Neural Networks refresher

This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. Computation graphs
4. Loss functions and training
5. Design patterns

This lecture

A quick review of topics you should have already seen before

1. *Neural networks*
2. Tensors
3. Computation graphs
4. Loss functions and training
5. Design patterns

Artificial neurons

*Functions that **very loosely** mimic a biological neuron*

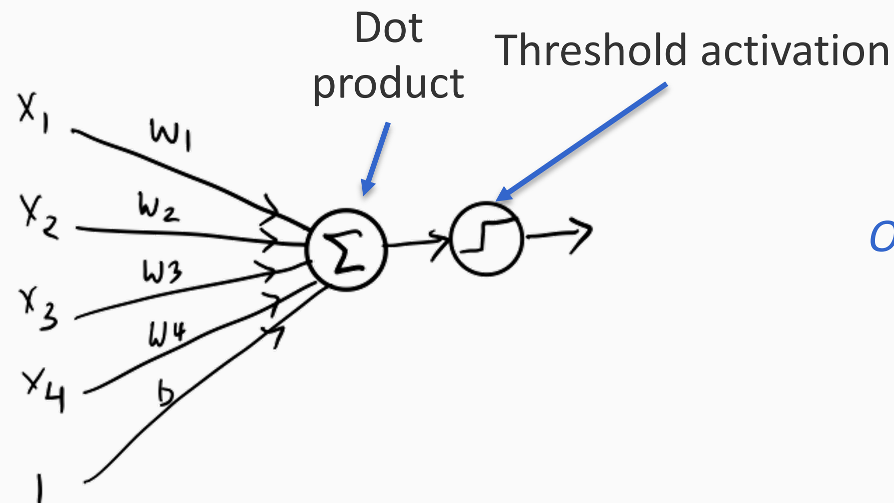
A basic neuron accepts a collection of inputs (a vector **x**) and produces an output by:

- Applying a dot product with weights **w** and adding a bias b
- Applying a (possibly non-linear) transformation called an ***activation***

$$output = activation(\boldsymbol{w}^T \boldsymbol{x} + b)$$

Dot product

Threshold activation



Other activations are possible

Activation functions

Also called transfer functions

$$output = activation(\boldsymbol{w}^T \boldsymbol{x} + b)$$

Name of the neuron	$activation(z)$
Linear unit	z
Threshold/sign unit	$\mathrm{sgn}(z)$
Sigmoid unit	$\frac{1}{1 + \exp(-z)}$
Rectified linear unit (ReLU)	$\max(0, z)$
Tanh unit	$\tanh(z)$

Many more activation functions exist (sinusoid, sinc, gaussian, polynomial…)

A neural network

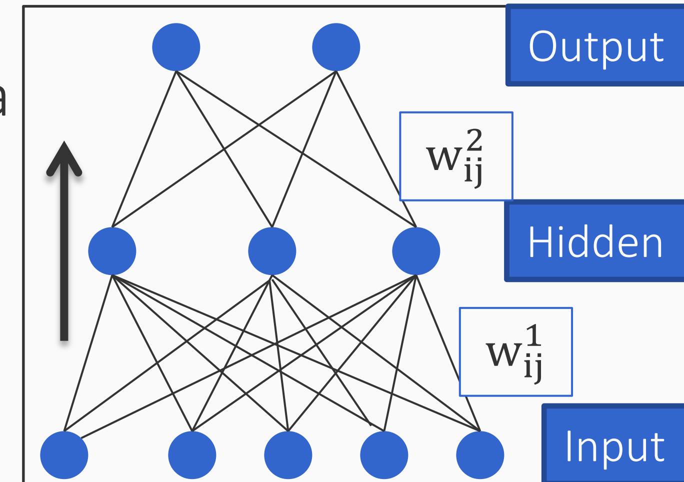
A function that converts inputs to outputs defined by a directed acyclic graph

- Nodes organized in layers, correspond to neurons
- Edges carry output of one neuron to another, associated with weights

A neural network

A function that converts inputs to outputs defined by a directed acyclic graph

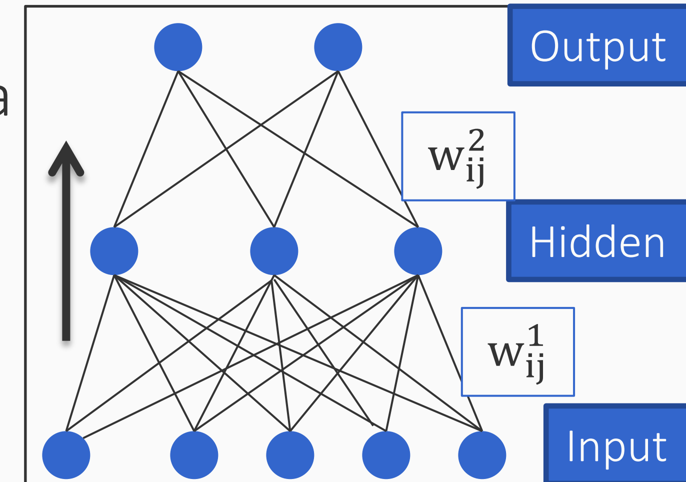
- Nodes organized in layers, correspond to neurons
- Edges carry output of one neuron to another, associated with weights



A neural network

A function that converts inputs to outputs defined by a directed acyclic graph

- Nodes organized in layers, correspond to neurons
- Edges carry output of one neuron to another, associated with weights



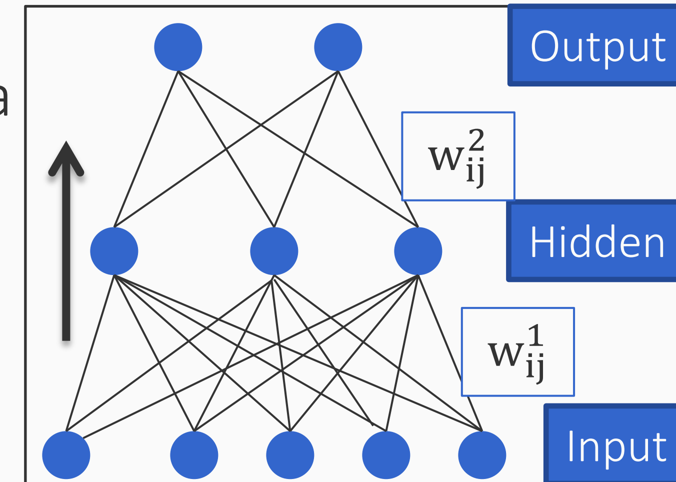
To define a neural network, we need to specify:

- The structure of the graph
  - How many nodes, the connectivity
- The activation function on each node
- The edge weights

A neural network

A function that converts inputs to outputs defined by a directed acyclic graph

- Nodes organized in layers, correspond to neurons
- Edges carry output of one neuron to another, associated with weights



To define a neural network, we need to specify:

- The structure of the graph
  - How many nodes, the connectivity
- The activation function on each node
- The edge weights

Called the *architecture* of the network
Typically predefined, part of the design of the classifier

Learned from data

This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. *Tensors*
3. Computation graphs
4. Loss functions and training
5. Design patterns

Neural networks are differentiable computation units that operate on and produce tensors

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"

Scalars (i.e., numbers) are tensors with zero dimensions. 3, -1, 1.1, …

Why zero dimensions? Because we need zero indices to find only element contained in it

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"

Vectors are one dimensional tensors:
[1,2,3], [-11.3, 0], …

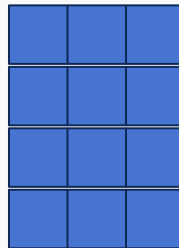
Why one dimensional? Because we need one index to address any element in the vector

The shape of this tensor is 6.
It is a six dimensional vector.

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"



Matrices are two dimensional tensors

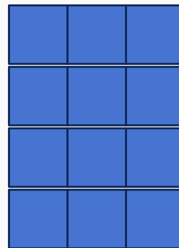
Why two dimensional? Because we need two indexes to address any element in it

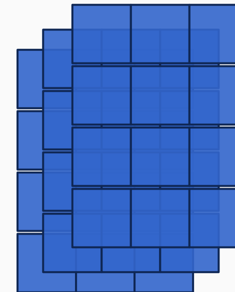
The shape of this tensor is (4, 3). It is a 4×3 matrix.

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"





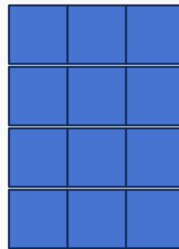
This is a three dimensional tensor. We need three indexes to address any element in it.

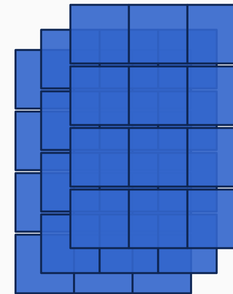
Its shape is (4, 3, 3)

Tensors: A quick primer

Tensors generalize vectors and matrices

- For the most part (in what we will see), whenever you see tensor, you can think "multi-dimensional arrays"





And so on…

Operations on tensors

Indexing to obtain sub-tensors (or scalars). Examples:

- $x[i], M[i,j], A[i,j,k], \ldots$ (sometimes written using subscripts): Look up an entry in a vector x or a matrix M or a 3-dimensional tensor A
- $M[i,:]$ (using `numpy` notation): Lookup the ith row of the matrix M
- $A[i,:,:]$ (using `numpy` notation): Lookup the ith slice of tensor A to produce a matrix
- $T[:,:,:,i]$ (using `numpy` notation): Lookup the ith sub-tensor of a 4-dimensional T to produce a 3-dimensional tensor

Operations on tensors

Tensors of the same shape can be:

- Added: Add the corresponding elements
- Multiplied element-wise: Multiply corresponding elements
- …any binary operation on numbers can be applied elementwise

Operations on tensors

Tensors can be multiplied using a generalization of matrix multiplication

Sometimes this is called **Tensor Mode-n Multiplication**

Operations on tensors

Tensors can be multiplied using a generalization of matrix multiplication

Sometimes this is called **Tensor Mode-n Multiplication**

Suppose we have $A \in \Re^{M \times N}, B \in \Re^{N \times K}$

We can define the product of A and B to produce a tensor C as follows:

$$C[m,k] = \sum_{n} A[m,n]B[n,k]$$

This is just matrix-matrix multiplication

Operations on tensors

Tensors can be multiplied using a generalization of matrix multiplication

Sometimes this is called **Tensor Mode-n Multiplication**

Suppose we have $A \in \Re^{M \times N}, B \in \Re^{N \times K}$
We can define the product of A and B to produce a tensor C as follows:

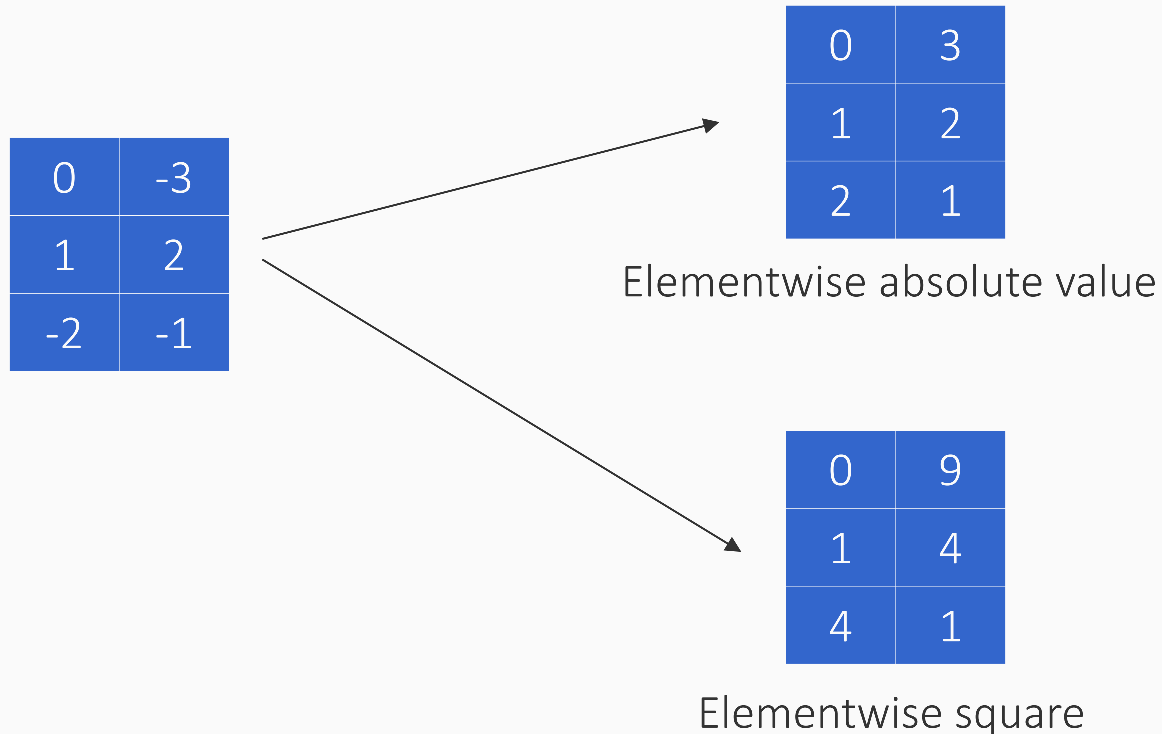
$$C[m,k] = \sum_{n} A[m,n]B[n,k]$$

Suppose we have $A \in \Re^{M \times N \times R}, B \in \Re^{N \times K}$
We can define the product of A and B to produce a tensor C as follows:

$$C[m,r,k] = \sum_{n} A[m,n,r]B[n,k]$$

Operations on tensors

Elementwise operations: Apply some function to each element of the tensor



Operations on tensors

Reshape: Re-organize the numbers in the tensor to produce a tensor of a different shape and/or dimensionality

4
3.5
1.2
-3
9
-1

4	-3
3.5	9
1.2	-1

A 6 dimensional tensor reshaped to a 3×2 matrix

There is a lot more about tensors that you can learn by working with them, e.g. with PyTorch

This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. *Computation graphs*
    1. What is the semantics of a computation graph?
    2. How do perform computations with them
4. Loss functions and training
5. Design patterns

Neural networks are differentiable computation units that operate on and produce tensors

Computation graphs

A language for constructing deep neural networks and loss functions

- A way to think about ***differentiable compute***

Key ideas:

- We can represent functions as graphs
- We can dynamically generate these graphs if necessary
- We can define algorithms over these graphs that map to learning and prediction
  - Prediction via the forward pass
  - Learning via gradients computed using the backward pass

This lecture

A quick review of topics you should have already seen before

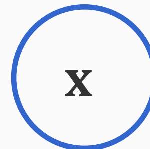
1. Neural networks
2. Tensors
3. *Computation graphs*
    1. What is the semantics of a computation graph?
    2. How do perform computations with them
4. Loss functions and training
5. Design patterns

Neural networks are differentiable computation units that operate on and produce tensors

Nodes represent values

Expression $\mathbf{x}$

Graph



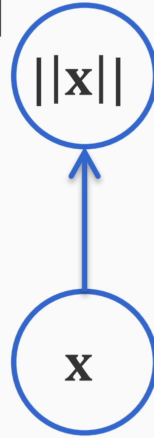
The value is implicitly or explicitly typed.

