**CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026**

# 3.Basics: autodiff, ML system architecture overview

*Lecturer: Hao Zhang*

*Scribe: Duan Wang, Jingwen Zhang, Wilson Sugiarto, Ka Sing He, Matthew Biehler, Atharva Hirulkar*

## 1 Just-in-Time (JIT) Compilation

### 1.1 Define-and-run vs. define-then-run

There are two main execution styles. In define-and-run, operations are written and executed immediately, which is convenient for development and debugging. In define-then-run, the computation graph is built first and executed afterward, which allows more optimization. JIT compilation tries to combine these two benefits: write code naturally like define-and-run, but still get the performance benefits of compilation.

### 1.2 What happens during JIT

Behind the scenes, JIT usually works in stages. First, the system captures the computation graph. Next, it lowers that graph into an intermediate representation that is easier to optimize. Finally, it compiles the graph into efficient code for the target backend. The goal is to improve execution speed without forcing the programmer to manually build a static graph.

### 1.3 Static graph requirement

The main issue with JIT is that it works best when the computation graph is static. A static graph has a fixed structure, so it can be optimized once and reused. If the graph changes depending on the input, then compilation becomes harder because the compiler cannot assume the same execution pattern every time.

## 2 Static vs. Dynamic Dataflow Graphs

### 2.1 Static dataflow graphs

A static dataflow graph is defined once, optimized once, and executed many times. After it is built, future executions follow the same structure. This makes optimization easier and usually gives better performance.

### 2.2 Dynamic dataflow graphs

A dynamic dataflow graph can change during execution depending on the input or control flow. This makes it more flexible, but also harder to implement, optimize, and debug. Because of this, systems usually prefer

to keep computations as static as possible.

### 2.3 LLMs: Static or Dynamic?

LLMs are mostly static during training because the computation pattern is largely regular. At inference time, they are more dynamic because the number of generated tokens can vary, so the amount of computation changes with the output length.

## 3 Ways to Handle Dynamic Behavior

### 3.1 Option 1: Skip JIT

One option is to avoid compiling dynamic graphs and just use define-and-run execution. This is simpler, but it gives up performance benefits.

### 3.2 Option 2: Add control flow operators

Another option is to add graph operators that represent control flow directly. This allows graphs to express conditions and branching, but it also makes the graph system more complicated.

### 3.3 Option 3: Piecewise compilation and guards

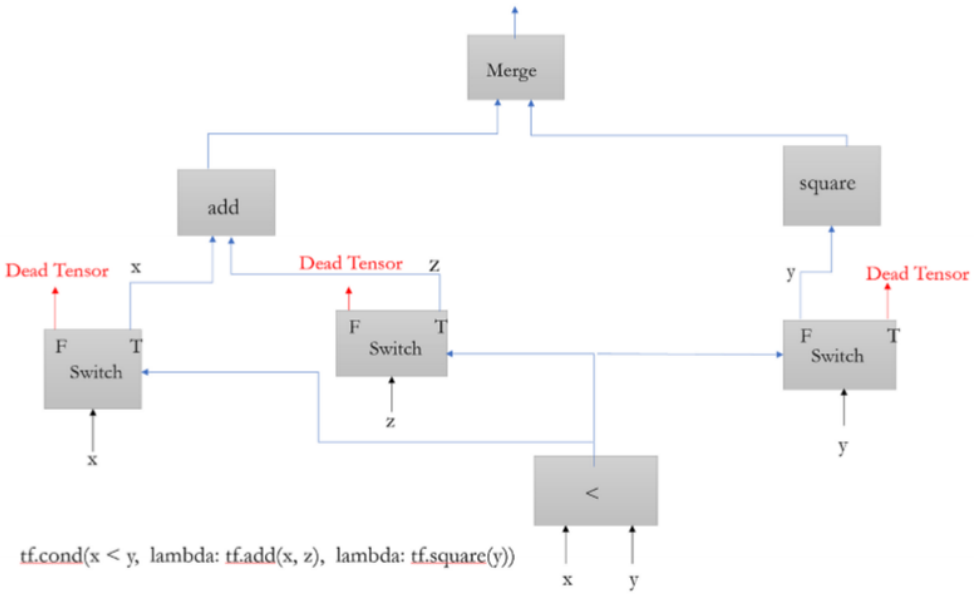
A more practical approach is piecewise compilation with guards. The system compiles parts of the program under certain assumptions, then switches or recompiles if those assumptions no longer hold. This is often the most successful approach for handling dynamic behavior.

