Computing Derivatives & Autograd

11485/685/785 Fall 2023: RECITATION 3

Dheeraj Pai and Miya Sylvester

September 15, 2023

Logistics and deadlines

- Sept 15: AWS Credit form
- Sept 15: Kaggle Setup Form - Piazza @82
- Sept 23: HW1P1 and HW1P2
- Study groups mentors - Piazza @346 (Change in group - check @346)

Recap from Lecture

- Neural Networks (NN) consist of parameters, mainly weights W and biases b.

Recap from Lecture

- Neural Networks (NN) consist of parameters, mainly weights W and biases b.
- Update Rule: $W = W - \eta \frac{dL}{dW}$

Recap from Lecture

- Neural Networks (NN) consist of parameters, mainly weights W and biases b.
- Update Rule: $W = W - \eta \frac{dL}{dW}$
- How do we compute derivatives $\frac{dL}{dW}$?

Table of Contents

1. Differentiation methods
2. Automatic differentiation
3. Automatic differentiation Libraries
4. HW1P1 Autograd

Symbolic Differentiation

Given Function

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

Symbolic Differentiation

Given Function

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

Goal

Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}$ (at the point x_1, x_2).

Symbolic Differentiation

Given Function

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

Goal

Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}$ (at the point x_1, x_2).

Partial Derivatives

Symbolic Differentiation

Given Function

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

Goal

Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}$ (at the point x_1, x_2).

Partial Derivatives

$$\frac{dy}{dx_1} = \frac{1}{x_1} + x_2$$

Symbolic Differentiation

Given Function

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

Goal

Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}$ (at the point x_1, x_2).

Partial Derivatives

$$\frac{dy}{dx_1} = \frac{1}{x_1} + x_2$$

$$\frac{dy}{dx_2} = x_1 - \cos(x_2)$$

Substitute the values of x_1, x_2

Numerical Differentiation

Function

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

Numerical Differentiation

Function

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

- Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}$ (at the point x_1, x_2)

Numerical Differentiation

Function

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

- Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}$ (at the point x_1, x_2)

Procedure

- Use the limit formula:

$$\lim_{h \to 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- Select a small h (10^{-5})

Numerical Differentiation

Function

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

- Given x_1, x_2 as inputs, calculate $\frac{\partial y}{\partial x_1}$ (at the point x_1, x_2)

Procedure

- Use the limit formula:

$$\lim_{h \to 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- Select a small h (10^{-5})
- Substitute the values of x_1, x_2, h

Questions?

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \text{MLP}(W, b, x)$$

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \mathrm{MLP}(W, b, x)$$

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \mathrm{MLP}(W, b, x)$$

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- What happens when we have 1 billion parameters?

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \mathrm{MLP}(W, b, x)$$

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- What happens when we have 1 billion parameters?
- Leads to 1 billion equations

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \mathrm{MLP}(W, b, x)$$

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- What happens when we have 1 billion parameters?
- Leads to 1 billion equations
- Each equation involves nearly 1 billion floating-point operations

Why Not Use Symbolic Differentiation for NNs?

Function Model

$$y = \text{MLP}(W, b, x)$$

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- What happens when we have 1 billion parameters?
- Leads to 1 billion equations
- Each equation involves nearly 1 billion floating-point operations
- Roughly 1×10^{18} FLOPS for a single gradient update

Why Not Use Numerical Differentiation?

$$y = \mathrm{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Why Not Use Numerical Differentiation?

$$y = \mathrm{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- Numerical instability and inaccuracy. How do you choose the most appropriate h?

Why Not Use Numerical Differentiation?

$$y = \mathrm{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- Numerical instability and inaccuracy. How do you choose the most appropriate h?
- What happens when we have 1 billion parameters?

Why Not Use Numerical Differentiation?

$$y = \text{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- Numerical instability and inaccuracy. How do you choose the most appropriate h?
- What happens when we have 1 billion parameters?
- Leads to 1 billion equations

Why Not Use Numerical Differentiation?

$$y = \text{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- Numerical instability and inaccuracy. How do you choose the most appropriate h?
- What happens when we have 1 billion parameters?
- Leads to 1 billion equations
- Each equation involves nearly 1 billion floating-point operations

Why Not Use Numerical Differentiation?

$$y = \mathrm{MLP}(W, b, x)$$

Goal

- Given x, W, b as inputs, calculate $\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial W}, \frac{\partial y}{\partial b}$ (at the point x, W, b)

Challenges

- Numerical instability and inaccuracy. How do you choose the most appropriate h?
- What happens when we have 1 billion parameters?
- Leads to 1 billion equations
- Each equation involves nearly 1 billion floating-point operations
- Roughly 1×10^{18} FLOPS for a single gradient update

Table of Contents

1. Differentiation methods
2. Automatic differentiation
3. Automatic differentiation Libraries
4. HW1P1 Autograd

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g(x))$$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$ from $\frac{dh}{dg}$ and $\frac{dg}{dx}$?

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$ from $\frac{dh}{dg}$ and $\frac{dg}{dx}$?

Chain Rule

- Decompose complex derivatives into simpler parts.
- $\frac{dY}{dx} = \frac{dh}{dg} \times \frac{dg}{dx}$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$ from $\frac{dh}{dg}$ and $\frac{dg}{dx}$?

Chain Rule

- Decompose complex derivatives into simpler parts.
- $\frac{dY}{dx} = \frac{dh}{dg} \times \frac{dg}{dx}$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g_1(x), g_2(x))$$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g_1(x), g_2(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$?

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g_1(x), g_2(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$?

