Baye's Rule & Conditional Probability

intuition: updating beliefs with evidence

Likelihood: the probability of "B" being True given "A" is true

After

$$* P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Before

Posterior: Probability of "A" being true given "B" being true

Prior: Probability of "A" being True

Probability of "B" being True

where does it show up?

- Naive Bayes Classifier
  - Assume features are conditionally independent

Distributions: How Probability is Spread over Possible outcomes

* Bernoulli

↳ for binary outcomes (success / failure)

$P(x=1) = p$

$P(x=0) = q = 1 - p$

↳ Mean: p

Variance: $p(1-p)$

↳ Ex: Coin flip.

* Binomial

↳ Models number of successes in n independent trails

$$P(x=k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

$\binom{n}{k}$: # of combos

↳ Mean: np

↳ Variance: $np(1-p)$

↳ Ex: how many of these 100 predictions were correct.

* Gaussian (Normal)

important!

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

σ^2: Variance

μ: Mean

↳ Noise assumptions in linear regression justifies least squares loss

↳ Minimizing MSE is Maximizing likelihood under gaussian noise

PDFs : Probability Density Function

Range : $[0, \infty)$

$$P(a \le x \le b) = \int_a^b f(x)\, dx$$

$f(x)$ ↖ PDF

Area under the curve = 1

* Bell curve (for Gaussian)
* At a point you'll get density

CDFs : Cumulative Distribution Function

Range : $[0, 1]$

$$F(x) = P(X \le x) = \int_{-\infty}^{x} f(t)\, dt$$

* S-curve
* whats the probability of getting a value up to x?

Relationship: $f(x) = \frac{d}{dx} F(x)$

PDF —derivative→ CDF

CDF —Integral→ PDF

Derivative to Gradient

* Derivative tells you the slope ← one input
  - ↳ how much f changes as x changes

* Gradient is just derivative w.r.t each input ← multiple inputs

weights

$$\nabla f(w) = \begin{bmatrix} df/dw_1 \\ df/dw_2 \\ \vdots \\ df/dw_n \end{bmatrix}$$

Partial derivative

↳ Gradient points in the direction of Steepest Ascent ← direction that increases f the fastest

$$w_1 = w_0 - \alpha \nabla f(w_0)$$

- w_1: new weights
- w_0: current weights
- α: learning Rate: how big of a Step to take
- $\nabla f(w_0)$: the gradient at current weights

Note: Gradient points toward increasing loss. We want to decrease loss so we go the opposite weight

α: Learning Rate

* If α is too large
  - ↳ we overshoot the minimum → Loss bounces around or diverges

* If α is too small
  - ↳ Learning is correct but slow will eventually converge

* when α is just right ← GOAL!
  - ↳ we get a function that steadily converges to minimum

Derivatives of Common Functions

Appear in loss functions

1) Power rule: $\frac{d}{dx} x^n = n x^{n-1}$

- $x^2 \Rightarrow 2x$
- $3x^3 \Rightarrow 9x^2$

2) Exponential: $\frac{d}{dx} e^{ax} = a e^x$

3) logarithm: $\frac{d}{dx} \ln(x) = \frac{1}{x}$

- log of a function: $\frac{d}{dx} (\ln(g(x)) = \frac{g'(x)}{g(x)}$

Derivative Rules

MSE loss through Prediction

$\frac{d}{dx} L = (y - wx)^2$

$l = -2x(y - wx)$

1) Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$

2) Product Rule: $\frac{d}{dx} f(x)g(x) = f'(x)g(x) + f(x)g'(x)$

3) Quotient Rule: $\frac{d}{dx} \frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$

The Flow:

linear model

→ Start by finding $\hat{y} = w^T x$

→ Define loss: $L(w) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)$

→ Take derivative: $\nabla L(w) = -\frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) x_i$ ← gradient

↳ Setting $\nabla L(w) = 0$:

* Closed-form Sol^n

$w = (x^T x)^{-1} x^T y$

No iteration needed and gives exact Solution

↳ Not setting $\nabla L(w)$ to 0

* Gradient descent

$w \leftarrow w - \alpha \nabla L(w)$

update

stepping until the gradient is close enough to 0

optimizations

Optimizing Loss Functions

Gradient Descent - Convergence Issues

- **Oscillation**: When the learning rate is **too large** or the loss surface has regions of high curvature, the algorithm oscillates around the minimum rather than converging smoothly.
- **Slow convergence in flat regions**: When gradients are small, parameter updates become negligible, leading to extremely slow progress.
- **Divergence**: If the learning rate exceeds a certain value for convex functions, the algorithm can **diverge** entirely, with the loss increasing without bound.
- Saddle points: In high dimensions, saddle points are ubiquitous. The **gradient at a saddle point is zero**, causing standard gradient descent to stall.

Optimizing Loss Functions

Gradient Descent - Momentum

- Standard gradient descent can oscillate in ravines
  - Areas where the surface curves **more steeply in one dimension** than another
  - Or they can get stuck in plateau / saddle points
- Momentum helps accelerate gradient descent by accumulating velocity in **directions of consistent gradient**

$$\theta_j \leftarrow \theta_j - \alpha \cdot \frac{\partial \ell_\theta(x)}{\partial \theta_j}$$

$$\theta_t = \theta_{t-1} - \alpha \nabla \ell_{\theta_{t-1}}$$

Optimizing Loss Functions

Gradient Descent - Momentum

- Momentum helps accelerate gradient descent by accumulating velocity in **directions of consistent gradient**

$$\theta_t = \theta_{t-1} - \alpha \nabla \ell_{\theta_{t-1}}$$

With Momentum

velocity vector

$$v_t = \beta \cdot v_{t-1} + \nabla \ell_{\theta_{t-1}}$$

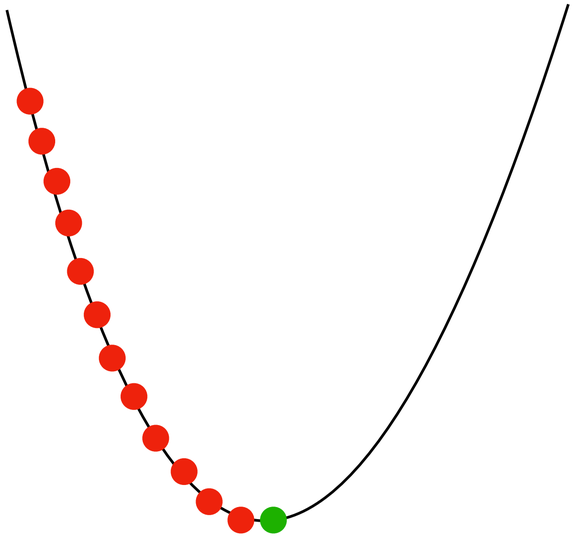
$$\theta_t = \theta_{t-1} - \alpha \cdot v_t$$

- β is the momentum coeff ($\beta = 0.9$)
- $\beta = 0$ we get gradient descent

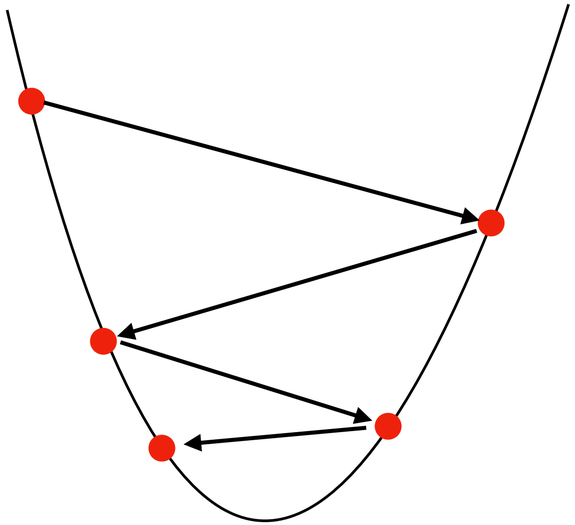
Think of momentum as gravity pulling a ball down a hill, the momentum will carry the ball through any flat or even small uphill regions