## 4 Control Flow Primitives

### 4.1 Switch and Merge

To represent conditionals in a dataflow graph, frameworks can use primitives like Switch and Merge. A Switch takes a data tensor and a boolean condition, then sends real data down one branch and a dead tensor down the other. A Merge takes a data tensor and dead tensor and returns the data tensor.

### 4.2 Example




- Check compare operator, outputs true or false
- Switch nodes takes boolean output as input and outputs tensors appropriately
- if true then switches for the add node output data tensors, so add node will have data tensors to add together, while switch for square node receives a dead tensor.
- Else switch nodes will output dead tensors for the add node, while switch for square node will output a data tensor.
- Merge the 2 outputs from add and square nodes, one output will always be dead so output the data tensor.

### 4.3 Problems with Control Flow Primitives

Although control flow primitives make dynamic logic possible inside graphs, they also increase graph complexity. After much research and experimentation, it was decided that it was far too complicated without yielding worth-while efficiency improvements.

## 5 Piecewise Compilation

### 5.1 Variable input shapes

If a graph takes input of shape $[x, c_1, c_2]$ where $x$ is variable and the others are constant, compiling every possible value of $x$ would be too expensive. A practical solution is to compile a limited set of common cases, such powers of 2 of $x$. During runtime, we choose the closest one and pad to the closest power of 2.

### 5.2 Static–dynamic–static structure

Another case is when a model is static, then dynamic in one part, then static again such as MoE. In that situation, the system can focus on handling the dynamic part separately while still compiling the static parts normally. This is the idea behind piecewise compilation.

Piecewise compilation on these cases are the current state-of-the-art approaches today when people do training or inference.

## 6 Short Summary

So far, we have summarized frameworks, data flow graphs, and how people compile these data flow graphs. In the later session, we will talk about how people actually compile these graphs.

### 6.1 MCQ Exercises

1. You are a machine learning engineer at a company that is providing LLM endpoints to users. Your goal is running efficient inference for these LLMs, You are given a framework which has both symbolic and imperative APIs. While designing your system, would you:

    1. Use symbolic mode for both testing and deployment of your system.
    2. **Use imperative mode for development and symbolic mode for deployment.**
    3. Use symbolic mode for development and imperative mode for deployment.
    4. Use imperative mode for both testing and deployment of your system.

2. Which of the following is **not** true about dataflow graphs?

    1. Static dataflow graphs are defined once and executed many times.
    2. No extra effort is required for batching optimization of static dataflow graphs.
    3. **Dynamic dataflow graphs are easy to debug.**
    4. Define-and-run is a possible way to handle dynamic dataflow graphs.

## 7 Where we are

Designing ML system is basically handling problems with **data**, **model** , and **compute**. We have been going through how to express the **model**, where the first part is math primitives (mostly matmul). What we need is a representation that connects these math primitives into a graph.

In terms of this representation, we have done the forward path where each node is a primitive that are tensors, and the edges are data flow direction. What we need is the representation that expresses the backward computation using math primitives for training purpose.

# 8 AutoDiff

## 8.1 Intuition

Basically, we are taking derivatives of the loss against each variable in the data flow. i.e. Given $f(\theta)$, we want to find $\frac{\partial f}{\partial \theta}$. To systematically compute these derivatives, one intuitive way is to pick a small $\epsilon$, then the gradient can be approximated by

$$\frac{\partial f}{\partial \theta} = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta)}{\epsilon} \approx \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} + o(\epsilon^2)$$

However, this approach is not desirable. The first reason is that it is only an approximation which will induce errors. And the second reason is evaluating gradients in this way requires performing forward pass twice, which is very computationally heavy.

## 8.2 Symbolic Differentiation

Alternatively, we use symbolic differentiation, which is done by directly write down the formula and derive the gradient following PD rules:

$$\frac{\partial(f(\theta)+g(\theta))}{\partial \theta} = \frac{\partial f(\theta)}{\partial \theta} + \frac{\partial g(\theta)}{\partial \theta}$$

$$\frac{\partial(f(\theta)g(\theta))}{\partial \theta} = g(\theta)\frac{\partial f(\theta)}{\partial \theta} + f(\theta)\frac{\partial g(\theta)}{\partial \theta}$$

$$\frac{\partial(f(g(\theta)))}{\partial \theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)}\frac{\partial g(\theta)}{\partial \theta}$$

The problem is: How we efficiently derive gradients of each parameter in the data flow graph?

