Deep Learning

Automatic Differentiation

Logistics

- Homework 1 due today
- Homework 2 out today, due in 2 weeks
- Today: Backpropogation/autograd.
  - Should be useful for homework 2!

What's on the Menu today?

- Recap
- Recap of freshman calculus
- Dive into chain rule
- Computational Graph & Reverse-mode differentiation (i.e. chain rule on steroids)

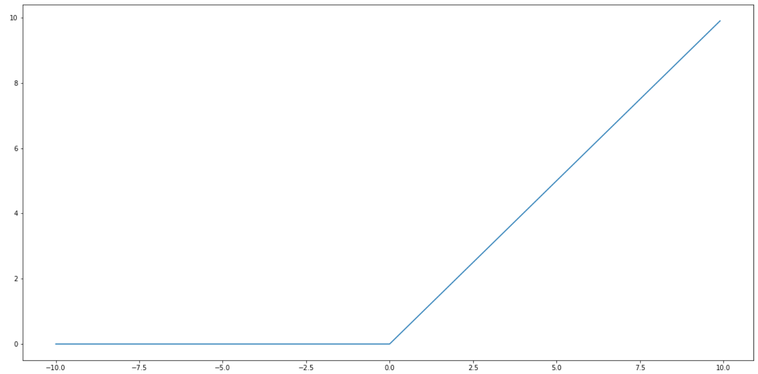
- DIY: Useful tricks

Recap

Last time: ReLU activation

- Modern deep networks usually use some variant of the *rectified linear unit* activation:

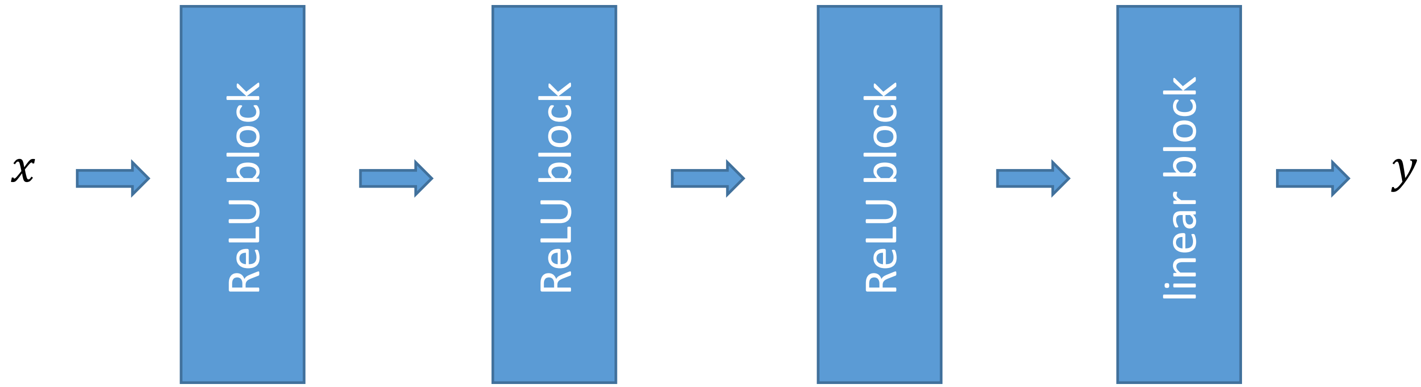
$$\sigma(x) = \max(x, 0)$$



- This still shares some inspiration with sigmoid: it is “off” when the input is negative.
- Since it does not saturate on the positive end, it has a much larger range of “learning”.

Last Time: ReLU activation MLP

- A more modern MLP looks like:



- ReLU networks are piecewise linear

Last Time: Cross-Entropy Loss

- Input 1: a *distribution* over the C possible output classes

$$p = (p_1, \dots, p_C), \qquad \sum_{i=1}^{C} p_i = 1$$

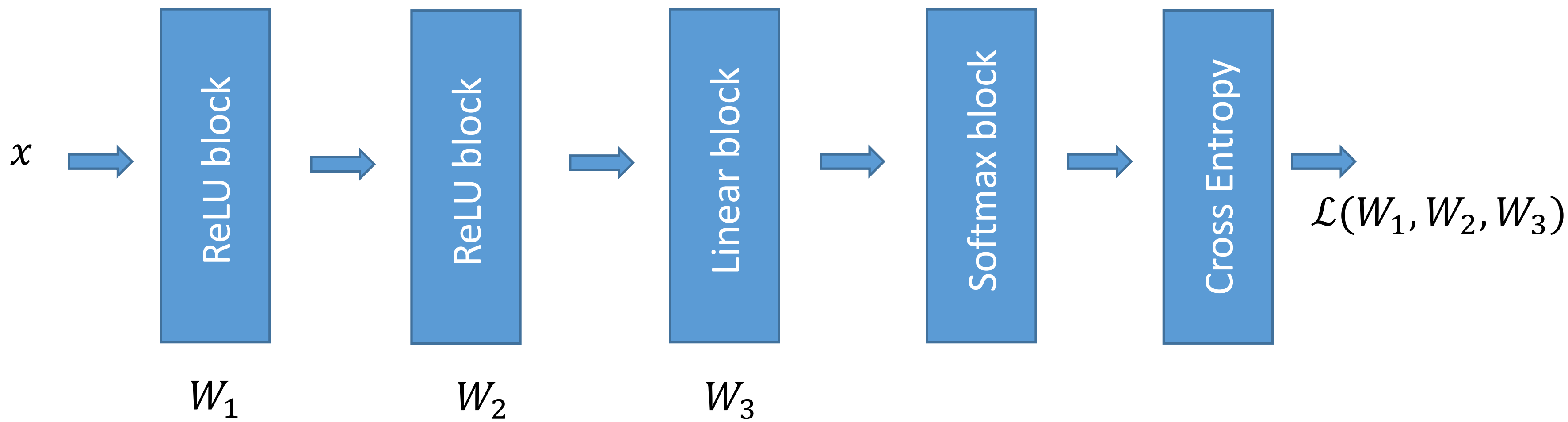
- Input 2: the true class value as a 1-hot vector:

$$y = (0, \dots, 1, \dots, 0)$$

- The cross-entropy loss is:

$$\ell(p, y) = \sum_i -1[y = i]\log(p_i)$$
$$= -\log(p_{correct\ class})$$

Today: Automatic Differentiation



- We need to compute gradients $\nabla_{W_i}\mathcal{L}(W_1, W_2, W_3)$
- You could code this up by hand for this network, but it would be a pain.
- Every time you change the network, you'd have to change the code!

Right matrix-multiplies

- In math class, you usually multiply matrices on the left:

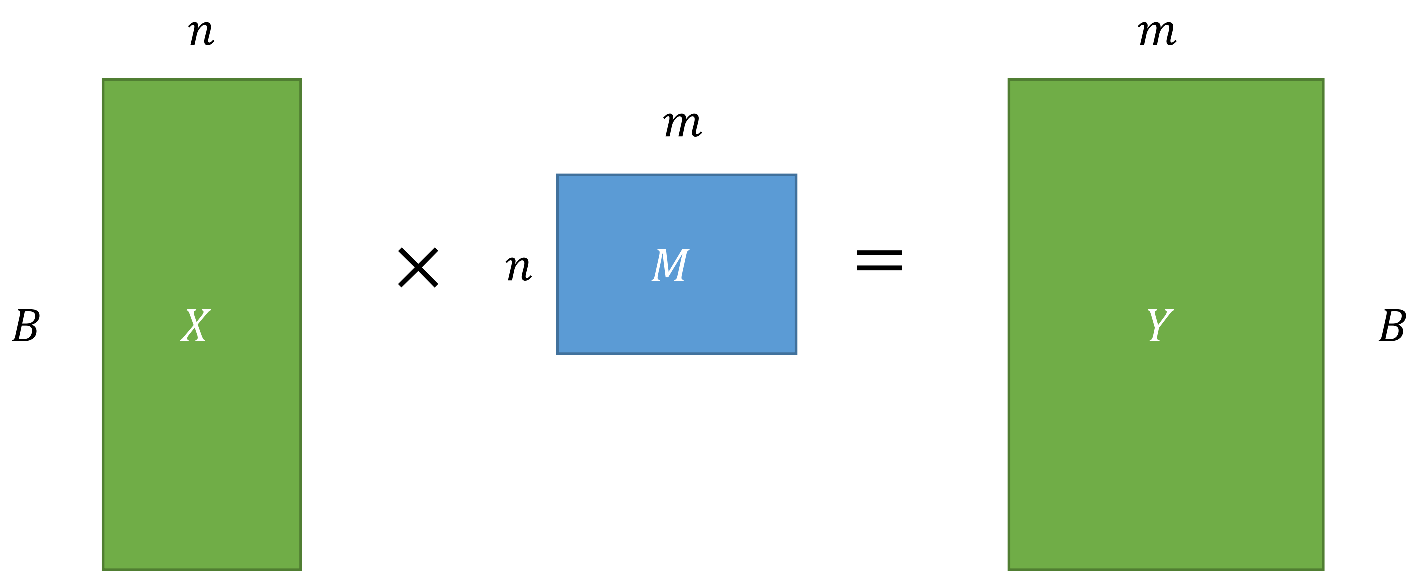
$$Mv$$

- In this lecture, we will multiply matrices on the right:

$$vM$$

- We do this to easily accommodate batch dimensions.

Right matrix-multiplies



A matrix $M \in \mathbb{R}^{n \times m}$ takes inputs in $\mathbb{R}^n$ and produces outputs in $\mathbb{R}^m$.

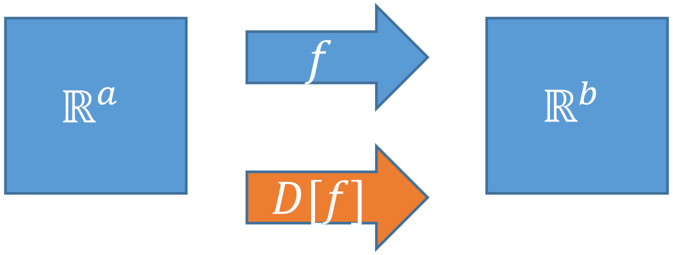
Recap of freshman calculus

What is a derivative, really?

- Let $f: \mathbb{R}^a \to \mathbb{R}^b$. Then the derivative is

$$D[f](x) \in \mathbb{R}^{a \times b}$$

- The derivative at x is a linear map (i.e. a matrix), whose inputs and outputs are the same dimension as f.



Special case: Gradients

- $f(x_1, x_2, x_3) = x_1 x_2 + x_3$
- $\nabla f(x_1, x_2, x_3) = (x_2, x_1, 1)$
- Technically, we should actually write:

$$\nabla f(x_1, x_2, x_3) = \begin{bmatrix} x_2 \\ x_1 \\ 1 \end{bmatrix}$$

- This is a 3×1 matrix, because f takes 3 dimensional inputs and has 1 dimensional outputs.

Derivative Formula

- The derivative has the formula:
$$D[f](x)_{ij} = \frac{df_j}{dx_i}(x)$$
- f_j is the j^{th} coordinate of f.
- What is the derivative of $f(x) = (x_1^2, x_1 x_2 x_3)$, which takes $\mathbb{R}^3$ to $\mathbb{R}^2$?
- Answer:
$$D[f](x) = \begin{bmatrix} 2x_1 & x_2 x_3 \\ 0 & x_1 x_3 \\ 0 & x_1 x_2 \end{bmatrix}$$

Another Example

- $f(x_1, x_2) = (x_2^2,\ x_1 + x_2,\ x_1 x_2)$
- $f_1(x_1, x_2) = x_2^2$
- $f_2(x_1, x_2) = x_1 + x_2$
- $f_3(x_1, x_2) = x_1 x_2$
- The derivative is 2×3:

$$D[f](x) = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

- Answer:

$$D[f](x) = \begin{bmatrix} 0 & 1 & x_2 \\ 2x_2 & 1 & x_1 \end{bmatrix}$$

Chain Rule

The Chain Rule

- In calculus we all learned:

$$(g \circ f)'(x) = f'(x)g'\big(f(x)\big)$$

- In higher dimensions, we have a matrix statement:

$$D[g \circ f](x) = D[f](x)D[g]\big(f(x)\big)$$

- Let's check the dimensions match:

- $D[f] \in \mathbb{R}^{a\times b}$ and $D[g] \in \mathbb{R}^{b\times c}$. $D[g \circ f] \in \mathbb{R}^{a\times c}$.

Example

- $f(x) = x^4$
- $g(y) = \sqrt{y}$

- $D[f](x) = 4x^3$
- $D[g](y) = \frac{1}{2\sqrt{y}}$
- $D[g \circ f](x) = D[f](x)D[g]\big(f(x)\big) = 2x$

Multiple Inputs

- Suppose f has multiple, multi-dimensional inputs:
$$f(X, Y): \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^c$$
- The partial derivatives of f with respect to the two arguments are defined analogously:
$$D_X[f](x, y) \in \mathbb{R}^{a \times c}$$
- The equation is:
$$D_X[f](x, y)_{ij} = \frac{\partial f_j}{\partial X_i}(x, y)$$

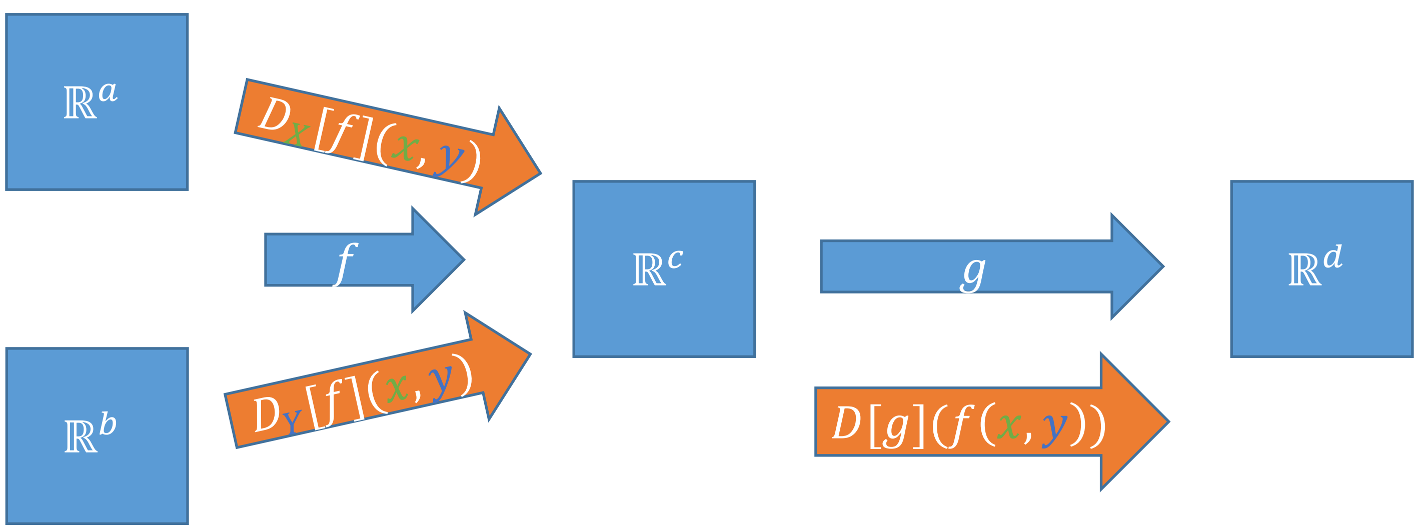
Chain rule with multiple inputs

- The chain rule works the same.
- Let $g: \mathbb{R}^c \to \mathbb{R}^d$ and

$$f(X, Y): \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^c$$

- The partial derivative is:

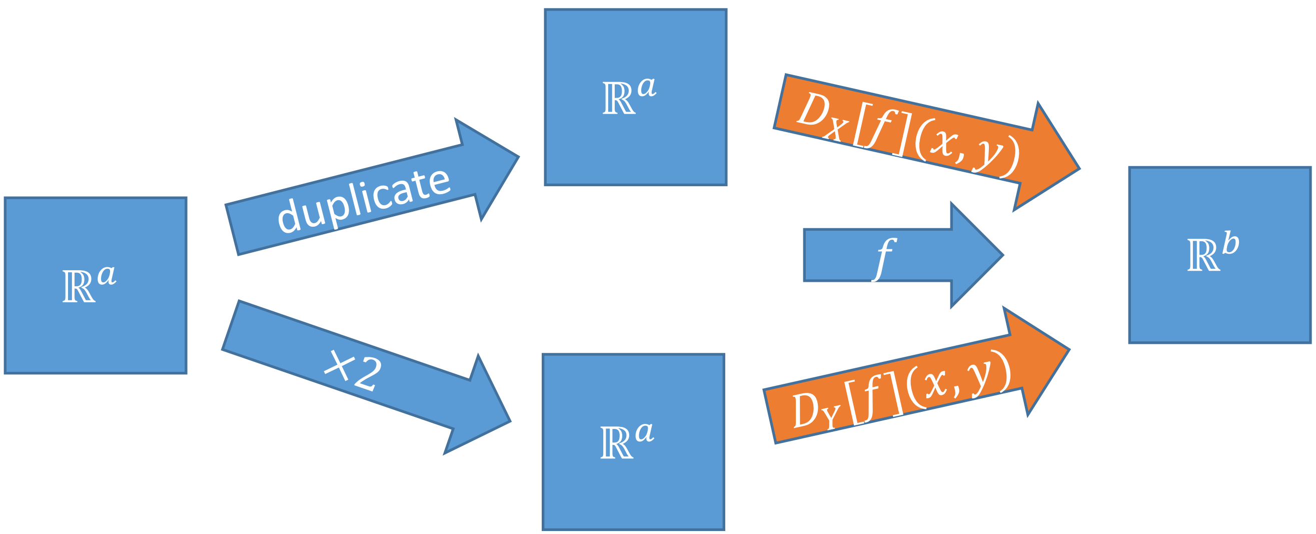
$$D_X[g \circ f](x, y) = D_X[f](x, y) D[g]\big(f(x, y)\big)$$



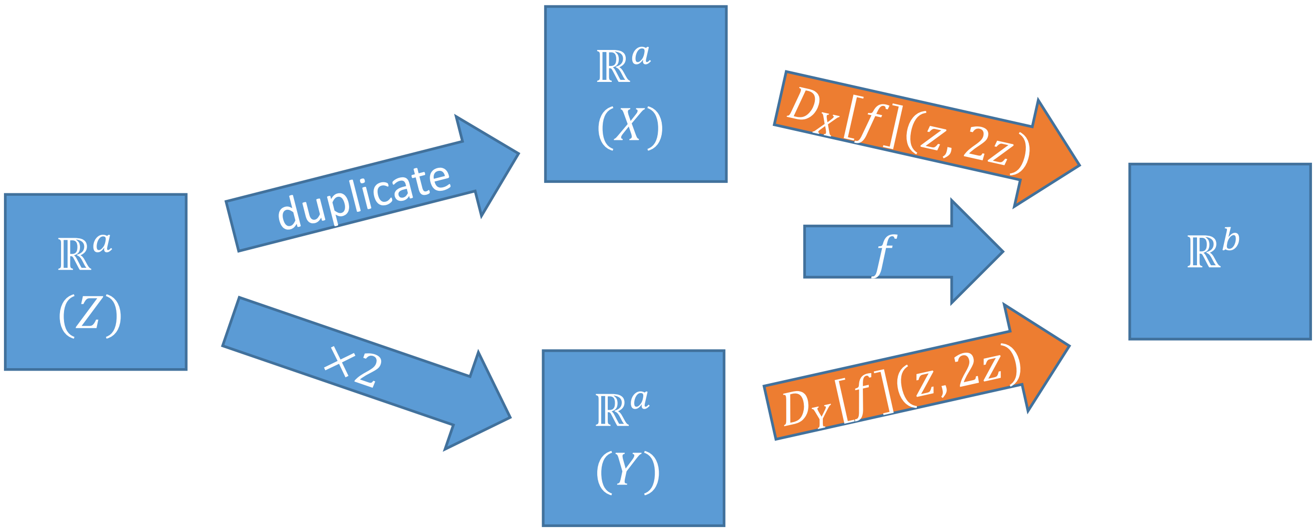
Re-Using Inputs

$$f(X, Y): \mathbb{R}^a \times \mathbb{R}^a \rightarrow \mathbb{R}^b$$
$$g(X) = f(X, 2X)$$

- What is $D[g]$?