It could represent a

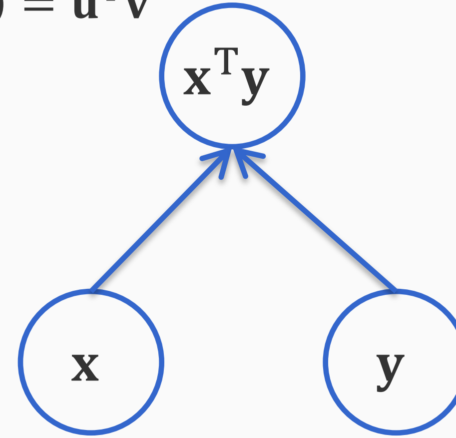
- Scalar (i.e. a number)
- A vector
- A matrix
- Or more generally, a tensor

Edges represent function arguments

$\mathrm{f}(\mathbf{u}) = ||\mathbf{u}||$

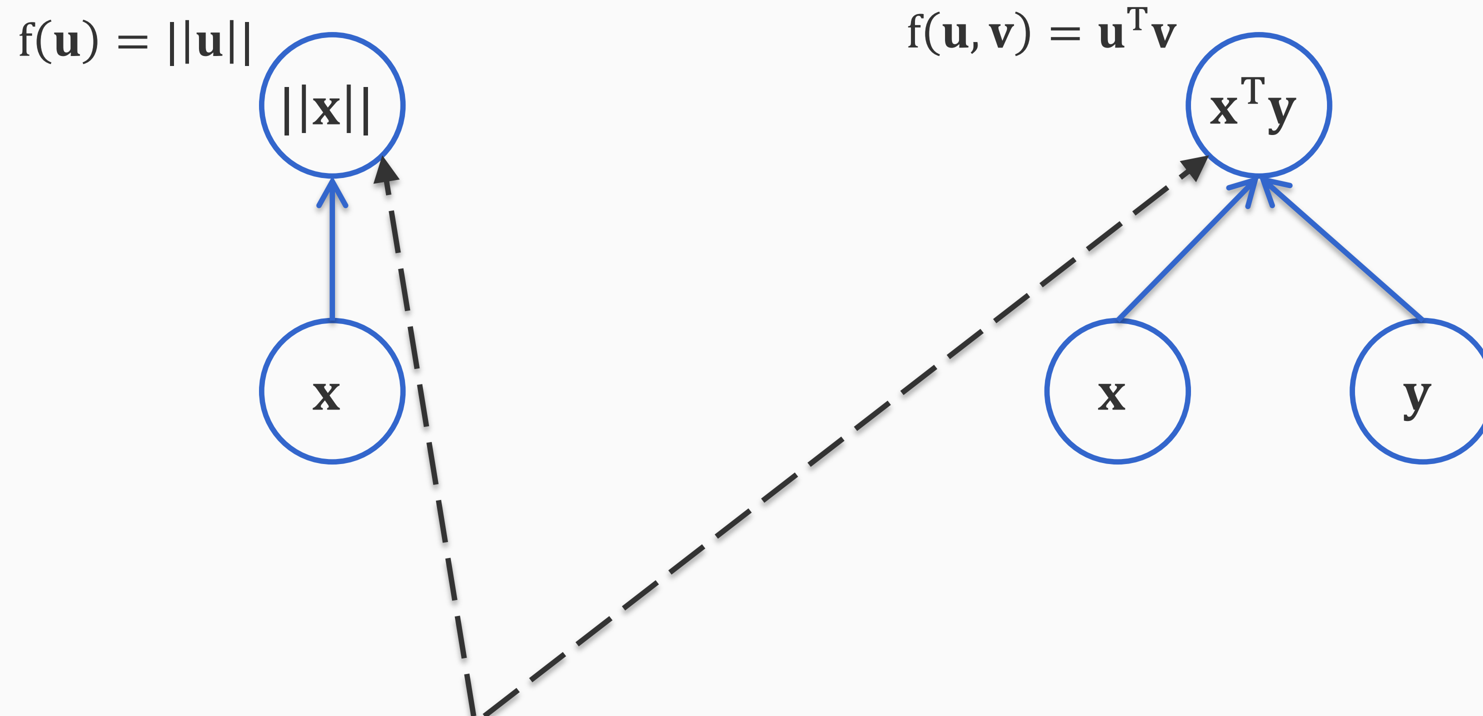


$\mathrm{f}(\mathbf{u}, \mathbf{v}) = \mathbf{u}^{\mathrm{T}}\mathbf{v}$

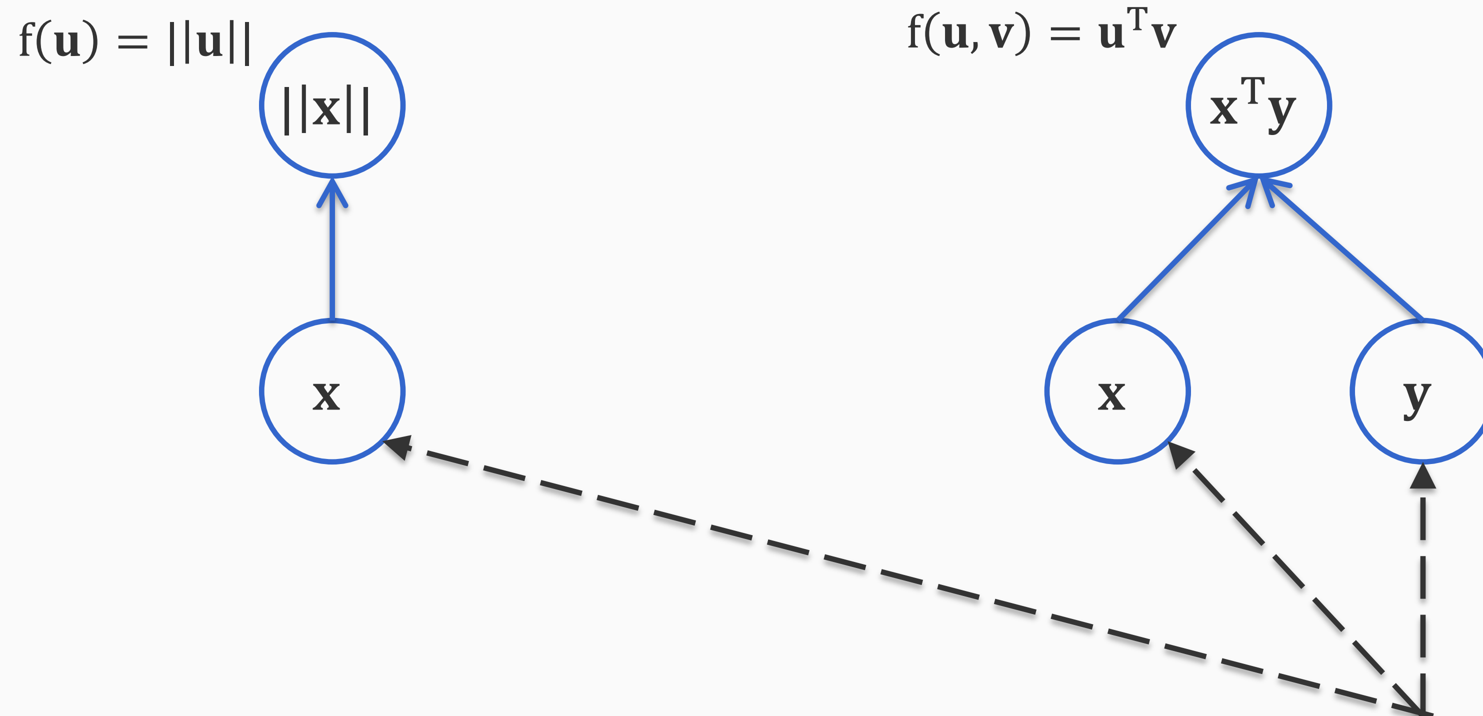


A node with an incoming edge is a function of the the parent node

Edges represent function arguments

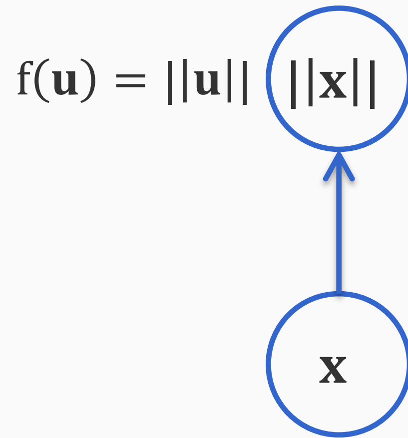


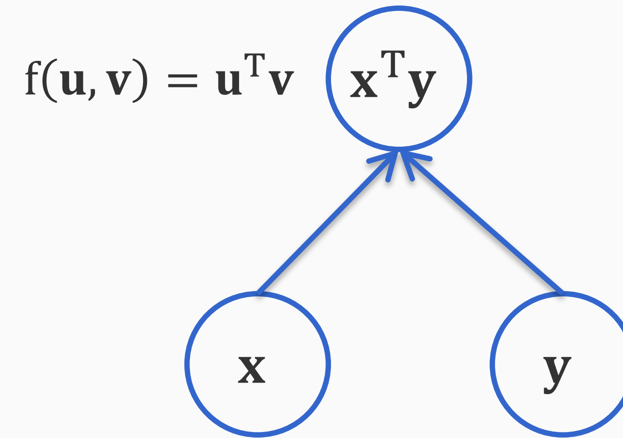
Edges represent function arguments



A node with an incoming edge is a function of the the parent node

Edges represent function arguments





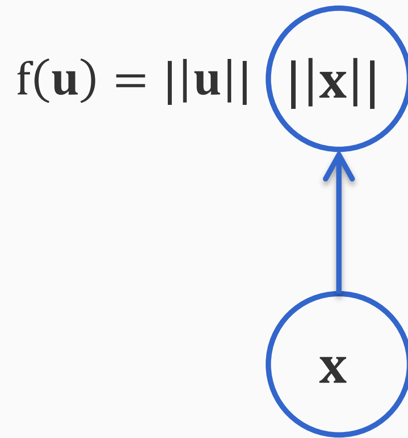
Each node knows how to compute two things:

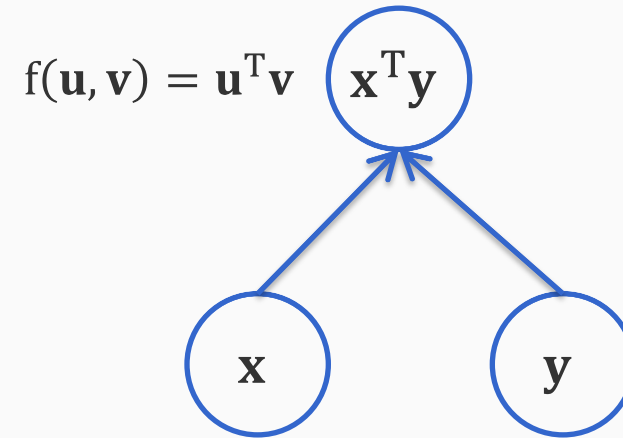
1. Its own value using its inputs
   - In these examples, the nodes on top compute $||\mathbf{x}||$ and $\mathbf{x}^{\mathrm{T}}\mathbf{y}$

Notation: We will write down what that function is next to the node.

When we write this, we will use formal arguments (here, the $\mathbf{u}$ and $\mathbf{v}$). Think of these as similar to the argument names we use when we declare functions while programming.