## 8.3 AutoDiff in data flow graph

In PA1, we need to implement a mini-TensorFlow by implementing the auto diff mechanism from scratch using a limited set of operators, and use the own framework to implement a language model. The model will be trained on the own framework and converge on 10 sentences and be able to generate tokens.

We use figure 1 as example to learn the autodiff rules. Our goal is to use PD and chain rules to calculate $\frac{\partial y}{\partial x_1}$.

There are 2 ways to apply the chain rule:

1. Forward: apply from left to right
2. Backward: apply from right to left

The backward direction fits with deep learning more than the forward direction. We will dive deeper on the two and see the reason behind.

# 9 Forward Mode AD

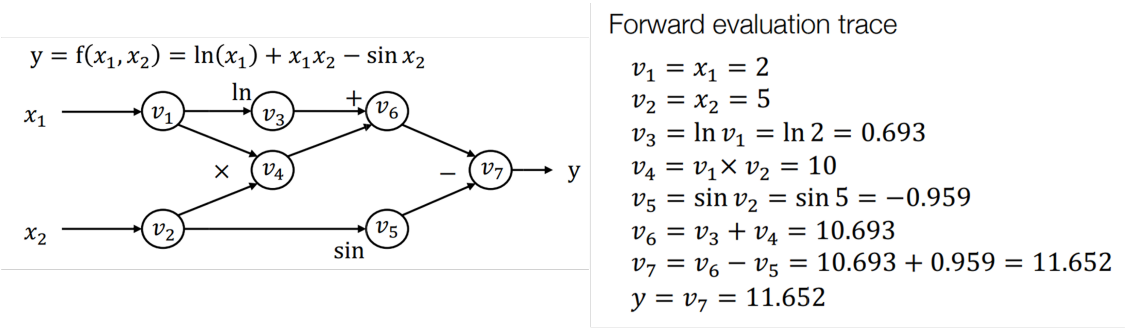



Figure 1: Data Flow Graph

The data flow graph is shown in Figure 1. The forward mode means that $\dot{v_i}$ can be derived as the partial derivative of $v_i$ against $x_1$ ($\dot{v_i} = \frac{\partial v_i}{\partial x_1}$). Each $\dot{v_i}$ is computed following the forward order of the graph from left to right. The compute trace is as follow:

1. First, $\dot{v_1}$ is defined as $\dot{v_1} = \frac{\partial v_1}{\partial x_1}$. According to the forward trace, $v_1 = x_1$, and then take the gradient. Therefore, $\dot{v_1}$ should be equal to 1.
2. For $\dot{v_2}$, by following the definition, $\dot{v_2} = \frac{\partial v_2}{\partial x_1}$. Because $v_2$ has low relation with $x_1$, $\dot{v_2}$ should be equal to 0.
3. By definition, $\dot{v_3} = \frac{\partial v_3}{\partial x_1}$. According to the relation between $v_3$ and $x_1$, $v_3 = \ln v_1$, where $v_1 = x_1$. Therefore, by taking the gradient gives $\dot{v_3} = \frac{\dot{v_1}}{v_1} = 0.5$.
4. Then, keep doing this from left to right following the forward trace, and try to figure out what is $\dot{v_i}$ for each of the intermediate checkpoint $v_i$.
5. Follow this trace, and eventually leads to the last variable $y$, where $y = v_7$. The equation is $\frac{\partial y}{\partial x_1} = \frac{\partial v_7}{\partial x_1}$. Since $\dot{v_1}$ to $\dot{v_6}$ have already been calculated, $\dot{v_7} = 5.5$ is obtained.

The forward auto differentiation lock on a specific input variable such as $x_1$, and keep deriving the gradient following the chain rule, all the way to the last variable, which is the target variable. It is the same for $x_2$.

To summarize this forward mode auto differentiation: Start from input nodes, in our case, $x_1$ and $x_2$; Then derive the gradient all the way to the output nodes. The target gradients $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are eventually obtained.

Pros and cons of forward mode auto differentiation: For a function f: $R^n \rightarrow R^k$, $n$ forward passes are required to get the gradient against each input, since we need to derive for each input variable ($x_1 ... \ x_n$). So why deep learning does not do forward mode auto differentiation? Because in deep learning, $n$ is very big, and $k$ is mostly equal to 1. We will need to perform $n$ forward passes to derive the gradients against every input dimension from $n$, which is bad. By the nature of deep learning, the input has high dimension, but there is only one scalar loss function. As a result, it is very heavy to do forward mode auto differentiation.