Chain Rule

- Decompose complex derivatives into simpler parts.
- $\frac{dY}{dx} = \frac{dh}{dg_1} \times \frac{dg_1}{dx} + \frac{dh}{dg_2} \times \frac{dg_2}{dx}$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x) = h(g_1(x), g_2(x))$$

Key Question

- Can we compute $\frac{dY}{dx}$?

Chain Rule

- Decompose complex derivatives into simpler parts.
- $\frac{dY}{dx} = \frac{dh}{dg_1} \times \frac{dg_1}{dx} + \frac{dh}{dg_2} \times \frac{dg_2}{dx}$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x_1, x_2, \dots, x_n) = h(g_1(x_1, x_2, \dots, x_n), g_2(x_1, x_2, \dots, x_n))$$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n))$$

Key Question

- Can we compute $\frac{dY}{dx_1} \ldots \frac{dY}{dx_n}$

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n))$$

Key Question

- Can we compute $\frac{dY}{dx_1} \ldots \frac{dY}{dx_n}$
- Answer: Yes.

Basic Idea - Chain rule

Function Decomposition

$$Y = f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n))$$

Key Question

- Can we compute $\frac{dY}{dx_1} \ldots \frac{dY}{dx_n}$
- Answer: Yes.

Chain Rule

- Decompose complex derivatives into simpler parts.

Autodiff Example

Decompose with intermediate variables

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

$$v_1 = x_1$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$v_1 = x_1$$
$$v_2 = x_2$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \ln(v_1) \end{aligned}$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \ln(v_1) \\ v_4 &= v_1 v_2 \end{aligned}$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \ln(v_1) \\ v_4 &= v_1 v_2 \\ v_5 &= \sin(v_2) \end{aligned}$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4
\end{aligned}$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Decompose with intermediate variables

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

$$\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y &= v_7 = v_6 - v_5
\end{aligned}$$

Examples adapted from: `https://dlsyscourse.org/slides/4-automatic-differentiation.pdf`

Autodiff Example

Intermediate Variables

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \ln(v_1) \\ v_4 &= v_1 v_2 \\ v_5 &= \sin(v_2) \\ v_6 &= v_3 + v_4 \\ y &= v_7 = v_6 - v_5 \end{aligned}$$

Can you compute all the derivatives $\frac{dy}{dv_i}$**?** By looking at only one equation at a time.

Questions?

Autodiff Example - Graph computation

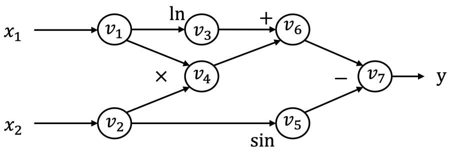
Intermediate Variables

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y &= v_7 = v_6 - v_5
\end{aligned}
$$

Autodiff Example - Graph computation

Intermediate Variables

$$v_1 = x_1$$
$$v_2 = x_2$$
$$v_3 = \ln(v_1)$$
$$v_4 = v_1 v_2$$
$$v_5 = \sin(v_2)$$
$$v_6 = v_3 + v_4$$
$$y = v_7 = v_6 - v_5$$



Autodiff Computation (Advantage)

Complex Model

$$y = \text{MLP}(W, b, x)$$

Computational Cost

- 1 Billion parameters

Autodiff Computation (Advantage)

Complex Model

$$y = \mathrm{MLP}(W, b, x)$$

Computational Cost

- 1 Billion parameters
- Forward pass $= 1B$ FLOPS

Autodiff Computation (Advantage)

Complex Model

$$y = \text{MLP}(W, b, x)$$

Computational Cost

- 1 Billion parameters
- Forward pass $= 1B$ FLOPS
- Backward pass $= 1B$ FLOPS

Total Cost

$$\text{Total} = 2B \text{ FLOPS} \lll 1e+18$$

Question

When is automatic differentiation a bad idea ?

Question

When is automatic differentiation a bad idea ?

Hint: Think about the memory to store the variables. What if you need the derivative of just one variable

Question

When is symbolic differentiation a good idea ?

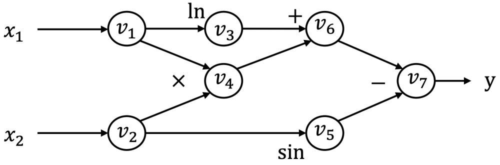
Question

When is symbolic differentiation a good idea ?

Hint: Integrals?

Automatic Differentiation in Practice

- Generate the computational graph while forward propagation
- Store each intermediate variables
- Chain rule to compute the derivatives.
- Optimize, parallelize based on variable dependencies
- Many more additional blocks - Accumulate Grad etc.



Images from https://dlsyscourse.org/slides/4-automatic-differentiation.pdf

Questions?