Optimizing Loss Functions

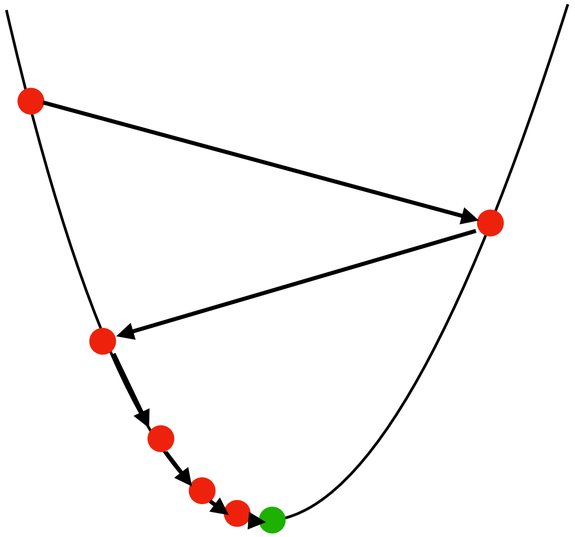
Gradient Descent - Adaptive Step Sizes



α is too small
Finds the optimal but too slow



α is too large
Might not find optimal
Could even begin to diverge



And keep reducing α as
number of epochs increases?

Optimizing Loss Functions

Gradient Descent - Per Parameter Adaptive Learning Rates

- A single global learning rate may be suboptimal
  - Some parameters might benefit from larger updates while others need smaller ones.
  - Adaptive methods adjust the learning rate for each parameter individually based on historical gradient information.

Optimizing Loss Functions

Gradient Descent - AdaGrad

- AdaGrad adapts the learning rate for **each parameter** based on the sum of squared historical gradients

$$G_t = G_{t-1} + (\nabla \ell_{\theta t-1})^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \cdot \nabla \ell_{\theta_{t-1}}$$

  - Parameters with large historical gradients receive smaller updates
  - Parameters with small historical gradients receive larger updates
  - **The limitation is that the accumulated sum G_t grows monotonically, eventually making the learning rate vanishingly small.**

Optimizing Loss Functions

Gradient Descent - RMSProp

- RMSprop addresses AdaGrad's diminishing learning rate by using an exponentially decaying average of squared gradients

$$G_t = \rho \cdot G_{t-1} + (1 - \rho) \cdot (\nabla \ell_{\theta t-1})^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \cdot \nabla \ell_{\theta_{t-1}}$$

  - The decay rate ρ is typically set to 0.9.
  - This prevents the learning rate from decaying to zero while still adapting to the gradient scale.

Optimizing Loss Functions

Gradient Descent - ADAM

- Adam (**Ada**ptive **M**oment Estimation) combines the benefits of **momentum** (first moment) with the adaptive learning rates of **RMSProp** (second moment)

Adam maintains **two** moving averages

First Moment (mean): $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla \ell_{\theta_{t-1}}$

Second Moment (variance): $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla \ell_{\theta_{t-1}})^2$

Update: $\theta_t = \theta_{t-1} - \dfrac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$

Default Hyperparameters: $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \alpha = 10^{-3}$

Bias Correction: Important for early iterations when estimates are biased towards 0

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

- Batch Gradient Descent
  - Use **entire training set per epoch**
  - The whole training dataset is used to compute a single parameter update
  - One epoch leads to **one** parameter update

$$\theta_t = \theta_{t-1} - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla \ell_{\theta_{t-1}}(x_i, y_i)$$