From a computational perspective, this explains why backpropagation is commonly used. Reverse-mode automatic differentiation derives gradients from the output back to the input. Reverse-mode AD exhibits the opposite characteristic: when the input dimension is very large but the output dimension is very small, the computational cost depends only on the output dimension.

# 10 Reverse Mode AD

Let us see how reverse auto differentiation works from the compute perspective. The same graph is used, as shown in Figure 1. Here, instead to directly deriving the intermediate gradient $\dot{v_i}$, the **adjoint** is derived. Adjoint is a component of the partial derivative, and sometimes equal to partial derivative. The notation $\overline{v_i}$ is used to denote the partial derivative of $\frac{\partial y}{\partial v_i}$. Adjoint values are derived from the end of the data flow graph all the way back to the start of the data flow graph. Each $\overline{v_i}$ in the reverse topological order of the graph is computed, which is so-called backward pass.

Here is the compute trace.

1. Start with the last node in the graph, which is $v_7$. By definition, $\overline{v_7} = \frac{\partial y}{\partial v_7}$. $y = v_7$, which is an identity function. Therefore, $\overline{v_7} = 1$.
2. Then, derive $\overline{v_6}$. According to the data flow graph, $v_6$ contributes to $v_7$ through the function "minus", $v_7 = v_6 - v_5$. Meanwhile, $y = v_7$. According to the chain rule, $\overline{v_6} = \overline{v_7}\frac{\partial y}{\partial v_6}$. Then substitute the values into the equation to get $\overline{v_6} = \overline{v_7} \times 1 = 1$.
3. Keep doing this using the reverse topological order, that is, derive $\overline{v_5}$, $\overline{v_4}$, $\overline{v_3}$, $\overline{v_2}$, $\overline{v_1}$. The data flow graph infers all the compute relation.
4. After traversing the entire reverse topological order, we eventually get $\overline{v_1} = \frac{\partial y}{\partial v_1} = 5.5$. That is the same value that we get in the forward mode AD.

Instead of directly compute gradient, adjoint are computed and propagated from the output node $n$ to the start. The gradient of the loss function $y$ against the inputs can be derived through one pass.

In this data flow graph, a small complication arises because an intermediate node may have multiple contributors, such as input nodes. The following case study illustrates this situation.

**Case study:**

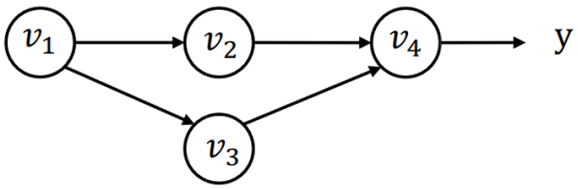



Figure 2: Case Study Data Flow Graph

According to the data flow graph in Figure 2, for the variable $v_4$, the variable $v_4$ receives two contributions from $v_2$ and $v_3$, while $v_2$ and $v_3$ all depends on a single input $v_1$. The objective is to derive $\overline{v_1}$. By definition $\overline{v_1} = \frac{\partial y}{\partial v_1}$. According to the chain rule,

$$\overline{v_1} = \frac{\partial y}{\partial v_1} = \frac{\partial f(v_2, v_3)}{\partial v_2}\frac{\partial v_2}{\partial v_1} + \frac{\partial f(v_2, v_3)}{\partial v_3}\frac{\partial v_3}{\partial v_1}. \tag{1}$$

Equation 1 derives the adjoint against each contributing pass, and then add them together. By simplifying this expression, equation 2 is obtained,

$$\overline{v_1} = \overline{v_2}\frac{\partial v_2}{\partial v_1} + \overline{v_3}\frac{\partial v_3}{\partial v_1}. \tag{2}$$

Real graph could be very complicated, where one variable could depend on multiple passes over other variables. In the general case,

$$\overline{v_i} = \sum_{j \in next(i)} \overline{v_{i \to j}}, \text{where } \overline{v_{i \to j}} = \overline{v_j} \frac{\partial v_j}{\partial v_i}. \tag{3}$$

That is, $\overline{v_i}$ is equally to the sum of all the partial adjoint of its future nodes $j$ that it has a contributing pass to. This equation 3 will be very important for us to implement the backward pass given arbitrary data flow graph.

To summarize the backward mode auto differentiation: The process starts from the output nodes, and derive the gradients all the way back to the input nodes.

