



CS 498: Machine Learning System Spring 2025

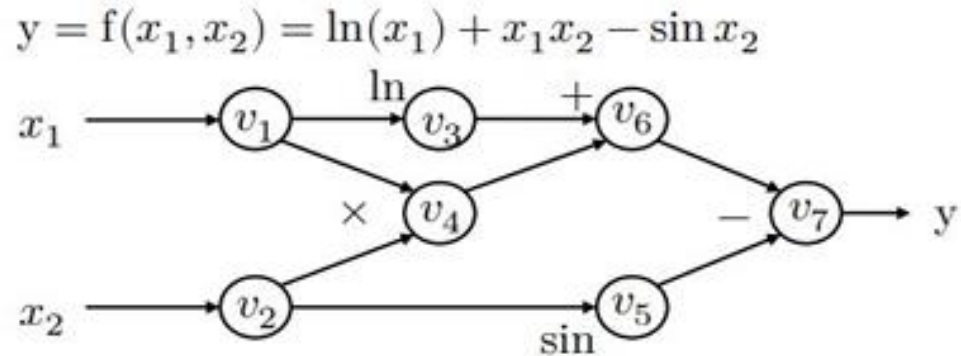
Minjia Zhang

The Grainger College of Engineering

Memory Optimization

- **Task:** Reduce the memory consumption to train a DL model
- **Main challenge:** Cost to store intermediate results and gradients
- **Background:** Reverse Mode Auto Differentiation
- **Key Ideas:**
 - Computation graph memory optimization (In-place Storing, Memory Sharing)
 - Save memory by re-computing (gradient checkpointing)

Background: Reverse Mode Auto Differentiation (AD)



Forward evaluation trace

$$v_1 = x_1 = 2$$

$$v_2 = x_2 = 5$$

$$v_3 = \ln v_1 = \ln 2 = 0.693$$

$$v_4 = v_1 \times v_2 = 10$$

$$v_5 = \sin v_2 = \sin 5 = -0.959$$

$$v_6 = v_3 + v_4 = 10.693$$

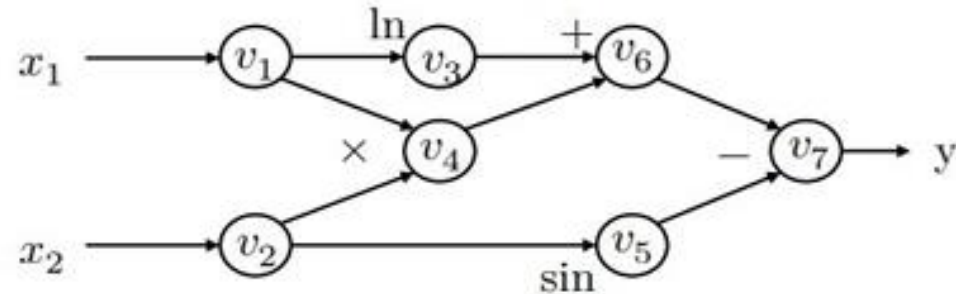
$$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$$

$$y = v_7 = 11.652$$

Background: Reverse Mode Auto Differentiation (AD)



$$y = f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin x_2$$



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 $\bar{v}_i = \frac{\partial y}{\partial v_i}$

We can then compute the \bar{v}_i iteratively in the **reverse** topological order of the computational graph

Reverse AD evaluation trace

$$\begin{aligned}\bar{v}_7 &= \frac{\partial y}{\partial v_7} = 1 \\ \bar{v}_6 &= \bar{v}_7 \frac{\partial v_7}{\partial v_6} = \bar{v}_7 \times 1 = 1 \\ \bar{v}_5 &= \bar{v}_7 \frac{\partial v_7}{\partial v_5} = \bar{v}_7 \times (-1) = -1 \\ \bar{v}_4 &= \bar{v}_6 \frac{\partial v_6}{\partial v_4} = \bar{v}_6 \times 1 = 1 \\ \bar{v}_3 &= \bar{v}_6 \frac{\partial v_6}{\partial v_3} = \bar{v}_6 \times 1 = 1 \\ \bar{v}_2 &= \bar{v}_5 \frac{\partial v_5}{\partial v_2} + \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_5 \times \cos v_2 + \bar{v}_4 \times v_1 = -0.284 + 2 = 1.716 \\ \bar{v}_1 &= \bar{v}_4 \frac{\partial v_4}{\partial v_1} + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_4 \times v_2 + \bar{v}_3 \frac{1}{v_1} = 5 + \frac{1}{2} = 5.5\end{aligned}$$

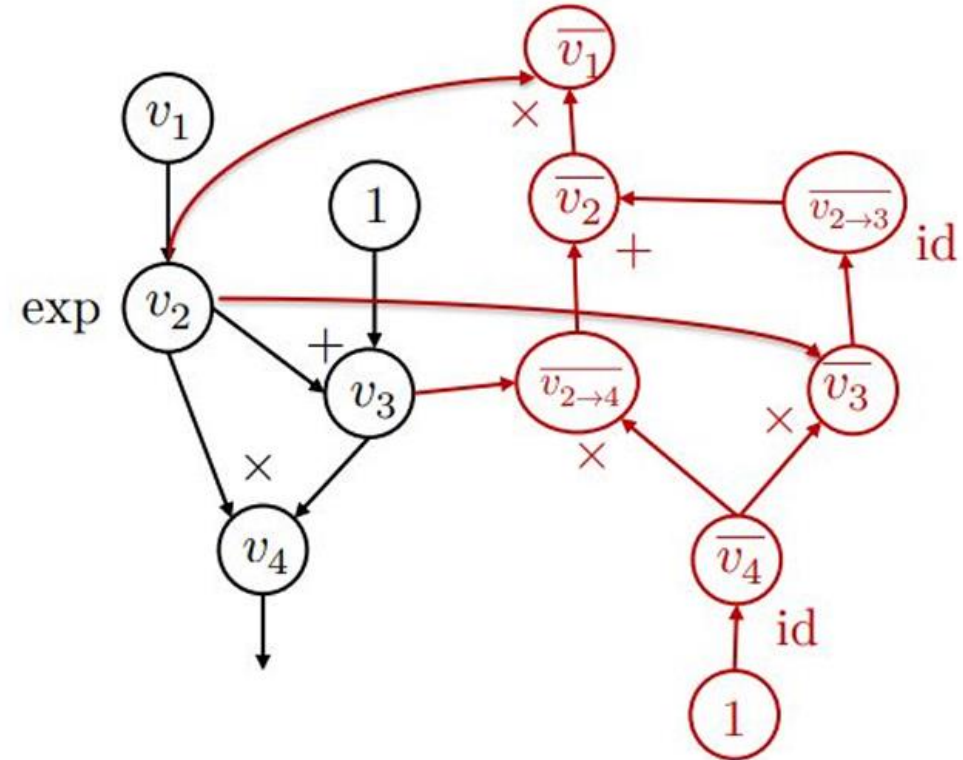
Background: Extending Computation Graph



```
def gradient(out):  
    node_to_grad = {out: [1]}  
    for i in reverse_topo_order(out):  
         $\bar{v}_i = \sum_j \bar{v}_{i \rightarrow j} = \text{sum}(\text{node\_to\_grad}[i])$   
        for  $k \in \text{inputs}(i)$ :  
            compute  $\bar{v}_{k \rightarrow i} = \bar{v}_i \frac{\partial v_i}{\partial v_k}$   
            append  $\bar{v}_{k \rightarrow i}$  to  $\text{node\_to\_grad}[k]$   
    return adjoint of input  $\bar{v}_{\text{input}}$ 
```



```
 $i = 2$   
node_to_grad: {  
    1:  $[\bar{v}_1]$   
    2:  $[\bar{v}_{2 \rightarrow 4}, \bar{v}_{2 \rightarrow 3}]$   
    3:  $[\bar{v}_3]$   
    4:  $[\bar{v}_4]$   
}
```

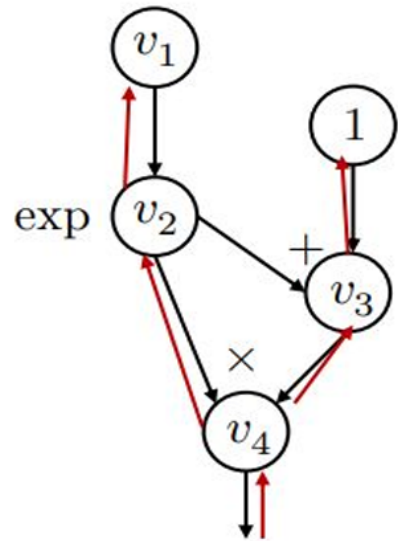


NOTE: id is identity function

Background: Reverse Mode AD vs Backprop



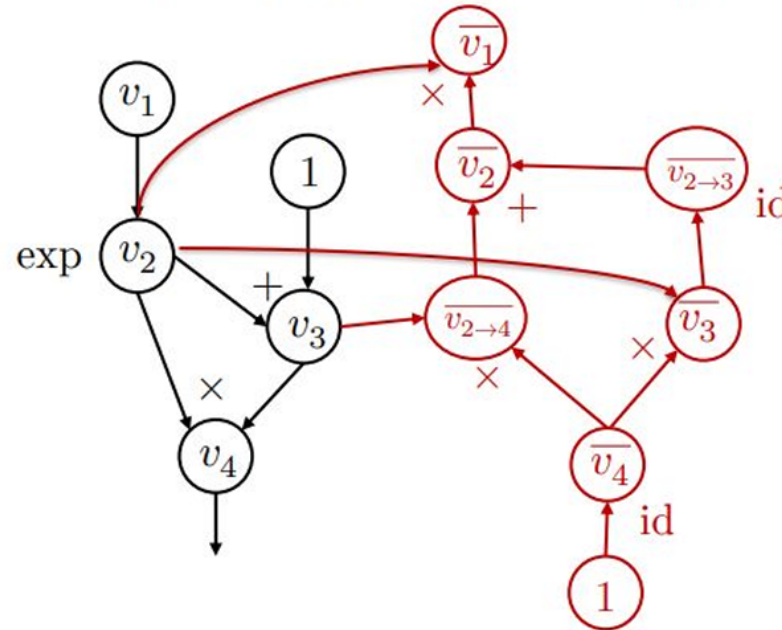
Backprop



- Run backward operations on the same forward graph.