Table of Contents

1 Differentiation methods

2 Automatic differentiation

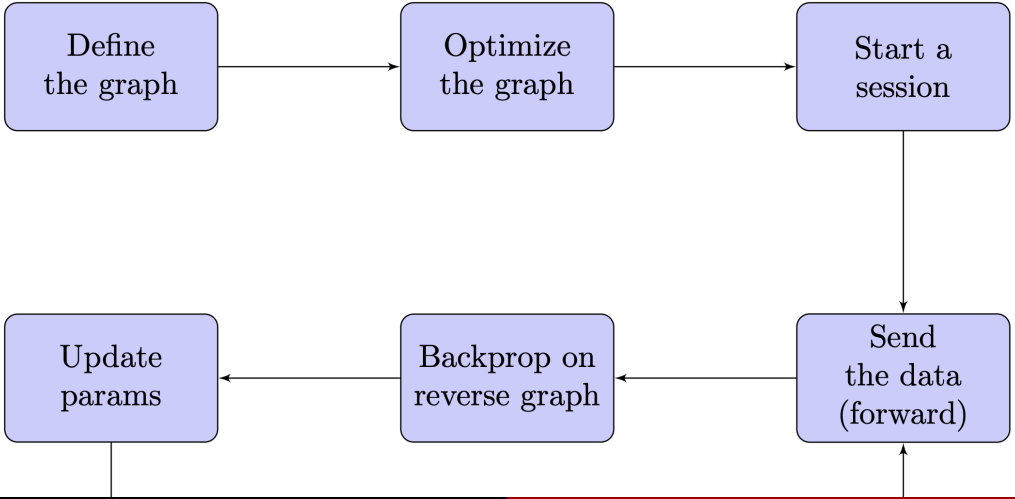
3 Automatic differentiation Libraries

4 HW1P1 Autograd

Automatic Differentiation Libraries

Static Vs Dynamic Graphs execution

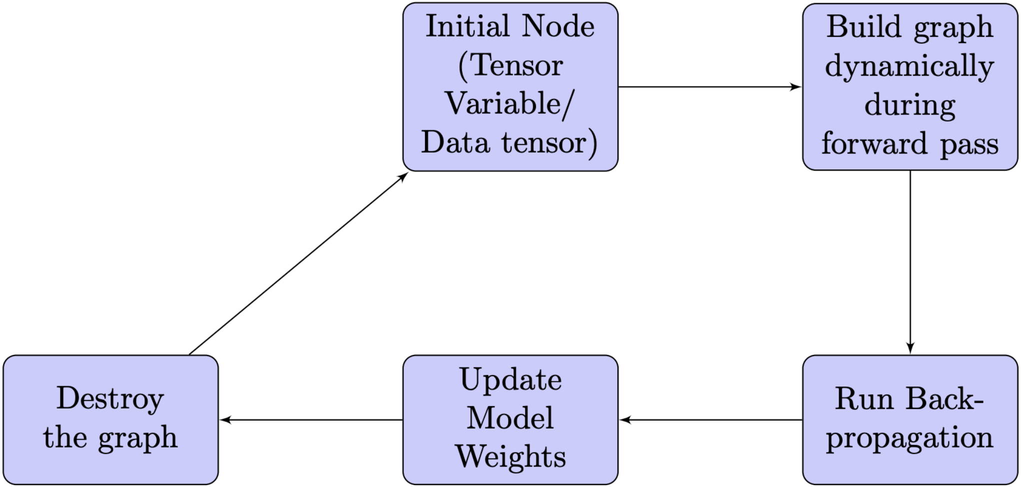
Static Graph Automatic Differentiation Execution



Optimization Techniques

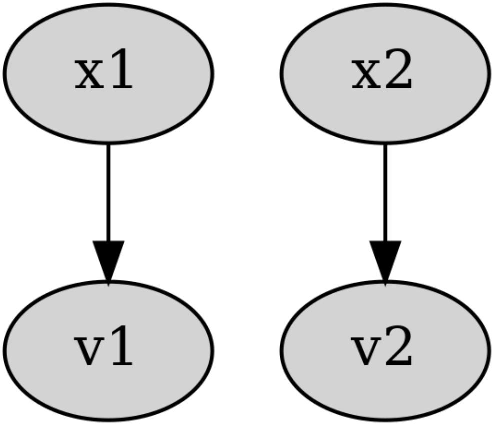
- Dead Code Elimination
- Common Subexpression Elimination
- Operator Fusion
- Memory Optimization
- Graph Pruning
- Kernel Fusion
- Data Layout Optimization
- Batching Optimization
- Pipeline Optimization
- Device Placement

Dynamic graph Automatic differentiation execution



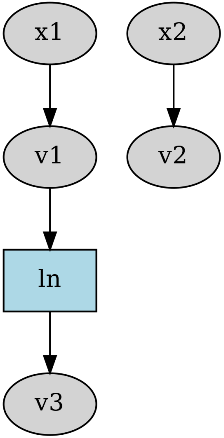
Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y = v_7 &= v_6 - v_5
\end{aligned}
$$



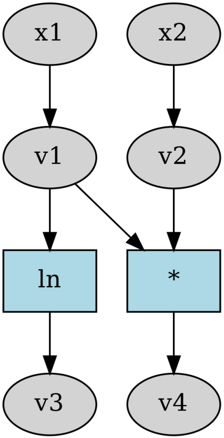
Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y = v_7 &= v_6 - v_5
\end{aligned}
$$



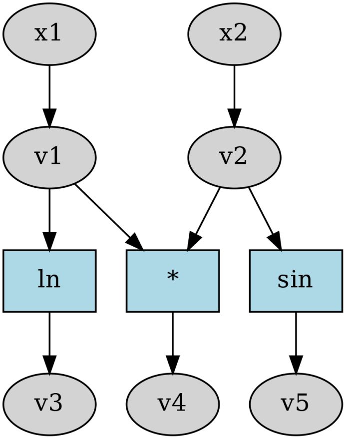
Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y = v_7 &= v_6 - v_5
\end{aligned}
$$



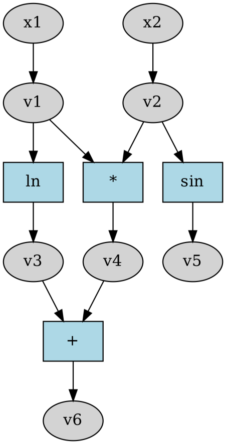
Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y = v_7 &= v_6 - v_5
\end{aligned}
$$



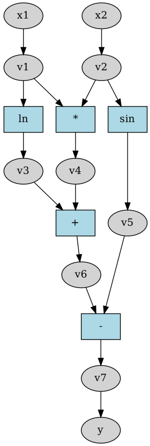
Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y &= v_7 = v_6 - v_5
\end{aligned}
$$



