Gradient Descent: The Foundation of ML Optimization

From Taylor Series to Modern Deep Learning

Nipun Batra and Teaching Staff

IIT Gandhinagar

August 28, 2025

Outline

The Core ML Problem

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

The Core ML Problem

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

Examples everywhere:

- Linear regression: $\min(y - \boldsymbol{X\theta})^2$
- Neural networks: min classification/regression loss
- Logistic regression: min cross-entropy loss

The Core ML Problem

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

Examples everywhere:

- Linear regression: $\min(y - \boldsymbol{X\theta})^2$
- Neural networks: min classification/regression loss
- Logistic regression: min cross-entropy loss

Challenge: Most ML problems have no closed-form solution

Geometric Intuition: Climbing Mountains

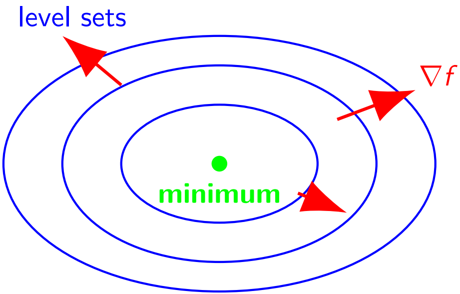
Imagine hiking in fog to reach the valley:

- Feel slope beneath your feet
- **Strategy:** Step in steepest downhill direction
- **Gradient** = steepest uphill
- **Negative gradient** = steepest downhill

Geometric Intuition: Climbing Mountains

Imagine hiking in fog to reach the valley:

- Feel slope beneath your feet
- **Strategy:** Step in steepest downhill direction
- **Gradient** = steepest uphill
- **Negative gradient** = steepest downhill



Think!

Why does following $-\nabla f$ lead us toward the minimum?

Think!

Why does following $-\nabla f$ lead us toward the minimum?

(Solution in Appendix)

Why Taylor Series?

Key insight: If we can't solve $\min f(\boldsymbol{x})$ exactly, approximate $f(\boldsymbol{x})$ locally!

Why Taylor Series?

Key insight: If we can't solve $\min f(\boldsymbol{x})$ exactly, approximate $f(\boldsymbol{x})$ locally!

Univariate Taylor expansion:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

Key insight: If we can't solve $\min f(\boldsymbol{x})$ exactly, approximate $f(\boldsymbol{x})$ locally!

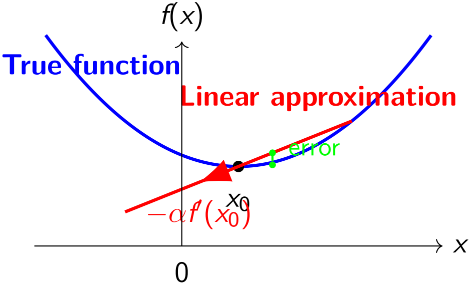
Univariate Taylor expansion:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$$

- **Zero-order:** $f(x) \approx f(x_0)$ (constant)
- **First-order:** adds linear term (tangent)
- **Second-order:** adds quadratic curvature

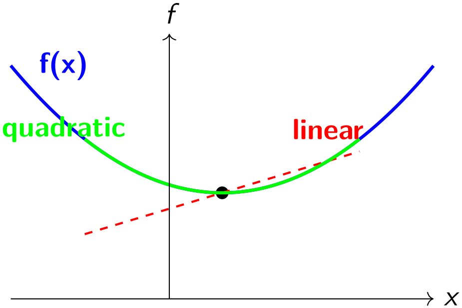
Visual: Tangent Line Approximation

Linear approximation: Use tangent line to approximate function locally



Key insight: Tangent gives best local linear approximation!

Visual: Adding Quadratic Term



Key insight: Higher-order terms give better approximations!

Concrete Example: $f(x) = \cos(x)$ at $x_0 = 0$

$$
\begin{align}
f(0) &= \cos(0) = 1 \tag{1}\\
f'(0) &= -\sin(0) = 0 \tag{2}\\
f''(0) &= -\cos(0) = -1 \tag{3}\\
f'''(0) &= \sin(0) = 0 \tag{4}\\
f^{(4)}(0) &= \cos(0) = 1 \tag{5}
\end{align}
$$

Concrete Example: $f(x) = \cos(x)$ at $x_0 = 0$

$$
\begin{align}
f(0) &= \cos(0) = 1 \tag{1}\\
f'(0) &= -\sin(0) = 0 \tag{2}\\
f''(0) &= -\cos(0) = -1 \tag{3}\\
f'''(0) &= \sin(0) = 0 \tag{4}\\
f^{(4)}(0) &= \cos(0) = 1 \tag{5}
\end{align}
$$

Taylor approximations:

$$
\begin{align}
\text{0th order:} \quad & f(x) \approx 1 \tag{6}\\
\text{2nd order:} \quad & f(x) \approx 1 - \frac{x^2}{2} \tag{7}\\
\text{4th order:} \quad & f(x) \approx 1 - \frac{x^2}{2} + \frac{x^4}{24} \tag{8}
\end{align}
$$

Multivariate Taylor Series

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} + \frac{1}{2} \Delta \boldsymbol{x}^T \nabla^2 f(\boldsymbol{x}_0) \Delta \boldsymbol{x}$$

