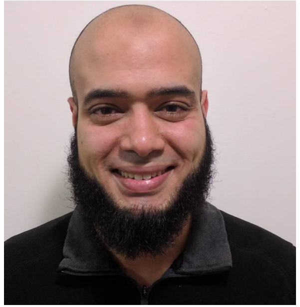
Machine Learning

Linear Regression using Gradient Descent

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Question!

- Assume our function is $f(x, y) = 4 * (1 - x)^2 + (y - 5)^2$
- We started gradient descent from point **(x=3, y=2)**
- Using learning rate of 0.5, what is the first updated x, y in the looping

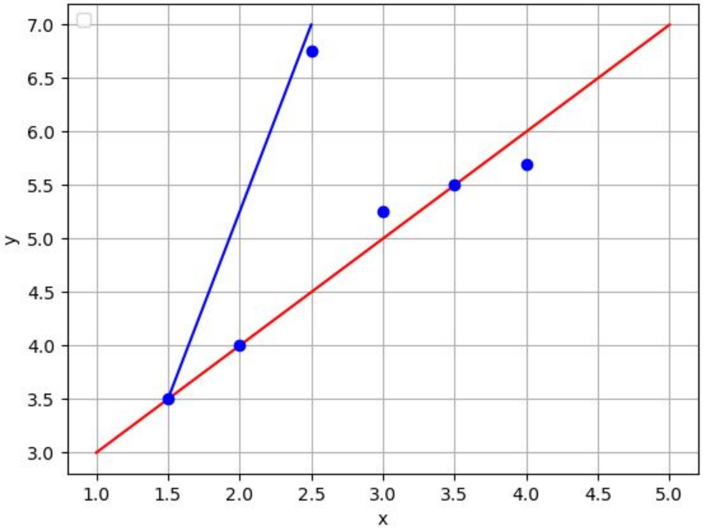
```python
# let's pretend answer is
M = [2, 3.5, 4, 10, 5]
C = [1, 6, 3]
# ground truth (X, Y)
X = [1, 2, 3, 4, 5, 6]
Y = [6, 5, 4, 3, 2, 1]

def compute_cost(m, c):
    cost = 0
    for (x, y_gt) in zip(X, Y):
        y_pd = m * x + c
        err = y_gt - y_pd
        squared_err = err ** 2
        cost += squared_err
    return cost / len(Y) / 2

if __name__ == '__main__':
    best_cost = float("inf")
    for m in M:
        for c in C:
            this_cost = compute_cost(m, c)

            if best_cost > this_cost:
                best_cost = this_cost
```

Recall

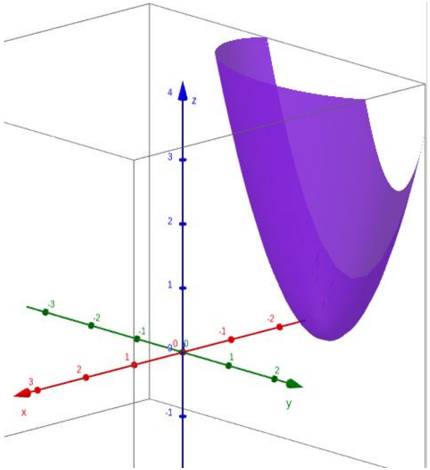


$$\underset{m,c}{\text{minimize}}\, cost(m, c)$$

$$cost(m, c) = \frac{1}{2N} \sum_{n=1}^{N} ((mx^{(n)} + c) - t^{(n)})^2$$

Is LR MSE function convex?

- Yes, for the linear regression, MSE is a **convex** function
    - Intuitively looks convex, but prove requires learning *Hessian matrix and Eigenvalues*
        - Ask ChatGpt: *Prove that linear regression with MSE error function is convex*