Edges represent function arguments

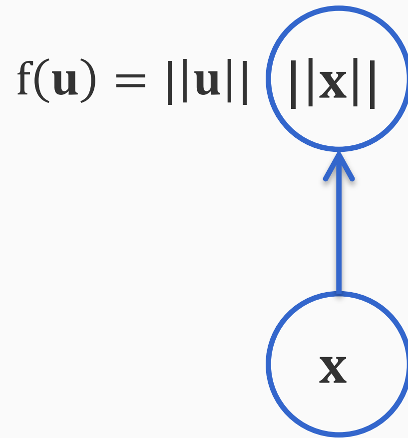


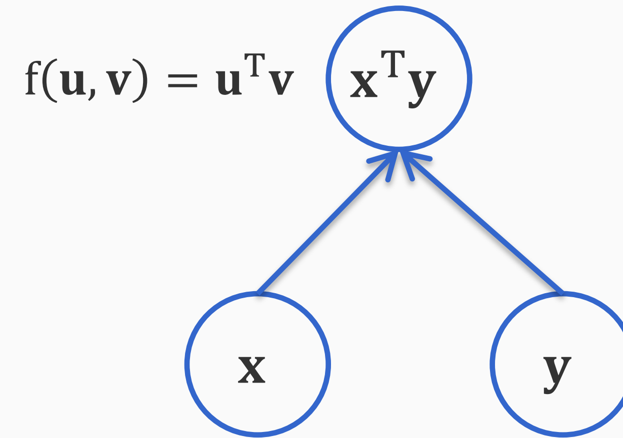


Each node knows how to compute two things:

1. Its own value using its inputs
   - In these examples, the nodes on top compute $||\mathbf{x}||$ and $\mathbf{x}^{\mathrm{T}}\mathbf{y}$
2. The value of its partial derivative with respect to each input
   - Left graph: the node on top knows to compute $\frac{\partial f}{\partial \mathbf{u}}$
   - Right graph: the node on top knows to compute $\frac{\partial f}{\partial \mathbf{u}}$ and $\frac{\partial f}{\partial \mathbf{v}}$

Graphs represent functions





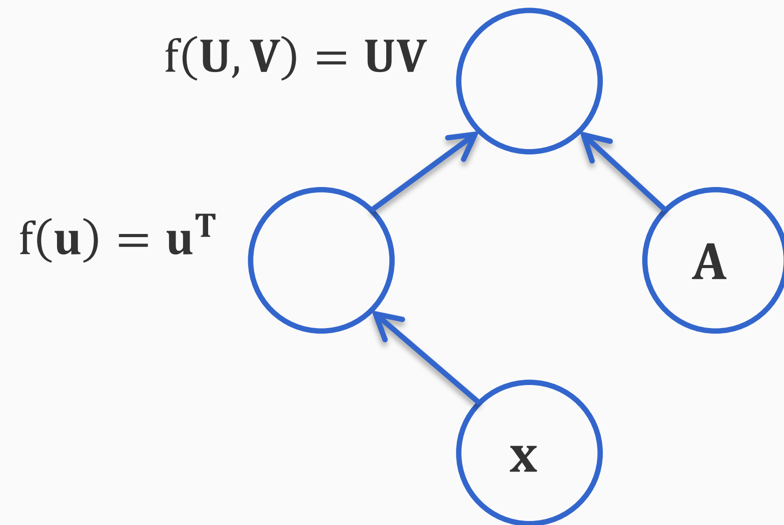
The functions expressed could be

- Nullary, i.e. with no arguments: if a node has no incoming edges
- Unary: if a node has one incoming edge
- Binary: if a node has two incoming edges
- …
- n-ary: if a node has n incoming edges

Let's see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}$

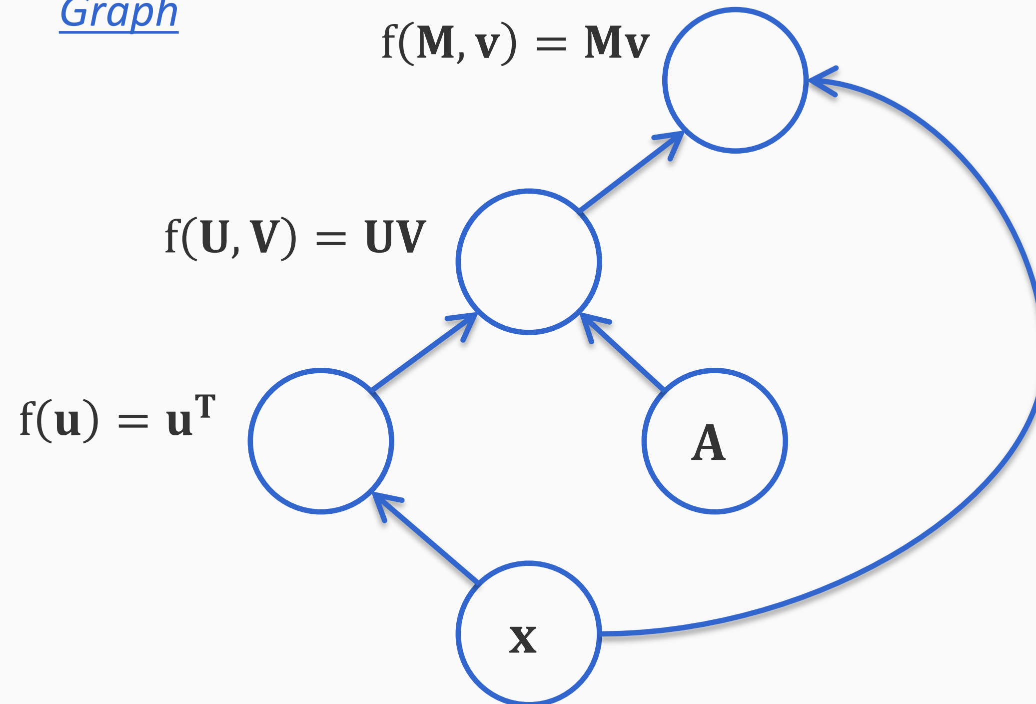
Graph



Let's see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

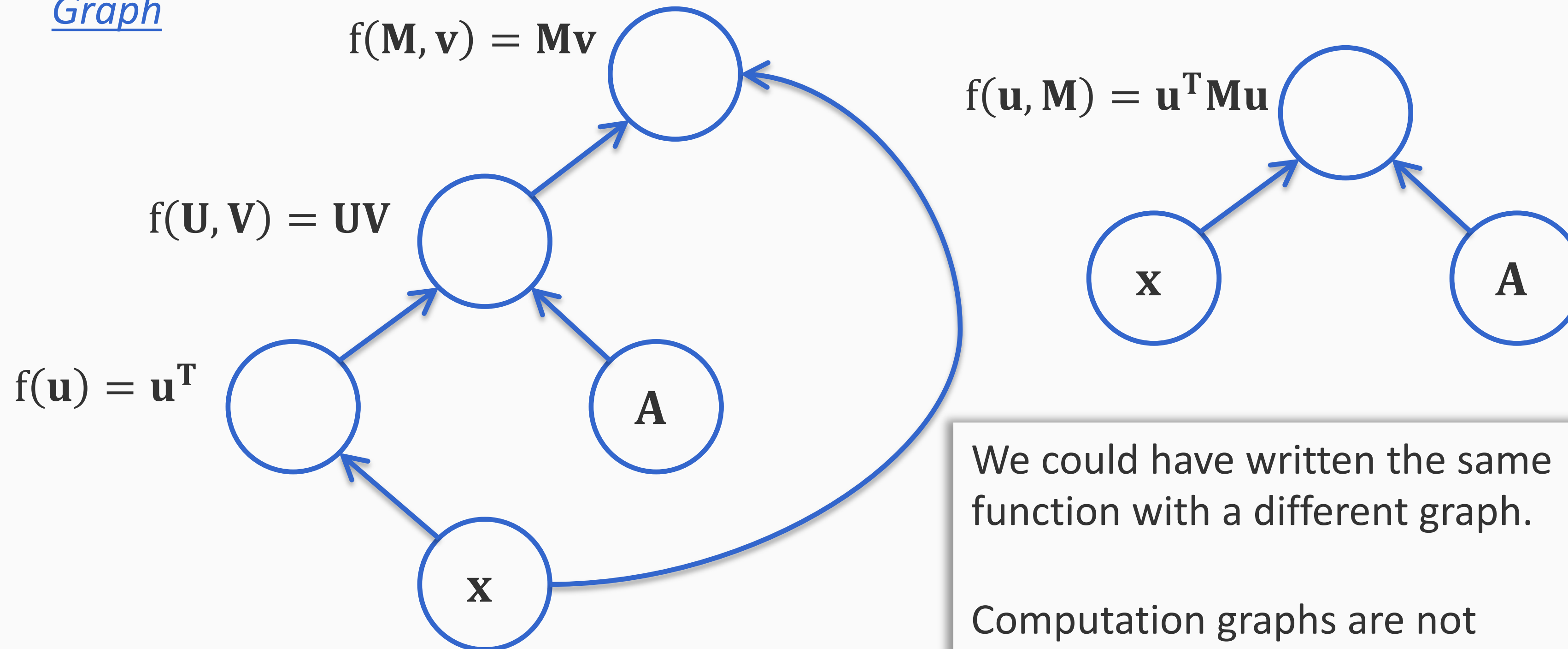
Graph



Let’s see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

Graph



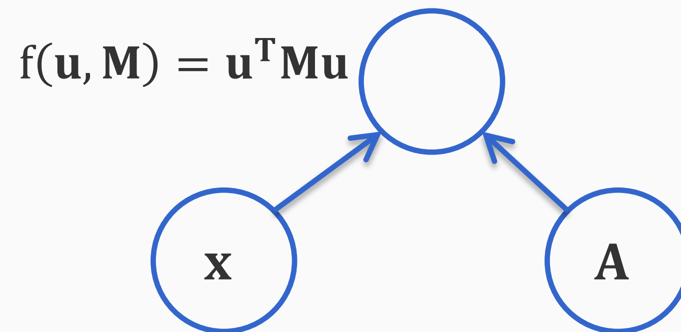
We could have written the same function with a different graph.

Computation graphs are not necessarily unique for a function

Let's see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

Graph



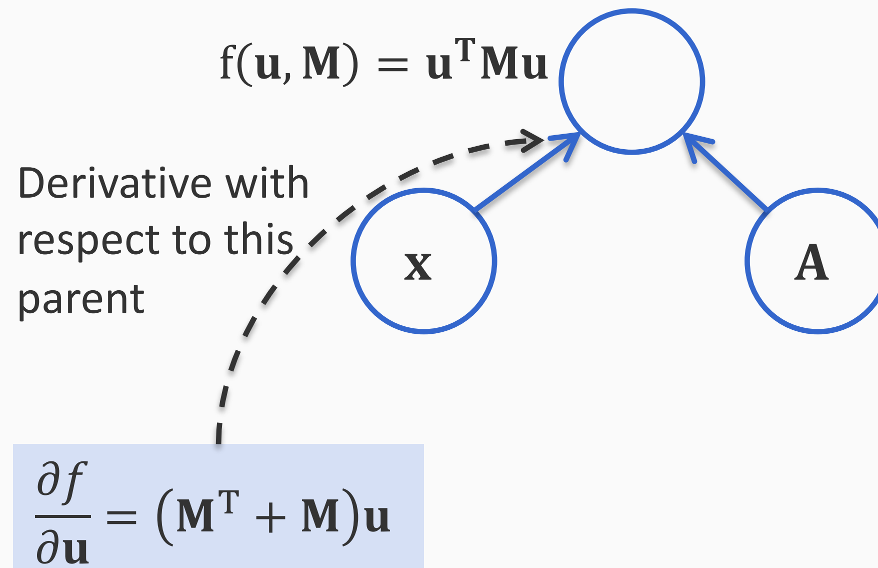
Remember: The nodes also know how to compute derivatives with respect to each parent

Let’s see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

Remember: The nodes also know how to compute derivatives with respect to each parent

Graph

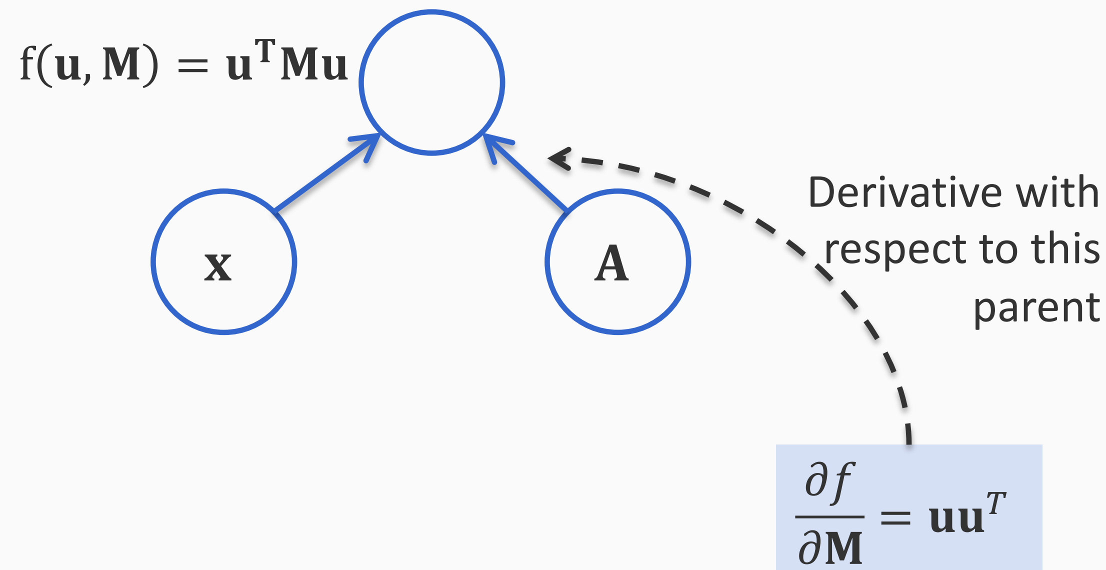


Let's see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

Graph

Remember: The nodes also know how to compute derivatives with respect to each parent