The pros and cons are the reverse of the forward mode auto differentiation. For a function f: $R^n \to R^k$, $k$ backward passes are required to get the gradients with respect to each input node. This is good because $k = 1$ in machine learning (a single loss function). The gradient of the loss function against all the variables can be obtained in one backward pass.

## 11 Construct the Backward Graph

The computational algorithm of auto differentiation has now been established. The next step is to express this backward process using system language, which is the data flow graph. The goal is to use the backward mode AD to derive the gradients according to a given forward data flow graph, and to express the backward pass as another graph. The forward pass graph and the backward pass graph are then combined and sent to another lower-level optimizer. In this case, everything is symbolic and can be optimized. For whatever inputs, it can follow the graph computation to derive its forward and backward passes automatically.

Let us introduce an important program example that will be in the homework, as shown in Figure 3. From a high-level perspective, the loop is pretty simple: take gradients of an given output node against every intermediate variable. This algorithm precisely follows the reverse mode auto differentiation and the equations mentioned in section 10. At each node in the intermediate checkpoint, identify all subsequent nodes to which it contributes. The partial adjoints from these nodes are aggregated and summed to produce the node's final gradient. This gradient is then propagated backward to the preceding nodes according to the reverse topological ordering.

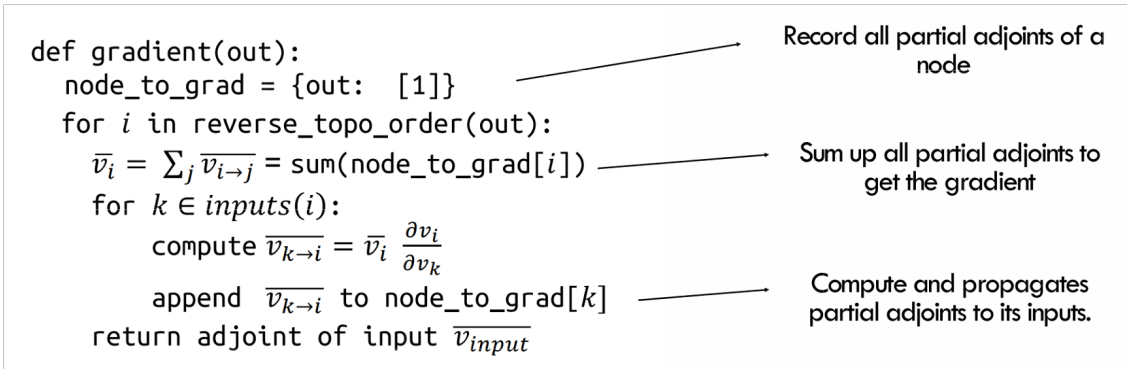



Figure 3: Reverse AutoDiff implementation

To parse the loop:

1. node_to_grad = {out : [1]}: First, create a dictionary, whose keys are the nodes name, and values are all the partial adjoint of the node.

2. $\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node to gradient[$i$]): Sum all the partial adjoint related to the node to get the gradient.

3. append $\overline{v_{k\to i}}$ to node_to_grad[k]: compute and propagates partial adjoints to its inputs.

## 11.1 Step-by-Step Symbolic Construction

To illustrate how a system builder implements the algorithm, let's trace a concrete function $f(v_1) = (\exp(v_1) + 1)\exp(v_1)$. The goal here is not to compute a numerical value, but to construct a set of new nodes that represent the gradient computation.

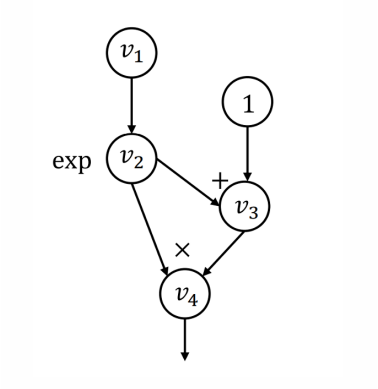



Figure 4: Original Forward Computational Graph

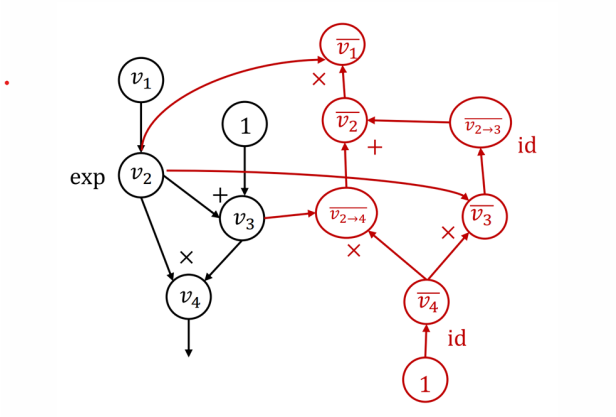



Figure 5: Completed Symbolic Backward Graph

1. **Initialize Output:** We start at the terminal node $v_4$ and create an identity gradient node $\overline{v_4} = 1$.

