Recitation 2:
Computing Derivatives

(A story of influences)

Today

- **Goal:** Conceptual understanding of the math behind backprop/autograd
- This will be helpful for **hw1p1** and DL in general
  - hw1p1 writeup should be enough to complete assignment
  - But this recitation will provide context, breadth, and depth
  - Concepts here will be useful throughout the course
- **We'll try to minimize overlap with the writeup to keep this helpful**

Today

- Questions? Raise your hand or chat; happy to help.
- But some requests:
  1. **Avoid getting too technical,** except if you’re confused or if you spot significant errors
     - There’s a lot of topics and all are complicated (with exceptions and edge-cases).
  2. **Try not to interrupt flow with long questions or too many follow-up questions**
     - Ideas need to feel coherent for students

Agenda

1. Motivation: Training and Loss
2. Backprop: Derivatives, Gradients, and the Chain Rule
3. Intro to Autograd (WIP; will cover in actual recitation)
4. Helpful code/math/tips
   - Depth-First Search and recursion (Autograd backward)
   - Derivatives on matrix operations (WIP; will cover in actual recitation)
5. Autograd example

Motivation: Training and Loss

Why Calculus?

- Training a NN is essentially an optimization problem.

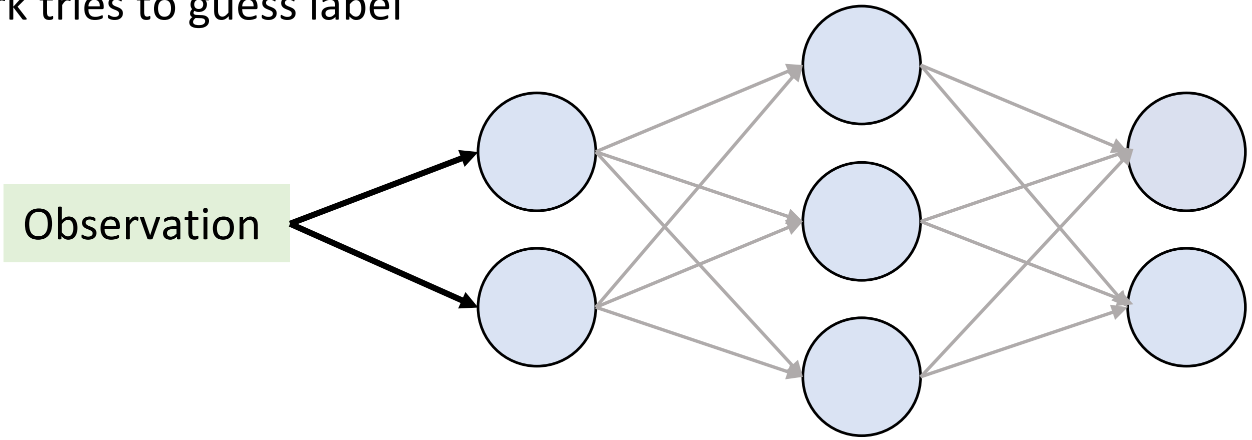
Goal: Minimize the loss by adjusting network parameters

- To see how an NN does this, let’s look at a single training loop iteration

Training a Neural Network

1. Forward Propagation

a. Provide observation to network, network tries to guess label



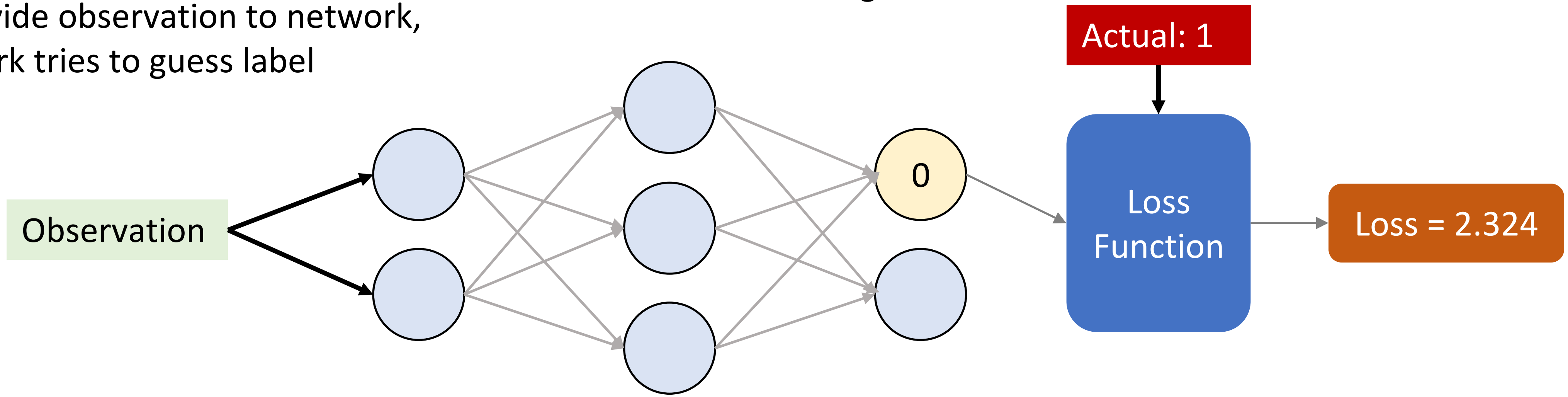
Training a Neural Network

1. Forward Propagation

a. Provide observation to network, network tries to guess label

b. Network makes guess

c. "Score" performance by generating a loss value.



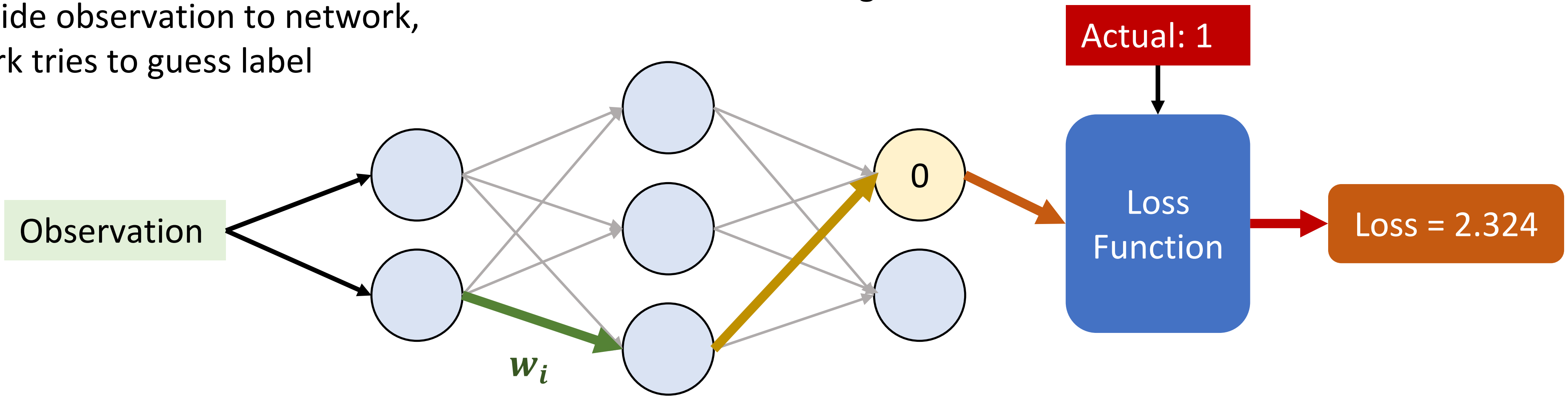
Training a Neural Network

1. Forward Propagation

a. Provide observation to network, network tries to guess label

b. Network makes guess

c. "Score" performance by generating a loss value.



