Lab 4: Computing Derivatives and Autograd

Andy Ye, Leo Xu, Alex Gichamba

20/09/2024

Agenda

- Differentiation Methods
- Automatic Differentiation
- P1 Autograd Walkthrough

Recap: Training by Backprop

Iterate:

1. Forward pass
2. Backward pass
3. Update parameters

Recap: Training by Backprop

Initialize weights $W^{(k)}$ for all layers $k = 1 \dots K$

Do: (*Gradient descent iterations*)

Initialize Loss $= 0$; for all i, j, k, initialize $\frac{d\text{Loss}}{dw_{i,j}^{(k)}} = 0$

For all $t = 1 : T$ (*Iterate over training instances*):

Forward pass:

$$\text{Compute Output } Y_t$$

$$\text{Loss} += \text{Div}(Y_t, d_t)$$

Backward pass: For all i, j, k:

$$\textit{Compute } \frac{d\text{Div}(Y_t, d_t)}{dw_{i,j}^{(k)}}$$

$$\frac{d\text{Loss}}{dw_{i,j}^{(k)}} += \frac{d\text{Div}(Y_t, d_t)}{dw_{i,j}^{(k)}}$$

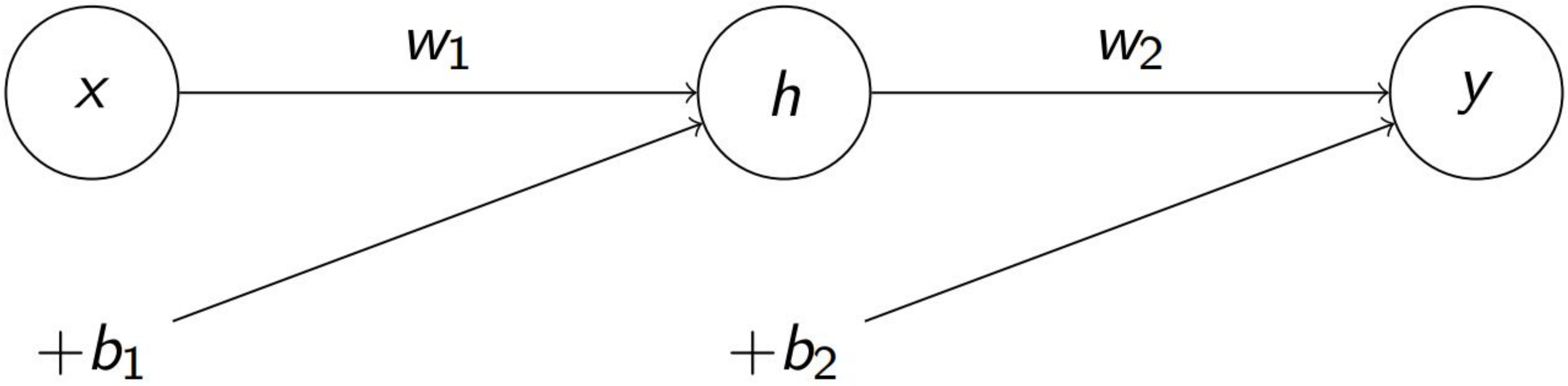
For all i, j, k, update:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{d\text{Loss}}{dw_{i,j}^{(k)}}$$

Until Loss has converged

Differentiation Methods

A simple network for illustration



- **Activation Function**: Sigmoid $\sigma(z) = \dfrac{1}{1 + e^{-z}}$

Forward Pass

- **Hidden Layer Output**:

$$h = \sigma(z_1) = \sigma(w_1 x + b_1)$$

- **Output Layer Output**:

$$y = \sigma(z_2) = \sigma(w_2 h + b_2)$$

Method 1: Hand-coding Derivatives (P1 style)

- **Definition**: Compute derivatives analytically using calculus rules.
- **Process**:
  - Compute $\frac{\partial y}{\partial w_2}$, $\frac{\partial y}{\partial h}$, $\frac{\partial h}{\partial w_1}$, etc.
  - Use chain rule to combine derivatives.
- **Example**:

$$\frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial w_2} = \sigma'(z_2)h$$

Pros and Cons of Hand-coding

Pros	Cons
Exact Fast if well implemented	Error prone Wasteful if you need generalizable functionality (like in a library) Time consuming to code

Method 2: Numerical Differentiation

- **Definition**: Approximate derivatives using finite differences.
- **Formula**:

$$\frac{\partial y}{\partial w} \approx \frac{y(w+\epsilon) - y(w-\epsilon)}{2\epsilon}$$

- **Considerations**:
  - Easy to code
  - Choice of ϵ affects accuracy.
  - Computationally expensive for high-dimensional problems.

