Calculus Refresher

CMPUT 366: Intelligent Systems

GBC §4.1, 4.3

Logistics

- Assignment #2 due **Monday, Feb 28 at 11:59pm**
  - Submit via eClass
- Next week is **reading week**
  - No lectures
  - No lab
- *After* reading week (**Mon, Feb 28**), lectures will be **in person**
  - CCIS L1-160

Lecture Outline

1. Recap
2. Hard-to-Sample Distributions
3. Gradient-based Optimization
4. Numerical Issues

Recap: Bayesian Learning

- In Bayesian Learning, we learn a **distribution** over models instead of a **single model**
- **Model averaging** to compute predictive distribution
- **Prior** can encode **bias** over models (like regularization)
- **Conjugate** models: can compute everything analytically

Recap: Monte Carlo

- Often we **cannot directly estimate** expectations from our model
  - Example: non-conjugate Bayesian models
- **Monte Carlo estimates:** Use a random sample from the distribution to estimate expectations by sample averages

1. Use an easier-to-sample **proposal** distribution instead
   - Rejection sampling
   - Importance sampling
2. Sample parts of the model **sequentially**

Forward Sampling in a Belief Network

- Sometimes we know how to sample **parts** of a large joint distribution in terms of other parts
  - E.g., belief networks: $P(X, Y, Z) = P(X)P(Y)P(Z \mid X, Y)$
  - We might be able to **directly** sample from each **conditional distribution** but not from the **joint distribution**
- Forward sampling:
  1. **Select** an ordering of variables consistent with the factoring
  2. **Repeat** until enough samples generated:
     For each variable X in the ordering:
     Sample $x_i \sim P(X \mid pa(X))$

Particle Filtering

- **Forward sampling** generates a value for each variable, then moves on to the next sample
- **Particle filtering** swaps the order:
  - Generate n values for variable X, then n values for variable Y, etc.
  - Especially useful when there is no fixed number of variables (e.g., in sequential models)
- Each sample is called a **particle**. Update its **weight** each time a value is sampled.
- Periodically **resample** from the particles with replacement, resetting weights to 1
  - High-probability particles likely to be **duplicated**
  - Low-probability particles likely to be **discarded**
- Resampling means the particles cover the distribution better

Rejection Sampling with Propositions

- How do we condition on some **propositional evidence** α?
- Repeat until enough samples accepted

e.g., $\alpha(x) = (x_1 > 0 \wedge x_4 \leq 12)$

1. **Sample** x from the **full joint distribution**
   (e.g., using **forward sampling** or **particle sampling**)

2. **If** $\alpha(x)$, then **accept** x
   Else reject

- Another view of this procedure:

1. **Approximate** the full joint distribution

2. **Condition** on evidence α

Loss Minimization

In supervised learning, we choose a **hypothesis** to **minimize** a **loss function**

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** $f(x)$

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

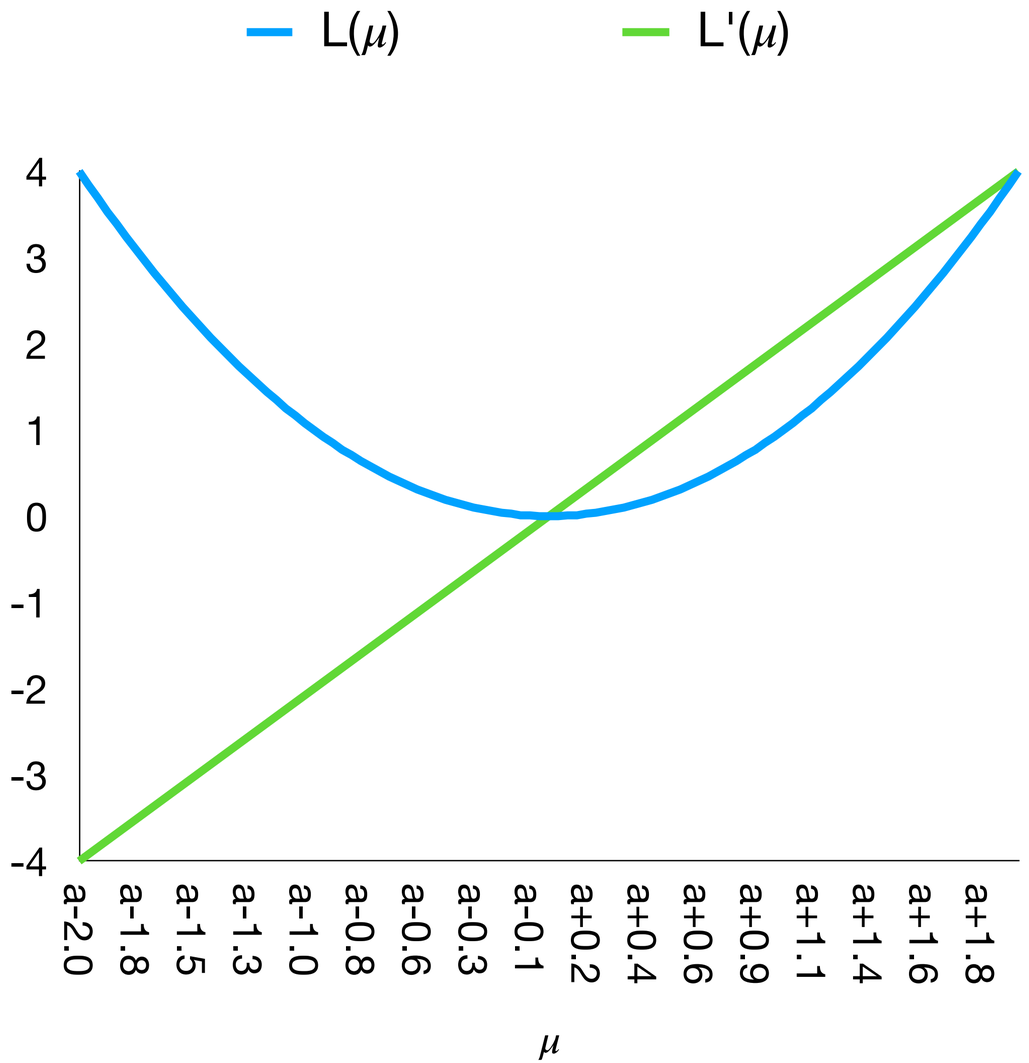
- Temperature example: Find μ that makes $L(\mu)$ small

