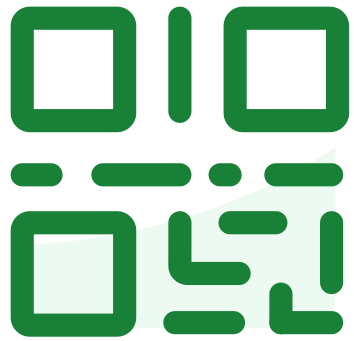


Do not edit
How to change the
design



**Join at slido.com
#9803087**

① The Slido app must be installed on every computer you're presenting from

slido



9803087

Lecture 13

Backpropagation

Computing gradients efficiently: numerical, symbolic, and automatic differentiation

EECS 189/289, Fall 2026 @ UC Berkeley

Joseph E. Gonzalez and Narges Norouzi



Recap: Stochastic Gradient Descent

Iteratively refine the weights by computing the gradient of the loss on a small sample of data.

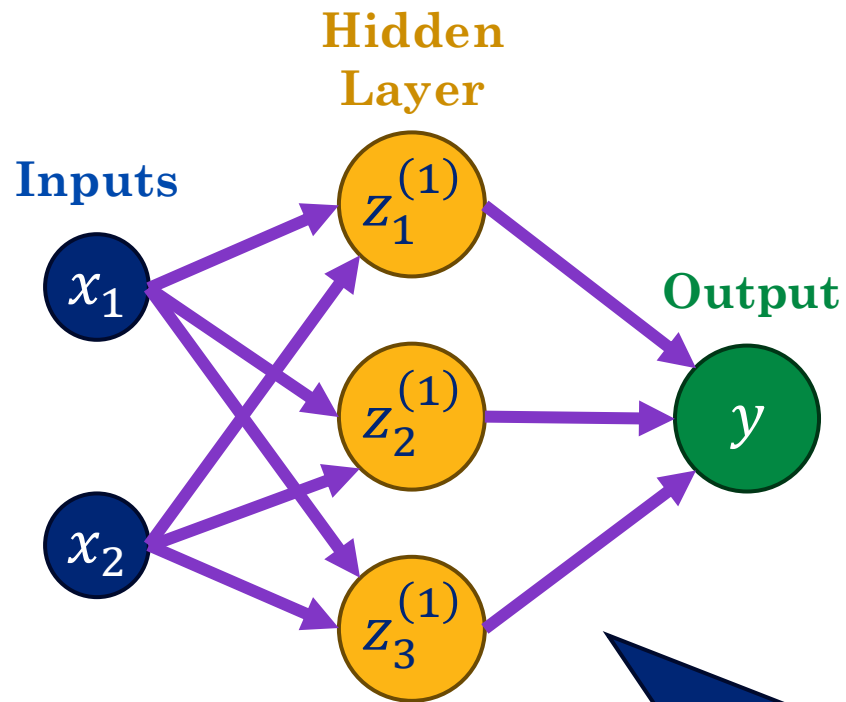
```
 $\mathbf{w}^{(0)} = \text{Initialize}()$   
for  $\tau$  in range(1,  $\tau_{\text{max}}$ ):  
     $\mathcal{B} = \text{RandBatch}(0, N, \text{size} = B)$   
     $\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} + \text{Step}(\nabla E_{\mathcal{B}}(\mathbf{w}^{(\tau-1)}), \tau, \beta, \eta)$   
    if  $\text{Converged}(\mathbf{w}^{(\tau)}, \mathbf{w}^{(\tau-1)}, \epsilon)$ : terminate
```

- Nearly all deep learning models are trained with variants of this algorithm.
- Today we will learn how to compute $\nabla E_{\mathcal{B}}$ efficiently and automatically



A Small Example

Consider the following single hidden layer classification network



No bias parameters in this example.

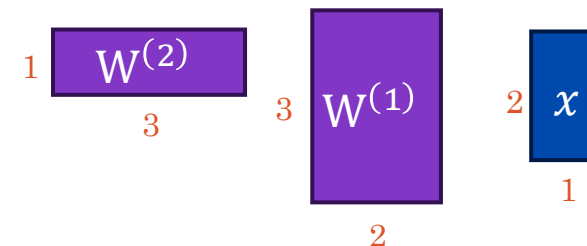
$$z_i^{(1)} = \sigma \left(\sum_{j=1}^2 w_{ij}^{(1)} x_j \right)$$

Sigmoid Activation Function

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

$$y(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{i=1}^3 w_{1i}^{(2)} z_i^{(1)} \right) = \sigma \left(\sum_{i=1}^3 w_{1i}^{(2)} \underbrace{\sigma \left(\sum_{j=1}^2 w_{ij}^{(1)} x_j \right)}_{\text{Substituting } z_i^{(1)}} \right)$$

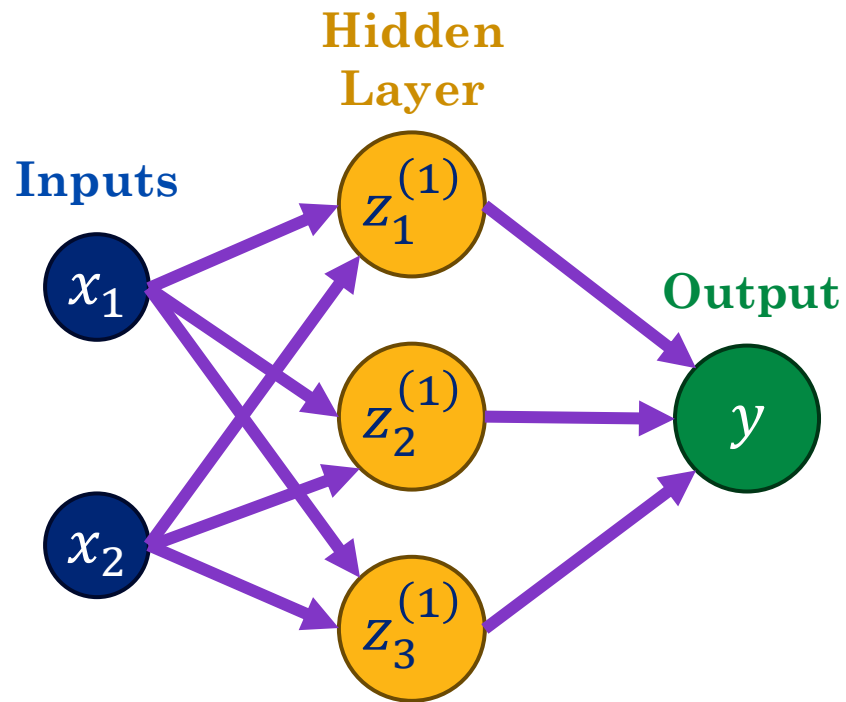
$$p(t = 1 | \mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w}) = \underbrace{\sigma \left(\mathbf{W}^{(2)} \sigma(\mathbf{W}^{(1)} \mathbf{x}) \right)}_{\text{Matrix Notation}}$$





Defining the Error Function

Computing the negative log likelihood of the data $\mathcal{D}$:



Sigmoid Activation Function

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

$$p(t = 1 \mid \mathbf{x}, \mathbf{w}) = \mathbf{y}(\mathbf{x}, \mathbf{w}) = \sigma \left(\mathbf{W}^{(2)} \sigma(\mathbf{W}^{(1)} \mathbf{x}) \right)$$

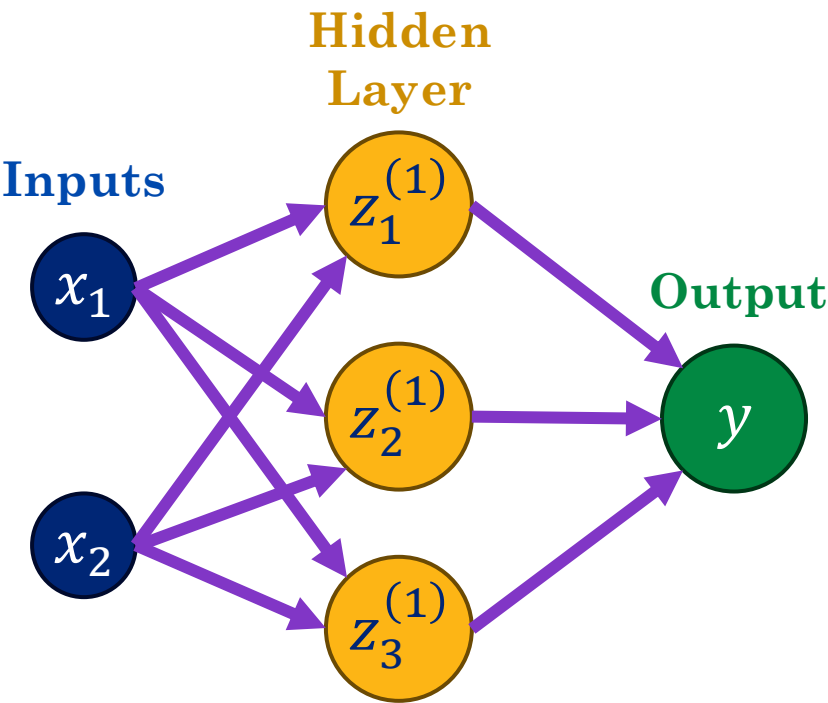
$$E(\mathbf{w}; \mathcal{D}) = -\ln \left(\underbrace{\prod_{n=1}^N \mathbf{y}(\mathbf{x}_n, \mathbf{w})^{t_n} (1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w}))^{(1-t_n)}}_{\text{Likelihood of the data.}} \right)$$

$$E(\mathbf{w}; \mathcal{D}) = - \underbrace{\sum_{n=1}^N [t_n \ln(\mathbf{y}(\mathbf{x}_n, \mathbf{w})) + (1 - t_n) \ln(1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w}))]}_{\text{Algebra!}}$$



Computing the Gradient

Computing the gradient $\nabla_{\mathbf{w}}$ of the negative log likelihood with respect to $\mathbf{w}$:



Sigmoid Activation Function

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

$$p(t = 1 \mid \mathbf{x}, \mathbf{w}) = \mathbf{y}(\mathbf{x}, \mathbf{w}) = \sigma \left(\mathbf{W}^{(2)} \sigma(\mathbf{W}^{(1)} \mathbf{x}) \right)$$
$$E(\mathbf{w}; \mathcal{D}) = - \sum_{n=1}^N [t_n \ln(\mathbf{y}(\mathbf{x}_n, \mathbf{w})) + (1 - t_n) \ln(1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w}))]$$

What is the shape of the gradient?
Recall $\mathbf{w} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$

3
 $\mathbf{W}^{(1)}$
2

1
 $\mathbf{W}^{(2)}$
3

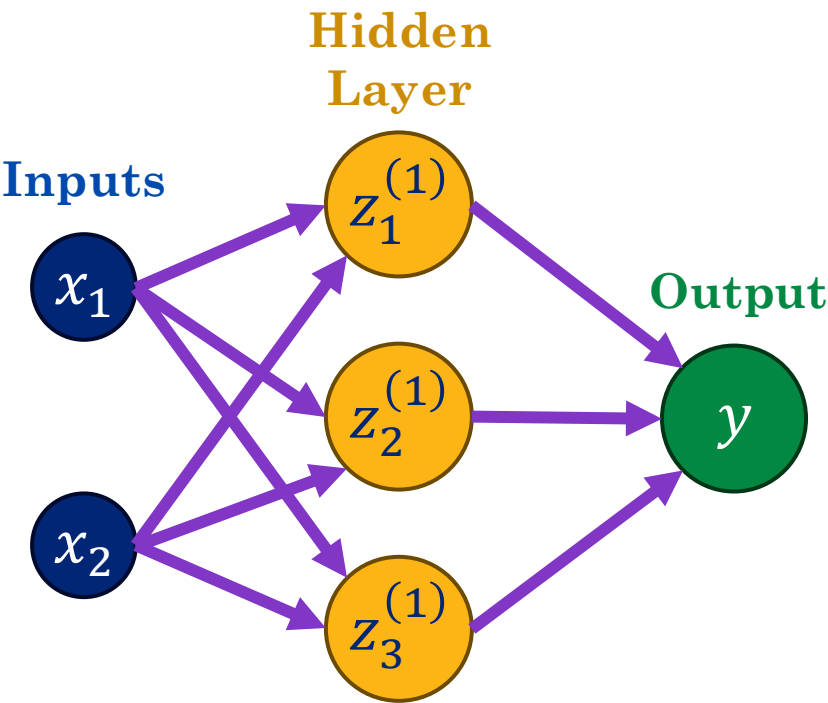


What is the shape of the gradient?



Computing the Gradient

Computing the gradient $\nabla_{\mathbf{w}}$ of the negative log likelihood with respect to $\mathbf{w}$:

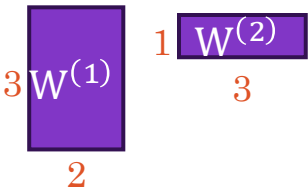


Sigmoid Activation Function

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

$$p(t = 1 \mid \mathbf{x}, \mathbf{w}) = \mathbf{y}(\mathbf{x}, \mathbf{w}) = \sigma \left(\mathbf{W}^{(2)} \sigma(\mathbf{W}^{(1)} \mathbf{x}) \right)$$
$$E(\mathbf{w}; \mathcal{D}) = - \sum_{n=1}^N [t_n \ln(\mathbf{y}(\mathbf{x}_n, \mathbf{w})) + (1 - t_n) \ln(1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w}))]$$

What is the shape of the gradient?
Recall $\mathbf{w} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$



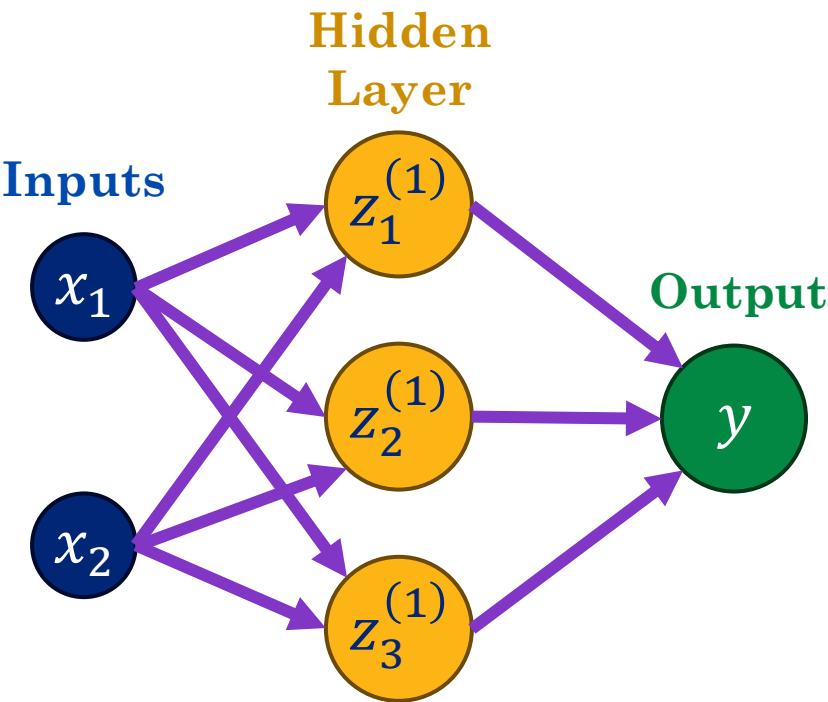
The gradient $\nabla_{\mathbf{w}}$ is the **vector** of **partial derivatives** for all the entries in $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$.

Here $\nabla_{\mathbf{w}} E(\mathbf{w}; \mathcal{D}) \in \mathbb{R}^{(3 \times 2 + 1 \times 3)} = \mathbb{R}^9$ (9 edges)



Computing the Gradient

Using the chain rule to compute the partial derivative with respect to w_i



Sigmoid Activation Function

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

$$p(t = 1 \mid \mathbf{x}, \mathbf{w}) = \mathbf{y}(\mathbf{x}, \mathbf{w}) = \sigma \left(\mathbf{W}^{(2)} \sigma(\mathbf{W}^{(1)} \mathbf{x}) \right)$$
$$E(\mathbf{w}; \mathcal{D}) = - \sum_{n=1}^N [t_n \ln(\mathbf{y}(\mathbf{x}_n, \mathbf{w})) + (1 - t_n) \ln(1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w}))]$$

Recall that $\frac{\partial}{\partial q} \ln(q) = \frac{1}{q}$ then with the **chain rule**:

$$\frac{\partial}{\partial w_i} E(\mathbf{w}; \mathcal{D}) = - \sum_{n=1}^N \left[t_n \left(\frac{1}{\mathbf{y}(\mathbf{x}_n, \mathbf{w})} \right) \frac{\partial \mathbf{y}(\mathbf{x}_n, \mathbf{w})}{\partial w_i} - (1 - t_n) \left(\frac{1}{1 - \mathbf{y}(\mathbf{x}_n, \mathbf{w})} \right) \frac{\partial \mathbf{y}(\mathbf{x}_n, \mathbf{w})}{\partial w_i} \right]$$

We need to compute the **partial derivative of the network $\mathbf{y}(\mathbf{x}, \mathbf{w})$** with respect to **each weight w_i**

PyTorch



What is PyTorch?

- PyTorch is a package used to build and train neural networks.
- **torch.tensor** is the basic datatype used to encode data, weights, and activations and is very similar `numpy.ndarrays`.
- Neural networks are built using the many primitives in the **torch.nn** module and defining the **forward** process of making a prediction.
- PyTorch uses automatic differentiation to efficiently compute the **gradient**.
- Training is accomplished by **(you) implementing** some variation of gradient descent.



9803087

torch.nn.Module

When building a new model you extend the nn.Module

```
class MLP(nn.Module):
```

Extend nn.Module

```
    def __init__(self, dims, activation="relu"):
        super().__init__()
        self.layers = nn.ModuleList(
            [nn.Linear(dims[i], dims[i+1]) for i in range(len(dims)-1)])
        if activation == "relu": self.act = nn.functional.relu
        else: self.act = torch.tanh

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            x = layer(x)
            if i < len(self.layers)-1:
                x = self.act(x)

        return x
```

Add modules to your module. These add parameters.