Recap: Neural Nets are just Nested Functions

$$
\begin{aligned}
\mathbf{z}_1 &= f_1(\mathbf{x}) \\
\mathbf{y}_1 &= g_1(\mathbf{z}_1) \\
\mathbf{z}_2 &= f_2(\mathbf{y}_1) \\
\mathbf{y}_2 &= g_2(\mathbf{z}_2) \\
&\vdots \\
\mathbf{y}_N &= g_N(f_N(\mathbf{y}_{N-1})) \\
D &= D(\mathbf{y}_N)
\end{aligned}
$$

Recap: Chain Rule for Differentiating Nested Functions

Chain Rule

$$D = D\left(y_N\left(z_N\left(y_{N-1}\left(z_{N-1}\left(\ldots y_1\left(z_1(\mathbf{x})\right)\right)\right)\right)\right)\right)$$

Gradient Calculation (Backward Pass)

$$\nabla_{\mathbf{x}} D = \nabla_{y_N} D \, \nabla_{z_N} y_N \, \nabla_{y_{N-1}} z_N \, \nabla_{z_{N-1}} y_{N-1} \cdots \nabla_{z_1} y_1 \, \nabla_{\mathbf{x}} z_1$$

Method 3: Automatic Differentiation

Forward Pass: Tracking Operations

- **Input Data**: Start with input $\mathbf{x}$.
- **Primitive Operations**: Each layer applies a simple operation (addition, multiplication, activation functions).
- **Track Computations**: Store a computational graph—keep track of all operations and intermediate results.
- **Intermediate Variables**: For each operation, save the result for use in the backward pass (e.g., layer outputs, activations).

Method 3: Automatic Differentiation

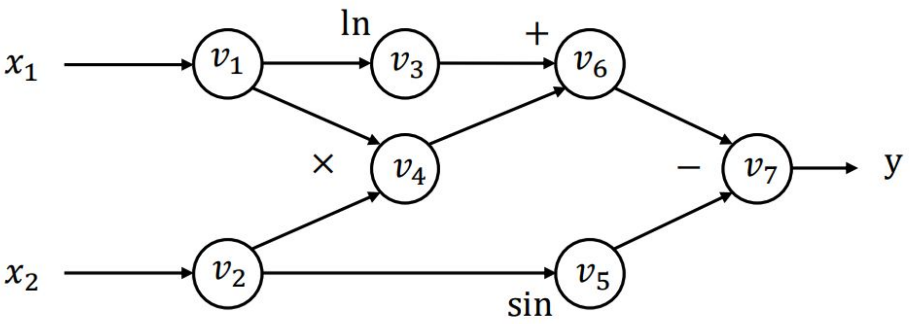
Backward Pass: Chain Rule to Compute Gradients

- **Gradient Calculation**: Starting from the final output, apply chain rule backwards to each primitive operation.
- **Propagate Gradients**: At each step, propagate gradients through the graph to the previous operations.
- **Use Intermediate Values**: Reuse the saved intermediate variables from the forward pass to compute gradients.
- **Update Parameters**: Once gradients are computed, update the model parameters (e.g., using gradient descent).

Automatic Differentiation

Computational graph

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$



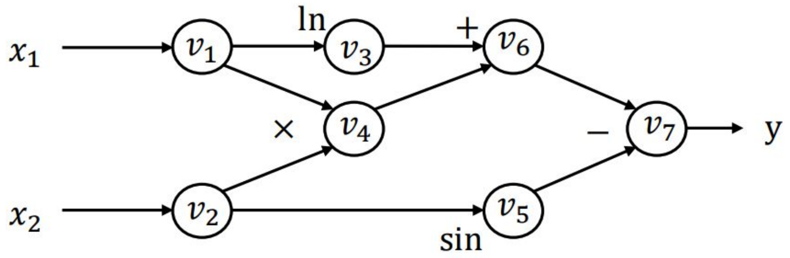
Forward evaluation trace

$$v_1 = x_1 = 2$$
$$v_2 = x_2 = 5$$
$$v_3 = \ln v_1 = \ln 2 = 0.693$$
$$v_4 = v_1 \times v_2 = 10$$
$$v_5 = \sin v_2 = \sin 5 = -0.959$$
$$v_6 = v_3 + v_4 = 10.693$$
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$
$$y = v_7 = 11.652$$

Each node represent an (intermediate) value in the computation. Edges present input output relations.