Re-Using Inputs



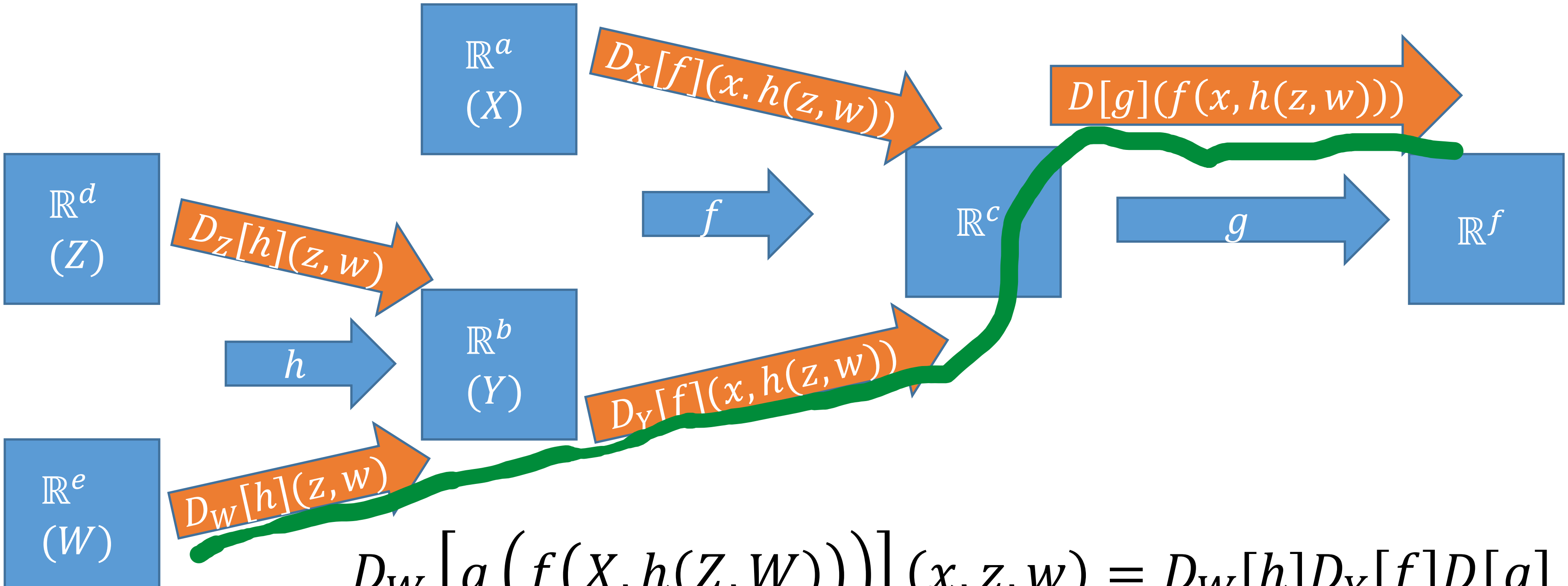
$$D_Z[f(Z, 2Z)] = D_X[f](z, z) + 2D_Y[f](z, 2z)$$

- When the inputs split, we need to sum the individual derivatives.

More complicated functions

$$g\colon \mathbb{R}^c \to \mathbb{R}^f$$
$$f(X,Y)\colon \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^c$$
$$h(Z,W)\colon \mathbb{R}^d \times \mathbb{R}^e \to \mathbb{R}^b$$

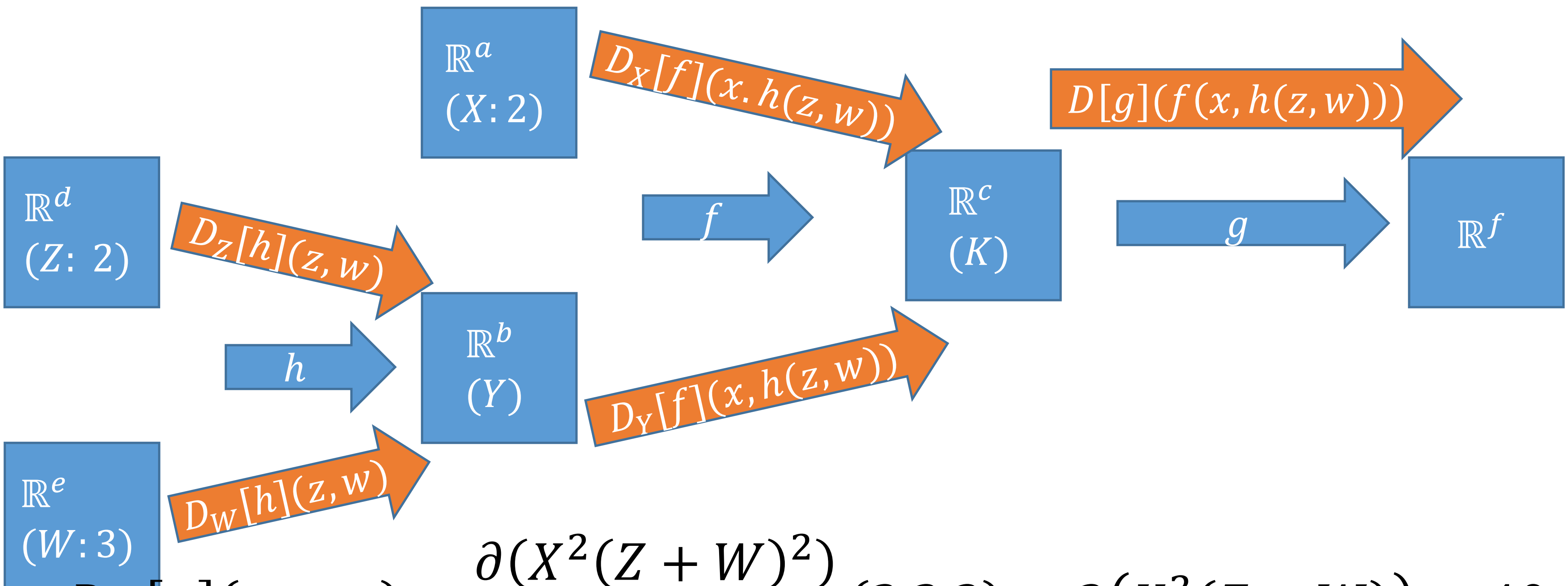
The function : $g\Big(f\big(X, h(Z,W)\big)\Big)$



$$D_W\Big[g\Big(f\big(X,h(Z,W)\big)\Big)\Big](x,z,w) = D_W[h]D_Y[f]D[g]$$

Example

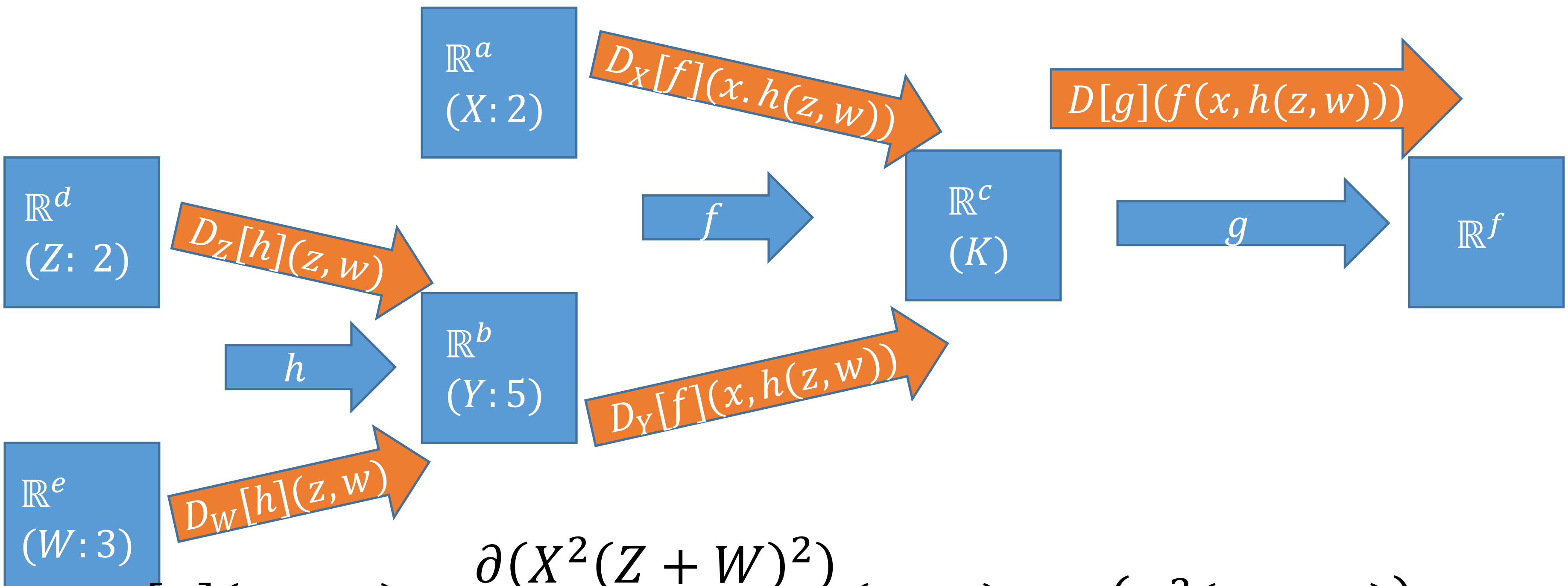
$$g(K) = K^2$$
$$f(X, Y) = XY$$
$$h(Z, W) = Z + W$$



$$D_W[g](x, z, w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(2,2,3) = 2\big(X^2(Z+W)\big) = 40$$

Example

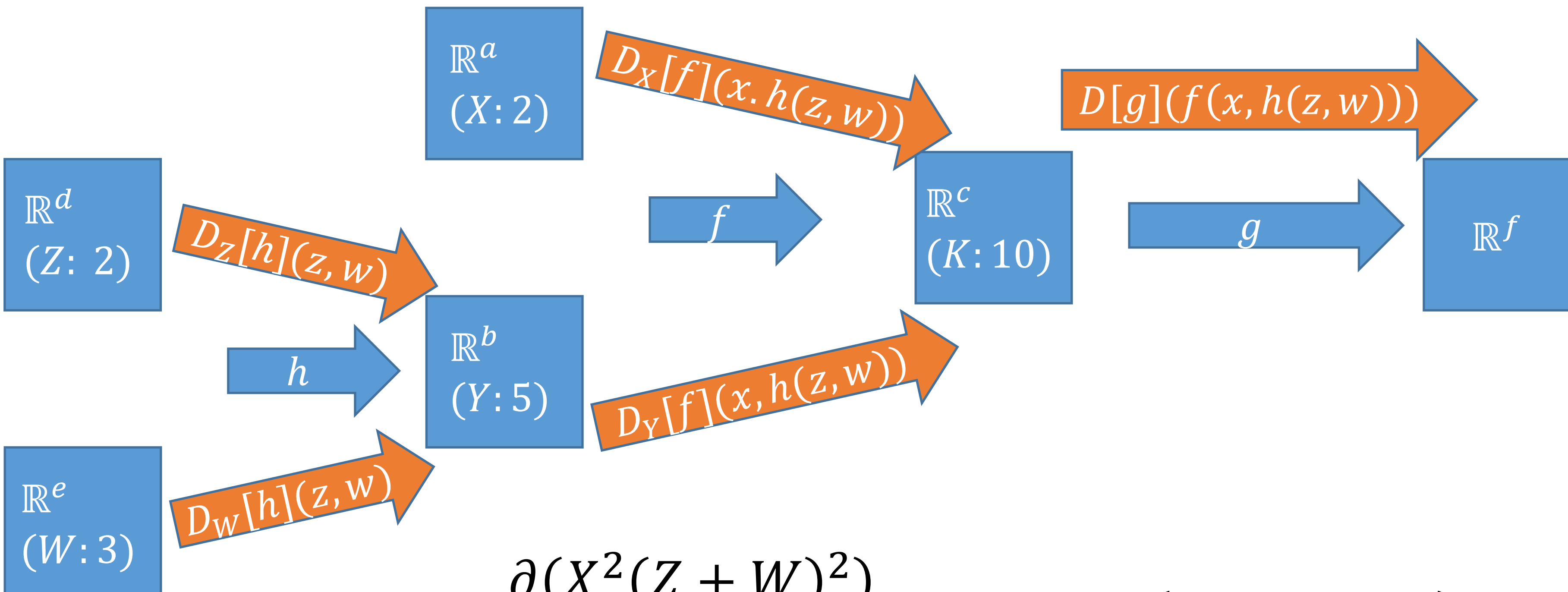
$$g(K) = K^2$$
$$f(X,Y) = XY$$
$$h(Z,W) = Z + W$$



$$D_W[g](x,z,w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(2,2,3) = 2\big(X^2(Z+W)\big) = 40$$

Example

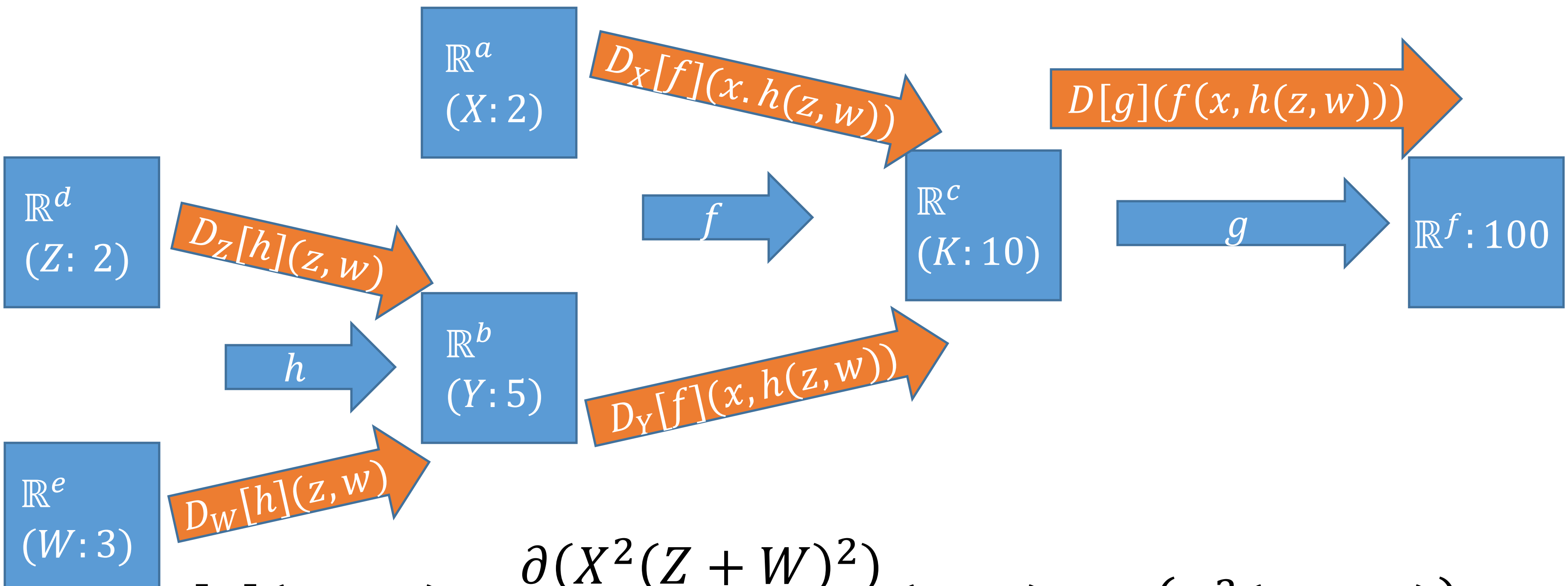
$$g(K) = K^2$$
$$f(X, Y) = XY$$
$$h(Z, W) = Z + W$$



$$D_W[g](x, z, w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(2,2,3) = 2\big(X^2(Z+W)\big) = 40$$

Example

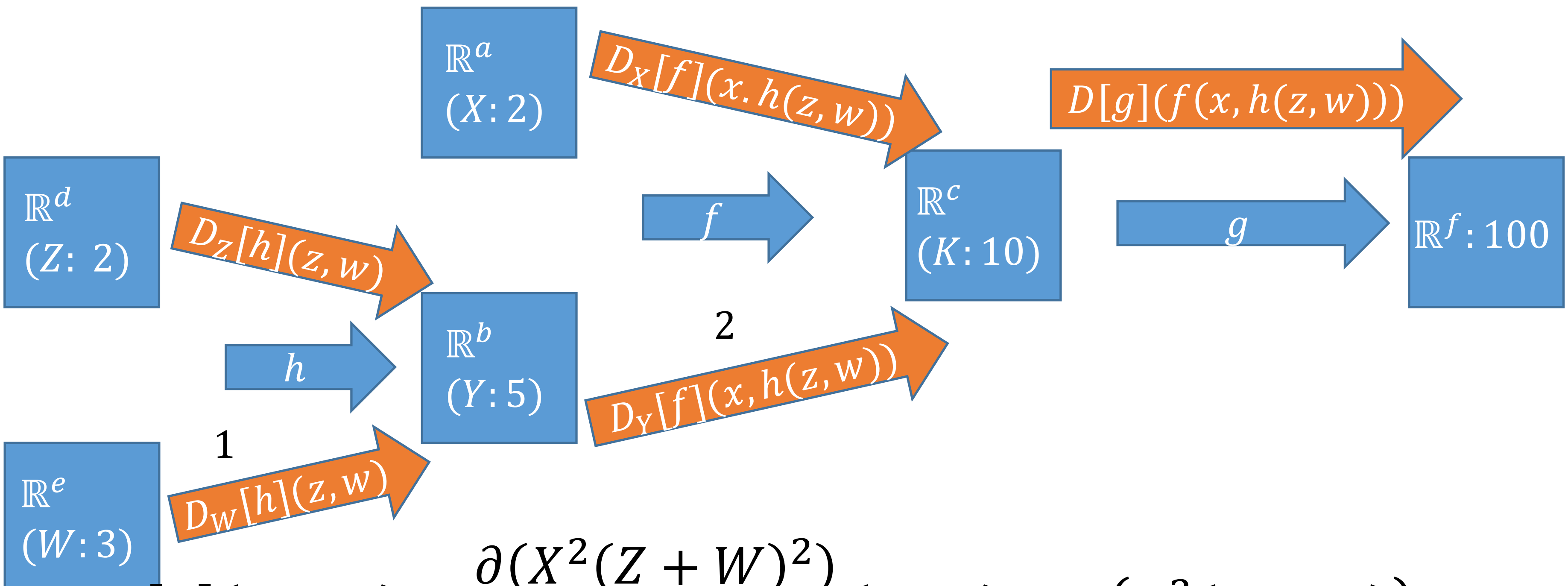
$$g(K) = K^2$$
$$f(X,Y) = XY$$
$$h(Z,W) = Z + W$$



$$D_W[g](x,z,w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(1,2,3) = 2\big(X^2(Z+W)\big) = 40$$