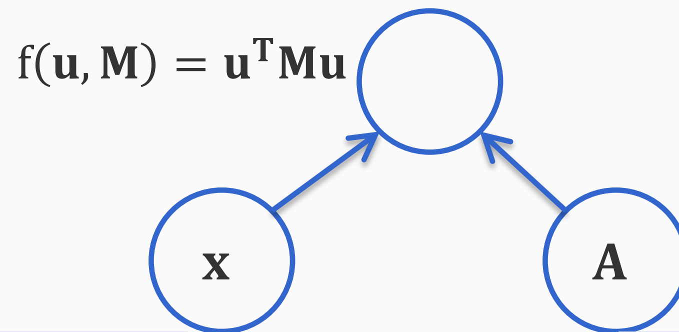


Let's see some functions as graphs

Expression $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$

$$\frac{\partial f}{\partial \mathbf{x}} = (\mathbf{A}^{\mathrm{T}} + \mathbf{A})\mathbf{x} \qquad \frac{\partial f}{\partial \mathbf{A}} = \mathbf{x}\mathbf{x}^{T}$$

Graph



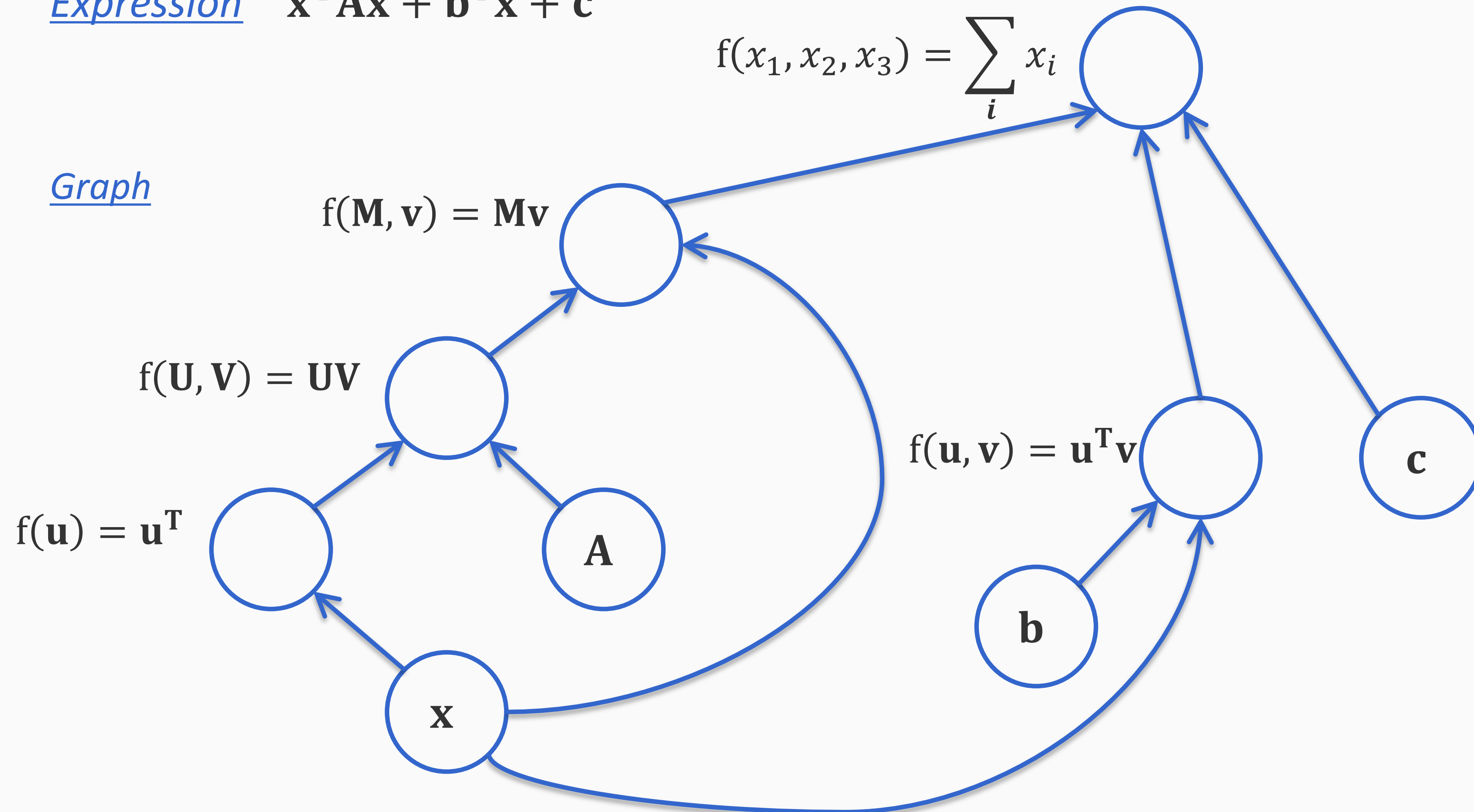
$$\frac{\partial f}{\partial \mathbf{u}} = (\mathbf{M}^{\mathrm{T}} + \mathbf{M})\mathbf{u}$$

$$\frac{\partial f}{\partial \mathbf{M}} = \mathbf{u}\mathbf{u}^{T}$$

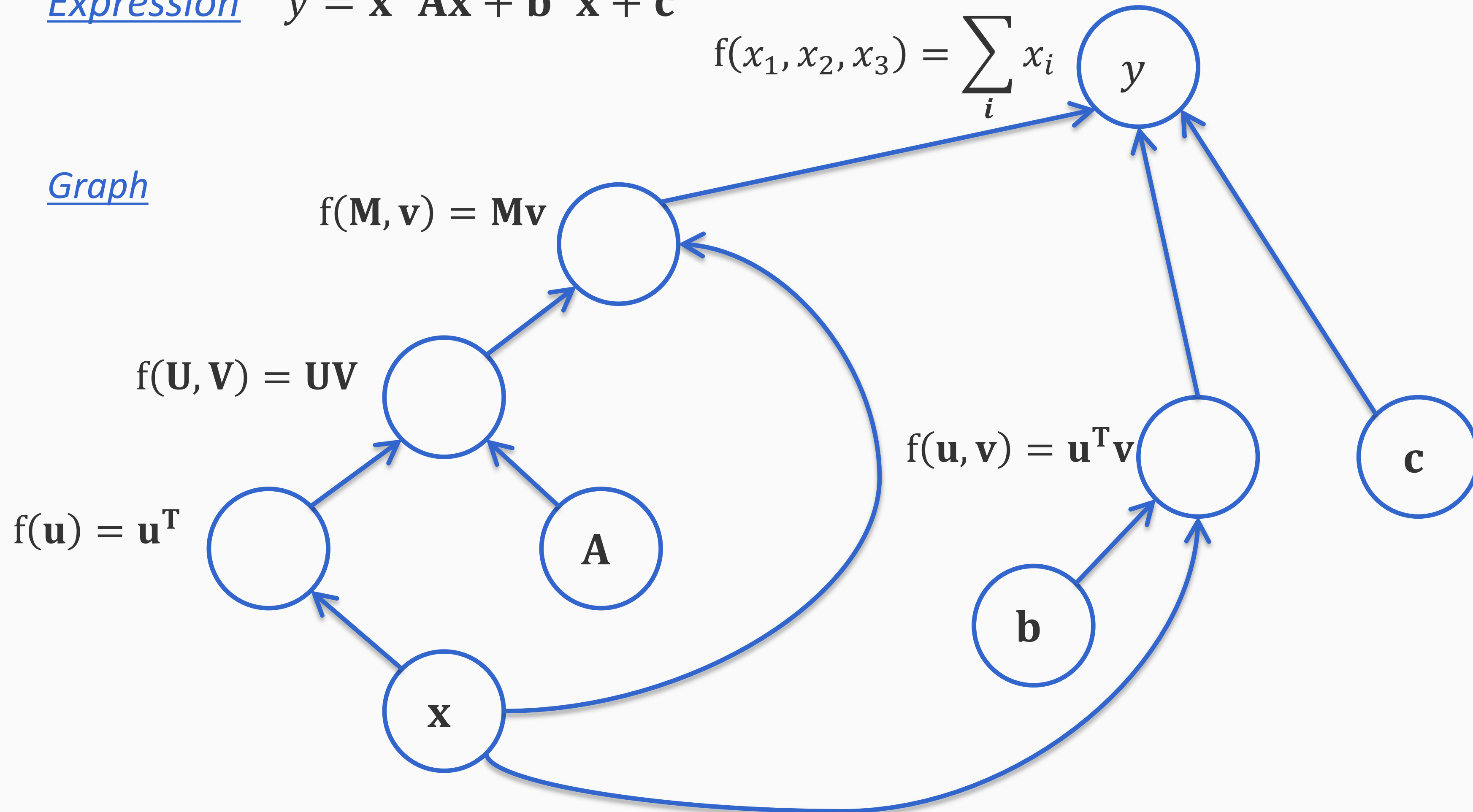
Remember: The nodes also know how to compute derivatives with respect to each parent

Together, we can compute derivatives of any function with respect to all its inputs, for any value of the input

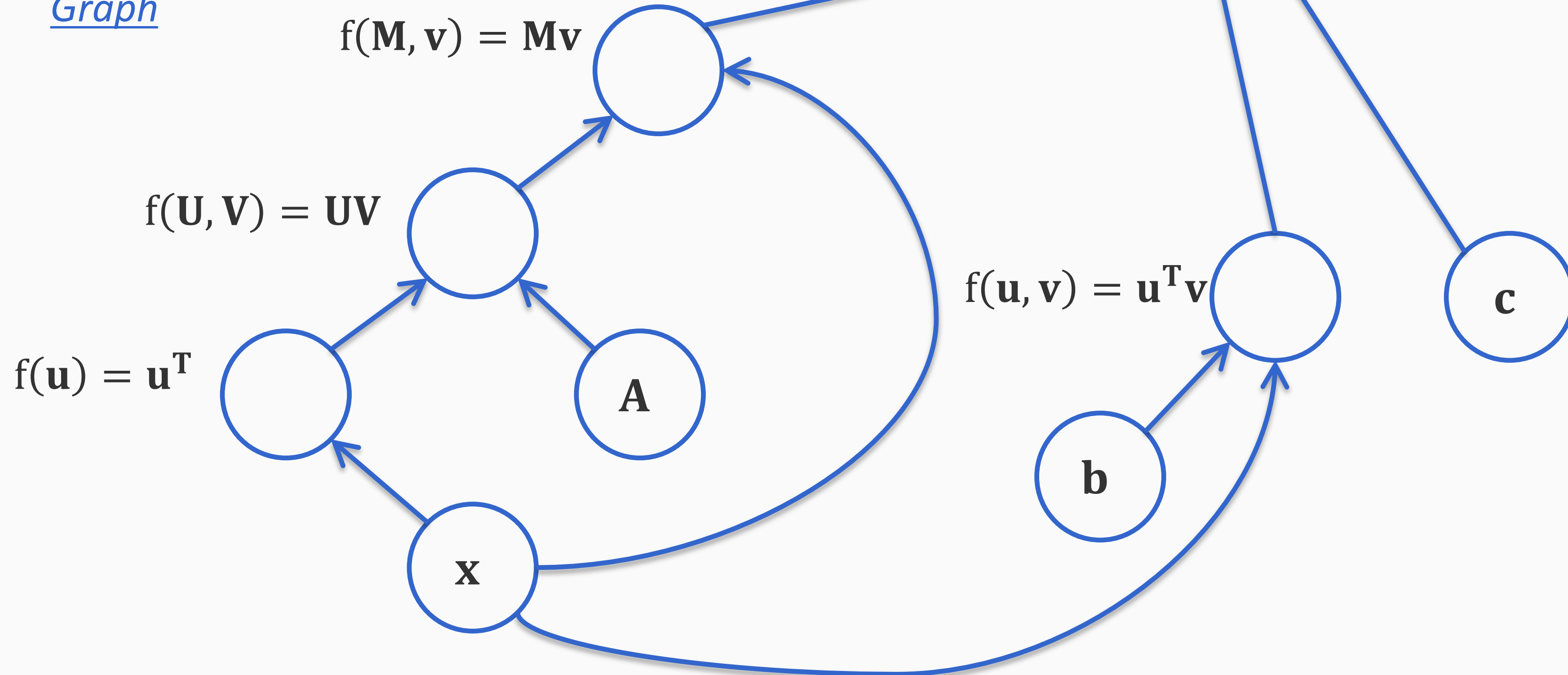
Let's see some functions as graphs



Let's see some functions as graphs



Let's see some functions as graphs



This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. *Computation graphs*
    1. What is the semantics of a computation graph?
    2. How do perform computations with them
4. Loss functions and training
5. Design patterns

Neural networks are differentiable computation units that operate on and produce tensors

Two algorithmic questions

1. Forward propagation
   - Given inputs to the graph, compute the value of the function expressed by the graph
   - Something to think about: Given a node, can we say which nodes are inputs? Which nodes are outputs?

2. Backpropagation
   - After computing the function value for an input, compute the gradient of the function at that input
   - Or equivalently: *How does the output change if I make a small change to the input?*

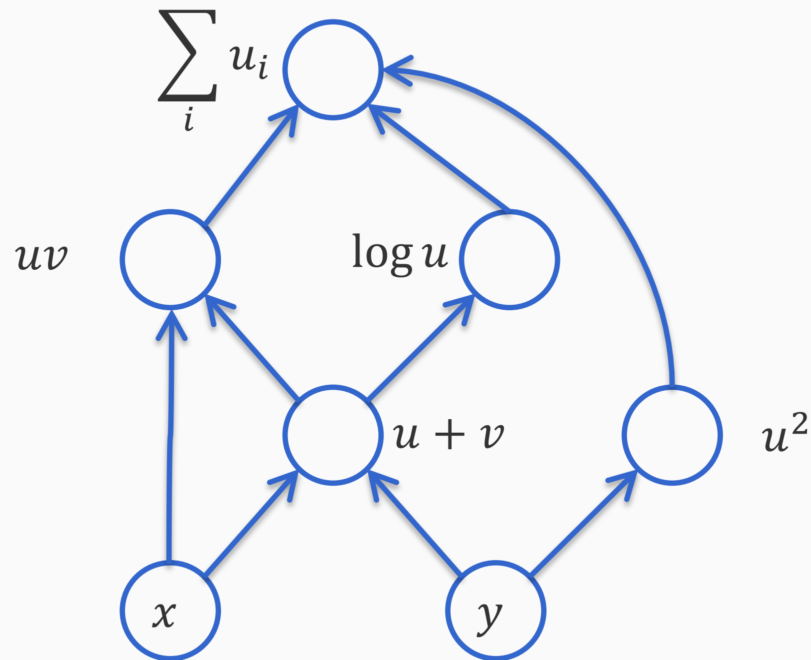
Forward propagation

Two algorithmic questions

1. Forward propagation
   - Given inputs to the graph, compute the value of the function expressed by the graph
   - Something to think about: Given a node, can we say which nodes are inputs? Which nodes are outputs?