Dynamic Graph execution

$$
\begin{aligned}
v_1 &= x_1 \\
v_2 &= x_2 \\
v_3 &= \ln(v_1) \\
v_4 &= v_1 v_2 \\
v_5 &= \sin(v_2) \\
v_6 &= v_3 + v_4 \\
y = v_7 &= v_6 - v_5
\end{aligned}
$$



Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**
  - Flexible and intuitive for researchers

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**
  - Flexible and intuitive for researchers
  - Easier debugging

Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**
  - Flexible and intuitive for researchers
  - Easier debugging
  - Slower performance

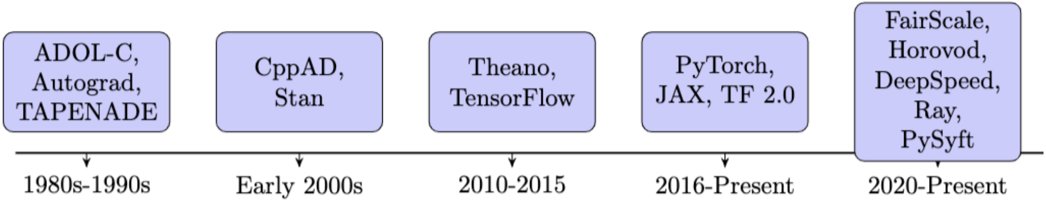
Static graph vs Dynamic graph

- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**
  - Flexible and intuitive for researchers
  - Easier debugging
  - Slower performance
  - Higher memory consumption

Static graph vs Dynamic graph

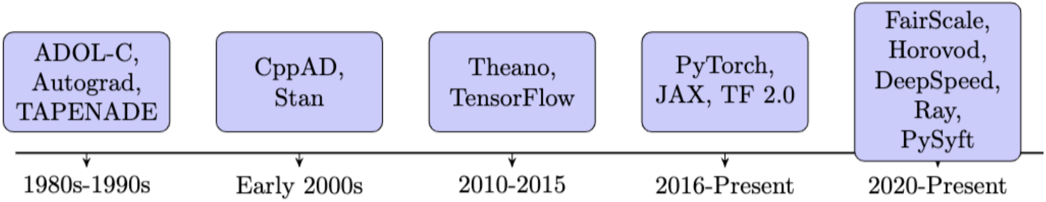
- **Static graphs:**
  - Optimization-friendly
  - Easy parallelization
  - Hard to debug (compiled graph)
  - Limited flexibility for conditionals
- **Dynamic graphs:**
  - Flexible and intuitive for researchers
  - Easier debugging
  - Slower performance
  - Higher memory consumption

History of Autodiff Libraries



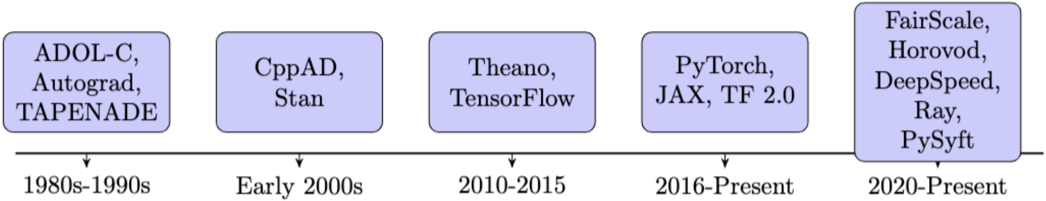
- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs

History of Autodiff Libraries



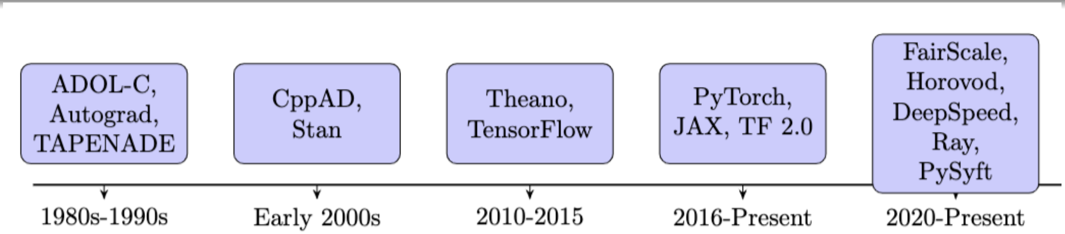
- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs
- 2010 - 2018: Python wrappers for other C/CUDA/Lua libraries (GPU support)

History of Autodiff Libraries



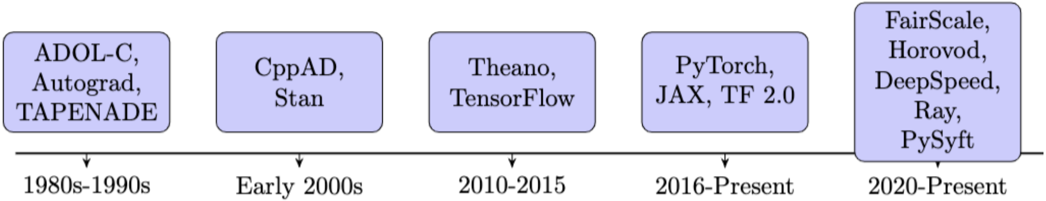
- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs
- 2010 - 2018: Python wrappers for other C/CUDA/Lua libraries (GPU support)
- 2015 - TF/Pytorch - C/Cuda DL focused optimizations.