2. **Inspect Consumers of $v_3$:** Moving to $v_3$, the system identifies its only consumer is $v_4$ through a multiplication operation. Thus, it generates a new node $\overline{v_3} = \overline{v_4} \times v_2$.

3. **Handle Multiconsumer Nodes ($v_2$):** The variable $v_2$ is consumed by both $v_3$ (via addition) and $v_4$ (via multiplication). Following the summation rule, the system constructs a summation node $\overline{v_2} = \overline{v_{2\to3}} + \overline{v_{2\to4}}$.

4. **Complete the Chain:** Finally, the gradient of the input $v_1$ is constructed by applying the local derivative of the exponential function: $\overline{v_1} = \overline{v_2} \times \exp(v_1)$.

### 11.2 Summary: Symbolic vs. Concrete Gradients

The primary advantage of this approach is that the backward graph is constructed in a **symbolic way**. Unlike numerical differentiation, which results in a single value for a single point, this symbolic graph is a standalone computational structure. Once generated, it can be fused with the forward graph and reused to compute gradients for any set of input values provided during training, which is the operational foundation of modern frameworks like PyTorch and TensorFlow.

## 12 System Architecture: Backpropagation vs. Reverse-mode AD

The professor highlights a critical distinction between the theoretical teaching of Backpropagation (BP) and the architecture of modern deep learning systems.

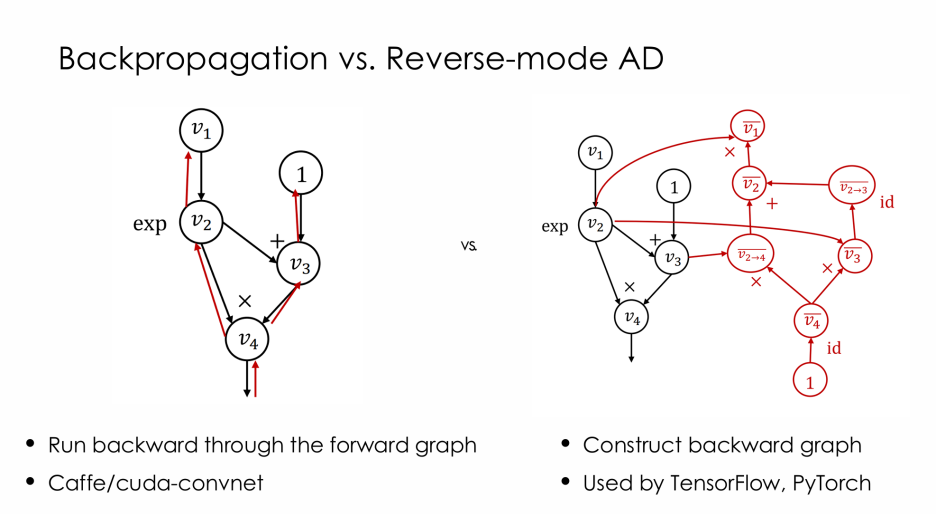



Figure 6: Backpropagation vs. Reverse-mode AD

- **Classical Backpropagation:** Older frameworks, such as Caffe or cuda-convnet, typically implemented BP by running computation backwards through the existing forward graph nodes.

- **Modern Reverse-mode AD:** Contemporary systems like TensorFlow and PyTorch do not simply "run backward." Instead, they explicitly construct a separate, symbolic backward graph that links back to the forward nodes. This provides a more holistic and flexible representation of the entire gradient computation process.

## 13 The "Incomplete" Problem: Toward ML Training

A key takeaway from this lecture is that a functional gradient graph is still "incomplete" for actual machine learning training. While the system can now derive gradients automatically, two essential components are still missing to form a full training loop:

1. **Parameter Updates:** The mechanism to actually apply the gradients to modify model weights.
2. **Optimizer Logic:** The integration of optimization algorithms (e.g., SGD, Adam, or Momentum) that determine how the update step is calculated.

These components, alongside the forward and backward graphs, form the "Master Equation" for ML systems: $\theta^{(t+1)} = f(\theta^{(t)}, \nabla_L(\theta^{(t)}, D^{(t)}))$.

