Calculus Refresher

CMPUT 366: Intelligent Systems

GBC §4.1, 4.3

Assignment #2

- Assignment #2 due **Friday, March 5 at 11:59pm**
  - Submit via eClass

Lecture Outline

1. Recap
2. Using Model Probabilities
3. Prior Distributions as Bias
4. Gradient-based optimization
5. 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

Using Model Probabilities

So we can estimate $\Pr(\theta \mid D)$. What can we do with it?

1. Parameter estimates
2. Target predictions (model averaging)
3. Target predictions (point estimates)

1. Parameter Estimates

- Sometimes, we really want to know the **parameters** of a model itself
- E.g., maybe I don't care about predicting the next coin flip, but I do want to know whether the coin is fair
- Can use $\Pr(\theta \mid D)$ to make statements like

$$\Pr(0.49 \leq \theta \leq 0.51) > 0.9$$

2. Model Averaging

- Sometimes we do want to make **predictions**:

$$\Pr(Y \mid D) = \sum_{\theta} \Pr(Y \mid \theta) \Pr(\theta \mid D)$$

- This is called the **posterior predictive distribution**
- **Question:** How is this different from just learning a point estimate of a model, and then predicting with that model?

3. Maximum A Posteriori

- Sometimes we do want to make predictions, **but...**

$$\Pr(Y \mid D) = \int_0^1 \Pr(Y \mid \theta)\Pr(\theta \mid D)d\theta$$

- the posterior predictive distribution may be **expensive** to compute (or even **intractable**)

- One possible solution is to use the **maximum a posterior** model as a point estimate:

$$\Pr(Y \mid D) \simeq \Pr(Y \mid \hat{\theta}) \qquad \text{where } \hat{\theta} = \arg\max_{\theta} \Pr(\theta \mid D)$$

- **Question:** Why would you do this instead of just using a point estimate that was computed in the usual way?

Prior Distributions as Bias

- Suppose I'm comparing two models, θ_1 and θ_2 such that

$$\Pr(D \mid \theta_1) = Pr(D \mid \theta_2)$$

- **Question:** Which model has higher **posterior probability**?
- Priors are a way of encoding **bias**: they tell use which models to prefer when the data doesn't