Example

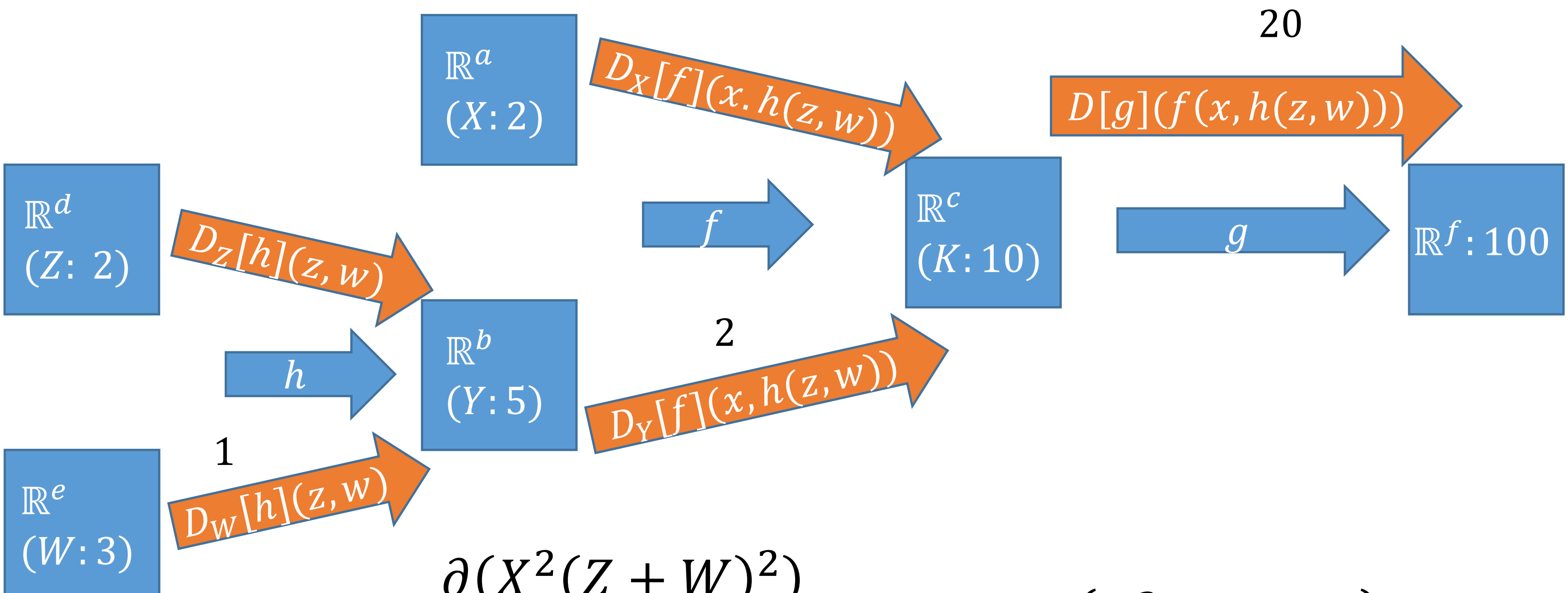
$$g(K) = K^2$$
$$f(X,Y) = XY$$
$$h(Z,W) = Z + W$$



$$D_W[g](x,z,w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(2,2,3) = 2\big(X^2(Z+W)\big) = 40$$

Example

$$g(K) = K^2$$
$$f(X,Y): XY$$
$$h(Z,W): Z + W$$



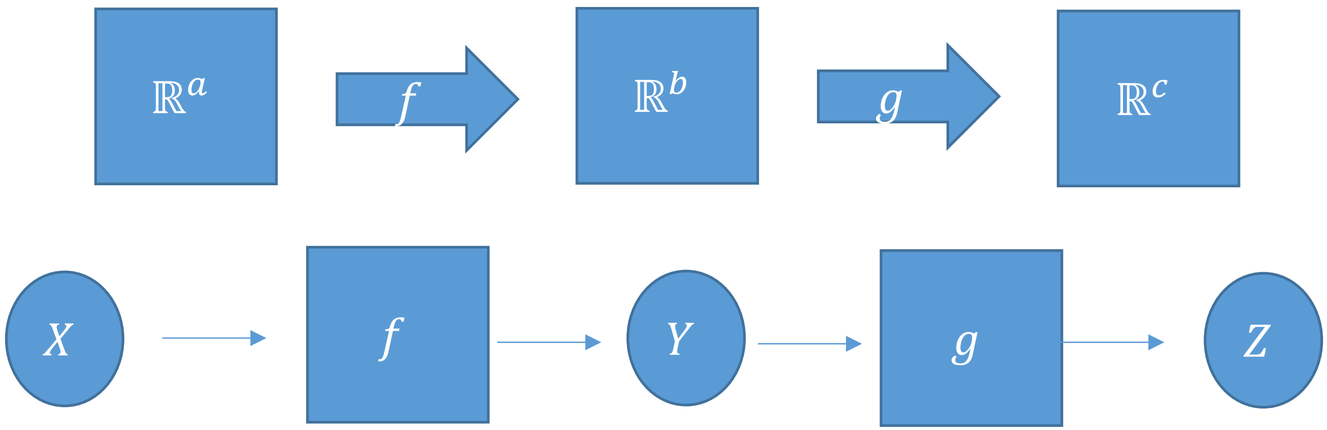
$$D_W[g](x,z,w) = \frac{\partial(X^2(Z+W)^2)}{\partial W}(2,2,3) = 2\big(X^2(Z+W)\big) = 40$$

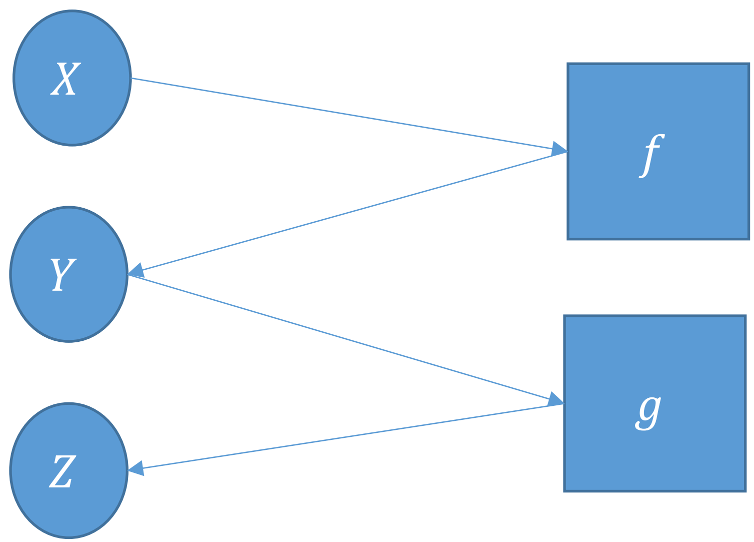
Computational Graph

Computational Graph

- Directed Acyclic graph with two kinds of nodes.
- Two kinds of nodes: "operation" nodes and "variable" nodes.
- Each "operation" node represents a function
- Each "variable" node represents either function inputs or outputs (or both)
- The graph is bipartite: variable nodes are only connected to operation nodes, and operation nodes are only connected to variable nodes.
- All source and sink nodes are variable nodes.

Computational Graph

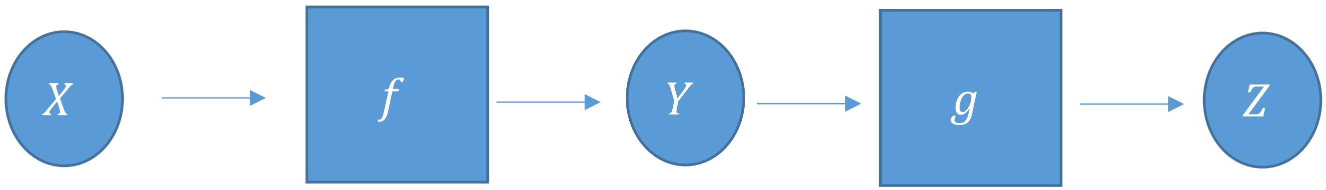




Variable nodes

Operation nodes

Data held by nodes

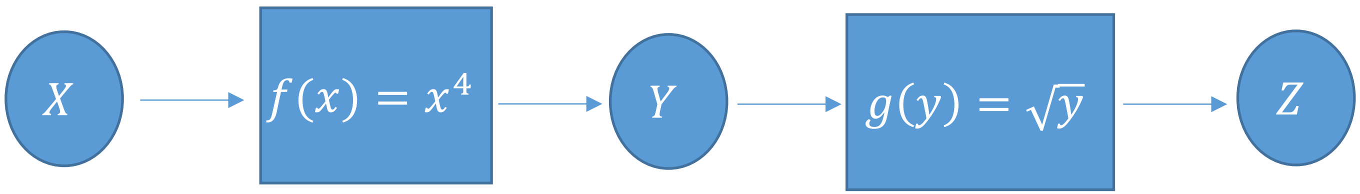


- Variable nodes will hold real numbers/vectors/matrices representing the "value" of the variable they represent.
- Operation nodes may hold various arbitrary state needed to implement either the forward or backward passes.

Backprop: “reverse-mode differentiation”

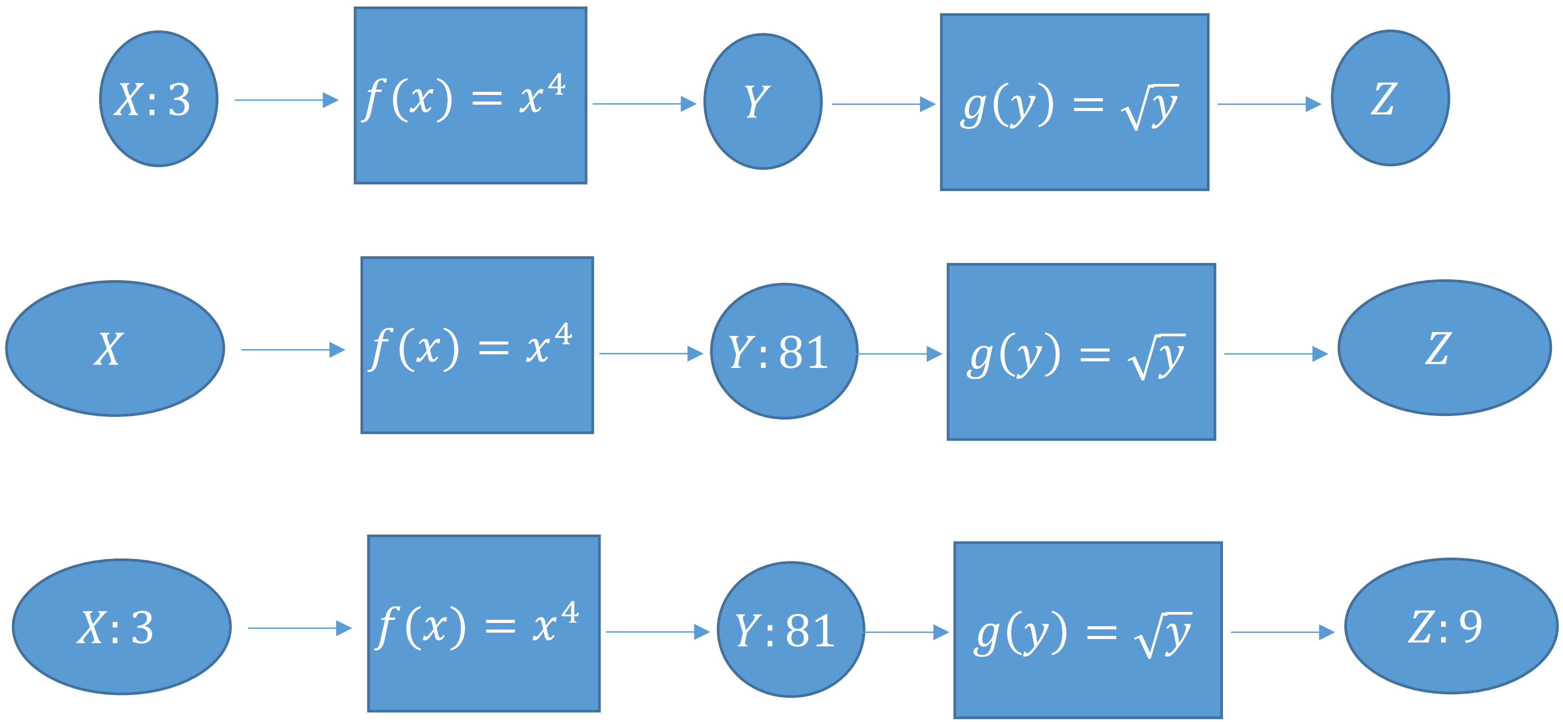
- Backpropagation is an algorithm for automatically computing the gradient of a function specified by a computation graph.
- It has two phases, the “forward pass” and the “backward pass”

Forward pass

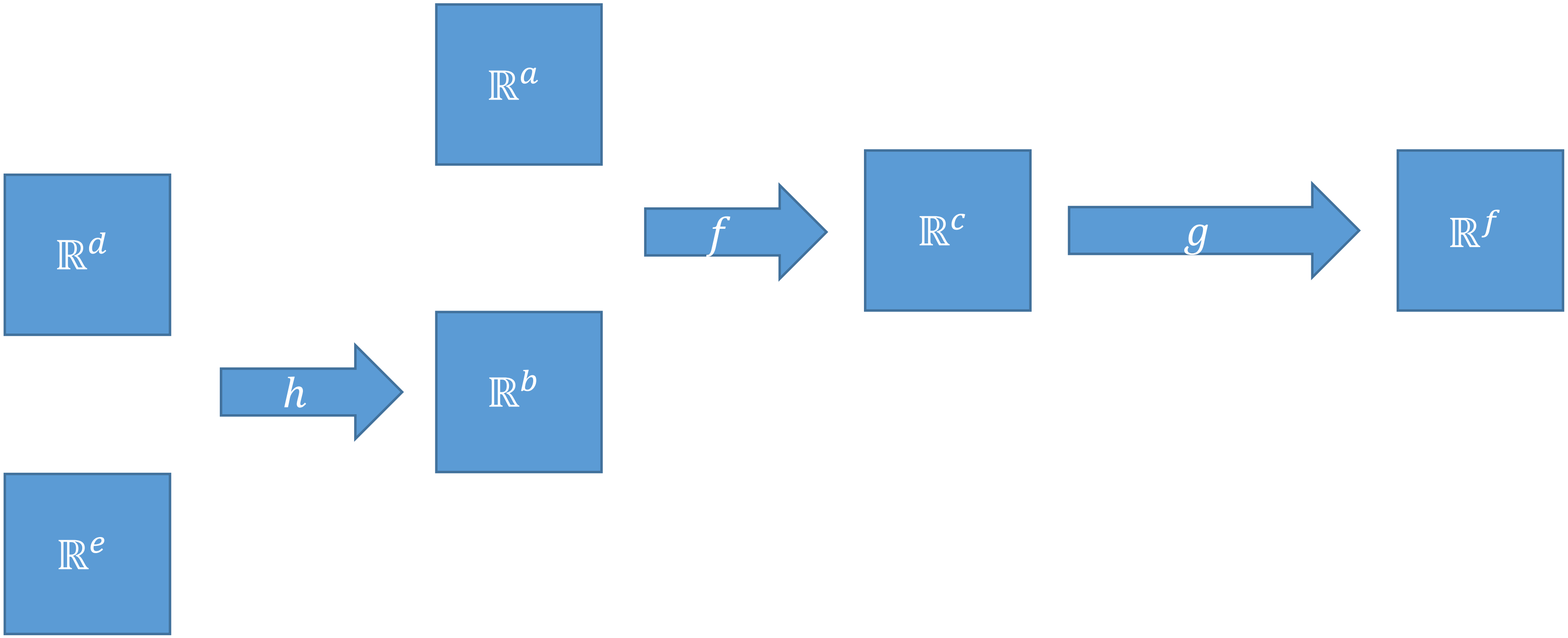


- **Step 1**: Start with seeding the source variable node with some value, say $X = 3$
- **Step 2**: Look at all the operation nodes for which all inputs have values assigned.
- **Step 3**: Compute these operations, and assign the output to the output variable nodes. Potentially save some state in the operation node.
- **Step 4**: Repeat