Multivariate Taylor Series

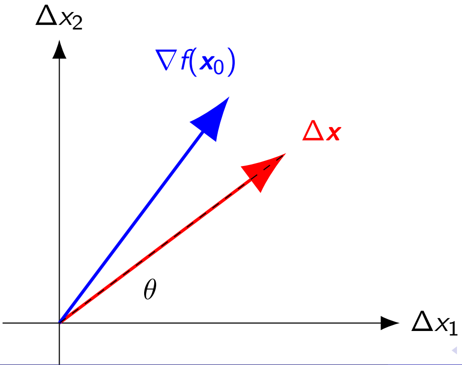
$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} + \frac{1}{2} \Delta \boldsymbol{x}^T \nabla^2 f(\boldsymbol{x}_0) \Delta \boldsymbol{x}$$

where $\Delta \boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_0$

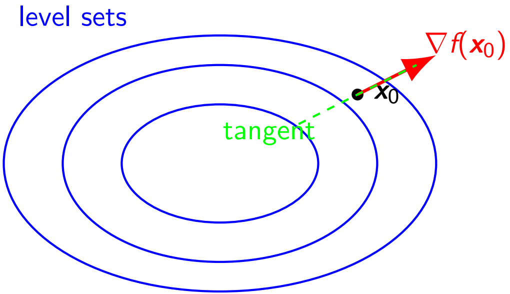
Multivariate Taylor Series

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} + \frac{1}{2} \Delta \boldsymbol{x}^T \nabla^2 f(\boldsymbol{x}_0) \Delta \boldsymbol{x}$$

where $\Delta \boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_0$



Visual: Multivariate Case with Level Sets



Key: Gradient $\perp$ level sets, tangent plane $\perp$ gradient

Why is Gradient Perpendicular to Level Sets?

Mathematical insight: Level set $= \{\boldsymbol{x} : f(\boldsymbol{x}) = c\}$ for constant c

Why is Gradient Perpendicular to Level Sets?

Mathematical insight: Level set $= \{\boldsymbol{x} : f(\boldsymbol{x}) = c\}$ for constant c

On level sets: Moving along the level curve keeps $f(\boldsymbol{x})$ constant

- If $\boldsymbol{x}(t)$ parameterizes level curve: $f(\boldsymbol{x}(t)) = c$ (constant)
- Taking derivative: $\frac{d}{dt}f(\boldsymbol{x}(t)) = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{x}'(t) = 0$

Why is Gradient Perpendicular to Level Sets?

Mathematical insight: Level set $= \{\boldsymbol{x} : f(\boldsymbol{x}) = c\}$ for constant c
On level sets: Moving along the level curve keeps $f(\boldsymbol{x})$ constant

- If $\boldsymbol{x}(t)$ parameterizes level curve: $f(\boldsymbol{x}(t)) = c$ (constant)
- Taking derivative: $\frac{d}{dt} f(\boldsymbol{x}(t)) = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{x}'(t) = 0$

Conclusion: $\nabla f(\boldsymbol{x}) \perp \boldsymbol{x}'(t)$ for any tangent direction $\boldsymbol{x}'(t)$

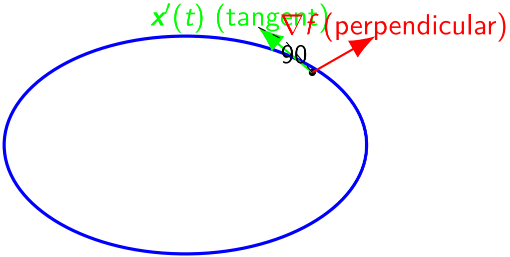
Why is Gradient Perpendicular to Level Sets?

Mathematical insight: Level set $= \{\boldsymbol{x} : f(\boldsymbol{x}) = c\}$ for constant c

On level sets: Moving along the level curve keeps $f(\boldsymbol{x})$ constant

- If $\boldsymbol{x}(t)$ parameterizes level curve: $f(\boldsymbol{x}(t)) = c$ (constant)
- Taking derivative: $\frac{d}{dt} f(\boldsymbol{x}(t)) = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{x}'(t) = 0$

Conclusion: $\nabla f(\boldsymbol{x}) \perp \boldsymbol{x}'(t)$ for any tangent direction $\boldsymbol{x}'(t)$



Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$

From Taylor Series to Gradient Descent

Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$

Using first-order Taylor approximation:

$$f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x}$$

From Taylor Series to Gradient Descent

Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$
Using first-order Taylor approximation:

$$f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x}$$

To minimize, we need: $\nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} < 0$

From Taylor Series to Gradient Descent

Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$
Using first-order Taylor approximation:

$$f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x}$$

To minimize, we need: $\nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} < 0$
Vector geometry insight: For vectors $\boldsymbol{a}$, $\boldsymbol{b}$: $\boldsymbol{a}^T \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos(\theta)$
Minimum when $\cos(\theta) = -1$ (opposite directions!)

From Taylor Series to Gradient Descent

Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$
Using first-order Taylor approximation:

$$f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x}$$

To minimize, we need: $\nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} < 0$
Vector geometry insight: For vectors $\boldsymbol{a}$, $\boldsymbol{b}$: $\boldsymbol{a}^T \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos(\theta)$
Minimum when $\cos(\theta) = -1$ (opposite directions!)
Optimal choice: $\Delta \boldsymbol{x} = -\alpha \nabla f(\boldsymbol{x}_0)$ where $\alpha > 0$

From Taylor Series to Gradient Descent

Goal: Find $\Delta \boldsymbol{x}$ such that $f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) < f(\boldsymbol{x}_0)$
Using first-order Taylor approximation:

$$f(\boldsymbol{x}_0 + \Delta \boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x}$$

To minimize, we need: $\nabla f(\boldsymbol{x}_0)^T \Delta \boldsymbol{x} < 0$
Vector geometry insight: For vectors $\boldsymbol{a}$, $\boldsymbol{b}$: $\boldsymbol{a}^T \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos(\theta)$
Minimum when $\cos(\theta) = -1$ (opposite directions!)
Optimal choice: $\Delta \boldsymbol{x} = -\alpha \nabla f(\boldsymbol{x}_0)$ where $\alpha > 0$

$$\boxed{\boldsymbol{x}_{\text{new}} = \boldsymbol{x}_{\text{old}} - \alpha \nabla f(\boldsymbol{x}_{\text{old}})}$$

The Gradient Descent Algorithm

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)}$$

The Gradient Descent Algorithm

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)}$$

Algorithm:

1. **Initialize:** θ_0 (random or educated guess)
2. **For** $t = 0, 1, 2, \ldots$ until convergence:
   - Compute gradient: $\boldsymbol{g}_t = \nabla f(\boldsymbol{\theta}_t)$
   - Update parameters: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$
   - Check convergence: $|\boldsymbol{g}_t| < \epsilon$

The Gradient Descent Algorithm

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)}$$

Algorithm:

1. **Initialize:** θ_0 (random or educated guess)
2. **For** $t = 0, 1, 2, \ldots$ until convergence:
   - Compute gradient: $\boldsymbol{g}_t = \nabla f(\boldsymbol{\theta}_t)$
   - Update parameters: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$
   - Check convergence: $|\boldsymbol{g}_t| < \epsilon$

Key properties:

- First-order method (uses gradients, not Hessians)
- Greedy local search
- Guaranteed convergence for convex functions

Linear Regression Problem

Learn: $y = \theta_0 + \theta_1 x$ from data

x	**y**
1	1
2	2
3	3

Linear Regression Problem

Learn: $y = \theta_0 + \theta_1 x$ from data

x	**y**
1	1
2	2
3	3

Cost Function (Mean Squared Error):

$$\text{MSE}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)^2$$

Learn: $y = \theta_0 + \theta_1 x$ from data

x	**y**
1	1
2	2
3	3

Cost Function (Mean Squared Error):

$$\text{MSE}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)^2$$

Goal: $(\theta_0^*, \theta_1^*) = \text{argmin}_{\theta_0,\theta_1} \text{MSE}(\theta_0, \theta_1)$

Computing Gradients for Linear Regression

We need: $\nabla \text{MSE} = \begin{bmatrix} \frac{\partial \text{MSE}}{\partial \theta_0} \\ \frac{\partial \text{MSE}}{\partial \theta_1} \end{bmatrix}$

Computing Gradients for Linear Regression

We need: $\nabla \text{MSE} = \begin{bmatrix} \frac{\partial \text{MSE}}{\partial \theta_0} \\ \frac{\partial \text{MSE}}{\partial \theta_1} \end{bmatrix}$

Partial derivatives:

$$\frac{\partial \text{MSE}}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)(-1) = -\frac{2}{n}\sum_{i=1}^{n}\epsilon_i \tag{9}$$

$$\frac{\partial \text{MSE}}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)(-x_i) = -\frac{2}{n}\sum_{i=1}^{n}\epsilon_i x_i \tag{10}$$

where $\epsilon_i = y_i - \hat{y}_i$ is the residual.

Computing Gradients for Linear Regression

We need: $\nabla \text{MSE} = \begin{bmatrix} \frac{\partial \text{MSE}}{\partial \theta_0} \\ \frac{\partial \text{MSE}}{\partial \theta_1} \end{bmatrix}$

Partial derivatives:

$$\frac{\partial \text{MSE}}{\partial \theta_0} = \frac{2}{n} \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i)(-1) = -\frac{2}{n} \sum_{i=1}^{n} \epsilon_i \tag{9}$$

$$\frac{\partial \text{MSE}}{\partial \theta_1} = \frac{2}{n} \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i)(-x_i) = -\frac{2}{n} \sum_{i=1}^{n} \epsilon_i x_i \tag{10}$$

where $\epsilon_i = y_i - \hat{y}_i$ is the residual.

Matrix form: $\nabla \text{MSE}(\boldsymbol{\theta}) = \frac{2}{n} \boldsymbol{X}^T (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})$

Step-by-Step Example

Initial: $\theta_0 = 4$, $\theta_1 = 0$, $\alpha = 0.1$

Step-by-Step Example

Initial: $\theta_0 = 4$, $\theta_1 = 0$, $\alpha = 0.1$

Iteration 1:

- Predictions: $\hat{y}_1 = 4$, $\hat{y}_2 = 4$, $\hat{y}_3 = 4$
- Errors: $\epsilon_1 = -3$, $\epsilon_2 = -2$, $\epsilon_3 = -1$
- $\frac{\partial \text{MSE}}{\partial \theta_0} = -\frac{2}{3}(-6) = 4$
- $\frac{\partial \text{MSE}}{\partial \theta_1} = -\frac{2}{3}(-10) = 6.67$
- $\theta_0 = 4 - 0.1 \times 4 = 3.6$
- $\theta_1 = 0 - 0.1 \times 6.67 = -0.67$

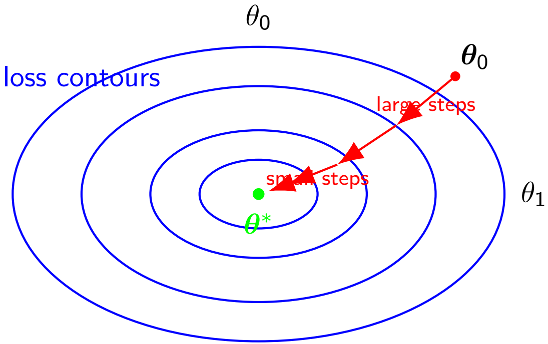
Step-by-Step Example

Initial: $\theta_0 = 4$, $\theta_1 = 0$, $\alpha = 0.1$
Iteration 1:

- Predictions: $\hat{y}_1 = 4$, $\hat{y}_2 = 4$, $\hat{y}_3 = 4$
- Errors: $\epsilon_1 = -3$, $\epsilon_2 = -2$, $\epsilon_3 = -1$
- $\frac{\partial \text{MSE}}{\partial \theta_0} = -\frac{2}{3}(-6) = 4$
- $\frac{\partial \text{MSE}}{\partial \theta_1} = -\frac{2}{3}(-10) = 6.67$
- $\theta_0 = 4 - 0.1 \times 4 = 3.6$
- $\theta_1 = 0 - 0.1 \times 6.67 = -0.67$

New parameters: $(\theta_0, \theta_1) = (3.6, -0.67)$

Visual: GD Path on Loss Surface



Notice: Algorithm takes larger steps when gradient is large!