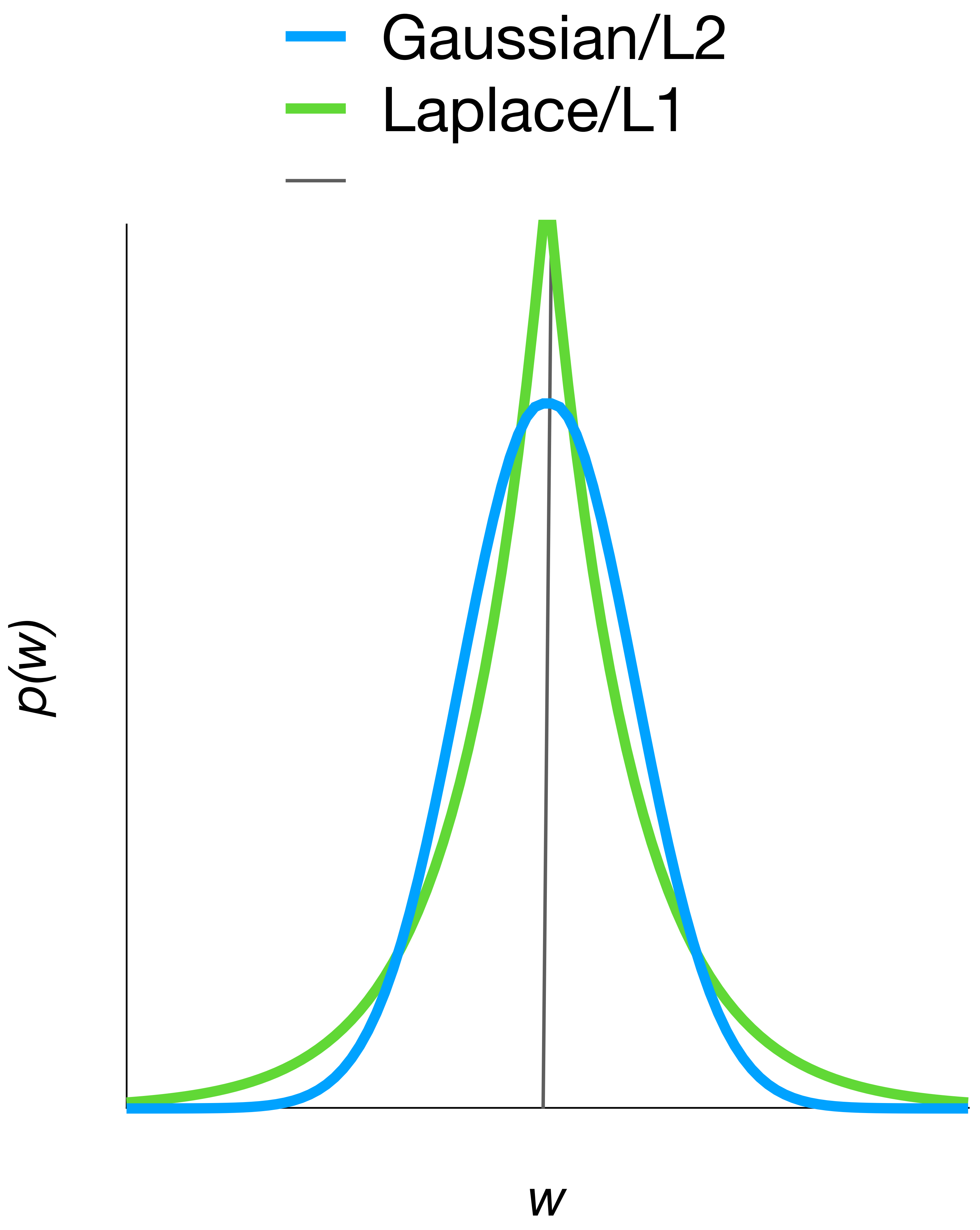
Priors for Pseudocounts

- We can straightforwardly encode pseudocounts as prior information in Beta-Binomial and Dirichlet-Multinomial models
- E.g., for pseudocounts k_1 and k_0,

$$p(\theta) = \mathrm{Beta}(1 + k_1, 1 + k_0)$$

Priors for Regularization

- Some **regularizers** can be encoded as priors also
- **L2 regularization** is equivalent to a **Gaussian** prior on the weights: $p(w) = \mathcal{N}(w \mid m, s)$
- **L1 regularization** is equivalent to a **Laplacian** prior on the weights: $p(w) = \exp(|w|)/2$



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** $f(x)$

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

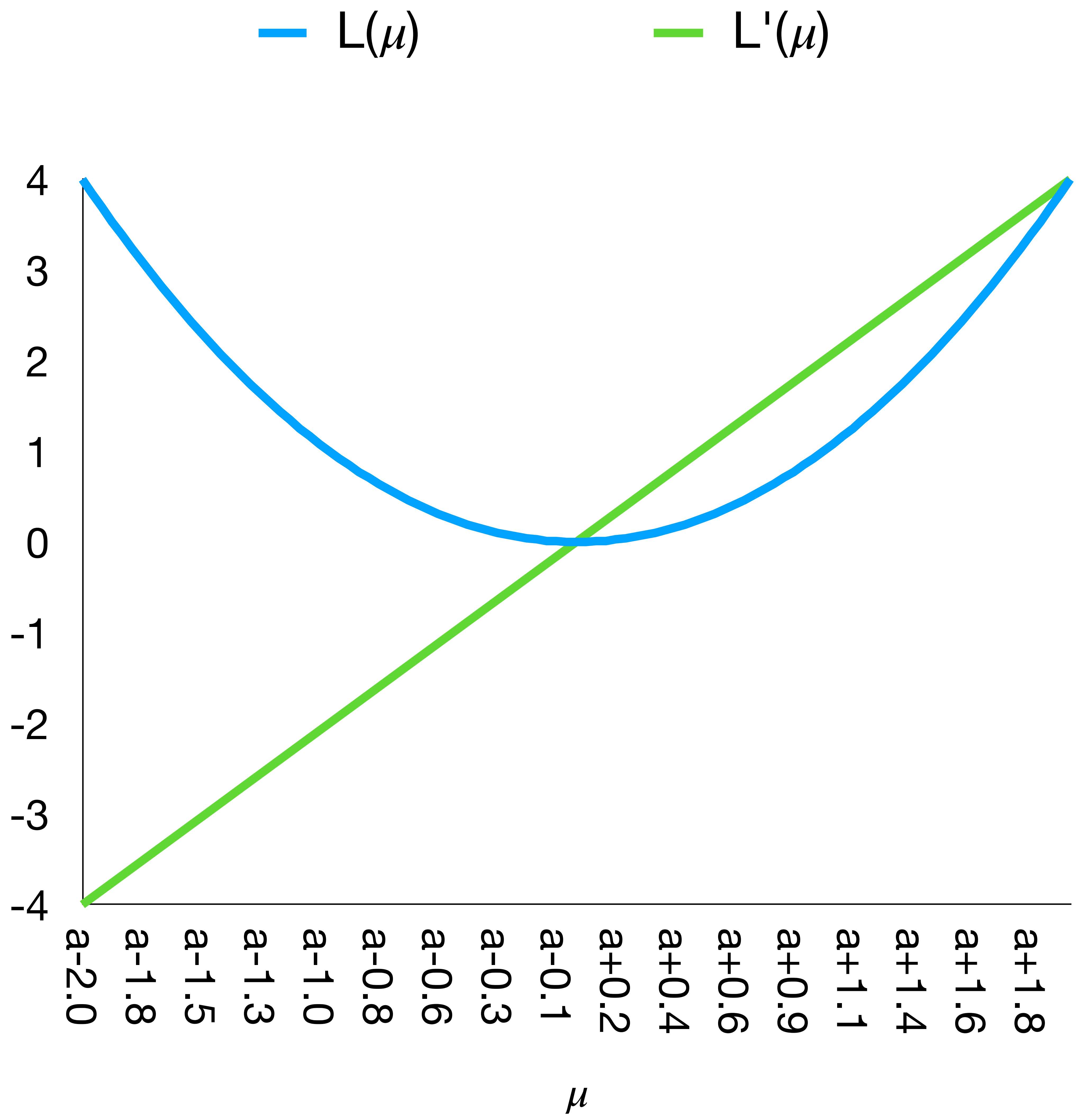
- Temperature example: Find μ that makes $L(\mu)$ small