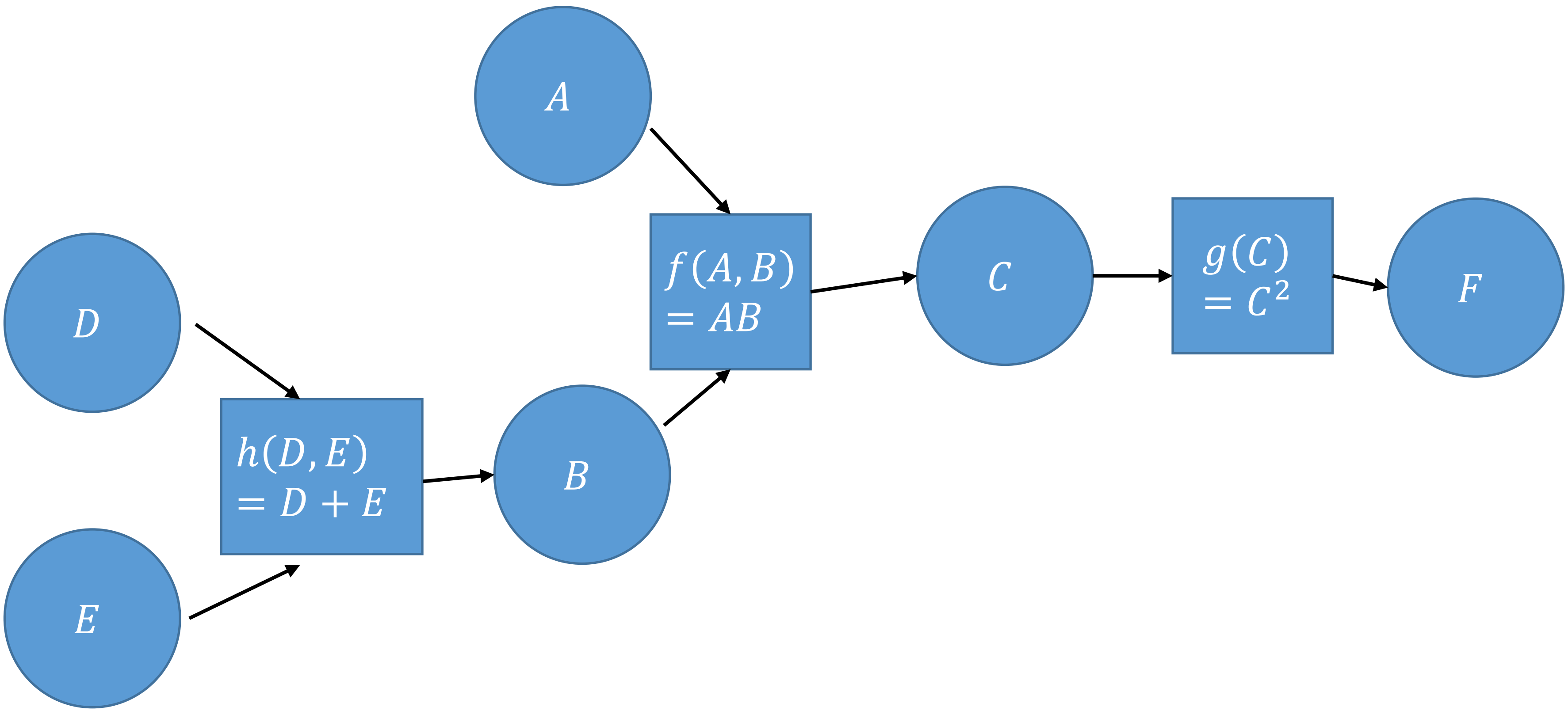
Forward Pass



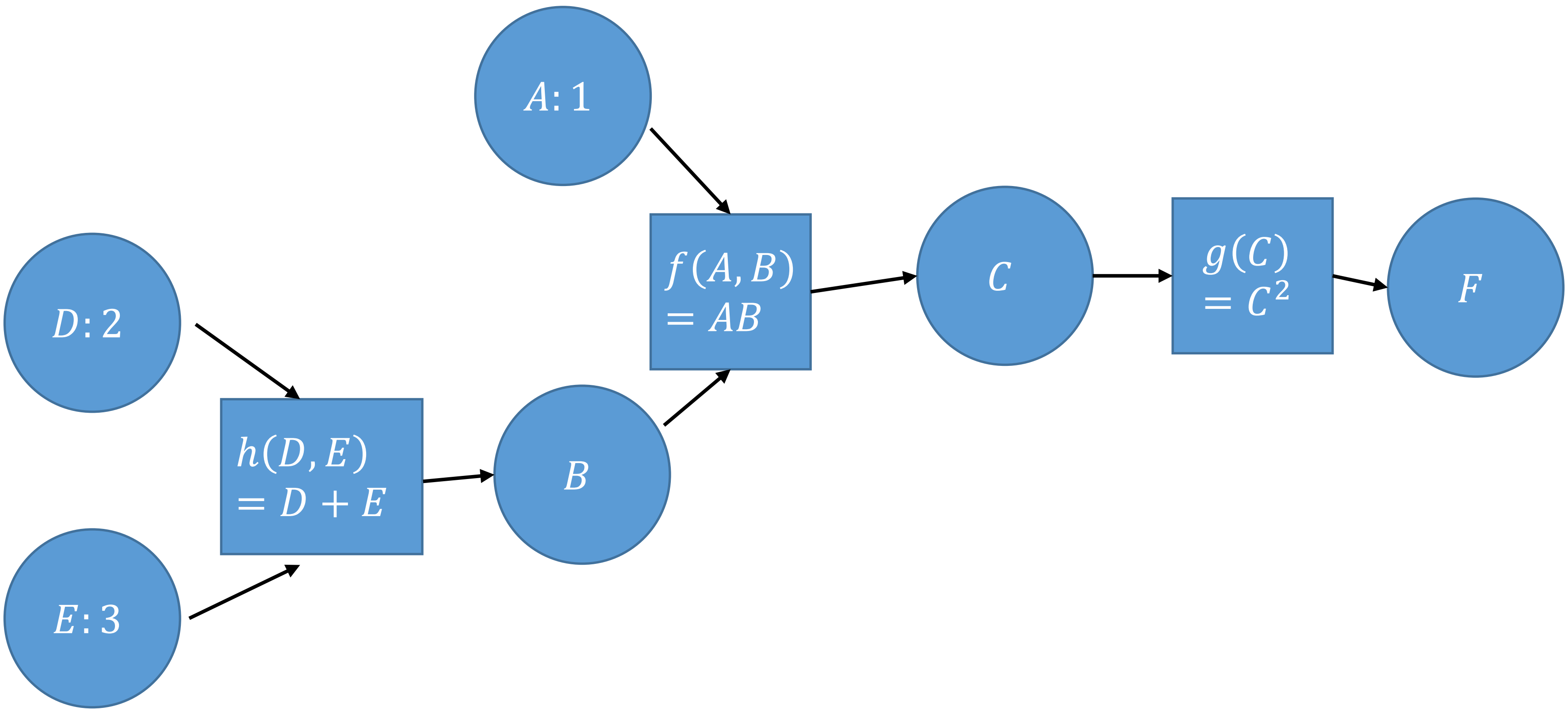
More Complicated Forward Pass



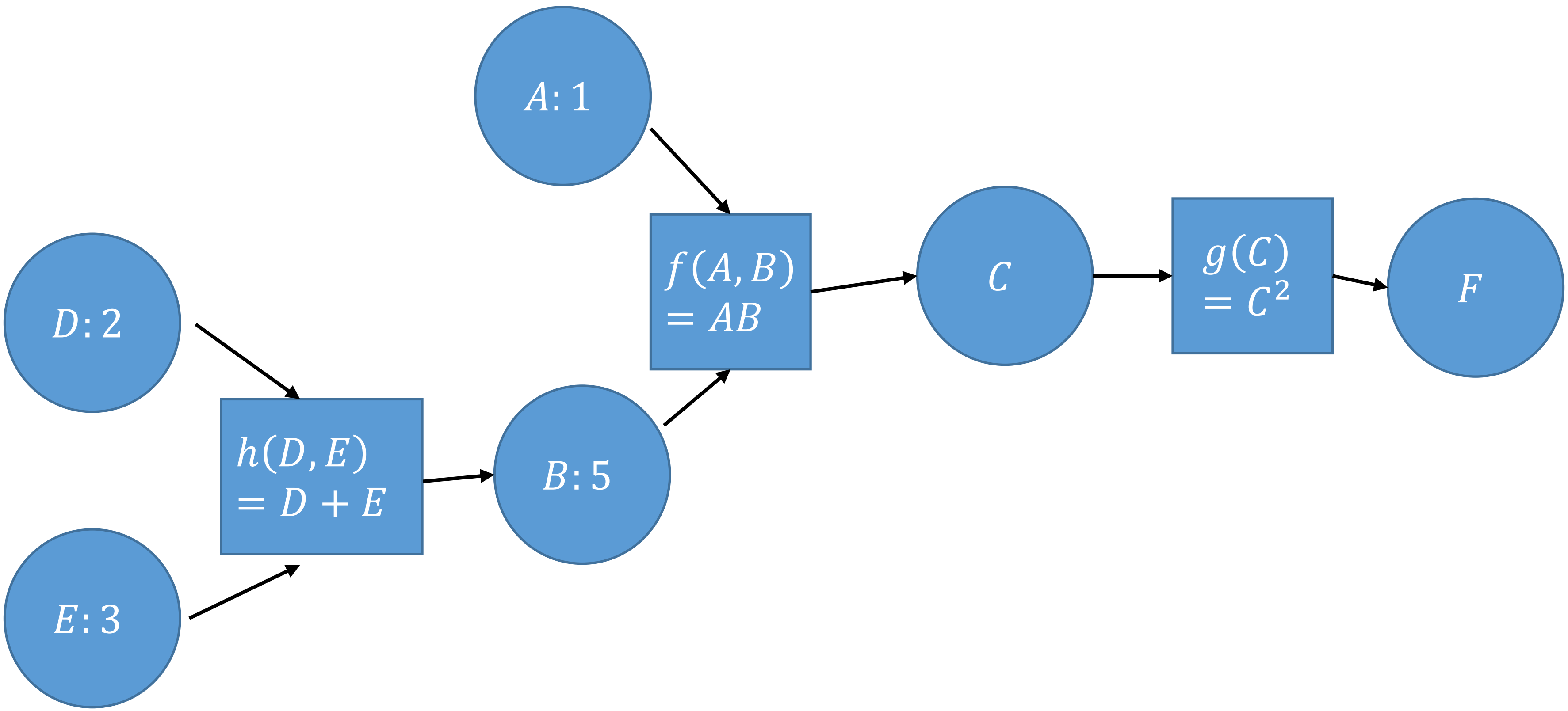
More Complicated Forward Pass



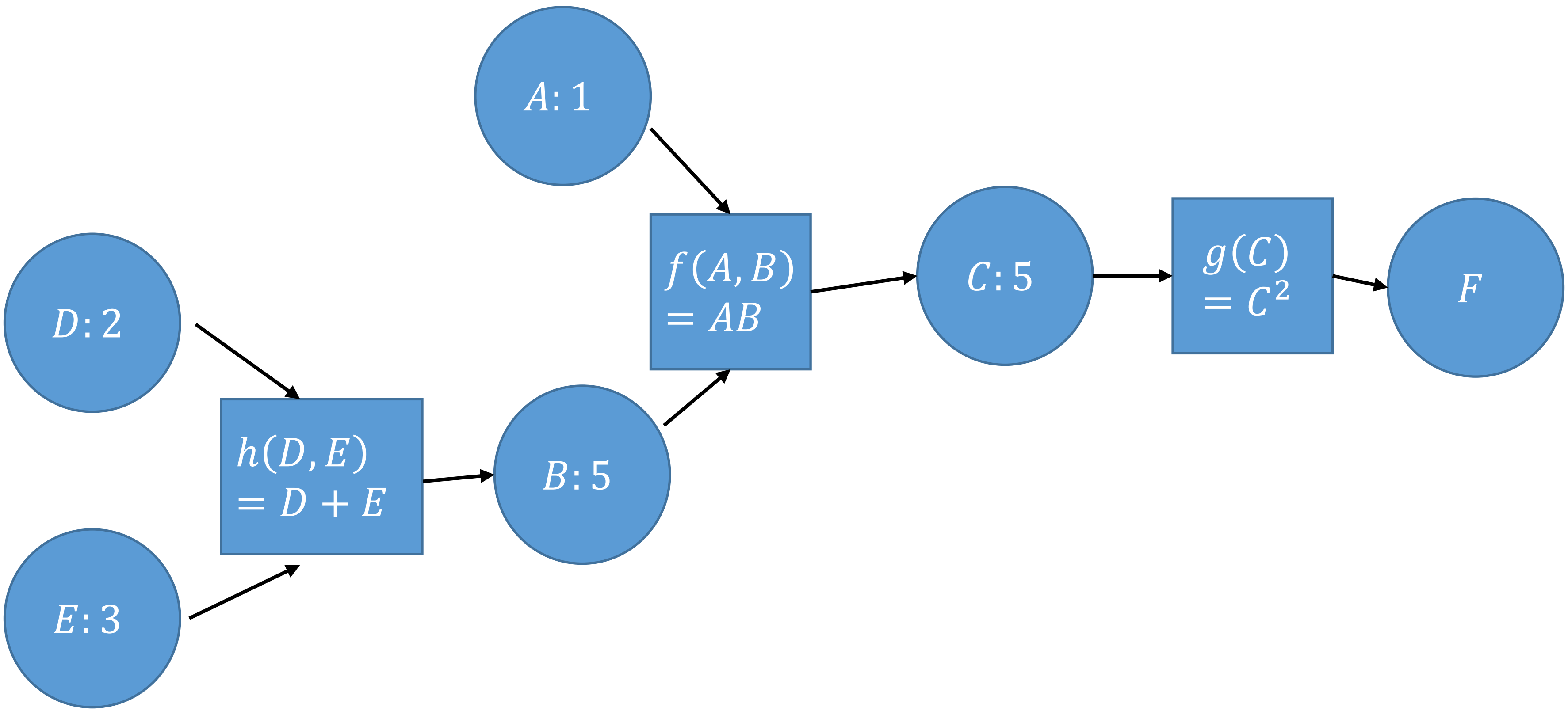
More Complicated Forward Pass



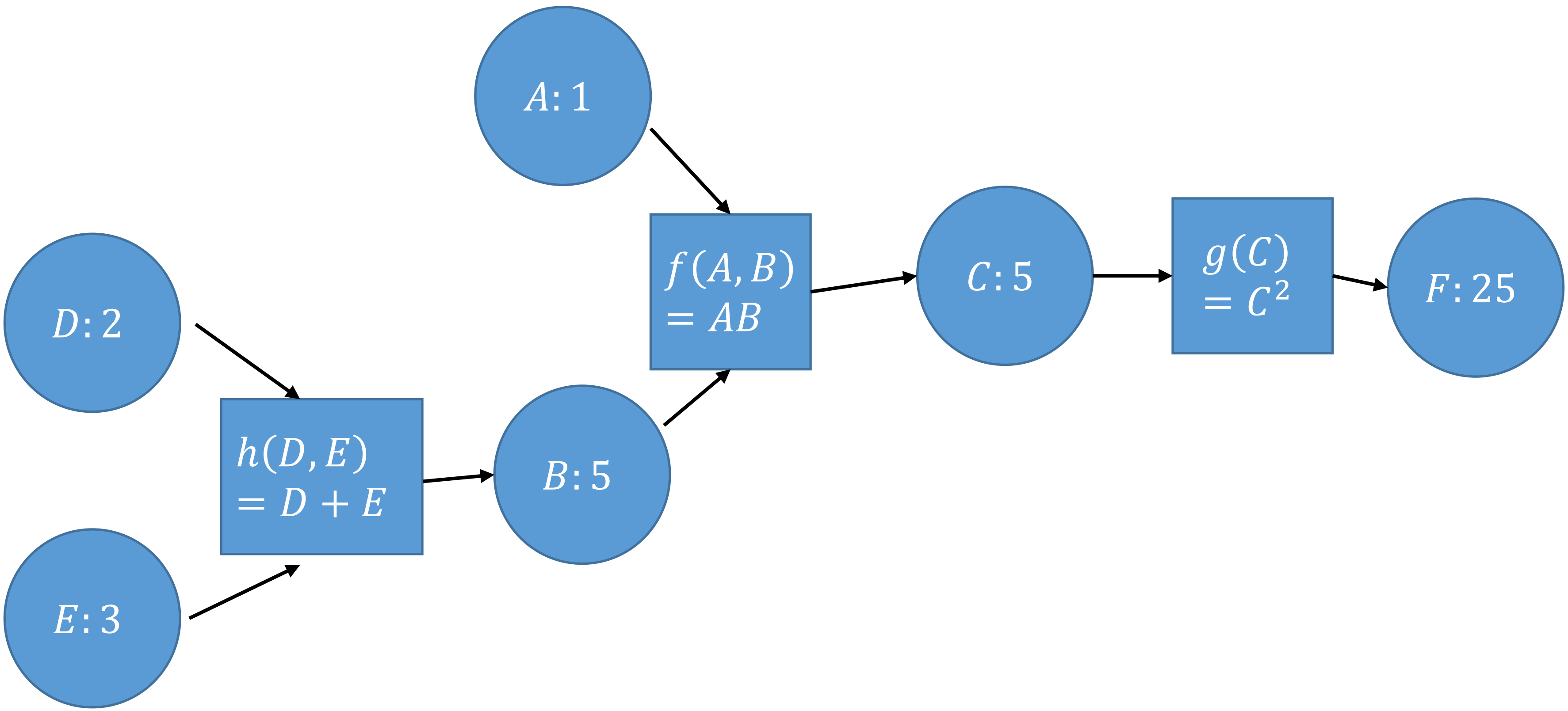
More Complicated Forward Pass



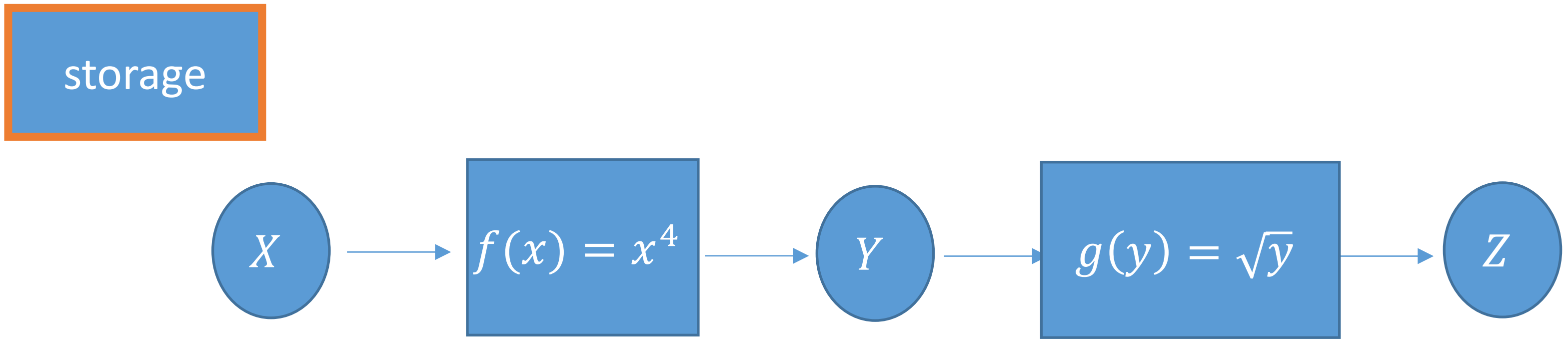
More Complicated Forward Pass



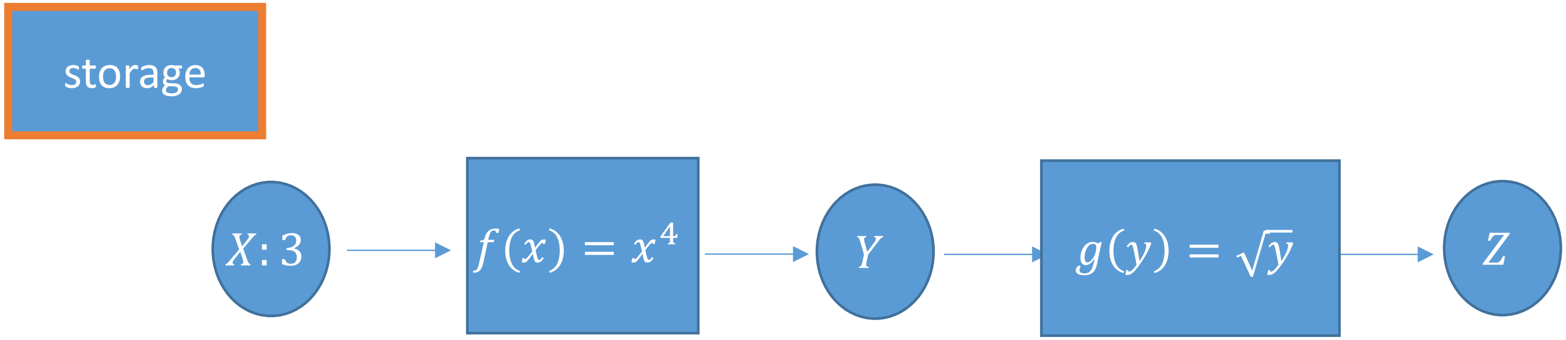
More Complicated Forward Pass



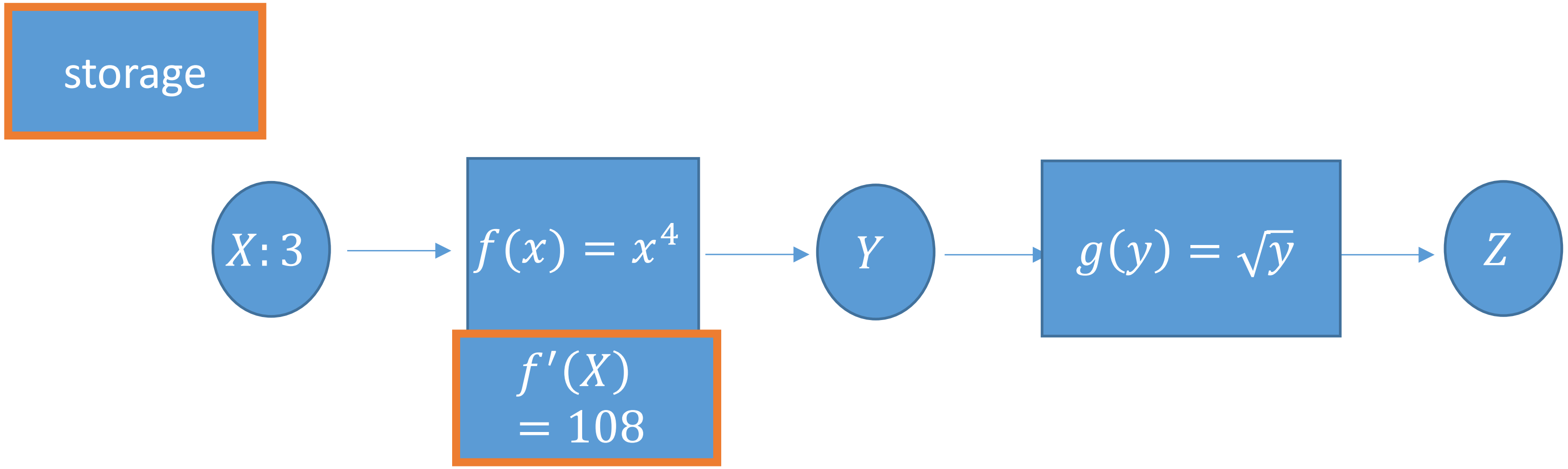
Compute Derivatives in Forward Pass



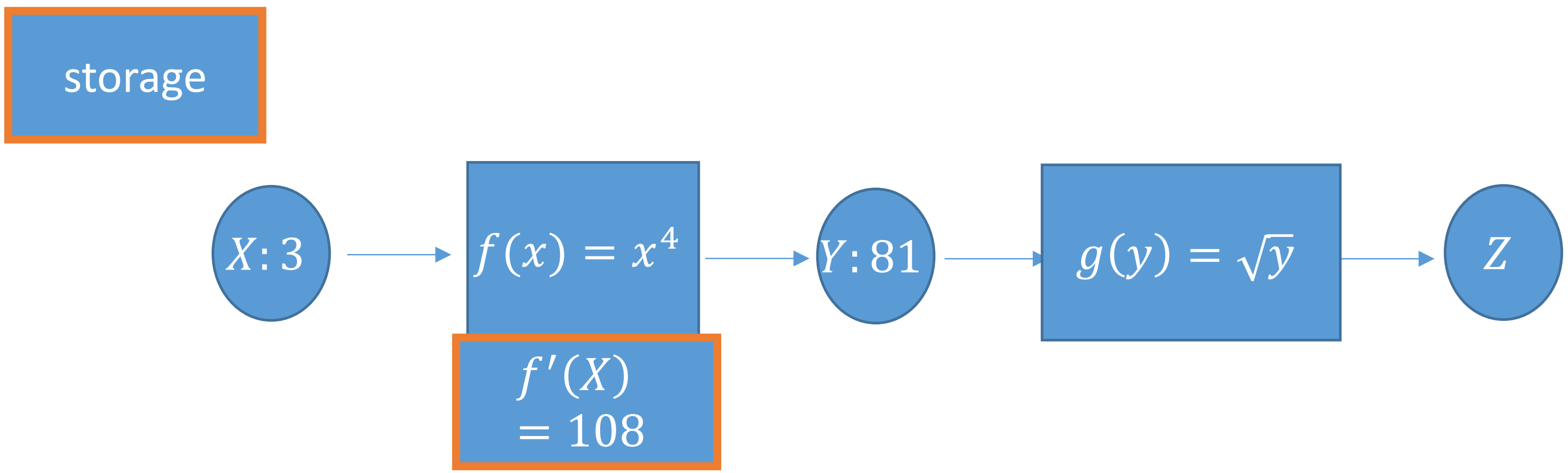
Compute Derivatives in Forward Pass