Forward mode automatic differentiation (AD)

$\mathrm{y} = \mathrm{f}(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$



Define $\dot{v_i} = \frac{\partial v_i}{\partial x_1}$

We can then compute the $\dot{v_i}$ iteratively in the forward topological order of the computational graph

Forward evaluation trace

$$
\begin{aligned}
&v_1 = x_1 = 2 \\
&v_2 = x_2 = 5 \\
&v_3 = \ln v_1 = \ln 2 = 0.693 \\
&v_4 = v_1 \times v_2 = 10 \\
&v_5 = \sin v_2 = \sin 5 = -0.959 \\
&v_6 = v_3 + v_4 = 10.693 \\
&v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
&y = v_7 = 11.652
\end{aligned}
$$

Forward AD trace

$$
\begin{aligned}
&\dot{v_1} = 1 \\
&\dot{v_2} = 0 \\
&\dot{v_3} = \dot{v_1}/v_1 = 0.5 \\
&\dot{v_4} = \dot{v_1} v_2 + \dot{v_2} v_1 = 1 \times 5 + 0 \times 2 = 5 \\
&\dot{v_5} = \dot{v_2} \cos v_2 = 0 \times \cos 5 = 0 \\
&\dot{v_6} = \dot{v_3} + \dot{v_4} = 0.5 + 5 = 5.5 \\
&\dot{v_7} = \dot{v_6} - \dot{v_5} = 5.5 - 0 = 5.5
\end{aligned}
$$

Now we have $\frac{\partial y}{\partial x_1} = \dot{v_7} = 5.5$

Limitation of forward mode AD

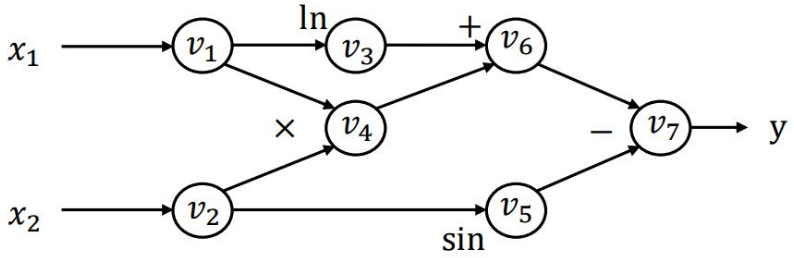
For $f: \mathbb{R}^n \to \mathbb{R}^k$, we need n forward AD passes to get the gradient with respect to each input.

We mostly care about the cases where $k = 1$ and large n .

In order to resolve the problem efficiently, we need to use another kind of AD.

Reverse mode automatic differentiation(AD)

$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$



Forward evaluation trace

$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times \ v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

Define adjoint $\overline{v_i} = \frac{\partial y}{\partial v_i}$

We can then compute the $\overline{v_i}$ iteratively in the **reverse** topological order of the computational graph

