aligned}$$

Gradient propagation through the SVD



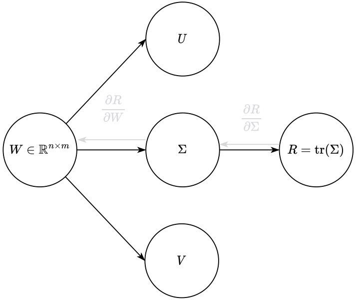
2. Note, that $U^TU = I \to dU^TU + U^TdU = 0$. But also $dU^TU = (U^TdU)^T$, which actually involves, that the matrix U^TdU is antisymmetric:

$$(U^TdU)^T + U^TdU = 0 \quad \to \quad \text{diag}(U^TdU) = (0, \dots, 0)$$

The same logic could be applied to the matrix V and

$$\text{diag}(dV^TV) = (0, \dots, 0)$$

Gradient propagation through the SVD



2. Note, that $U^TU = I \to dU^TU + U^TdU = 0$. But also $dU^TU = (U^TdU)^T$, which actually involves, that the matrix U^TdU is antisymmetric:

$$(U^TdU)^T + U^TdU = 0 \quad \to \quad \text{diag}(U^TdU) = (0, \dots, 0)$$

The same logic could be applied to the matrix V and

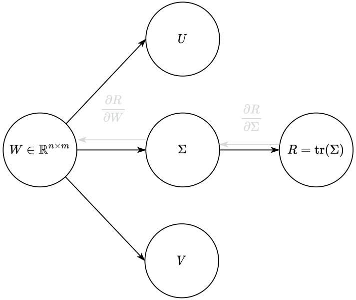
$$\text{diag}(dV^TV) = (0, \dots, 0)$$

3. At the same time, the matrix $d\Sigma$ is diagonal, which means (look at the 1.) that

$$\text{diag}(U^TdWV) = d\Sigma$$

Here on both sides, we have diagonal matrices.

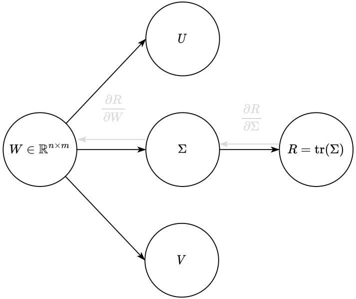
Gradient propagation through the SVD



4. Now, we can decompose the differential of the loss function as a function of Σ - such problems arise in ML problems, where we need to restrict the matrix rank:

$$
\begin{aligned}
dL &= \left\langle \frac{\partial L}{\partial \Sigma}, d\Sigma \right\rangle \\
&= \left\langle \frac{\partial L}{\partial \Sigma}, \text{diag}(U^T dW V) \right\rangle \\
&= \text{tr}\left(\frac{\partial L}{\partial \Sigma}^T \text{diag}(U^T dW V) \right)
\end{aligned}
$$

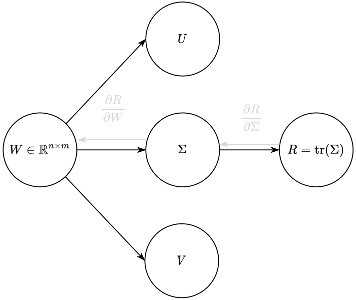
Gradient propagation through the SVD



5. As soon as we have diagonal matrices inside the product, the trace of the diagonal part of the matrix will be equal to the trace of the whole matrix:

$$
\begin{aligned}
dL &= \text{tr}\left(\frac{\partial L}{\partial \Sigma}^T \text{diag}(U^T dW V)\right) \\
&= \text{tr}\left(\frac{\partial L}{\partial \Sigma}^T U^T dW V\right) \\
&= \left\langle \frac{\partial L}{\partial \Sigma}, U^T dW V \right\rangle \\
&= \left\langle U \frac{\partial L}{\partial \Sigma} V^T, dW \right\rangle
\end{aligned}
$$

Gradient propagation through the SVD



6. Finally, using another parametrization of the differential

$$\left\langle U \frac{\partial L}{\partial \Sigma} V^T, dW \right\rangle = \left\langle \frac{\partial L}{\partial W}, dW \right\rangle$$

$$\frac{\partial L}{\partial W} = U \frac{\partial L}{\partial \Sigma} V^T,$$

This nice result allows us to connect the gradients $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial \Sigma}$.