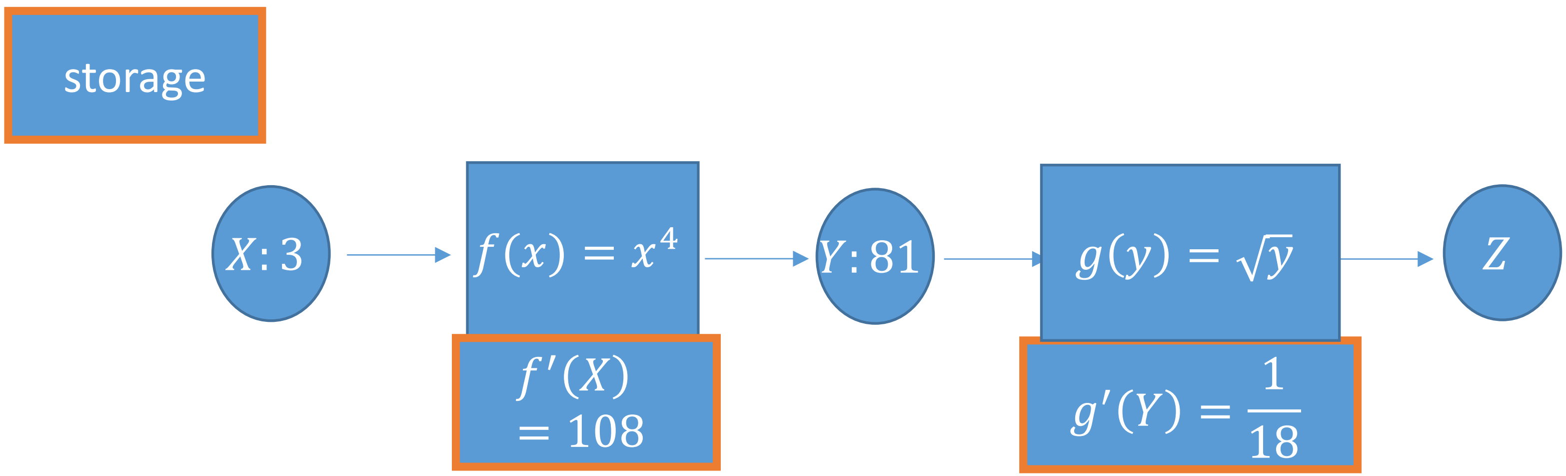
Compute Derivatives in Forward Pass



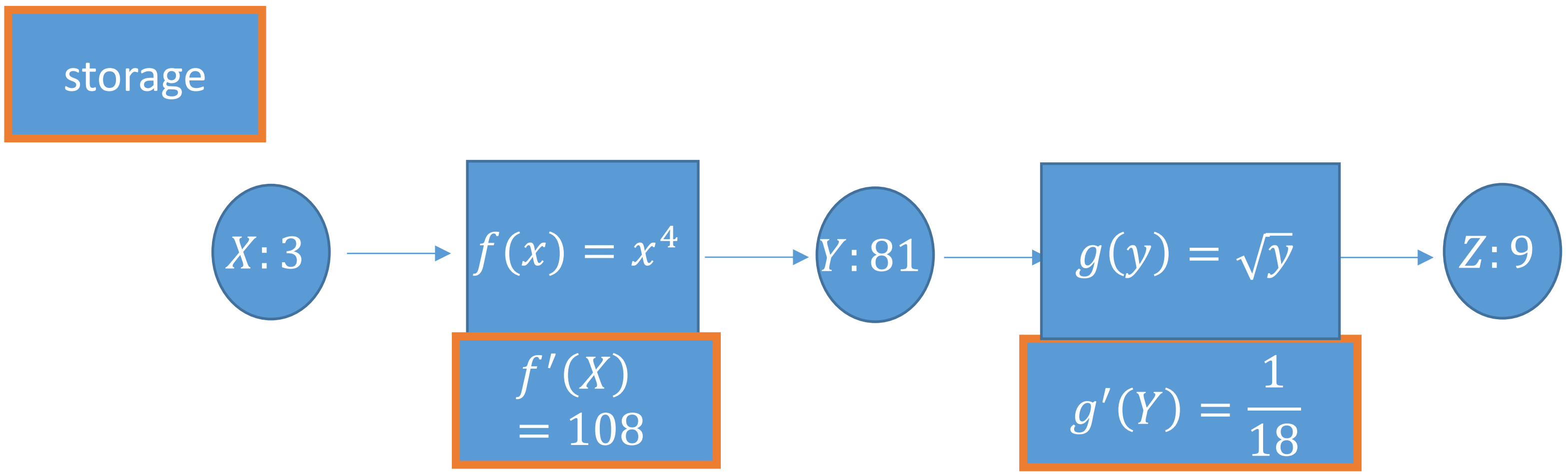
Compute Derivatives in Forward Pass



Compute Derivatives in Forward Pass



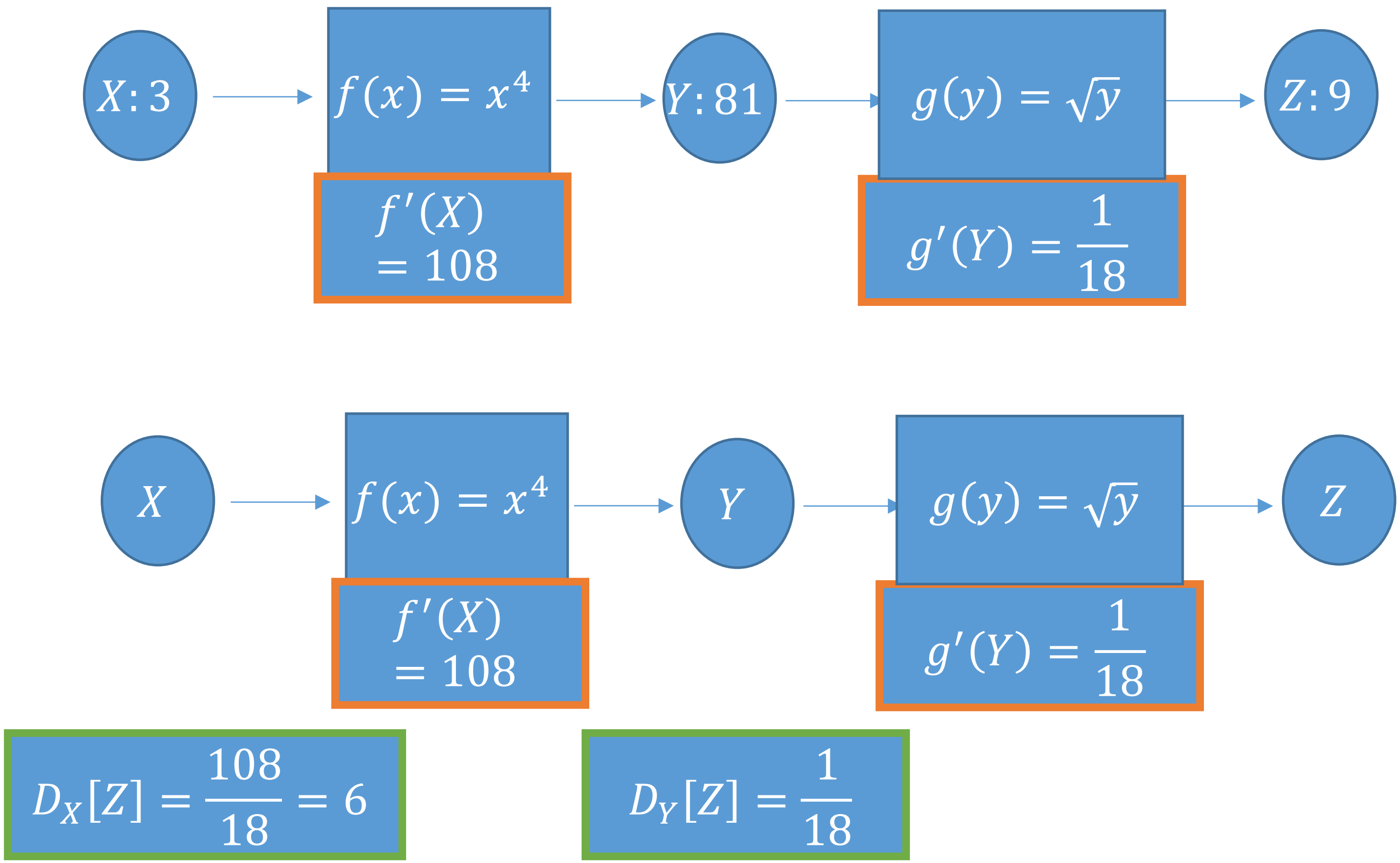
Compute Derivatives in Forward Pass



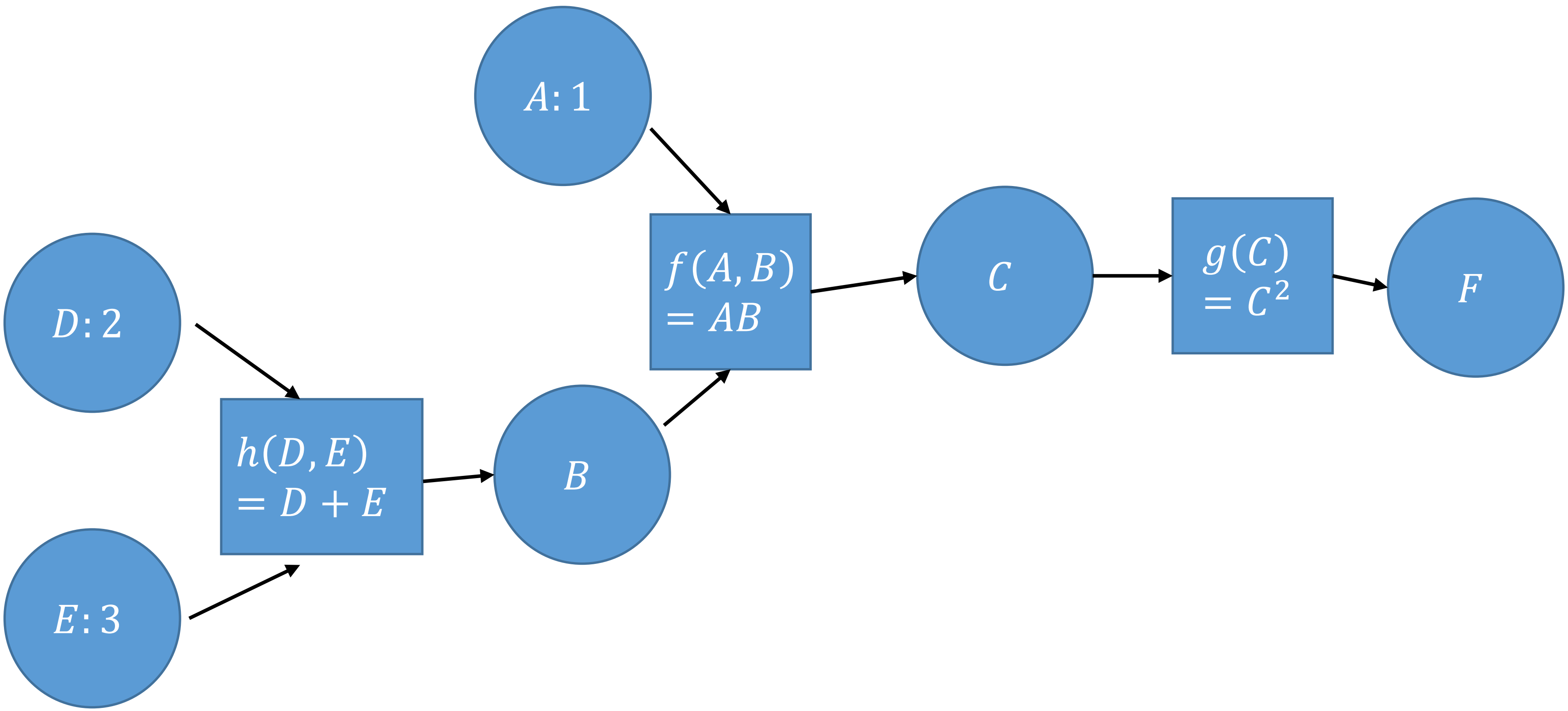
Backward Pass

- The backward pass uses the data in the forward pass to compute the gradient of the sink node with respect to every variable node.
- Forward pass propagates from the source variable nodes to the sink variable.
- Backward pass propagates backwards from sink node to source.
- You MUST do a forward pass before the backward pass.