2. Backpropagation

Starting from the loss and moving backward through the network, calculate gradient of loss w.r.t. each param ($\frac{\partial \text{loss}}{\partial w_i}$)

Goal is to understand how adjusting each param would affect the loss.

$$\frac{\partial \text{Loss}}{\partial w_i} = \frac{\partial \text{Loss}}{\partial \text{LossFunc}} \cdot \frac{\partial \text{LossFunc}}{\partial \text{Guess}} \cdot \frac{\partial \text{Guess}}{\partial w_j} \cdot \frac{\partial w_j}{\partial w_i}$$

(For each w_i)

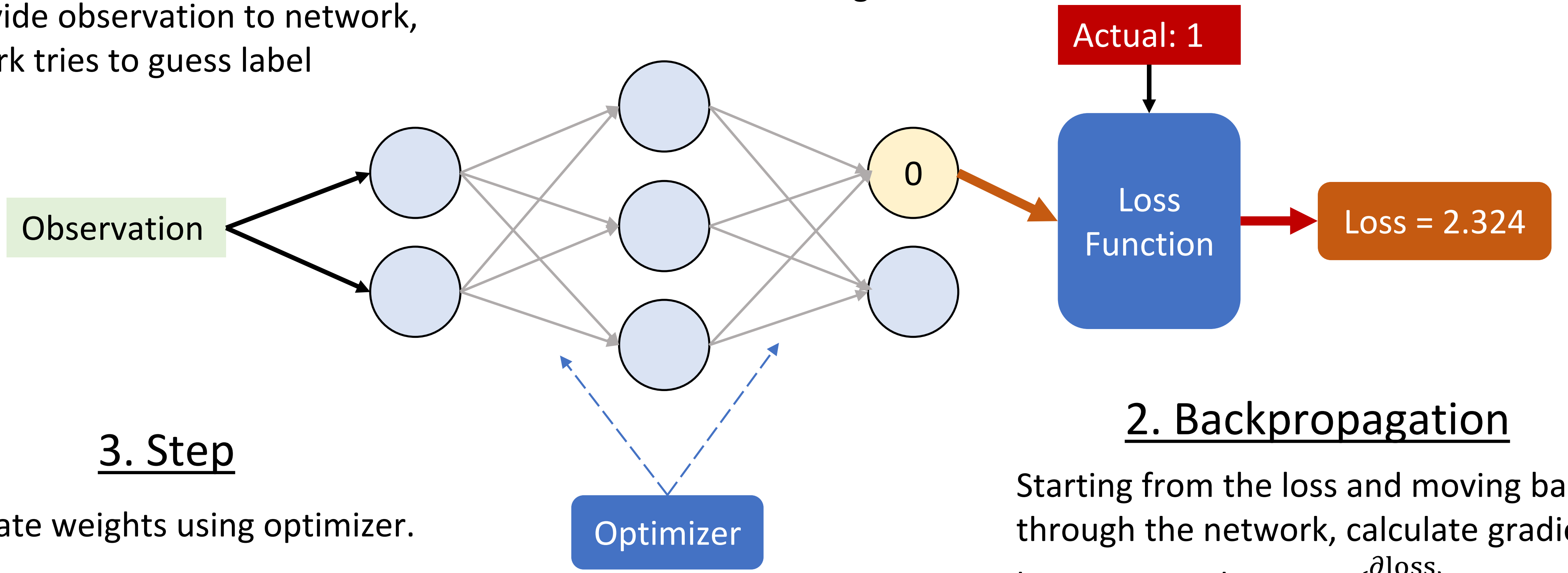
Training a Neural Network

1. Forward Propagation

a. Provide observation to network, network tries to guess label

b. Network makes guess

c. “Score” performance by generating a loss value.



2. Backpropagation

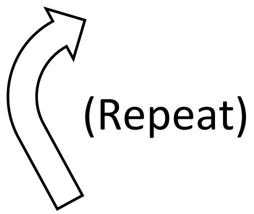
Starting from the loss and moving backward through the network, calculate gradient of loss w.r.t. each param ($\frac{\partial \text{loss}}{\partial w_i}$)

Goal is to understand how adjusting each param would affect the loss.

3. Step

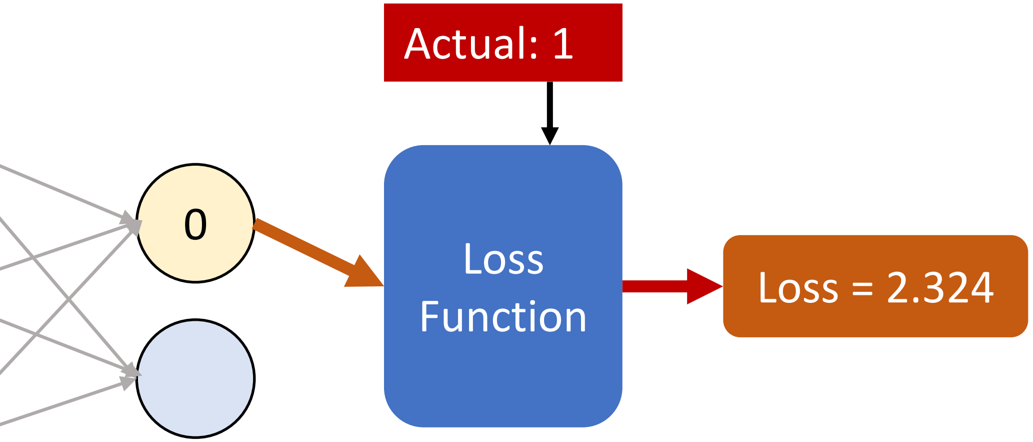
Update weights using optimizer.

The optimizer, based on the gradients, determines how to update weights in order to minimize loss



Loss Values

Loss Function & Value



- **Really important in ML and optimization**
- General metric for evaluating performance
- Minimizing an (appropriate) loss metric should cause improved performance

Example: CrossEntropyLoss

- Task: classifying dogs and cats

Observations (batch of 3)

Network

Example: CrossEntropyLoss

- Task: classifying dogs and cats

Observations (batch of 3)

Network

Logits (label confidence)

	Cat	Dog
Img_1	1.01	**1.2**
Img_2	**12.5**	0.02
Img_3	0.23	**1.35**

Size: (batch_size, num_possible_labels)

Example: CrossEntropyLoss

- Task: classifying dogs and cats

Observations (batch of 3)

Network

Logits (label confidence)

Notice:
incorrect guess

	Cat	Dog
Img_1	1.01	**1.2**
Img_2	**12.5**	0.02
Img_3	0.23	**1.35**

Size: (batch_size, num_possible_labels)

Example: CrossEntropyLoss

Logits (label confidence)

	Cat	Dog
Img_1	1.01	**1.2**
Img_2	**12.5**	0.02
Img_3	0.23	**1.35**

True labels

0	**Cat**
0	**Cat**
1	**Dog**

Size: (batch_size,)

Example: CrossEntropyLoss

Logits (label confidence)

	Cat	**Dog**
Img_1	1.01	**1.2**
Img_2	**12.5**	0.02
Img_3	0.23	**1.35**

True labels

0	**Cat**
0	**Cat**
1	**Dog**

CrossEntropyLoss

Example: CrossEntropyLoss

Logits (label confidence)

	Cat	Dog
Img_1	1.01	1.2
Img_2	12.5	0.02
Img_3	0.23	1.35

True labels

0	**Cat**
0	**Cat**
1	**Dog**

CrossEntropyLoss

0.3583 **Average Loss Value for Batch**

Loss Value - Notes

- Details of CrossEntropyLoss calculation in hw1p1 writeup
- There are many other possible ways to define loss, and each incentivize/punish different aspects of network training
- In general:
  - Loss value is **one** float for the entire batch
    - Aggregate loss of each observation using summing or averaging
      - (Usually averaging; we'll do averaging in hw1p1)

Why loss instead of accuracy?