Hessian vector product without the Hessian

When you need some information about the curvature of the function you usually need to work with the hessian. However, when the dimension of the problem is large it is challenging. For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian at a point $x \in \mathbb{R}^n$ is written as $\nabla^2 f(x)$. A Hessian-vector product function is then able to evaluate

$$v \mapsto \nabla^2 f(x) \cdot v$$

for any vector $v \in \mathbb{R}^n$. We have to use the identity

$$\nabla^2 f(x) v = \nabla [x \mapsto \nabla f(x) \cdot v] = \nabla g(x),$$

where $g(x) = \nabla f(x)^T \cdot v$ is a new vector-valued function that dots the gradient of f at x with the vector v.

```
import jax.numpy as jnp

def hvp(f, x, v):
    return grad(lambda x: jnp.vdot(grad(f)(x), v))(x)
```

Hutchinson Trace Estimation [2]

This example illustrates the estimation the Hessian trace of a neural network using Hutchinson's method, which is an algorithm to obtain such an estimate from matrix-vector products:

Let $X \in \mathbb{R}^{d \times d}$ and $v \in \mathbb{R}^d$ be a random vector such that $\mathbb{E}[vv^T] = I$. Then,

$$\mathrm{Tr}(X) = \mathbb{E}[v^T X v] = \frac{1}{V}\sum_{i=1}^{V} v_i^T X v_i$$

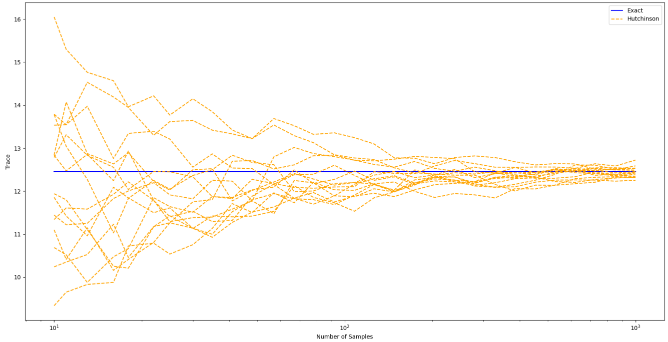


Figure 31: Source

[2] A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines - M.F. Hutchinson, 1990

Activation checkpointing

The animated visualization of the above approaches

An example of using a gradient checkpointing

What automatic differentiation (AD) is NOT:

- AD is not a finite differences

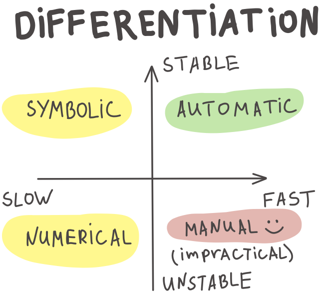


Figure 32: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative

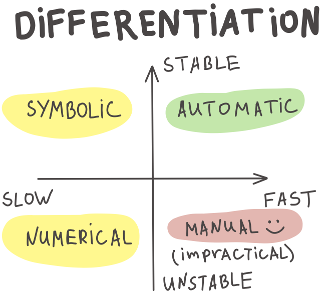


Figure 32: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule

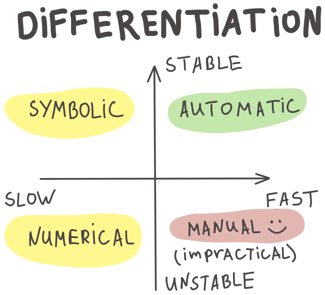


Figure 32: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation

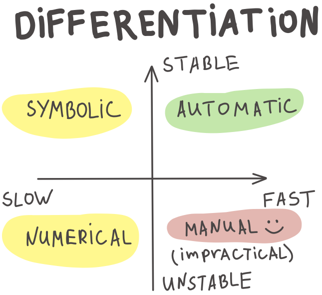


Figure 32: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation
- AD (reverse mode) is time-efficient and numerically stable

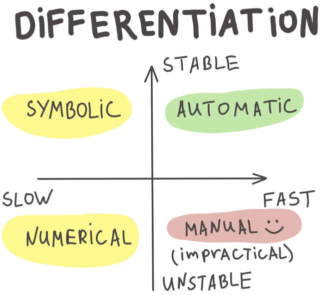


Figure 32: Different approaches for taking derivatives

What automatic differentiation (AD) is NOT:

- AD is not a finite differences
- AD is not a symbolic derivative
- AD is not just the chain rule
- AD is not just backpropagation
- AD (reverse mode) is time-efficient and numerically stable
- AD (reverse mode) is memory inefficient (you need to store all intermediate computations from the forward pass).

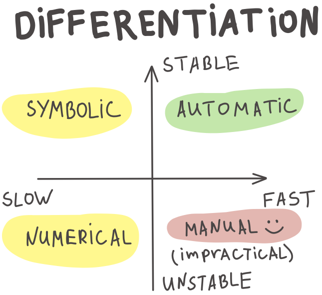


Figure 32: Different approaches for taking derivatives

Code

Open In Colab ♣