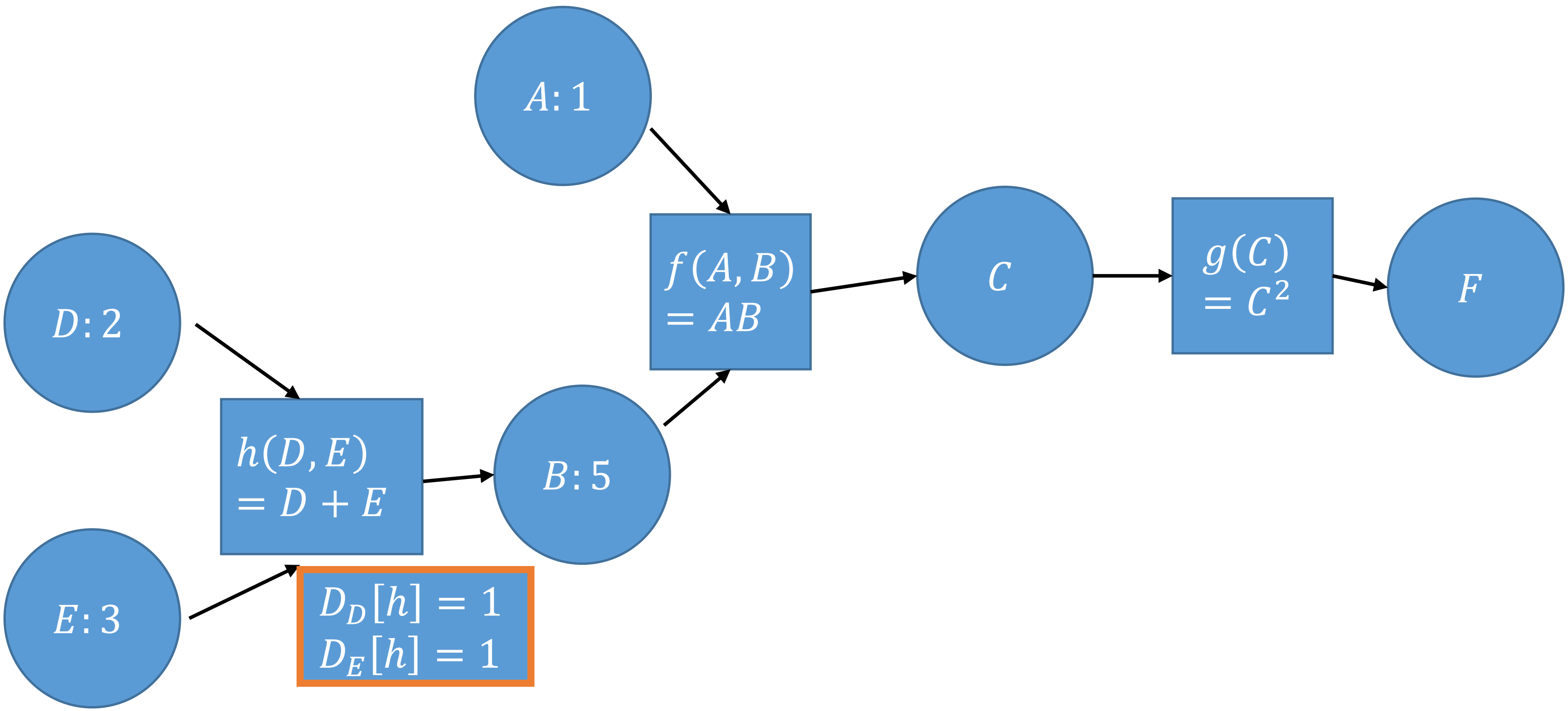
Backward Pass



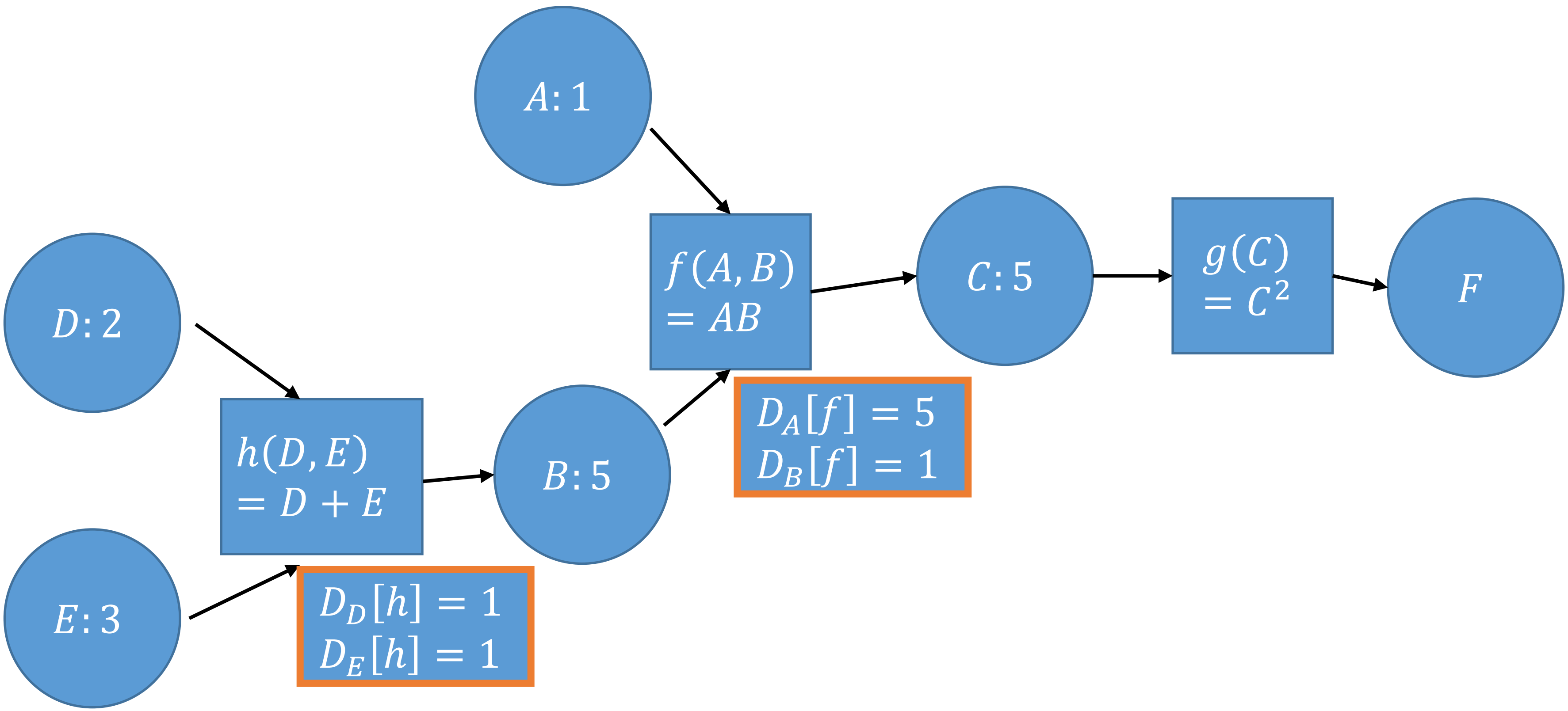
More Complicated Forward Pass



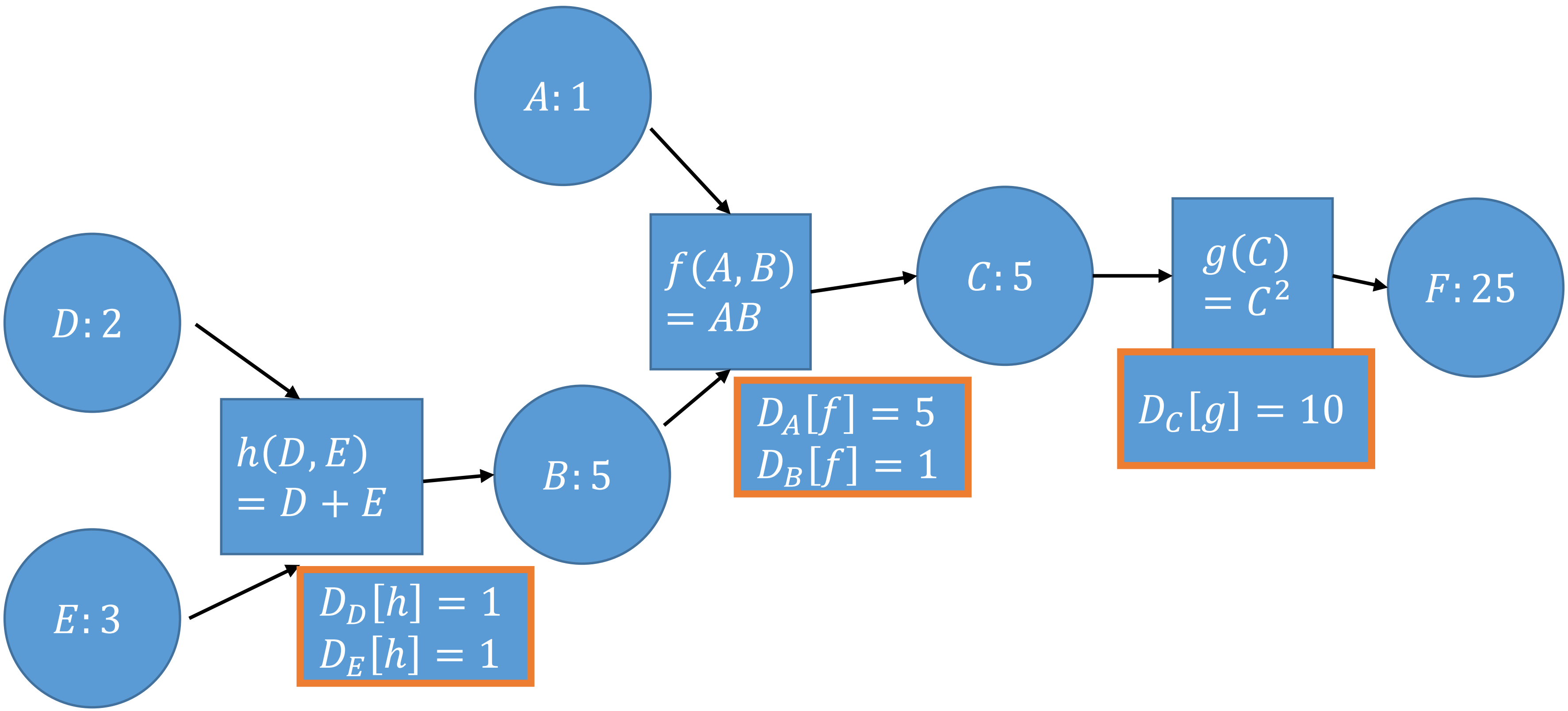
More Complicated Forward Pass



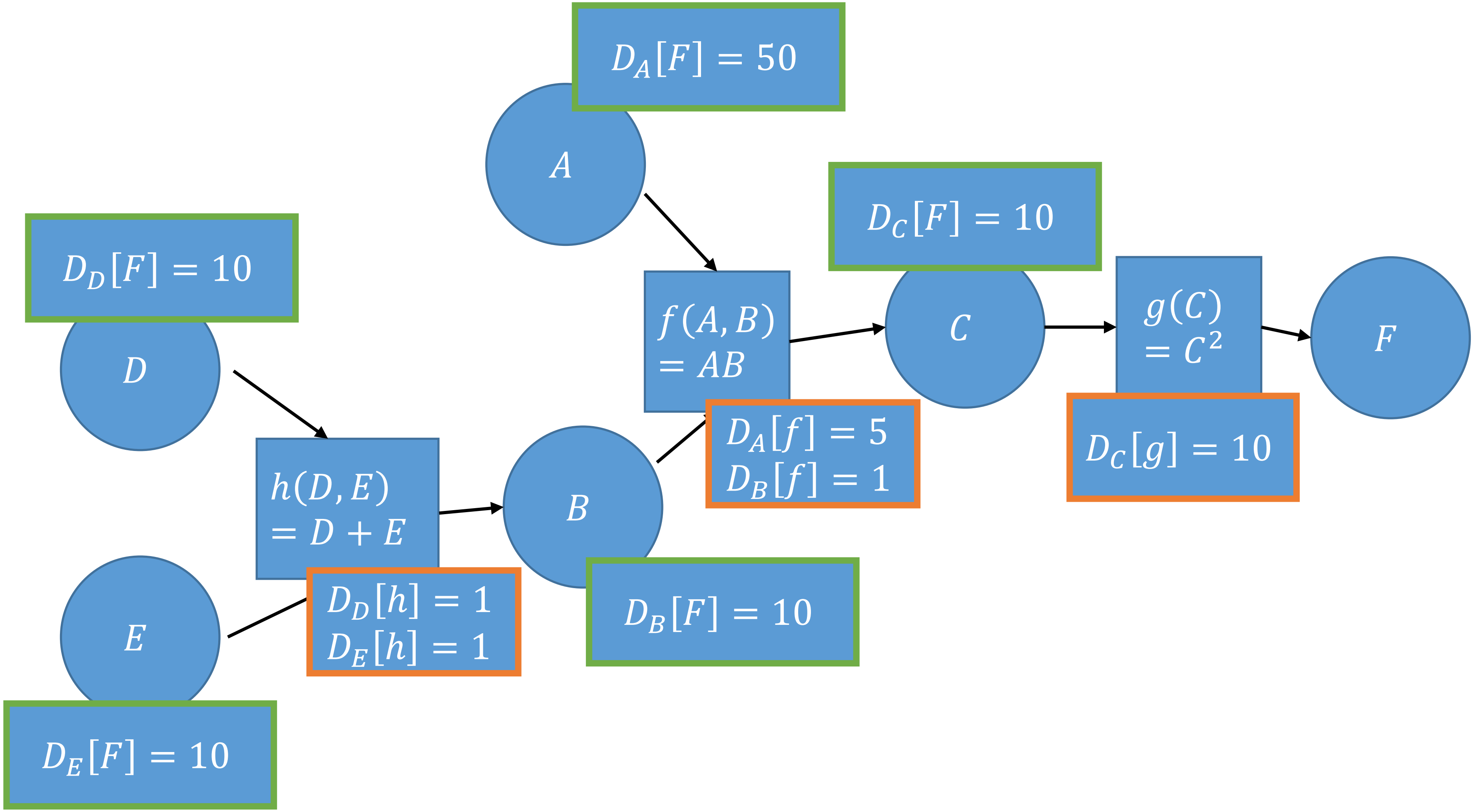
More Complicated Forward Pass



More Complicated Forward Pass



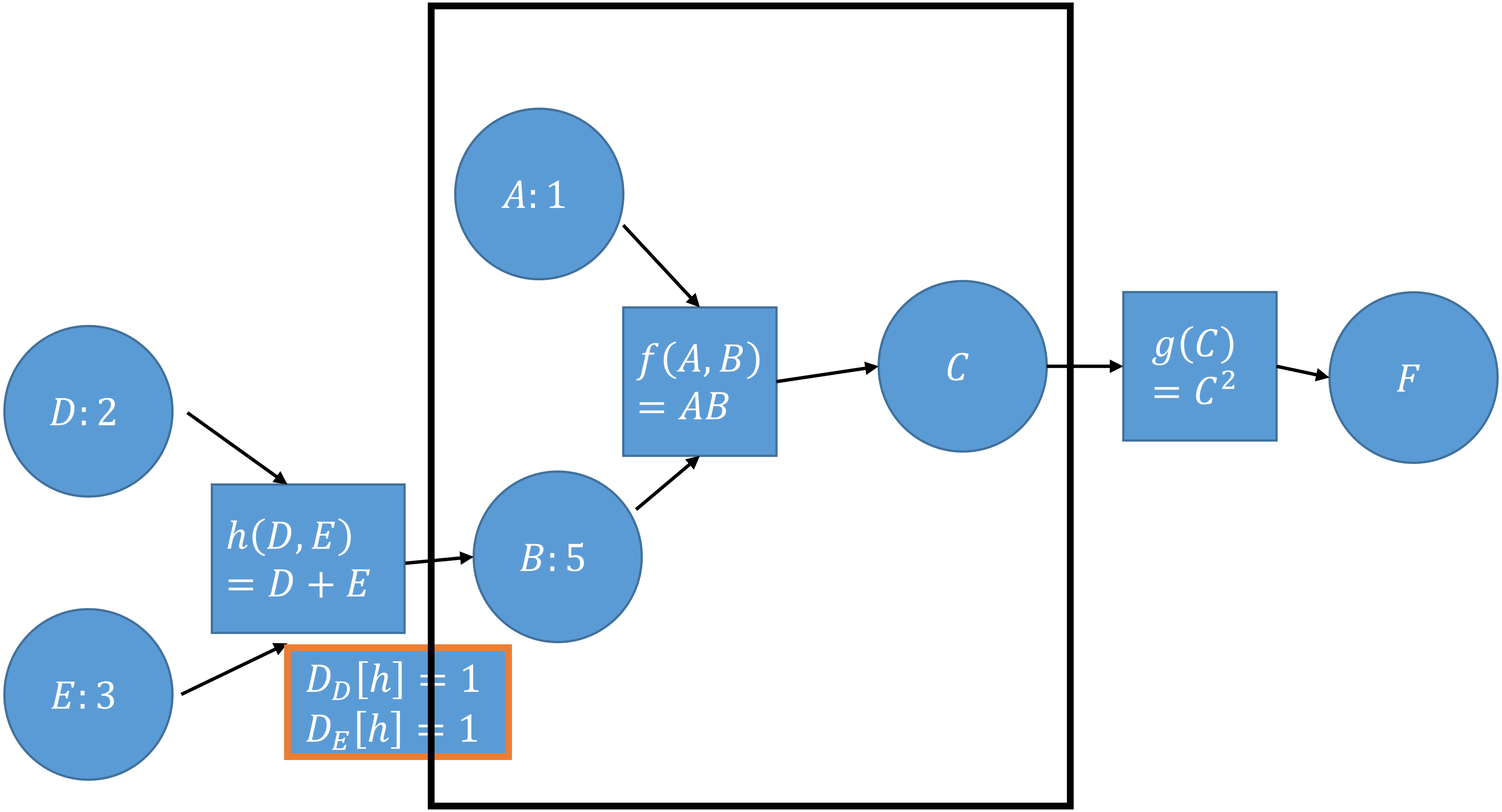
More Complicated Backward Pass



Backprop is "local"

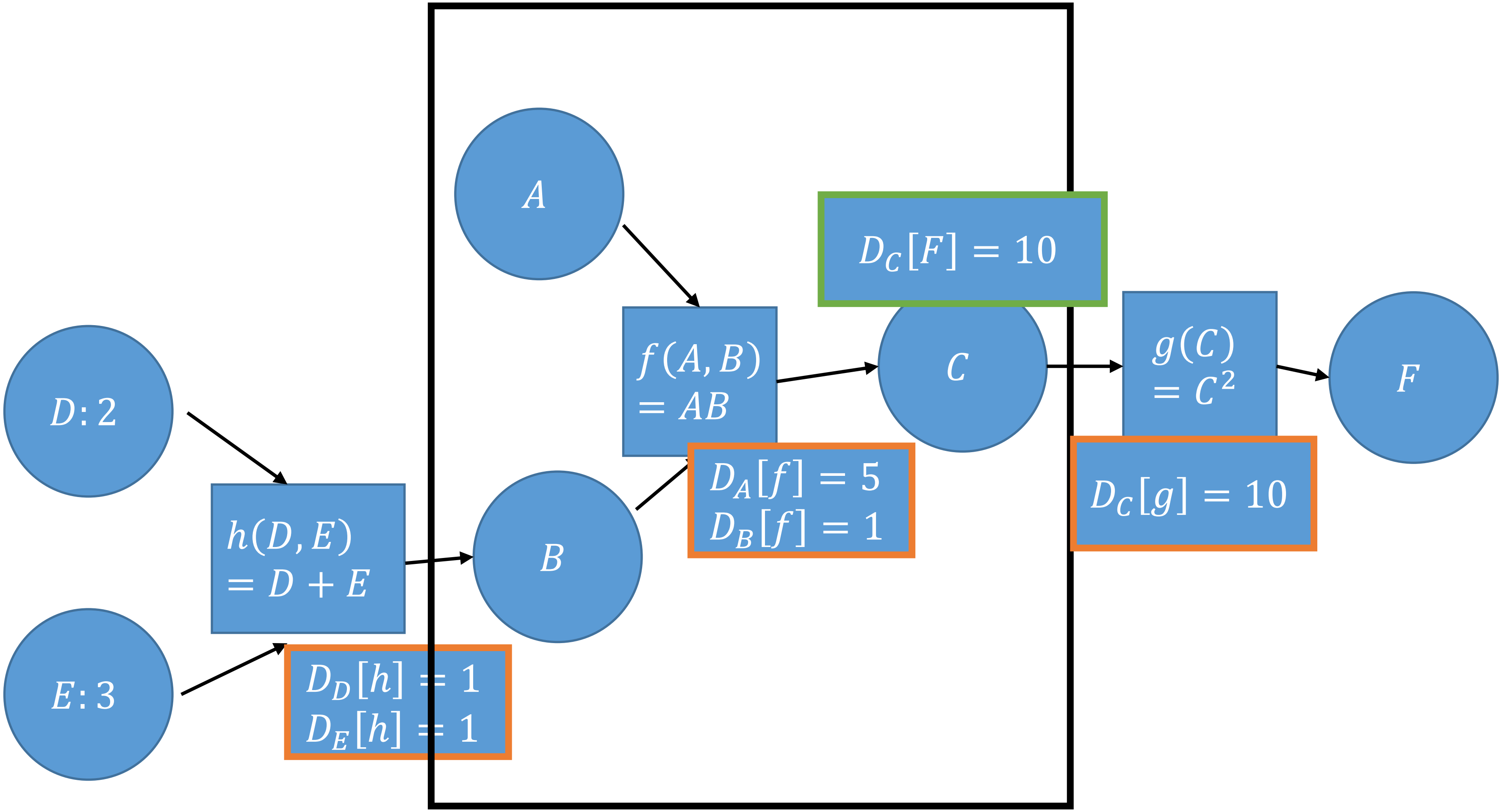
- Each "operator" node only needs to see its immediate neighbors in order to compute the both the forward and backward passes.

Forward pass is "local"



The f node doesn't need to know anything
Outside this box

Backward pass is "local"



The f node doesn't need to know anything
Outside this box

In Code

- Operation nodes have pointers to the input variable nodes and the output variable nodes.