Compute the **forward** pass using the layer modules.

Implement Gradient Descent with Optimizer



In PyTorch you implement **your own gradient descent loop** but you can use built in update rules (e.g., Adam) called **optimizers**.

```
loader = DataLoader(tr_data, batch_size=32, shuffle=True)
```

```
optimizer = torch.optim.Adam(model.parameters(), eta)
```

```
for epoch in range(nepochs):
```

```
    for (x, t) in loader:
```

```
        optimizer.zero_grad()
```

```
        loss = loss_fn(model(x), t)
```

```
        loss.backward()
```

```
        optimizer.step()
```

Minibatch SGD
DataLoader

Adam optimizer

Epoch is one pass over data

Load one minibatch at a time

Compute the gradient
of the loss

Use optimizer to update all the parameters (1-step)

Demo

PyTorch



9803087



Numerical Differentiation



Proposal: Numerical Differentiation

We could estimate the gradient numerically by computing:

$$\frac{\partial}{\partial w_i} E(\mathbf{w}; \mathcal{D}) \approx \frac{E(\mathbf{w} + \epsilon \mathbf{I}_i; \mathcal{D}) - E(\mathbf{w} - \epsilon \mathbf{I}_i; \mathcal{D})}{2\epsilon}$$

for a small value of ϵ (where $\mathbf{I}_i$ is the i^{th} column of the identity matrix).

```
def numeric_gradient(f, x1, x2, h=1e-6):  
    """  
    Central difference estimate of the gradient of f at (x1, x2).  
    """  
    return [(f(x1 + h, x2) - f(x1 - h, x2)) / (2 * h),  
            (f(x1, x2 + h) - f(x1, x2 - h)) / (2 * h)]
```

```
numeric_gradient(f, 1, 2)  
  
[16.778112197357586, 8.805202936557066]
```

Proposal: Numerical Differentiation



Advantages

- Can apply to ***any* error function** and network architecture.
- Easy to implement.
- Useful for **checking** a backpropagation implementation (Bishop-8.1.4).

Disadvantages

- **Approximate:** the error is $\mathcal{O}(\epsilon^2)$ until round-off error dominates, so ϵ cannot be made arbitrarily small.
- **Expensive.** Why?



Cost of Numerical Differentiation

Computing the **gradient** using numerical differentiation:

$$\frac{\partial}{\partial w_i} E(\mathbf{w}; \mathcal{D}) \approx \frac{E(\mathbf{w} + \epsilon \mathbf{I}_i; \mathcal{D}) - E(\mathbf{w} - \epsilon \mathbf{I}_i; \mathcal{D})}{2\epsilon}$$

for a W dimensional weight vector $\mathbf{w} \in \mathbb{R}^W$ requires:

$$\left[\frac{\partial}{\partial w_1} E(\mathbf{w}; \mathcal{D}), \dots, \frac{\partial}{\partial w_W} E(\mathbf{w}; \mathcal{D}) \right]$$

$2W$ error calculations. Recall each **error calculation** requires $\mathcal{O}(BW)$ computation for B (batch-size) data points.

- **Total cost:** $\mathcal{O}(BW^2)$ per gradient step.
 - For real problems (minibatch) often $B \approx 1000$ and $W \gg 10^6 \rightarrow BW^2 \gg 10^{3+2*6} = 10^{15}$
 - Stochastic gradient with batch size 1: $W^2 \gg 10^{12}$

Symbolic Differentiation



Proposal: Symbolic Differentiation

Continue to use calculus to derive the gradient equation.

$$\frac{\partial}{\partial w_i} E(\mathbf{w}; \mathcal{D}) = - \sum_{n=1}^N \left[t_n \left(\frac{1}{y(\mathbf{x}_n, \mathbf{w})} \right) \frac{\partial y(\mathbf{x}_n, \mathbf{w})}{\partial w_i} - (1 - t_n) \left(\frac{1}{1 - y(\mathbf{x}_n, \mathbf{w})} \right) \frac{\partial y(\mathbf{x}_n, \mathbf{w})}{\partial w_i} \right]$$

for our small network: $y(\mathbf{x}, \mathbf{w}) = \sigma \left(W^{(2)} \sigma(W^{(1)} \mathbf{x}) \right)$ and using the identity:

$$\frac{\partial}{\partial t} \sigma(t) = \sigma(t)(1 - \sigma(t))$$

Then taking the partial derivative of the ij^{th} entry of $W^{(1)}$ we get:

$$\frac{\partial}{\partial W_{ij}^{(1)}} y(\mathbf{x}, \mathbf{w}) = \sigma \left(W^{(2)} \sigma(W^{(1)} \mathbf{x}) \right) \left(1 - \sigma \left(W^{(2)} \sigma(W^{(1)} \mathbf{x}) \right) \right) W_{ij}^{(2)} \frac{\partial}{\partial W_{ij}^{(1)}} \sigma(W^{(1)} \mathbf{x})$$

$$\frac{\partial}{\partial W_{ij}^{(1)}} y(\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w}) (1 - y(\mathbf{x}, \mathbf{w})) W_{1i}^{(2)} \sigma(W^{(1)} \mathbf{x})_i \left(1 - \sigma(W^{(1)} \mathbf{x})_i \right) x_j$$

Painful and error prone (it is easy to get an index wrong), and **not automatic**.



Symbolic Computer Algebra Systems

Computer algebra systems like SymPy can be used to automate the differentiation process.

- Express functions **symbolically**
- Automatically construct **derivative expressions**

```
n = sp.symbols('n', integer=True)      # number of data points (symbolic length)
i = sp.Idx('i', n)                     # index variable for summation
w0, w1 = sp.symbols('w0 w1')          # weights
x = sp.IndexedBase('x', shape=(n,))    # indexed variable for x data
y = sp.IndexedBase('y', shape=(n,))    # indexed variable for y data

E_i = (y[i] - sp.sin(w0 + w1 * x[i]))**2
E = sp.summation(E_i, (i, 0, n - 1)) / n

gE = [sp.diff(E, var) for var in (w0, w1)]
gE

[Sum(-2*(-sin(w0 + w1*x[i]) + y[i])*cos(w0 + w1*x[i]), (i, 0, n - 1))/n,
Sum(-2*(-sin(w0 + w1*x[i]) + y[i])*cos(w0 + w1*x[i])*x[i], (i, 0, n - 1))/n]
```



9803087

Demo

Numerical Differentiation + SymPy

```
import sympy as sp
# define our symbols
x1, x2 = sp.symbols('x1 x2')
# define a symbolic expression for the error
E = x1 * x2 + sp.exp(x1 * x2) - sp.sin(x2)
# compute the gradient of E with respect to x1 and x2
gE = [sp.diff(E, var) for var in (x1, x2)]
gE
```

Symbolic Computer Algebra Systems



Computer algebra systems like SymPy can be used to automate the differentiation process.

- Express functions **symbolically**
- Automatically construct **derivative expressions**

Advantages:

- **Automatic:** algebraic systems can be applied to “any” expression
 - Some issues with error functions that contain control flow/loops.
- **Exact:** Doesn't require epsilon approximation

Disadvantages:

- **Inefficient:** algebraic derivative expressions could be significantly larger and contain many repeated expressions:
 - Why?



Expression Swell

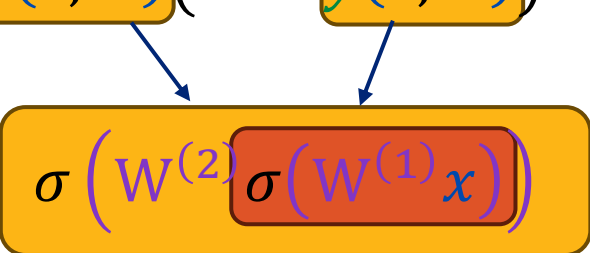
Consider computing the derivative of an expression like:

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

If we apply the product rule, we get:

$$\frac{\partial}{\partial x} f(x) = h(x)u(x) \frac{\partial}{\partial x} g(x) + g(x)u(x) \frac{\partial}{\partial x} h(x) + g(x)h(x) \frac{\partial}{\partial x} u(x)$$

Notice that there is **significant redundancy**. This is especially the case with neural networks (from earlier):

$$\frac{\partial}{\partial W_{ij}^{(1)}} y(\mathbf{x}, \mathbf{w}) = \boxed{y(\mathbf{x}, \mathbf{w})} (1 - \boxed{y(\mathbf{x}, \mathbf{w})}) W_{1i}^{(2)} \boxed{\sigma(W^{(1)} \mathbf{x})_i} (1 - \boxed{\sigma(W^{(1)} \mathbf{x})_i}) x_j$$


The diagram illustrates the simplification of the derivative expression. It shows the derivative of the output $y(\mathbf{x}, \mathbf{w})$ with respect to the weight $W_{ij}^{(1)}$. The expression is broken down into several components, some of which are highlighted in boxes. Two arrows point from the boxed terms $y(\mathbf{x}, \mathbf{w})$ and $1 - y(\mathbf{x}, \mathbf{w})$ to a single box containing the expression $\sigma(W^{(2)} \sigma(W^{(1)} \mathbf{x}))$, indicating that these terms can be simplified together.

Automatic Differentiation



Automatic Differentiation (Autodiff)

Automatically generate a **program** to compute the derivative **efficiently** from the “**forward**” **computation** of the error.

- **Automatic:** derivative program is constructed by tracing the forward computation graph.
- **Accurate:** derivative is computed to machine precision.
- **Efficient:** reuse redundant computation

There are two approaches to Autodiff:

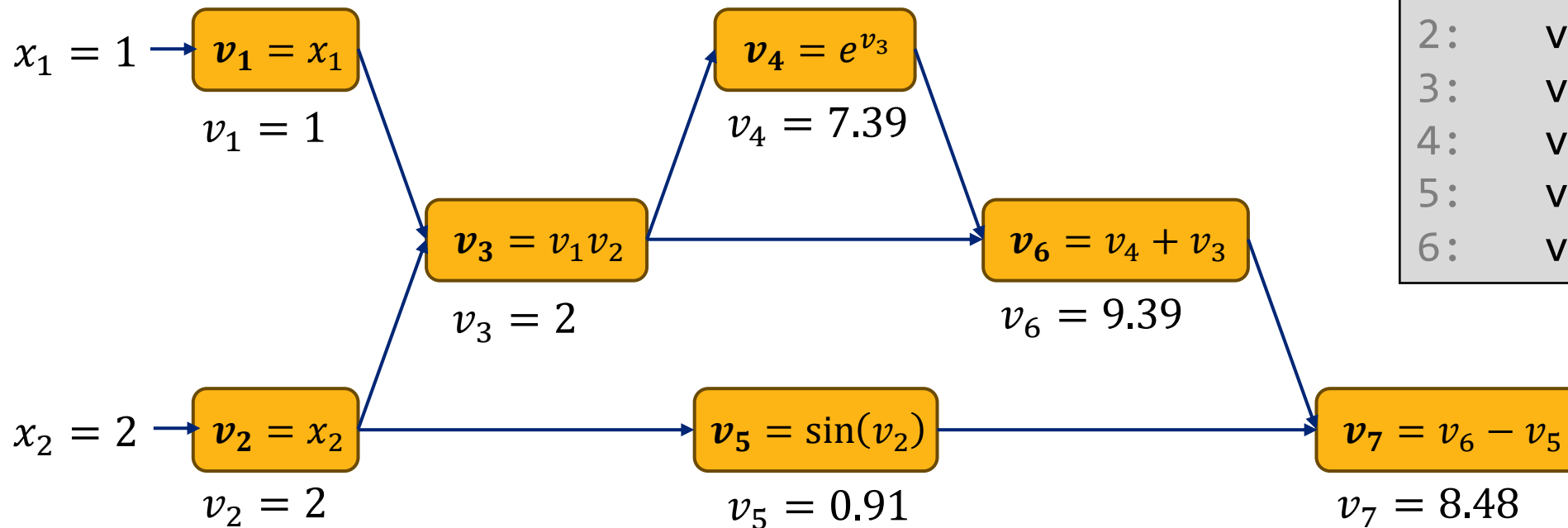
- **Forward Mode:** Best for $f: \mathbb{R} \rightarrow \mathbb{R}^D$
- **Reverse Mode:** Best for $f: \mathbb{R}^D \rightarrow \mathbb{R}^1$ (deep learning)

Forward Tracing the Computation Graph



We construct the graph by evaluating the forward computation:

$$f(x_1, x_2) = x_1 x_2 + e^{(x_1 x_2)} - \sin(x_2)$$



Program:

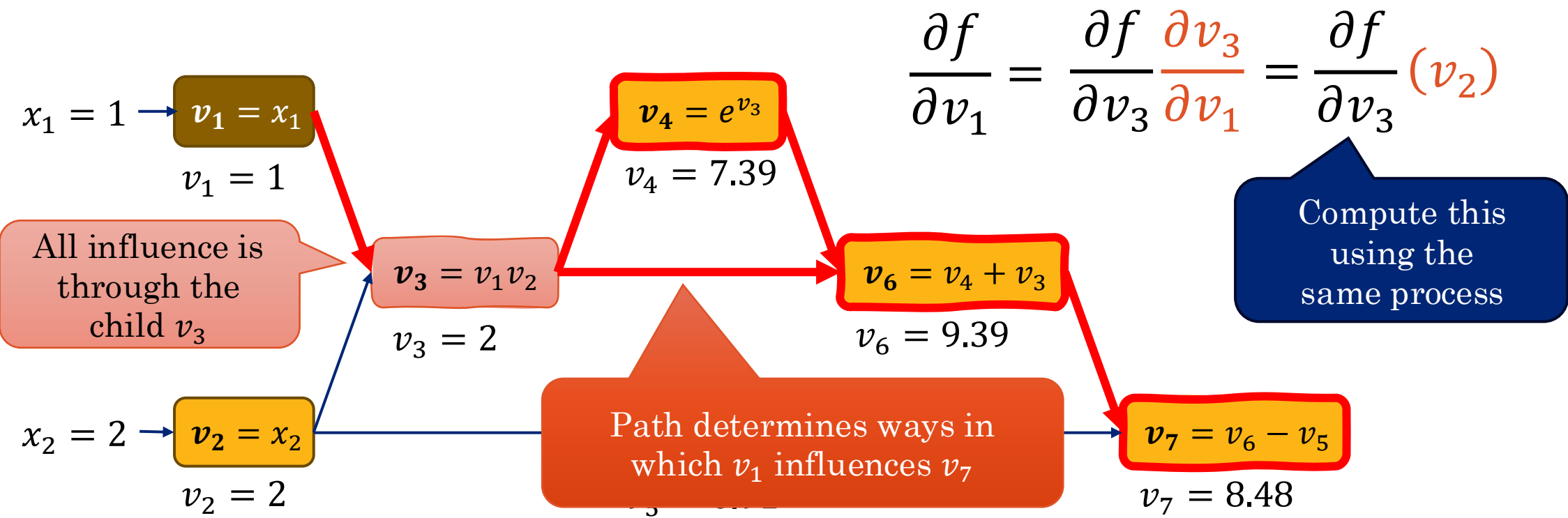
```
0:    v1 = x1
1:    v2 = x2
2:    v3 = v1 * v2
3:    v4 = exp(v3)
4:    v5 = sin(v2)
5:    v6 = v4 + v3
6:    v7 = v6 - v5
```

In this program we have broken each expression into a separate line.



Differentiating wrt v_1 Using the Chain Rule

Computing the derivative of $\frac{\partial f}{\partial v_1}$ by observing the **path** from v_1 to v_7 .