- Used in first generation deep learning frameworks (caffe, torch)

Reverse mode AD by extending computational graph



- Construct separate graph nodes for reverse AD

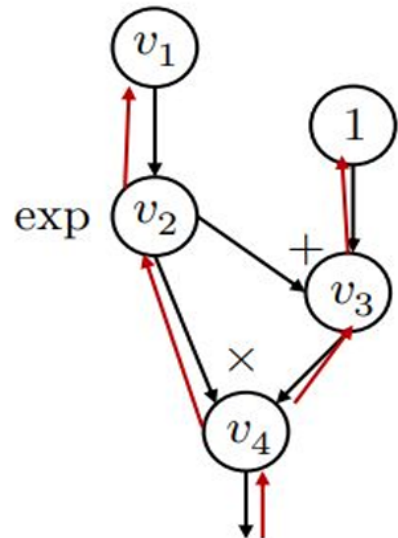
- Used by modern DL frameworks (PyTorch, TensorFlow)

Question: Why do we need automatic differentiation that extends the graph instead of backprop in graph?

Background: Reverse Mode AD vs Backprop



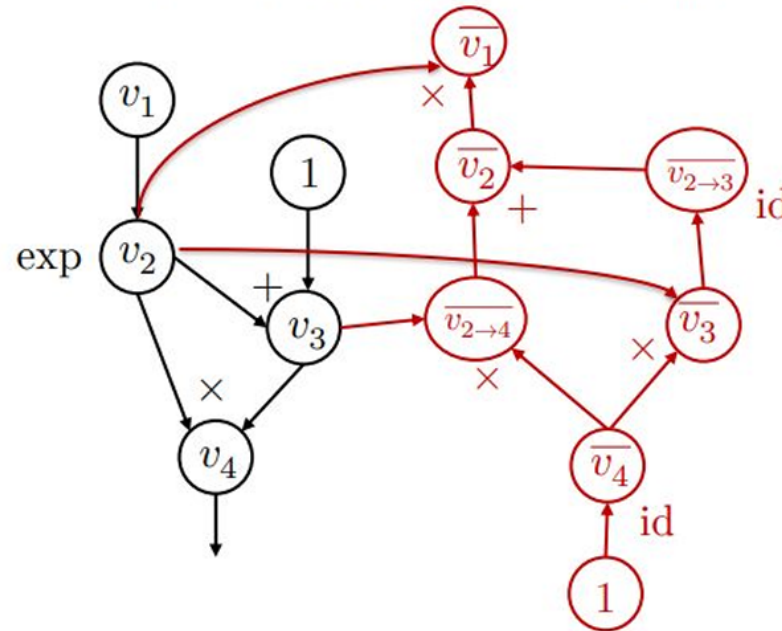
Backprop



- Run backward operations on the same forward graph.

- Used in first generation deep learning frameworks (caffe, torch)

Reverse mode AD by extending computational graph



- Construct separate graph nodes for reverse AD

- Used by modern DL frameworks (PyTorch, TensorFlow)

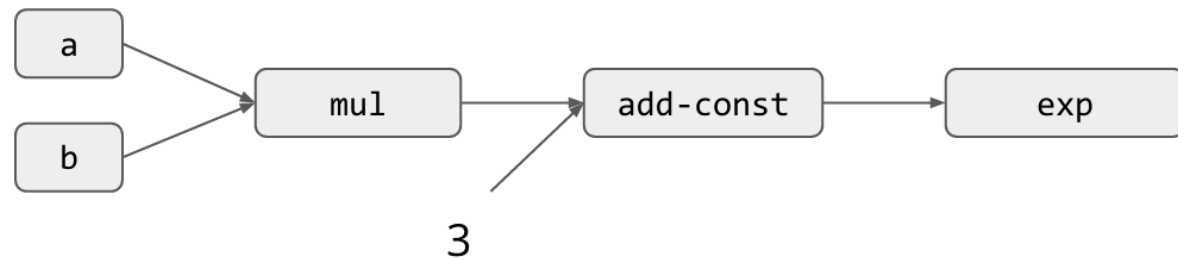
Advantages of using Explicit Extended Backward Path:

- Clearly describes computation dependency for better memory managements;
- Ability to have a different backward path (multiple out-going edges, gradient aggregation instead of introducing an explicit split layer);
- Ability to calculate higher order gradients (e.g., Hessian)

How to build an Executor for a Given Graph?

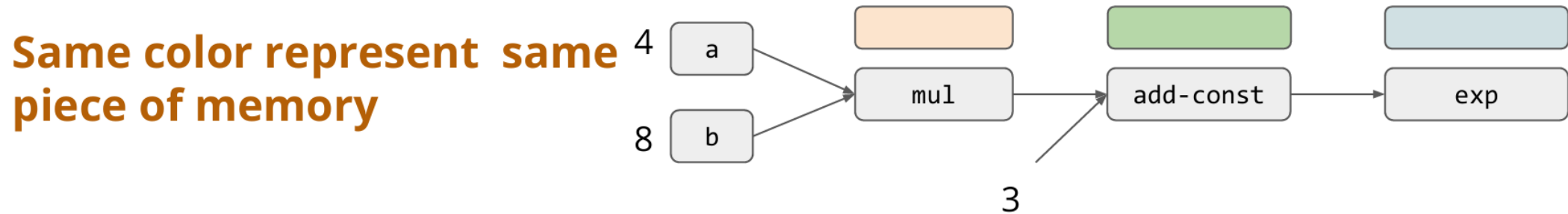


Computational Graph for $\exp(a * b + 3)$



1. **Allocate** temp memory for intermediate computation

Computational Graph for $\exp(a * b + 3)$

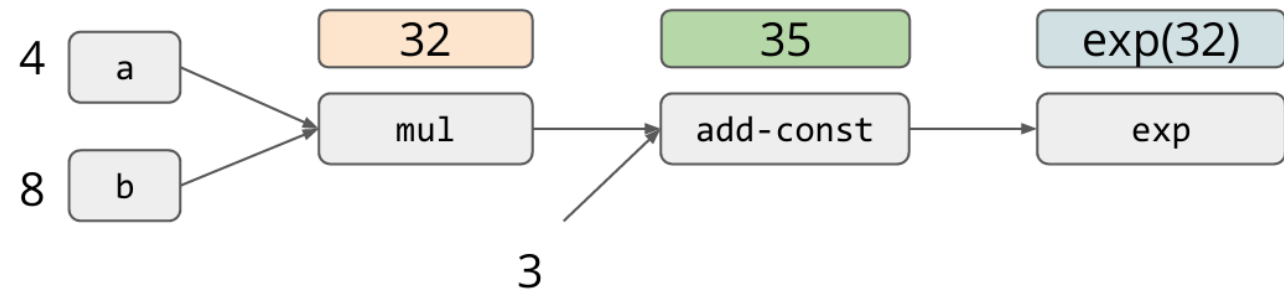


Build an Executor for a Given Graph



1. **Allocate** temp memory for intermediate computation
2. **Traverse and execute** the graph by topo order.

Computational Graph for $\exp(a * b + 3)$



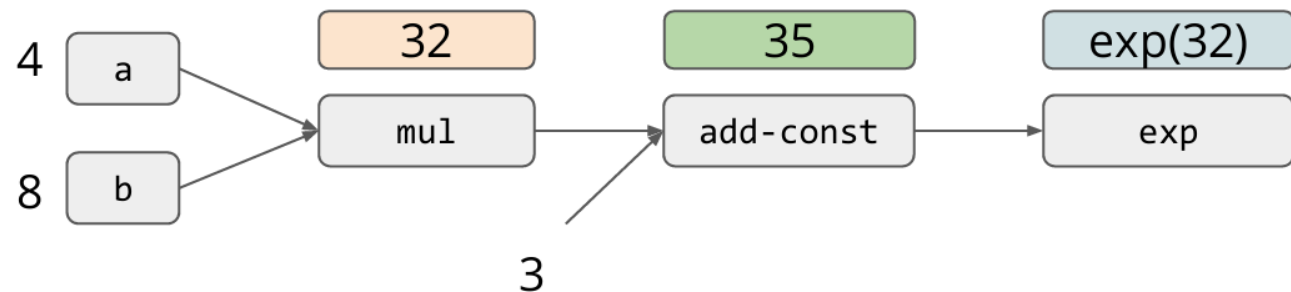
Build an Executor for a Given Graph



1. **Allocate** temp memory for intermediate computation
2. **Traverse and execute** the graph by topo order.