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** $f'(x) = \dfrac{d}{dx} f(x)$ 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**: $h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$
- *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 $g(x_i) = f(x_1, \ldots, \mathbf{x_i}, \ldots, x_n)$:

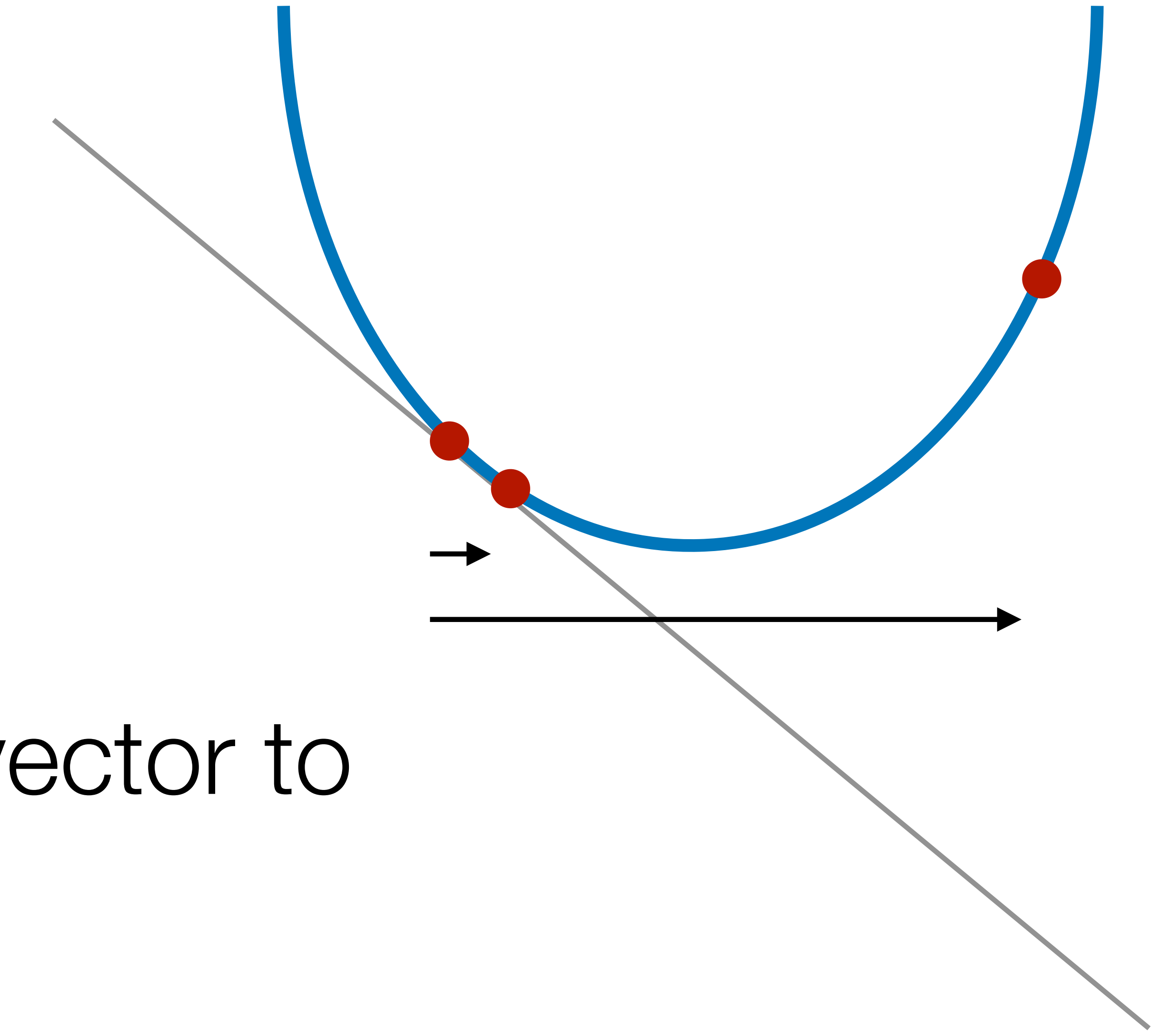
$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \frac{d}{dx_i} g(x_i).$$