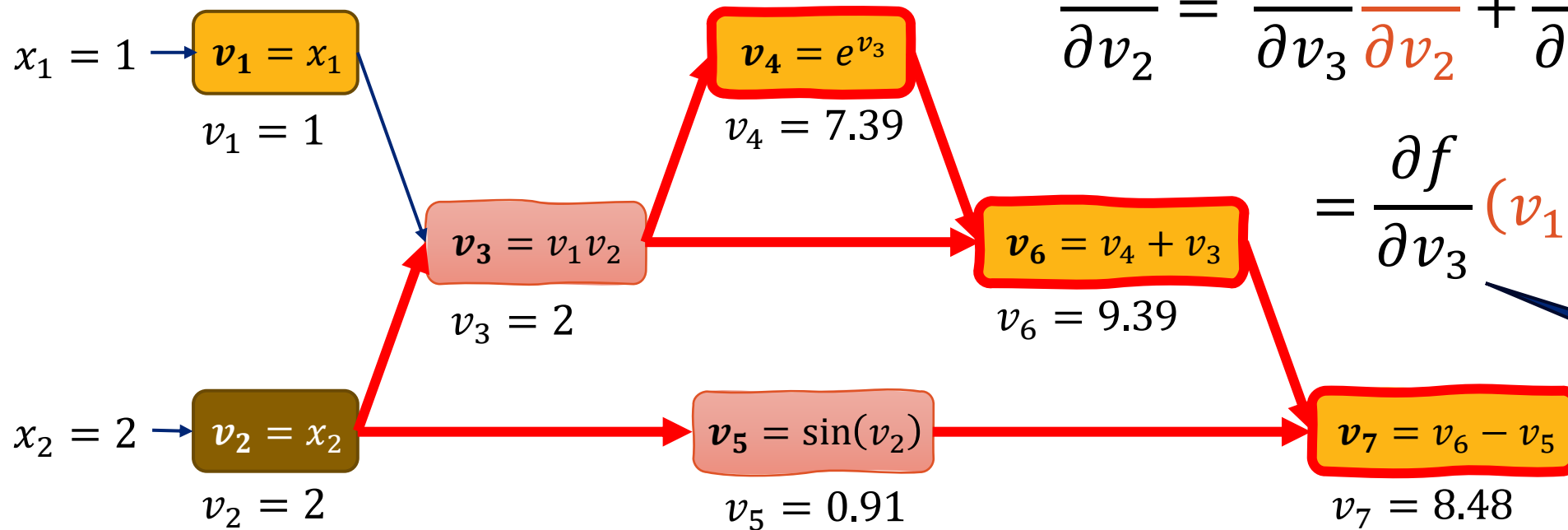


$$f(x_1, x_2) = x_1 x_2 + e^{(x_1 x_2)} - \sin(x_2)$$



Differentiating wrt v_2 Using the Chain Rule

Computing the derivative of $\frac{\partial f}{\partial v_2}$ by observing the paths from v_2 to v_7 .



$$\frac{\partial f}{\partial v_2} = \frac{\partial f}{\partial v_3} \frac{\partial v_3}{\partial v_2} + \frac{\partial f}{\partial v_5} \frac{\partial v_5}{\partial v_2}$$

$$= \frac{\partial f}{\partial v_3} (v_1) + \frac{\partial f}{\partial v_5} \cos(v_2)$$

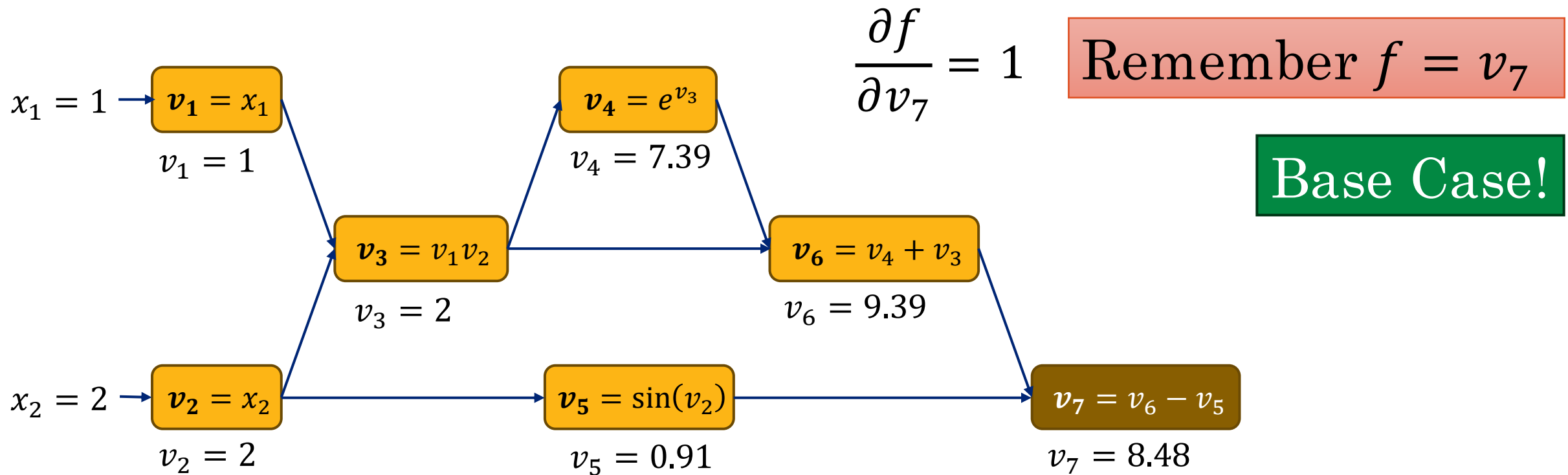
Compute these using the same process

$$f(x_1, x_2) = x_1 x_2 + e^{(x_1 x_2)} - \sin(x_2)$$



Differentiating wrt v_7 Using the Chain Rule

Computing the derivative of $\frac{\partial f}{\partial v_7}$ by observing the paths from v_7 to v_7 .

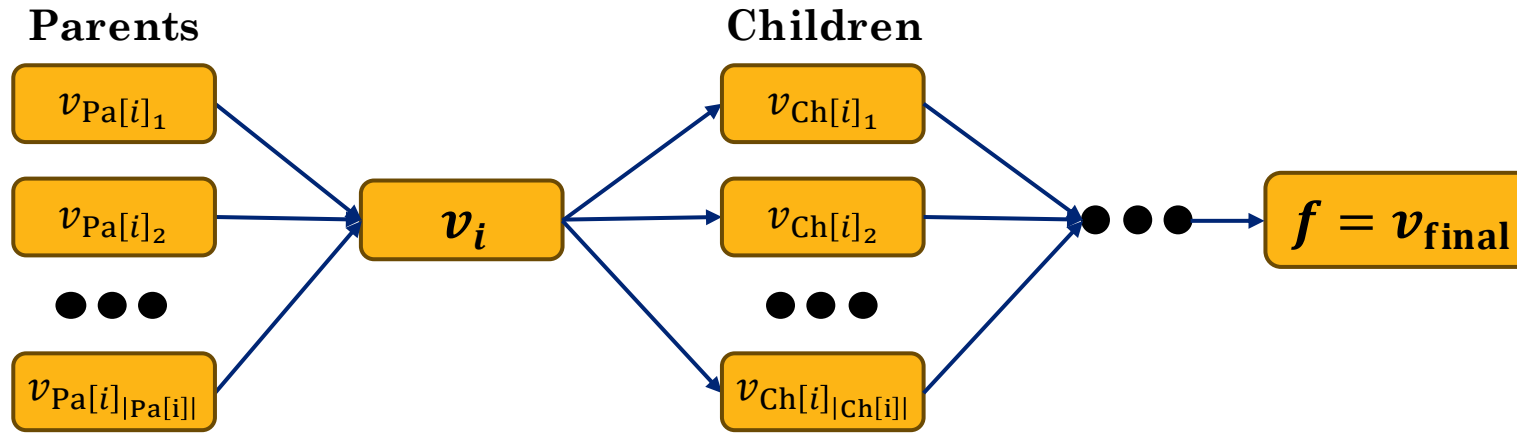


$$f(x_1, x_2) = x_1 x_2 + e^{(x_1 x_2)} - \sin(x_2)$$



Backward Propagation (Recursion)

Consider the computation graph around v_i with a **single output** $f = v_{\text{final}}$



Define **adjoint variable** a_i^f and use the **multivariate chain rule** to obtain:

Recursion:
$$a_i^f = \frac{\partial f}{\partial v_i} = \sum_{j \in \text{Ch}[i]} \left(\frac{\partial f}{\partial v_j} \right) \left(\frac{\partial v_j}{\partial v_i} \right) = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

Where $\text{Ch}[i]$ are the children of vertex i in the computation graph.



Intuition Behind the Multivariate Chain Rule

For the following, how does z change for a small change dt in t ?

$$z = f(g(t), h(t))$$

We take a local first order approximation:

$$dz = \frac{\partial f}{\partial g} dg + \frac{\partial f}{\partial h} dh$$

Small changes in response to each input accumulate.

Applying the same first order approximation to dg and dh

$$dg = \frac{\partial g}{\partial t} dt; \quad dh = \frac{\partial h}{\partial t} dt$$

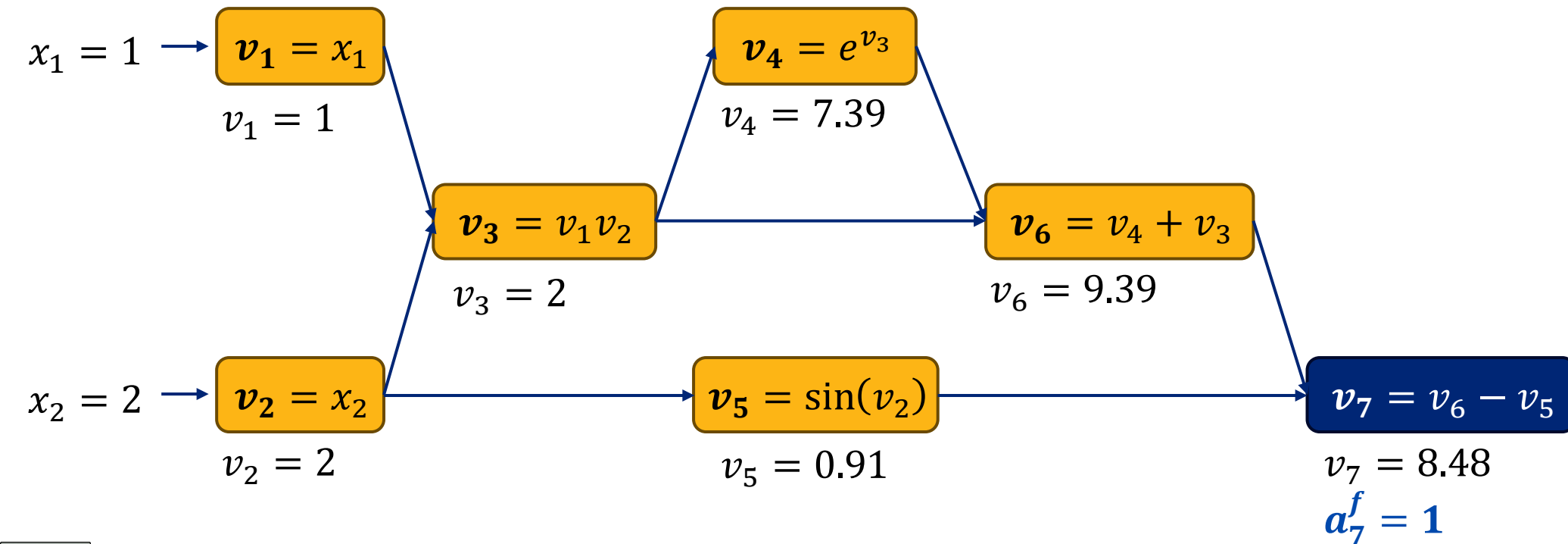
Substituting dg and dh and then dividing both sides by dt :

$$dz = \frac{\partial f}{\partial g} \frac{\partial g}{\partial t} dt + \frac{\partial f}{\partial h} \frac{\partial h}{\partial t} dt \Rightarrow \frac{dz}{dt} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial t} + \frac{\partial f}{\partial h} \frac{\partial h}{\partial t}$$

Backward Autodiff Example

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

- ➔ 1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$
2. Compute the adjoint for each variable **working backward**.

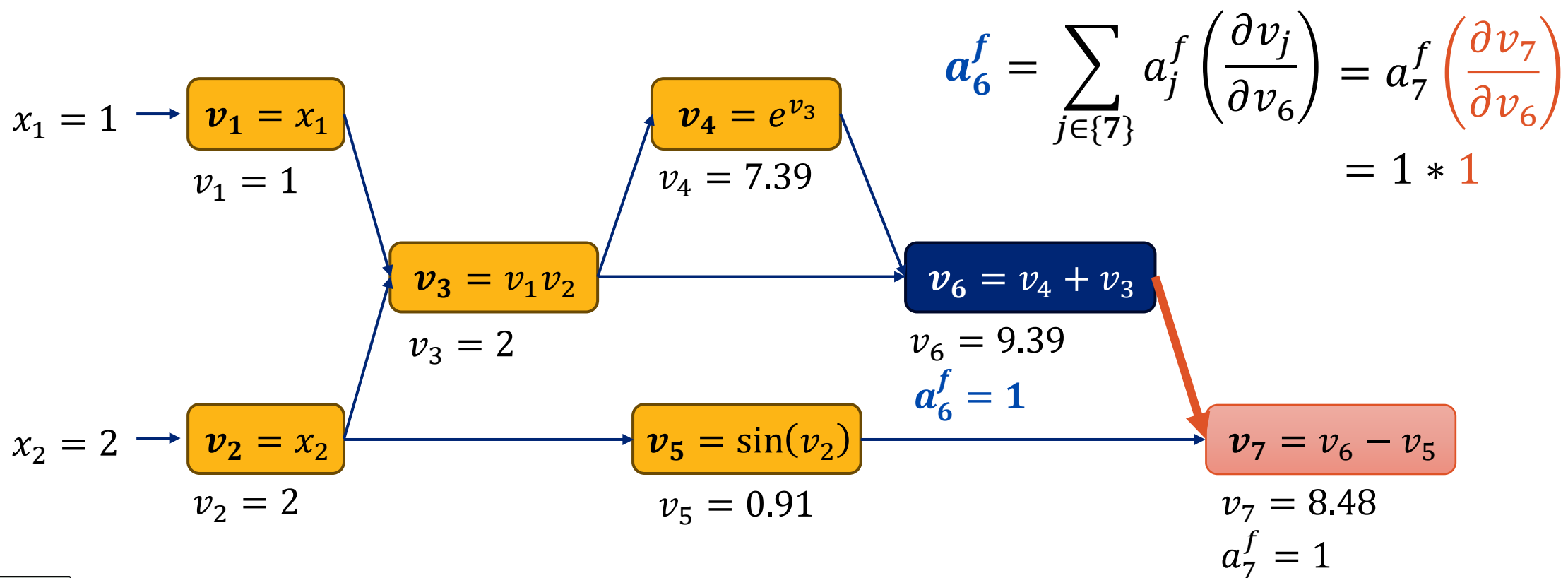


Backward Autodiff a_6^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.



9803087

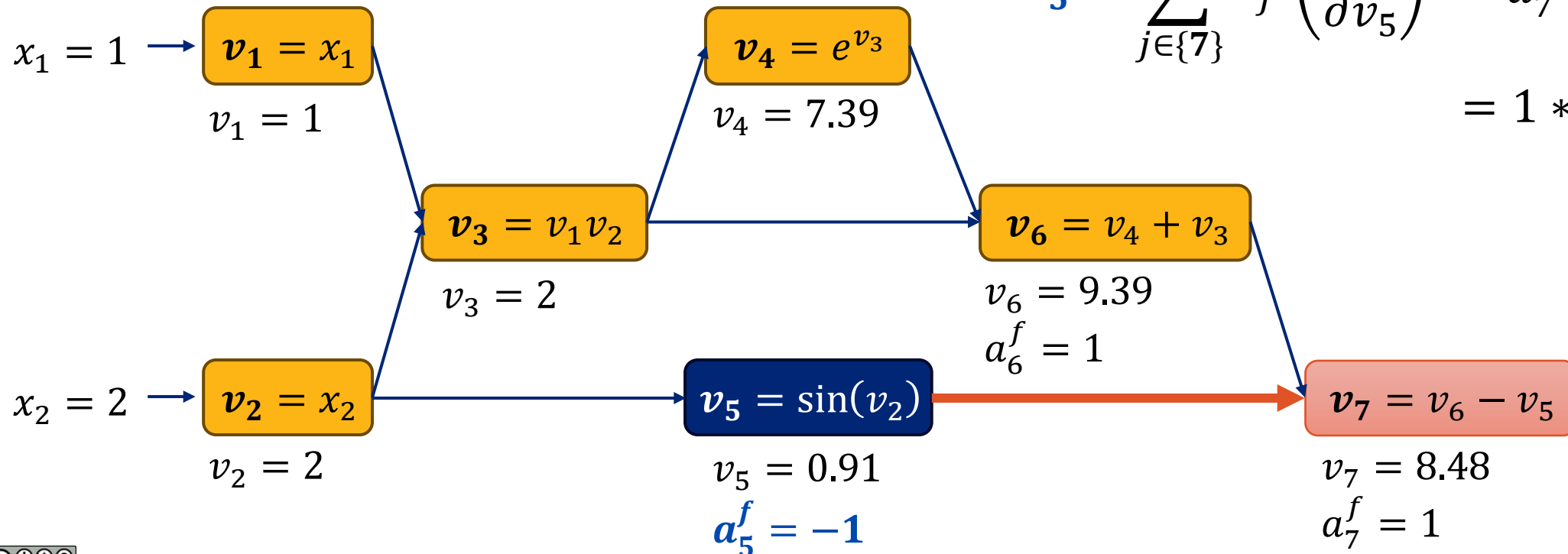
Backward Autodiff a_5^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.

$$a_5^f = \sum_{j \in \{7\}} a_j^f \left(\frac{\partial v_j}{\partial v_5} \right) = a_7^f \left(\frac{\partial v_7}{\partial v_5} \right) = 1 * (-1)$$



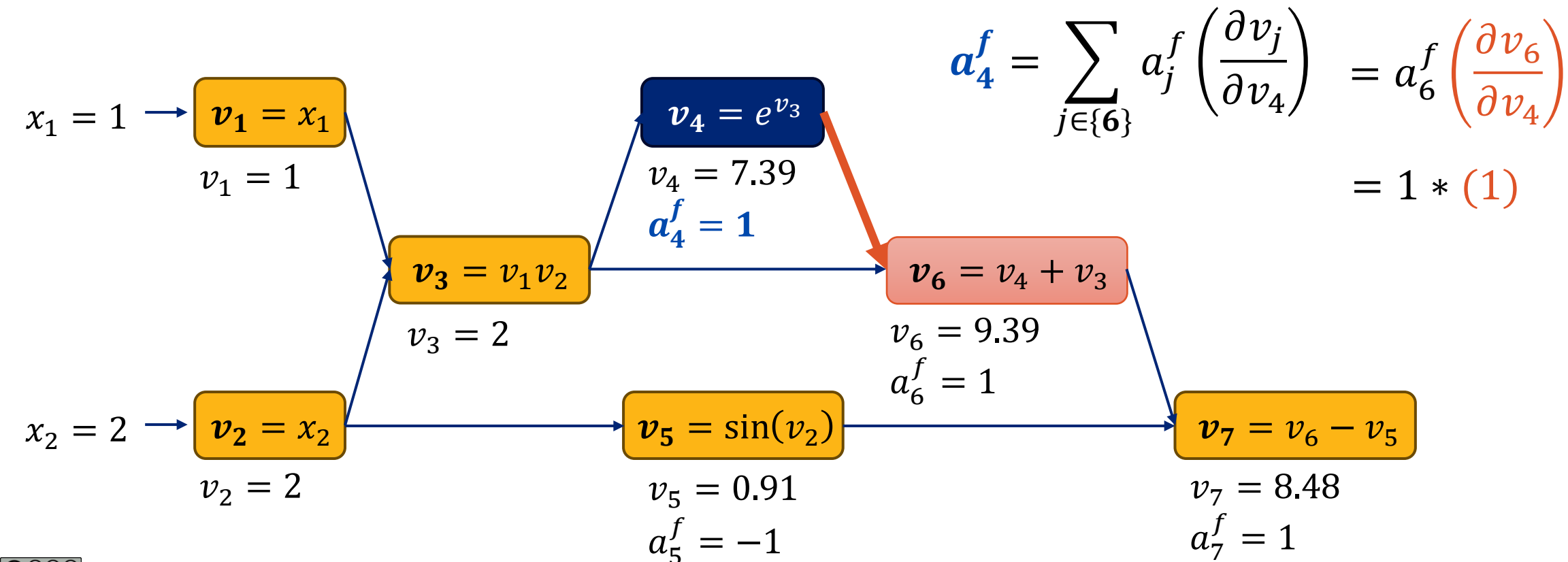
9803087

Backward Autodiff a_4^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.