Temporary space linear to number of ops

Computational Graph for $\exp(a * b + 3)$



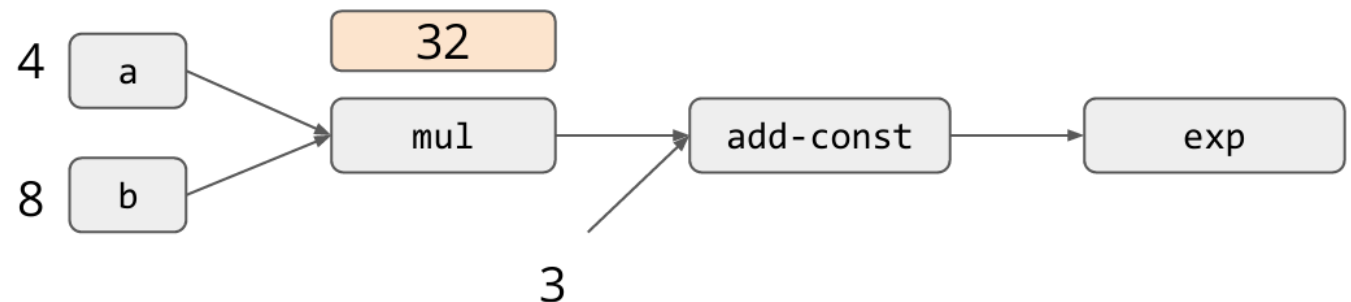


Dynamic Memory Allocation

Static Memory Allocation

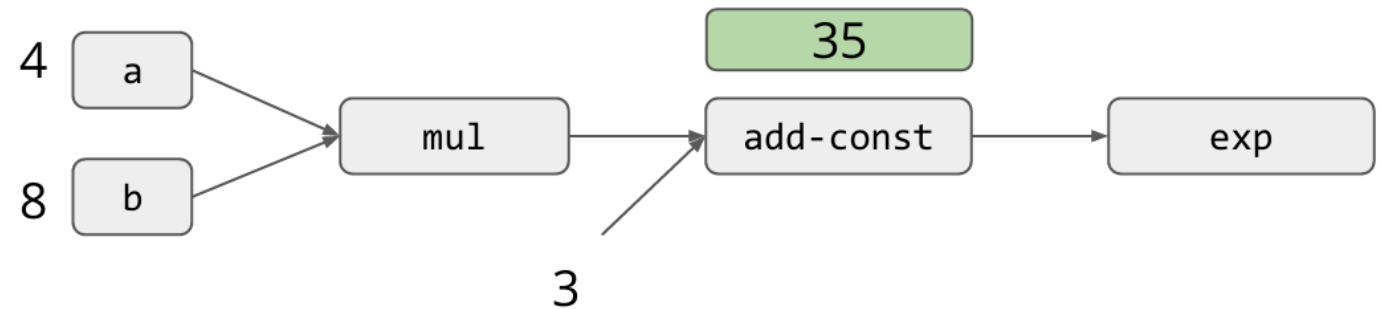
1. **Allocate** when needed
2. **Recycle** when a memory is not needed.

Memory Pool



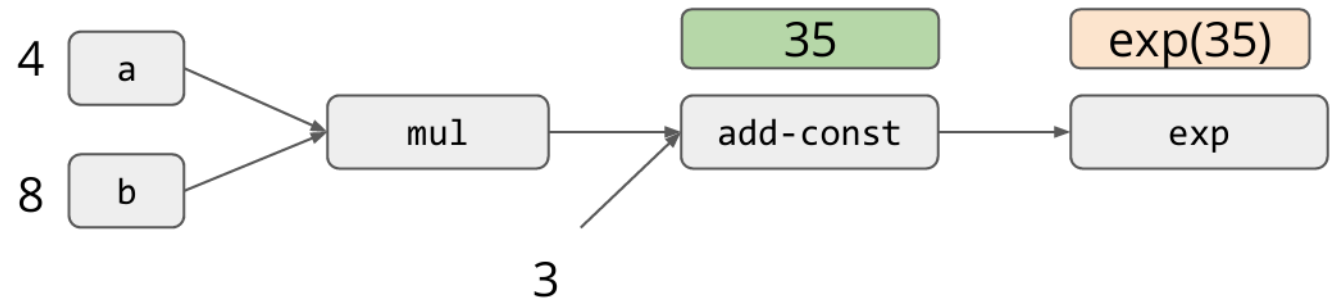
1. **Allocate** when needed
2. **Recycle** when a memory is not needed.

Memory Pool



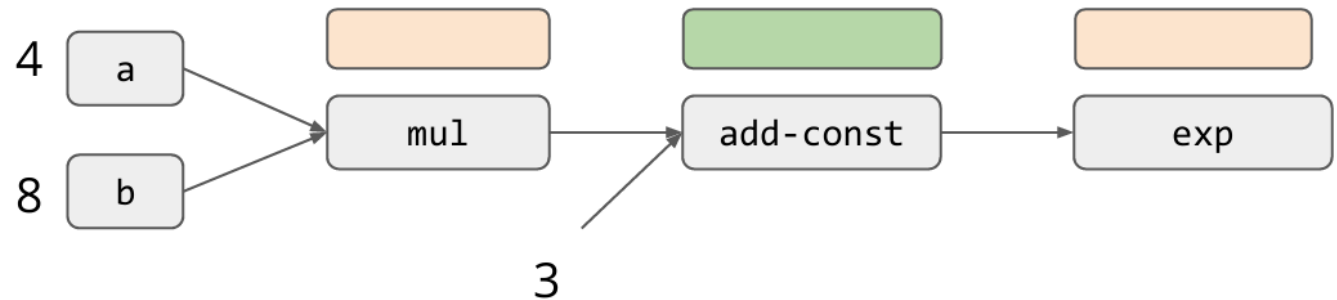
1. **Allocate** when needed
2. **Recycle** when a memory is not needed.

Memory Pool



1. Plan for reuse **ahead of time**
2. Analog: register allocation algorithm in compiler

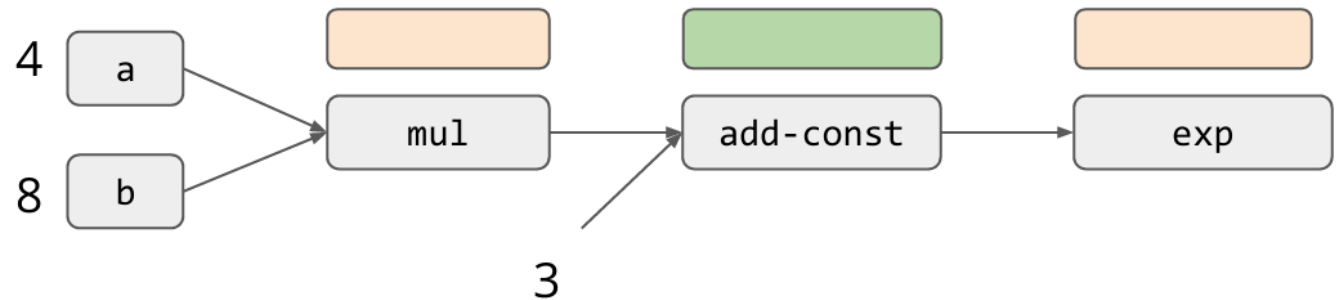
Same color represent same piece of memory



1. Plan for reuse **ahead of time**
2. Analog: register allocation algorithm in compiler

TensorFlow 1.x, PyTorch 2.0 (static graph mode)

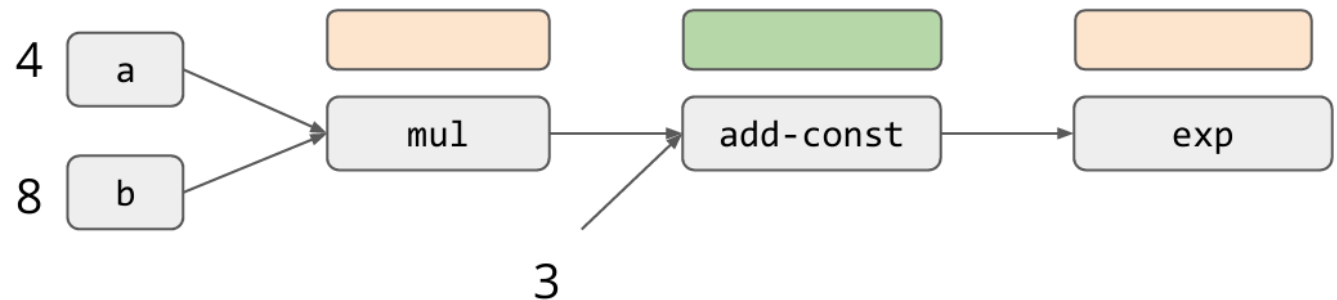
Same color represent same piece of memory



1. Plan for reuse **ahead of time**
2. Analog: register allocation algorithm in compiler

Question: What are differences, pros, and cons between static and dynamic allocation?

