# Gradient Descent in Gory Detail:

## OLS to Mini-batching and Beyond

Hands-on session linear regression + gradient descent

Jacob Munson & Will Hammond

Montana State University | TAILS | 10/17/2025 AD

## Learning goals

- Understand ordinary least squares (OLS) for linear regression:
  - Model
  - Loss
  - Closed-form solution
- Gradient descent:
  - Derive gradients of mean-squared error (MSE)
  - Implement gradient descent (GD) updates
- Compare GD flavors: batch GD vs. mini-batch vs. stochastic GD
- Diagnose convergence (learning rate schedules, conditioning, normalization)

# Motivation: Why Linear Regression?

**Scenario:** We observe data pairs $(x_i, y_i)$ and suspect a linear trend

- Goal: find a line $\hat{y}_i = \beta_0 + \beta_1 x_i$ that best fits the data
- Minimize Mean Squared Error (MSE):

$$J(\beta_0, \beta_1) = \frac{1}{n} \sum_i (\hat{y}_i - y_i)^2$$

- Captures linear input-output relationships
- Basis for understanding gradient descent
- **In the figure:** the dashed black line is the *true fit* ($y = 1.5x + 3$), the red line is the *OLS fit*, and gray dots are noisy observed data.

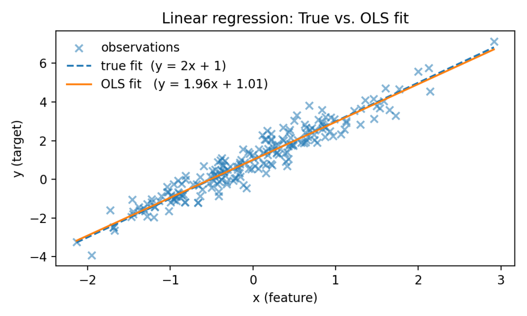



**Figure 1:** Simulated data and fitted line.

# Data and notation

**Matrix calculus:** simplifies gradients and updates for all parameters simultaneously.

- Let $X \in \mathbb{R}^{n \times d}$
  (rows are examples, columns are features)
- Response $y \in \mathbb{R}^n$
- Parameters $\beta \in \mathbb{R}^d$
- Generalize scalar case $(\beta_0, \beta_1)$ to vector form:
  - Predictions: $\hat{y} = X\beta$
  - Loss (MSE): $J(\beta) = \frac{1}{n}\|X\beta - y\|_2^2$

**Notation summary**

| symbol | shape | comment |
|---|---|---|
| $X$ | $n \times d$ | design matrix |
| $y$ | $n \times 1$ | targets |
| $\beta$ | $d \times 1$ | parameters |

# OLS: closed-form solution

$$J(\beta) = \frac{1}{n}(X\beta - y)^\top (X\beta - y)$$

$$\nabla_\beta J(\beta) = \frac{2}{n} X^\top (X\beta - y)$$

Set gradient to zero: $X^\top X\,\beta = X^\top y$

$$\Rightarrow \quad \beta^{\mathrm{OLS}} = (X^\top X)^{-1} X^\top y \quad (\text{if } X^\top X \text{ invertible})$$

# Warm-up: Paper & Pencil View

Given data points $(x_i, y_i)$, assume a linear model:

$$\hat{y}_i = \beta_0 + \beta_1 x_i$$

**Our goal:** find $\beta_0, \beta_1$ minimizing the Mean Squared Error (MSE)

$$J(\beta_0, \beta_1) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

**Exercise:**

1. Write down $J(\beta_0, \beta_1)$ explicitly in terms of $\beta_0, \beta_1$.
2. Compute partial derivatives:
$$\frac{\partial J}{\partial \beta_0} = \ldots, \quad \frac{\partial J}{\partial \beta_1} = \ldots$$
3. Write gradient descent step

# Gradient Descent: Elementwise & Matrix Form

## Elementwise Form

$$\hat{y}_i = \beta_0 + \sum_{j=1}^{d} x_{ij}\beta_j$$

$$J(\beta) = \frac{1}{n}\sum_i (\hat{y}_i - y_i)^2$$

$$\frac{\partial J}{\partial \beta_j} = \frac{2}{n}\sum_i x_{ij}(\hat{y}_i - y_i)$$