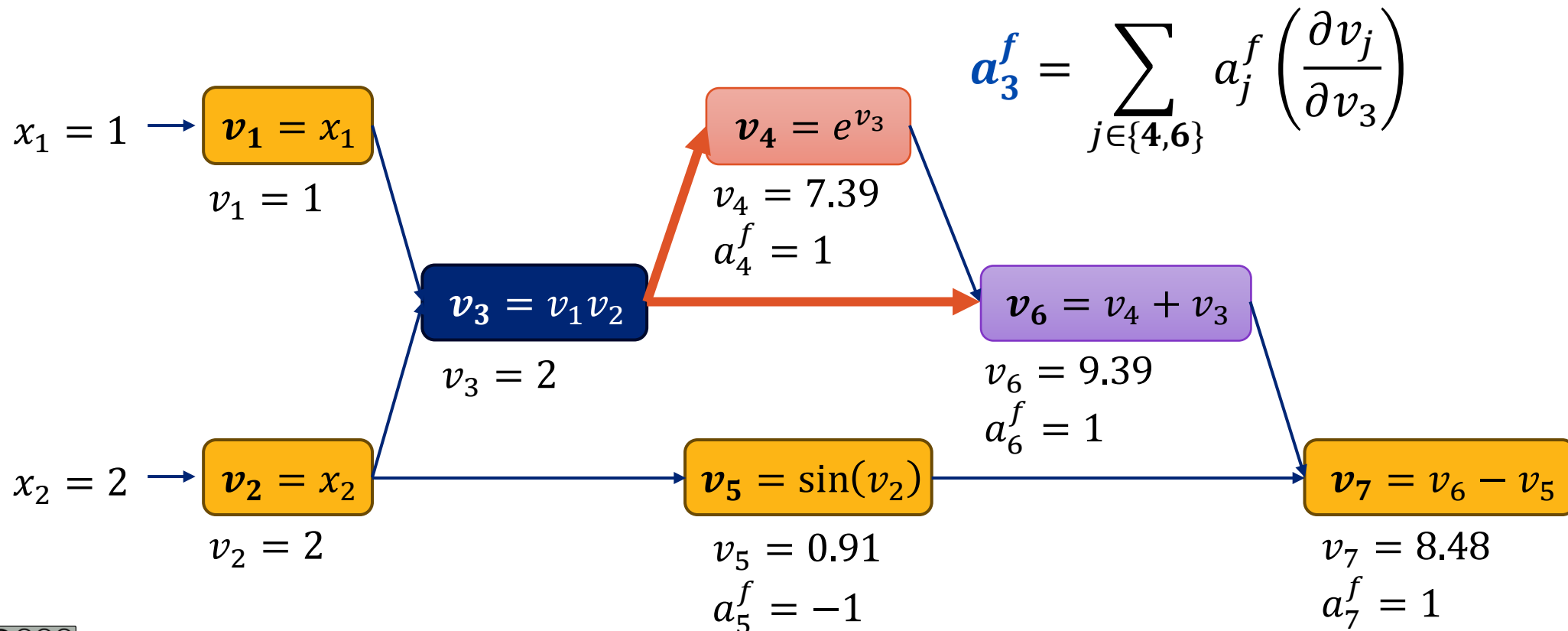
9803087

Backward Autodiff a_3^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.



9803087



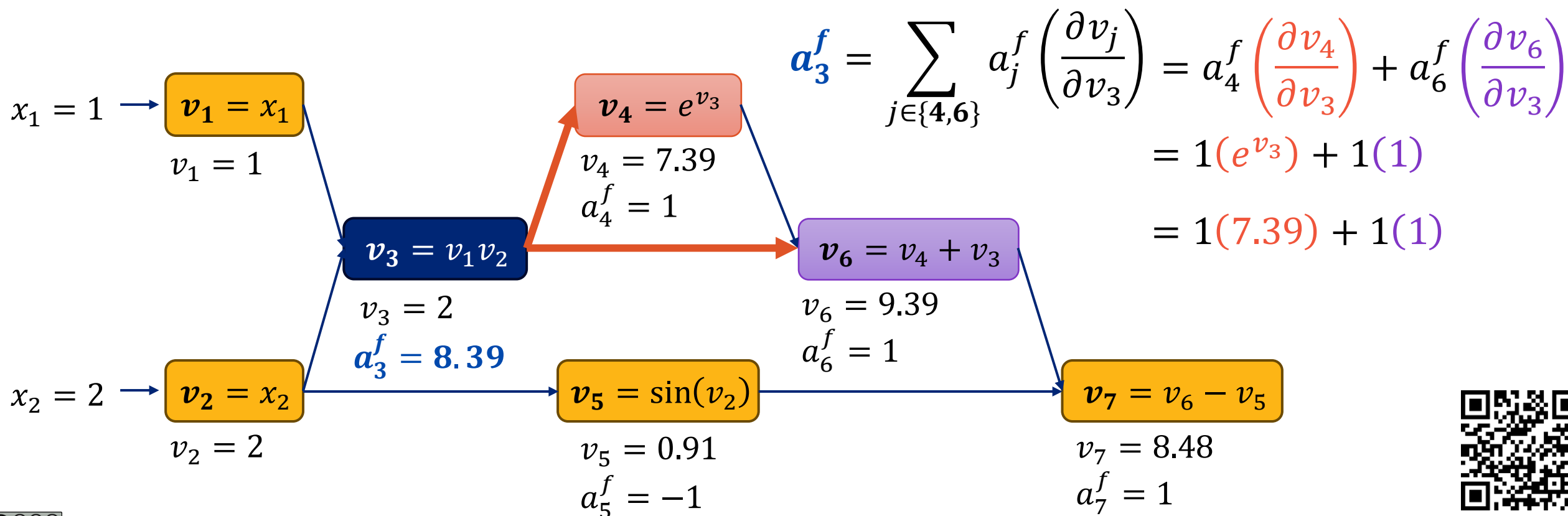
What is the value of the adjoint variable a_3 ?

Backward Autodiff a_3^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.

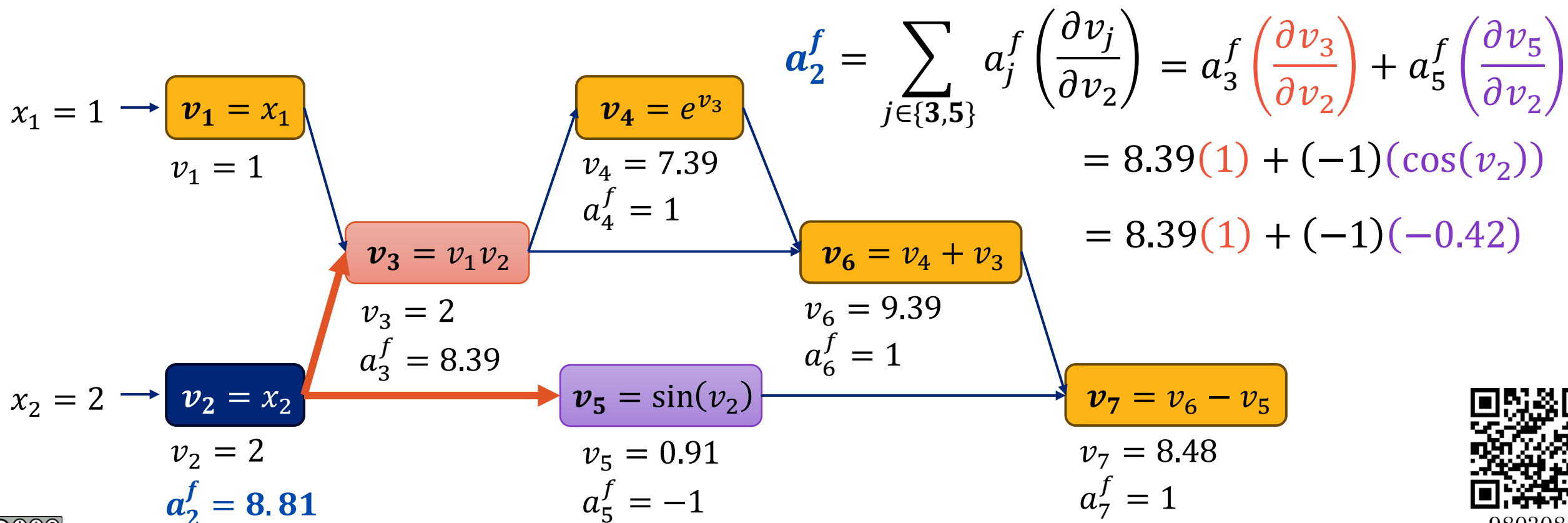


Backward Autodiff a_2^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➡ 2. Compute the adjoint for each variable **working backward**.



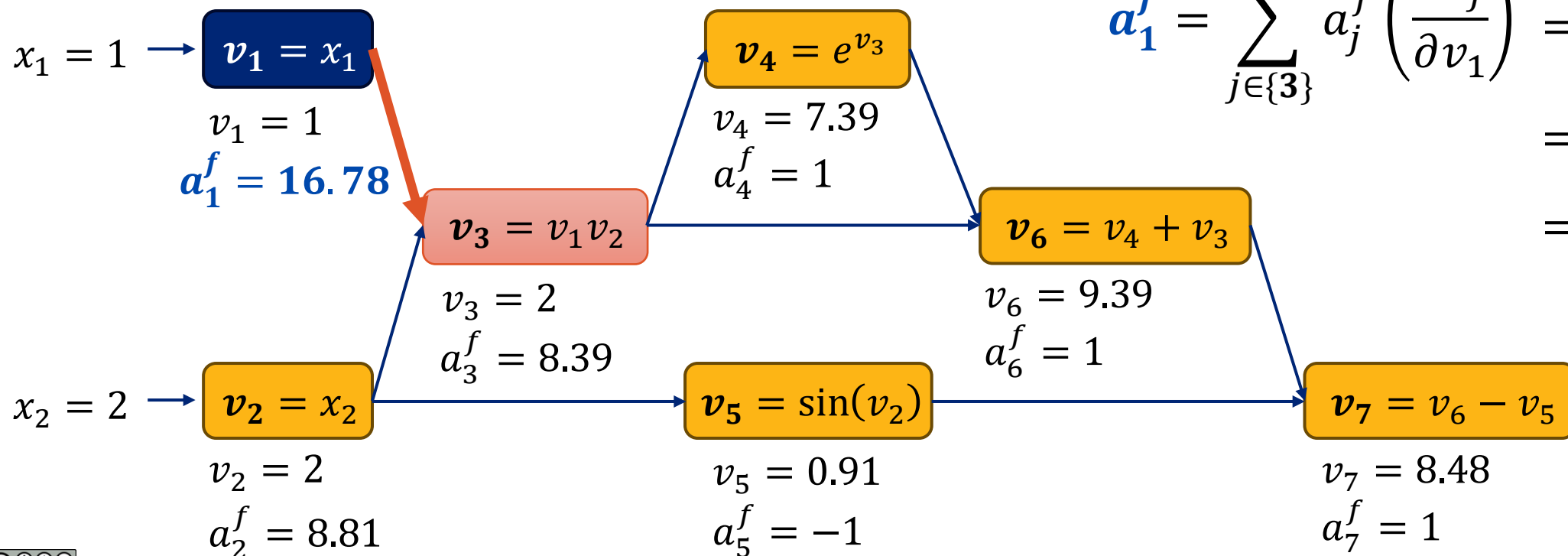
9803087

Backward Autodiff a_1^f

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$

➔ 2. Compute the adjoint for each variable **working backward**.



$$\begin{aligned} a_1^f &= \sum_{j \in \{3\}} a_j^f \left(\frac{\partial v_j}{\partial v_1} \right) = a_3^f \left(\frac{\partial v_3}{\partial v_1} \right) \\ &= 8.39 (v_2) \\ &= 8.39 (2) \end{aligned}$$

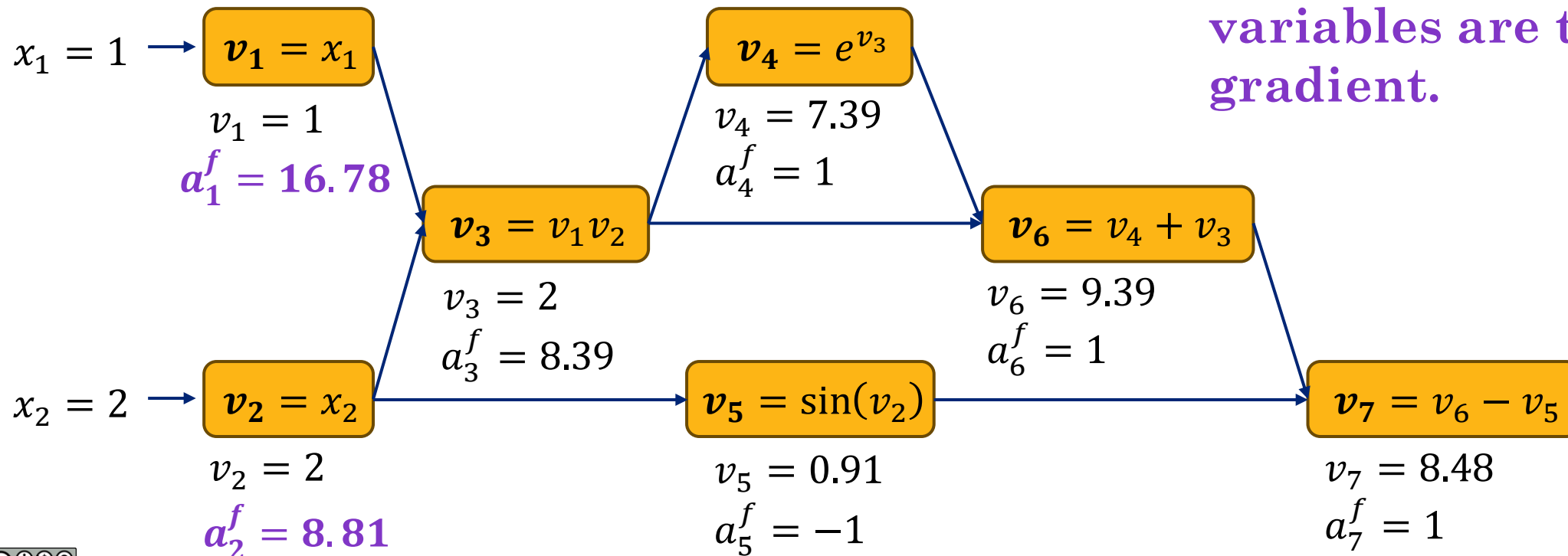


Obtaining the Gradient

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

1. Initialize the final adjoint $a_f^f = \frac{\partial f}{\partial f} = 1$
2. Compute the adjoint for each variable **working backward**.

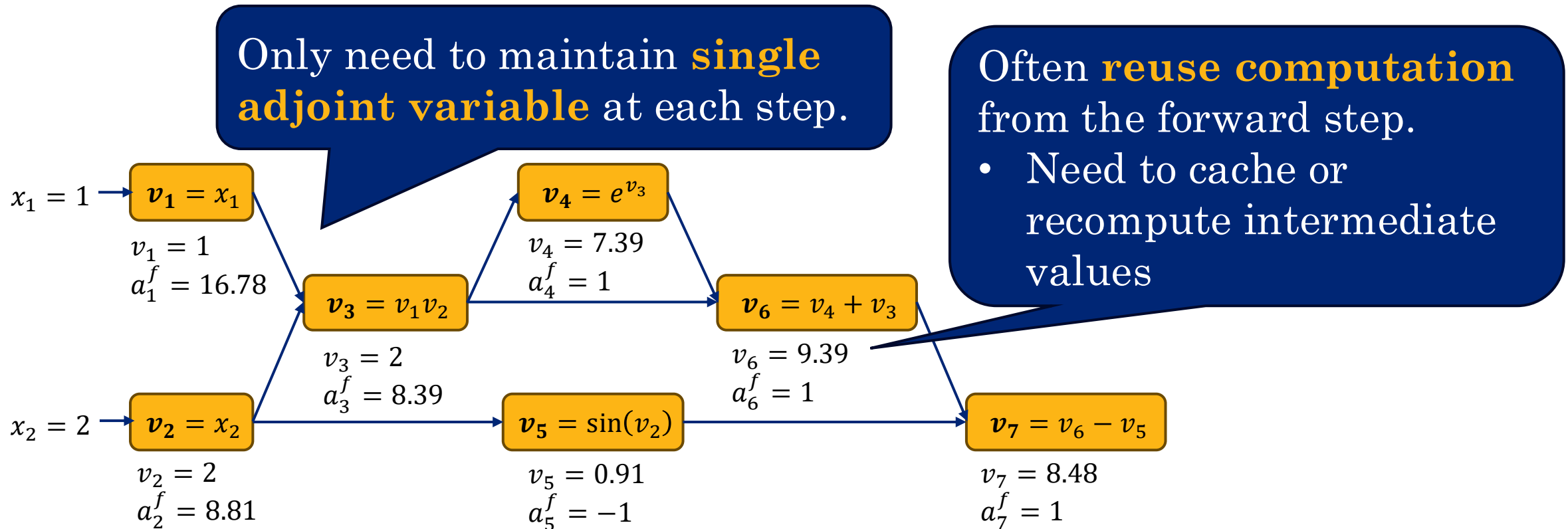
➡ 3. The final input adjoint variables are the gradient.