Same color represent same piece of memory



Dynamic Memory Allocation

- Can be done at runtime

Static Memory Allocation

- Determined at compile time

Dynamic Memory Allocation

- Can be done at runtime
- Flexible, more efficient use of memory

Static Memory Allocation

- Determined at compile time
- No runtime overhead

Dynamic Memory Allocation

- Can be done at runtime
- Flexible, more efficient use of memory
- Potential for fragmentation

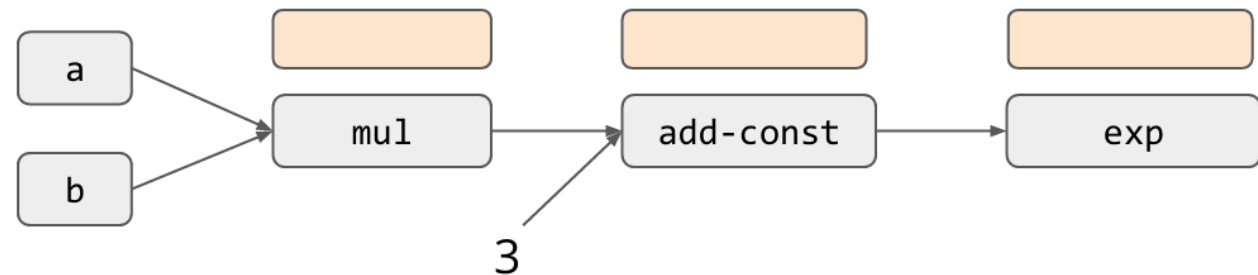
Static Memory Allocation

- Determined at compile time
- No runtime overhead
- Inflexible, can lead to wasted memory

- **Inplace** store the result in the input
- **Normal Sharing** reuse memory that are no longer needed.

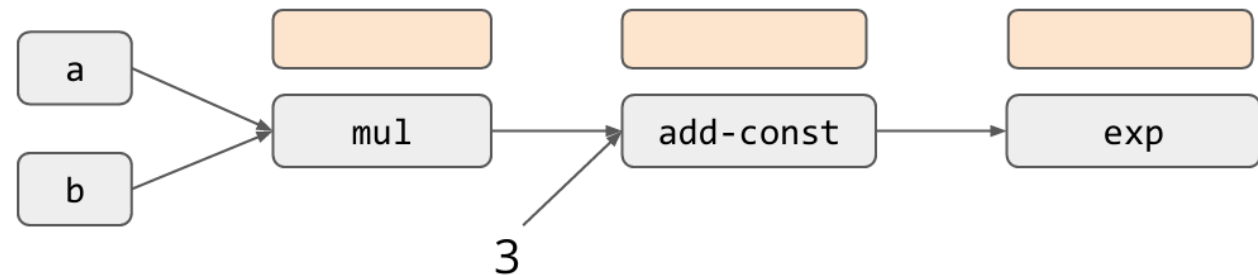
- Operations that modify data directly in the memory location where it is stored
- Examples: In-place activation functions, in-place BatchNorm
- Benefits:
 - Reduce memory footprint
 - Improve computation efficiency

Computational Graph for $\exp(a * b + 3)$



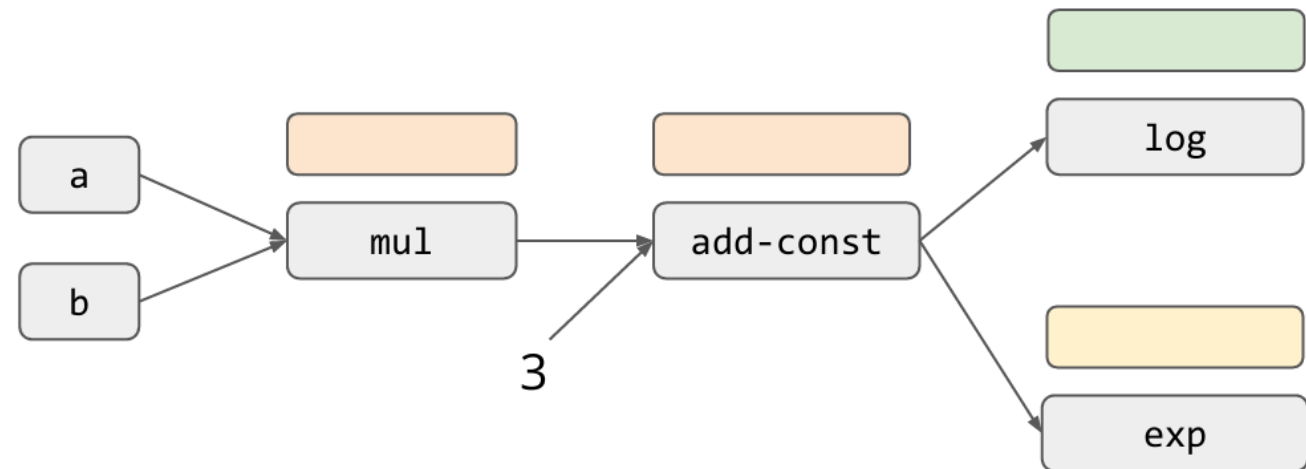
- Operations that modify data directly in the memory location where it is stored
- Examples: In-place activation functions, in-place BatchNorm
- Benefits:
 - Reduce memory footprint
 - Improve computation efficiency

Computational Graph for $\exp(a * b + 3)$



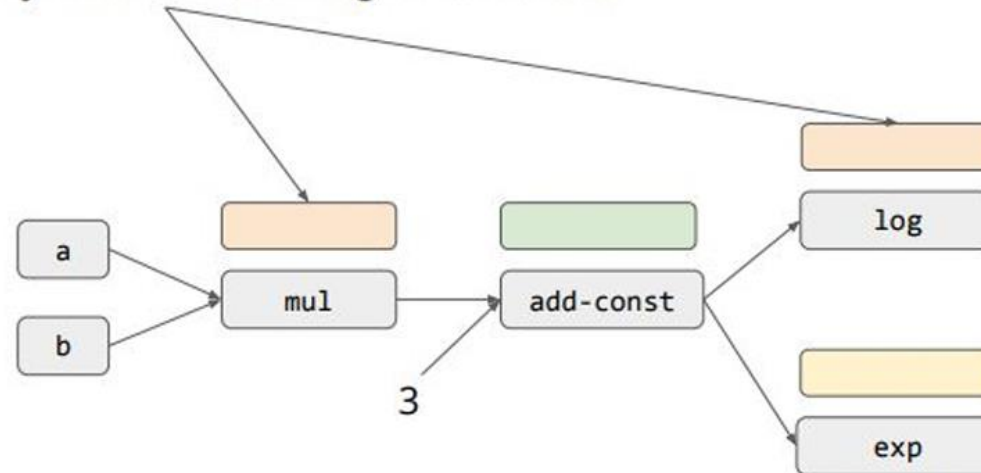
Question: What operation cannot be done in-place?

- We can only do in-place computation if the result op is the only consumer of the current value (i.e., no other operator depends on the current value)



- Memory used by intermediate results that are no longer needed can be recycled and used in another node;
- Shared memory between the nodes whose lifetime do not overlap

Recycle memory that is no longer needed.

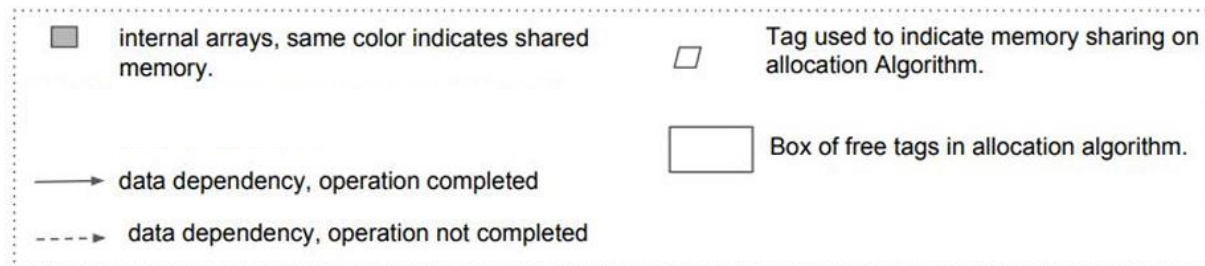
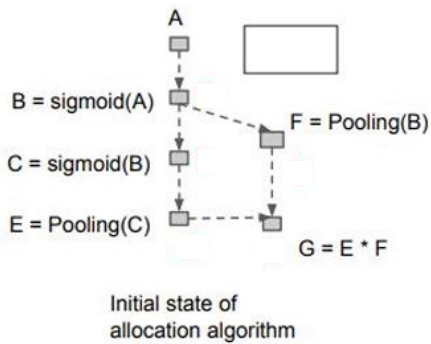


- Static Memory Planning: plan for reuse **ahead of time**

Memory Allocation/Planning Algorithm



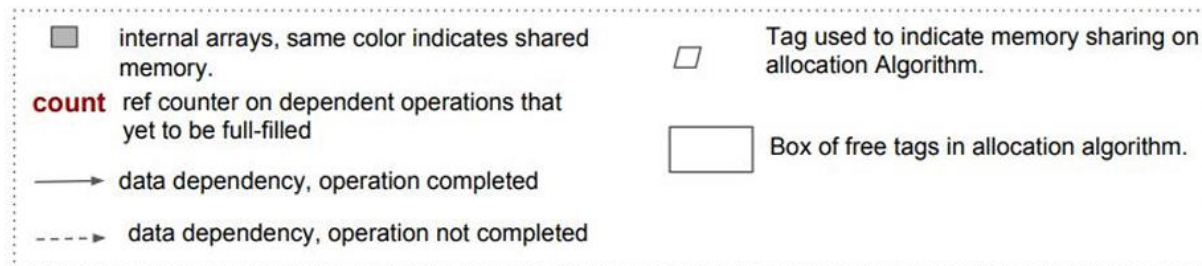
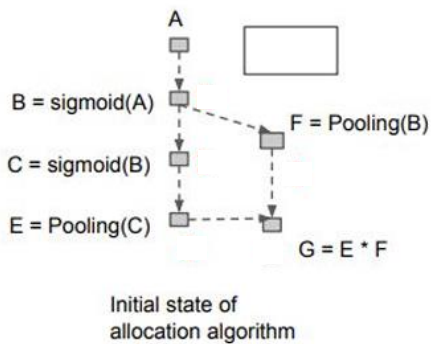
- Static Memory Planning: plan for reuse **ahead of time**