2. Backpropagation
   - After computing the function value for an input, compute the gradient of the function at that input
   - Or equivalently: *How does the output change if I make a small change to the input?*

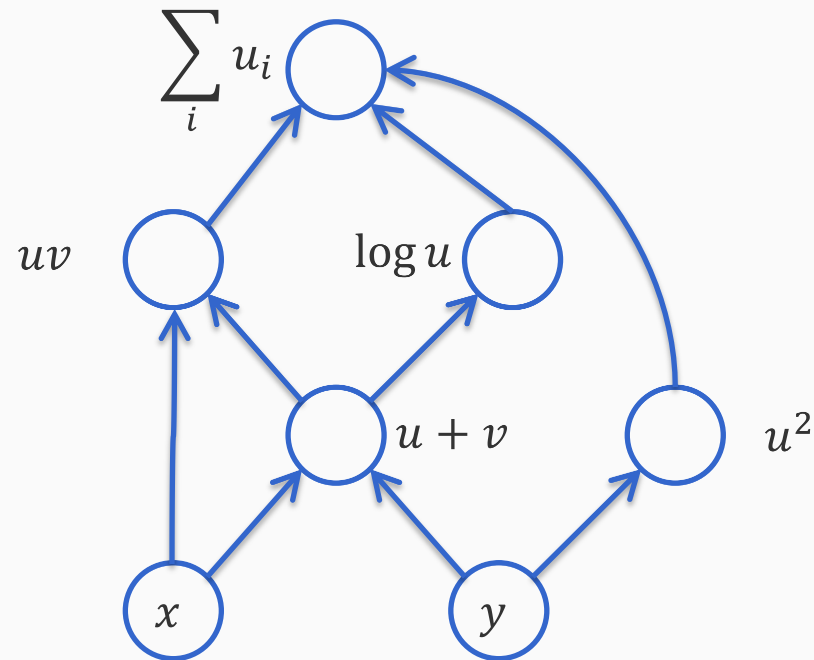
Forward pass: An example



Conventions:

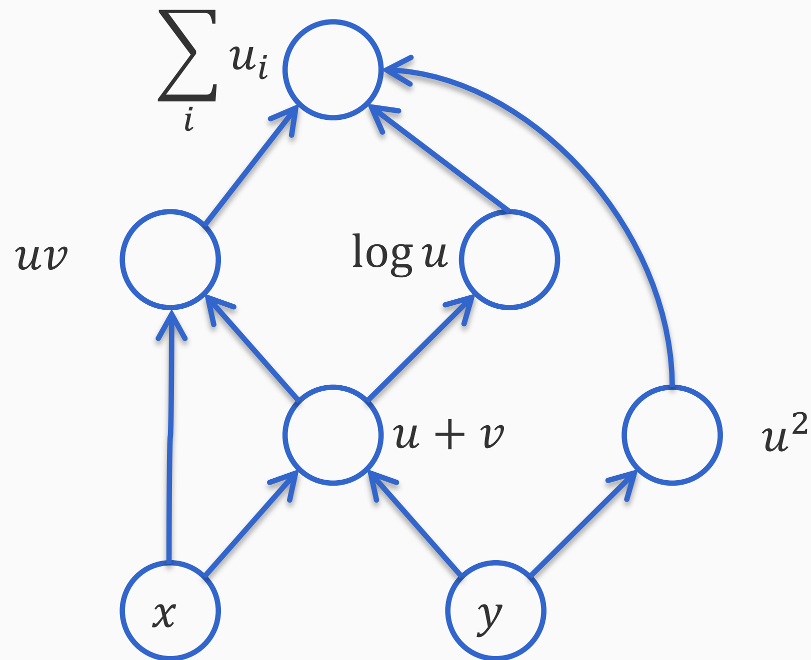
1. Any expression next to a node is the function it computes
2. All the variables in the expression are inputs to the node from left to right.

Forward pass



What function does this compute?

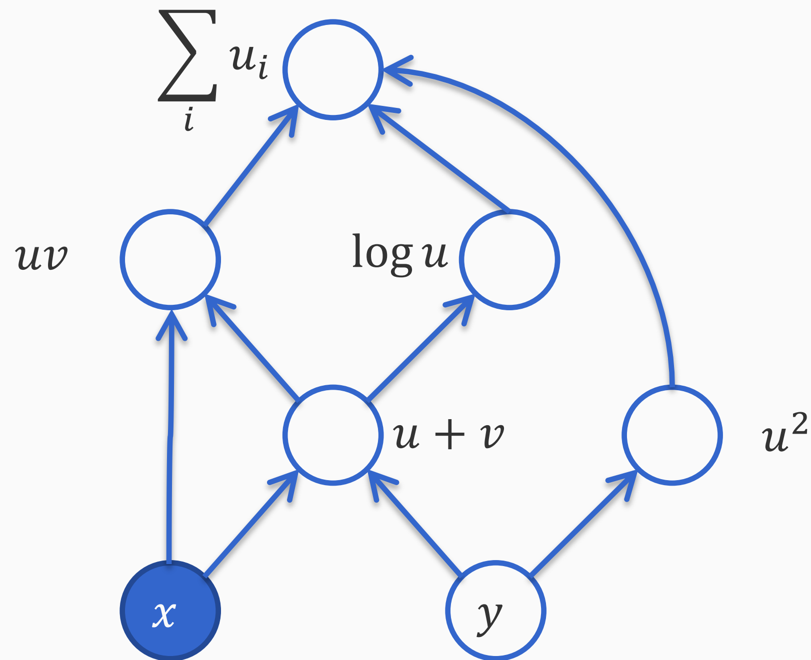
Forward pass



What function does this compute?

Suppose we shade nodes whose values we know (i.e. we have computed).

Forward pass

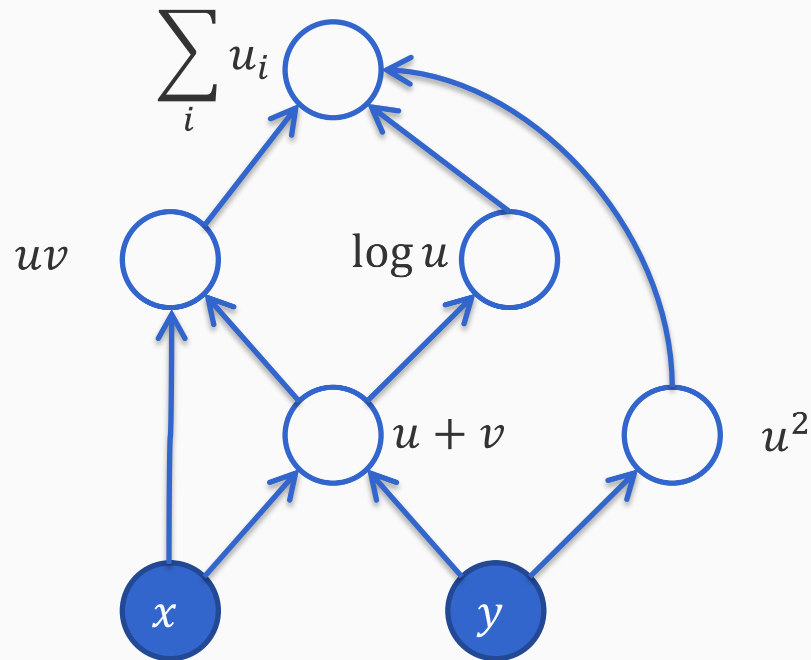


What function does this compute?

x

Suppose we shade nodes whose values we know (i.e. we have computed).

Forward pass



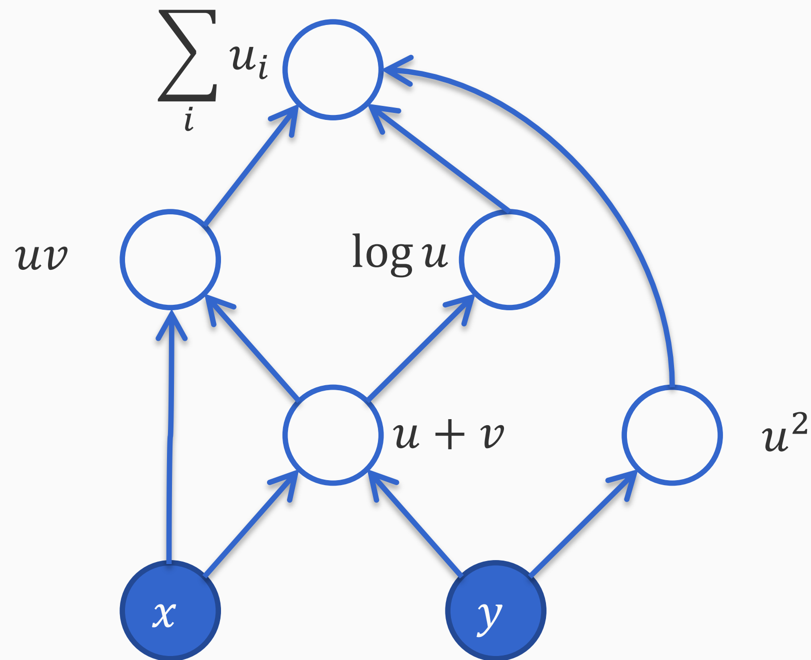
What function does this compute?

x

y

Suppose we shade nodes whose values we know (i.e. we have computed).

Forward pass



What function does this compute?