- Loss vs. accuracy (correct guesses / total)?
  - Loss is hard to interpret, which is bad
    - $0 \leq \text{Loss} \leq \ln(\text{num\_classes})$
  - BUT it captures more detail
    - Accuracy only cares about the final answer
    - Loss looks at the confidence in all labels
  - ALSO it's 'smoother'
    - In loss, partially correct answers are better than very incorrect
    - In accuracy, partially correct == very incorrect
- Compromise: train on loss, validate on accuracy
  - Makes validation results interpretable

Summary

- Loss value evaluates network performance
- The lower the loss, the better the performance
- This means:
  - Our goal is to modify network params to lower loss

Backprop:

Derivatives, Gradients, and the Chain Rule

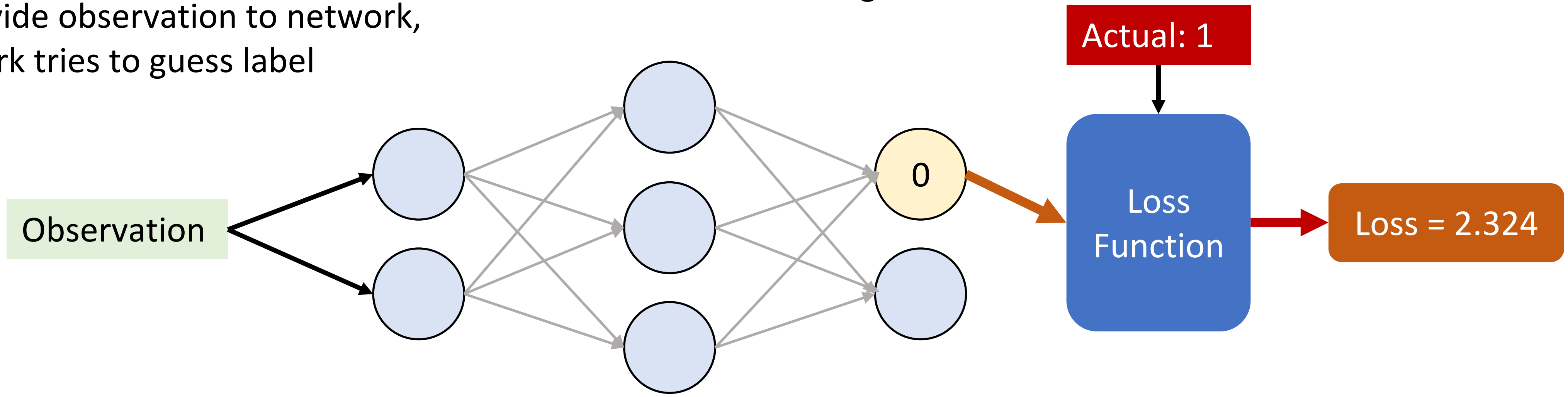
So far:

1. Forward Propagation

a. Provide observation to network, network tries to guess label

b. Network makes guess

c. “Score” performance by generating a loss value.



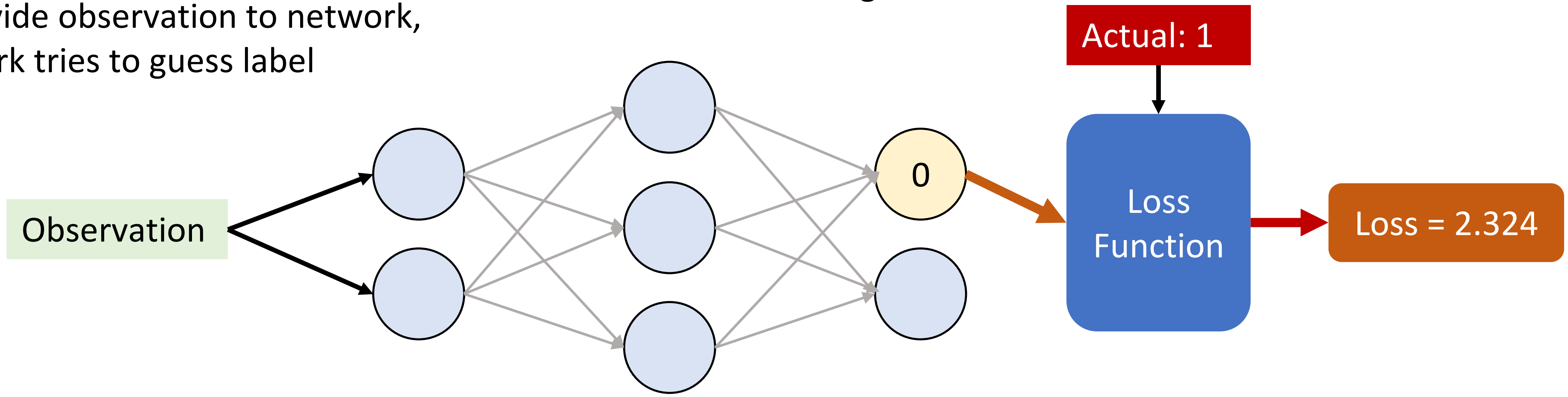
So far:

1. Forward Propagation

a. Provide observation to network, network tries to guess label

b. Network makes guess

c. “Score” performance by generating a loss value.



2. Backpropagation

Determine how each weight affects the loss by calculating partial derivative

3. Step

Adjust weights using those gradients

Backprop Interlude:

(Re)defining the Derivative

(Re)defining the Derivative

- You probably have experience with scalar derivatives and a bit of multivariable calc
- But how does that extend to **matrix derivatives**?

- **Now: intuition and context of scalar and matrix derivatives**
  - This should help you understand what derivatives actually do, how this applies to matrices, and what the **shapes** of the input/output/derivative matrices are.
  - This is better than memorizing properties.

Scalar Derivatives (α definition)

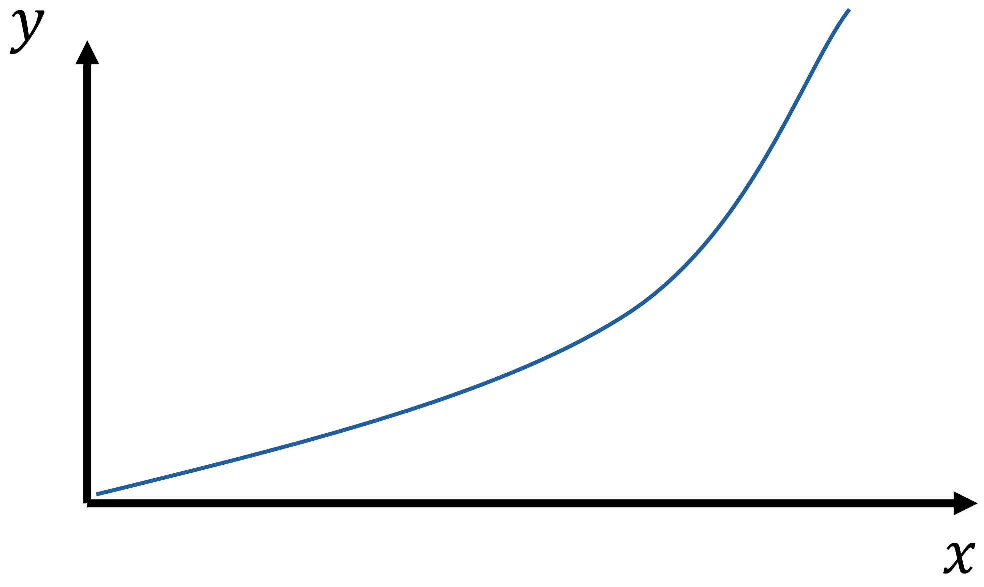
$$f(x) = y$$

x and y are scalars

Scalar Derivatives (α definition)

$$f(x) = y$$

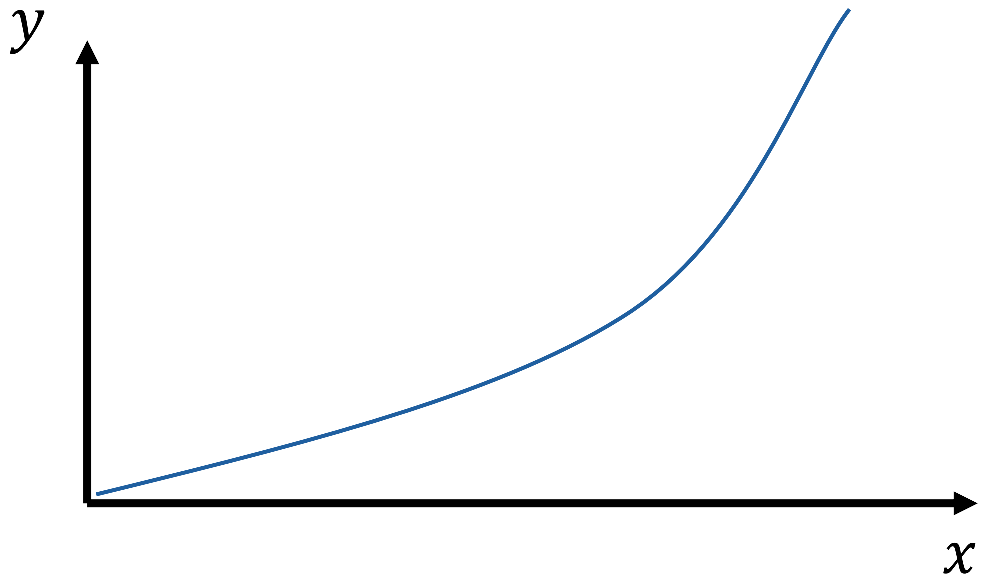
x and y are scalars



Scalar Derivatives (α definition)