Memory Allocation/Planning Algorithm: Liveness Analysis



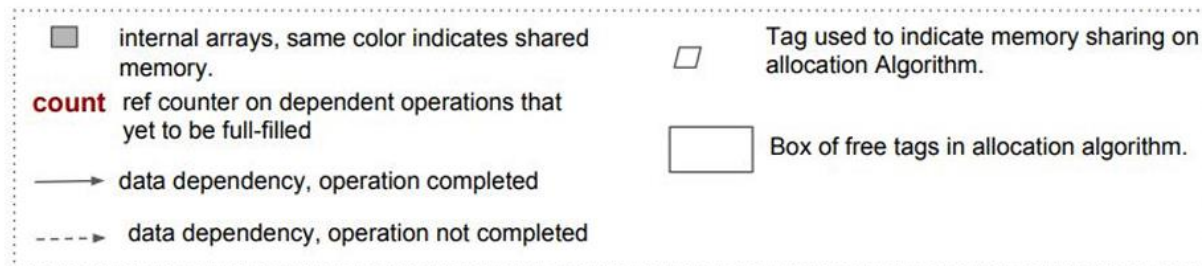
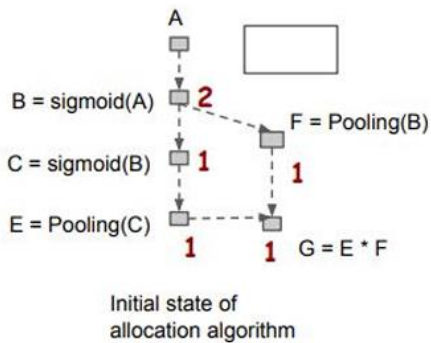
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



Memory Allocation/Planning Algorithm



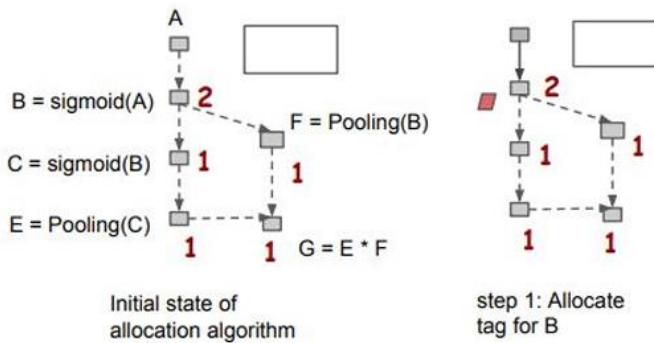
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



Memory Allocation/Planning Algorithm



- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused

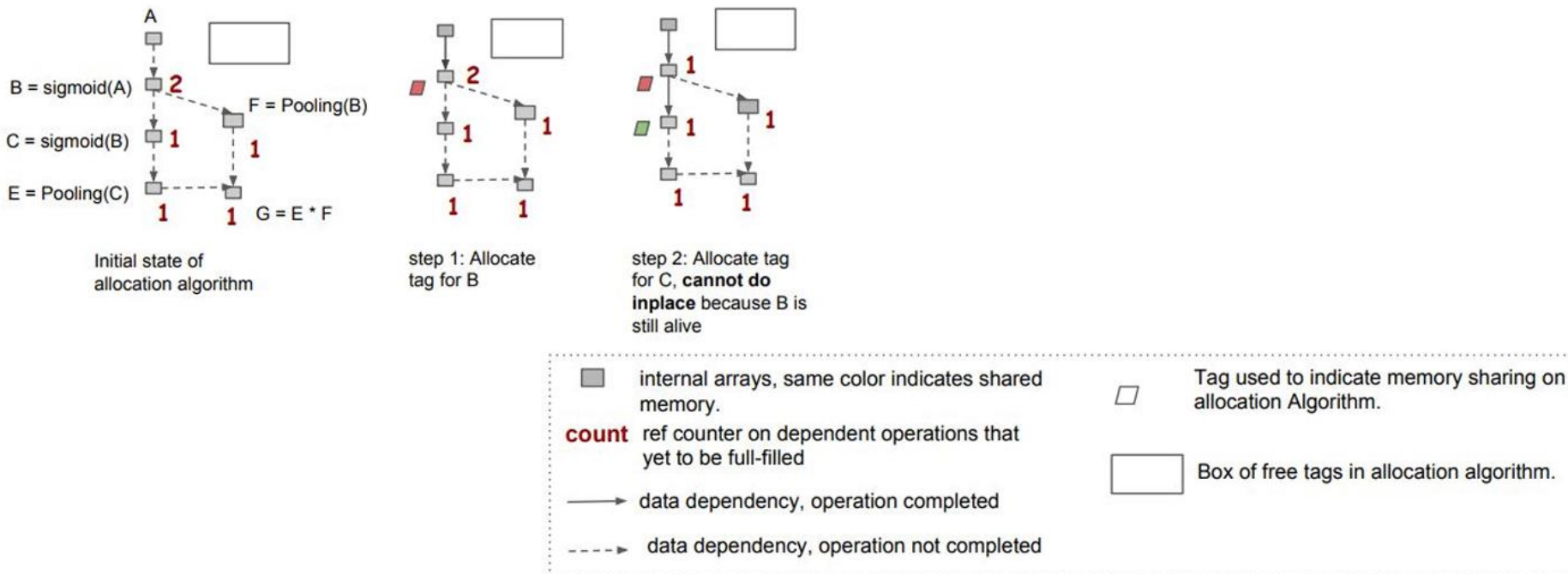


| | |
|---|--|
| internal arrays, same color indicates shared memory. | Tag used to indicate memory sharing on allocation Algorithm. |
| count ref counter on dependent operations that yet to be full-filled | Box of free tags in allocation algorithm. |
| data dependency, operation completed | |
| data dependency, operation not completed | |

Memory Allocation/Planning Algorithm



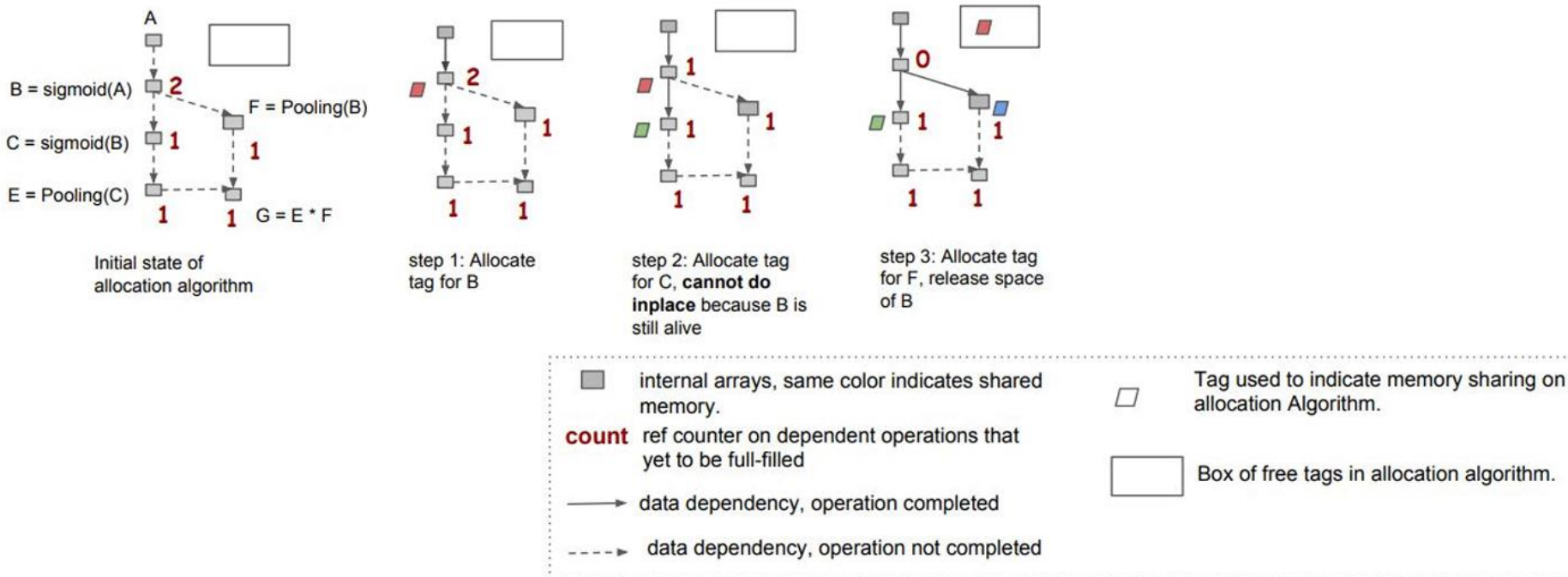
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