This equation conceptually unifies the three core phases of the system's execution:

- **Forward Pass:** Computing the loss $L$ using operators like `matmul`, `relu`, and `MSE`.
- **Backward Pass:** Constructing the symbolic graph for $\nabla_L$ and generating gradient operators (e.g., `relu'`).
- **Update Phase:** Applying the results via update operators (e.g., `sub`) to adjust weights $\theta$.

The lecture concludes by posing a challenge for the homework: deriving the gradients for **Softmax cross entropy**. Looking ahead, the next session will dive into **Operator Optimization**—exploring how to make these primitives run efficiently on diverse hardware platforms like GPUs and TPUs.

## 14 Softmax Cross Entropy

$$y_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \quad L = -\sum_i t_i \log y_i$$

$$\frac{\partial L}{\partial x_i} = y_i - t_i$$

**Insight:** This simplification is widely used for efficient and numerically stable training.

## 15 ML Systems: Grand Challenges

- Fast execution
- Scalability
- Memory efficiency
- Hardware portability
- Energy efficiency
- Ease of programming and debugging

# 16 ML System Architecture

1. Dataflow Graph
2. Automatic Differentiation
3. Graph Optimization
4. Parallelization
5. Operator Optimization / Compilation
6. Runtime (Scheduling and Memory Management)

**Insight:** Each layer transforms the computation closer to hardware-efficient execution.

# 17 Graph Optimization

Goal:

$$G \to G' \quad \text{such that } G' \text{ executes faster}$$

**Techniques:**

- Operator fusion
- Constant folding
- Dead code elimination

# 18 Case Study: Conv2D + BatchNorm Fusion

$$W' = W \cdot R(c), \quad B' = B \cdot R(c) + P(c)$$

**Why Fusion Improves Performance:**

- Reduces memory access
- Minimizes kernel launches
- Improves cache locality

**Key Insight:**

Memory movement is often the dominant cost in modern ML systems.

---

# 19 Summary

## 19.1 JIT Compilation & Dataflow Graphs

**Execution styles.** *Define-and-run* executes operations immediately (easy to debug); *define-then-run* builds the full graph first (easier to optimize). JIT compilation bridges the two: code is written naturally but compiled for performance.

JIT works best on *static* graphs—fixed structure that can be optimized once and reused. *Dynamic* graphs (whose structure depends on input or control flow) are harder to compile and debug, so systems prefer static computation whenever possible.

**LLMs** are mostly static during training but dynamic at inference time, since the number of generated tokens varies per input.

### Handling Dynamic Behavior

Three strategies exist: (1) skip JIT entirely and use define-and-run; (2) embed control flow operators (Switch/Merge) directly in the graph—powerful but overly complex in practice; (3) **piecewise compilation with guards**—compile static segments normally, handle dynamic parts separately, and recompile when assumptions break. This last approach is the current state-of-the-art for both training and inference.

## 19.2 Automatic Differentiation

### Motivation

Training requires $\frac{\partial \mathcal{L}}{\partial \theta}$ for every parameter $\theta$. Finite differences

$$\frac{\partial f}{\partial \theta} \approx \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$

are inaccurate and require two forward passes per parameter—prohibitively expensive for large models.

### Forward Mode AD

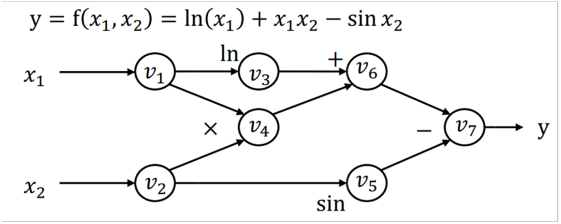
Each intermediate variable $v_i$ tracks $\dot{v}_i = \frac{\partial v_i}{\partial x_j}$ for a *single* chosen input $x_j$. Derivatives propagate left-to-right through the graph. For $f : \mathbb{R}^n \to \mathbb{R}^k$ this requires $n$ forward passes—impractical when $n$ (number of parameters) is large and $k = 1$ (a scalar loss).

