

# Recitation 2: Computing Derivatives

(A story of influences)

# Today

- **Goal:** Conceptual understanding of the math behind backprop/autograd
- Will also give some tips/hints for **hw1p1**
  - hw1p1 writeup should be enough to complete assignment
  - But this recitation will provide context, breadth, and depth
- But this will also be useful for DL in general
- **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 asking questions that are too technical/detailed/niche**
    - Except if you think it's important for hw1p1
    - Or if you spot a significant error
    - 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. Tips: code/math/resources
  - Depth-First Search and recursion (Autograd backward)
  - Derivatives on matrix operations
  - Useful links/videos
4. Autograd example

# Motivation: Training and Loss

# Why Calculus?

- Training a NN is essentially an optimization problem.

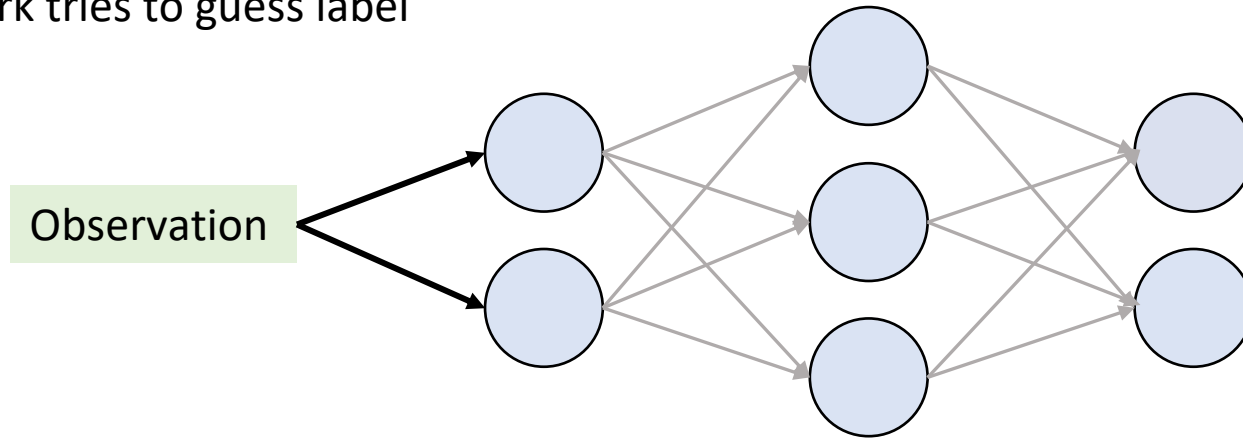
Goal: Minimize the loss by adjusting network parameters

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

# Training a Neural Network

## 1. Forward Propagation

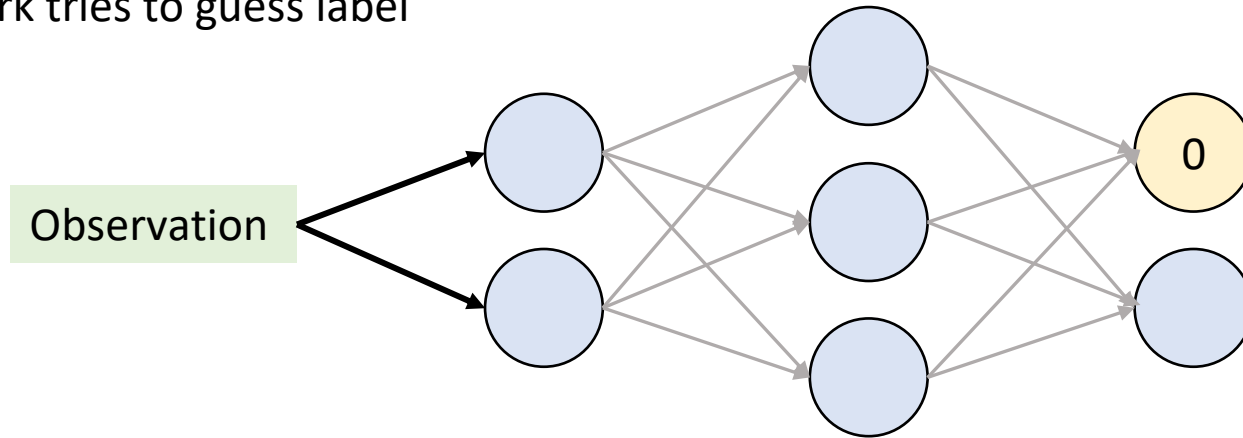
a. Provide observation to network,  
network tries to guess label



# Training a Neural Network

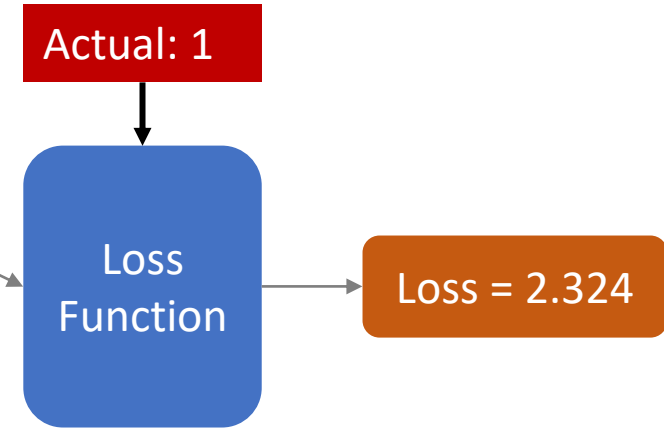
## 1. Forward Propagation

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



b. Network makes guess

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

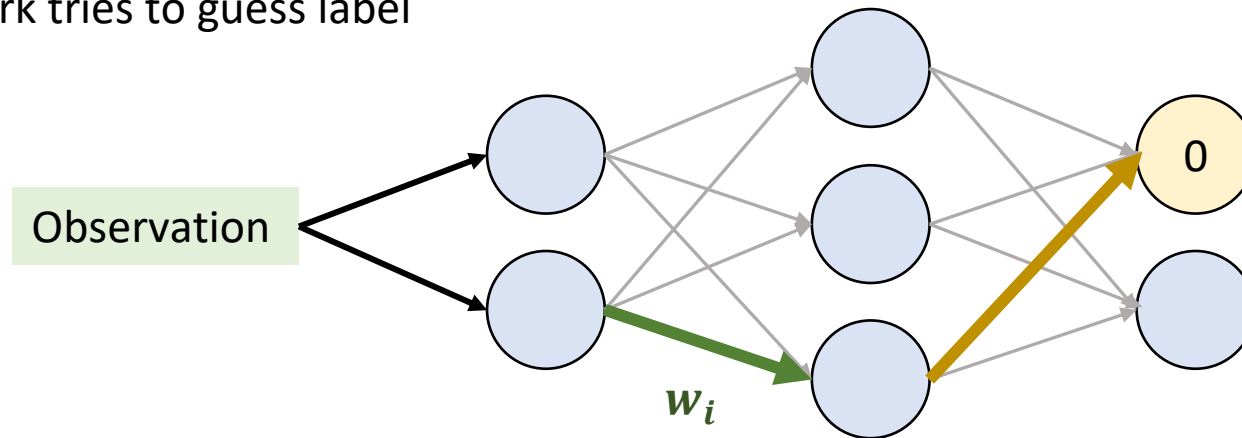




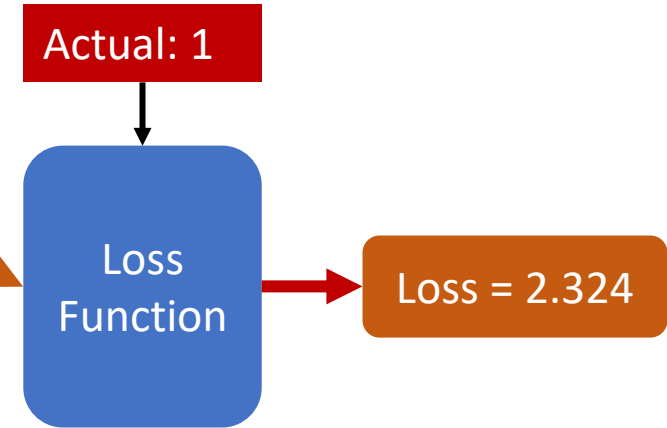
# Training a Neural Network

## 1. Forward Propagation

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



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  $w_i$ )

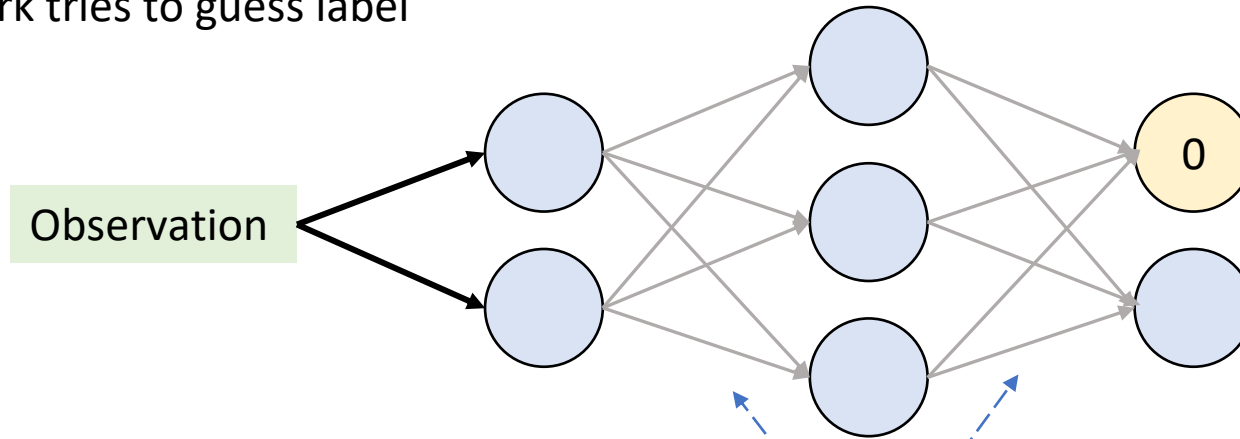
# Training a Neural Network

## 1. Forward Propagation

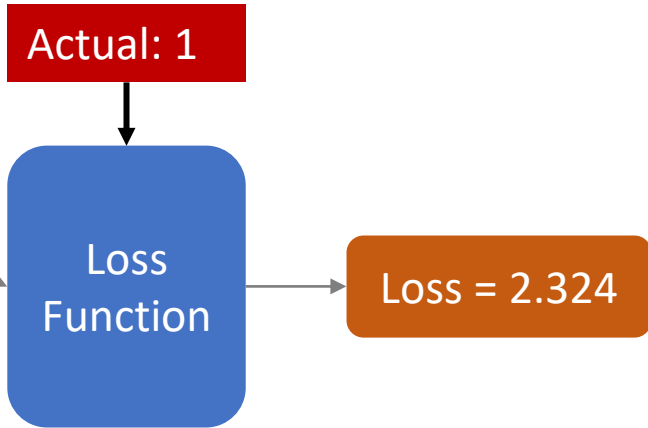
a. Provide observation to network, network tries to guess label

Observation

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.

