Course Evaluations

1. **More examples**
   - This was the top request
2. **Visuals/diagrams**
3. **Extra resources**
   - Problem sets
   - Content from the the web

Course Evaluations

4. **Too fast**
    - topics seem to get left behind pretty fast
    - topics build on each other; easy to get lost in the middle
5. **Recaps appreciated**
6. **Bigger fonts please**
7. **Please go over code part of the assignment in lecture**

Going Forward

1. **Example** at start of every lecture
2. At least one **diagram** for visual learners
3. Fonts: More willing to split over slides
4. Code walkthrough **in labs**

Calculus Refresher

CMPUT 366: Intelligent Systems

GBC §4.1, 4.3

Lecture Outline

1. Midterm course evaluations
2. Recap
3. Gradient-based optimization
4. Overflow and underflow

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** probabilities or 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
2. Sample parts of the model **sequentially**

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(μ) 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)

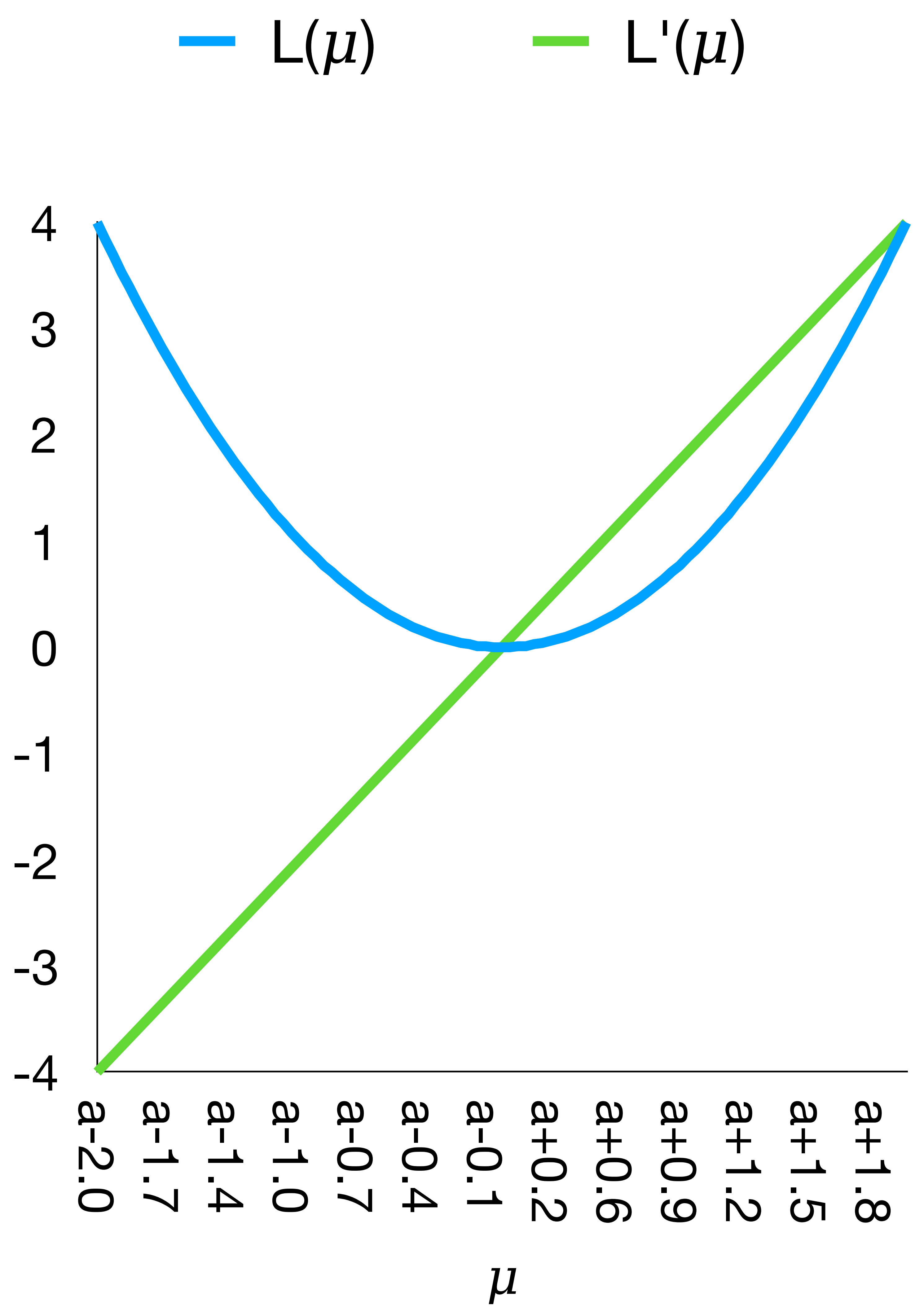
- For **continuous** domains, neighbourhood is less well-defined

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:* $(x_1^{(1)}, x_2^{(1)}, y^{(1)}), \ldots, (x_1^{(m)}, x_2^{(m)}, y^{(m)}) = \left\{(\mathbf{x}^{(i)}, y^{(i)}) \mid 1 \leq i \leq m\right\}$
- *Hypothesis class:* Linear regression: $h(\mathbf{x; w}) = w_1x_1 + w_2x_2$
- *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(**x**) change when we **only change one** of its inputs x_i?

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

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

Gradient: A vector that contains all of the 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 direction of the gradient

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

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

learning rate

A: That is an empirical question with no "right" answer.
We try different learning rates and see which works well.

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

Underflow

$$\underbrace{1.001\ldots011010}_{significand} \times 2^{\overbrace{1001\ldots0011}^{exponent}}$$

- Numbers that are smaller than $1.00...01 \times 2^{-1111...1111}$ will be rounded down 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 numbers
    - Overflows for "large" 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**?)

A: Because the when the large number is e.g., 1.000...000 x 2^n, the difference between 1.000...000 x 2^n and 1.000...001 x 2^n might be larger than the small number.

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

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?

A: Output of exp is always positive

- But exp **overflows** very quickly:
  - Solution: softmax($\mathbf{z}$)   where $\mathbf{z} = \mathbf{x} - \max_j x_j$

Log

- Dataset likelihoods grow small **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 ≈ -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