The Gradient Descent Family

Three main variants based on data usage per update:

Method	Data per update	Updates per epoch	Convergence
Batch GD	n (all)	1	Smooth
SGD	1	n	Noisy
Mini-batch GD	b (batch size)	n/b	Balanced

The Gradient Descent Family

Three main variants based on data usage per update:

Method	Data per update	Updates per epoch	Convergence
Batch GD	n (all)	1	Smooth
SGD	1	n	Noisy
Mini-batch GD	b (batch size)	n/b	Balanced

Trade-offs: Computational cost vs. convergence stability vs. memory

The Gradient Descent Family

Three main variants based on data usage per update:

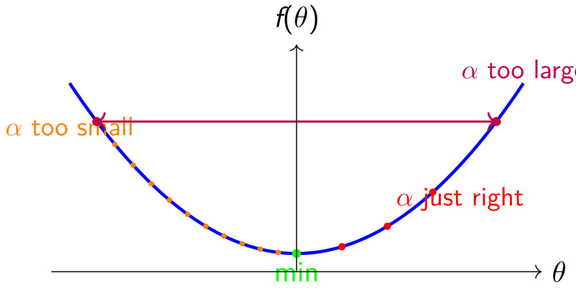
Method	Data per update	Updates per epoch	Convergence
Batch GD	n (all)	1	Smooth
SGD	1	n	Noisy
Mini-batch GD	b (batch size)	n/b	Balanced

Trade-offs: Computational cost vs. convergence stability vs. memory

Modern ML: Mini-batch GD with batch sizes 32-256 is most common

- Good balance of stability and efficiency
- Enables parallel computation (GPUs love batches!)

Learning Rate Effects



Pop Quiz #1

For dataset with 1000 samples and batch size 50:

1. How many iterations per epoch for mini-batch GD?
2. If SGD takes 1000 epochs to converge, roughly how many epochs should mini-batch take?
3. Why might SGD be noisier than batch GD?

Pop Quiz #1

For dataset with 1000 samples and batch size 50:

1. How many iterations per epoch for mini-batch GD?
2. If SGD takes 1000 epochs to converge, roughly how many epochs should mini-batch take?
3. Why might SGD be noisier than batch GD?

(Solutions in Appendix)

Convergence Rates for Convex Functions

L-smooth convex: $\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$
With step size $\alpha \in (0, 1/L]$:

$$f(\boldsymbol{x}_t) - f(\boldsymbol{x}^*) \leq \frac{L\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2}{2t}$$

Convergence Rates for Convex Functions

L-smooth convex: $\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$
With step size $\alpha \in (0, 1/L]$:

$$f(\boldsymbol{x}_t) - f(\boldsymbol{x}^*) \leq \frac{L\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2}{2t}$$

Rate: $O(1/t)$ (sublinear)

Convergence Rates for Convex Functions

L-smooth convex: $\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$
With step size $\alpha \in (0, 1/L]$:

$$f(\boldsymbol{x}_t) - f(\boldsymbol{x}^*) \leq \frac{L\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2}{2t}$$

Rate: $O(1/t)$ (sublinear)
μ-strongly convex + L-smooth:

$$f(\boldsymbol{x}_t) - f(\boldsymbol{x}^*) \leq \left(1 - \frac{\mu}{L}\right)^t (f(\boldsymbol{x}_0) - f(\boldsymbol{x}^*))$$

Rate: Linear convergence! Condition number $\kappa = L/\mu$

SGD as Unbiased Estimator

True gradient: $\nabla L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$
SGD gradient estimate: $\nabla \tilde{L}(\boldsymbol{\theta}) = \nabla \ell(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$, where $(\boldsymbol{x}, y)$ is sampled uniformly from $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$

SGD as Unbiased Estimator

True gradient: $\nabla L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$
SGD gradient estimate: $\nabla \tilde{L}(\boldsymbol{\theta}) = \nabla \ell(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$, where $(\boldsymbol{x}, y)$ is sampled uniformly from $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$
Unbiased estimator property:

$$\mathbb{E}[\nabla \tilde{L}(\boldsymbol{\theta})] = \nabla L(\boldsymbol{\theta})$$

SGD as Unbiased Estimator

True gradient: $\nabla L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$
SGD gradient estimate: $\nabla \tilde{L}(\boldsymbol{\theta}) = \nabla \ell(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$, where $(\boldsymbol{x}, y)$ is sampled uniformly from $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$
Unbiased estimator property:

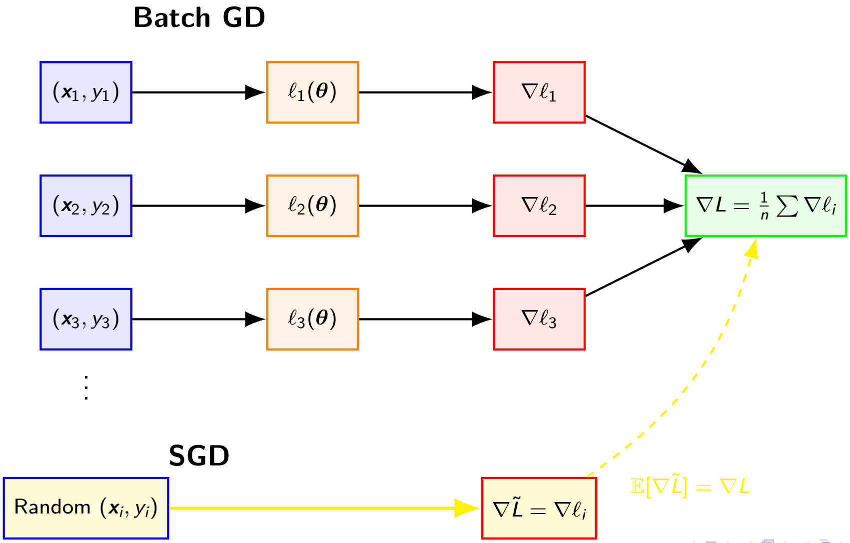
$$\mathbb{E}[\nabla \tilde{L}(\boldsymbol{\theta})] = \nabla L(\boldsymbol{\theta})$$