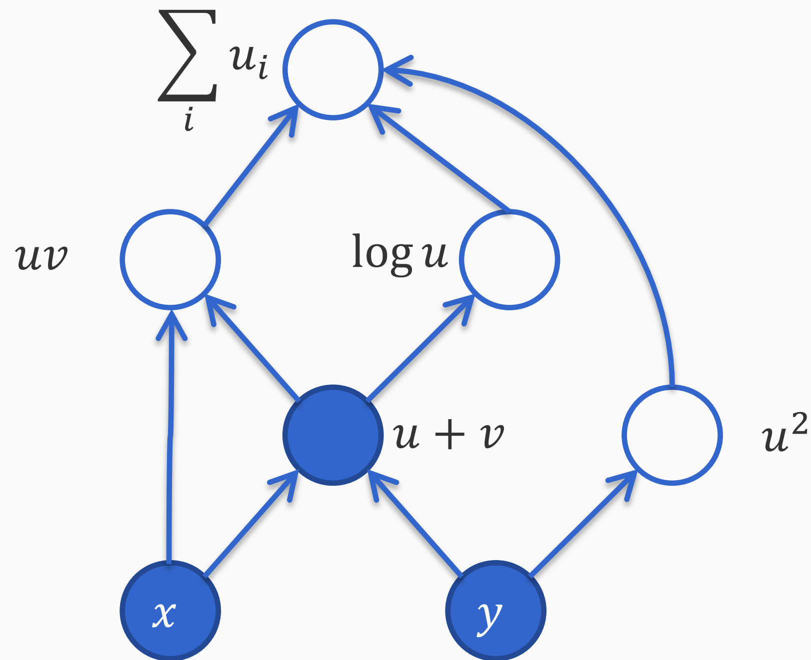
x

y

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

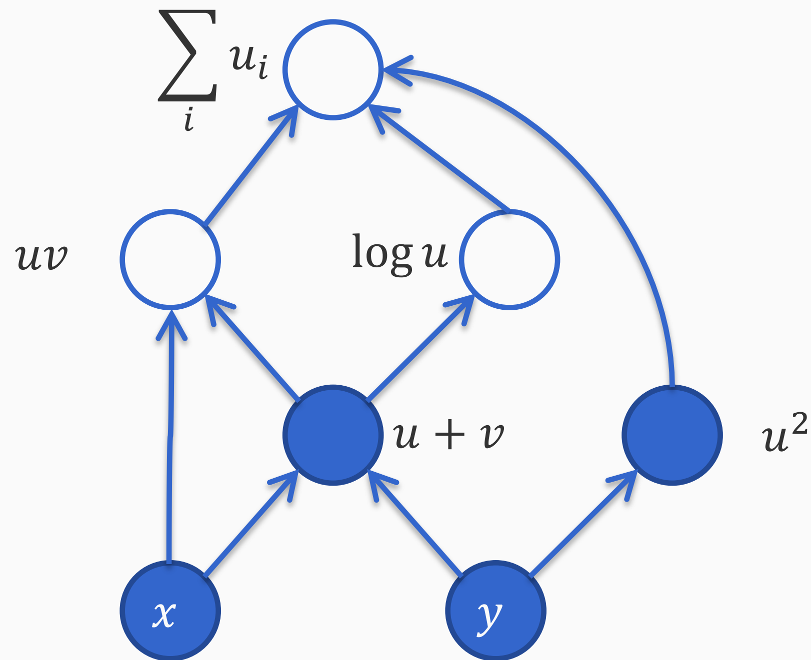
y

$x + y$

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

y

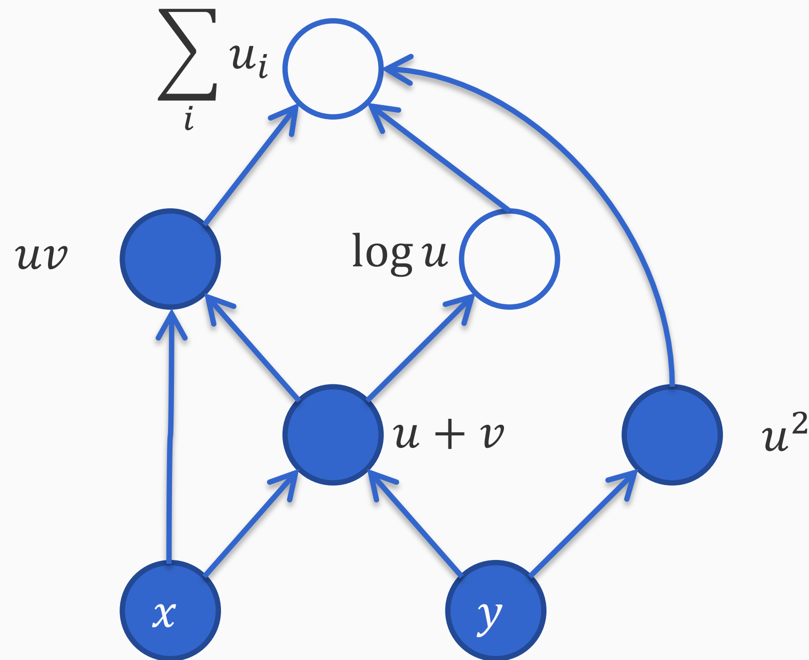
$x + y$

y^2

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

y

$x+y$

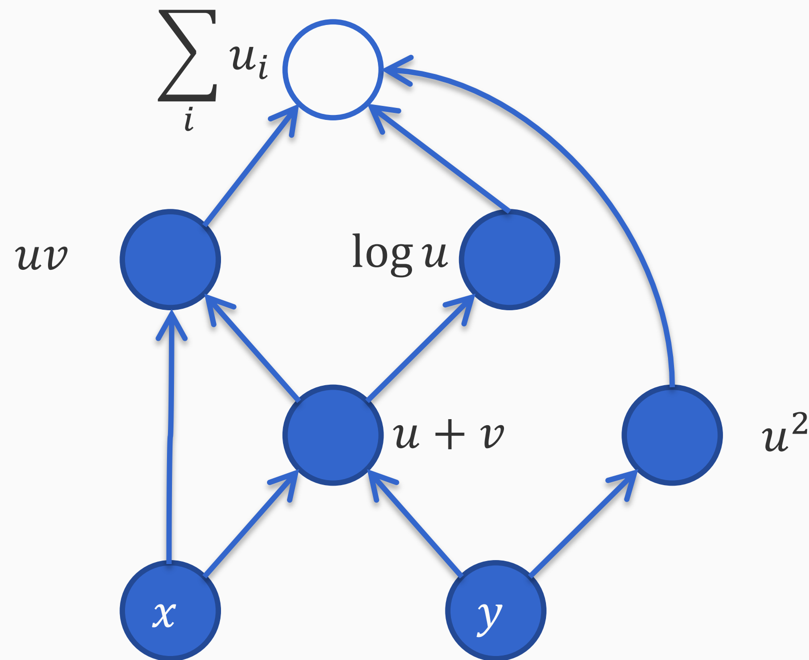
y^2

$x(x+y)$

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

y

$x + y$

y^2

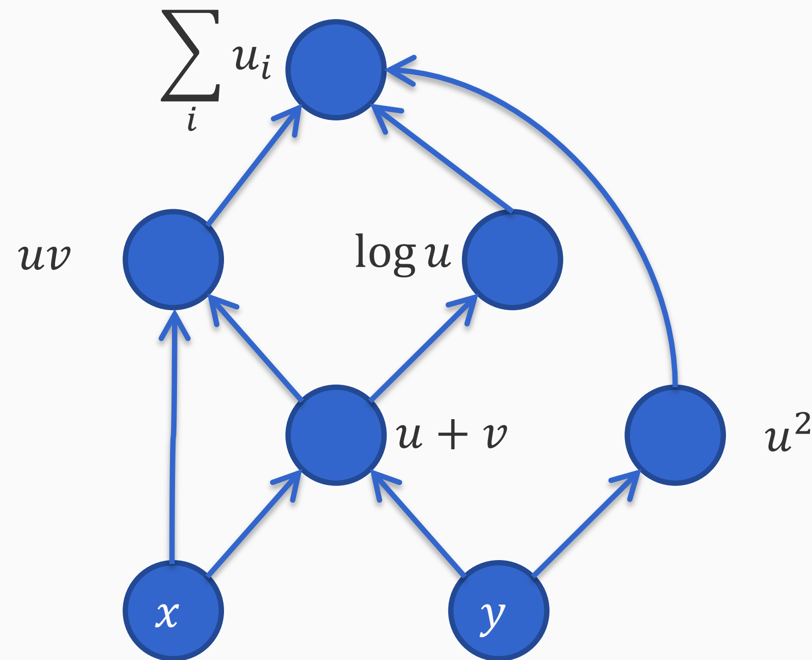
$x(x + y)$

$\log(x + y)$

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

y

$x + y$

y^2

$x(x + y)$

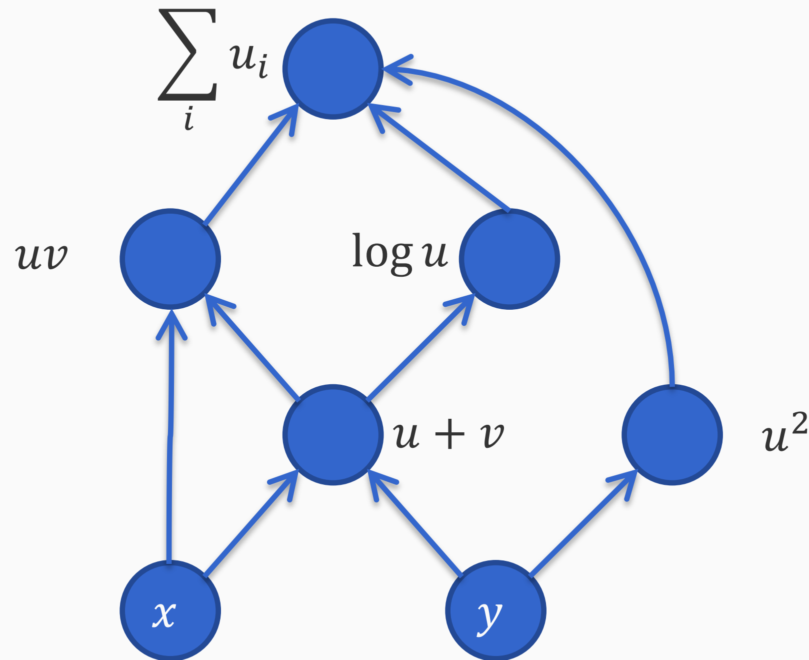
$\log(x + y)$

$\mathrm{x}(\mathrm{x} + \mathrm{y}) + \log(x + y) + y^2$

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward pass



What function does this compute?

x

y

$x + y$

y^2

$x(x + y)$

$\log(x + y)$

$\mathrm{x}(\mathrm{x} + \mathrm{y}) + \log(x + y) + y^2$

This gives us the function

Suppose we shade nodes whose values we know (i.e. we have computed).

We can only compute the value of a node if we know the values of all its inputs

Forward propagation

Given a computation graph G and values of its input nodes:

For each node in the graph, in topological order:

- Compute the value of that node

Forward propagation

Given a computation graph G and values of its input nodes:

For each node in the graph, in topological order:

Compute the value of that node

Why topological order: Ensures that children are computed before parents.

Two algorithmic questions

1. Forward propagation
   - Given inputs to the graph, compute the value of the function expressed by the graph
   - Something to think about: Given a node, can we say which nodes are inputs? Which nodes are outputs?

2. Backpropagation
   - After computing the function value for an input, compute the gradient of the function at that input
   - Or equivalently: *How does the output change if I make a small change to the input?*

Calculus refresher: The chain rule

Suppose we have two functions f and g

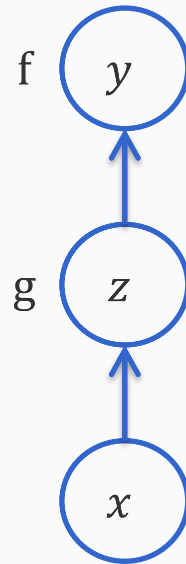
We wish to compute the gradient of $\mathrm{y} = f\left(g(x)\right)$.

We know that $\frac{dy}{dx} = f'\left(g(x)\right) \cdot g'(x)$

Or equivalently: if $z = g(x)$ and $y = f(z)$, then

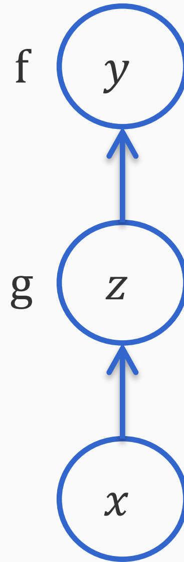
$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

Or equivalently: In terms of computation graphs



The forward pass gives us z and y

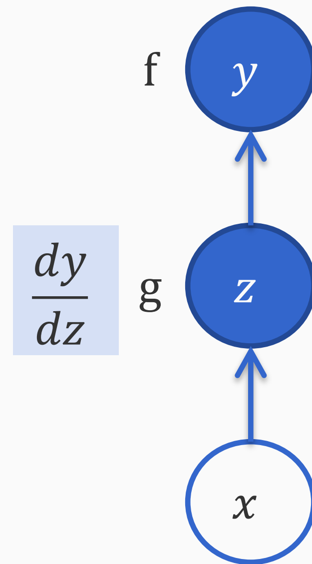
Or equivalently: In terms of computation graphs



The forward pass gives us z and y

Remember that each node knows not only how to compute its value given inputs, but also how to compute gradients

Or equivalently: In terms of computation graphs

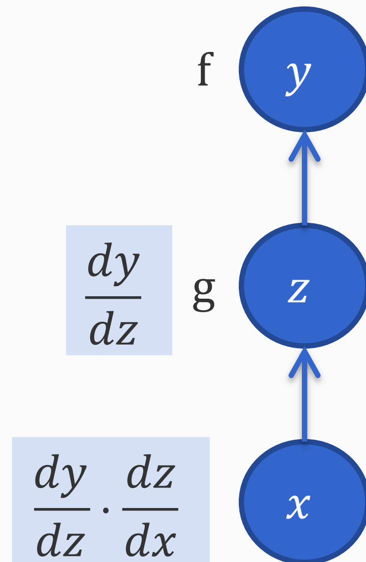


The forward pass gives us z and y

Remember that each node knows not only how to compute its value given inputs, but also how to compute gradients

Start from the root of the graph and work backwards.

Or equivalently: In terms of computation graphs



The forward pass gives us z and y

Remember that each node knows not only how to compute its value given inputs, but also how to compute gradients

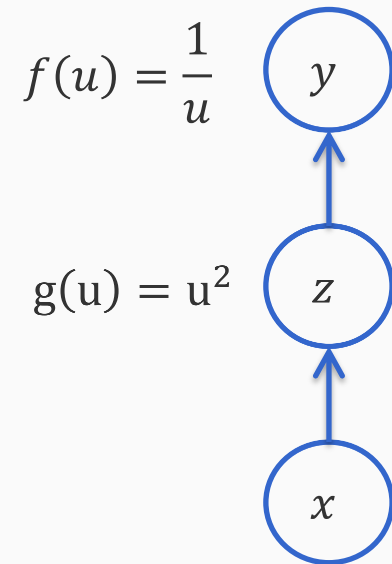
Start from the root of the graph and work backwards.

When traversing an edge backwards to a new node:

> the gradient of the root with respect to that node is the product of the gradient at the parent with the derivative along that edge

A concrete example

$$y = \frac{1}{x^2}$$

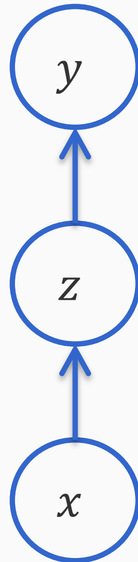


A concrete example

$$y = \frac{1}{x^2}$$

$$\frac{df}{du} = -\frac{1}{u^2} \qquad f(u) = \frac{1}{u}$$

$$\frac{dg}{du} = 2u \qquad g(u) = u^2$$

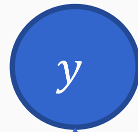


Let’s also explicitly write down the derivatives.

A concrete example

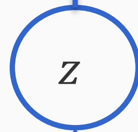
$$y = \frac{1}{x^2}$$

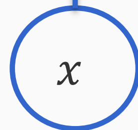
$$\frac{df}{du} = -\frac{1}{u^2} \qquad f(u) = \frac{1}{u}$$



$$\frac{dy}{dy} = 1$$

$$\frac{dg}{du} = 2u \qquad g(u) = u^2$$





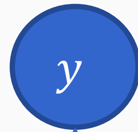
Now, we can proceed backwards from the output

At each step, we compute the gradient of the function represented by the graph with respect to the node that we are at.