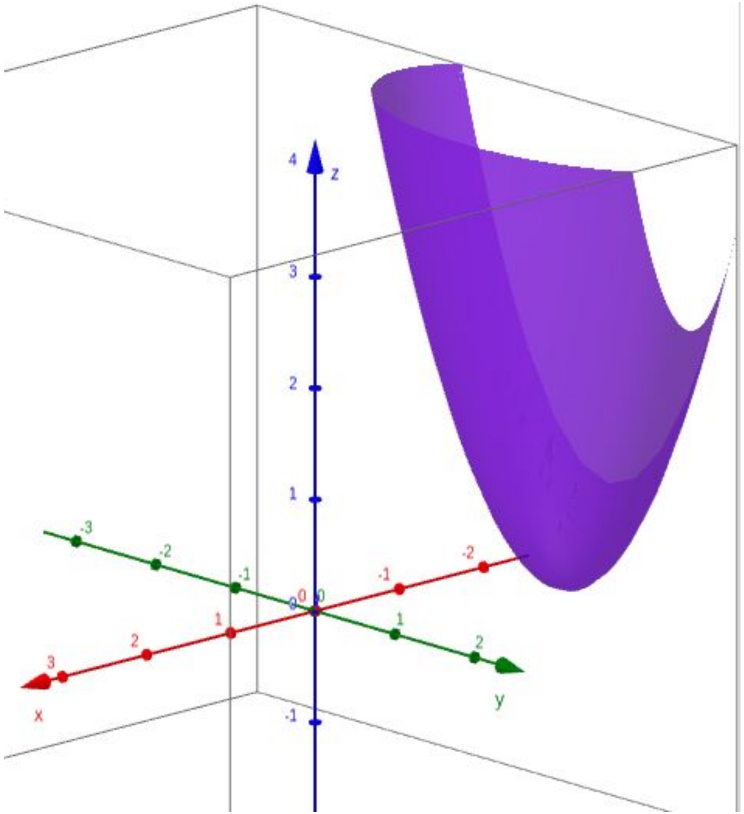
$$\underset{m,c}{\text{minimize}}\, cost(m,c)$$

$$cost(m,c) = \frac{1}{2N}\sum_{n=1}^{N}((mx^{(n)}+c)-t^{(n)})^2$$

Our function

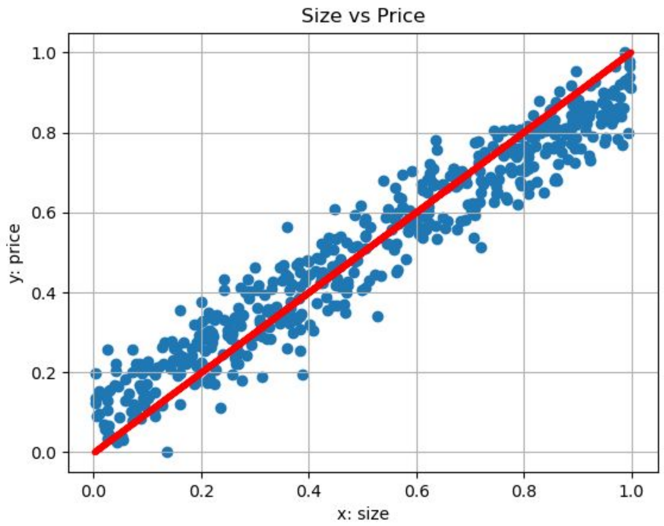
- If the x-axis represents m and the y-axis represents c
- What is the z-axis?

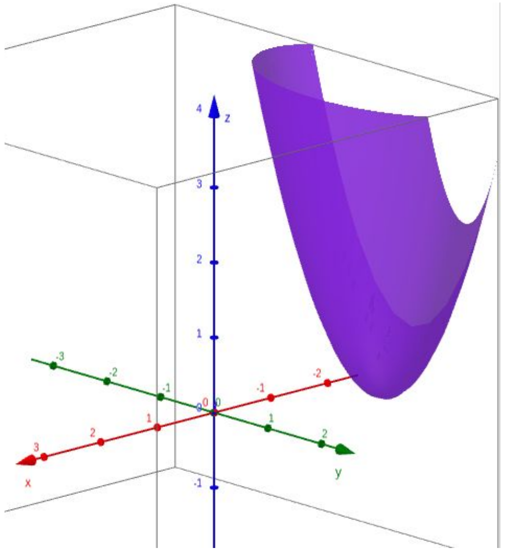
$$\underset{m,c}{\text{minimize}}\, cost(m, c)$$



2 Different Graphs

- We have m points
- On the right, the function f(m, c) tells us how good a specific pair (m, c) is
- We use the optimal (m, c) to represent the line of such data





Line of best fit!