Memory Allocation/Planning Algorithm



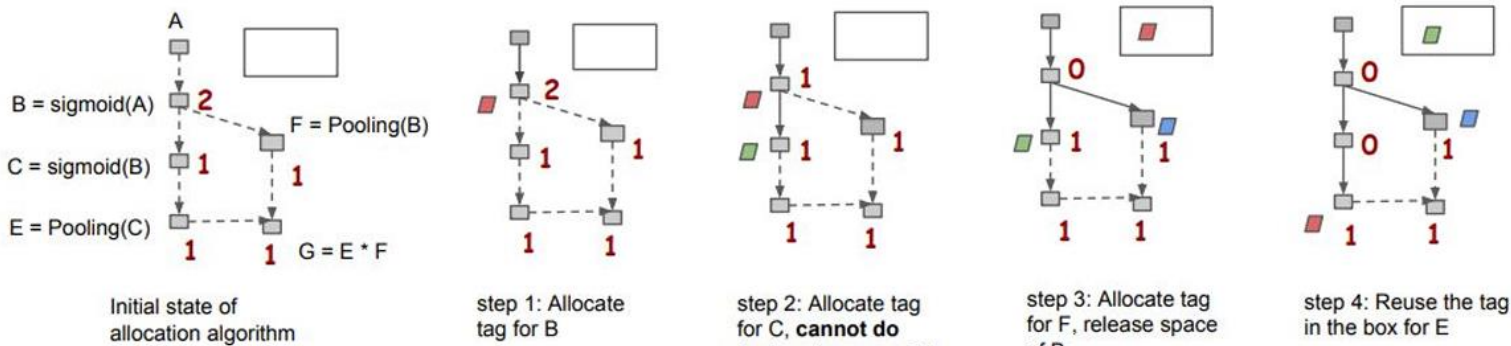
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



Memory Allocation/Planning Algorithm



- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused

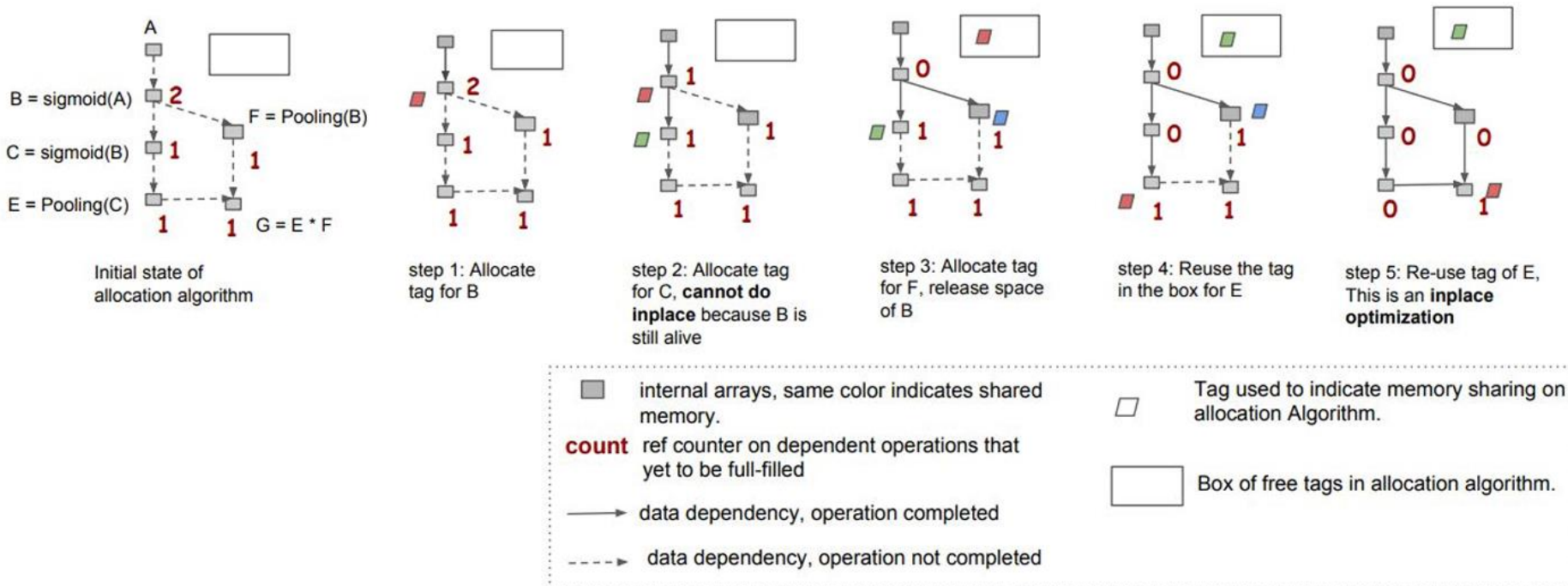


| | | | |
|--------------|--|--|--|
| | internal arrays, same color indicates shared memory. | | Tag used to indicate memory sharing on allocation Algorithm. |
| count | ref counter on dependent operations that yet to be full-filled | | Box of free tags in allocation algorithm. |
| | data dependency, operation completed | | |
| | data dependency, operation not completed | | |

Memory Allocation/Planning Algorithm



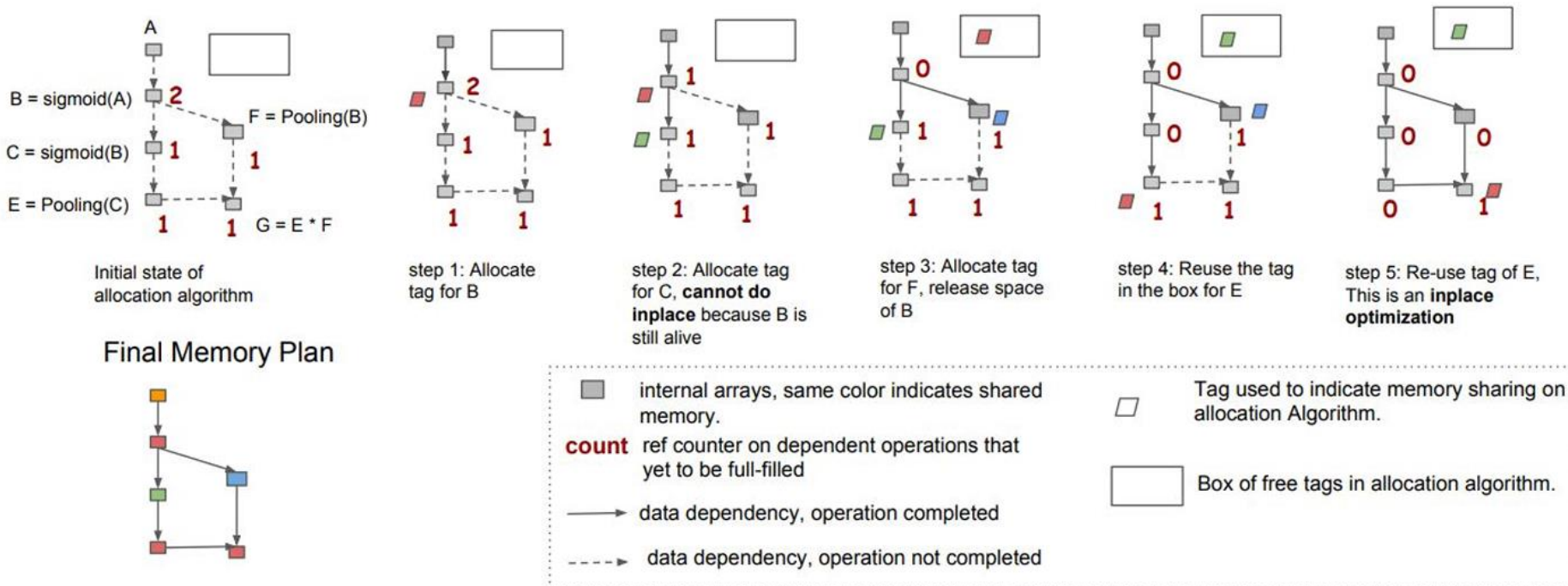
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