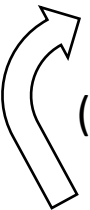
## 3. Step

Update weights using optimizer.

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

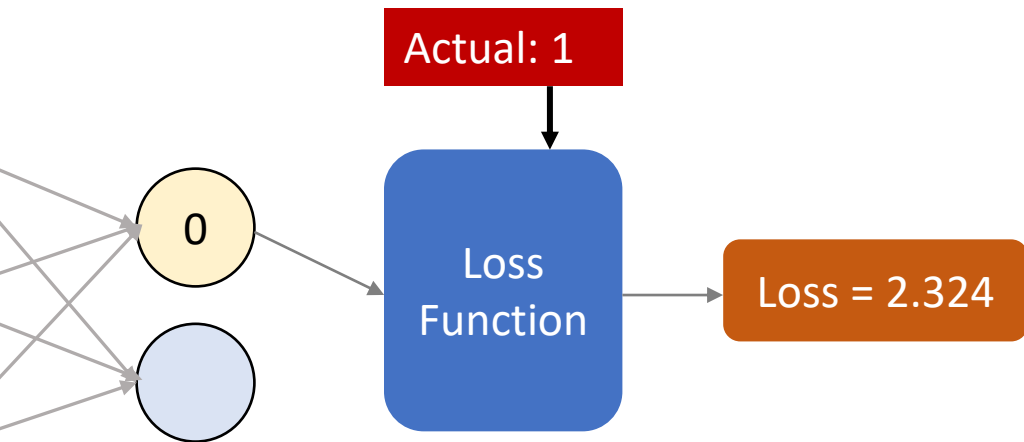
Optimizer

(Repeat)



# Loss Values

# Loss Function & Value

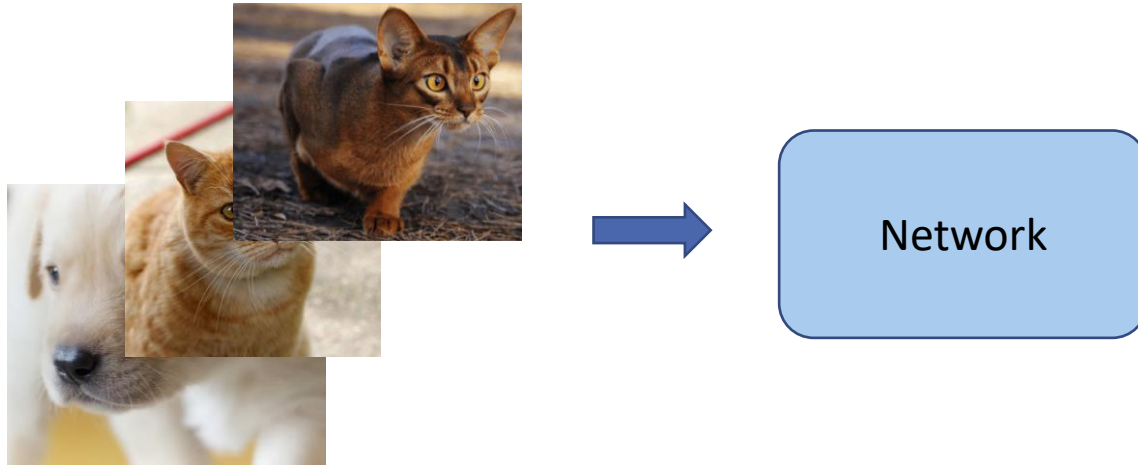


- **Really important in ML and optimization**
- General metric for evaluating performance
- Minimizing an (appropriate) loss metric should cause improved performance

# Example: CrossEntropyLoss

- Task: classifying dogs and cats

Observations (batch of 3)



Batching: passing in multiple input at once, calculating average loss across the batch

# Example: CrossEntropyLoss

- Task: classifying dogs and cats

Observations (batch of 3)



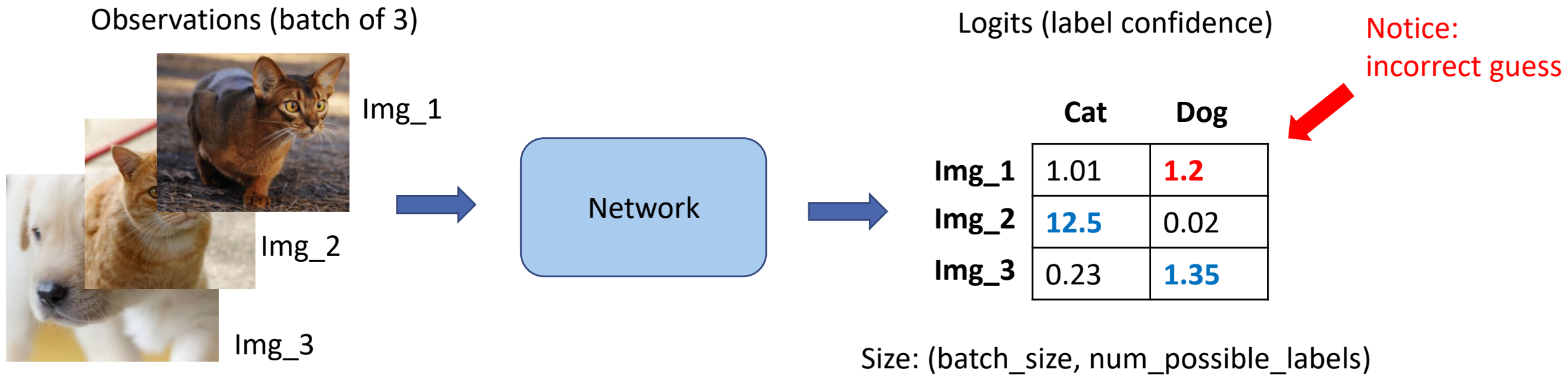
Logits (label confidence)

	Cat	Dog
Img_1	1.01	<b>1.2</b>
Img_2	<b>12.5</b>	0.02
Img_3	0.23	<b>1.35</b>

Size: (batch\_size, num\_possible\_labels)

# Example: CrossEntropyLoss

- Task: classifying dogs and cats



# Example: CrossEntropyLoss

Logits (label confidence)

	<b>Cat</b>	<b>Dog</b>
<b>Img_1</b>	1.01	<b>1.2</b>
<b>Img_2</b>	<b>12.5</b>	0.02
<b>Img_3</b>	0.23	<b>1.35</b>