A concrete example

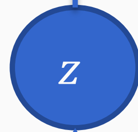
$$y = \frac{1}{x^2}$$

$$\frac{df}{du} = -\frac{1}{u^2} \qquad f(u) = \frac{1}{u}$$

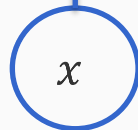


$$\frac{dy}{dy} = 1$$

$$\frac{dg}{du} = 2u \qquad \mathrm{g(u)} = \mathrm{u}^2$$



$$\frac{dy}{dz} = \frac{dy}{dy} \cdot \left(\frac{df}{du}\right)_{u=z} = 1 \cdot \left(-\frac{1}{z^2}\right) = -\frac{1}{z^2}$$

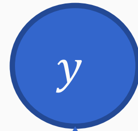


Product of the gradient so far and
the derivative computed at this step

A concrete example

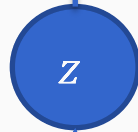
$$y = \frac{1}{x^2}$$

$$\frac{df}{du} = -\frac{1}{u^2} \qquad f(u) = \frac{1}{u}$$

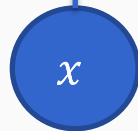


$$\frac{dy}{dy} = 1$$

$$\frac{dg}{du} = 2u \qquad \mathrm{g(u) = u^2}$$



$$\frac{dy}{dz} = -\frac{1}{z^2}$$

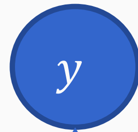


$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \left(\frac{dg}{du}\right)_{u=x} = -\frac{1}{z^2} \cdot 2x = -\frac{2\mathrm{x}}{\mathrm{z}^2}$$

A concrete example

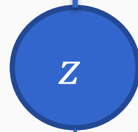
$$y = \frac{1}{x^2}$$

$$\frac{df}{du} = -\frac{1}{u^2} \qquad f(u) = \frac{1}{u}$$

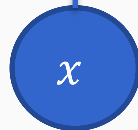


$$\frac{dy}{dy} = 1$$

$$\frac{dg}{du} = 2u \qquad \mathrm{g(u) = u^2}$$



$$\frac{dy}{dz} = -\frac{1}{z^2}$$



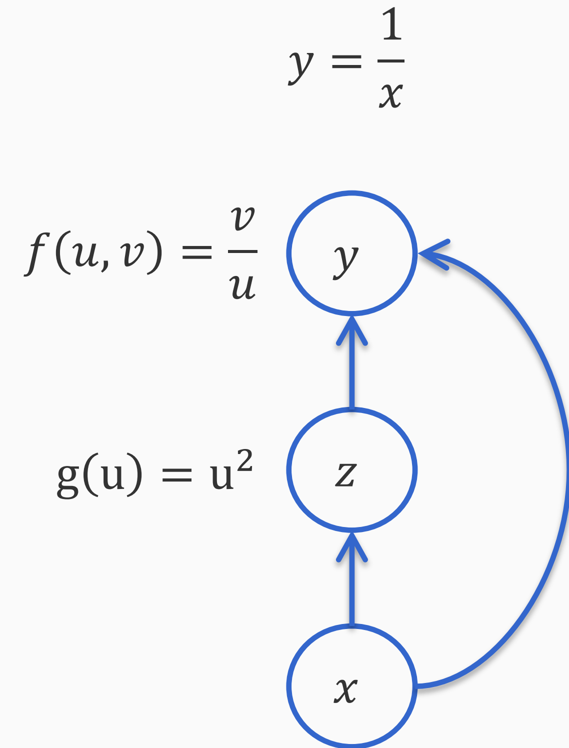
$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \left(\frac{dg}{du}\right)_{u=x} = -\frac{1}{z^2} \cdot 2x = -\frac{2\mathrm{x}}{\mathrm{z}^2}$$

We can simplify this to get $-\frac{2}{x^3}$

A concrete example

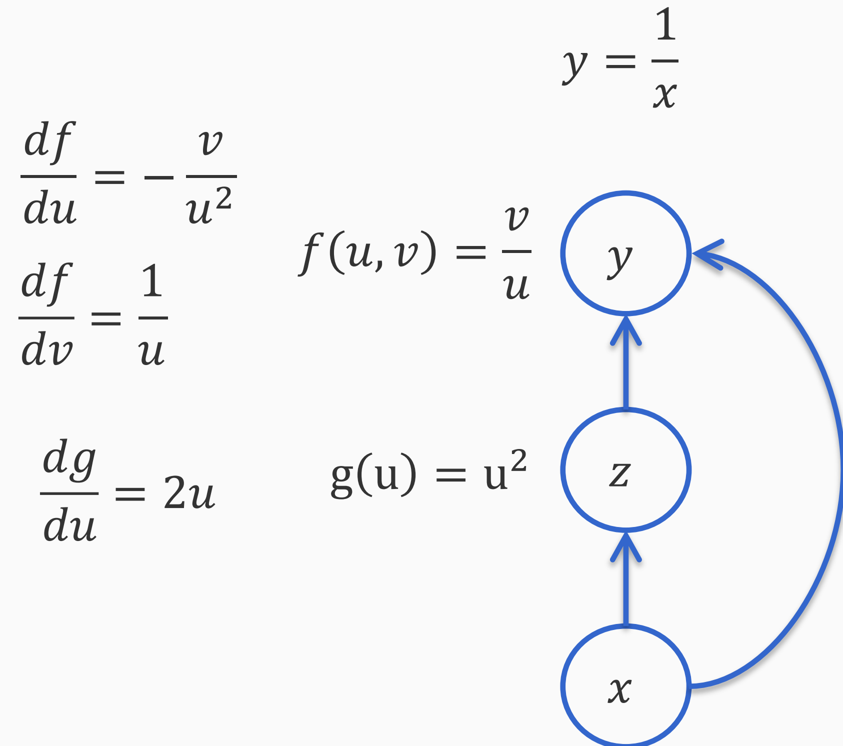
with multiple outgoing edges

$$y = \frac{1}{x}$$



A concrete example

with multiple outgoing edges



Let's also explicitly write down the derivatives. Note that f has two derivatives because it has two inputs.

A concrete example

with multiple outgoing edges

$$y = \frac{1}{x}$$

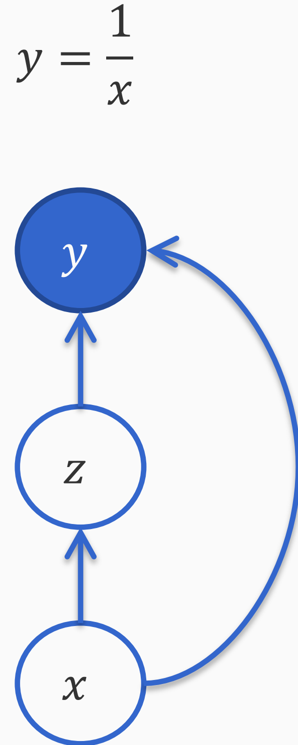
$$\frac{df}{du} = -\frac{v}{u^2}$$

$$\frac{df}{dv} = \frac{1}{u}$$

$$\frac{dg}{du} = 2u$$

$$f(u, v) = \frac{v}{u}$$

$$\mathrm{g(u) = u^2}$$



$$\frac{dy}{dy} = 1$$

A concrete example

with multiple outgoing edges

$$y = \frac{1}{x}$$

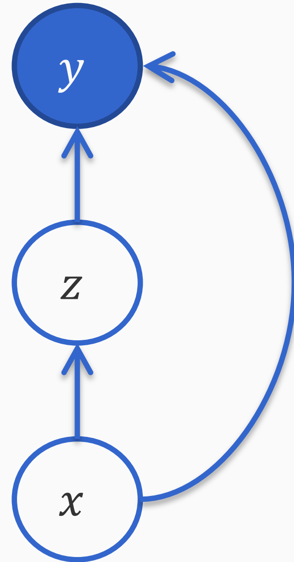
$$\frac{df}{du} = -\frac{v}{u^2}$$

$$\frac{df}{dv} = \frac{1}{u}$$

$$\frac{dg}{du} = 2u$$

$$f(u, v) = \frac{v}{u}$$

$$g(u) = u^2$$



$$\frac{dy}{dy} = 1$$

At this point, we can compute the gradient of y with respect to z by following the edge from y to z.

But we can not follow the edge from y to x because all of x's descendants are not marked as done.

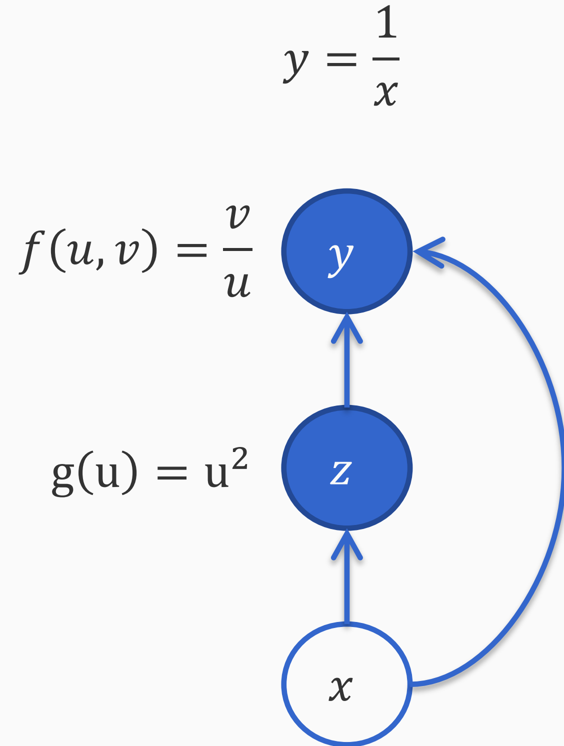
A concrete example

with multiple outgoing edges

$$\frac{df}{du} = -\frac{v}{u^2}$$

$$\frac{df}{dv} = \frac{1}{u}$$

$$\frac{dg}{du} = 2u$$



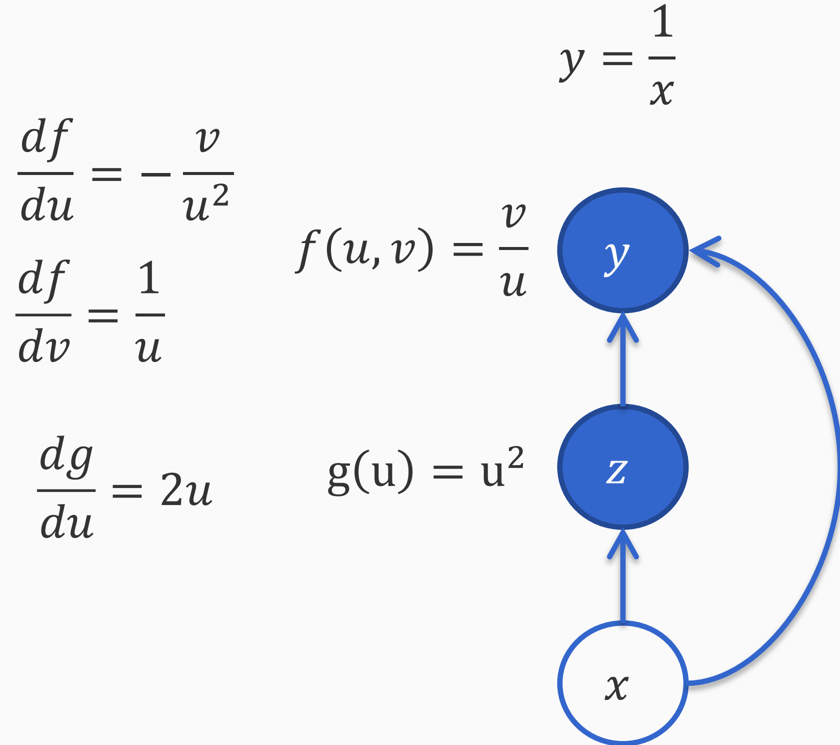
$$\frac{dy}{dy} = 1$$

$$\frac{dy}{dz} = \frac{dy}{dy} \cdot \left(\frac{df}{du}\right)_{u=z} = 1 \cdot \left(-\frac{x}{z^2}\right) = -\frac{x}{z^2}$$

Product of the gradient so far and the derivative computed at this step

A concrete example

with multiple outgoing edges



$$\frac{dy}{dy} = 1$$

$$\frac{dy}{dz} = \frac{dy}{dy} \cdot \left(\frac{df}{du}\right)_{u=z} = 1 \cdot \left(-\frac{x}{z^2}\right) = -\frac{x}{z^2}$$

Now we can get to x

There are multiple backward paths into x.
The general rule: Add the gradients along all the paths.