$$f(x) = y$$

x and y are scalars

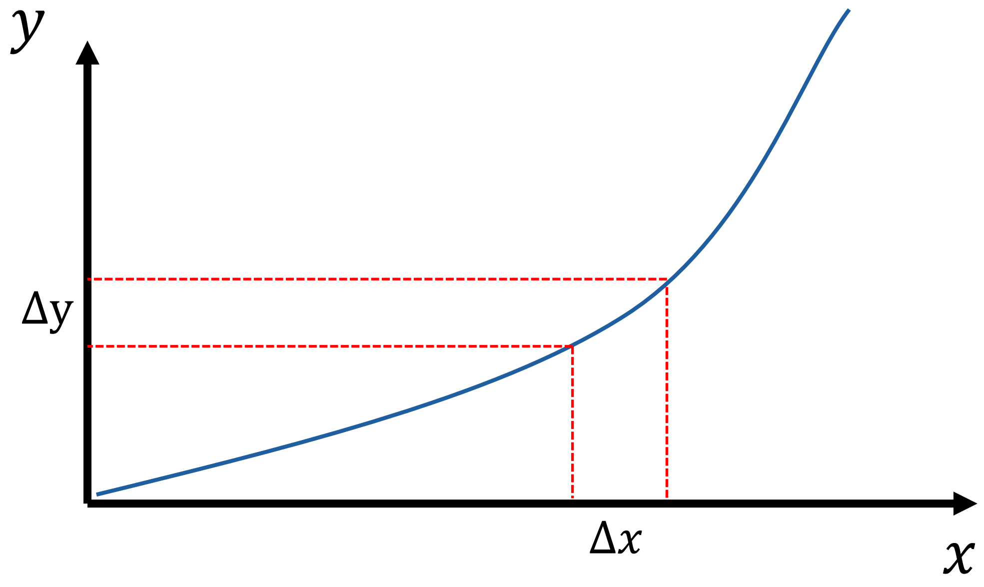


Goal: determine how changing the input affects the output

Scalar Derivatives (α definition)

$$f(x) = y$$

x and y are scalars



Goal: Find Δy given Δx

Scalar Derivatives (α definition)

We define relationship between Δx and Δy as α.

$$\Delta y = \alpha \Delta x$$

- α is some factor multiplied to $\Delta \mathrm{x}$ that gives Δy

Scalar Derivatives (α definition)

We define relationship between $\boldsymbol{\Delta x}$ and $\boldsymbol{\Delta y}$ as $\boldsymbol{\alpha}$.

Derivative $f'(x)$

$$\Delta y = \alpha \Delta x$$

- α is some factor multiplied to Δx that gives Δy.
- **Plot twist: $\boldsymbol{\alpha}$ is the derivative $\boldsymbol{f'(x)}$**

Derivatives (scalar in, scalar out)

$$\Delta y = f'(x)\,\Delta x$$

- Key idea: the derivative is not just a value (i.e. 'the slope')
- The derivative is a **linear transformation,** mapping Δx onto Δy.

$$f'(x) \colon \Delta x \longmapsto \Delta y$$

$$\mathbb{R}^1 \longmapsto \mathbb{R}^1$$

Derivatives (vector in, scalar out)

Let's go to *higher dimensions*. **Multiple arguments and scalar output**.

$$f(x_1, \dots, x_D) = y$$

Vector-scalar derivatives use the same general form as scalar-scalar derivatives.

To do this, group the input variables into a 1-D vector $\mathbf{x}$.

$$\Delta y = \boldsymbol{\alpha} \cdot \mathbf{x}$$

$$= \begin{bmatrix} a_1 & \dots & a_D \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Note: vectors are notated in bold and unitalicized font.

Derivatives (vector in, scalar out)

$$\Delta y = \boldsymbol{\alpha} \cdot \mathbf{x}$$

$$= \begin{bmatrix} a_1 & \dots & a_D \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Same thing below, but in more familiar notation:

$$\Delta y = \frac{\partial y}{\partial \mathbf{x}} \cdot \mathbf{x}$$

$$= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Derivatives (vector in, scalar out)

$$\Delta y = \boldsymbol{\alpha} \cdot \mathbf{x}$$

$$= \begin{bmatrix} a_1 & \dots & a_D \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Same thing below, but in more familiar notation:

This is the derivative

(1 x D) row vector

$$\Delta y = \frac{\partial y}{\partial \mathbf{x}} \cdot \mathbf{x}$$

$$= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Derivatives (vector in, scalar out)

In summary, for a function of multiple arguments $\mathbf{x}$ and scalar output y

$$f(\boldsymbol{x}) = y$$

Its derivative is this:

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix}$$

Note: the derivative's shape will always be transposed from the input shape.

This will be true for ALL matrix derivatives (See next slide for why)

Derivatives are Dot Products

Recall:

$$\Delta y = \nabla_{\mathbf{x}} y \cdot \mathbf{x}$$

$$= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

By notational convention:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^T$$

Derivatives (vector in, vector out)

- For a function that inputs and outputs vectors, $\nabla_{\mathbf{x}}\mathbf{y}$ is the **"Jacobian"**.

Input **Output**

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_K \end{bmatrix} \Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \cdots & \frac{\partial y_K}{\partial x_D} \end{bmatrix}$$

$\mathrm{D} \times 1$ $\quad$ $\mathrm{K} \times 1$ $\quad$ $\mathrm{K} \times \mathrm{D}$

Derivatives (vector in, vector out)

- For a function that inputs and outputs vectors, $\nabla_{\mathbf{x}}\mathbf{y}$ is the **"Jacobian"**.

Input	**Output**
$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}$	$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_K \end{bmatrix}$
$\mathrm{D} \times 1$	$\mathrm{K} \times 1$

Each row is a vector-scalar derivative

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \boxed{\frac{\partial y_1}{\partial x_1} \cdots \frac{\partial y_1}{\partial x_D}} \\ \vdots \quad \ddots \quad \vdots \\ \frac{\partial y_K}{\partial x_1} \cdots \frac{\partial y_K}{\partial x_D} \end{bmatrix}$$

$\mathrm{K} \times \mathrm{D}$

Note: each row of the derivative matrix is essentially a vector-scalar matrix from the previous slide

Summary

Covered 3 cases:

1. Scalar/scalar function derivative $f'(x)$
2. Vector/scalar derivative $\frac{\partial y}{\partial \mathbf{x}} = \left[\frac{\partial y}{\partial x_1} \quad \dots \quad \frac{\partial y}{\partial x_D}\right]$
3. Vector/vector derivative $\left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)$

Key Ideas

- The derivative is the **best linear approximation** of f at a point
- The derivative describes **the effect of each input on the output**

But what is the gradient?

'Gradients' are the **transpose of a vector-scalar derivative**

$$\nabla f = \left(\frac{\partial y}{\partial \mathbf{x}}\right)^T = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_D} \end{bmatrix}$$

They're technically different from normal derivatives, but have many similar properties. So in conversation, people will often interchange the two.