Proof sketch:

$$\mathbb{E}[\nabla \tilde{L}(\boldsymbol{\theta})] = \sum_{i=1}^{n} \frac{1}{n} \nabla \ell(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i) = \nabla L(\boldsymbol{\theta})$$

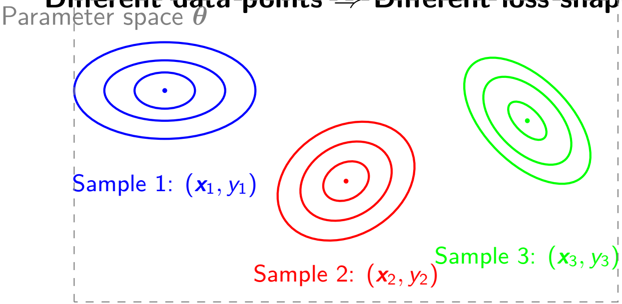
Implication: Individual SGD steps might be "wrong", but average in correct direction!

SGD Computational Graph: From Samples to Gradients



Each data point creates its own loss landscape

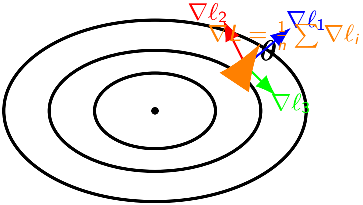
Different data points $\Rightarrow$ Different loss shapes



Key: Each sample $(\boldsymbol{x}_i, y_i)$ defines loss $\ell_i(\boldsymbol{\theta})$ with different optimal $\boldsymbol{\theta}_i^*$

Visual Intuition 2: Averaging Individual Gradients

True gradient = Average of individual gradients



$$\nabla L(\boldsymbol{\theta}) = \frac{1}{n}[\nabla \ell_1(\boldsymbol{\theta}) + \nabla \ell_2(\boldsymbol{\theta}) + \nabla \ell_3(\boldsymbol{\theta})]$$

SGD: Pick one random individual gradient instead of computing expensive average!

Why Unbiasedness Matters

Intuitive analogy: Asking random people for directions

- Individual answers might be slightly off
- But if no systematic bias, average direction is correct
- SGD does the same with gradient estimates!

Why Unbiasedness Matters

Intuitive analogy: Asking random people for directions

- Individual answers might be slightly off
- But if no systematic bias, average direction is correct
- SGD does the same with gradient estimates!

The noise can be beneficial:

- Helps escape local minima in non-convex problems
- Provides implicit regularization
- Enables online learning

GD vs Normal Equation

For linear regression:
Normal equation: $\hat{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$

- Time complexity: $O(d^2 n + d^3)$

GD vs Normal Equation

For linear regression:

Normal equation: $\hat{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$

- Time complexity: $O(d^2 n + d^3)$

Gradient descent: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta}_t - \boldsymbol{y})$

- Time complexity per iteration: $O(dn)$
- Total: $O(T \cdot dn)$ for T iterations

GD vs Normal Equation

For linear regression:

Normal equation: $\hat{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$

- Time complexity: $O(d^2 n + d^3)$

Gradient descent: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta}_t - \boldsymbol{y})$

- Time complexity per iteration: $O(dn)$
- Total: $O(T \cdot dn)$ for T iterations

When to use which?

- Few features ($d < 1000$): Normal equation
- Many features ($d > 10000$): Gradient descent
- Non-linear models: Only gradient descent works

Beyond Basic Gradient Descent

Momentum: $\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + (1-\beta)\boldsymbol{g}_t$, $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{v}_{t+1}$
Adam: Combines momentum + adaptive learning rates

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t$$

Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Beyond Basic Gradient Descent

Momentum: $\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + (1-\beta)\boldsymbol{g}_t$, $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \boldsymbol{v}_{t+1}$
Adam: Combines momentum + adaptive learning rates

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t$$

Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$
Why these improvements?

- Handle different parameter scales automatically
- Accelerate convergence in relevant directions
- Reduce oscillations in narrow valleys

Second-Order Methods

Newton's method: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha[\nabla^2 f(\boldsymbol{\theta}_t)]^{-1}\nabla f(\boldsymbol{\theta}_t)$
Gauss-Newton: For least squares problems
L-BFGS: Quasi-Newton method (approximates Hessian)

Second-Order Methods

Newton's method: $\theta_{t+1} = \theta_t - \alpha[\nabla^2 f(\theta_t)]^{-1}\nabla f(\theta_t)$
Gauss-Newton: For least squares problems
L-BFGS: Quasi-Newton method (approximates Hessian)
Trade-off: Faster convergence vs. $O(d^3)$ cost per iteration

Second-Order Methods

Newton's method: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha[\nabla^2 f(\boldsymbol{\theta}_t)]^{-1}\nabla f(\boldsymbol{\theta}_t)$
Gauss-Newton: For least squares problems
L-BFGS: Quasi-Newton method (approximates Hessian)
Trade-off: Faster convergence vs. $O(d^3)$ cost per iteration
Line search methods: Adaptive step size via Armijo condition

Gradient Descent in Deep Learning

Every modern deep learning framework uses GD variants!