Memory Allocation/Planning Algorithm



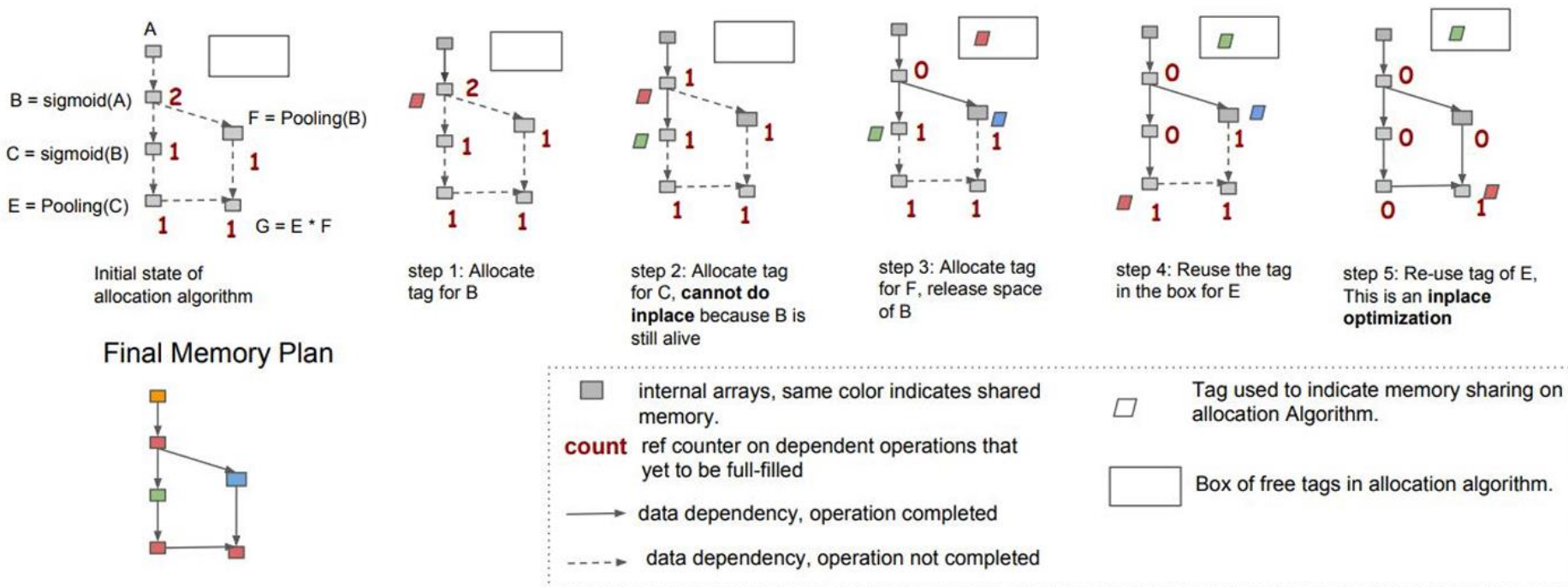
- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness counters**: Track the number of operations that need to be fulfilled before memory can be reused



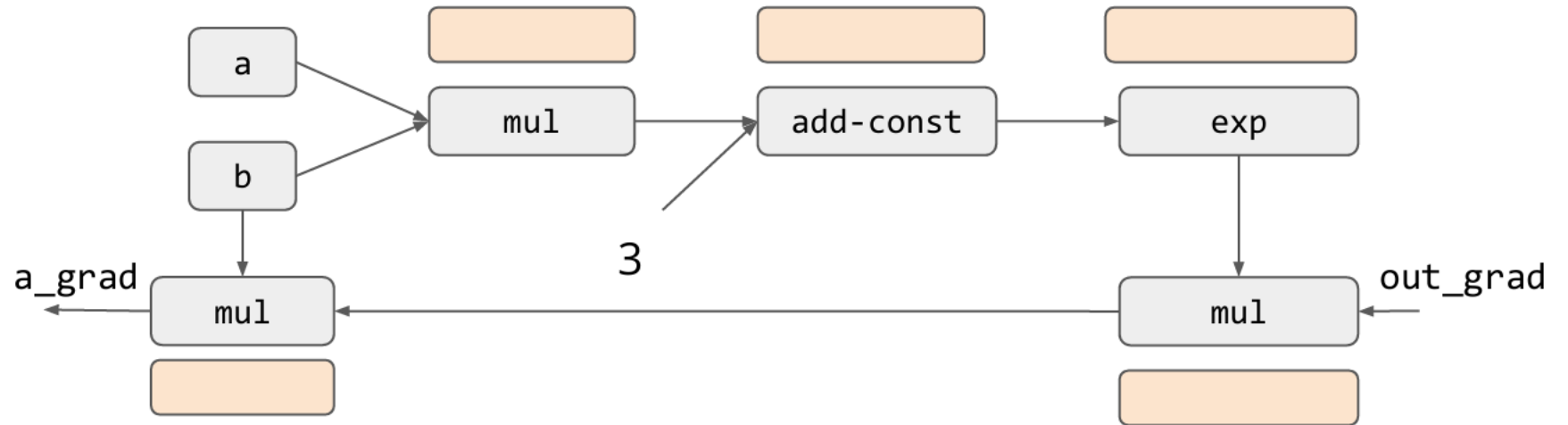
Memory Allocation/Planning Algorithm



- Static Memory Planning: plan for reuse **ahead of time**
- **Liveness** counters: Track the number of operations that need to be fulfilled before memory can be reused
- Dependencies: Important to declare minimum dependencies to optimize memory usage (DL frameworks already handle this for common operators)



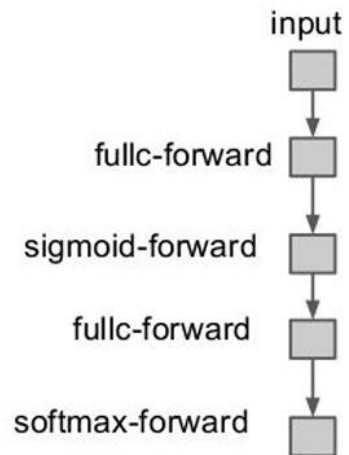
Back to the Question: Why do we need automatic differentiation that extends the graph instead of backprop in graph?



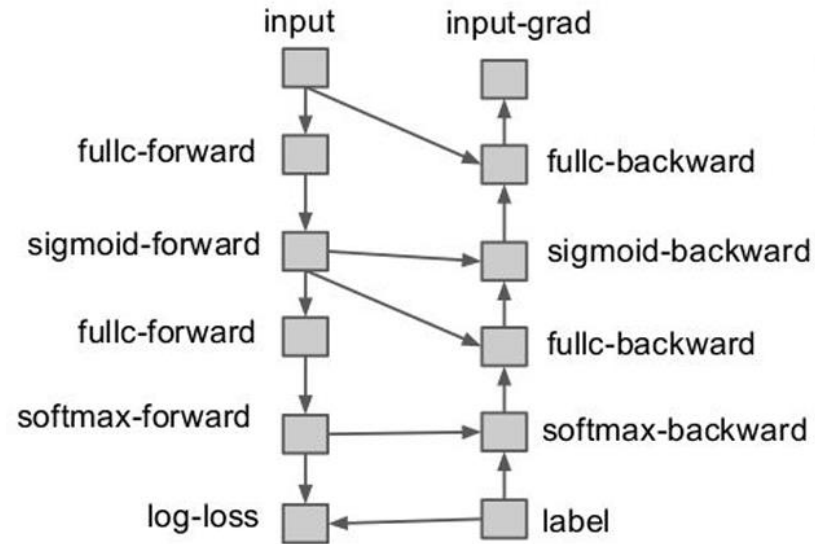
Memory Optimization on a Two Layer MLP



Network Configuration



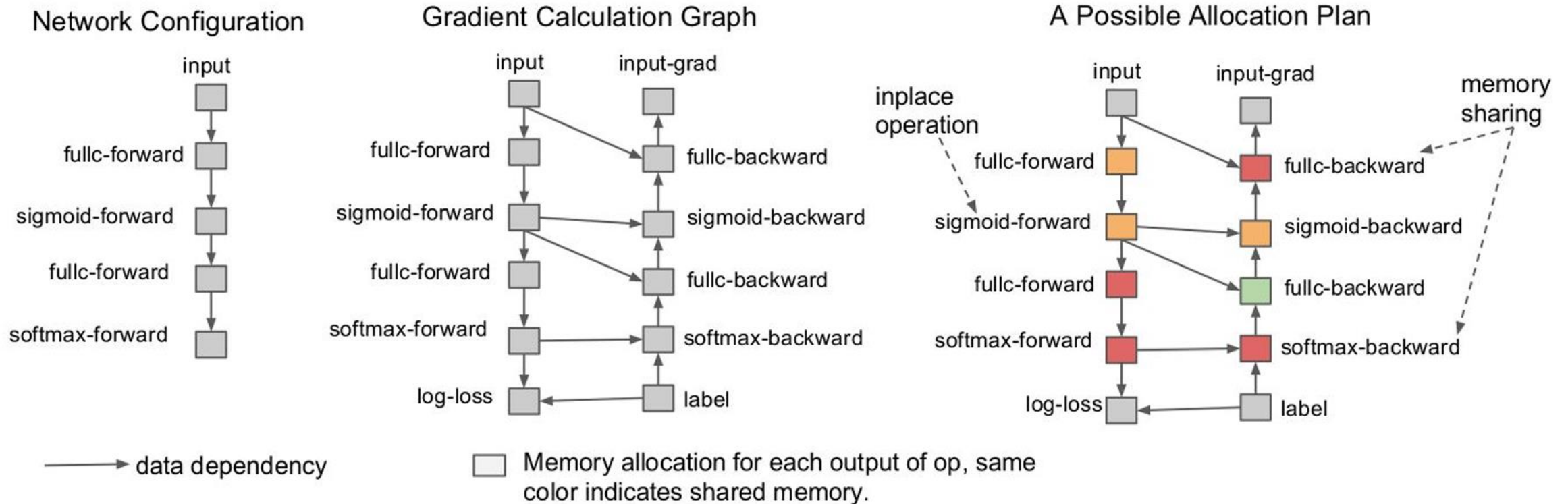
Gradient Calculation Graph



→ data dependency

□ Memory allocation for each output of op, same color indicates shared memory.

Memory Optimization on a Two Layer MLP



We can save memory usage by a **constant factor** using memory sharing for training, how can we do better?

We can save memory usage by a **constant factor** using memory sharing for training, how can we do better?