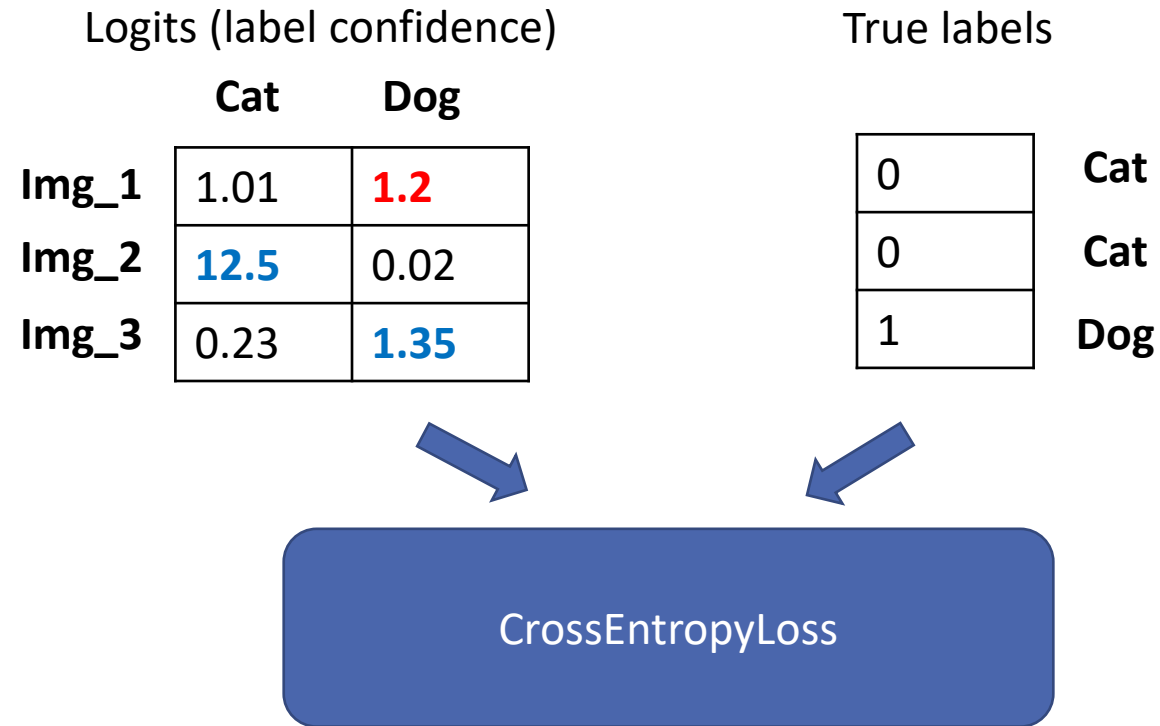
True labels

0	<b>Cat</b>
0	<b>Cat</b>
1	<b>Dog</b>

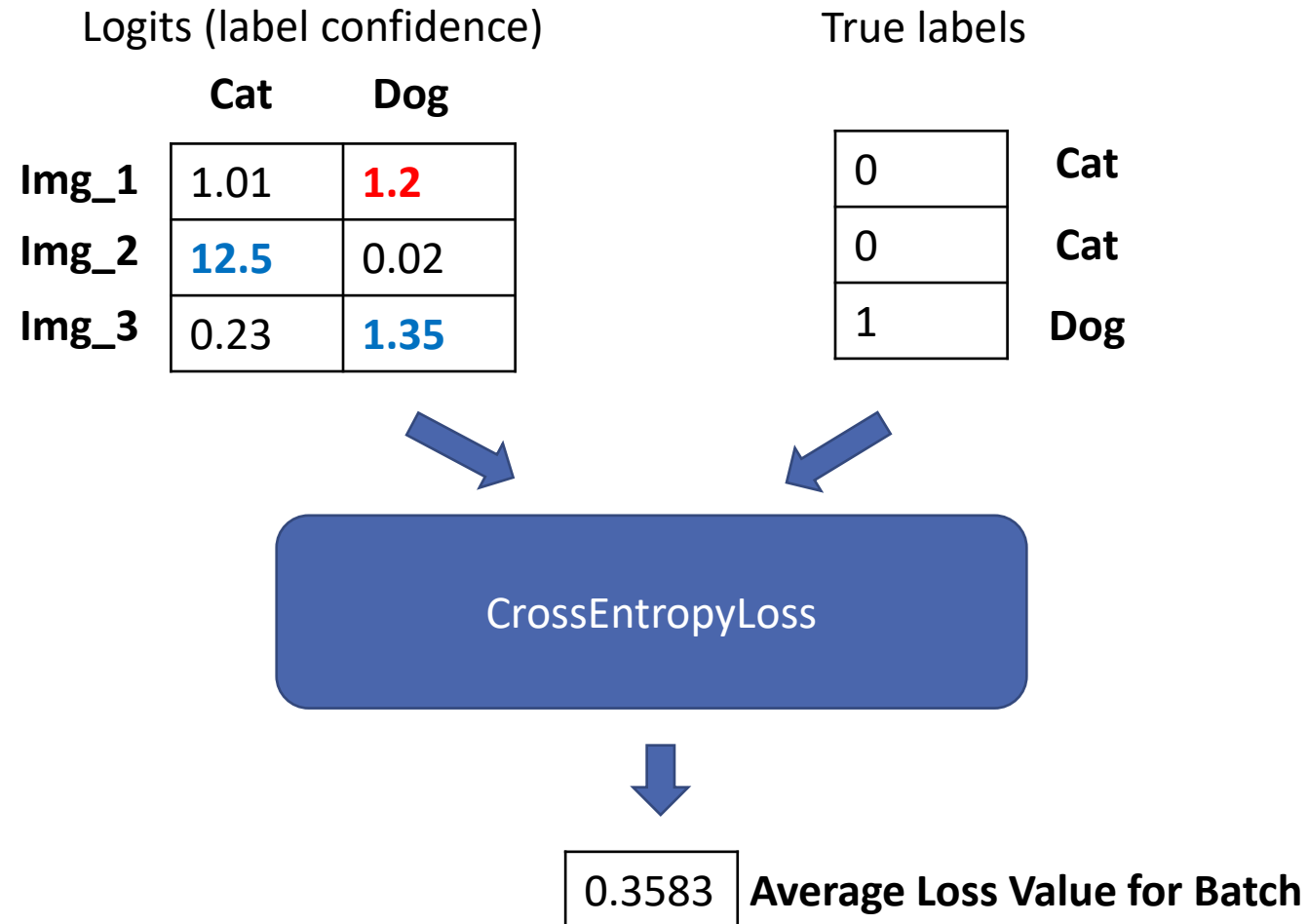
Size: (batch\_size, )



# Example: CrossEntropyLoss



# Example: CrossEntropyLoss



# 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 isn't desirable
    - $0 \leq \text{XELoss} \leq \ln(\text{num\_classes})$
  - Loss functions are 'smoother'
    - In loss, partially correct answers are better than very incorrect
    - In accuracy, partially correct == very incorrect
- Compromise: train on loss, validate on accuracy
  - Using accuracy during validation makes 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

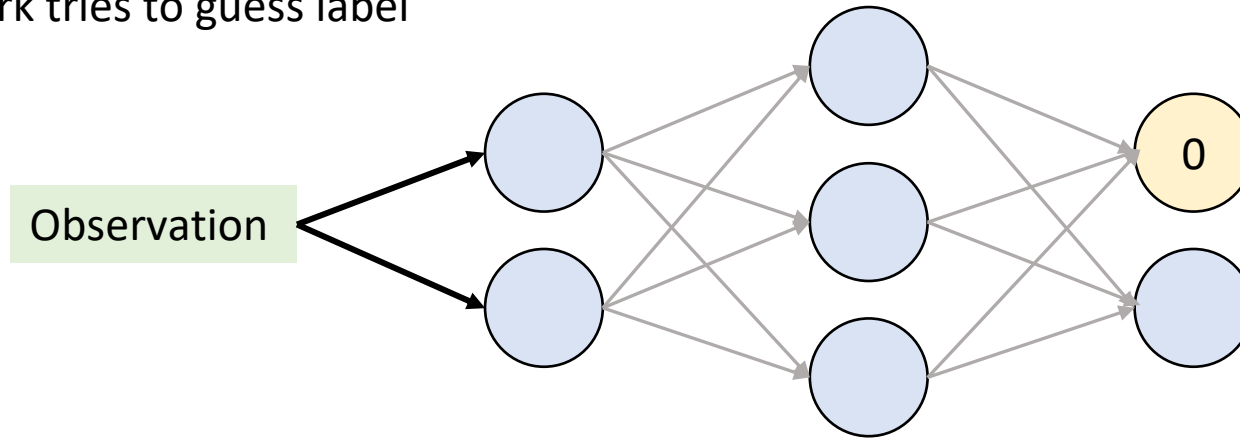
# Backprop:

Derivatives, Gradients, and the Chain Rule

# So far:

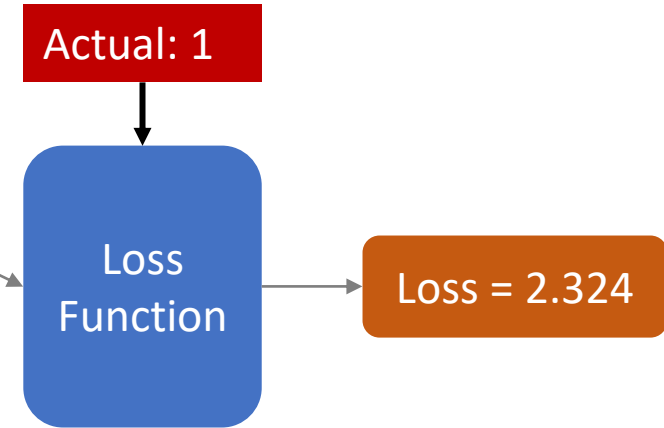
## 1. Forward Propagation

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



b. Network makes guess

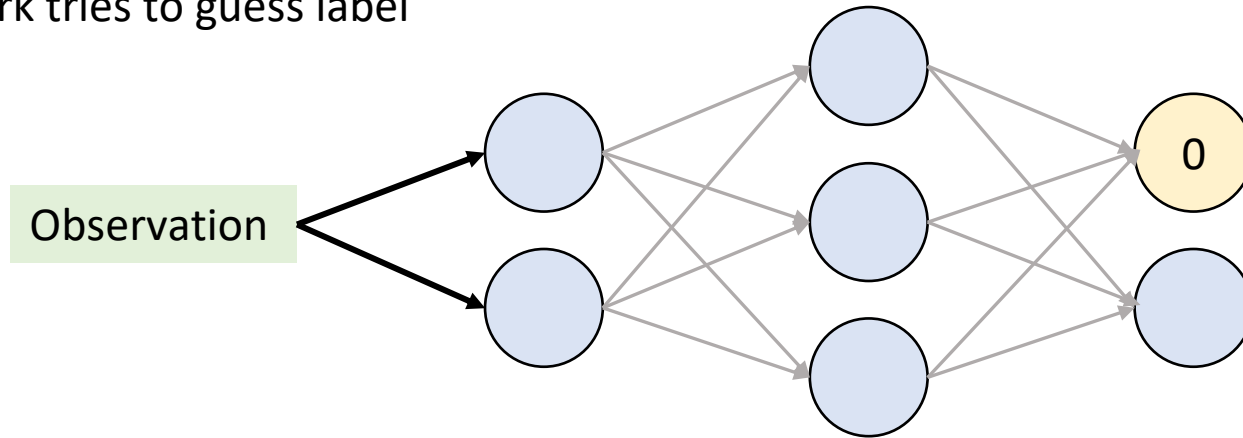
c. "Score" performance by generating a loss value.