History of Autodiff Libraries



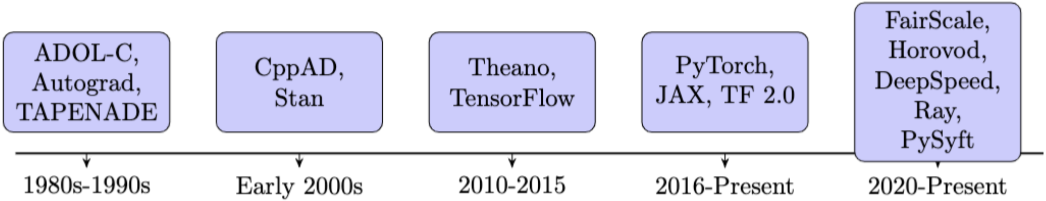
- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs
- 2010 - 2018: Python wrappers for other C/CUDA/Lua libraries (GPU support)
- 2015 - TF/Pytorch - C/Cuda DL focused optimizations.
- 2016 - Pytorch - dynamic graphs

History of Autodiff Libraries



- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs
- 2010 - 2018: Python wrappers for other C/CUDA/Lua libraries (GPU support)
- 2015 - TF/Pytorch - C/Cuda DL focused optimizations.
- 2016 - Pytorch - dynamic graphs
- 2018 - JAX - duck typing of numpy, TF 2.0

History of Autodiff Libraries



- 1980s - 1990s: Mostly C/FORTRAN based. Static graphs
- 2010 - 2018: Python wrappers for other C/CUDA/Lua libraries (GPU support)
- 2015 - TF/Pytorch - C/Cuda DL focused optimizations.
- 2016 - Pytorch - dynamic graphs
- 2018 - JAX - duck typing of numpy, TF 2.0
- 2020 - Fairscale, Deepspeed, NCLL, Pytorch distributed

Evolution of Autodiff Libraries in 2010s and 2020s

- **2010s: diverging philosophies**
  - Static graphs: TensorFlow 1.x, Caffe - industry
  - Dynamic graphs: PyTorch 0.x - research

Evolution of Autodiff Libraries in 2010s and 2020s

- **2010s: diverging philosophies**
  - Static graphs: TensorFlow 1.x, Caffe - industry
  - Dynamic graphs: PyTorch 0.x - research
- **2020s (PyTorch $\approx$ TensorFlow):**
  - PyTorch:
    - Optimizations for nn.Modules
    - Torch compile
    - Optimized with caching
  - TensorFlow:
    - Eager execution

Evolution of Autodiff Libraries in 2010s and 2020s

- **2010s: diverging philosophies**
  - Static graphs: TensorFlow 1.x, Caffe - industry
  - Dynamic graphs: PyTorch 0.x - research
- **2020s (PyTorch $\approx$ TensorFlow):**
  - PyTorch:
    - Optimizations for nn.Modules
    - Torch compile
    - Optimized with caching
  - TensorFlow:
    - Eager execution
  - Industry:
    - Distributed training: Fairscale, Deepspeed,

[Rajbhandari, Samyam, et al. "ZeRO: Memory optimizations toward training trillion parameter models."]

Distributed AD: PyTorch, FairScale & DeepSpeed

- **PyTorch:**
  - Native support for data parallelism, AccumulateGrad
  - Distributed Data Parallel (DDP) for multi-GPU training
- **FairScale:**
  - Optimization techniques like ZeRO, Sharded DDP (Sharded models)
  - Memory vs communication optimizations
- **DeepSpeed:**
  - Specialized for very large models (100B+ parameters)
  - ZeRO-3 for extreme memory efficiency
  - Pipeline parallelism for layer distribution
- **Common Features:**
  - Activation Checkpointing
  - Sharded models, distributed training, compensate compute and network against memory

References

- https://dlsyscourse.org/
- https://en.wikipedia.org/wiki/Automatic_differentiation
- https://deeplearning.cs.cmu.edu/S23/document/recitation/Recitation3/s23_Recitation_3_AutoDiff__Backprop.pdf
- Fairscale: https://github.com/facebookresearch/fairscale
- Baydin, Atilim Gunes, et al. "Automatic differentiation in machine learning: a survey." Journal of Marchine Learning Research 18 (2018): 1-43.

Table of Contents

1 Differentiation methods

2 Automatic differentiation

3 Automatic differentiation Libraries

4 HW1P1 Autograd

HW1P1 Autograd Overview

- Create an Automatic Differentiation library using Numpy.

HW1P1 Autograd Overview

- Create an Automatic Differentiation library using Numpy.
- Implement an engine *Autograd* that stores every operation in sequence (equivalent to a computation graph!).

HW1P1 Autograd Overview

- Create an Automatic Differentiation library using Numpy.
- Implement an engine *Autograd* that stores every operation in sequence (equivalent to a computation graph!).
- Build activations, losses, layers using primitive operations.

HW1P1 Autograd Overview

- Create an Automatic Differentiation library using Numpy.
- Implement an engine *Autograd* that stores every operation in sequence (equivalent to a computation graph!).
- Build activations, losses, layers using primitive operations.
- Run MLP using your Autograd engine.

HW1P1 Autograd Overview

- Create an Automatic Differentiation library using Numpy.
- Implement an engine *Autograd* that stores every operation in sequence (equivalent to a computation graph!).
- Build activations, losses, layers using primitive operations.
- Run MLP using your Autograd engine.
- 50 Marks (Largest HW part 1 Bonus)

Some Tips for Autograd HW

- You don't need to implement the graph for HW; a Python list will do the job.

Some Tips for Autograd HW

- You don't need to implement the graph for HW; a Python list will do the job.
- Explicitly add operations and nodes to the list.

Some Tips for Autograd HW

- You don't need to implement the graph for HW; a Python list will do the job.
- Explicitly add operations and nodes to the list.
- Define minimal primitive backward functions - `mul_backward`, `matmul_backward`, etc.