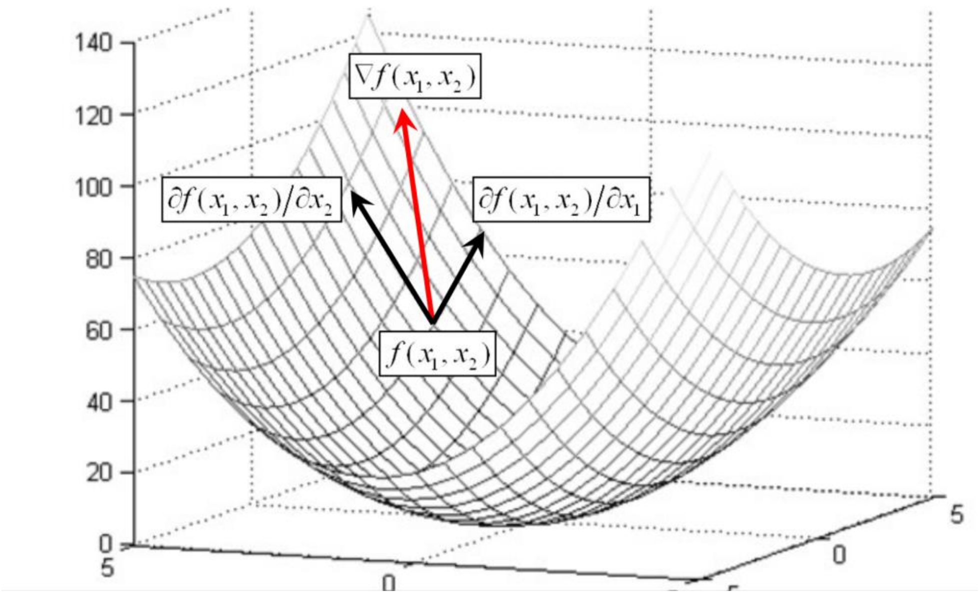
One difference: interpretation

While the derivative projects change in output onto change in input, the gradient is that change in input interpreted as a vector. Also, as it's a tangent vector to the input space at a point, you can interpret it in the context of the input space. Derivative would be cotangent.

(^ you don't need to fully understand this for class, don't worry (see here for more))

But what is the gradient?

- One nice property: Great for **optimization** (finding max/min)
  - The gradient is a vector that points towards the '**direction' of steepest increase**.



img source

- If **maximizing**, follow the gradient.
- If **minimizing**, go in the opposite direction (gradient descent)

Partial vs. Total Derivatives

$$\frac{dy}{d\mathbf{x}} \quad \text{vs} \quad \frac{\partial y}{\partial x_i}$$

- The total influence of $\mathbf{x}$ on y
- (Was today's topic, same as $\boldsymbol{\alpha}$ or ∇)

- The influence of just x_i on y
- Assumes other variables are held constant

Remember before:

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix}$$

But this is pretty idealized; if variables influence each other, it gets messy

Things get messy

Find $\frac{dy}{dx}$ for $\boldsymbol{f(x,z) = y}$, where $\boldsymbol{z = g(x,w)}$

Things get messy

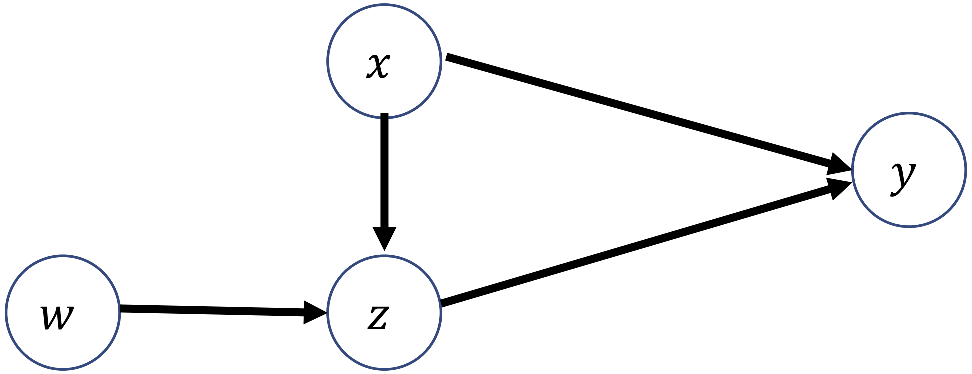
Find $\frac{dy}{dx}$ for $\boldsymbol{f(x,z) = y}$, where $\boldsymbol{z = g(x,w)}$

x affects y twice; directly in f, and indirectly through z.

Things get messy

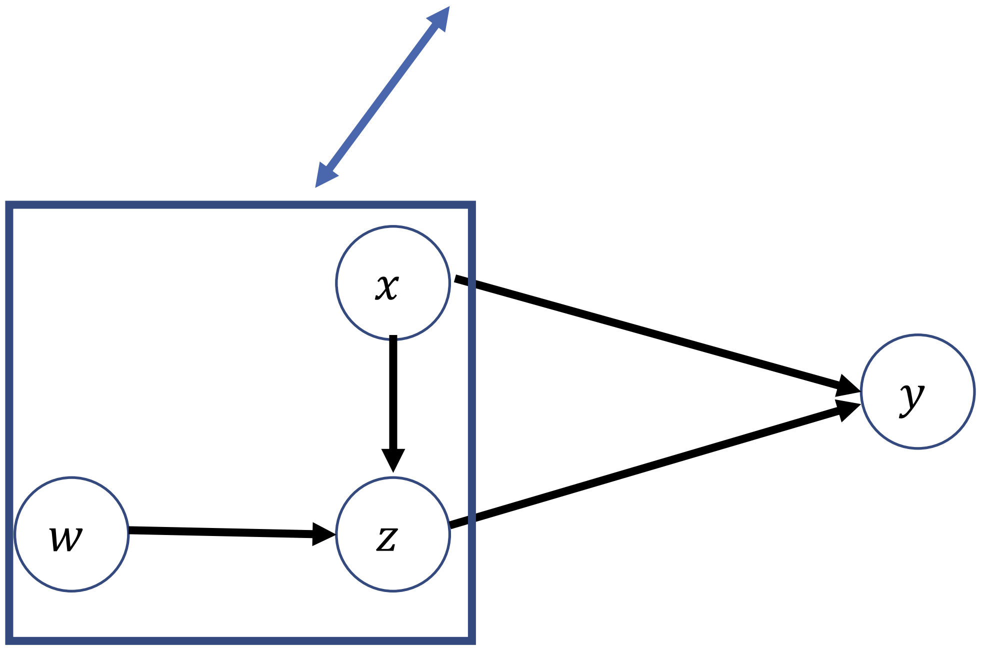
Find $\boldsymbol{\frac{dy}{dx}}$ for $\boldsymbol{f(x, z) = y}$, where $\boldsymbol{z = g(x, w)}$

x affects y twice; directly in f, and indirectly through z.