Gradient descent:

$$\beta_j^{(t+1)} = \beta_j^{(t)} - \eta \frac{\partial J}{\partial \beta_j}$$

## Matrix Form

$$\hat{y} = X\beta, \quad J(\beta) = \frac{1}{n}\|X\beta - y\|_2^2$$

Gradient:

$$\nabla_\beta J = \frac{2}{n} X^\top (X\beta - y)$$

Update rule:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \nabla_\beta J$$

**Same math, compactly!**

- Vectorization = efficiency
- Often easier for analysis and coding

# Gradient Descent

**Objective:** $J(\beta) = \frac{1}{n}\|X\beta - y\|_2^2$

**Gradient:**[1] $\nabla J(\beta) = \frac{2}{n}X^\top(X\beta - y)$

**Update:** $\beta^{(t+1)} = \beta^{(t)} - \eta\nabla J(\beta^{(t)})$

## Algorithm (pseudocode)

```
initialize beta randomly or zeros
repeat until convergence:
    grad = (2/n) * X.T @ (X @ beta - y)
    beta = beta - eta * grad
    (should do) monitor J(beta) or validation error
```

[1] We often drop the factor of 2 → "absorb into $\eta$"

# Gradient Descent Variants

**General update rule:**

$$\beta^{(t+1)} = \beta^{(t)} - \eta \, \frac{2}{|\mathcal{B}_t|} \, X_{\mathcal{B}_t}^{\top}(X_{\mathcal{B}_t}\beta^{(t)} - y_{\mathcal{B}_t})$$

- $\mathcal{B}_t$ is the *batch* of samples used at iteration $t$.
- The batch size $|\mathcal{B}_t|$ determines the variant.

**Special cases:**

- $|\mathcal{B}_t| = n \Rightarrow$ **Batch Gradient Descent**
  Deterministic, stable but slow for large $n$.
- $1 < |\mathcal{B}_t| < n \Rightarrow$ **Mini-batch (Stochastic) Gradient Descent**
  Balances efficiency and stability; default in deep learning.
- $|\mathcal{B}_t| = 1 \Rightarrow$ **Stochastic Gradient Descent**
  Highly stochastic updates; rarely used in pure form.

*Note:* In practice, "SGD" often refers to the mini-batch case rather than strictly $B=1$.

# Full Algorithm: Mini-batch Gradient Descent

```
# Mini-batch Gradient Descent for Linear Regression
import numpy as np

# X: n x d matrix, y: n-vector
n, d = X.shape
beta = np.zeros(d)
eta = 0.1 # learning rate
B = 32 # batch size
n_epochs = 100

for epoch in range(n_epochs):
    perm = np.random.permutation(n)
    X, y = X[perm], y[perm]
    for i in range(0, n, B):
        idx = slice(i, i+B)
        Xb, yb = X[idx], y[idx]
        grad = (2/B) * Xb.T @ (Xb @ beta - yb)
        beta -= eta * grad

print("Learned coefficients:", beta)
```

# Ridge (L2) regularization

Objective: $J_\lambda(\beta) = \frac{1}{n}\|X\beta - y\|^2 + \lambda\|\beta\|^2$.
Closed form: $\beta = (X^\top X + n\lambda I)^{-1} X^\top y$.
Gradient: $\nabla J_\lambda = \frac{2}{n} X^\top (X\beta - y) + 2\lambda\beta$.

Effect: shrinks coefficients, improves conditioning of $X^\top X$; GD tolerates larger $\eta$.

# Compare OLS vs GD variants

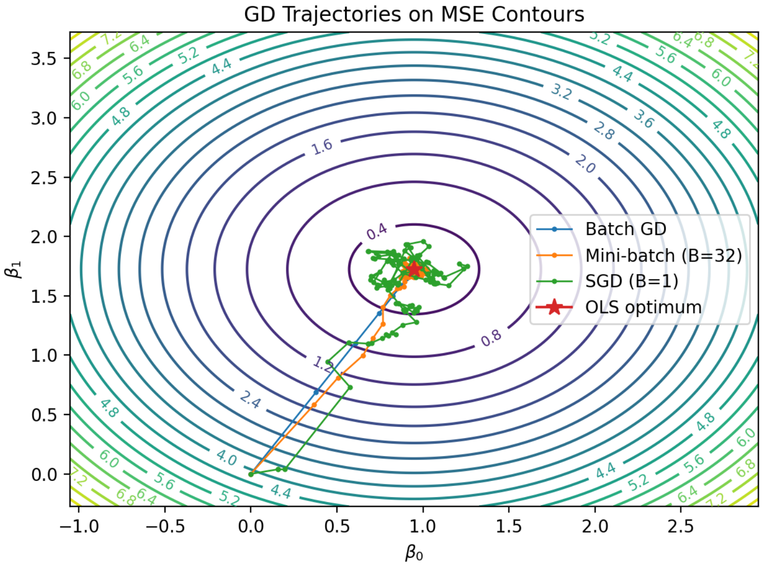
**Metrics**

- Parameter error $\|\beta_{\text{est}} - \beta_{\text{OLS}}\|$.
- Train/validation MSE.
- Time-to-$\varepsilon$ and epochs to converge.
- Sensitivity to feature scaling.

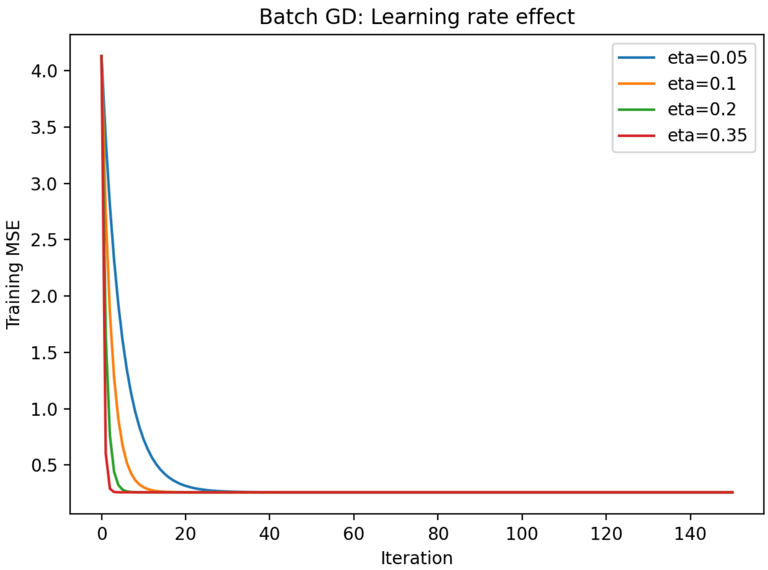
**Why pick one over another?**

- OLS is one-shot but unstable for ill-conditioned/huge $X$.
- GD scales to large $n, d$ and streaming data.
- Mini-batch leverages hardware; SGD can escape shallow minima in nonconvex problems.

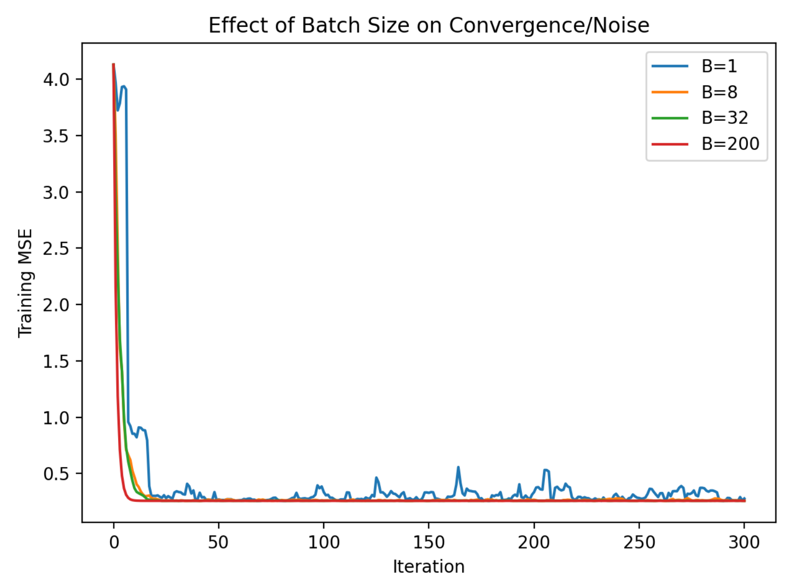
# GD trajectories on MSE contours



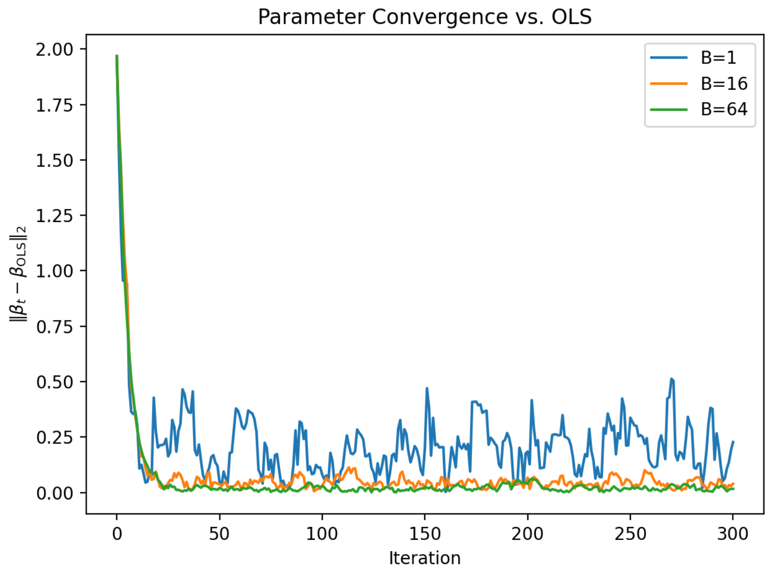
# Learning rate effect (batch GD)

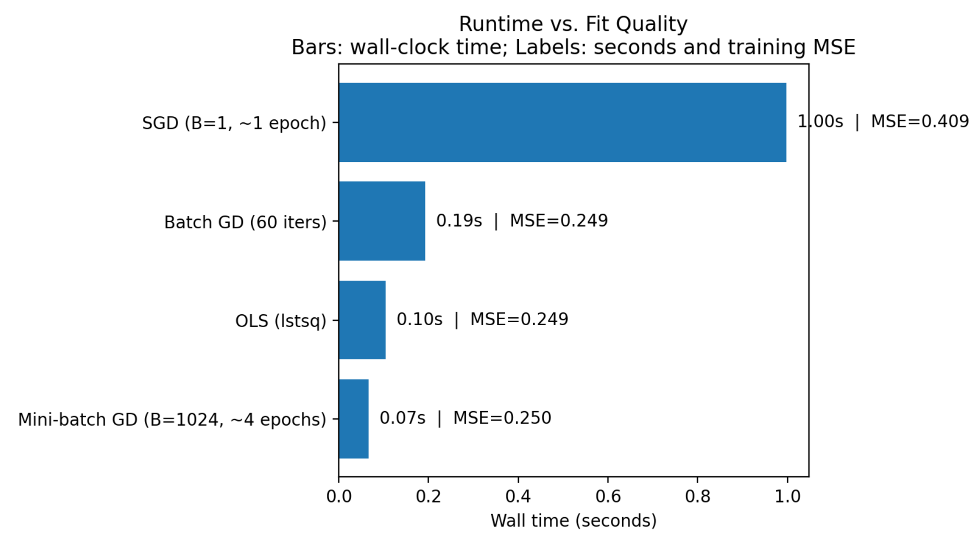


# Batch size vs. convergence noise



# Parameter convergence toward OLS




Runtime vs. Fit Quality
Bars: wall-clock time; Labels: seconds and training MSE
SGD (B=1, ~1 epoch)
Batch GD (60 iters)
OLS (lstsq)
Mini-batch GD (B=1024, ~4 epochs)
1.00s | MSE=0.409
0.19s | MSE=0.249
0.10s | MSE=0.249
0.07s | MSE=0.250
0.0
0.2
0.4
0.6
0.8
1.0
Wall time (seconds)

# Complexity and Runtime Takeaways

**Computational Complexity**

- **Closed-form OLS:** $\mathcal{O}(nd^2 + d^3)$ time
- **Batch Gradient Descent:** $\mathcal{O}(nd)$ per iteration
- **Mini-batch / SGD:** $\mathcal{O}(Bd)$ per step

**Runtime Insights**

- **OLS:** One-shot and very fast for small $d$ and moderate $n$ (exact up to FP precision)
- **GD/Mini-batch:** Preferred for large $n$ or streaming data
- **Practical tuning:** Tune learning rate $\eta$; monitor wall time and validation error.

# Common pitfalls

- Forgetting intercept column or feature scaling.
- Learning rate too big (divergence) or too small (stalling).
- Not shuffling mini-batches; data order bias.
- Evaluating only training MSE; always hold out validation.

**Thanks! Questions + discussion.**