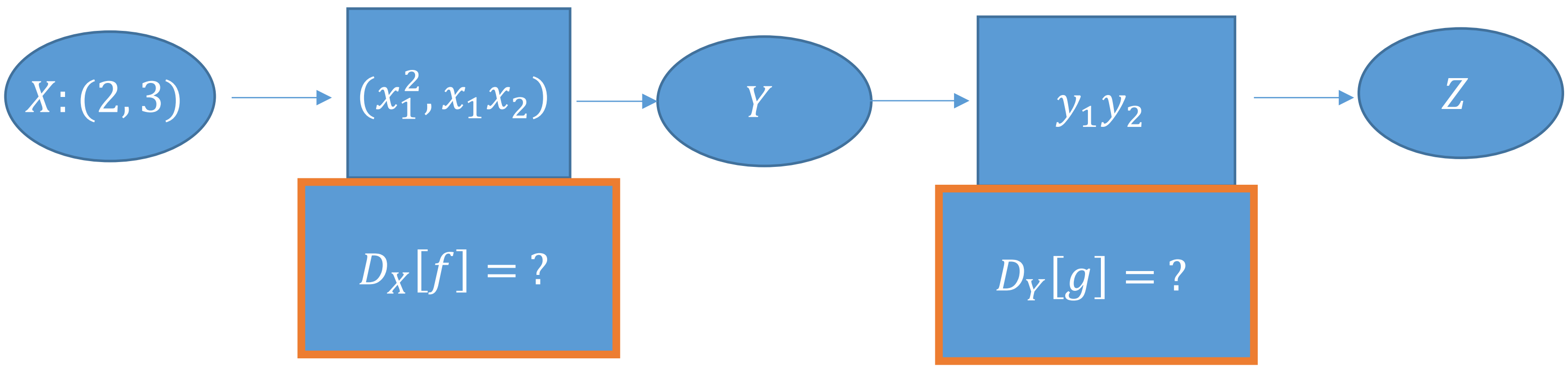
def forward(self, x):

1. Computes the output of the operation given the input, and assigns this output to all the output nodes.
2. Computes and stores relevant state (like the derivative).

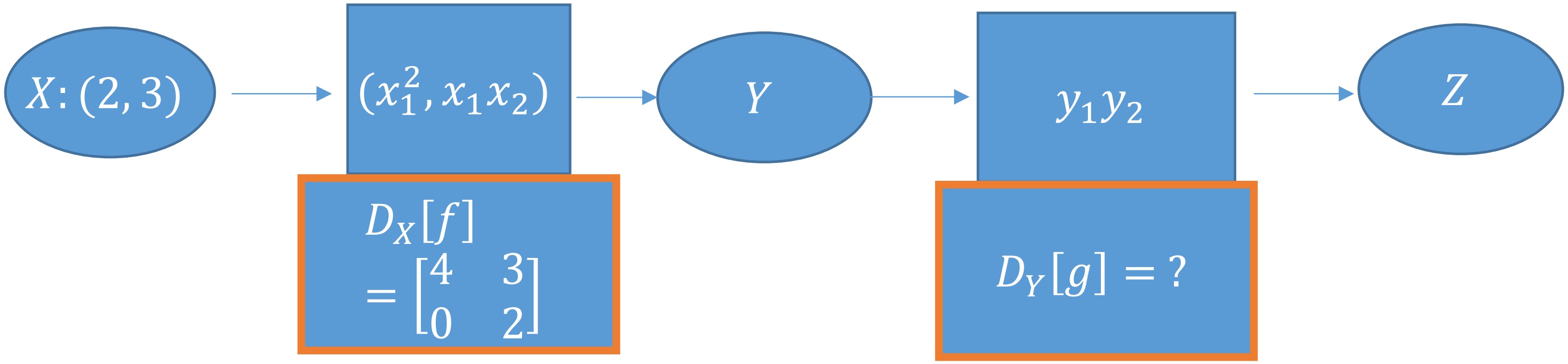
def backward(self, x):

1. Takes input equal to the derivative with respect to the output.
2. Multiplies by stored derivatives to compute derivatives with respect to the inputs.

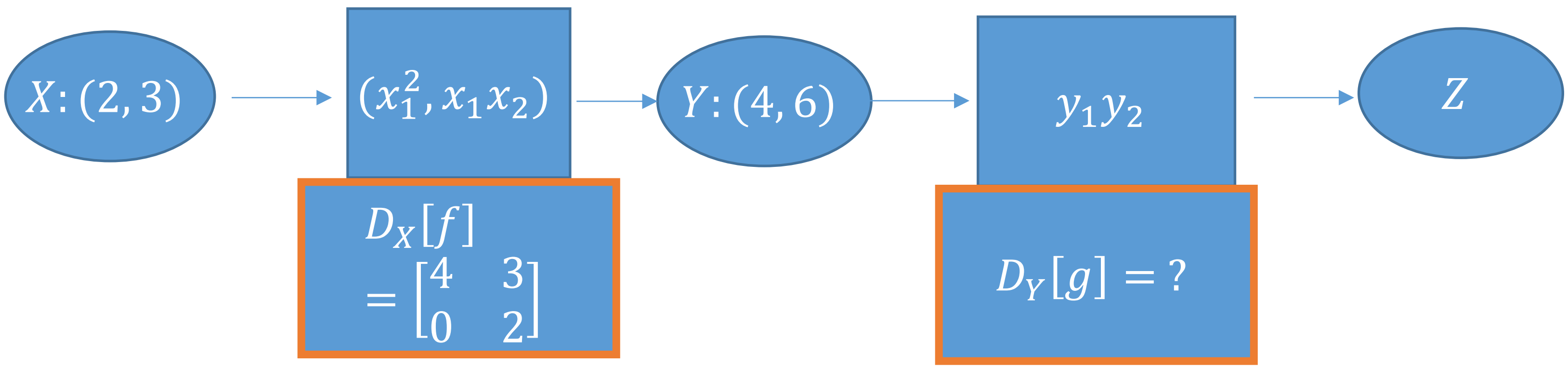
Forward Pass: Beyond Scalars



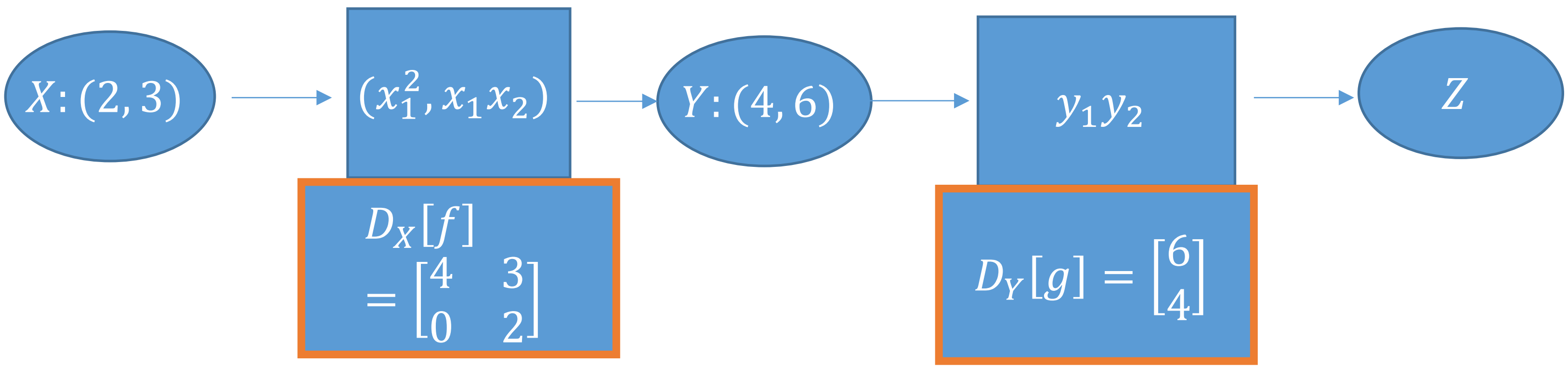
Forward Pass: Beyond Scalars



Forward Pass: Beyond Scalars

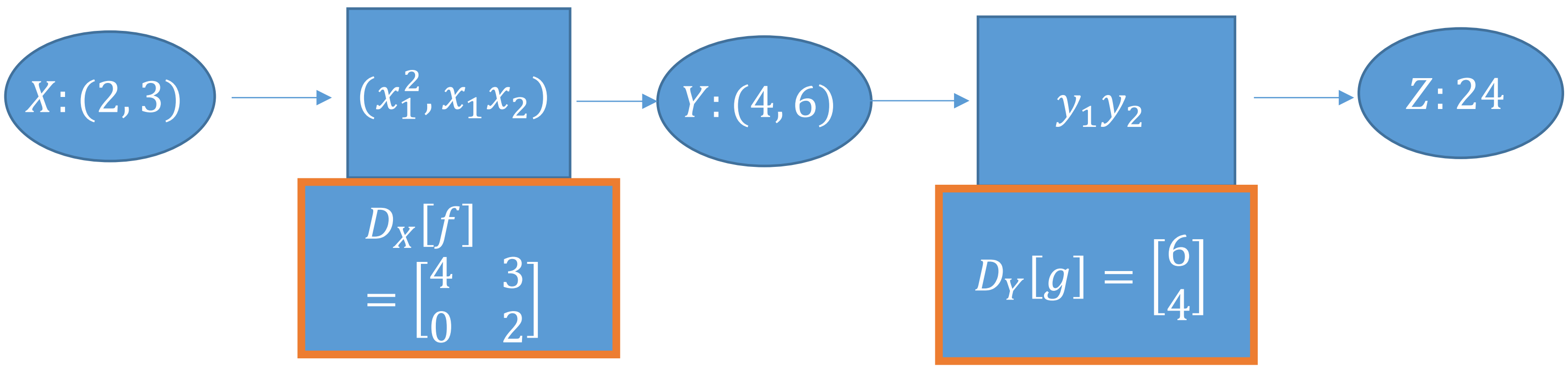


Forward Pass: Beyond Scalars

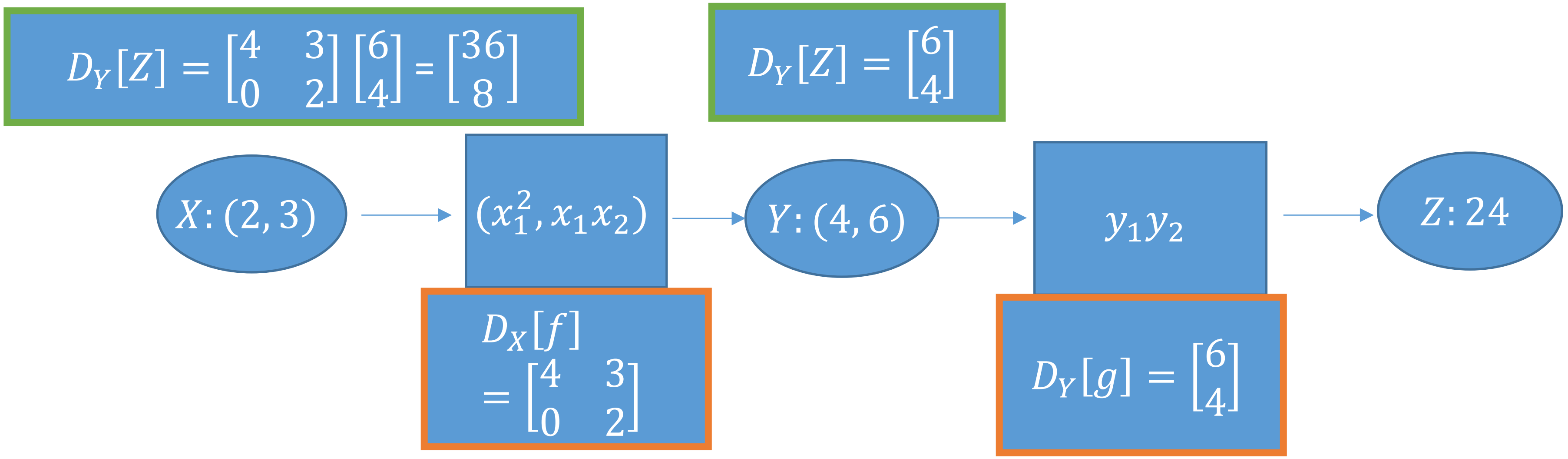


- What should the $D_X[Z]$ be?
- $Z = x_1^3 x_2$
- $D_X[Z] = \begin{bmatrix} 3x_1^2 x_2 \\ x_1^3 \end{bmatrix} = \begin{bmatrix} 36 \\ 8 \end{bmatrix}$

Backward Pass: Beyond Scalars

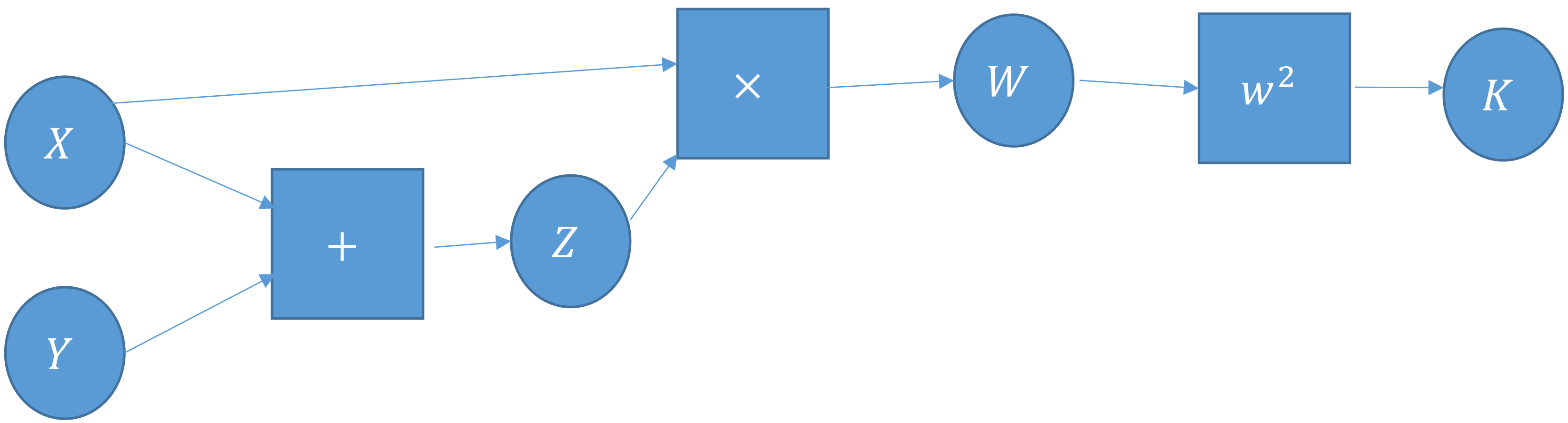


Backward Pass: Beyond Scalars



Even simple functions are operations

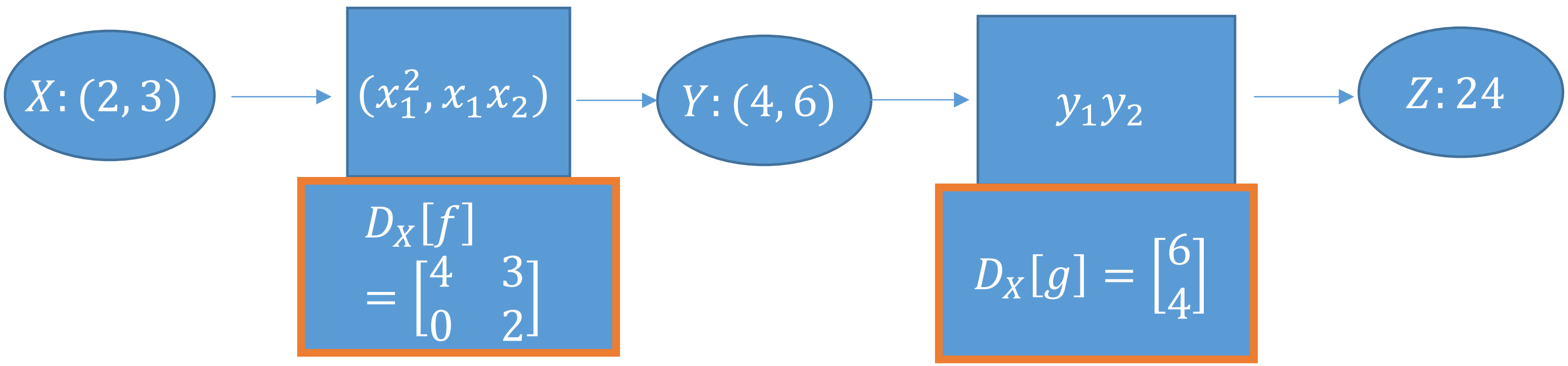
- $f(x, y) = x(x + y)^2$



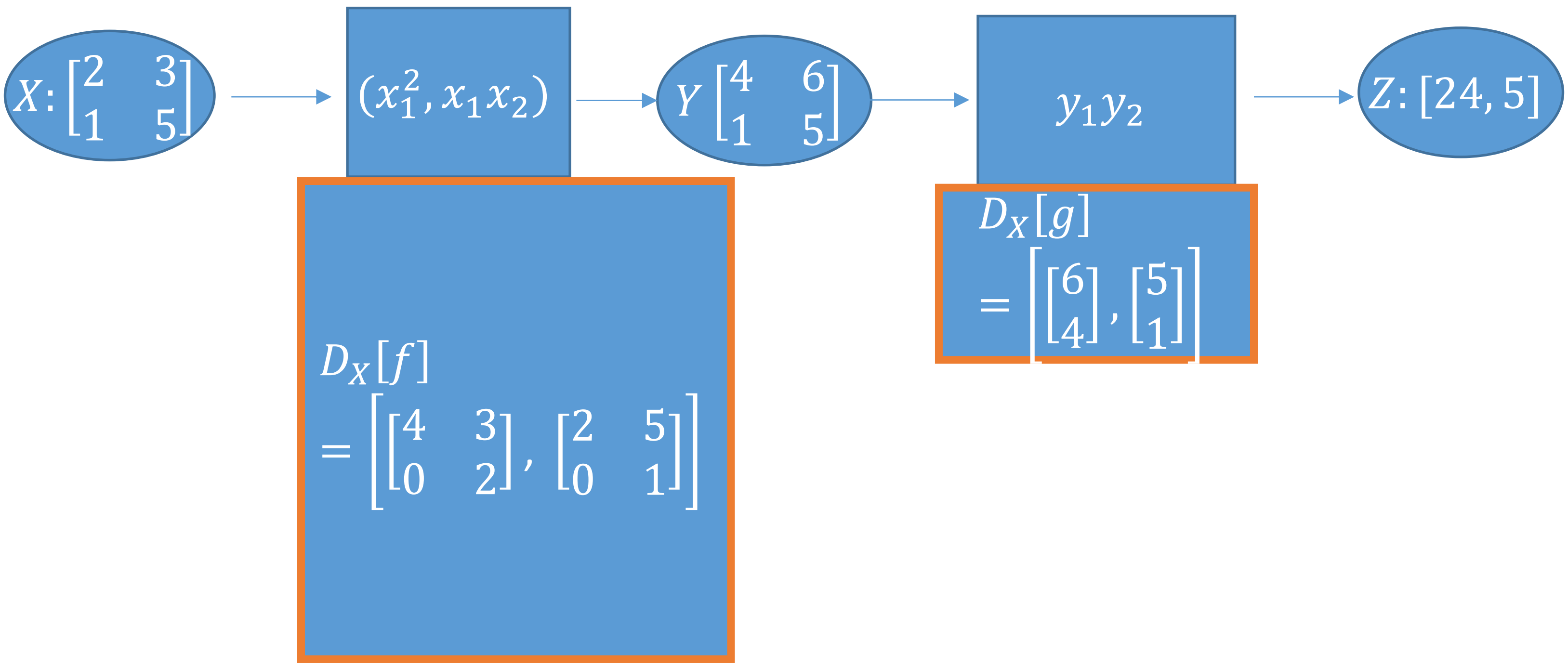
Derivatives of Batched functions

- Given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^d$, there is a *batched* function that distributed f along a batch dimension.
- $f_{batched}: \mathbb{R}^{B \times n} \rightarrow \mathbb{R}^{B \times d}$
- $f_{batched}(X)[i] = f(x[i])$
- All the functions on your homework are batched functions.
- To compute derivatives of batched functions, compute derivatives of each individual row.