Key extensions:

- **Backpropagation:** Efficient gradient computation
- **Automatic differentiation:** PyTorch/TensorFlow handle gradients
- **GPU acceleration:** Parallel mini-batch gradients
- **Mixed precision:** 16-bit + 32-bit arithmetic

Common strategies:

- Grid search: $\alpha \in \{0.001, 0.01, 0.1, 1.0\}$
- Learning rate schedules: Start high, decay over time
- Adaptive methods: Let algorithm adjust automatically
- Learning rate finder: Gradually increase and watch loss

Learning Rate Selection

Common strategies:

- Grid search: $\alpha \in \{0.001, 0.01, 0.1, 1.0\}$
- Learning rate schedules: Start high, decay over time
- Adaptive methods: Let algorithm adjust automatically
- Learning rate finder: Gradually increase and watch loss

Warning signs:

- Loss exploding $\Rightarrow \alpha$ too high
- Very slow convergence $\Rightarrow \alpha$ too low
- Oscillating loss $\Rightarrow$ Try smaller α or momentum

Other Practical Considerations

Feature scaling: Standardize features: $(x - \mu)/\sigma$

Convergence criteria:

- Gradient magnitude: $\|\nabla f(\boldsymbol{\theta})\| < \epsilon$
- Function change: $|f(\boldsymbol{\theta}_{t+1}) - f(\boldsymbol{\theta}_t)| < \epsilon$
- Maximum iterations: Simple upper bound

Other Practical Considerations

Feature scaling: Standardize features: $(x - \mu)/\sigma$

Convergence criteria:

- Gradient magnitude: $\|\nabla f(\boldsymbol{\theta})\| < \epsilon$
- Function change: $|f(\boldsymbol{\theta}_{t+1}) - f(\boldsymbol{\theta}_t)| < \epsilon$
- Maximum iterations: Simple upper bound

Common pitfalls:

- Poor initialization (use Xavier/He for neural networks)
- Poor feature scaling (different parameter scales)
- Not monitoring validation performance

Think!

For L-smooth function, why might $\alpha > 2/L$ cause divergence on a quadratic?

Think!

For L-smooth function, why might $\alpha > 2/L$ cause divergence on a quadratic?

(Solution in Appendix)

What We've Learned

Core journey:

- **Mathematical foundation:** Taylor series approximation
- **Key insight:** Follow $-\nabla f$ for steepest descent
- **Algorithm:** $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)$
- **Variants:** Batch, SGD, mini-batch (use mini-batch!)
- **Theory:** Convergence rates depend on function properties

What We've Learned

Core journey:

- **Mathematical foundation:** Taylor series approximation
- **Key insight:** Follow $-\nabla f$ for steepest descent
- **Algorithm:** $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)$
- **Variants:** Batch, SGD, mini-batch (use mini-batch!)
- **Theory:** Convergence rates depend on function properties

From theory to practice:

- Tune learning rates carefully
- Scale features properly
- Monitor diagnostics
- Consider advanced optimizers (Adam, momentum)

What We've Learned

Core journey:

- **Mathematical foundation:** Taylor series approximation
- **Key insight:** Follow $-\nabla f$ for steepest descent
- **Algorithm:** $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t)$
- **Variants:** Batch, SGD, mini-batch (use mini-batch!)
- **Theory:** Convergence rates depend on function properties

From theory to practice:

- Tune learning rates carefully
- Scale features properly
- Monitor diagnostics
- Consider advanced optimizers (Adam, momentum)

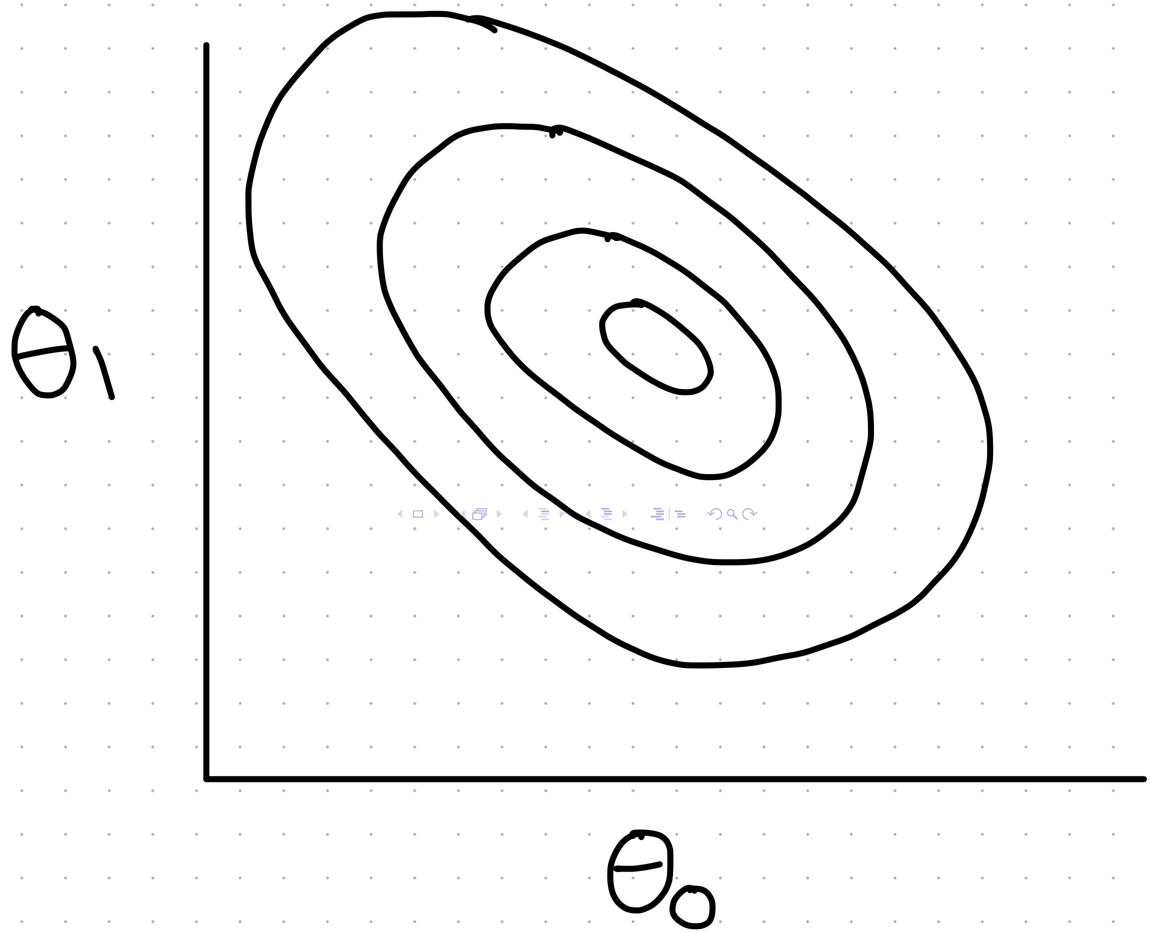
Gradient descent powers modern machine learning!