### Reverse Mode AD (Backpropagation)

Instead of tracking derivatives forward, **adjoints** $\overline{v}_i = \frac{\partial y}{\partial v_i}$ propagate right-to-left in reverse topological order. The key recurrence is:

$$\overline{v}_i = \sum_{j \in \text{next}(i)} \overline{v}_{i\to j}, \qquad \overline{v}_{i\to j} = \overline{v}_j \cdot \frac{\partial v_j}{\partial v_i}.$$

For $f : \mathbb{R}^n \to \mathbb{R}^k$, only $k$ backward passes are needed. Since $k = 1$ in ML, *all* parameter gradients are obtained in a single backward pass—this is why deep learning universally uses reverse-mode AD.




Forward evaluation trace

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

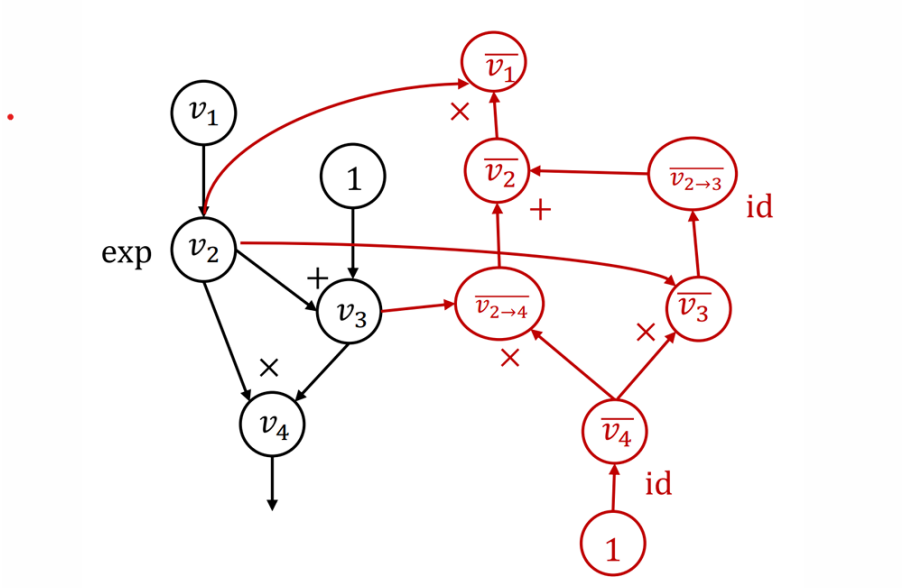
## Constructing the Backward Graph

Modern frameworks (TensorFlow, PyTorch) do not simply "run backward" through the forward graph (as older tools like Caffe did). They **symbolically construct a separate backward graph** that references forward nodes. The algorithm in pseudocode:

```
node_to_grad = {output: [1]}

for node in reverse_topological_order:
    grad[node] = sum(node_to_grad[node])
    for input in inputs(node):
        partial = grad[node] * d(node)/d(input)
        node_to_grad[input].append(partial)
```

Because the backward graph is symbolic, it can be fused with the forward graph, optimized, and reused for any input—the operational foundation of modern ML frameworks.



### 19.3 Complete Training Pipeline

Having forward and backward graphs is still *incomplete* for training. Two additional components are required:

1. **Parameter updates** — applying gradients to modify weights.
2. **Optimizer logic** — algorithms such as SGD, Adam, or Momentum.

Together these form the master training equation:

$$\theta^{(t+1)} = f\Big(\theta^{(t)},\, \nabla_{\mathcal{L}}\Big(\theta^{(t)}, D^{(t)}\Big)\Big)\,.$$

### 19.4 ML System Architecture Overview

Designing an ML system is fundamentally about managing **data**, **model**, and **compute**. A full system stack consists of six layers, each transforming the computation closer to hardware-efficient execution:

1. Dataflow Graph
2. Automatic Differentiation
3. Graph Optimization (operator fusion, constant folding, dead code elimination)
4. Parallelization
5. Operator Optimization / Compilation
6. Runtime (scheduling and memory management)

**Graph optimization example.** Fusing Conv2D and BatchNorm into a single kernel reduces memory accesses, minimizes kernel launches, and improves cache locality. Memory movement is often the dominant cost in modern ML systems, making such fusions highly impactful.

## 20 Takeaways

- Reverse-mode autodiff enables efficient gradient computation
- Training consists of forward, backward, and update stages
- Computational graphs enable optimization and compilation
- System-level design is critical for achieving high performance

## Scribe Responsibilities

- Wilson Sugiarto: Section 1 - 5
- Ka Sing He: Section 6 - 8

- Duan Wang: Section 9 - 11.0
- Jingwen Zhang: Section 11.1 - 13
- Atharva Hirulkar: Section 14 - 18
- Matthew Biehler: Summary (Section 19)