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:

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

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

Underflow

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

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

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

- Numbers bigger than $1.111\ldots1111 \times 2^{1111}$ will be rounded up to **infinity**
- Numbers smaller than $-1.111\ldots1111 \times 2^{1111}$ 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

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

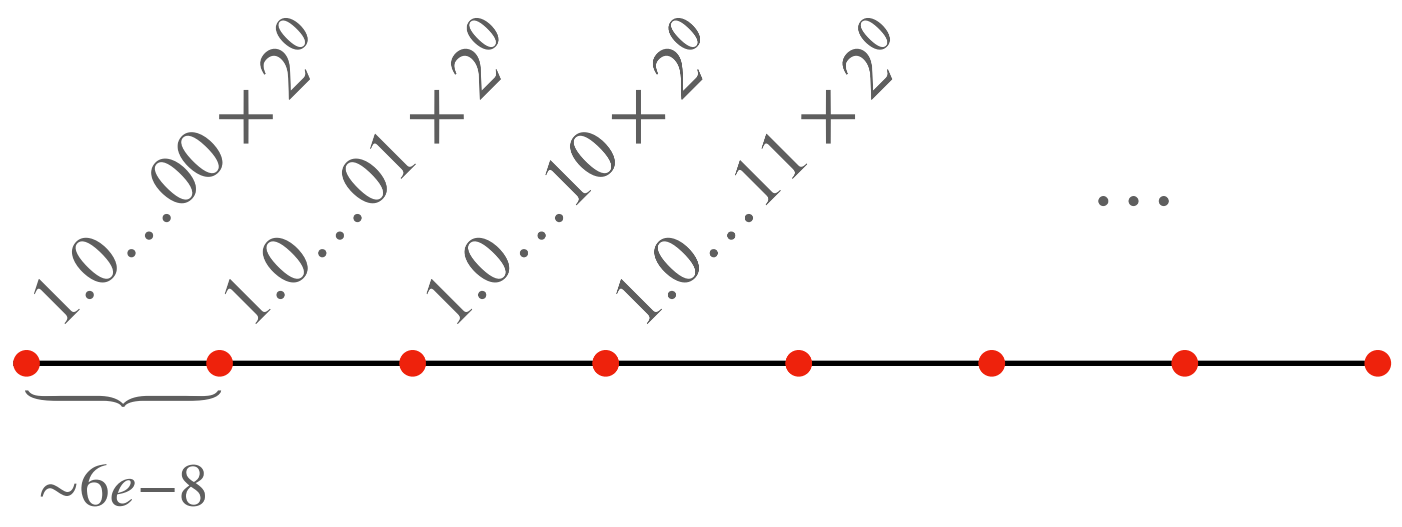
- Adding a small number to a large number can have no effect (**why**?)

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