Some Tips for Autograd HW

- You don't need to implement the graph for HW; a Python list will do the job.
- Explicitly add operations and nodes to the list.
- Define minimal primitive backward functions - `mul_backward`, `matmul_backward`, etc.
- Extremely simple if you read the writeup and all the comments in the handout.

File Structure

- Build these files on their own first, and then put it into your Part 1 homeworks, if you want

2 MyTorch Structure

In HW1P1, your implementation of MyTorch worked at the granularity of a single layer - thus, stacking several Linear Layers followed by activations (and, optionally, BatchNorm) allowed you to build your very own MLP. In this bonus assignment, we will build an alternative implementation of MyTorch based on a popular Automatic Differentiation framework called Autograd that works at the granularity of a single operation. As you will discover, this alternate implementation more closely resembles the internal working of popular Deep Learning frameworks such as PyTorch and TensorFlow (version 2.0 onwards), and offers more flexibility in building arbitrary network architectures. For Homework 1 Bonus, MyTorch will have the following structure:

handout

- `mytorch/`
  - `autograd_engine.py`
  - `utils.py`
  - `sandbox.py`
  - `nn/`
    - `functional.py`
    - `modules/`
      - `activation.py`
      - `linear.py`
      - `loss.py`
- `hw1_bonus/`
  - `data`
  - `mlp.py`
  - `mlp_runner.py`
- `autograder/`
  - `runner.py`
  - `helpers.py`
  - `test_activation.py`
  - `test_autograd.py`
  - `test_functional.py`
  - `test_linear.py`
  - `test_loss.py`
- `create_tarball.sh`

Autograd Implementation

- Take advantage of ordering
- Backpropagation is to iterate backwards on an operation list

3 Autograd

3.1 Background : Automatic Differentiation

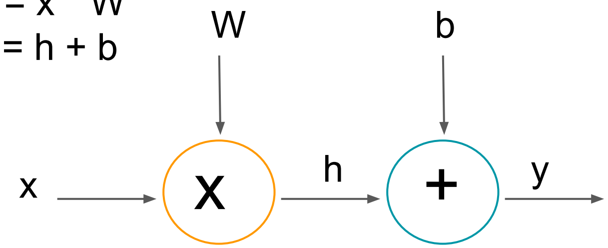
Automatic Differentiation [1], or "Autodiff", is a framework that allows us to calculate the derivatives of any arbitrarily complex mathematical function. It does so by repeatedly applying the chain rule of differentiation since all computer functions can be rewritten in the form of nested differentiable operations. Autodiff, which is different from Symbolic Differentiation, and Numerical Differentiation, has several desirable properties: two that we care most about are computational efficiency, and numerical accuracy. In practice, there are several different ways to implement autodiff, which can be broadly categorised into two types - forward accumulation, or forward mode (which computes the derivatives of the chain rule from inside to outside) and reverse accumulation, or reverse mode (which computes the derivatives of the chain rule from outside to inside). "Autograd" is just one such implementation of reverse mode automatic differentiation, which is most widely used in the context of machine learning applications.

3.2 Autograd and Backprop

Recall from Lecture 2, "The neural network as a universal approximator", that neural networks are just large, large functions. Also recall from Lecture 3, that in order to train a neural network, we need to calculate the derivatives (or gradients) of this large function (with respect to its inputs) - which is the backpropagation algorithm - and use these gradients in an optimisation algorithm such as gradient descent to update the parameters of the network. Finally, recall from earlier in this writeup that autodiff provides an efficient way to compute exactly these required gradients by repeatedly applying the chain rule. The Autograd framework keeps track of the sequence of operations that are performed on the input data leading up to the final loss calculation. It then performs backpropagation and calculates all the necessary gradients.

Autograd Implementation

h = x * W
y = h + b

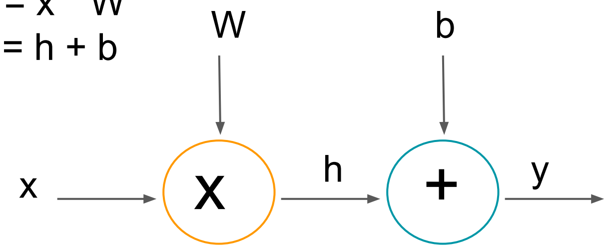


3.3 Implementation Details

While several popular implementations of Autograd deal with complex data structures (such as computational graphs), our implementation will be far simpler and resemble a single "linear" sequence of operations going forward and backward. This is based on the key observation that regardless of the actual network architecture that one constructs (a graph, or otherwise) there is a sequential order in which all operations can be performed in order to achieve the correct result. (Readers who have a CS background may draw an analogy with the concept of serialized transactions/operations in distributed/parallel computing). We break down our implementation into two main classes - the `Operation`, and the `Autograd` classes - and a single helper class (`GradientBuffer`).

Autograd Implementation

h = x * W
y = h + b



3.3 Implementation Details

While several popular implementations of Autograd deal with complex data structures (such as computational graphs), our implementation will be far simpler and resemble a single "linear" sequence of operations going forward and backward. This is based on the key observation that regardless of the actual network architecture that one constructs (a graph, or otherwise) there is a sequential order in which all operations can be performed in order to achieve the correct result. (Readers who have a CS background may draw an analogy with the concept of serialized transactions/operations in distributed/parallel computing). We break down our implementation into two main classes - the `Operation`, and the `Autograd` classes - and a single helper class (`GradientBuffer`).