Idea: Drop some of the intermediate results and recover them from an extra forward computation (from the closest checkpointed results) when needed.

We can save memory usage by a **constant factor** using memory sharing for training, how can we do better?

Idea: Drop some of the intermediate results and recover them from an extra forward computation (from the closest checkpointed results) when needed.


Algorithm:

1. Divide the network into several segments;
2. Store the output of each segment and drops all the intermediate results within each segment;
3. Recompute dropped results at segment-level during back-propagation

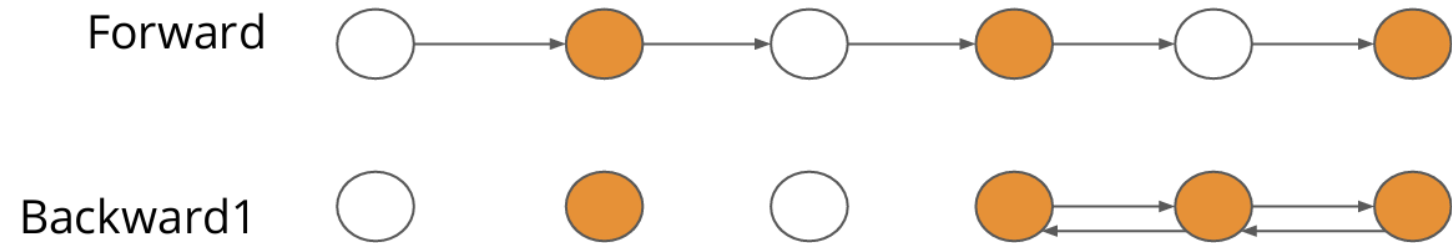
Checkpointing: storing only a few intermediate results during the forward pass.
Recomputation: Recomputing the necessary values during the backward for gradient calculation





 Data to be checkpointed for backprop

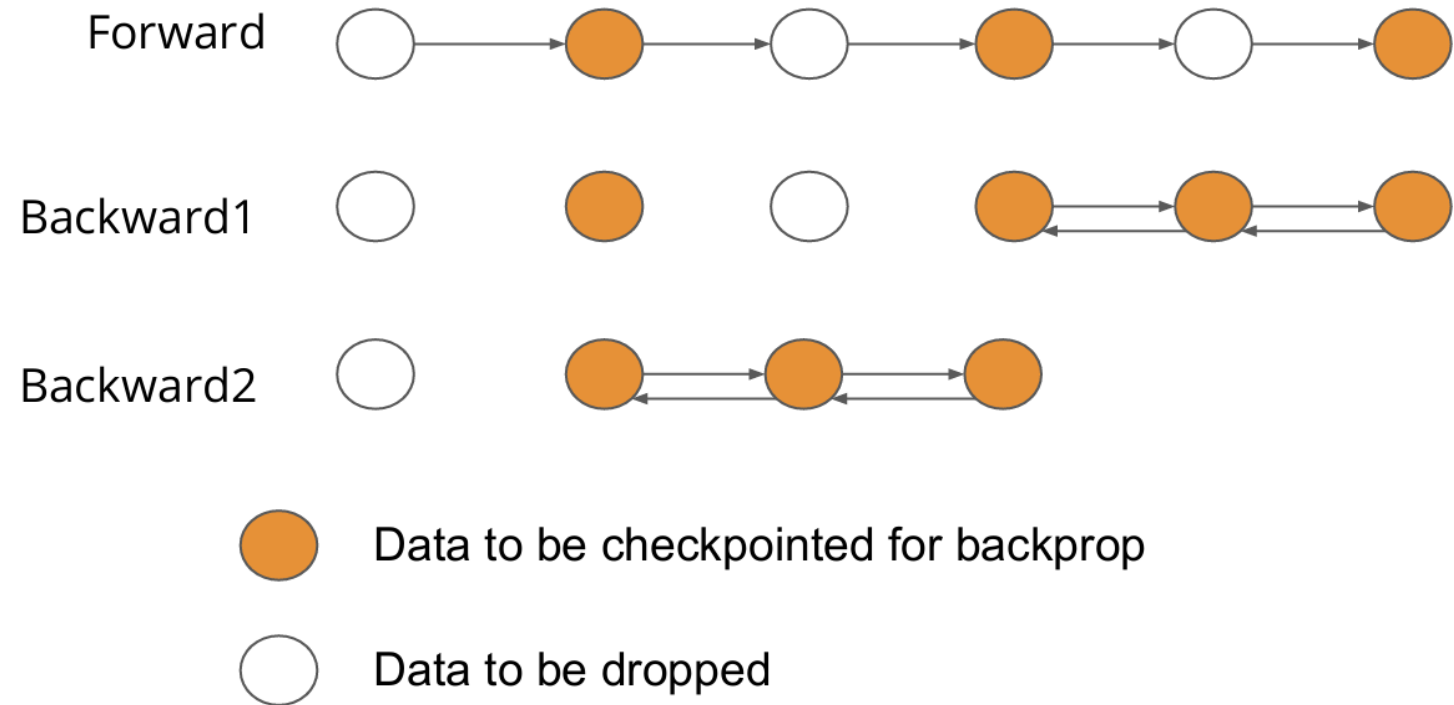
 Data to be dropped

Checkpointing: storing only a few intermediate results during the forward pass.
Recomputation: Recomputing the necessary values during the backward for gradient calculation



-  Data to be checkpointed for backprop
-  Data to be dropped

Checkpointing: storing only a few intermediate results during the forward pass.
Recomputation: Recomputing the necessary values during the backward for gradient calculation



Sublinear Memory Cost: $O(\sqrt{n})$



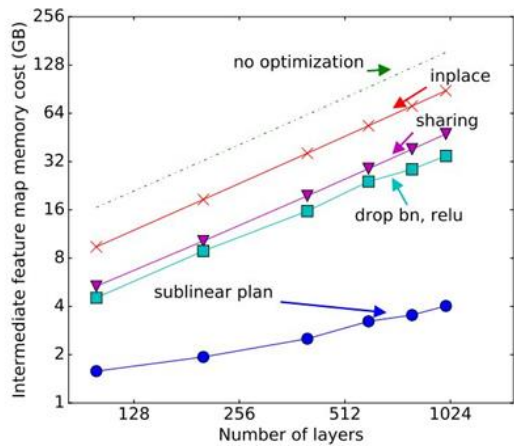
Drop the results of low compute cost operations, e.g., Conv-BatchNorm-Activation, drop results of the batch norm, activation, and pooling, but keep the results of convolution

Assume we divide the n network into k segments, the activation memory to train the network:

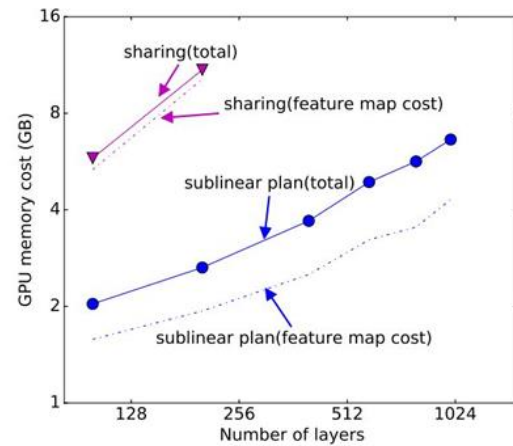
$$\text{cost-total} = \max_{i=1, \dots, k} \text{cost-of-segment}(i) + O(k) = O\left(\frac{n}{k}\right) + O(k)$$

Assume segments are equally divided, setting $k = \sqrt{n}$, we get $O(2\sqrt{n})$ with only one additional forward pass.

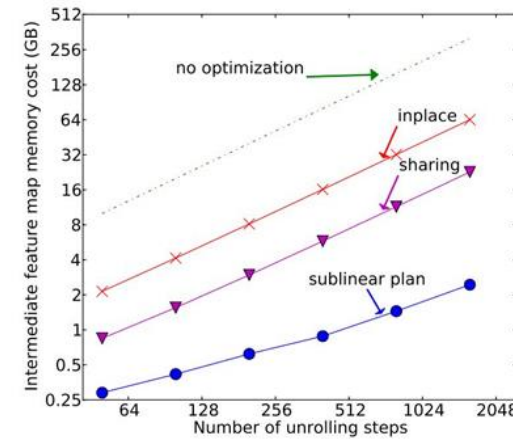
Running different schemes on top of MXNet on ResNet and LSTM.



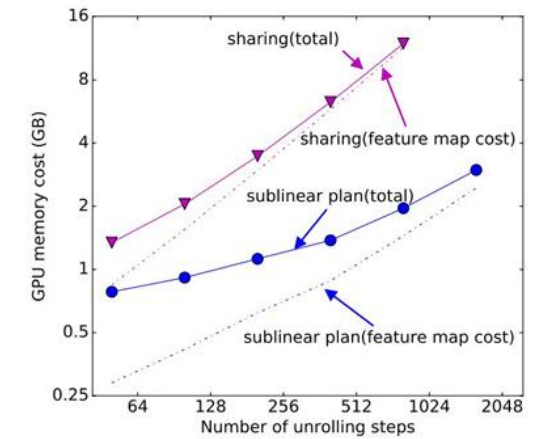
(a) Feature map memory cost estimation



(b) Runtime total memory cost



(a) Feature map memory cost estimation



(b) Runtime total memory cost

Training time roughly increase 30%.

Questions?