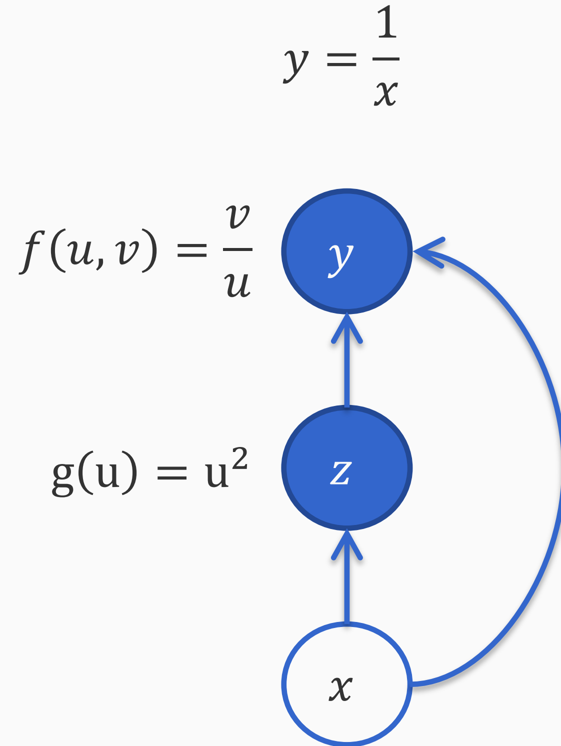
A concrete example

with multiple outgoing edges

$$\frac{df}{du} = -\frac{v}{u^2}$$

$$\frac{df}{dv} = \frac{1}{u}$$

$$\frac{dg}{du} = 2u$$



$$\frac{dy}{dy} = 1$$

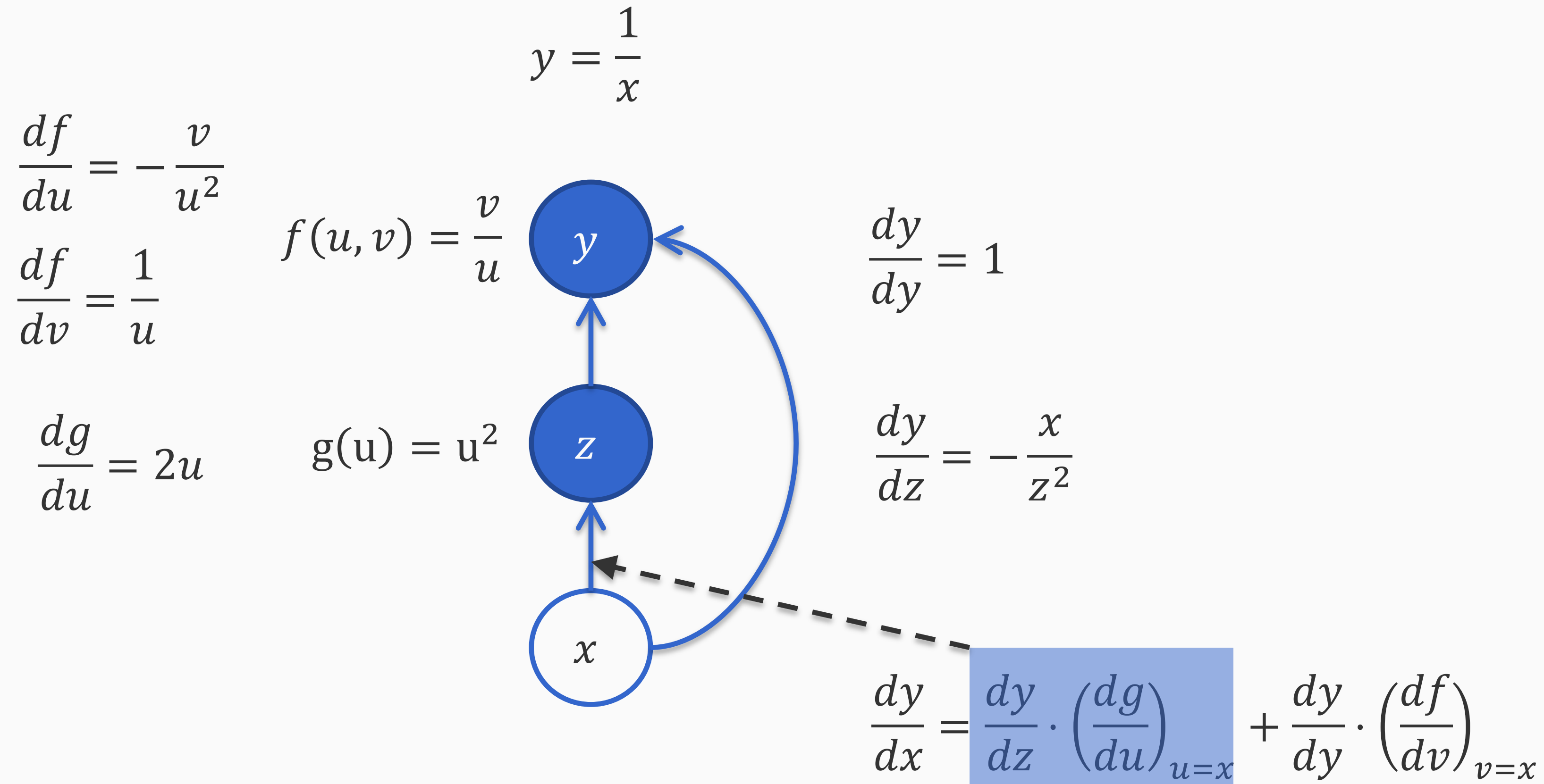
$$\frac{dy}{dz} = -\frac{x}{z^2}$$

$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \left(\frac{dg}{du}\right)_{u=x} + \frac{dy}{dy} \cdot \left(\frac{df}{dv}\right)_{v=x}$$

There are multiple backward paths into x.
The general rule: Add the gradients along all the paths.

A concrete example

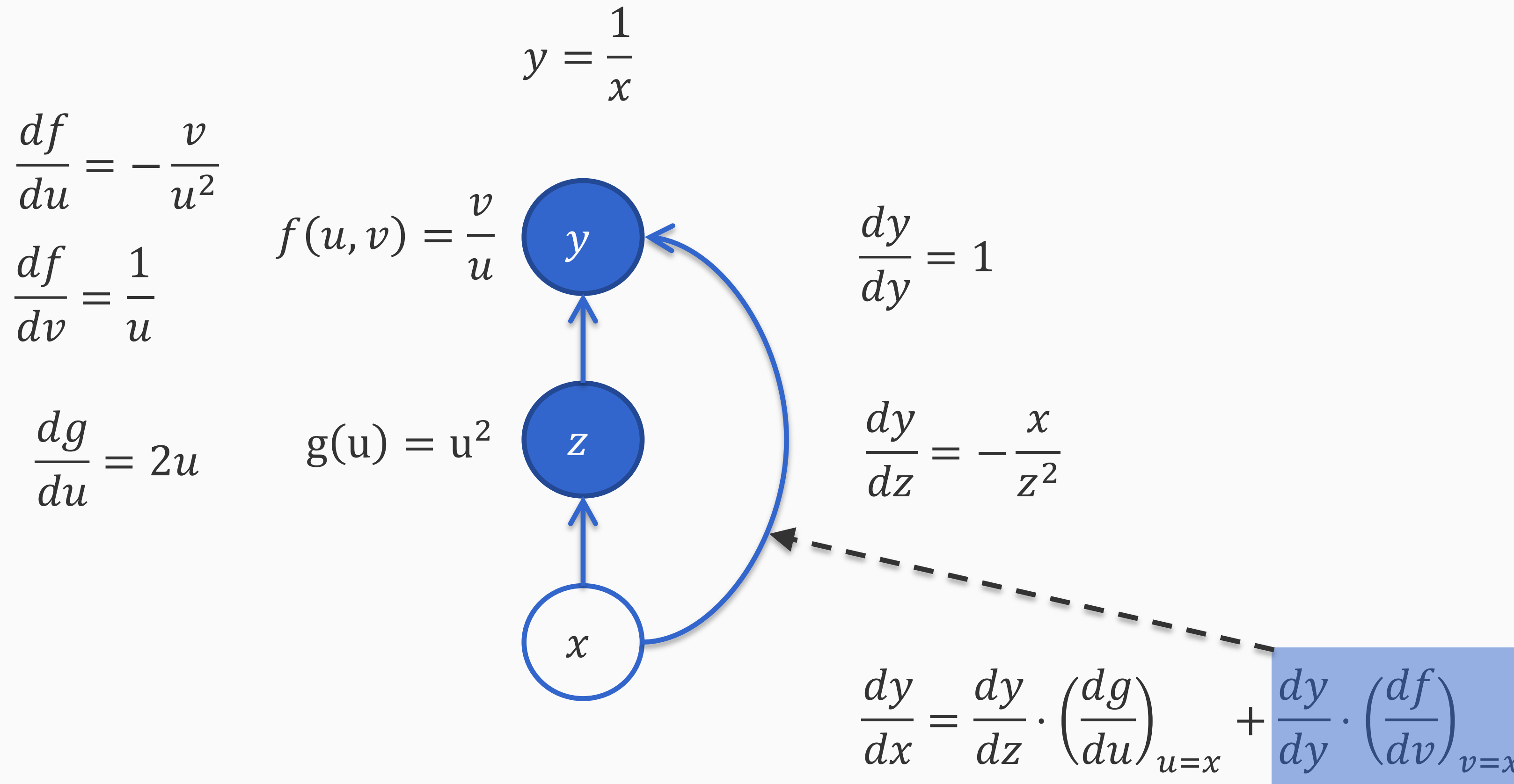
with multiple outgoing edges



There are multiple backward paths into x.
The general rule: Add the gradients along all the paths.

A concrete example

with multiple outgoing edges



There are multiple backward paths into x.
The general rule: Add the gradients along all the paths.

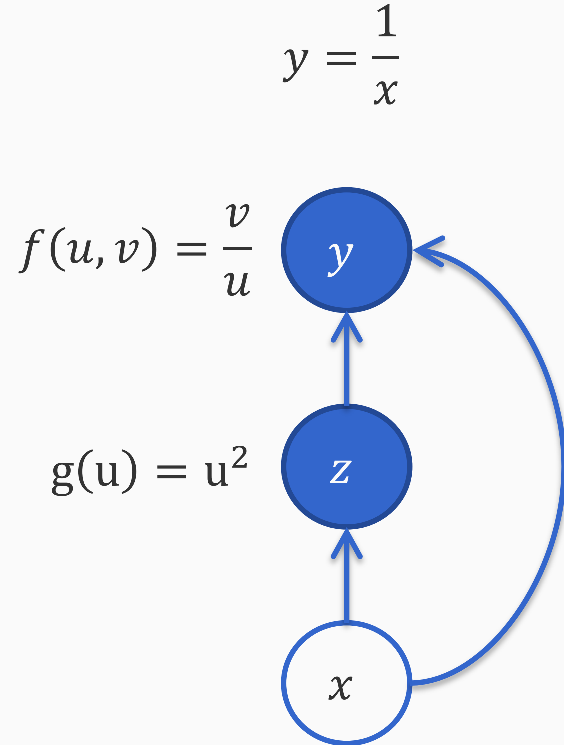
A concrete example

with multiple outgoing edges

$$\frac{df}{du} = -\frac{v}{u^2}$$

$$\frac{df}{dv} = \frac{1}{u}$$

$$\frac{dg}{du} = 2u$$



$$\frac{dy}{dy} = 1$$

$$\frac{dy}{dz} = -\frac{x}{z^2}$$

$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \left(\frac{dg}{du}\right)_{u=x} + \frac{dy}{dy} \cdot \left(\frac{df}{dv}\right)_{v=x}$$

$$\frac{dy}{dx} = -\frac{x}{z^2} \cdot 2x + 1 \cdot \frac{1}{z} = -\frac{2x^2}{z^2} + \frac{1}{z} = -\frac{1}{x^2}$$

Backpropagation, in general

After we have done the forward propagation,

Loop over the nodes in reverse topological order starting with a final goal node

- Compute derivatives of final goal node value with respect to each edge's tail node
  - If there are multiple outgoing edges from a node, sum up all the derivatives for the edges

This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. Computation graphs
4. Loss functions and training
5. Design patterns

The standard process of training neural networks

1. Design the graph that defines the computation you want

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters
   2. Define the loss for the network over the current example

      Characterizes the idea of “how bad is the result that was just computed”

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters
   2. Define the loss for the network over the current example

      Characterizes the idea of "how bad is the result that was just computed"
   3. Compute the gradient of the loss using backpropagation

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters
   2. Define the loss for the network over the current example

      Characterizes the idea of "how bad is the result that was just computed"
   3. Compute the gradient of the loss using backpropagation
   4. Update the parameters

Neural networks are data-driven programs

The forward pass allows us to compute the result of computations on examples

The backward pass over loss functions allows us to compute the update to the parameters that produced the loss

Both loss functions and neural networks are computation graphs

This abstraction allows us to think of neural networks as functions (in a programming sense) that will be "filled in" by data

This lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. Computation graphs
4. Loss functions and training
5. Design patterns

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters
   2. Define the loss for the network over the current example

      Characterizes the idea of "how bad is the result that was just computed"
   3. Compute the gradient of the loss using backpropagation
   4. Update the parameters

We have different design choices here

The standard process of training neural networks

1. Design the graph that defines the computation you want

2. Initialize the graph

   Either randomly, or with pre-trained parameters

3. Iterate over example (or mini-batches of examples):
   1. Run the forward pass to calculate the result of the computation using the current parameters
   2. Define the loss for the network over the current example

      Characterizes the idea of "how bad is the result that was just computed"
   3. Compute the gradient of the loss using backpropagation
   4. Update the parameters

We have different design choices here

Standard libraries offer many choices

https://pytorch.org/docs/stable/nn.html

torch.nn

These are the basic building blocks for graphs:

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization
    - Aliases

Nearly everything here is a computation graph

That is, they have the same semantics as we saw before

Over the semester, we will be looking at some non-standard design choices for loss functions and networks

Neural network architecture design typically involves standard building blocks

- Softmax: Convert a set of k real valued scores into a distribution over k items
- Multilayer Perceptron (commonly two layered): Abstract an unknown function that produces a tensor
- Attention: Assign an "relevance" score or distribution over a set of items given some context
- Self-attention: Attention over a elements of a sequence, where the context is the sequence itself
- Recurrent networks: Process sequences (text, speech, time series, etc) where the computation for each time step depends on previous ones
- Convolution: Aggregate local features in a tensor, typically used for images
- Transformer: Encode a collection of items with self-attention + MLP
- Graph neural network: Encode graphs in a way that is aware of the graph structure

There are standard building blocks for loss functions

- Cross entropy loss: How far is the distribution produced over a set of categories (via softmax) from a desired one?

- Squared loss or MSE loss: How far is a real number from a desired one?

Other losses exist too (e.g. ranking loss)

What we saw in this lecture

A quick review of topics you should have already seen before

1. Neural networks
2. Tensors
3. Computation graphs
4. Loss functions and training
5. Design patterns