Deep Dive: Stochastic Gradient Descent Theory

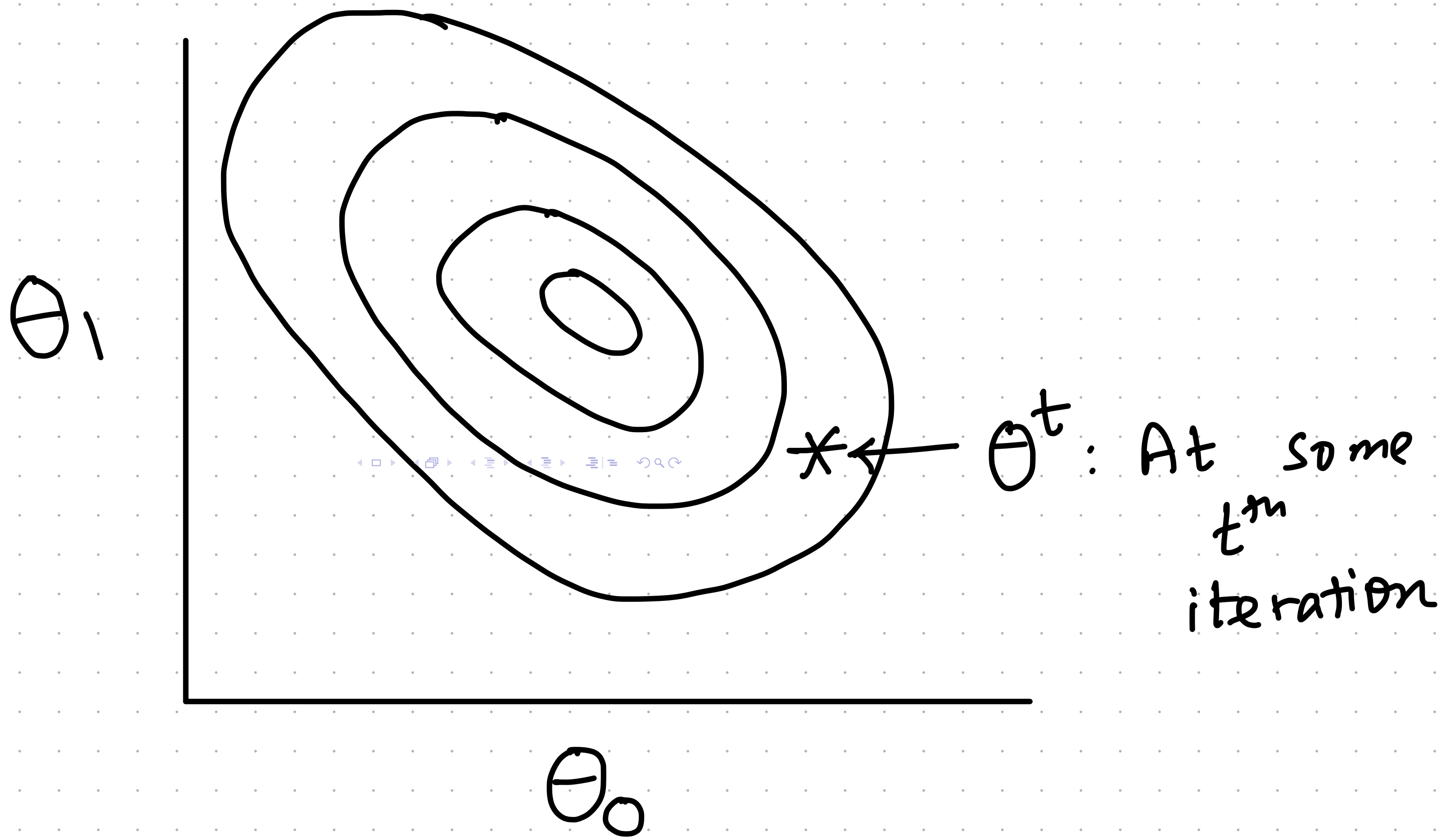
For comprehensive mathematical analysis:

X	y
$-\; x_1^T \;-$	y_1
$\vdots$	
$-\; x_N^T \;-$	y_N

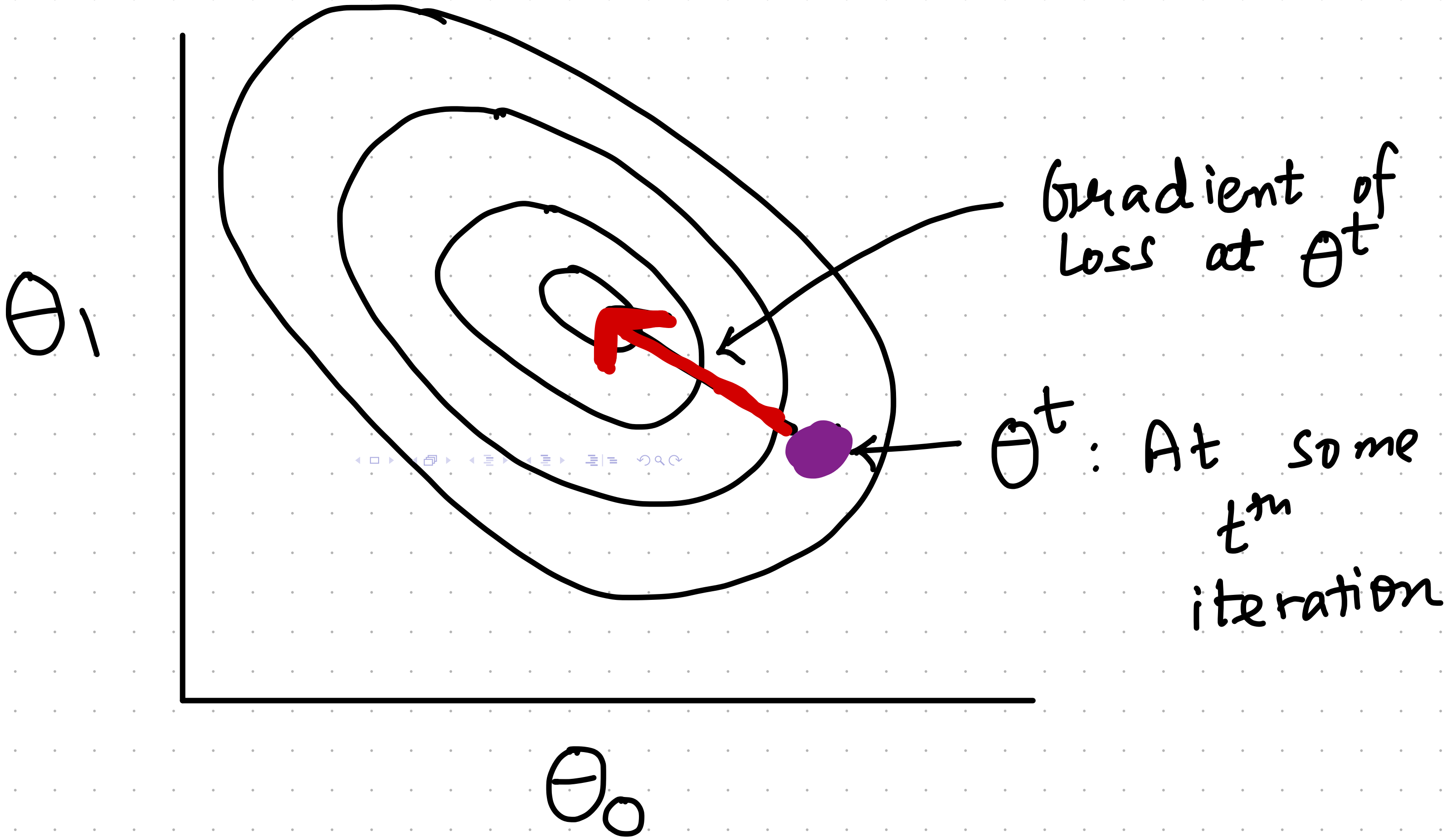
X	y	$\hat{y} = f(x, \theta)$
$— \, x_1^T \, —$	y_1	$\hat{y}_1$
$\vdots$		$\vdots$
$— \, x_N^T \, —$	y_N	$\hat{y}_N$



LOSS SURFACE OVER
'N' EXAMPLES



LOSS SURFACE OVER
'N' EXAMPLES



LOSS SURFACE OVER 6N EXAMPLES

Think! Solutions

Think! 1: Why does $-\nabla f$ lead toward minimum?
The gradient $\nabla f(\boldsymbol{x})$ points in direction of steepest *ascent*. To *descend* (minimize), we go in the opposite direction: $-\nabla f(\boldsymbol{x})$.

Think! Solutions

Think! 1: Why does $-\nabla f$ lead toward minimum?
The gradient $\nabla f(\boldsymbol{x})$ points in direction of steepest *ascent*. To *descend* (minimize), we go in the opposite direction: $-\nabla f(\boldsymbol{x})$.

Think! 2: Why might $\alpha > 2/L$ cause divergence?
For quadratic $f(x) = \frac{L}{2}x^2$, we have $f'(x) = Lx$. The update becomes:
$x_{t+1} = x_t - \alpha L x_t = (1 - \alpha L)x_t$.
If $\alpha L > 2$, then $|1 - \alpha L| > 1$, causing the sequence to diverge.

Pop Quiz Solutions

Pop Quiz #1: For 1000 samples, batch size 50:

1. Mini-batch iterations per epoch: $1000/50 = 20$
2. If SGD takes 1000 epochs, mini-batch might take ≈ 50 epochs (rough estimate)
3. SGD is noisier because it uses only 1 sample per update vs. all samples for batch GD

Pop Quiz Solutions

Pop Quiz #1: For 1000 samples, batch size 50:

1. Mini-batch iterations per epoch: $1000/50 = 20$
2. If SGD takes 1000 epochs, mini-batch might take ≈ 50 epochs (rough estimate)
3. SGD is noisier because it uses only 1 sample per update vs. all samples for batch GD

Additional insight: The noise in SGD can actually help escape local minima in non-convex optimization problems!

References & Further Reading

Essential references:

- Boyd & Vandenberghe: *Convex Optimization*
- Nocedal & Wright: *Numerical Optimization*
- Goodfellow et al.: *Deep Learning*, Chapters 4 & 8

References & Further Reading

Essential references:

- Boyd & Vandenberghe: *Convex Optimization*
- Nocedal & Wright: *Numerical Optimization*
- Goodfellow et al.: *Deep Learning*, Chapters 4 & 8

Interactive resources:

- Gradient descent visualizations online
- Implement from scratch in NumPy/PyTorch
- Experiment with different learning rates

References & Further Reading

Essential references:

- Boyd & Vandenberghe: *Convex Optimization*
- Nocedal & Wright: *Numerical Optimization*
- Goodfellow et al.: *Deep Learning*, Chapters 4 & 8

Interactive resources:

- Gradient descent visualizations online
- Implement from scratch in NumPy/PyTorch
- Experiment with different learning rates

Next lecture: Advanced Optimization Techniques
Practice: Implement GD for your favorite ML model!