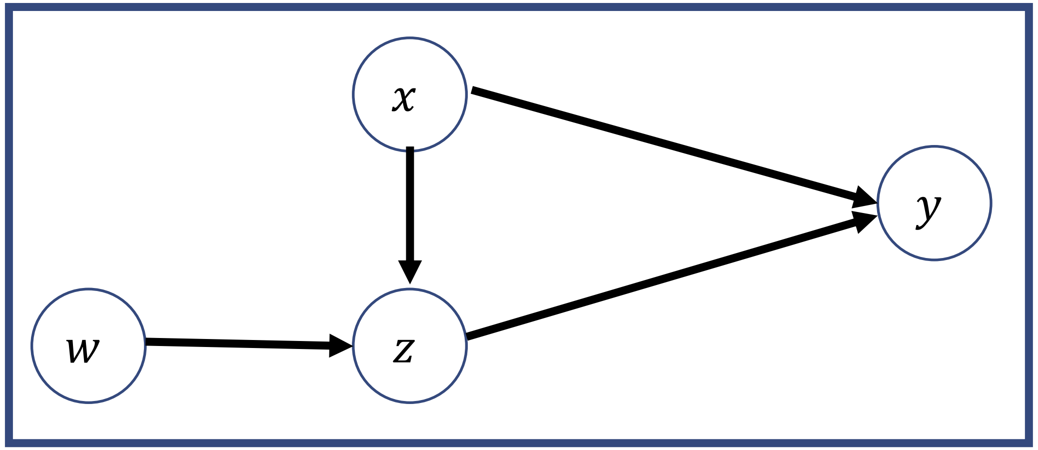
Things get messy

Find $\frac{dy}{dx}$ for $f(x, z) = y$, where $z = g(x, w)$



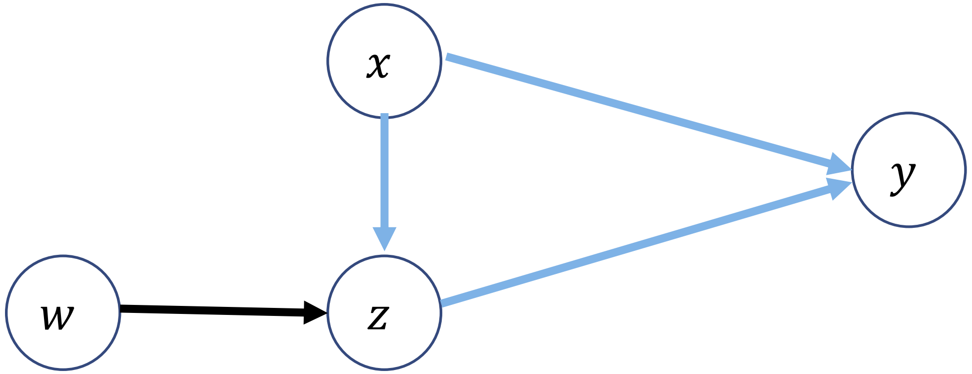
Things get messy

Find $\frac{dy}{dx}$ for $f(x, z) = y$, where $z = g(x, w)$



Things get messy

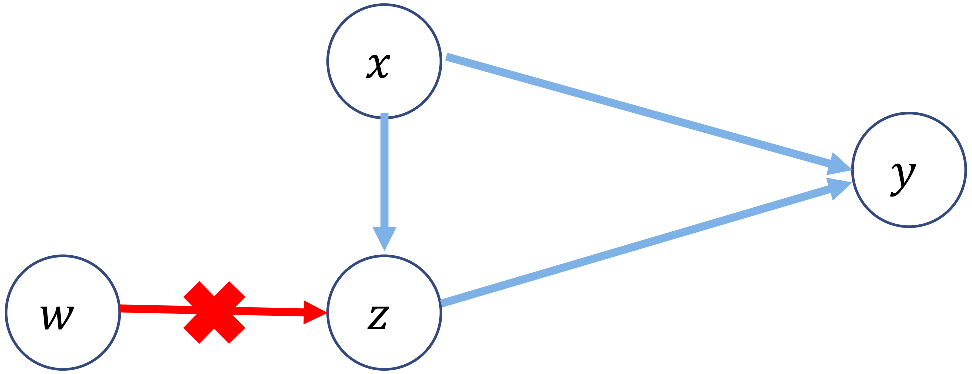
Find $\frac{dy}{dx}$ for $\boldsymbol{f(x,z) = y}$, where $\boldsymbol{z = g(x,w)}$



Goal: get only x's influence on y

Things get messy

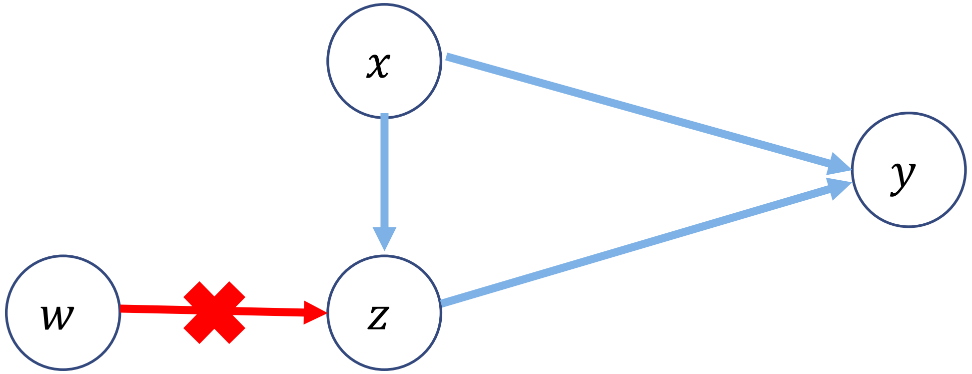
Find $\frac{dy}{dx}$ for $f(x, z) = y$, where $z = g(x, w)$



If we just said $\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}$, we'd end up including w's influence on y.

Things get messy

Find $\frac{dy}{dx}$ for $f(x, z) = y$, where $z = g(x, w)$



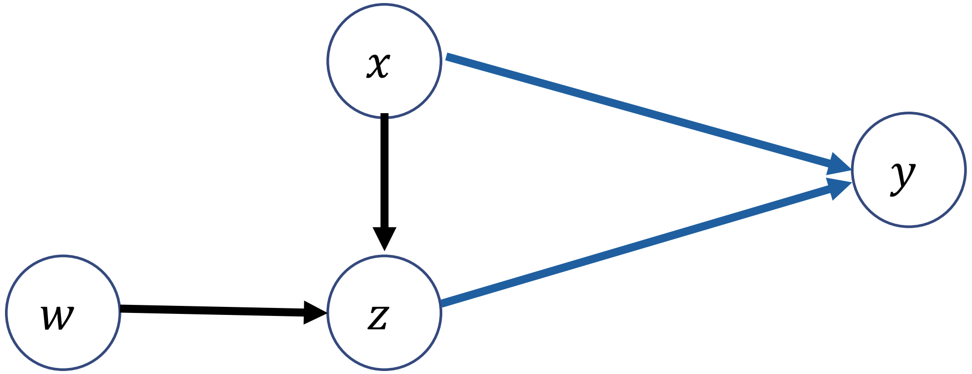
It's time for... "***<u>the chain rule</u>***"

The Chain Rule

- The chain rule is used to properly **account for influences in nested functions**
    - Recursively calculates derivatives on nested functions w.r.t. target

The Chain Rule

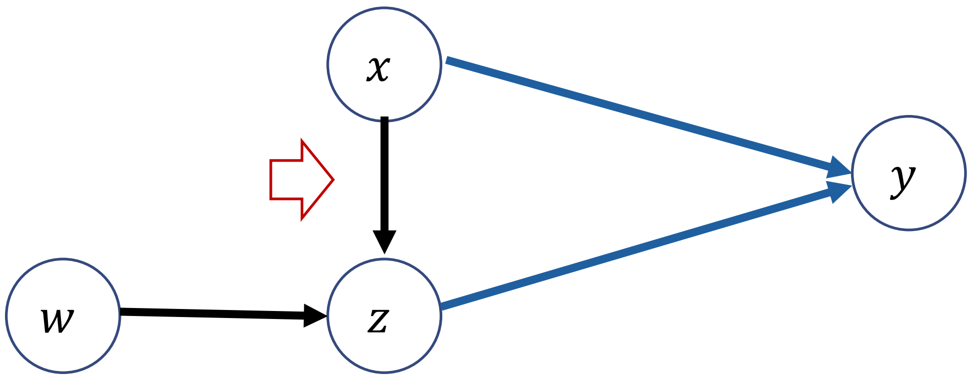
- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target



$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}$$

The Chain Rule

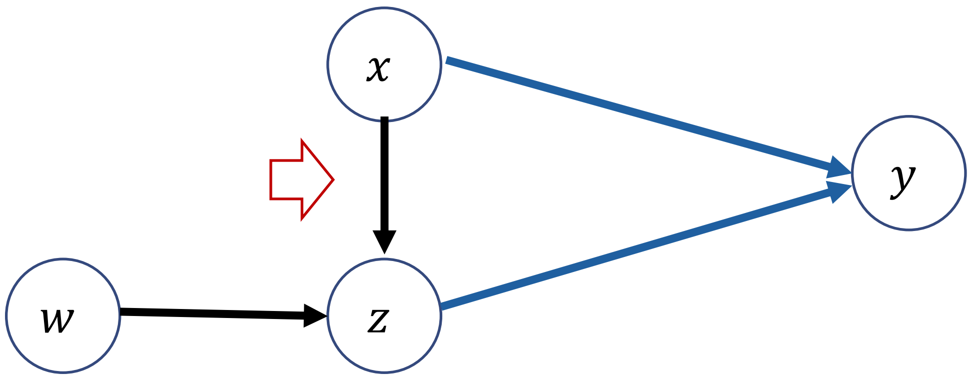
- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target