Cost Analysis



Backpropagation is efficient when there is a **single output** and **many inputs**.



Need to run **backpropagation repeatedly** for multiple outputs.



Cost of Computing the Gradient

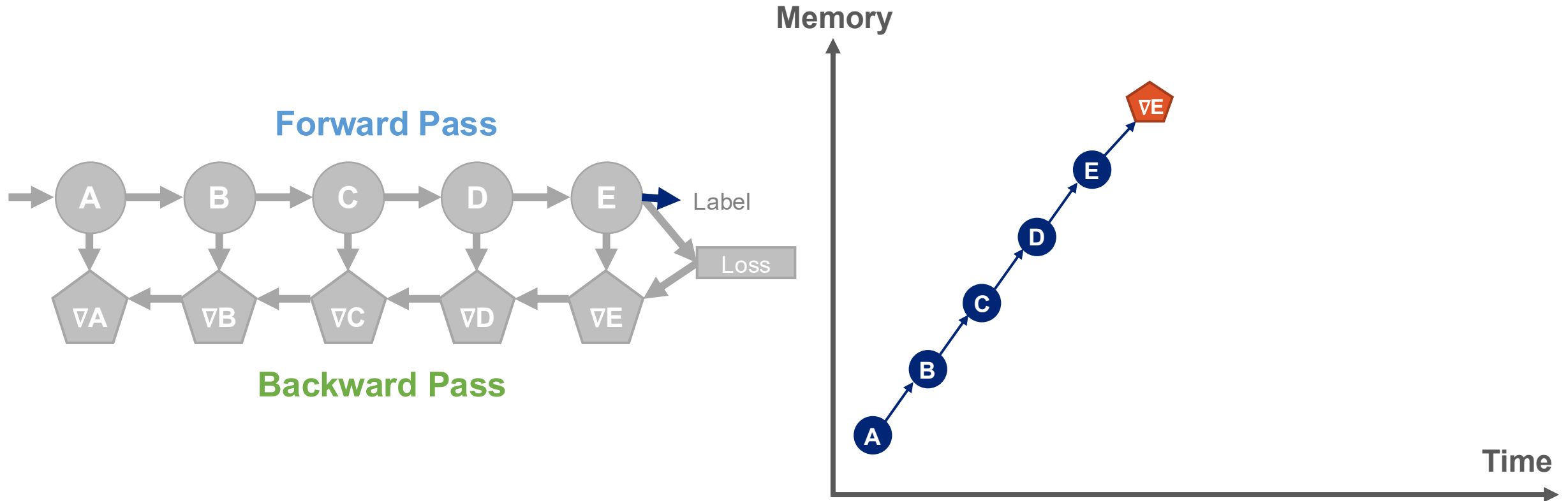
For a network with W weights, evaluating the error for one data point costs $\mathcal{O}(W)$ (Bishop-8.1.4). For the gradient:

- **Numerical differentiation:** $2W$ error evaluations, $\mathcal{O}(W^2)$.
- **Symbolic differentiation:** expression swell; shared subexpressions are repeated.
- **Backpropagation (reverse mode autodiff) :** one forward and one backward pass, $\mathcal{O}(W)$.

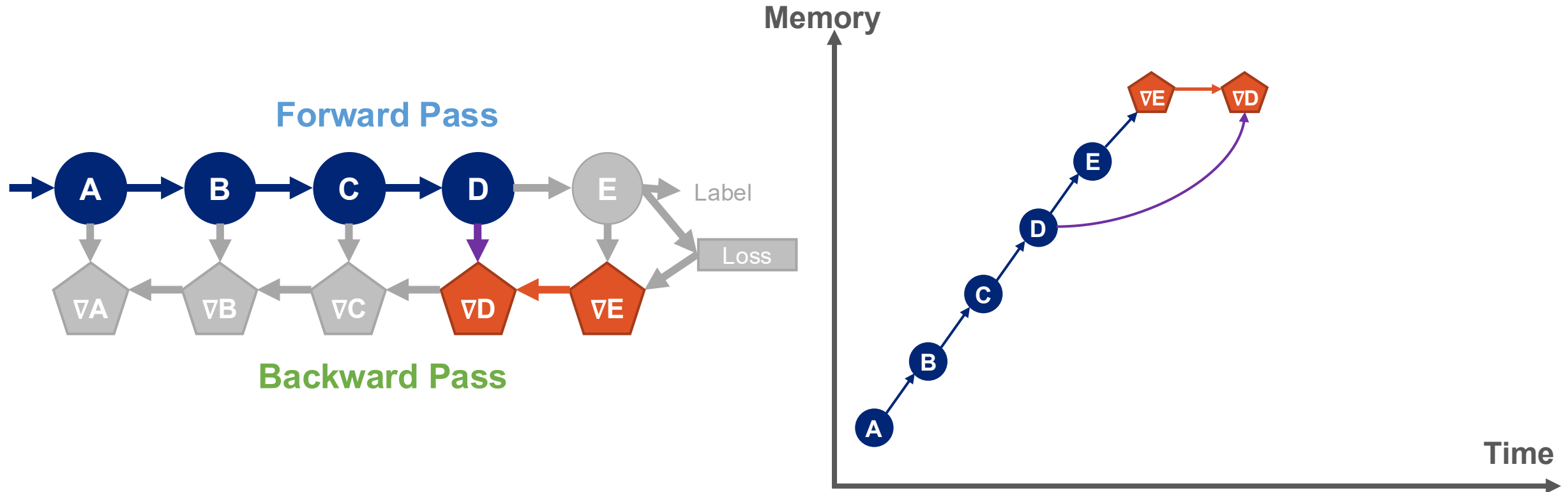
The **hidden cost of backpropagation** is **memory**: all the intermediate values of the forward pass are must be **stored** (or **recomputed**) for the backward pass.

- For modern hardware, **memory** (not compute) is often a bottleneck and so we often keep some of the activations and **recompute the rest**.

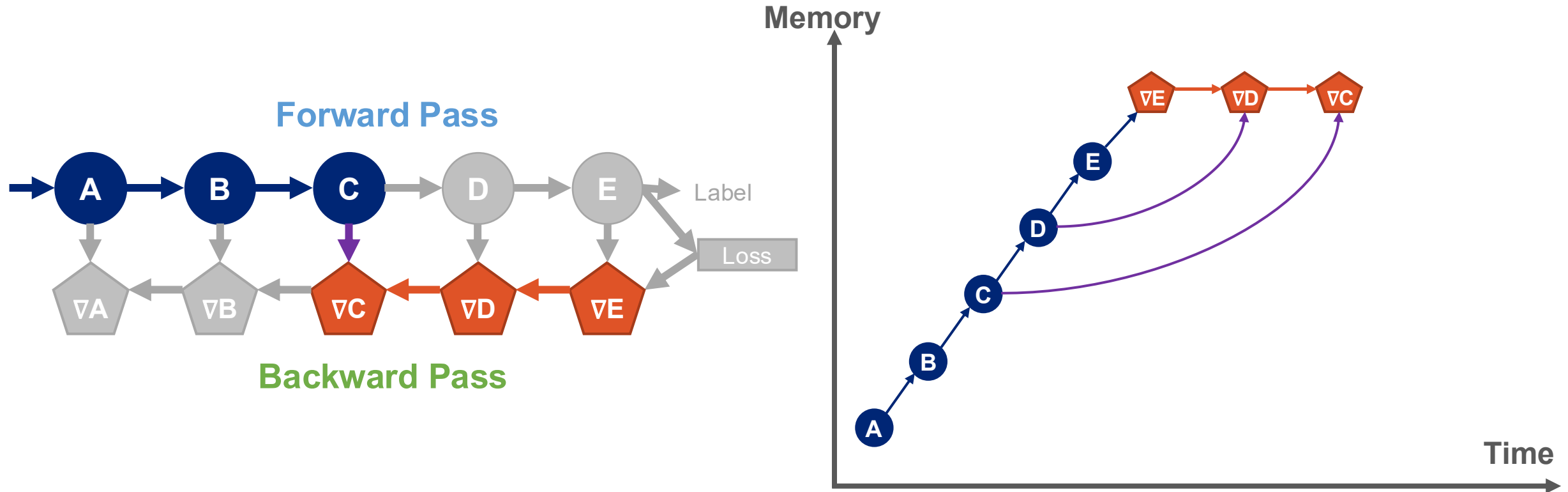
Computing Activations During the Forward Pass



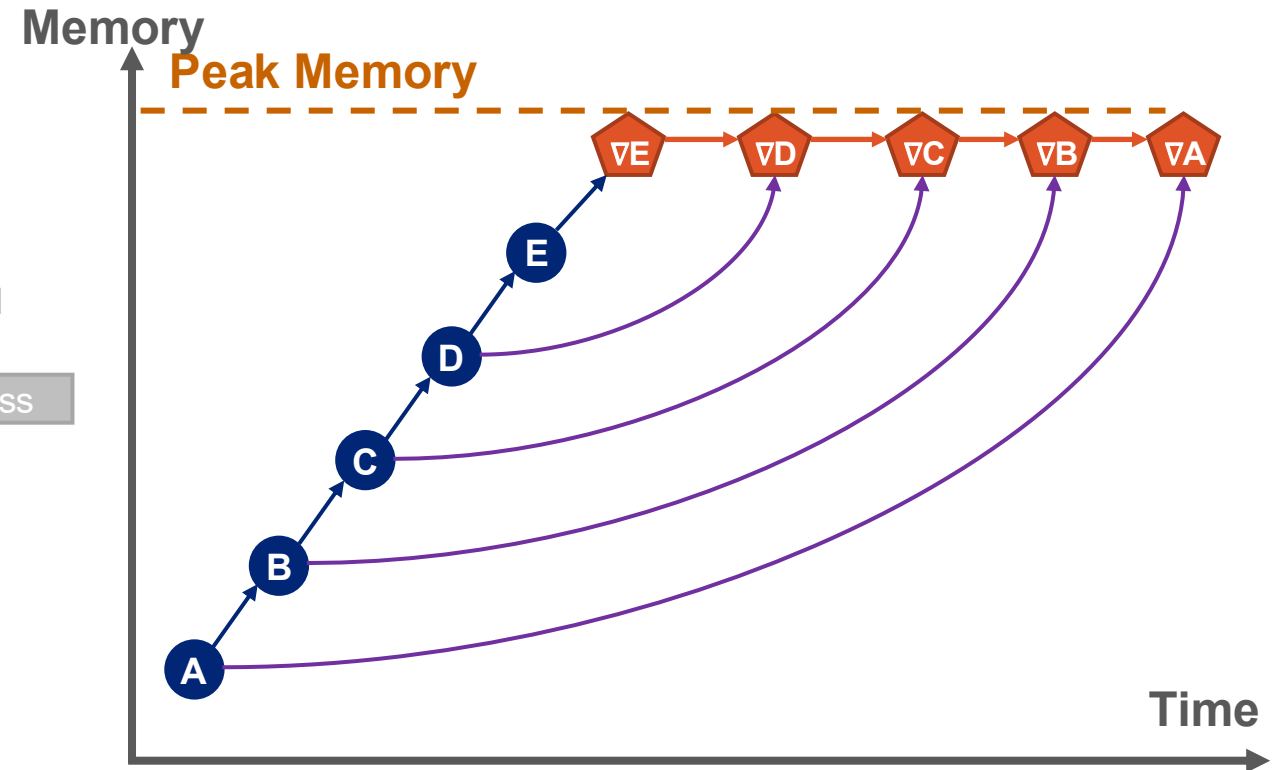
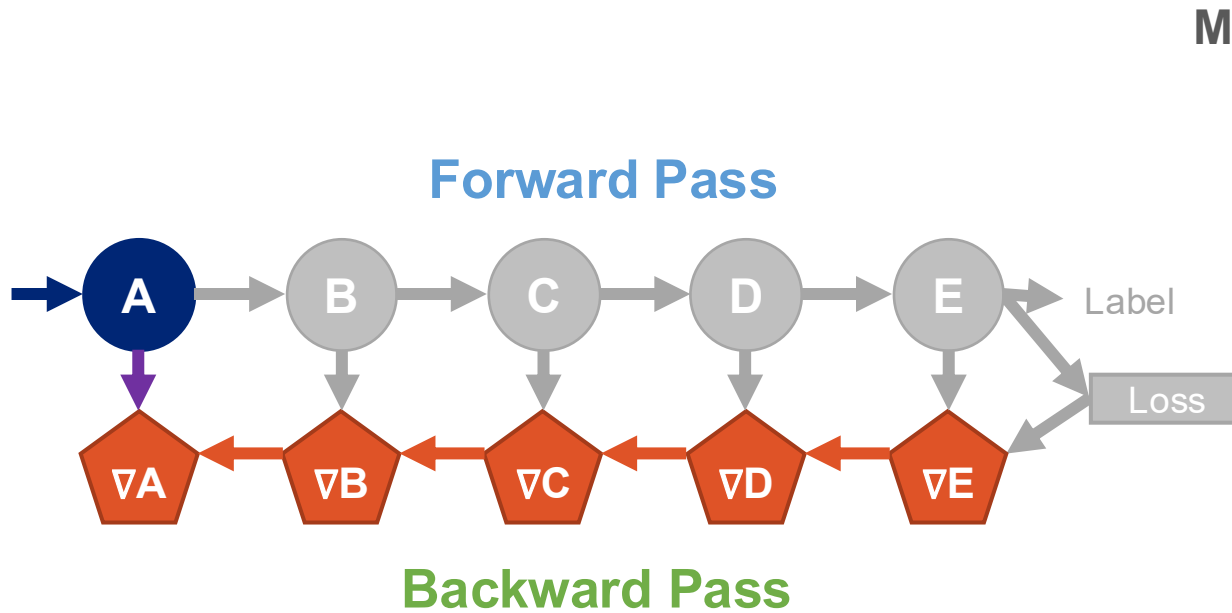
Using Activations During Backward Pass