# So far:

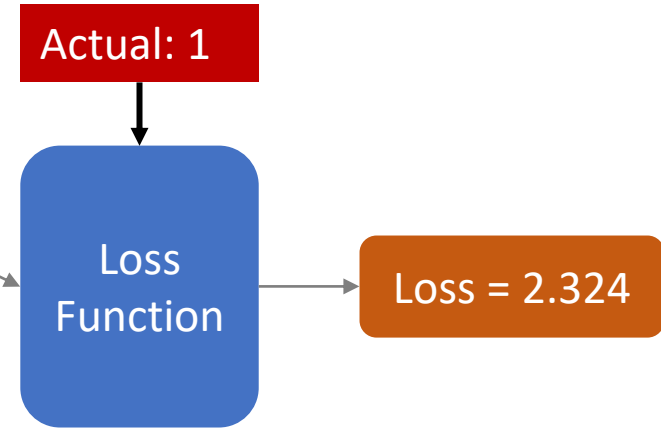
## 1. Forward Propagation

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



b. Network makes guess

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



## 3. Step

Adjust weights using those gradients

## 2. Backpropagation

Determine how each weight affects the loss by calculating partial derivative of loss w.r.t. each weight



# 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 ( $\alpha$ definition)

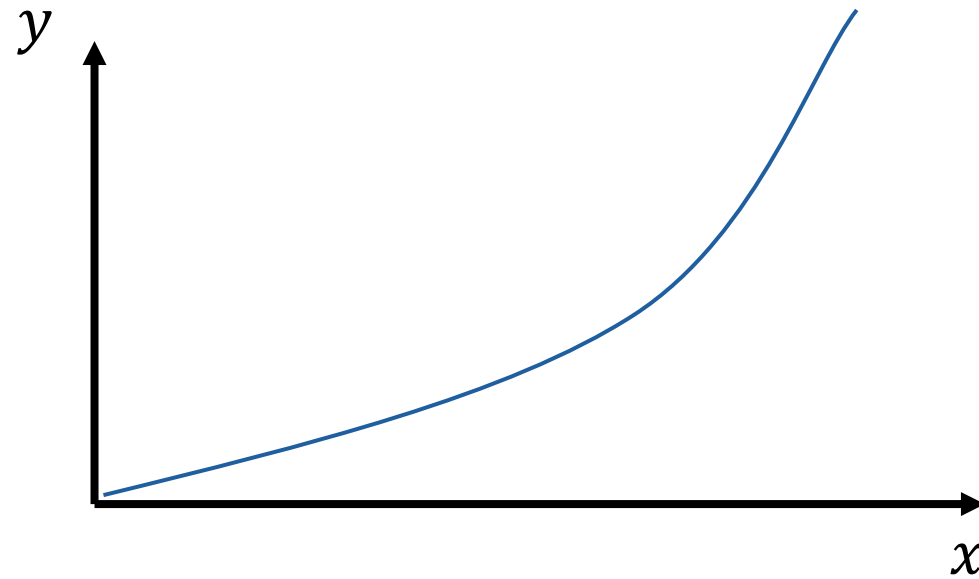
$$f(x) = y$$

$x$  and  $y$  are scalars

# Scalar Derivatives ( $\alpha$ definition)

$$f(x) = y$$

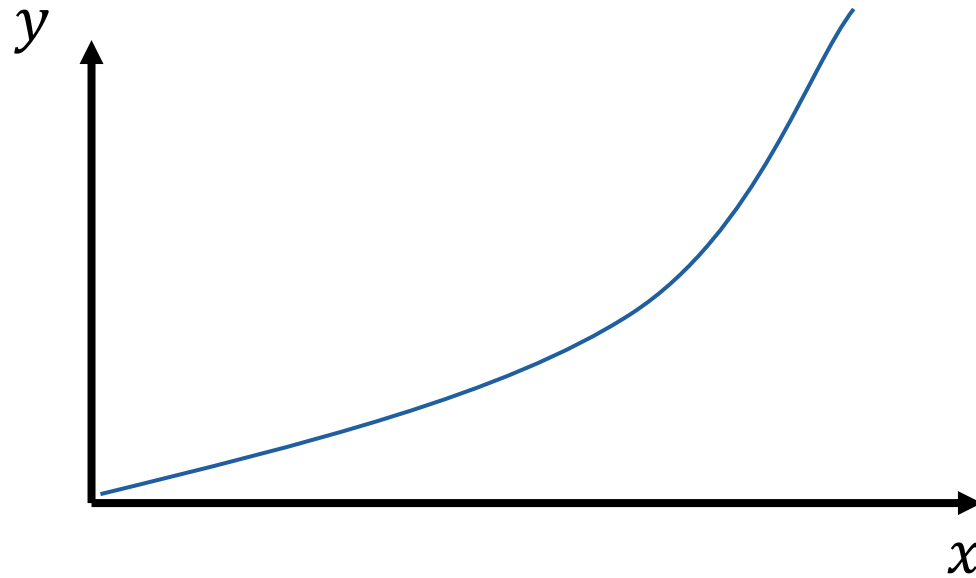
$x$  and  $y$  are scalars



# Scalar Derivatives ( $\alpha$ definition)

$$f(x) = y$$

$x$  and  $y$  are scalars

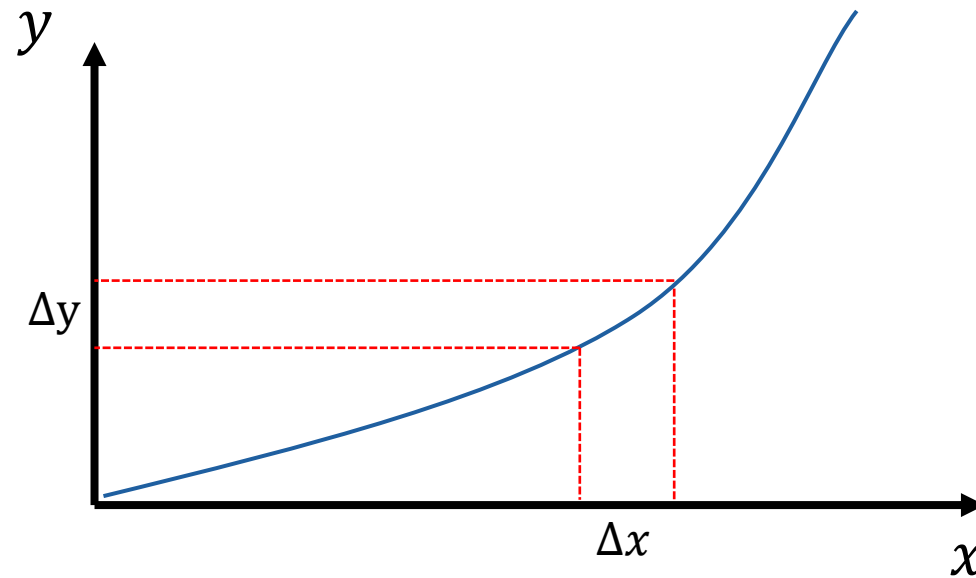


**Goal: determine how changing the input affects the output**

# Scalar Derivatives ( $\alpha$ definition)

$$f(x) = y$$

$x$  and  $y$  are scalars



**Goal: Find  $\Delta y$  given  $\Delta x$**

# Scalar Derivatives ( $\alpha$ definition)

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

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


- $\alpha$  is some factor multiplied to  $\Delta x$  that results in  $\Delta y$

# Scalar Derivatives ( $\alpha$ definition)

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

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

Derivative  $f'(x)$



- $\alpha$  is some factor multiplied to  $\Delta x$  that results in  $\Delta y$
- **Plot twist:**  $\alpha$  is the derivative  $f'(x)$



# 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  $\Delta x$  onto  $\Delta y$ .

$$f'(x): \Delta x \mapsto \Delta y$$

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

# 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  $\mathbf{x}$ .

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

$$= [a_1 \quad \dots \quad a_D] \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

Note: vectors are notated in bold and unitalicized font.

# Derivatives (vector in, scalar out)

Same thing, but in more familiar notation:

$$\begin{aligned}\Delta y &= \nabla_{\mathbf{x}} y \cdot \Delta \mathbf{x} \\ &= \left[ \frac{\partial y}{\partial x_1} \quad \cdots \quad \frac{\partial y}{\partial x_D} \right] \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix} \\ &\quad (1 \times D) \text{ row vector}\end{aligned}$$

# Derivatives (vector in, scalar out)

Same thing, but in more familiar notation:

This is the “full” derivative

$$\nabla_{\mathbf{x}}y = \frac{dy}{d\mathbf{x}}$$
$$\Delta y = \nabla_{\mathbf{x}}y \cdot \Delta \mathbf{x}$$
$$= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}$$

(1 x D) row vector

# Derivatives (vector in, scalar out)

In summary, for a function with  $(D \times 1)$  vector input  $\mathbf{x}$  and scalar output  $y$

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

Its derivative is a  $(1 \times D)$  row vector:

$$\nabla_{\mathbf{x}}y = \left[ \frac{\partial y}{\partial x_1} \quad \dots \quad \frac{\partial y}{\partial x_D} \right]$$

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

Why are the shape of derivatives transposed from input?

Recall:

$$\begin{aligned}\Delta y &= \nabla_{\mathbf{x}} y \cdot \Delta \mathbf{x} \\ &= \begin{bmatrix} \frac{\partial y}{\partial x_1} & \dots & \frac{\partial y}{\partial x_D} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix}\end{aligned}$$

By notational convention for dot products:

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

# Derivatives (vector in, vector out)

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

Input	Output		
$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix}$	$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_K \end{bmatrix}$	$\Rightarrow$	$\nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \dots & \frac{\partial y_K}{\partial x_D} \end{bmatrix}$
$D \times 1$	$K \times 1$		$K \times D$

# Derivatives (vector in, vector out)

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

Input                  Output

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

$D \times 1$                    $K \times 1$      $K \times D$

Each row is a vector-scalar derivative

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  $\nabla_{\mathbf{x}}y = \left[ \frac{\partial y}{\partial x_1} \quad \dots \quad \frac{\partial y}{\partial x_D} \right]$

3. Vector/vector derivative  $\nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \dots & \frac{\partial y_K}{\partial x_D} \end{bmatrix}$

## Key Ideas

- The derivative is the **best linear approximation** of  $f$  at a point
- The derivative is a **linear transformation** (matrix multiplication)
- The derivative describes **the effect of each input on the output**

# Backprop Interlude:

## Derivatives vs. Gradients

# But what is the gradient?

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

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

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

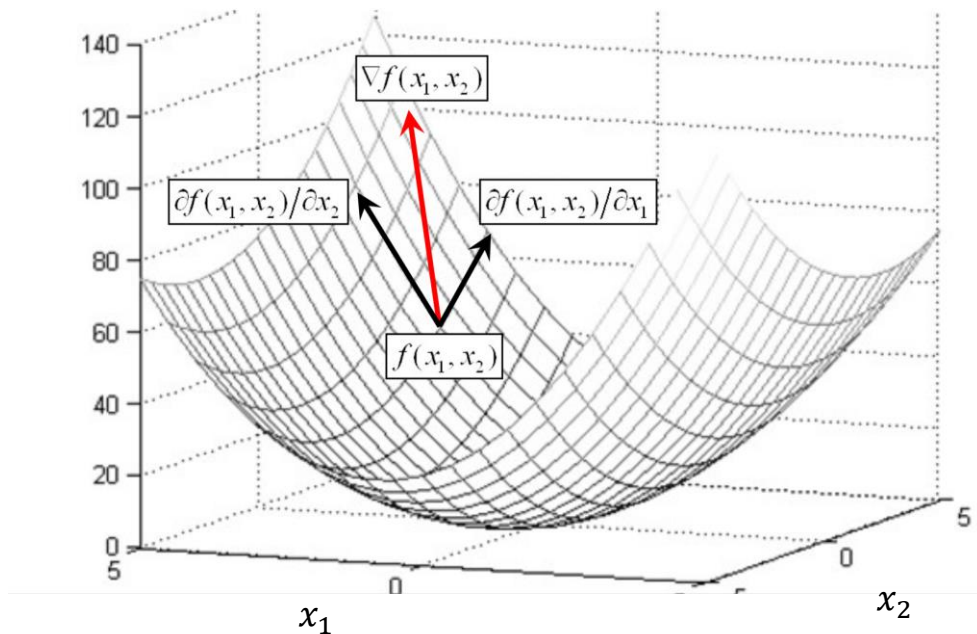
## **One difference: interpretation**

While the derivative projects change in input onto change in output, 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 vector, making it harder to interpret.

(^ 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**.
  - The length of the gradient vector  $\|\nabla f\|$  is the rate of increase in that direction



[img source](#)

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

# Backprop Interlude:

Full vs. Partial Derivatives & The Chain Rule

# Partial vs. Total Derivatives

$$\frac{dy}{d\mathbf{x}}$$

vs

$$\frac{\partial y}{\partial x_i}$$

- $\nabla_{\mathbf{x}}y$
- Total influence of  $\mathbf{x} = [x_1, \dots, x_i]$  on  $y$

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

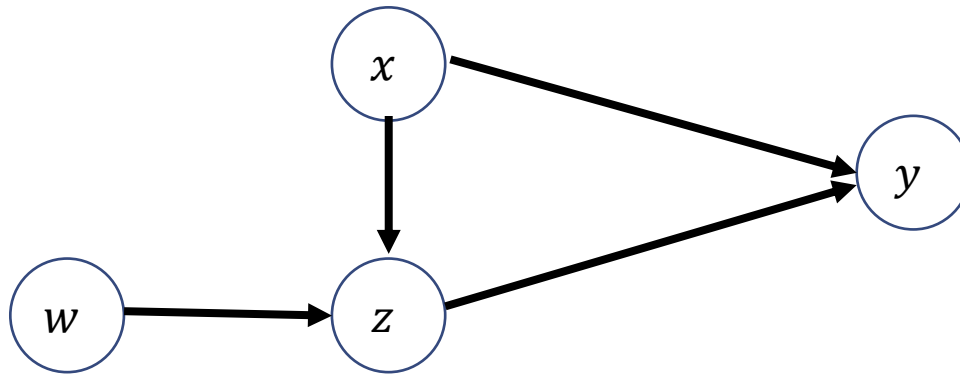
Once variables influence each other, it gets messy

# Things get messy

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

# Things get messy

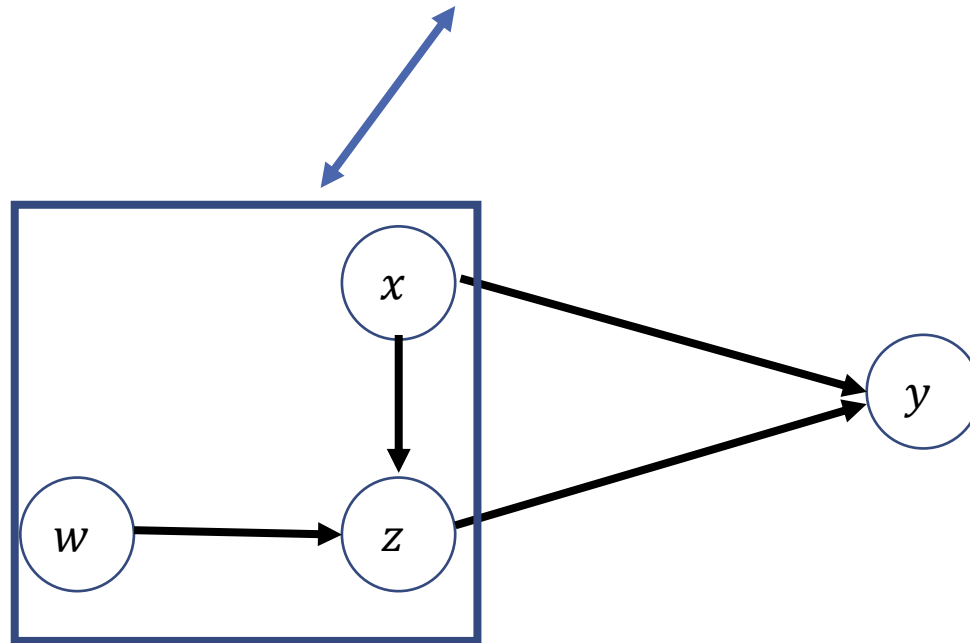
Find  $\frac{dy}{dx}$  for  $f(x, z) = y$ , where  $z = g(x, w)$





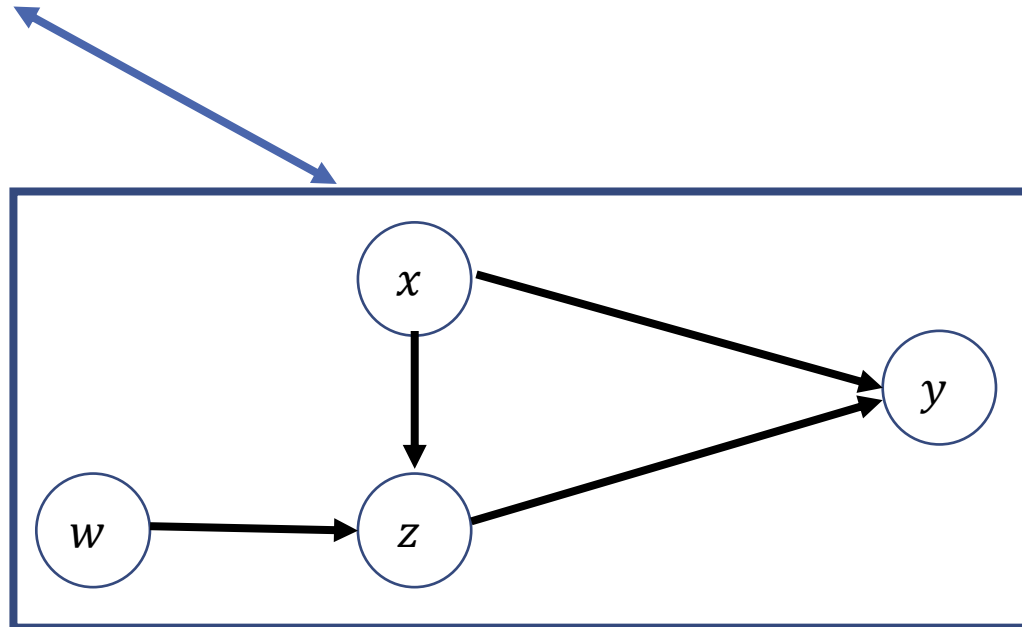
# Things get messy

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



# Things get messy

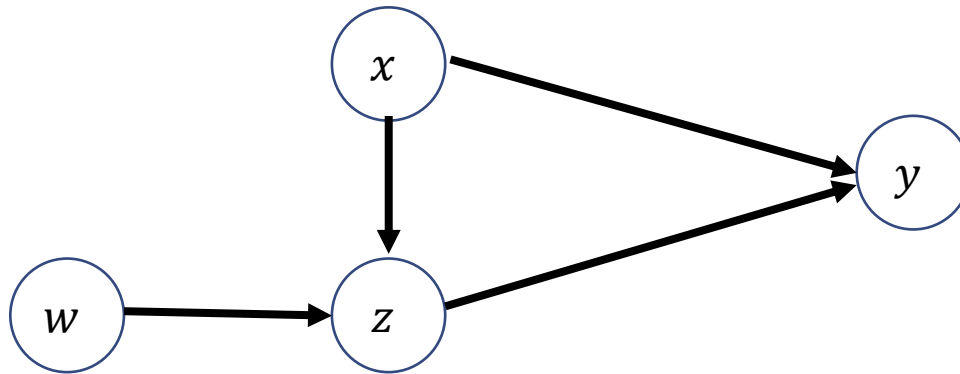
Find  $\frac{dy}{dx}$  for  $f(x, z) = y$ , where  $z = g(x, w)$



# Things get messy

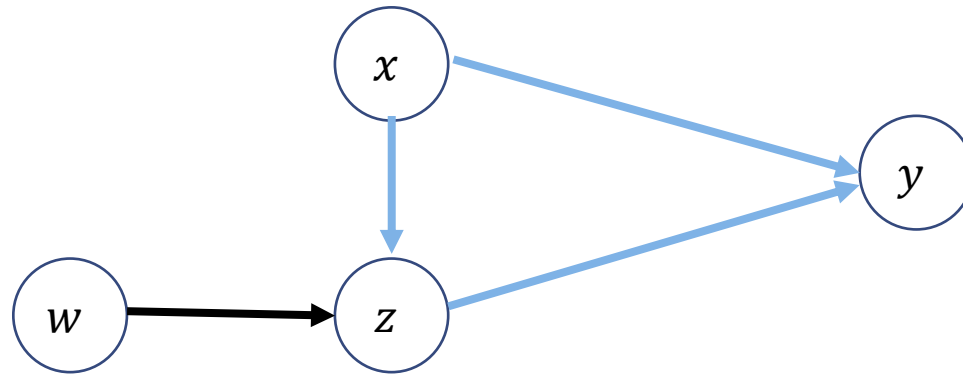
Find  $\frac{dy}{dx}$  for  $f(x, z) = y$ , where  $z = g(x, w)$

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



# Things get messy

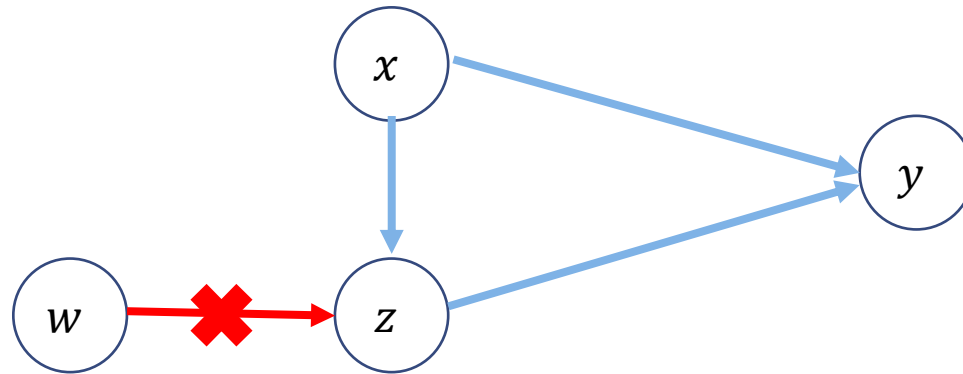
Find  $\frac{dy}{dx}$  for  $f(x, z) = y$ , where  $z = g(x, w)$



Goal: get only  $x$ 's influence on  $y$

# Things get messy

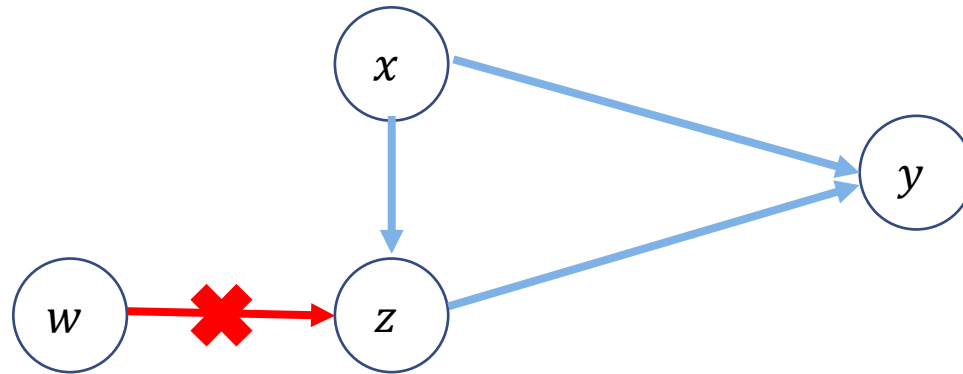
Find  $\frac{dy}{dx}$  for  $f(x, z) = y$ , where  $z = g(x, w)$



Can't do  $\frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}$  because  $z$  is influenced by  $w$

# Things get messy

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



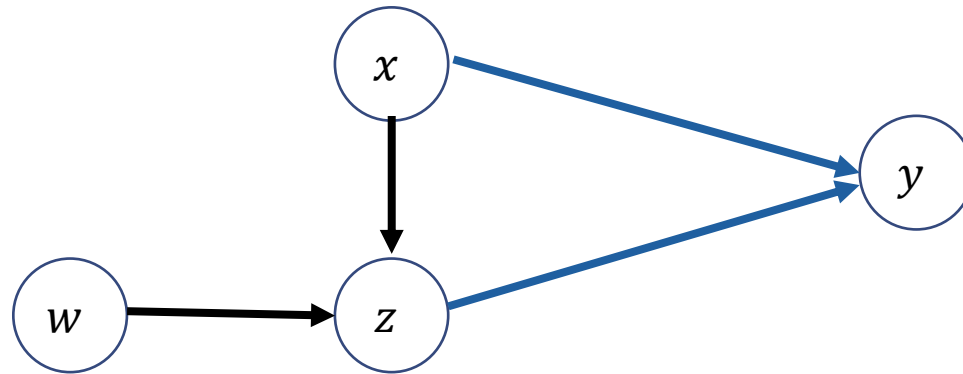
It's time for... ***“the chain rule”***

# 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

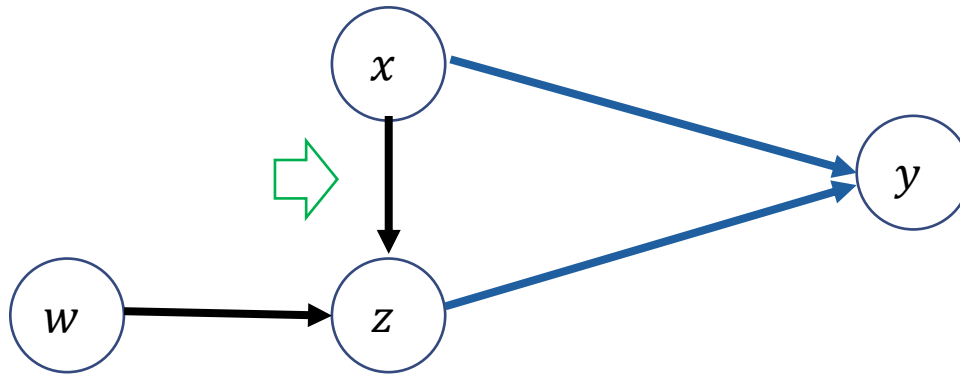


$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z}$$



# The Chain Rule

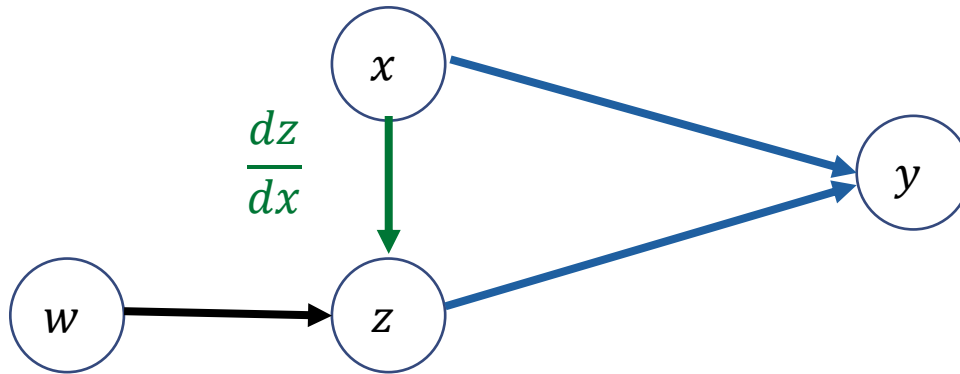
- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target



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

# The Chain Rule

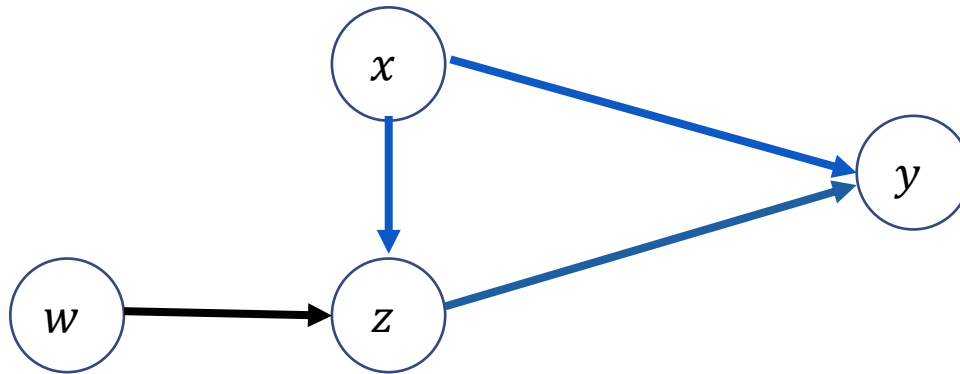
- The chain rule is used to properly **account for influences in nested functions**
  - Recursively calculates derivatives on nested functions w.r.t. target



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

# 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

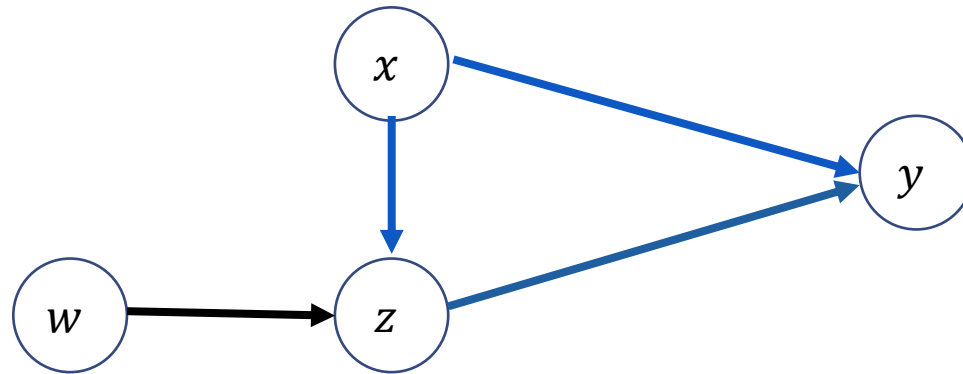


$$\frac{dy}{dx} = \frac{\partial y}{\partial x} + \frac{\partial y}{\partial z} \frac{dz}{dx}$$

Done!

# 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



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

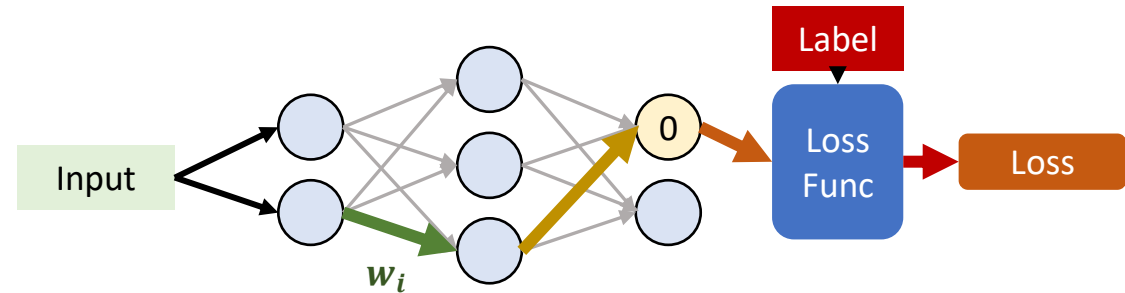
For NNs, we use this to isolate the influences of each weight matrix on the loss

# Backprop Conclusion:

Conclusion and a Note

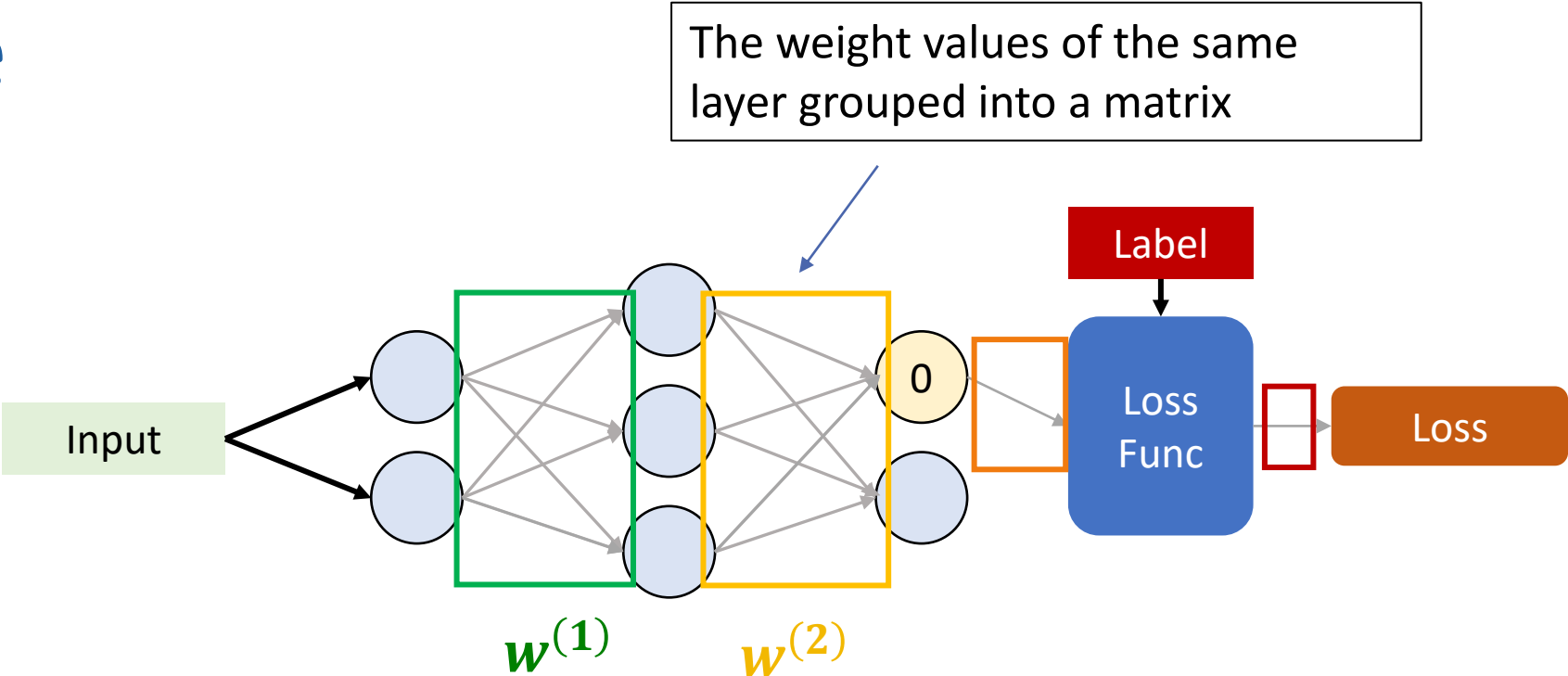
# Summary

- In backprop, our goal is to determine how each weight influences the loss
- To do so, we take a partial gradient of the loss w.r.t. each weight
  - This isolates each weight's influence on the loss
  - This requires the chain rule



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

# Note



**To be more precise,** the actual backprop calculates partials w.r.t. each weight matrix all at once. So the calculation looks more like:

$$\frac{\partial \text{Loss}}{\partial w^{(1)}} = \frac{\partial \text{Loss}}{\partial \text{LossFunc}} \cdot \frac{\partial \text{LossFunc}}{\partial \text{Guess}} \cdot \frac{\partial \text{Guess}}{\partial w^{(2)}} \cdot \frac{\partial w^{(2)}}{\partial w^{(1)}}$$

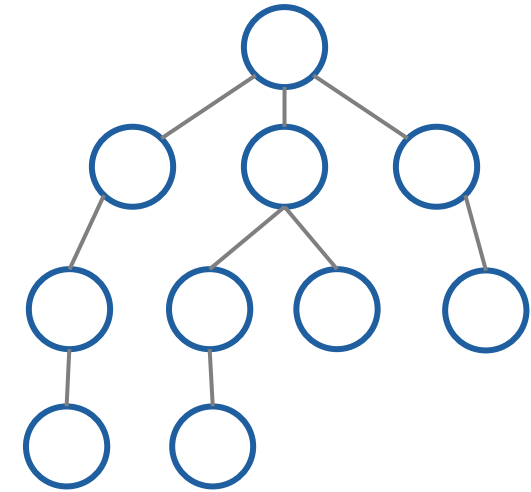
# HW1P1 Help & Tips

DFS and Recursion



# Depth-First Search (DFS)

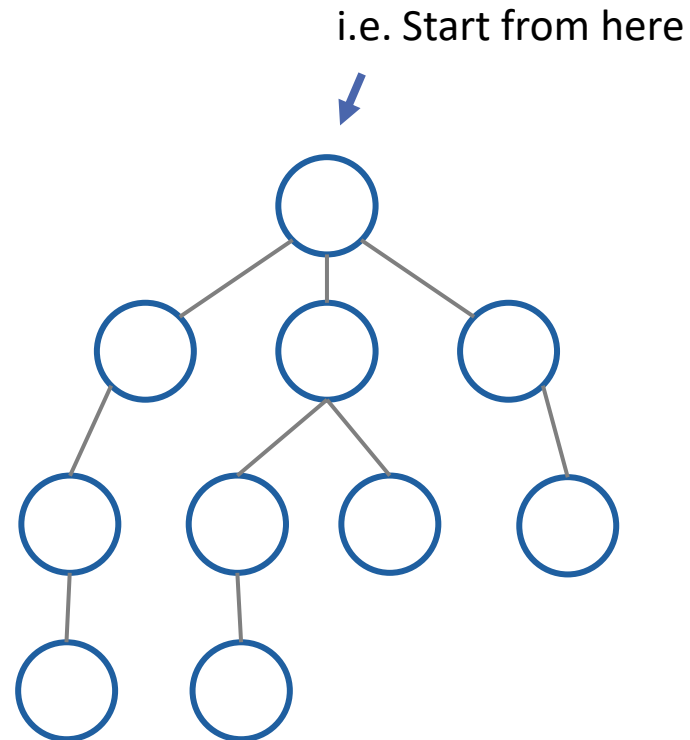
- We'll briefly cover DFS, as it's needed for autograd
- **Algorithm used to traverse nodes in trees/graph**
  - Anything with vertices/edges; directed or undirected



Example of a graph

# Depth-First Search (DFS)

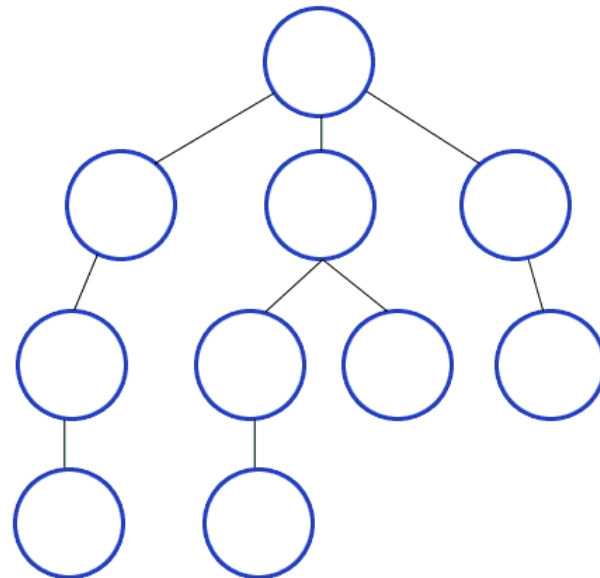
Goal: To visit every node in the graph, starting from some node



# Depth-First Search (DFS)

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

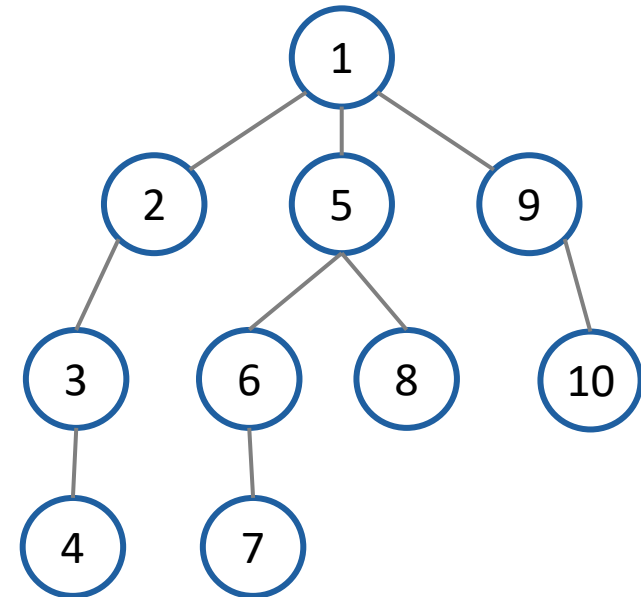
i.e. Start from here



[\(Animated GIF source\)](#)

# 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



# Recursion

- 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 stop the recursion
  - Otherwise it never stops 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  $\geq 3$
- 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
```

- Added some print statements so you can track when each line of code was executed

# 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
  - **Will need to do so in hw1p1**
- **Modifications (needed in hw1p1)**
  - 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))
```

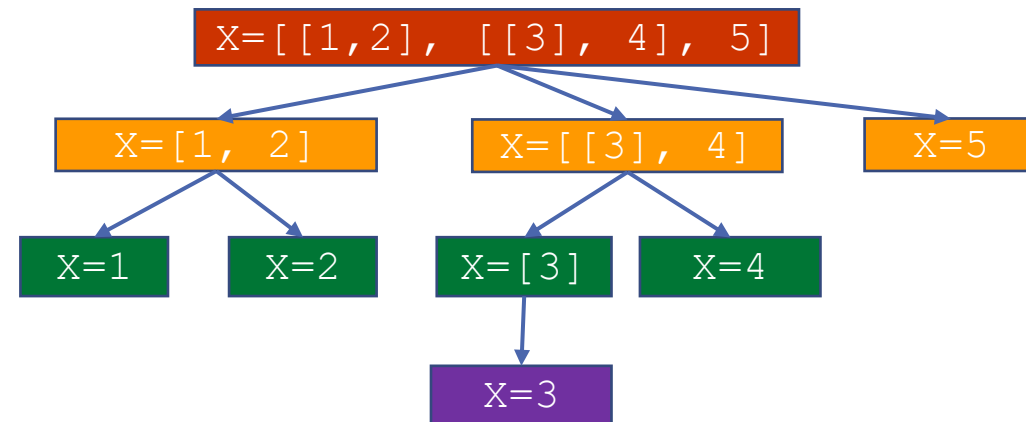
---

- At each recursive call, checks if input is a list
  - If so, it iterates through each item in the list
  - If not, base case. No return statement, but effectively returns None

# 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
```



Looks like a DFS....

# HW1P1 Help & Tips

## Matrix Operation Derivatives

# Matrix Operation Derivatives

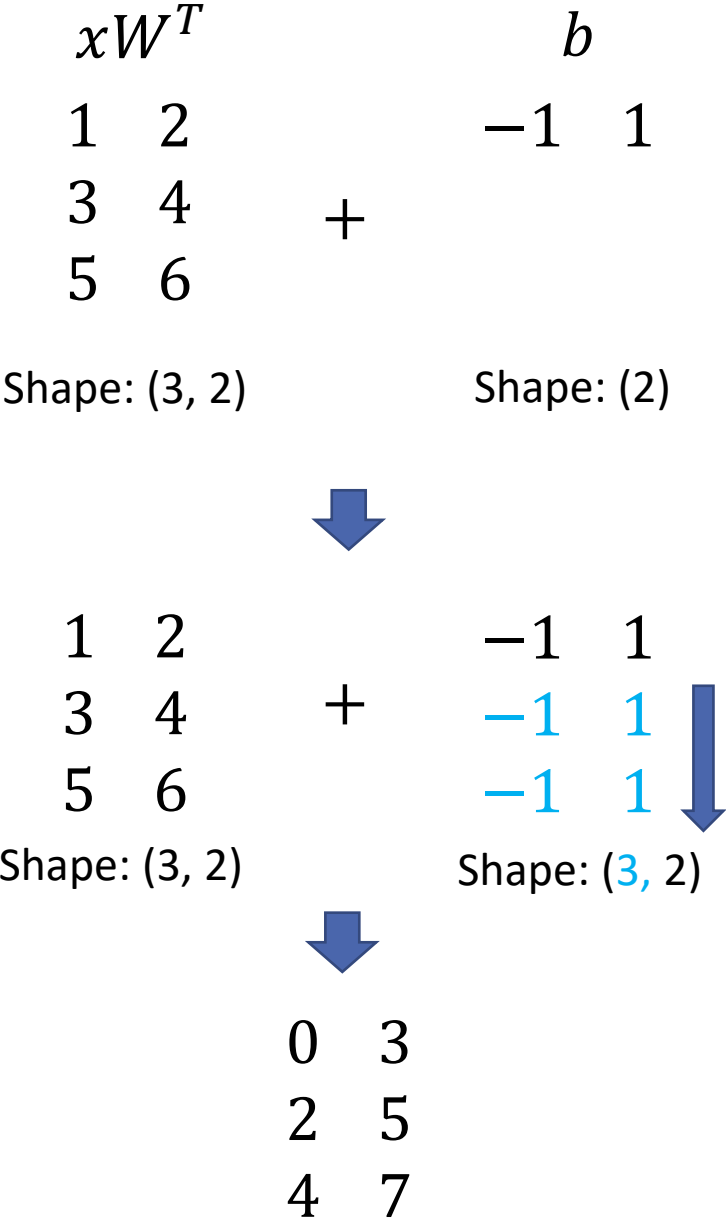
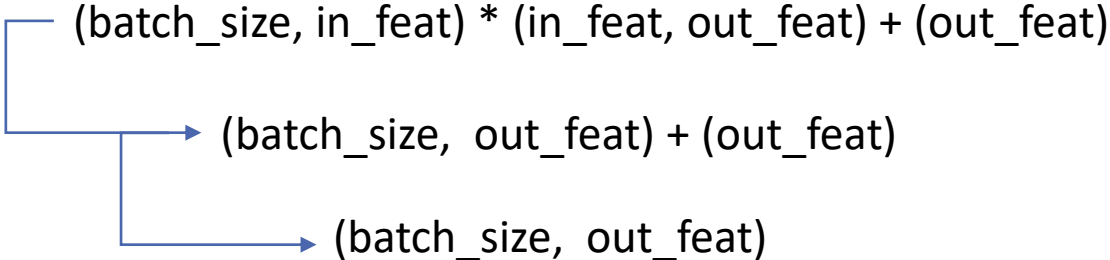
- Next, we'll give you the derivatives of a few matrix operations (in math)
- **You'll need to translate math to code, but warning: not all derivatives will translate neatly to your operations**
- But this should be a good starting point

**Advice:** When implementing a new operation, read the real Torch doc for it first. That'll tell you which parameters are worth implementing in `forward()` and saving in `ctx` for backward

# Broadcasting

- How Torch/NumPy handles operations between Tensors with different shapes
- You'll need to understand this for Problem 2.1: Linear Layer, which needs it in `Add`

$$\text{Linear}(x) = xW^T + b$$



# Broadcasting Tips

- Broadcasting in `forward()` is often handled by NumPy for you
- `backward()` is the challenge

**Advice:** In `functional.py`, implement a function:

```
unbroadcast(grad, shape, to_keep=0)
```

You can use this function in the backward of most operations to **undo broadcasting (hint: by summing)**.

Remember: this shouldn't add to the comp graph.

**Typical `backward()` for a broadcastable function**

1. Calculate gradients w.r.t. each input
2. Unbroadcast grads
3. Return unbroadcasted grads



# Broadcasting Backprop Example

```
>>> a = torch.tensor([[1.,2.],[3.,4.],[5.,6.]], requires_grad=True)
>>> b = torch.tensor([-1.,1.], requires_grad=True)
>>> c = (a+b).sum()
>>> c
tensor(21., grad_fn=<SumBackward0>)
>>> c.backward()
>>> a.grad
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
>>> b.grad
tensor([3., 3.]])
```

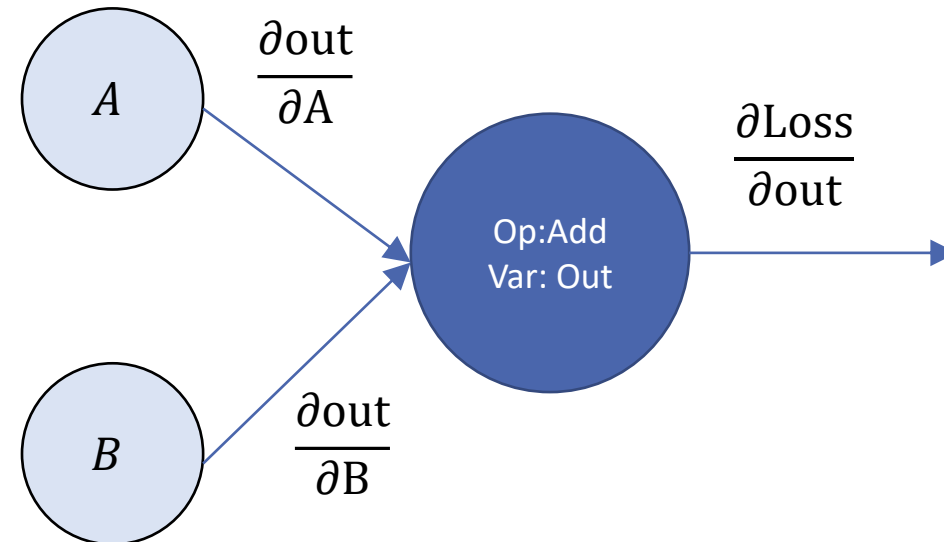
# Add (Function)

`torch.add(A, B)` = out, broadcastable tensor args

$$\frac{\partial \text{Loss}}{\partial A} = \frac{\partial \text{Loss}}{\partial \text{out}} \frac{\partial \text{out}}{\partial A}$$

$$\frac{\partial \text{Loss}}{\partial B} = \frac{\partial \text{Loss}}{\partial \text{out}} \frac{\partial \text{out}}{\partial B}$$

$\frac{\partial \text{out}}{\partial A}$  and  $\frac{\partial \text{out}}{\partial B}$  are tensors of ones



**Hint for Sub (Function):**  $A + (-B) = A - B$

**Hint for Div (Function):** Quotient Rule

**Hint for Mul (Function):** Page 11 on writeup

# Extra Resources

# 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:

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

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.

# 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](#)

# Broadcasting Resources

## Official Documentation

[Torch docs](#) and [NumPy docs](#)

## From Reshmi's Piazza Post (@262)

<https://machinelearningmastery.com/broadcasting-with-numpy-arrays/>

<https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html>

[https://www.pythonlikeyoumeanit.com/Module3\\_IntroducingNumpy/Broadcasting.html](https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/Broadcasting.html)

<https://stackoverflow.com/questions/51371070/how-does-pytorch-broadcasting-work>

# Other Resources

- [https://en.wikipedia.org/wiki/Matrix\\_calculus](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://en.wikipedia.org/wiki/Automatic_differentiation)
- <https://numpy.org/doc/stable/reference/routines.linalg.html>
  - NumPy's matrix operations documentation