$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z} ?$$

The Chain Rule

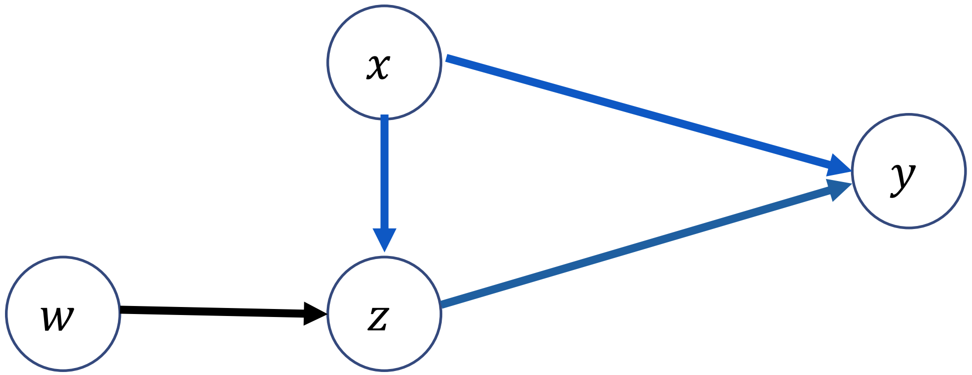
- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target



$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}\frac{dz}{dx}$$

The Chain Rule

- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target

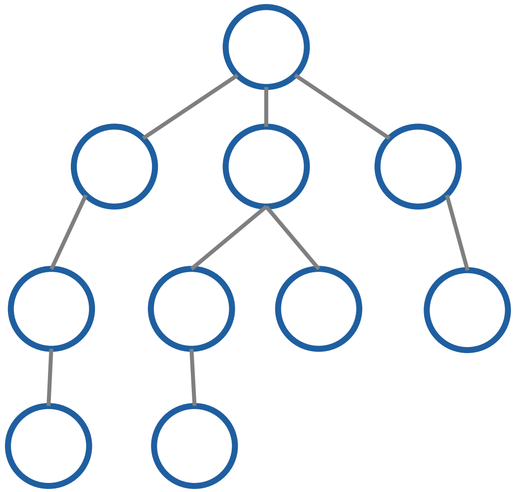


$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}\frac{dz}{dx}$$

HW1P1 Help & Tips

Depth-First Search (DFS)

- We'll briefly cover DFS, as it's needed for autograd
- **Algorithm used to traverse nodes in trees/graph**
  - Anything with vertices/edges; directed or undirected

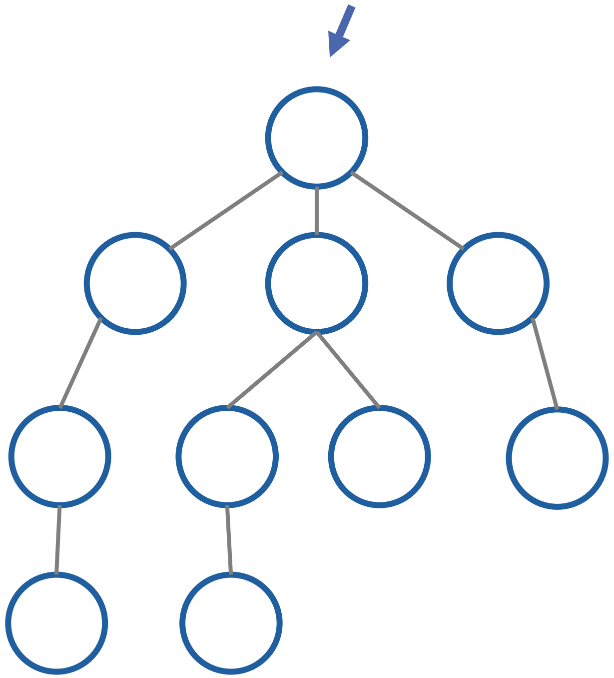


Example of a graph

Depth-First Search (DFS)

Goal: To visit every node in the graph, starting from some node

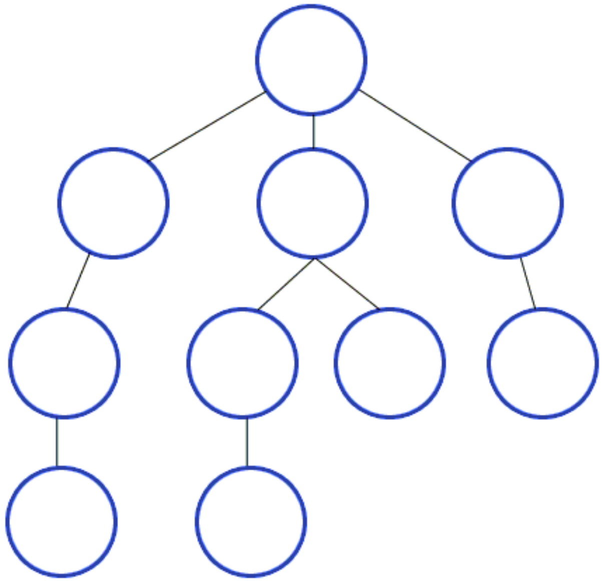
i.e. Start from here



Depth-First Search (DFS)

Goal: To visit every node in the graph, starting from some node

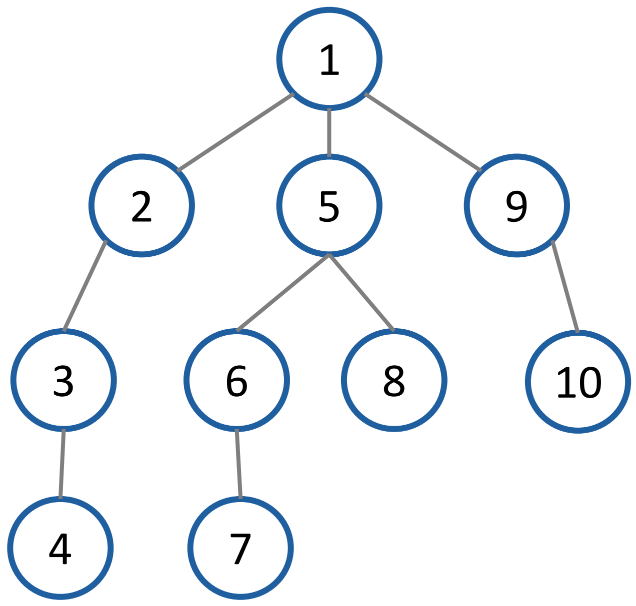
i.e. Start from here



(Animated GIF source)

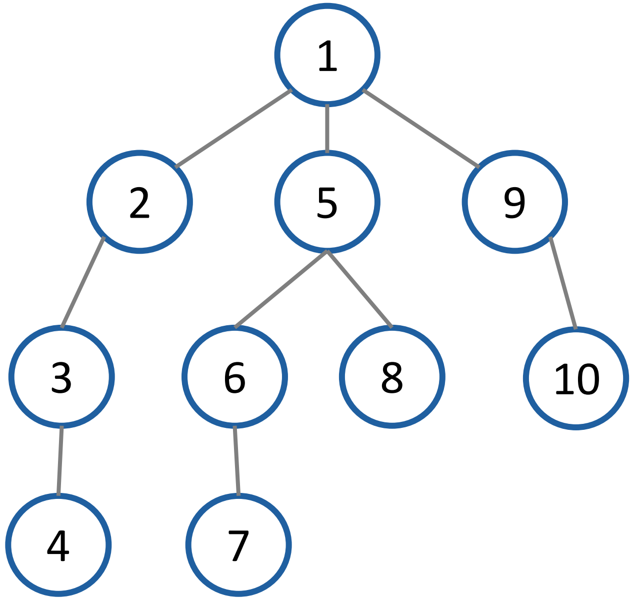
Depth-First Search (DFS)