Sum over the whole training dataset

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

- Stochastic Gradient Descent
  - Use **one** randomly selected training data point at each step
  - Parameters are updated after looking at each data point
  - One epoch leads to **m** parameter updates

$$\theta_t = \theta_{t-1} - \alpha \nabla \ell_{\theta_{t-1}}(x_i, y_i)$$

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

- Mini-Batch Gradient Descent
  - A compromise between batch and stochastic variants
  - Use a small batch of randomly sampled training data points
  - Typical batch sizes are $B = 32, 64, 128, 256, 512, 1024$

$$\theta_t = \theta_{t-1} - \alpha \frac{1}{B} \sum_{i=1}^{B} \nabla \ell_{\theta_{t-1}}(x_i, y_i)$$

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

Batch Pros:

Stable Convergence: No noise in gradient estimates means smooth, predictable progress toward the minimum

Guaranteed Descent: Each update is guaranteed to reduce the loss (with appropriate learning rate)

Simple learning rate selection: The lack of noise means you can often use larger learning rates without instability

Parallelizable Gradient Computation: The sum over all samples can be computed in parallel across multiple processors

Stochastic Pros:

Fast Updates: Each parameter update is computationally cheap, allowing rapid initial progress.

Memory Efficient: Only one sample needs to be in memory at a time.

Escapes Local Minima: The inherent noise helps the algorithm escape shallow local minima and saddle points. The stochasticity acts as implicit regularization

Online Learning: Can naturally incorporate new data as it arrives - just perform an update on each new sample

Better Generalization: The noise can prevent overfitting to the training set.

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

Batch Cons:

Computationally Expensive: For large datasets, computing the full gradient is very slow. A dataset with 10 million samples requires processing all 10 million before a single update.

Memory Intensive: The entire dataset must fit in memory.

Redundant Computation: Many datasets contain redundant or similar samples. BGD computes gradients for all of them even when a subset would provide nearly the same information.

Poor Escape From Local Minima: The **deterministic** nature means the algorithm follows the same path every time and can get permanently stuck in local minima or saddle points.

Slow for Online Learning: Cannot incorporate new data without reprocessing everything.

Stochastic Cons:

High Variance: Individual gradient estimates can be very noisy, causing erratic updates.

Unstable Convergence: The loss curve is noisy. The algorithm may step away from the minimum even when near it.

Requires Learning Rate Decay: To converge to a minimum (rather than oscillating around it), the **learning rate must decrease** over time, adding hyperparameters.

Poor Hardware Utilization: Modern GPUs are optimized for **parallel operations on batches,** not sequential single-sample operations. SGD fails to exploit this.

Sensitive to Sample Ordering: The order in which samples are presented can affect results, requiring careful shuffling.

Gradient Descent

Batch vs Mini-Batch vs Stochastic Gradient Descent

Mini-Batch

Variance Reduction: Averaging over B samples reduces gradient variance by a factor of B compared to pure SGD, while still maintaining some beneficial noise

Hardware Efficiency: GPUs perform matrix operations in parallel. A batch size of 64 is nearly as fast as a batch size of 1 on modern hardware, giving essentially 64× speedup over SGD

Memory-Computation Tradeoff: Batch size can be tuned to maximize GPU memory utilization without requiring the full dataset

Balances Exploration and Exploitation: Enough noise to escape poor regions, enough signal to make consistent progress.

Gradient Descent

Gradient Descent vs Closed Form

Gradient Descent

+ Linear increase in m (# training data) and n (# features)
+ Generally applicable to multiple models
+ Guaranteed to reach global optimum for convex functions and appropriate learning rate
- Need to choose learning rate α and stopping conditions
- Need to choose optimization method (Adam, RMSProp etc..)
- Might get stuck in local optima / saddle point
- Needs feature scaling

Closed Form

$$\theta = (X^T X)^{-1} X^T Y$$

+ No parameter tuning
+ Gives global optimum
- Not generally applicable to any learning algorithm
- Slow computation - scales with n^3 where n is number of features