Using Activations During Backward Pass



Using Activations During Backward Pass

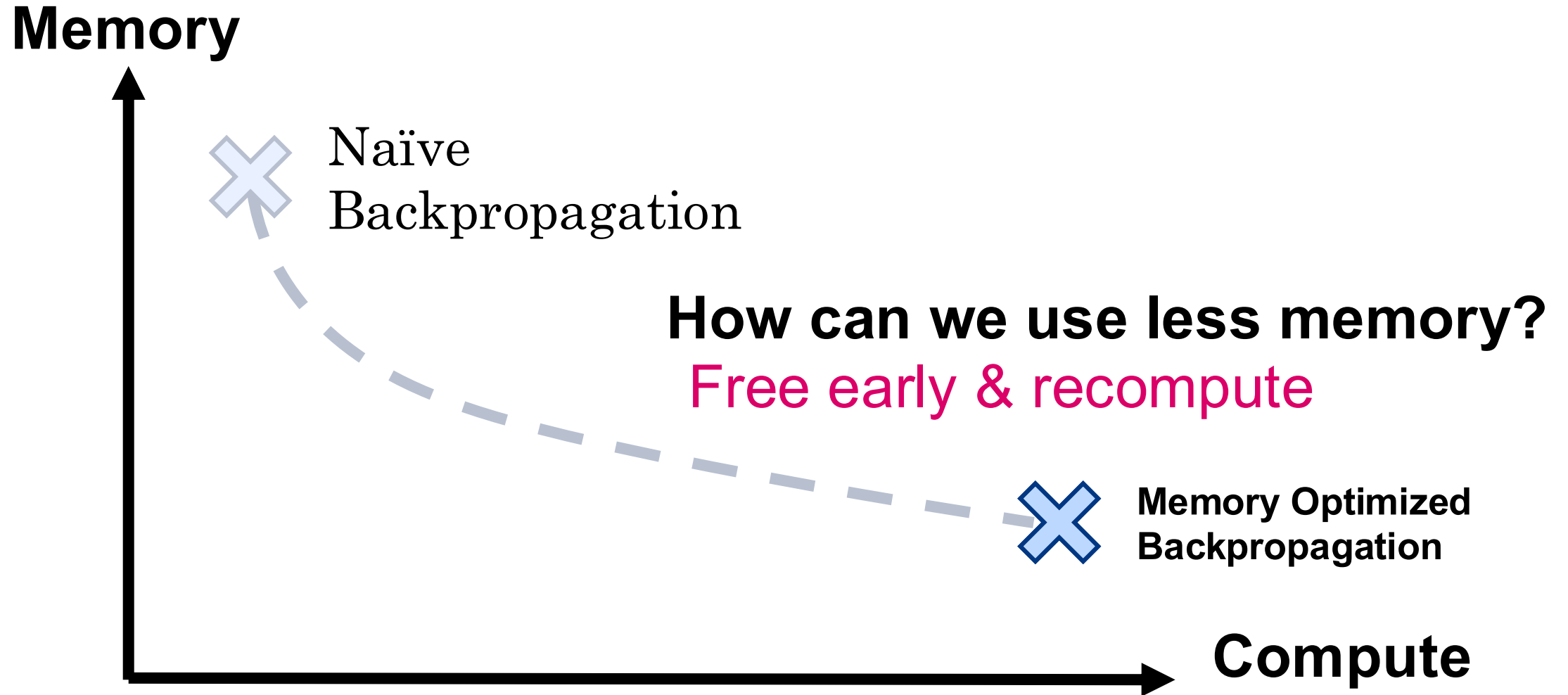


Memory-optimized backpropagation policy

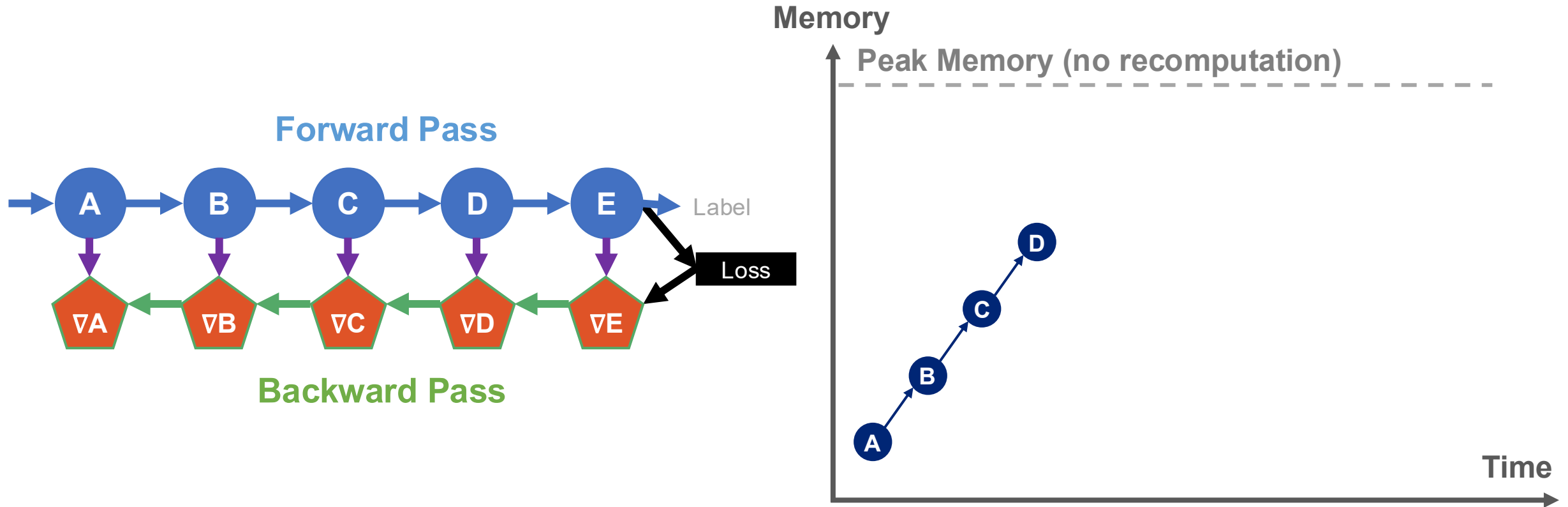
Recompute layers as needed



9803087



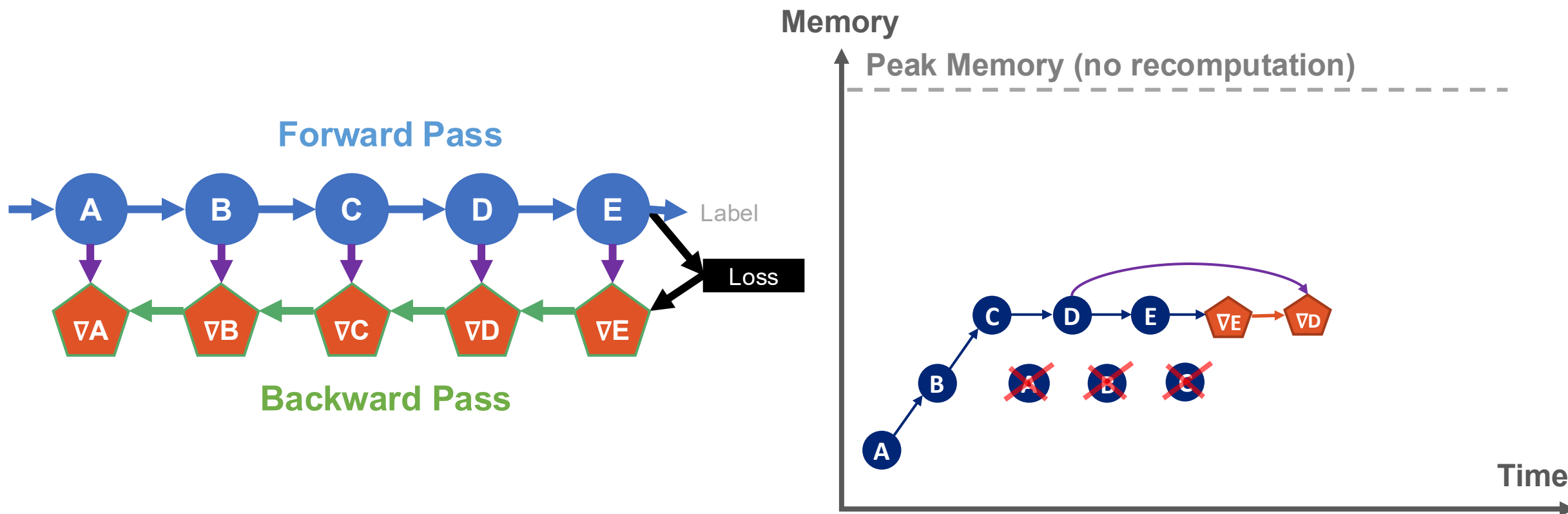
Limit Memory During Forward Pass



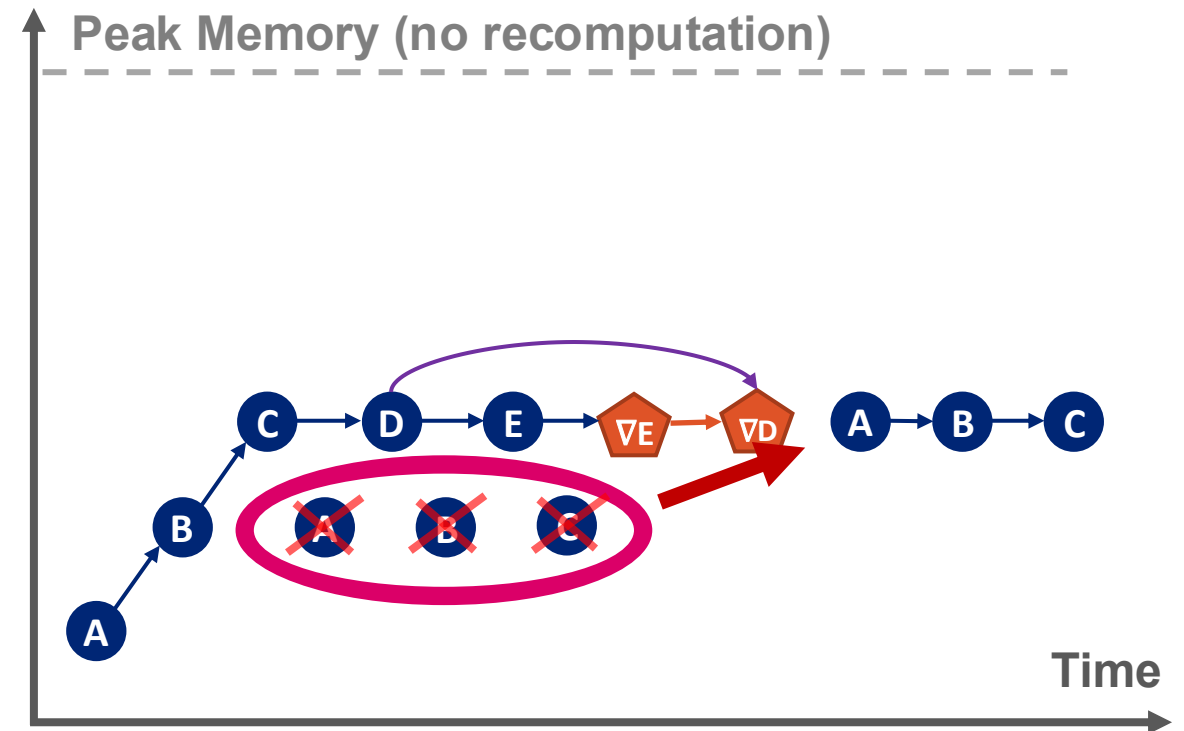
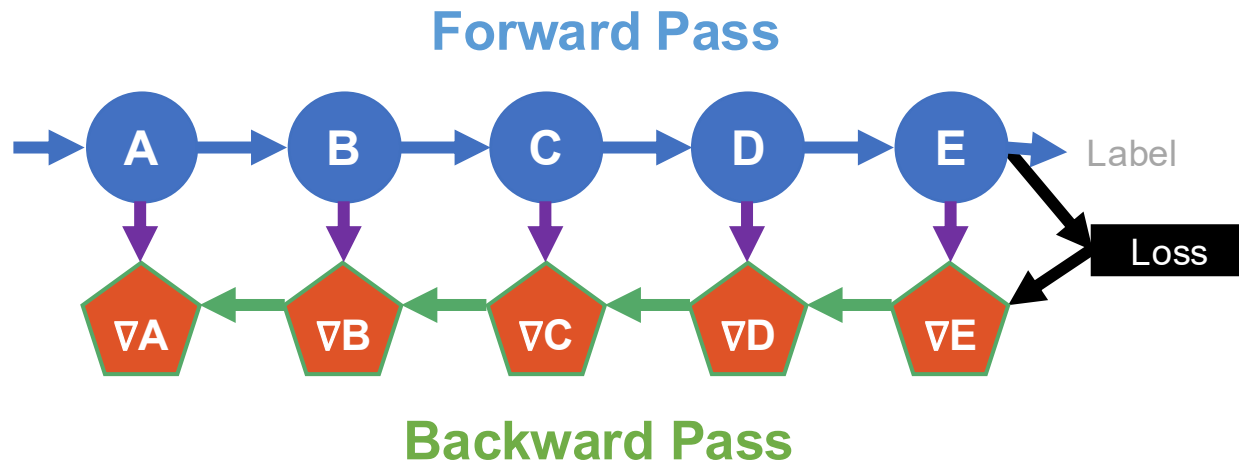
Clear Activations from Memory



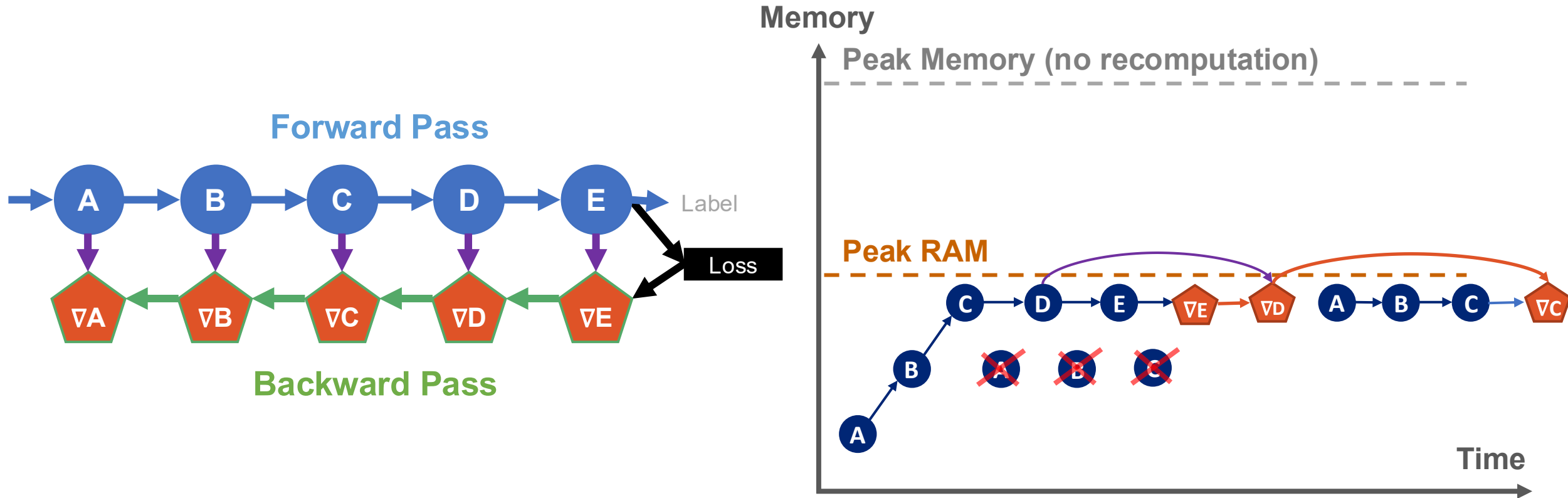
9803087



Recompute Activations During Backprop

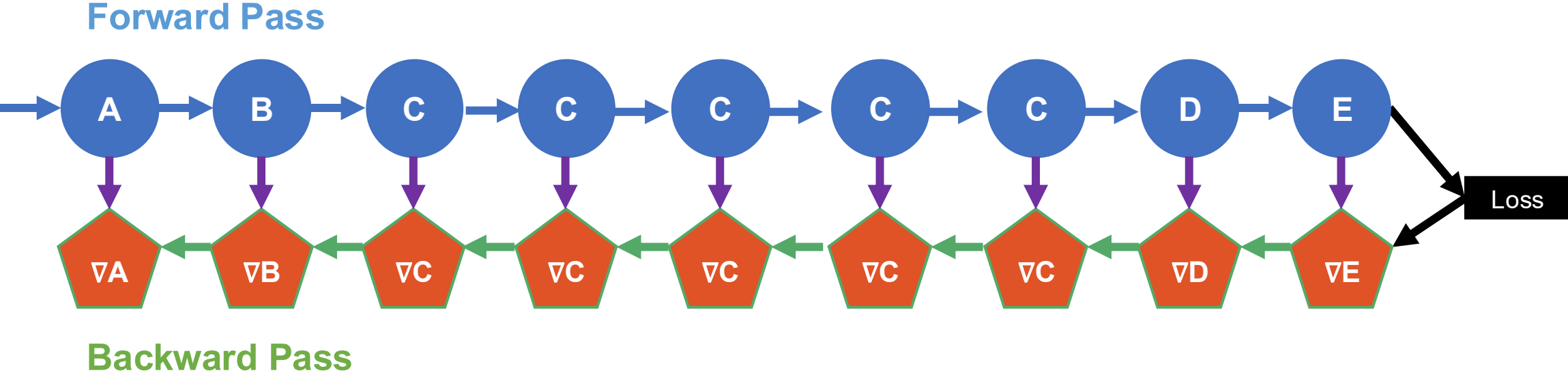


Recompute Activations During Backprop



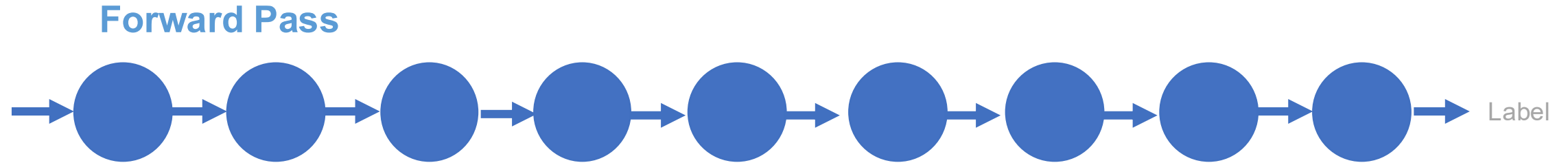


How to choose which layers to recompute?





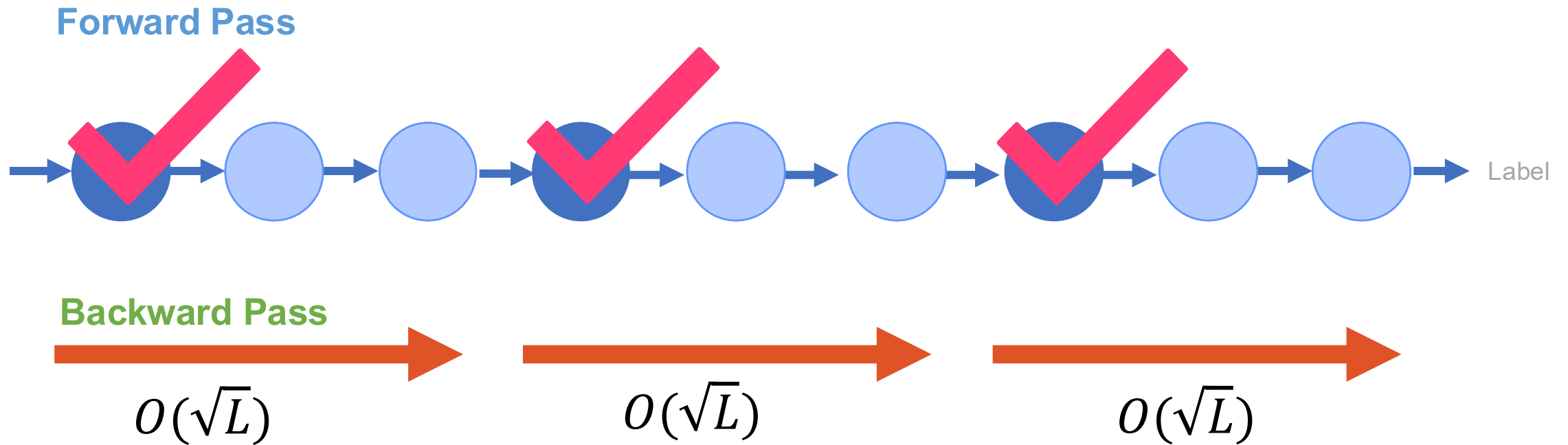
How to choose which layers to recompute?





9803087

Keep only every $\sqrt{L}$ for an L layer neural network.



Memory: $O(\sqrt{L})$ for L layers.
Compute: $O(L)$ for L layers.