- There's multiple ways to implement DFS, but our implementation of autograd uses **recursion**
- Recursion
  - When a function calls itself, leading to 'nested' calls



Depth-First Search (DFS)

- There's multiple ways to implement DFS, but our implementation of autograd uses **recursion**
- Recursion
  - When a function calls itself, leading to 'nested' calls



Recursion

- Useful for tasks where you're repeating instructions/checks
  - For example, traversing graphs
- Essentially performs 'iterative' tasks (just like `while` loops)
  - In fact, iteration and recursion are equally expressive
- Similar to `while` loops, you generally need one or more **base case(s)** that tell the function when to stop recursing
  - Otherwise it recurses infinitely and crashes your computer ☹

Recursion (Simple Example)

```
def greater_than_three(x):
	print("Recursive call, x=" + str(x))
	if x < 3:
		result = greater_than_three(x + 1)
		print("Received: x=" + str(result) + " and returning upward.")
		return result
	else:
		print("Hit base case. x=" + str(x))
		return x
```

- This method will continually make recursive calls until the base case
  - Base case: input value is >=3
- After hitting the base case, repeatedly close the nested iterations

Recursion (Simple Example)

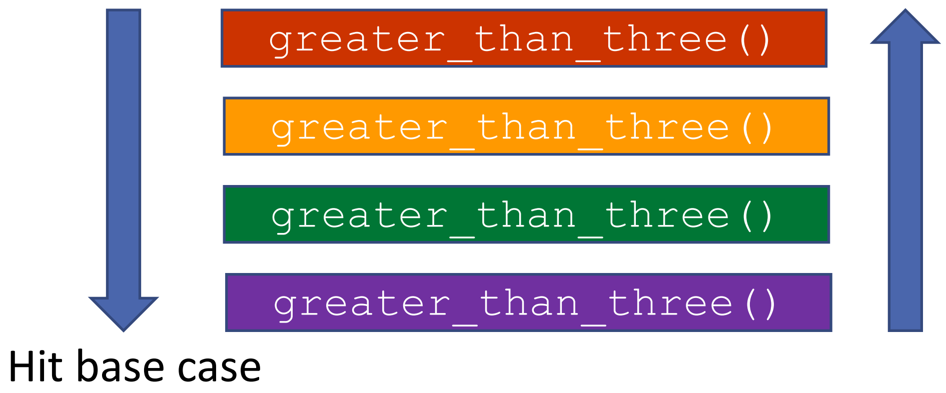
```
def greater_than_three(x):
        print("Recursive call, x=" + str(x))
        if x < 3:
                result = greater_than_three(x + 1)
                print("Received: x=" + str(result) + " and returning upward.")
                return result
        else:
                print("Hit base case. x=" + str(x))
                return x
```

```
>>> result = greater_than_three(0)
Recursive call, x=0
Recursive call, x=1
Recursive call, x=2
Recursive call, x=3
Hit base case (>=3). x=3
Received: x=3 and returning upward.
Received: x=3 and returning upward.
Received: x=3 and returning upward.
>>> print("Final result: x=" + str(result))
Final result: 3
```

Recursion (Simple Example)

```
def greater_than_three(x):
        print("Recursive call, x=" + str(x))
        if x < 3:
                result = greater_than_three(x + 1)
                print("Received: x=" + str(result) + " and returning upward.")
                return result
        else:
                print("Hit base case. x=" + str(x))
                return x
```

```
>>> result = greater_than_three(0)
Recursive call, x=0
Recursive call, x=1
Recursive call, x=2
Recursive call, x=3
Hit base case (>=3). x=3
Received: x=3 and returning upward.
Received: x=3 and returning upward.
Received: x=3 and returning upward.
>>> print("Final result: x=" + str(result))
Final result: 3
```



Recursion (Simple Example)

```
def greater_than_three(x):
        print("Recursive call, x=" + str(x))
        if x < 3:
                result = greater_than_three(x + 1)
                print("Received: x=" + str(result) + " and returning upward.")
                return result
        else:
                print("Hit base case. x=" + str(x))
                return x
```

```
# Here's an example where
# the base case is already met

>>> result = greater_than_three(4)
Recursive call, x=4
Hit base case (>=3). x=4
>>> print("Final result: x=" + str(result))
Final result: 4

# No nested calls were made.
```

Recursion

- You can modify the previous example to achieve different things
- For example, you don’t always need to return an output
- You can also ‘branch’
  - Calling the function multiple times on the same ‘level’

Recursion (Branching Example)

```
def branching_recursion(x):
    print("Recursive call, x=" + str(x))
    if isinstance(x, list):
        for item in x:
            branching_recursion(item)
    else:
        print("Hit base case (No more nested lists). x=" + str(x))
```

```
>>> branching_recursion([[1, 2], [[3], 4], 5])
Recursive call, x=[[1, 2], [[3], 4], 5]
Recursive call, x=[1, 2]
Recursive call, x=1
Hit base case (No more nested lists). x=1
Recursive call, x=2
Hit base case (No more nested lists). x=2
Recursive call, x=[[3], 4]
Recursive call, x=[3]
Recursive call, x=3
Hit base case (No more nested lists). x=3
Recursive call, x=4
Hit base case (No more nested lists). x=4
Recursive call, x=5
Hit base case (No more nested lists). x=5
```

Interpret this yourself for now, will discuss in detail on Friday

Extra Resources

Scalar Deriv. Cheat Sheet

Rule	$f(x)$	Scalar derivative notation with respect to x	Example
Constant	c	0	$\frac{d}{dx}99 = 0$
Multiplication by constant	cf	$c\frac{df}{dx}$	$\frac{d}{dx}3x = 3$
Power Rule	x^n	nx^{n-1}	$\frac{d}{dx}x^3 = 3x^2$
Sum Rule	$f + g$	$\frac{df}{dx} + \frac{dg}{dx}$	$\frac{d}{dx}(x^2 + 3x) = 2x + 3$
Difference Rule	$f - g$	$\frac{df}{dx} - \frac{dg}{dx}$	$\frac{d}{dx}(x^2 - 3x) = 2x - 3$
Product Rule	fg	$f\frac{dg}{dx} + \frac{df}{dx}g$	$\frac{d}{dx}x^2x = x^2 + x2x = 3x^2$
Chain Rule	$f(g(x))$	$\frac{df(u)}{du}\frac{du}{dx}$, let $u = g(x)$	$\frac{d}{dx}ln(x^2) = \frac{1}{x^2}2x = \frac{2}{x}$

Table Source

Good Resources

[The Matrix Calculus You Need For Deep Learning]

Matrix Calculus Reference

Gradients and Jacobians

The *gradient* of a function of two variables is a horizontal 2-vector:

$$\nabla f(x,y) = [\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}]$$

The *Jacobian* of a vector-valued function that is a function of a vector is an $m \times$
possible scalar partial derivatives:

Nice reference, with DL-specific examples and explanations

Good Resources

[Stanford CS231N – Vector, Matrix, and Tensor Derivatives]

5 The chain rule in combination with vectors and matrices

Now that we have worked through a couple of basic examples, let's combine these ideas with an example of the chain rule. Again, assuming $\vec{y}$ and $\vec{x}$ are column vectors, let's start with the equation

$$\vec{y} = VW\vec{x},$$

and try to compute the derivative of $\vec{y}$ with respect to $\vec{x}$. We could simply observe that the product of two matrices V and W is simply another matrix, call it U, and therefore

$$\frac{d\vec{y}}{d\vec{x}} = VW = U.$$

Clear rules and examples of how to take matrix derivatives.

Good Resources

- https://en.wikipedia.org/wiki/Matrix_calculus
  - Another excellent reference; just be careful about notation
- Khan Academy's article on gradients
  - **Simple/intuitive** visualizations and explanation
- https://en.wikipedia.org/wiki/Backpropagation
- https://en.wikipedia.org/wiki/Automatic_differentiation
- https://numpy.org/doc/stable/reference/routines.linalg.html
  - NumPy's matrix operations documentation