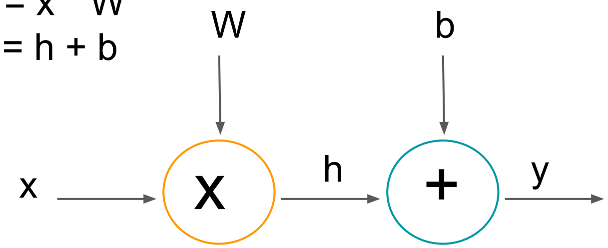
$$\frac{dL}{dh} = \frac{dL}{dy}\frac{dy}{dx}$$

$$\frac{dL}{db} = \frac{dL}{dy}\frac{dy}{db}$$

$$\frac{dL}{dh} \quad \frac{dL}{db} \quad \frac{dL}{dy}$$

Autograd Implementation

h = x * W
y = h + b

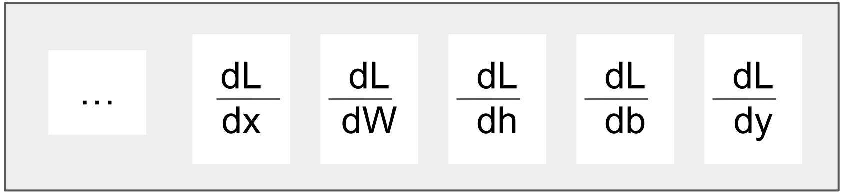


$$\frac{dL}{dx} = \frac{dL}{dh}\frac{dh}{dx}$$

$$\frac{dL}{dW} = \frac{dL}{dh}\frac{dh}{dW}$$

$$\frac{dL}{dh} = \frac{dL}{dy}\frac{dy}{dx}$$

$$\frac{dL}{db} = \frac{dL}{dy}\frac{dy}{db}$$



3.3 Implementation Details

While several popular implementations of Autograd deal with complex data structures (such as computational graphs), our implementation will be far simpler and resemble a single "linear" sequence of operations going forward and backward. This is based on the key observation that regardless of the actual network architecture that one constructs (a graph, or otherwise) there is a sequential order in which all operations can be performed in order to achieve the correct result. (Readers who have a CS background may draw an analogy with the concept of serialized transactions/operations in distributed/parallel computing). We break down our implementation into two main classes - the `Operation`, and the `Autograd` classes - and a single helper class (`GradientBuffer`).

Operation Class

3.3.1 Operation Class

The objects of this class represent every operation that is performed in the network. Thus, for every operation that you perform on the data (say, multiplication, or addition), you will need to initialize a new `Operation` object that specifies the type of operation being performed. Note that to calculate the derivative of any operation in the network, we need to know the inputs that were passed to this node, and the outputs that were generated. Storing the type of operation, the inputs, and the outputs are the primary responsibilities of the `Operation` class.

Class attributes:

- `inputs`: The inputs to the operation.
- `outputs`: The output(s) generated by applying the operation to the inputs.
- `gradients_to_update`: These are the gradients corresponding to the operation inputs that must be updated on the backward pass.
- `backward_function`: A backward function implemented for a specific operation (ex: `add_backward` for operation add - see section 4.1.2 for more details). This function is called during backward pass to calculate and update the gradients for operation inputs.

Autograd Class

3.3.2 Autograd Class

This is the main class for autograd engine that is responsible for keeping track of the sequence of operations being performed, and kicking off the backprop algorithm once the forward pass is complete.

Class attributes:

- `gradient_buffer`: An instance of the `GradientBuffer` class, used to store a mapping between input data and their gradients.
- `operation_list`: A Python list that is used to store sequence of operations that are performed on the input data. Concretely, this stores `Operation` objects.

Class methods:

- `add_operation(inputs, output, gradients_to_update, backward_operation)`: Initialises a new instance of `Operation` with given arguments, and adds it to `operation_list`.
- `backward(divergence)`: Kicks off backpropagation. Traverses the `operation_list` in reverse and calculates the gradients at every node.

For this assignment, you will need to implement `add_operation`and `backward` methods.

GradientBuffer Class

For this assignment, you will need to implement `add_operation`and `backward` methods.

3.3.3 GradientBuffer Class

This is a simple wrapper class around a Python dictionary with a few useful methods that allow for storing and updating the gradients. While it's not necessary to modify this class for your assignment, we strongly recommend familiarizing yourself with the class attributes and methods to gain a better understanding of its functionality.

Class attributes:

- `memory`: A Python dictionary that holds the NumPy array corresponding to its gradient. For a given NumPy array `np_array`, the key is the memory location of `np_array` and the value is the gradient array associated with `np_array`. Note: Using the memory location as a key is a simple trick that eliminates the need to perform extra bookkeeping of maintaining unique keys for all gradients.

Class methods:

- `get_memory_loc(np_array)`: Returns the memory location of `np_array`, used in other functions to get keys.
- `is_in_memory(np_array)`: Checks if a gradient array corresponding to `np_array` is already in memory.
- `add_spot(np_array)`: Allocates a zero gradient array corresponding to `np_array` in `memory`.
- `update_param(np_array, gradient)`: Increments the gradient array corresponding to `np_array` by the amount of `gradient`.
- `get_param(np_array)`: Returns the gradient array corresponding to `np_array`.
- `clear()`: Clears the `memory` dictionary.

Example Walkthrough

3.4 Example Walkthrough

Suppose that we are building a single layer MLP. Therefore, we perform the following operations on input data x:

$$h = x * W \tag{1}$$
$$y = h + b \tag{2}$$

The following steps need to be executed for this simple operation:

- First, we need to create an instance of autograd engine using:

  ```
  autograd = autograd_engine.Autograd()
  ```

- For equation (1), we need to add a node to the computation graph performing multiplication, which would be done in the following way:

  ```
  autograd_engine.add_operation(
      inputs = [x, W], output = h,
      gradients_to_update = [None, dW],
      backward_operation = matmul_backward
  )
  ```

- Similarly for equation (2),

  ```
  autograd_engine.add_operation(
      inputs = [h, b], output = y,
      gradients_to_update = [None, db],
      backward_operation = add_backward
  )
  ```

- Invoke backpropagation by:

  ```
  autograd_engine.backward(divergence)
  ```

  `dW` and `db` should be updated after this.