Backpropagation at Scale

- **Activation checkpointing / recomputation:** Save only selected activations during the forward pass and **recompute** others during backward → trades compute for memory.
- **Gradient accumulation:**
 - **Issue:** Processing a large minibatch at once requires storing activations for the **entire minibatch** until they are used during backpropagation.
 - **Solution:** Split the minibatch into smaller **microbatches**, run forward + backward on each, and **accumulate the gradients**.
 - In PyTorch, `loss.backward()` **adds to** `.grad`; call `optimizer.zero_grad()` only after `optimizer.step()`.

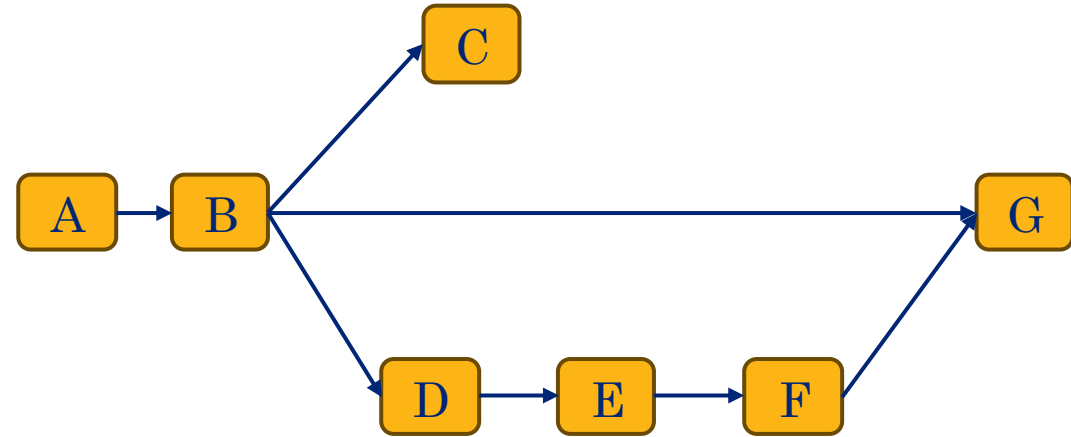


3940975

This demo may be helpful for the homework!

Demo

Autodiff



Reminder:

- Make sure to traverse in reverse topological ordering.
 - The **adjoint at B** can't be computed until the **adjoint at D is computed**.
- The **adjoint at C is zero** (no path from C to G).

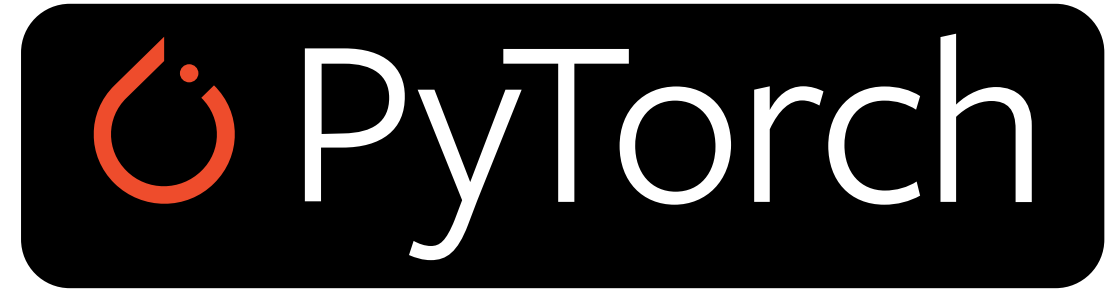
Homework 3 Overview

From Scalars to Tensors



Coding Part 1:

- Implement autograd for tensors
 - Lecture demo notebook that implemented autograd for scalars will be helpful!
- Implement optimizers
- Build and train a model using your implementations
 - This is literally (a simpler version of) PyTorch!



Coding Part 2:

- Introduction to PyTorch
- Building models
- Creating dataloaders
- Training and Evaluation

Optimizers

You'll be reading about optimizers.

- [Adam](#)
- [AdamW](#) (w/ weight decay)

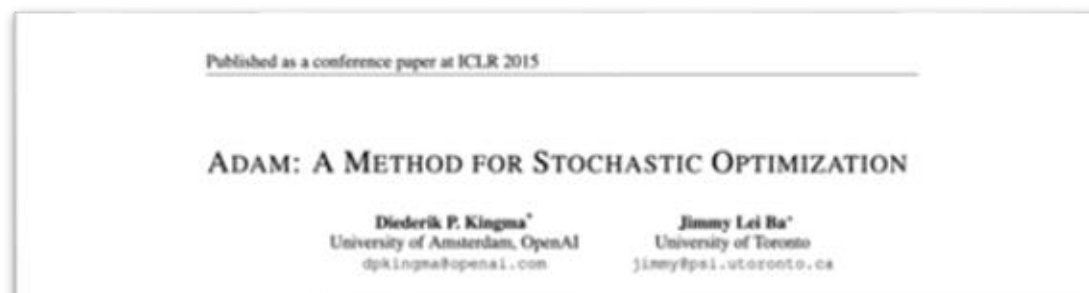
The papers are super dense, but some very nice blog posts:

- [Understanding Deep Learning Optimizers](#)
- [Adam vs. AdamW](#)

And discussion from CS 182 (Spring 2021):

- Section 3 of [Discussion 2](#)

The blogs and CS 182 Discussion suffice for the HW, but you're welcome to read the papers as well.



9803087

Homework Summary



Coding

- Pt1: Autograd for matrices + Optimizers
- Pt2: Introduction to PyTorch

Written

- Logistic Regression
- Backprop
- Optimizers
- Normalization (next lecture!)

Paper (Blog) Reading

- Blogs:
 - [Understanding Deep Learning Optimizers](#)
 - [Adam vs AdamW](#)
- CS 182 Sp21:
 - Section 3 of [Disc 2](#)
- (Optional) Papers:
 - [Adam](#)
 - [AdamW](#)



Important HW3 Dates

- Releases tomorrow (Friday, Oct. 9)
- You are responsible for 4 assignments on Pensive:
 - HW 3 Written (Friday, Oct. 23)
 - HW 3 Paper Write-Up (Friday, Oct. 23)
 - HW 3 Coding Part 1 (Friday, Oct. 23)
 - Submit the exported zip file; the autograder will automatically upload to HW 3 Coding Part 1 PDF
 - Watch the setup and submission walkthroughs on Askademia
 - HW 3 Coding Part 2 (Friday, Oct. 30)
 - The autograder will automatically upload to HW 3 Coding Part 2 PDF
- **Start early!**

Good luck and have fun :)

Additional Reading



- Bishop and Bishop, Deep Learning: Foundations and Concepts, Chapter 8
- [Automatic Differentiation in Machine Learning: A Survey \(2018\)](#)
- [A Gentle Introduction to torch.autograd](#)

Lecture 13

Backpropagation

Reading: *Chapter 8* in Bishop Deep Learning Textbook



Homework Hints:

Backward Rules for Matrix Operations

Each operation maps the adjoint of its output, $G = \partial E / \partial C$, to the adjoints of its inputs. Every adjoint has the same shape as its variable.

- **Matrix product** $C = AB$: $\frac{\partial E}{\partial A} = GB^T$, $\frac{\partial E}{\partial B} = A^T G$
- **Elementwise function** $C = h(A)$: $\frac{\partial E}{\partial A} = G \odot h'(A)$
- **Elementwise product** $C = A \odot B$: $\frac{\partial E}{\partial A} = G \odot B$, $\frac{\partial E}{\partial B} = G \odot A$
- **Broadcasting**, e.g., adding a bias row vector $\mathbf{b}$ to every row of A : $\partial E / \partial \mathbf{b}$ is G summed over the rows.
- **Sum** $c = \sum_{i,j} A_{ij}$: $\partial E / \partial A_{ij} = G$ for every entry.

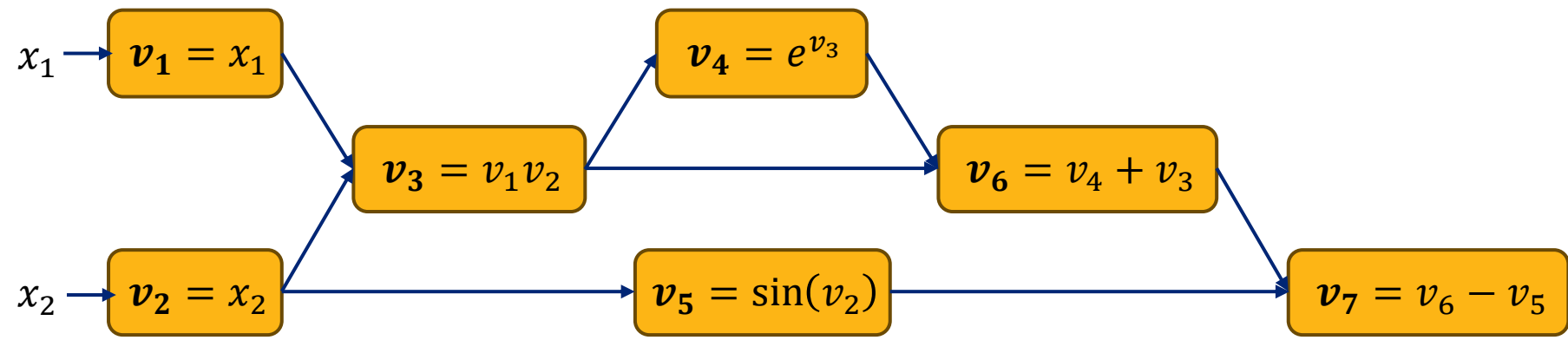
Applying these rules in reverse topological order is what `loss.backward()` does in PyTorch.

Bonus Material

Forward Differentiation



Chain Rule on The Computation Graph



If we want to compute the derivative $\frac{\partial}{\partial x_2} f$ using the chain rule:

$$\frac{\partial v_7}{\partial x_2} = \frac{\partial v_7}{\partial v_6} \left(\frac{\partial v_6}{\partial x_2} \right) + \frac{\partial v_7}{\partial v_5} \left(\frac{\partial v_5}{\partial x_2} \right) = \left(\frac{\partial v_6}{\partial x_2} \right) - \left(\frac{\partial v_5}{\partial x_2} \right)$$

$$\begin{aligned} \frac{\partial v_3}{\partial x_2} &= \frac{\partial v_3}{\partial v_1} \left(\frac{\partial v_1}{\partial x_2} \right) + \frac{\partial v_3}{\partial v_2} \left(\frac{\partial v_2}{\partial x_2} \right) \\ &= v_2 \left(\frac{\partial v_1}{\partial x_2} \right) + v_1 \left(\frac{\partial v_2}{\partial x_2} \right) \end{aligned}$$

We can generalize these expressions with:

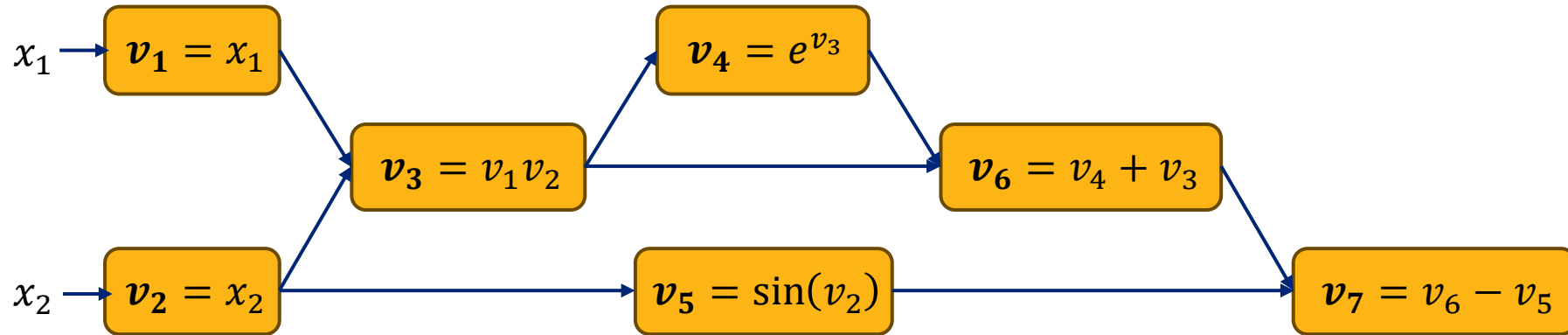
$$\frac{\partial v_i}{\partial x_2} = \sum_{j \in \mathbf{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \left(\frac{\partial v_j}{\partial x_2} \right)$$

where $\mathbf{Pa}[i]$ are the parents of i in the computation graph.





Chain Rule on The Computation Graph



If we define a new **tangent variable** representing the partial derivative $\partial_{x_k}^i = \frac{\partial v_i}{\partial x_k}$

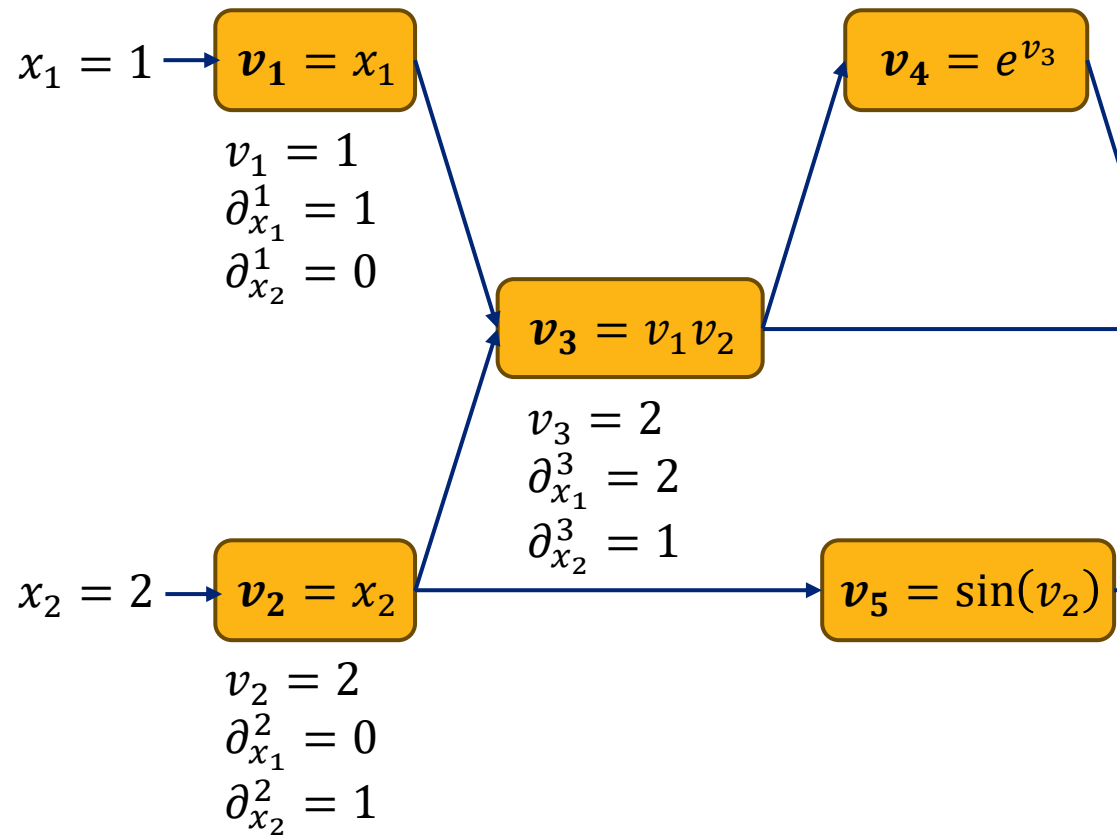
$$\partial_{x_k}^i = \frac{\partial v_i}{\partial x_k} = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \left(\frac{\partial v_j}{\partial x_k} \right) = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$

During the **forward computation** we can compute the value of v_i and $\partial_{x_k}^i$:

- Notice that v_i and $\partial_{x_k}^i$ only depend on their parents.
- Need to maintain separate $\partial_{x_k}^i$ for each variable k

Forward Autodiff Example

$$\partial_{x_k}^i = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$



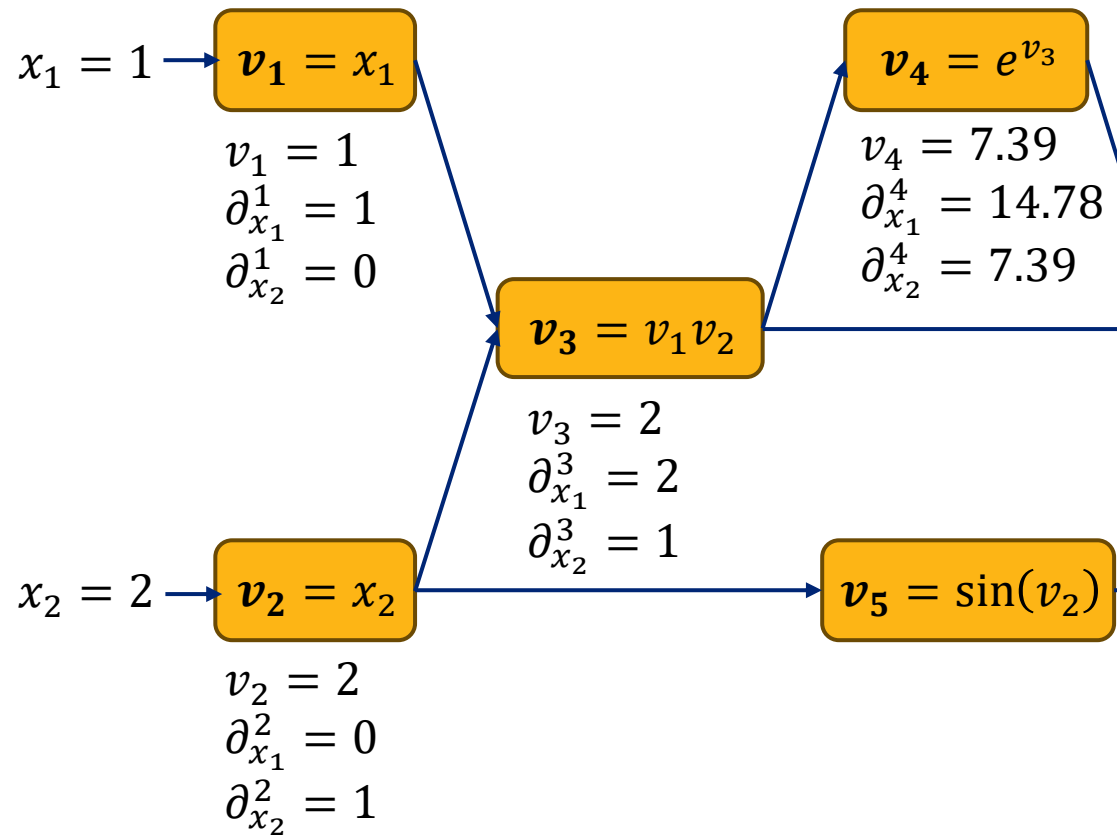
Derivation

$$\begin{aligned} \partial_{x_1}^3 &= \sum_{j \in \{1,2\}} \left(\frac{\partial v_3}{\partial v_j} \right) \partial_{x_1}^j = v_2 \partial_{x_1}^1 + v_1 \partial_{x_1}^2 \\ &= 2 * 1 + 1 * 0 \\ &= 2 \end{aligned}$$

$$\begin{aligned} \partial_{x_2}^3 &= \sum_{j \in \{1,2\}} \left(\frac{\partial v_3}{\partial v_j} \right) \partial_{x_2}^j = v_2 \partial_{x_2}^1 + v_1 \partial_{x_2}^2 \\ &= 2 * 0 + 1 * 1 \\ &= 1 \end{aligned}$$