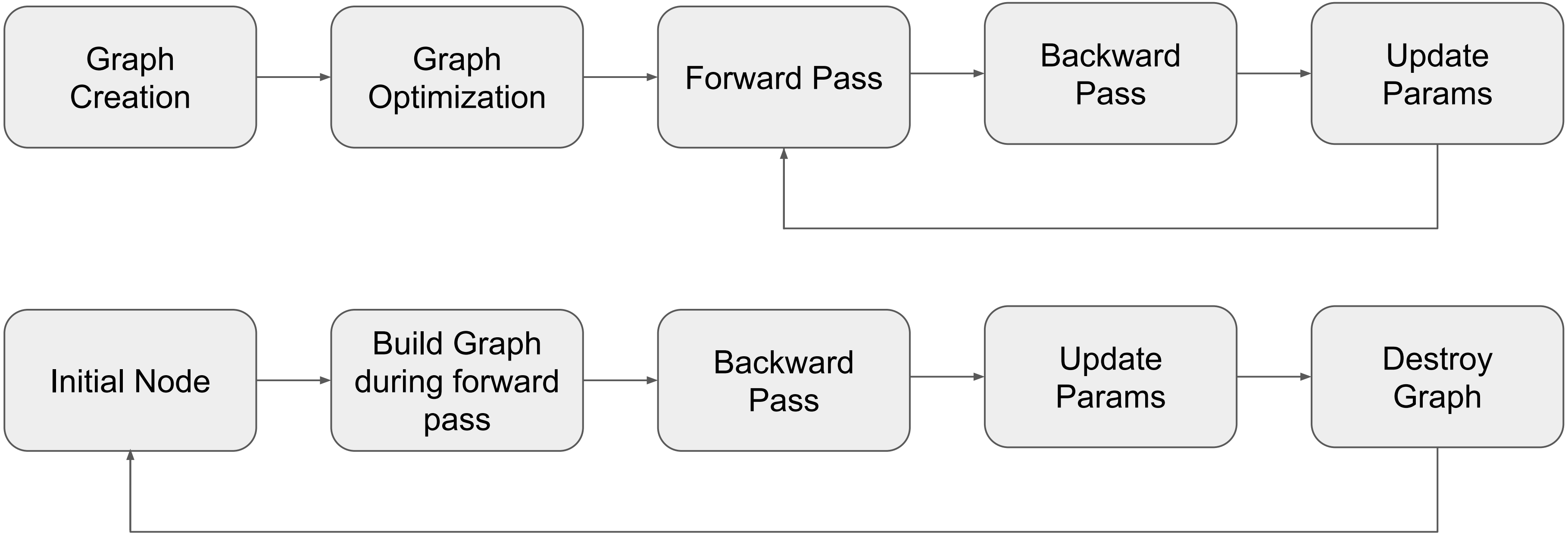
Reverse AD evaluation trace

$$
\begin{aligned}
\overline{v_7} &= \frac{\partial y}{\partial v_7} = 1 \\
\overline{v_6} &= \overline{v_7}\frac{\partial v_7}{\partial v_6} = \overline{v_7} \times 1 = 1 \\
\overline{v_5} &= \overline{v_7}\frac{\partial v_7}{\partial v_5} = \overline{v_7} \times (-1) = -1 \\
\overline{v_4} &= \overline{v_6}\frac{\partial v_6}{\partial v_4} = \overline{v_6} \times 1 = 1 \\
\overline{v_3} &= \overline{v_6}\frac{\partial v_6}{\partial v_3} = \overline{v_6} \times 1 = 1 \\
\overline{v_2} &= \overline{v_5}\frac{\partial v_5}{\partial v_2} + \overline{v_4}\frac{\partial v_4}{\partial v_2} = \overline{v_5} \times \cos v_2 + \overline{v_4} \times v_1 = -0.284 + 2 = 1.716 \\
\overline{v_1} &= \overline{v_4}\frac{\partial v_4}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1} = \overline{v_4} \ \times v_2 + \ \overline{v_3}\ \frac{1}{v_1} = 5 + \frac{1}{2} = 5.5
\end{aligned}
$$

Dynamic vs Static Graphs

- Static graphs:
  - Built before execution (forward pass)
  - Pros
    - Easier to Optimize
    - Better Performance
  - Cons
    - Less flexible
    - Harder to debug

Dynamic vs Static Graphs



Dynamic vs Static Graphs

- Dynamic graphs:
  - Built during execution (forward pass)
  - Pros
    - More flexible
    - Easy to debug
  - Cons
    - Slower than static graphs

Dynamic vs Static Graphs

Frameworks using Static Graphs:

- TensorFlow 1.x
- Caffe

Frameworks using Dynamic Graphs:

- PyTorch
- TensorFlow 2.x
- JAX

Additional Readings

- https://dlsyscourse.org/
- https://github.com/jax-ml/jax
- Baydin, Atilim Gunes, et al. "Automatic differentiation in machine learning: a survey." Journal of Machine Learning Research 18 (2018): 1-43.

Code Walkthrough