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

- *Hypothesis class:* **Linear regression**: $h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

- *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 $g(x_i) = f(x_1, \ldots, \mathbf{x_i}, \ldots, x_n)$:

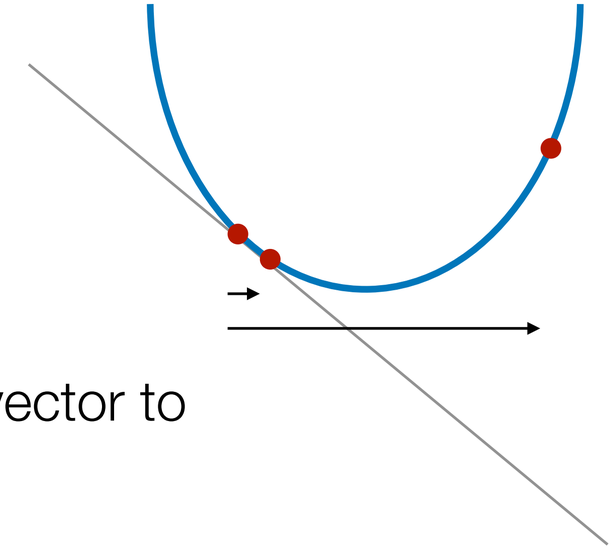
$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \frac{d}{dx_i} g(x_i).$$

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 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 implementation:

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

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

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

Where Do Gradients Come From?

2. Method of differences

$$\nabla L(\mathbf{x})_i \approx L(\mathbf{x} + \epsilon \mathbf{e}_i) - L(\mathbf{x})$$

(for "sufficiently" tiny ϵ)

Question: Why would we ever do this?

Question: What are the drawbacks?

Where Do Gradients Come From?

3. The Chain Rule (of Calculus)

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

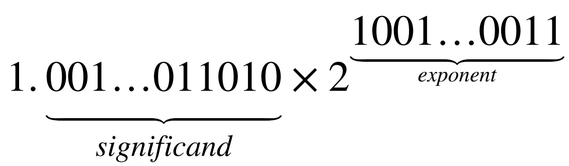
i.e., $h(x) = f(g(x)) \implies h'(x) = f'(g(x))g'(x)$

- 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

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**:
  - $1.\underbrace{001...011011}_{significand} \times 2^{\underbrace{1001..0011}_{exponent}}$
  - *Single precision:* 24 bits **significand**, 8 bits **exponent**
  - *Double precision:* 53 bits significand, 11 bits exponent
- **Deep learning** typically uses single precision!

Underflow



- Positive numbers that are smaller than 1.00...01 × $2^{-1111...1111}$ will be rounded down to **zero**
  - Negative numbers that are bigger than -1.00...01 × $2^{-1111...1111}$ 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

$$1.\underbrace{001\ldots011010}_{significand} \times 2^{\overbrace{1001\ldots0011}^{exponent}}$$

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

Addition/Subtraction

$$1.\underbrace{001\ldots011010}_{significand} \times 2^{\overbrace{1001\ldots0011}^{exponent}}$$

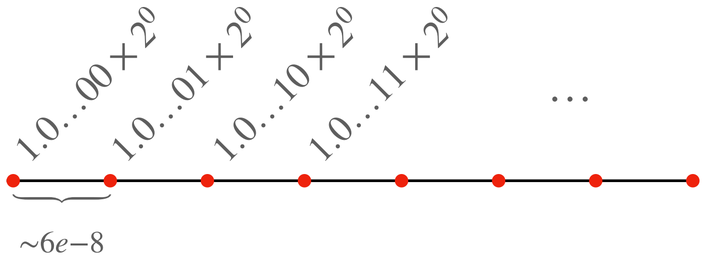
- Adding a small number to a large number can have no effect (**why**?)

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

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

- **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: $softmax(\mathbf{z})$ 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

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

- 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***

  - Theano, Tensorflow both **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"
- **Learning rate** controls how fast we walk uphill
- Deep learning is fraught with **numerical** issues:
  - Underflow, overflow, magnitude mismatches
  - Use **standard implementations** whenever possible