The concept above could be leveraged in building more complex computation steps (with few lines of code).

Example Walkthrough

```python
import numpy as np
from mytorch.nn.functional import matmul_backward, add_backward

class Linear():
    def __init__(self, in_features, out_features, autograd_engine):
        self.W = np.random.uniform(-np.sqrt(1 / in_features), np.sqrt(1
                                   size=(out_features, in_features))  #
        self.b = np.random.uniform(-np.sqrt(1 / in_features), np.sqrt(1
                                   size=(out_features, 1))  # just chang
        self.dW = np.zeros(self.W.shape)
        self.db = np.zeros(self.b.shape)

        self.momentum_W = np.zeros(self.W.shape)
        self.momentum_b = np.zeros(self.b.shape)

        self.autograd_engine = autograd_engine

    def __call__(self, x):
        return self.forward(x)
```

3.4 Example Walkthrough

Suppose that we are building a single layer MLP. Therefore, we perform the following operations on input data x:

$$h = x * W \tag{1}$$

$$y = h + b \tag{2}$$

The following steps need to be executed for this simple operation:

stance of autograd engine using:

```
ngine.Autograd()
```

add a node to the computation graph performing multiplication, which g way:

```
peration(
output = h,
te = [None, dW],
n = matmul_backward
```

```
peration(
output = y,
te = [None, db],
backward_operation = add_backward
)
```

- Invoke backpropagation by:

```
autograd_engine.backward(divergence)
```

  dW and db should be updated after this.

The concept above could be leveraged in building more complex computation steps (with few lines of code).

Operation Class & Functional.py (& Sandbox.py)

```python
def add_backward(grad_output, a, b):
    a_grad = grad_output * np.ones(a.shape)
    b_grad = grad_output * np.ones(b.shape)
    return a_grad, b_grad


def sub_backward(grad_output, a, b):
    # TODO: implement the backward function for subtraction.
    raise NotImplementedError


def matmul_backward(grad_output, a, b):
    # TODO: implement the backward function for matrix product.
    raise NotImplementedError


def outer_backward(grad_output, a, b):
    assert a.shape[0] == 1 or a.ndim == 1
    assert b.shape[0] == 1 or b.ndim == 1
    # TODO: implement the backward function for outer product.
    raise NotImplementedError


def mul_backward(grad_output, a, b):
    # TODO: implement the backward function for multiply.
    raise NotImplementedError


def div_backward(grad_output, a, b):
    # TODO: implement the backward function for division.
    raise NotImplementedError

```

3.3.1 Operation Class

The objects of this class represent every operation that is performed in the network. Thus, for every operation that you perform on the data (say, multiplication, or addition), you will need to initialize a new `Operation` object that specifies the type of operation being performed. Note that to calculate the derivative of any operation in the network, we need to know the inputs that were passed to this node, and the outputs that were generated. Storing the type of operation, the inputs, and the outputs are the primary responsibilities of the `Operation` class.

Class attributes:

- `inputs`: The inputs to the operation.
- `outputs`: The output(s) generated by applying the operation to the inputs.
- `gradients_to_update`: These are the gradients corresponding to the operation inputs that must be updated on the backward pass.
- `backward_function`: A backward function implemented for a specific operation (ex: `add_backward` for operation add - see section 4.1.2 for more details). This function is called during backward pass to calculate and update the gradients for operation inputs.

No backward in Linear.py & Activation.py

```python
    def __call__(self, x):
        return self.forward(x)


    def forward(self, x):
        """
            Computes the affine transformation forward pass of the Linear Layer

            Args:
                - x (np.ndarray): the input array,

            Returns:
                - (np.ndarray), the output of this forward computation.
        """
        # TODO: Use the primitive operations to calculate the affine transformat
        #       of the linear layer
        # TODO: Remember to use add_operation to record these operations in
        #       the autograd engine after each operation

        # TODO: remember to return the computed value
        raise NotImplementedError
```

```python
class Identity(Activation):
    """
    Identity function (already implemented).
    """

    # This class is a gimme as it is already implemented for you

    def __init__(self, autograd_engine):
        super(Identity, self).__init__(autograd_engine)

    def forward(self, x):
        raise NotImplementedError


class Sigmoid(Activation):
    def __init__(self, autograd_engine):
        super(Sigmoid, self).__init__(autograd_engine)

    def forward(self, x):
        raise NotImplementedError


class Tanh(Activation):
    def __init__(self, autograd_engine):
        super(Tanh, self).__init__(autograd_engine)

    def forward(self, x):
        raise NotImplementedError


class ReLU(Activation):
    def __init__(self, autograd_engine):
        super(ReLU, self).__init__(autograd_engine)

    def forward(self, x):
        raise NotImplementedError
```

Loss.py & Pytorch

```python

# Hint: To simplify things you can just make a backward for this loss and not
# try to do it for every operation.
class SoftmaxCrossEntropy(LossFN):
    def __init__(self, autograd_engine):
        super(SoftmaxCrossEntropy, self).__init__(autograd_engine)

    def forward(self, y, y_hat):
        # TODO: calculate loss value and set self.loss_val
        # To simplify things, add a single operation corresponding to the
        # backward function created for this loss

        # self.loss_val = ...
        # return self.loss_val
        raise NotImplemented

```