Forward Autodiff Example

$$\partial_{x_k}^i = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$



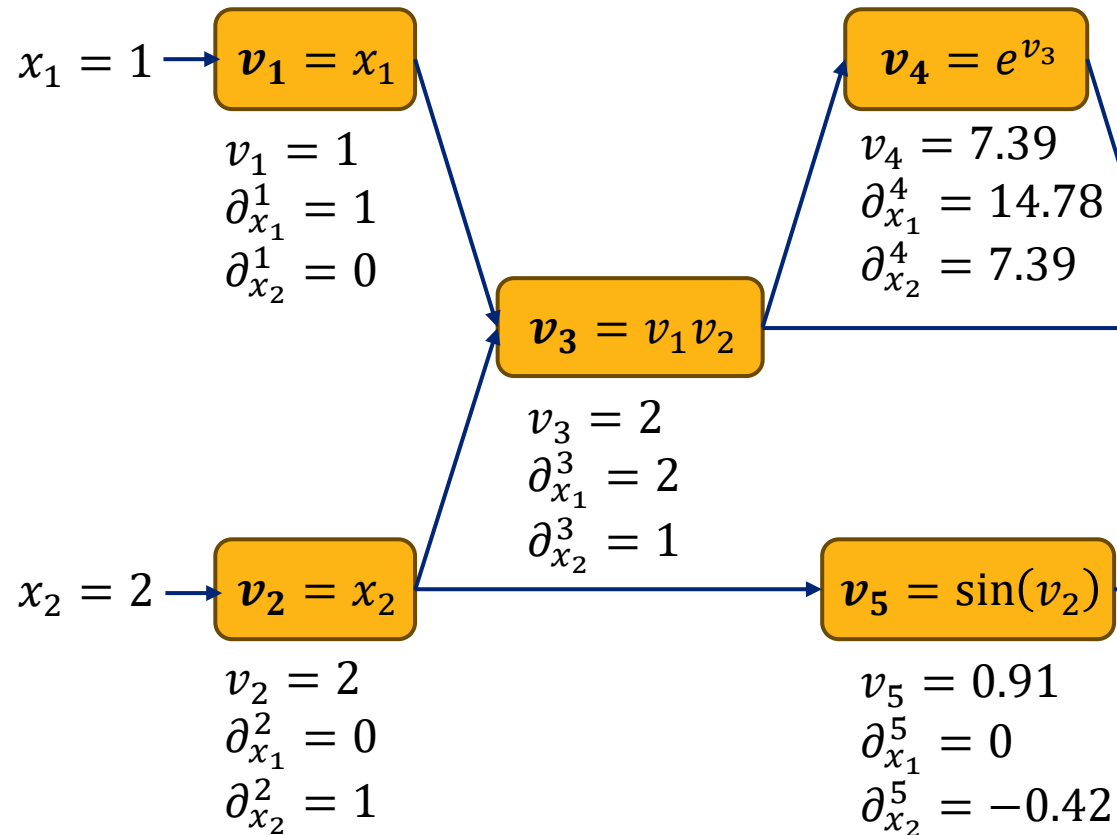
Derivation

$$\begin{aligned} \partial_{x_1}^4 &= \sum_{j \in \{3\}} \left(\frac{\partial v_4}{\partial v_j} \right) \partial_{x_1}^j = \left(\frac{\partial}{\partial v_3} e^{v_3} \right) \partial_{x_1}^3 \\ &= (e^{v_3}) \partial_{x_1}^3 \\ &= (v_4) 2 = 14.78 \end{aligned}$$

$$\begin{aligned} \partial_{x_2}^4 &= \sum_{j \in \{3\}} \left(\frac{\partial v_4}{\partial v_j} \right) \partial_{x_2}^j = \left(\frac{\partial}{\partial v_3} e^{v_3} \right) \partial_{x_2}^3 \\ &= (v_4) \partial_{x_2}^3 \\ &= (v_4) 1 = 7.39 \end{aligned}$$

Forward Autodiff Example

$$\partial_{x_k}^i = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$



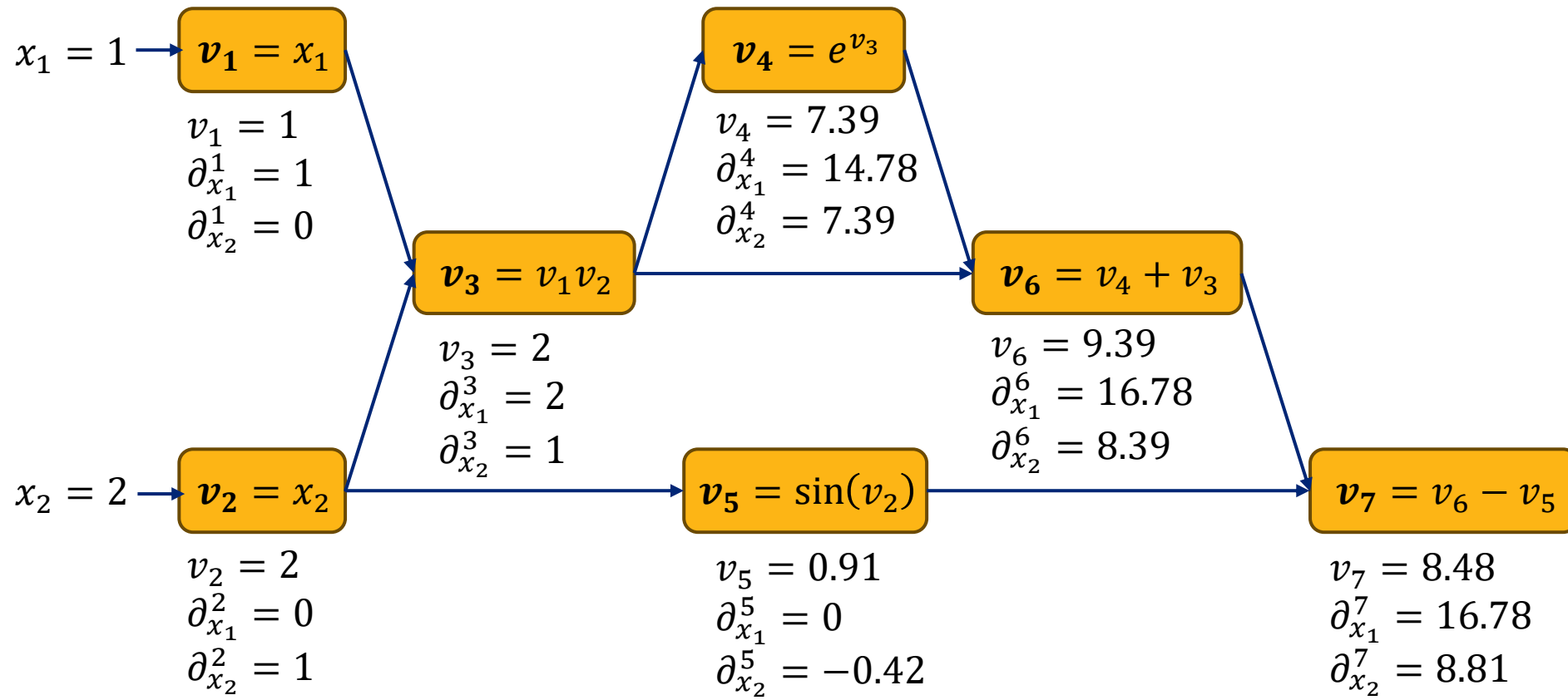
Derivation

$$\begin{aligned} \partial_{x_1}^5 &= \sum_{j \in \{2\}} \left(\frac{\partial v_5}{\partial v_j} \right) \partial_{x_1}^j = \left(\frac{\partial}{\partial v_2} \sin(v_2) \right) \partial_{x_1}^2 \\ &= (\cos(v_2)) \partial_{x_1}^2 \\ &= (-0.42) 0 = 0 \end{aligned}$$

$$\begin{aligned} \partial_{x_2}^5 &= \sum_{j \in \{2\}} \left(\frac{\partial v_5}{\partial v_j} \right) \partial_{x_2}^j = \left(\frac{\partial}{\partial v_2} \sin(v_2) \right) \partial_{x_2}^2 \\ &= (\cos(v_2)) \partial_{x_2}^2 \\ &= (-0.42) 1 = -0.42 \end{aligned}$$

Forward Autodiff Example

$$\partial_{x_k}^i = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$

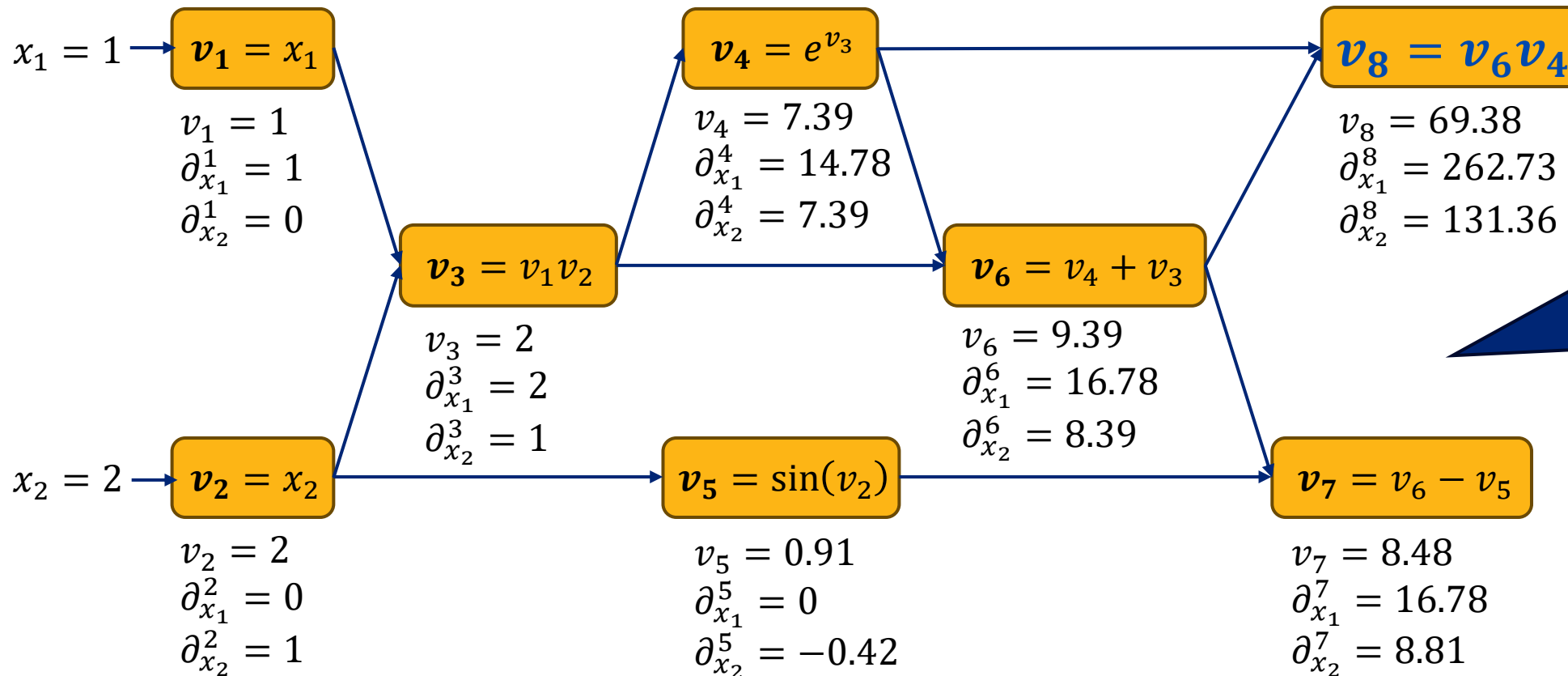


Support multiple outputs

$$v_7 = f_1(x_1, x_2) = x_1 x_2 + e^{(x_1 x_2)} - \sin(x_2)$$

$$v_8 = f_2(x_1, x_2) = (x_1 x_2 + e^{(x_1 x_2)}) e^{(x_1 x_2)}$$

Additional function output
can **reuse computation**.



Essentially,
we computed
the gradient
at **every**
step!



37



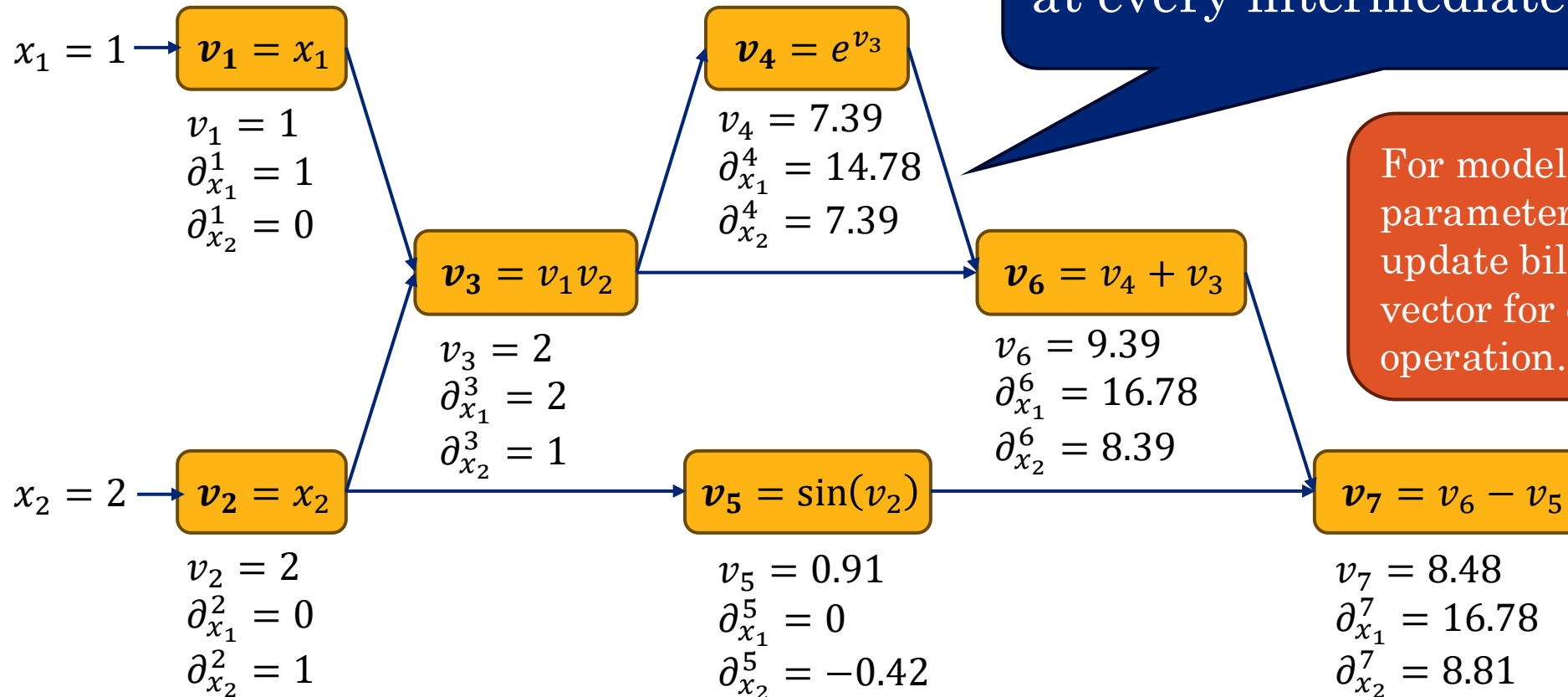
Expensive for Lots of Variables



9803087



Need to **maintain full gradient** at every intermediate step.



For models with billions of parameters, need to store and update billion-dimension vector for every arithmetic operation.



Adjoint vs Tangent Variables

We have now defined **tangent** and **adjoint** variables for **forward** and **backward** autodiff respectively:

Tangent Variables

$$\partial_{x_k}^i = \sum_{j \in \text{Pa}[i]} \left(\frac{\partial v_i}{\partial v_j} \right) \partial_{x_k}^j$$

Adjoint Variables

$$a_i^f = \sum_{j \in \text{Ch}[i]} a_j^f \left(\frac{\partial v_j}{\partial v_i} \right)$$

Notice how they build from one side of the graph to the other:

