

Calculus Refresher

CMPUT 261: Introduction to Artificial Intelligence

GBC §4.1, 4.3

Assignments

- **Assignment #2** is now available
 - Due **Oct 21/2025** (two weeks from today) at **11:59pm**
- **Assignment #1 marking**
 - There was a bug in question (2e) of the assignment; nobody who attempted this question will lose marks
 - Some marks were deducted for questions (1a) and (1b) that should not have been
 - Mark updates will be completed by next week

Lecture Outline

1. Recap
2. Gradient-based Optimization & Gradients
3. Numerical Issues

After this lecture, you should be able to:

- Apply the chain rule of calculus to functions of one or multiple arguments
- Explain the advantages and disadvantages of the method of differences
- Describe the numerical problems with softmax and how to solve them
- Explain why log probabilities are more numerically stable than probabilities

Loss Minimization

In supervised learning, we choose a **hypothesis** to **minimize** a **loss function** (+ bias)

Example: Predict the **temperature**

- *Dataset:* temperatures $y^{(i)}$ from a random sample of days
- *Hypothesis class:* Always predict the **same value** μ
- *Loss function:*

$$L(\mu) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu)^2$$

Optimization

Optimization: finding a value of x that **minimizes** $f(x)$

$$x^* = \arg \min_x f(x)$$

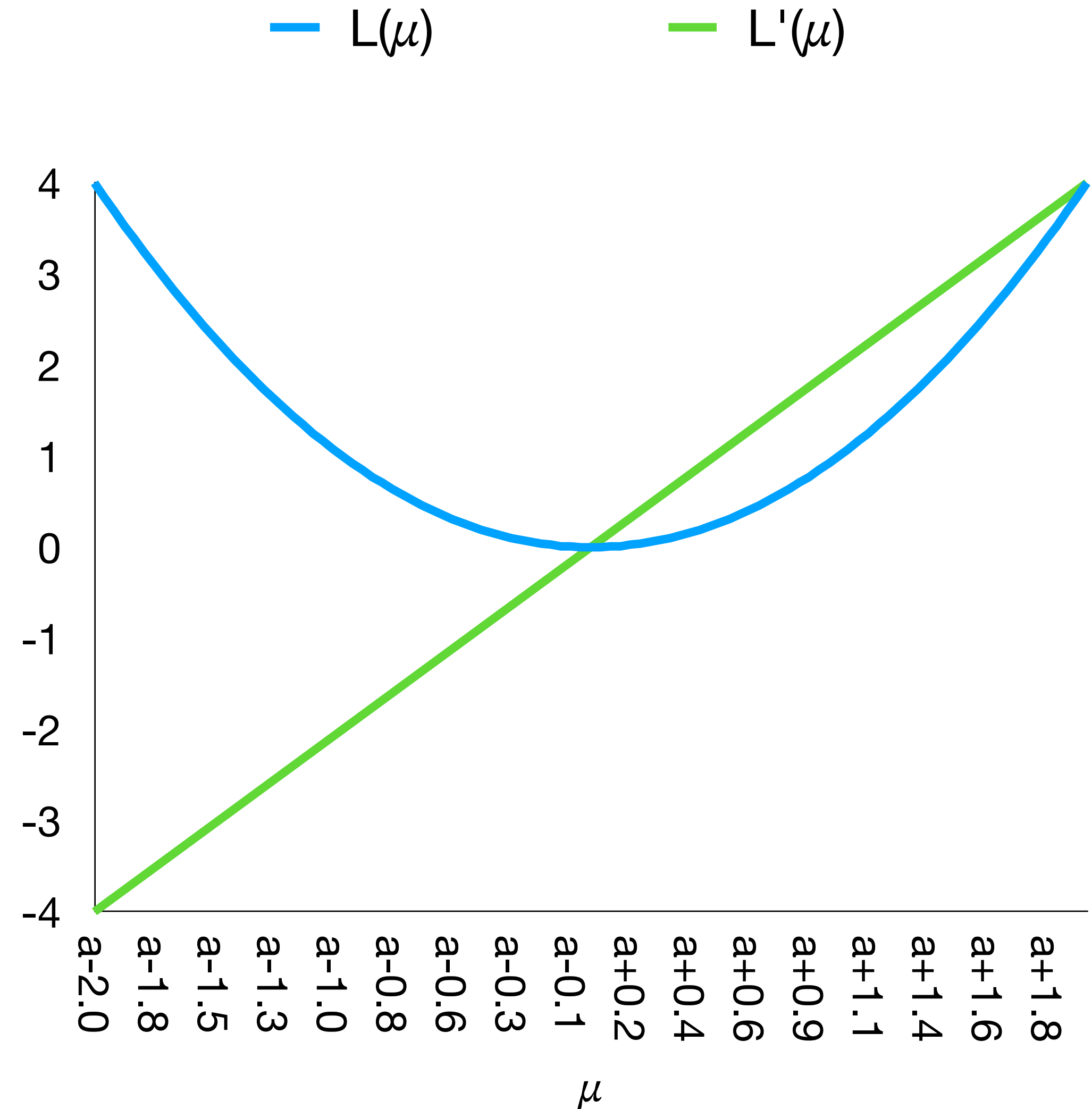
- Temperature example: Find μ that makes $L(\mu)$ small

Gradient descent: Iteratively move from current estimate in the direction that makes $f(x)$ **smaller**

- For **discrete** domains, this is just **hill climbing**:
Iteratively choose the **neighbour** that has minimum $f(x)$
- For **continuous** domains, neighbourhood is less well-defined

Derivatives

- The **derivative** $f'(x) = \frac{d}{dx}f(x)$ of a function $f(x)$ is the **slope** of f at point x
- When $f'(x) > 0$, f **increases** with small enough increases in x
- When $f'(x) < 0$, f **decreases** with small enough increases in x



Multiple Inputs

Example:

Predict the temperature **based on** pressure and humidity

- *Dataset:*

$$\left(x_1^{(1)}, x_2^{(1)}, y^{(1)}\right), \dots, \left(x_1^{(m)}, x_2^{(m)}, y^{(m)}\right) = \left\{(\mathbf{x}^{(i)}, y^{(i)}) \mid 1 \leq i \leq m\right\}$$

- *Hypothesis class:* **Linear regression**: $h(\mathbf{x}; \mathbf{w}) = w_0 + w_1x_1 + w_2x_2$

- *Loss function:*

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - h(\mathbf{x}^{(i)}; \mathbf{w})\right)^2$$

Partial Derivatives

Partial derivatives: How much does $f(\mathbf{x})$ change when we **only change one** of its inputs x_i ?

- Can think of this as the derivative of a **conditional** function $g(z) = f(x_1, \dots, \mathbf{z}, \dots, x_n)$:

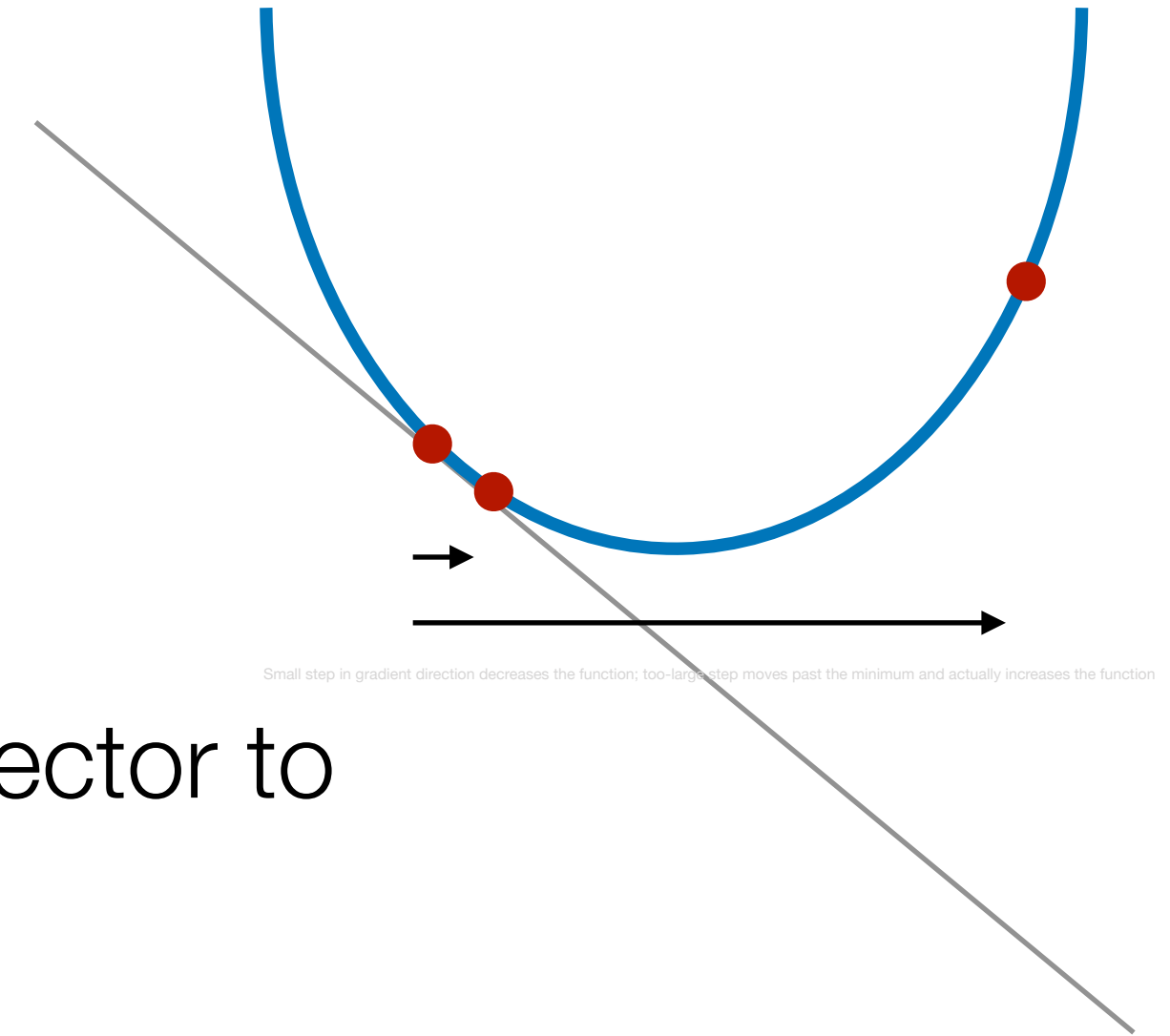
$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \frac{d}{dx_i} g(x_i).$$

Gradient

- The **gradient** of a function $f(\mathbf{x})$ is just a **vector** that contains all of its **partial derivatives**:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{bmatrix}$$

Gradient Descent



- The gradient of a function tells how to change every element of a vector to **increase** the function
 - If the partial derivative of x_i is positive, increase x_i
- **Gradient descent:**
Iteratively choose new values of $\mathbf{x}$ in the (opposite) direction of the gradient:

$$\mathbf{x}^{new} = \mathbf{x}^{old} - \eta \nabla f(\mathbf{x}^{old}) .$$

- This only works for **sufficiently small** changes (**why?**)
- **Question:** How much should we change $\mathbf{x}^{old}$? learning rate

Where Do Gradients Come From?

Question: How do we compute the gradients we need for gradient descent?

1. Analytic expressions / direct derivation
2. Method of differences
3. The Chain Rule (of Calculus)

1. Analytic Expressions: 1D Derivatives

$$\begin{aligned} L(\mu) &= \frac{1}{n} \sum_{i=1}^n (y(i) - \mu)^2 \\ &= \frac{1}{n} \sum_{i=1}^n \left[y(i)^2 - 2y(i)\mu + \mu^2 \right] \end{aligned}$$

$$\frac{d}{d\mu} L(\mu) = \frac{1}{n} \sum_{i=1}^n \left[-2y(i) + 2\mu \right]$$

Analytic Expressions: Multiple Arguments

To analytically find the **gradient** of a multi-input function, find the **partial derivative** for each of the inputs (and then collect in a vector).

$$\begin{aligned} L(\mathbf{w}) &= \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} \right)^2 \\ &= \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - w_1 x_1^{(i)} - w_2 x_2^{(i)} \right)^2 \\ &= \frac{1}{n} \sum_{i=1}^n w_1^2 x_1^{(i)2} + 2w_1 w_2 x_1^{(i)} x_2^{(i)} - 2w_1 x_1^{(i)} y + w_2^2 x_2^{(i)2} - 2w_2 x_2^{(i)} y + y^2 \end{aligned}$$

Analytic Expressions: Multiple Arguments

To analytically find the **gradient** of a multi-input function, find the **partial derivative** for each of the inputs (and then collect in a vector).

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n w_1^2 x_1^{(i)2} + 2w_1 w_2 x_1^{(i)} x_2^{(i)} - 2w_1 x_1^{(i)} y + w_2^2 x_2^{(i)2} - 2w_2 x_2^{(i)} y + y^2$$

$$\frac{\partial}{\partial w_1} L(w_1, w_2) = \frac{1}{n} \sum_{i=1}^n 2w_1 x_1^{(i)2} + 2w_2 x_1^{(i)} x_2^{(i)} - 2x_1^{(i)} y$$

$$\frac{\partial}{\partial w_2} L(w_1, w_2) = \frac{1}{n} \sum_{i=1}^n 2w_2 x_2^{(i)2} - 2w_1 x_1^{(i)} x_2^{(i)} + 2x_2^{(i)} y$$

Analytic Expressions: Multiple Arguments

To analytically find the **gradient** of a multi-input function, find the **partial derivative** for each of the inputs (and then collect in a vector).

$$\nabla L(w_1, w_2) = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum_{i=1}^n 2w_1 x_1^{(i)2} + 2w_2 x_1^{(i)} x_2^{(i)} - 2x_1^{(i)} y \\ \frac{1}{n} \sum_{i=1}^n 2w_2 x_2^{(i)2} - 2w_1 x_1^{(i)} x_2^{(i)} + 2x_2^{(i)} y \end{bmatrix}$$

2. Method of Differences

$$\frac{\partial}{\partial w_i} L(\mathbf{w}) \approx L(\mathbf{w} + \epsilon \mathbf{e}_i) - L(\mathbf{w})$$

Vector of 0's with a 1 in i -th position

e.g., $\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

(for "sufficiently" tiny ϵ)

Question: Why would we ever do this?

Question: What are the drawbacks?

3. Chain Rule (of Calculus): 1D Derivatives

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

$$\text{i.e., } h(x) = f(g(x)) \implies h'(x) = f'(g(x))g'(x)$$

- If we know formulas for the derivatives of **components** of a function, then we can build up the derivative of their composition mechanically
- Most prominent example: **Back-propagation** in neural networks

Chain Rule (of Calculus): Multiple Intermediate Arguments

What if $h(x) = f(g_1(x), g_2(x))$?

$$\frac{dh}{dx} = \frac{\partial f}{\partial g_1} \frac{dg_1}{dx} + \frac{\partial f}{\partial g_2} \frac{dg_2}{dx}$$

Question: Why do we **add** the partials via the two arguments?

(*) Chain Rule (of Calculus): Multiple Arguments

For multiple arguments, things look more complicated, but it's the same idea:

$$h(w_1, w_2) = f(g_1(w_1, w_2), g_2(w_1, w_2))$$

$$\nabla h(w_1, w_2) = \begin{bmatrix} | & | \\ \nabla_{\mathbf{w}} g_1(w_1, w_2) & \nabla_{\mathbf{w}} g_2(w_1, w_2) \\ | & | \end{bmatrix} \nabla_{g(\mathbf{w})} f(g_1(w_1, w_2), g_2(w_1, w_2))$$

$$= \begin{bmatrix} \frac{\partial g_1(w_1, w_2)}{\partial w_1} & \frac{\partial g_2(w_1, w_2)}{\partial w_1} \\ \frac{\partial g_1(w_1, w_2)}{\partial w_2} & \frac{\partial g_2(w_1, w_2)}{\partial w_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_1(w_1)} \\ \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_2(w_2)} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_1(w_1)} \frac{\partial g_1(w_1, w_2)}{\partial w_1} + \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_2(w_2)} \frac{\partial g_2(w_1, w_2)}{\partial w_1} \\ \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_1(w_1)} \frac{\partial g_1(w_1, w_2)}{\partial w_2} + \frac{\partial f(g_1(w_1, w_2), g_2(w_1, w_2))}{\partial g_2(w_2)} \frac{\partial g_2(w_1, w_2)}{\partial w_2} \end{bmatrix}$$

Approximating Real Numbers

- Computers store real numbers as **finite number** of bits
- **Problem:** There are an **infinite number** of real numbers in any interval
- Real numbers are encoded as **floating point numbers**:
 - $$\underbrace{1.001\dots011011}_{\text{significand}} \times 2^{\underbrace{1001\dots0011}_{\text{exponent}}}$$
 - *Single precision:* 24 bits **significand**, 8 bits **exponent**
 - *Double precision:* 53 bits significand, 11 bits exponent
- **Deep learning** often uses single precision!

Underflow

$$\underbrace{1.001\dots011010}_{\text{significand}} \times 2^{\underbrace{1001\dots0011}_{\text{exponent}}}$$

- Positive numbers that are smaller than $1.00\dots01 \times 2^{-1111\dots1111}$ will be rounded down to **zero**
 - Negative numbers that are bigger than $-1.00\dots01 \times 2^{-1111\dots1111}$ will be **rounded up to zero**
- Sometimes that's okay! (Almost every number gets rounded)
- Often it's not (**when?**)
 - Denominators: causes divide-by-zero
 - log: returns -inf
 - log(negative): returns nan

Overflow

$$\underbrace{1.001\dots 011010}_{\text{significand}} \times 2^{\underbrace{1001\dots 0011}_{\text{exponent}}}$$

- Numbers bigger than $1.111\dots 1111 \times 2^{1111}$ will be rounded up to **infinity**
- Numbers smaller than $-1.111\dots 1111 \times 2^{1111}$ will be rounded down to **negative infinity**
- **exp** is used very frequently
 - Underflows for very negative inputs
 - Overflows for "large" positive inputs
 - **89** counts as "large"

Addition/Subtraction

$$\underbrace{1.001\dots011010}_{\text{significand}} \times 2^{\underbrace{1001\dots0011}_{\text{exponent}}}$$

- Adding a small number to a large number can have no effect (**why?**)

Example:

```
>>> A = np.array([0., 1e-8])
```

```
>>> A = np.array([0., 1e-8]).astype('float32')
```

```
>>> A.argmax()
```

```
1
```

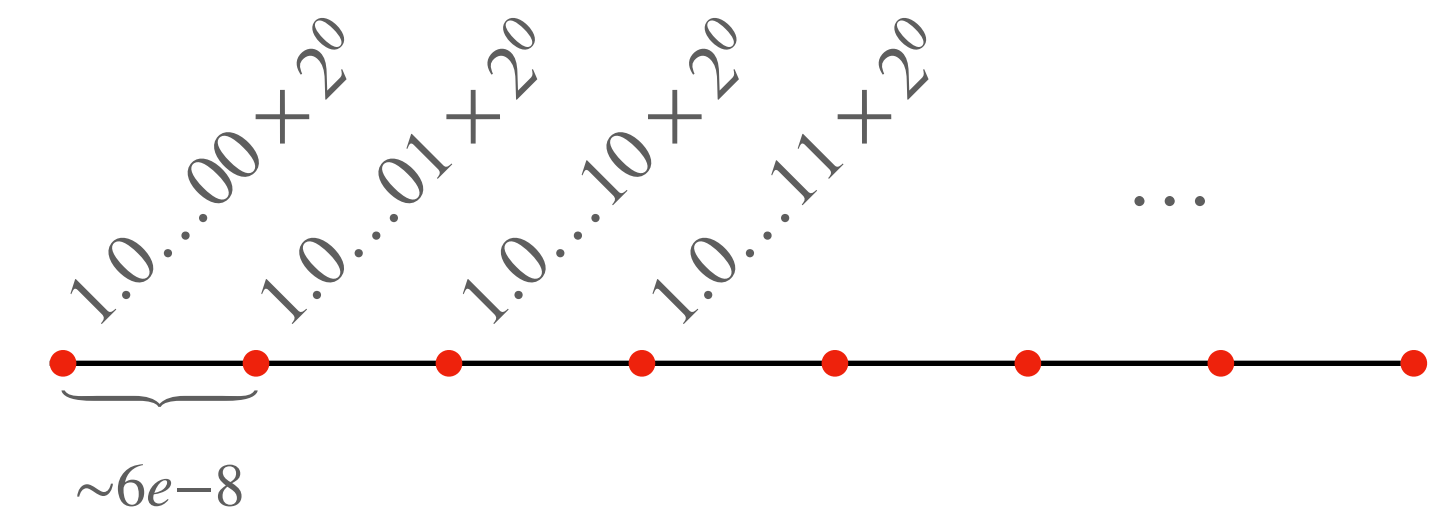
```
>>> (A + 1).argmax()
```

```
0
```

```
>>> A+1
```

```
array([1., 1.], dtype=float32)
```

1e-8 is **not** the
smallest possible
float32



$$2^{-24} \approx 5.9 \times 10^{-8}$$

Softmax

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}$$

- **Softmax** is a very common function
- Used to convert a vector of activations (i.e., numbers) into a **probability distribution**
 - **Question:** Why not normalize them directly without **exp**?
- But **exp overflows** very quickly:
 - Solution: $\text{softmax}(\mathbf{z})$ where $\mathbf{z} = \mathbf{x} - \max_j x_j$

Log

- Dataset likelihoods shrink **exponentially** quickly in the **number of datapoints**

- **Example:**

- Likelihood of a sequence of 5 fair coin tosses = $2^{-5} = 1/32$
- Likelihood of a sequence of 100 fair coin tosses = 2^{-100}

- **Solution:** Use log-probabilities instead of probabilities

$$\log(p_1 p_2 p_3 \dots p_n) = \log p_1 + \dots + \log p_n$$

- log-prob of 1000 fair coin tosses is $1000 \log 0.5 \approx -693$

General Solution

- **Question:**
What is the most general solution to numerical problems?
- ***Standard libraries***
 - PyTorch, Theano, Tensorflow, etc. **detect** common unstable expressions
 - scipy, numpy have stable implementations of many common patterns (e.g., softmax, logsumexp, sigmoid)

Summary

- **Gradients** are just vectors of **partial derivatives**
 - Gradients point "uphill"
- **Chain Rule of Calculus** lets us compute derivatives of function compositions using derivatives of simpler functions
- **Learning rate** controls how fast we walk uphill
- Deep learning is fraught with **numerical** issues:
 - Underflow, overflow, magnitude mismatches
 - Use **standard implementations** whenever possible