Derivatives of Batched Functions



Derivatives of Batched Functions



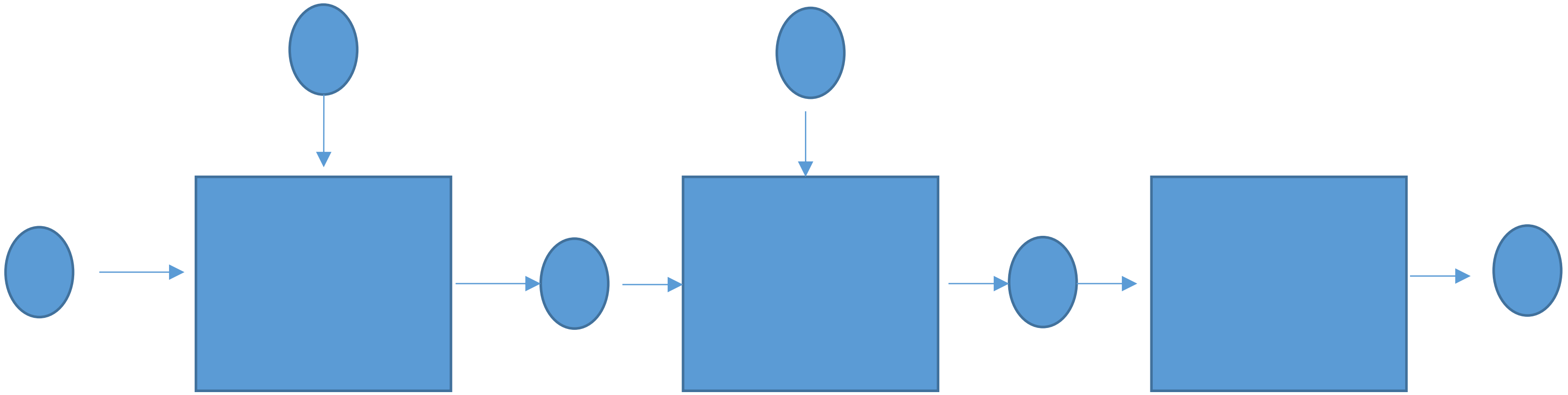
Derivatives of Matrix Functions

- To compute derivatives of a function that takes a matrix input, $f: \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^d$, unroll the matrix into a vector.
- You will need to reshape the gradient back into the same shape as the matrix.
- To avoid unrolling, use matrix derivatives from HW 1.

DIY: useful tricks and more examples

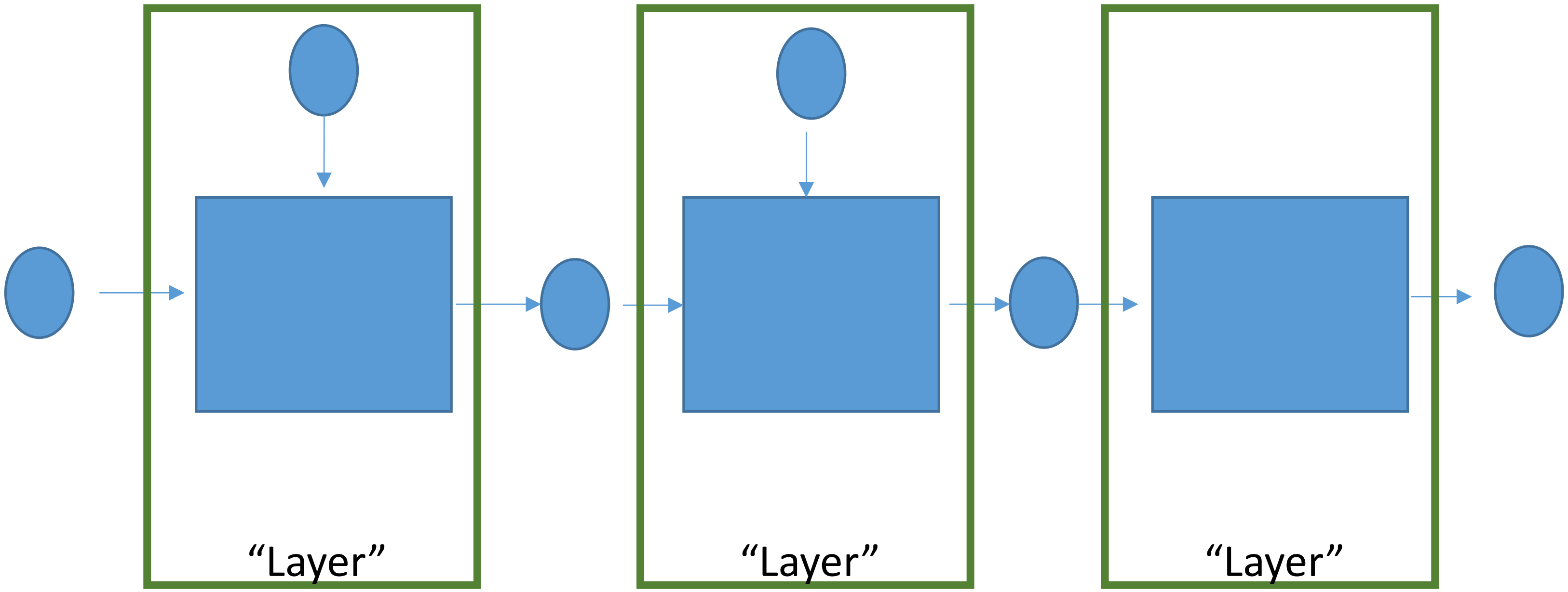
Simplified Graphs (on HW)

- Consider only graphs like:



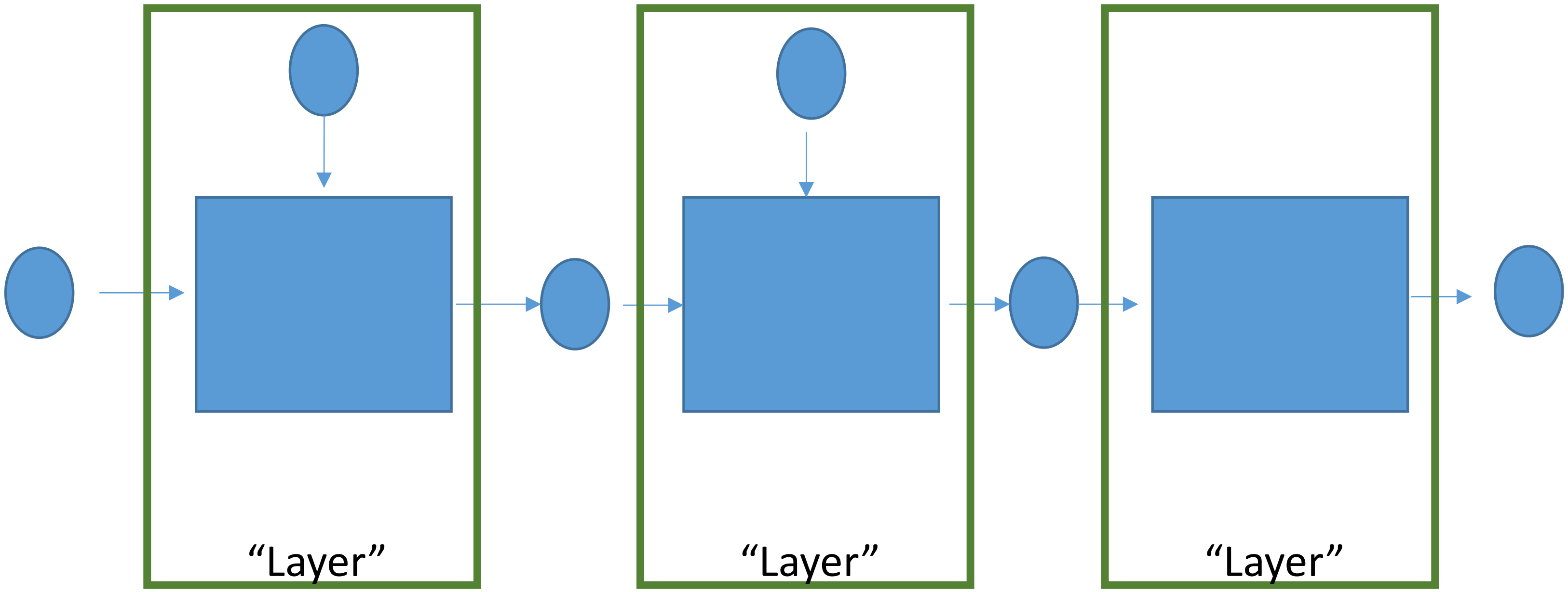
Simplified Graphs (on HW)

- Consider only graphs like:



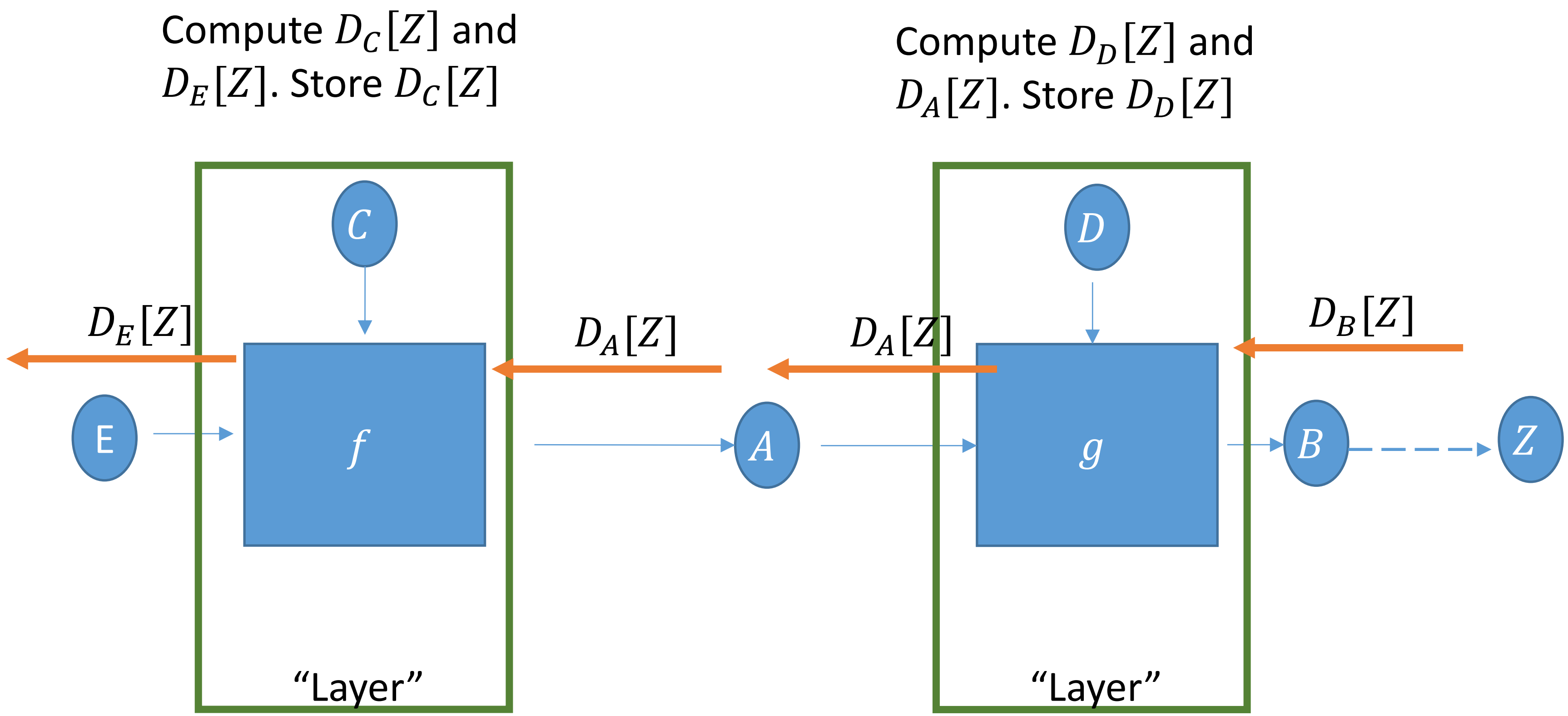
Simplified Graphs (on HW)

- Consider only graphs like:



- Only need gradients for variables inside layers

Backward Function (on HW)



Only consider the case $Z \in \mathbb{R}$, so that the shape of $D_X[Z]$ is always the same as X for all variables.

np.einsum

- Super-useful function used to express complicated matrix manipulations
- There are good tutorials online: see https://ajcr.net/Basic-guide-to-einsum/

np.einsum example

- `np.einsum('abc, bad -> ab', X, Y)`
- X and Y are both 3-dimensional matrices (tensors).
- X is associated with the string 'abc'
- Y is associated with the string 'bad'
- If X has shape [A, B, C], then Y has shape [B, A, D] for some other D.
- The output is a 2-dimensional tensor of shape [A, B]

np.einsum example

- `np.einsum('abc, bad -> ab', X, Y)`

- Let Z be the output. Then

- $Z[i,j] = \sum_{k,w} X[i,j,k] \times Y[j,i,w]$

- We summed over indices *not* present in Z

Another example

- `np.einsum('abcd, daa, be -> adf', X, Y, Z)`
- Inputs X, Y and Z have shapes [A, B, C, D], [D, A, A] and [B, E]
- Output W has shape [A, D, F]

- $W[a, d, f] = \sum_{b,c,e} X[a, b, c, d] \times Y[d, a, a] \times Z[b, e]$

Some familiar examples

- `Z= np.einsum('ij, jk -> ik', X, Y)`
- $Z[i,k] = \sum_j X[i,j] Y[j,k]$
  - Matrix multiplication

- `Z= np.einsum('ij-> ji', X)`
- $Z[j,i] = X[i,j]$
  - Matrix transpose

- `Z = np.einsum('ii->', X)`
- $Z = \sum_i X[i,i] = trace(X)$

Batched matrix multiplication

- $X \in \mathbb{R}^{B \times n}$
- $M \in \mathbb{R}^{B \times n \times m}$
- X is a batch of n dimensional vectors.
- M is a batch of $n \times m$ matrices.
- We want to compute $Y \in \mathbb{R}^{B \times m}$, where the ith row of Y is $X[i]M[i] \in \mathbb{R}^{m}$.
- How to write this?

Batched matrix multiply

- $X \in \mathbb{R}^{B \times n}$
- $M \in \mathbb{R}^{B \times n \times m}$
- We want to compute $Y \in \mathbb{R}^{B \times m}$, where the ith row of Y is $X[i]M[i] \in \mathbb{R}^m$.
- $Y[i,j] = \sum_k X[i,k]M[i,k,j]$
- `Y = np.einsum('ik, ikj -> ij', X, M)`

- Einsum in general is very good for batched operations.

Matrices and Tensors

- A matrix in $\mathbb{R}^{n \times m}$ is an n by m two dimensional array of real numbers.
  - A "list of lists". The outer list is size n, the inner lists are all size m.
- A tensor in $\mathbb{R}^{d_1 \times \cdots \times d_n}$ is a d_1 by d_2 by… by d_n n-dimensional array of real numbers.
  - A "list of lists of … of lists". Outer list has d_1 elements, inner-most lists all have d_n elements.
- We say a tensor in $\mathbb{R}^{a \times b \times c}$ has "shape" $[a, b, c]$.
- We access a tensor via multi-indices: $A[0,2,3]$.

Tensor Contraction

- Tensor contraction is the g generalization of matrix multiplication.
- Two tensors A and B can be contracted along one dimension if the last entry of A's shape is the same as the first entry of B's shape:
  - A has shape $[2, 45, 7]$ and B has shape $[7, 3, 67]$
- The output will have shape given by removing the common entry from A and B's shape and concatenating the lists:
  - $contract(A, B)$ has shape $[2, 45, 3, 67]$

Tensor Contraction Formula

- If A has shape $[a_1, \ldots, a_n, c]$ and b has shape $[c, b_1, \ldots, b_m]$, then

$$contract(A, B)[a_1, \ldots, a_n, b_1, \ldots, b_m] = \sum_{i=1}^{c} A[a_1, \ldots, a_n, i] \cdot B[i, b_1, \ldots, b_m]$$

- If $n = m = 1$, this is just matrix multiplication:

$$contract(A, B)[a, b] = \sum_{i=1}^{c} A[a, i] \cdot B[i, b]$$

Contraction along Many Dimensions

- Contraction along one dimension requires the last dimension of A to match the first dimension of B.
- Contraction along d dimensions requires the last d dimensions of A to match the first d dimensions of B.
- If $shape(A) = [2,5,4,6]$ and $shape(B) = [4,6,8,2]$, then A and B can be contracted along 2 dimensions.
  - The shape of $contract(A, B, 2)$ is [2,5,8,2].
- This is the `tensordot` function in `numpy`.

Contraction along Many Dimensions

- If A has shape $[a_1, \ldots, a_n, c_1, \ldots, c_k]$ and B has shape $[c_1, \ldots, c_{1=k}, b_1, \ldots, b_m]$, then $contract(A, B, k)$ has shape $[a_1, \ldots, a_n, b_1, \ldots, b_m]$.
- The conctraction is computed by:
$contract(A, B, k)[a_1, \ldots, a_n, b_1, \ldots, b_m]$
$= \sum_{i_1=1}^{c_1} \ldots \sum_{i_k=1}^{c_k} A[a_1, \ldots, a_n, i_1, \ldots i_k] \cdot B[i_1, \ldots, i_k, b_1, \ldots, b_m]$

What is the point of contraction?

- Using tensor contraction, we can write a linear map that takes matrices as both inputs and outputs.
- Let A have shape $[2, 3, 4, 5]$. Then A specifies a linear map that takes 2×3 matrices as input, and outputs 4×5 matrices:
- $A(M) = contract(M\ , A, 2)$
- Tensor contraction allows us to specify linear maps between arbitrary tensor shapes as also tensors of larger shape.

Derivatives of Tensor-valued functions

- Let $f: \mathbb{R}^{d_1 \times d_2 \times d_3} \to \mathbb{R}^{a \times b}$. The derivative is
$$D[f](x) \in \mathbb{R}^{d_1 \times d_2 \times d_3 \times a \times b}$$
- The derivative is a tensor that can contract all of the dimensions of the input and produce the dimensions of the output:
$$D[f](x): \mathbb{R}^{d_1 \times d_2 \times d_3} \to \mathbb{R}^{a \times b}$$
- The derivative formula:
$$D[f](x)[i,j,k,z,w] = \frac{df_{z,w}}{dx_{i,j,k}}(x)$$

Derivatives of Scalar-valued functions

- Let $f: \mathbb{R}^{d_1 \times d_2 \times d_3} \to \mathbb{R}$. The derivative is
$$D[f](x) \in \mathbb{R}^{d_1 \times d_2 \times d_3 \times 1}$$
- In this special case, we will usually drop the $\times 1$ of $D[f](x)$ to be in $\mathbb{R}^{d_1 \times d_2 \times d_3}$.
- The tensor contraction is:
$$contract\big(X, D[f](x)\big) = sum(X \odot D[f](x))$$
- Here $\odot$ is the coordinate-wise product (just normal multiplication in numpy), and sum just adds all the entries.