- A linear regression algorithm determines the **line-of-best-fit** by minimizing the **sum of squared error terms** (between target and prediction)
  - Also known as ‘Linear least squares’
- In 2D, the line equation is: $y = mx+c$
  - It has 2 parameters to find (m, c)
- Once we have the parameters, we can predict new queries
  - If the data really comes from a straight line, the parameters can be generated from any two points. However, this is typically not the case
  - In practice, there will be noise. We need to find pair (m, c) that has the minimum cost!
- We can compute the derivatives and find a **closed formula**
- But, let’s use a **general method** that will help us in more scenarios

Using Gradient Descent (GD)

- Recall solving: $f(x, y) = 2x^2 - 4xy + y^4 + 2$
  - Applying GD
    - Partial derivative on x then y.
    - Change x and y towards the local minimum
- Our function cost(m, c) is EXACTLY the same thing
  - The only difference is that the cost is based on **m** training examples
- Question: What is the partial derivative of LR cost function relative to m?

$$\text{cost(m, c)} = \frac{1}{2m}\sum_{i=1}^{m}((mx^{(i)} + c) - y^{(i)})^2$$

Using Gradient Descent (GD)

- Let's assume we have only 3 training examples
- Then our function cost equals the following:
  - ⅙ (m * x1 + c - y1)^2 +
  - ⅙ (m * x2 + c - y2)^2 +
  - ⅙ (m * x3 + c - y3)^2
    - xs and ys here are input data NOT variables. Variables are m and c
- Compute partial derivatives relative to m, c

$$cost(m,c) = \frac{1}{2N}\sum_{n=1}^{N}((mx^{(n)}+c)-t^{(n)})^2$$

Partial Derivative: Single training example

- Let's apply the partial derivative on the first term only:
- $\frac{1}{6} (m * x1 + c - y1)^2$
- With respect to m: df/dm
  - $\frac{1}{6}$, x1, y1, c are just constants
  - $f^2 \Rightarrow 2 * f * f'$
- df/dm = $\frac{1}{3} (m * x1 + c - y1) * x1$
- df/dc = $\frac{1}{3} (m * x1 + c - y1)$

Partial Derivative: The whole dataset

- To compute df/dm and df/dc for the whole dataset, we just **sum the N terms** of the m training examples!

$$cost(m,c) = \frac{1}{2N}\sum_{n=1}^{N}((mx^{(n)} + c) - t^{(n)})^2$$

$$\frac{\partial cost(m,c)}{\partial m} = \frac{1}{N}\sum_{n=1}^{N}((mx^n + c) - t^n) * x^n$$

Any interesting observation?

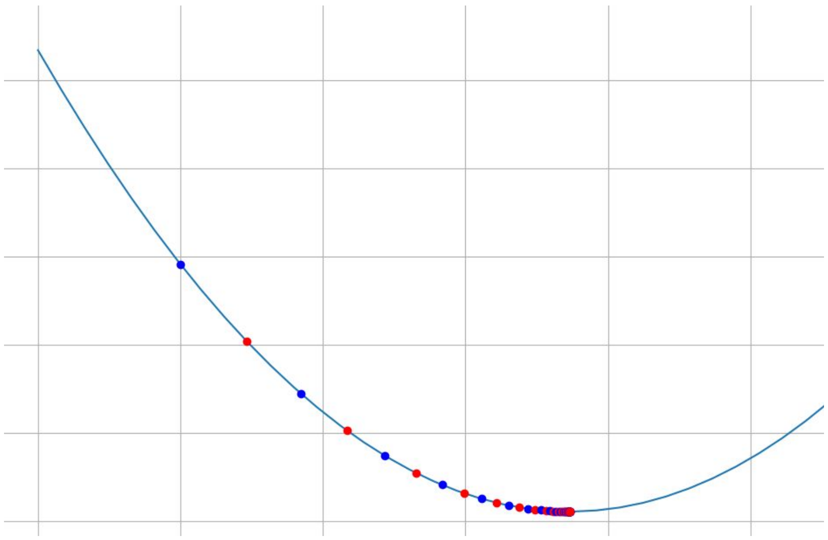
$$\frac{\partial cost(m,c)}{\partial c} = \frac{1}{N}\sum_{n=1}^{N}((mx^n + c) - t^n)$$

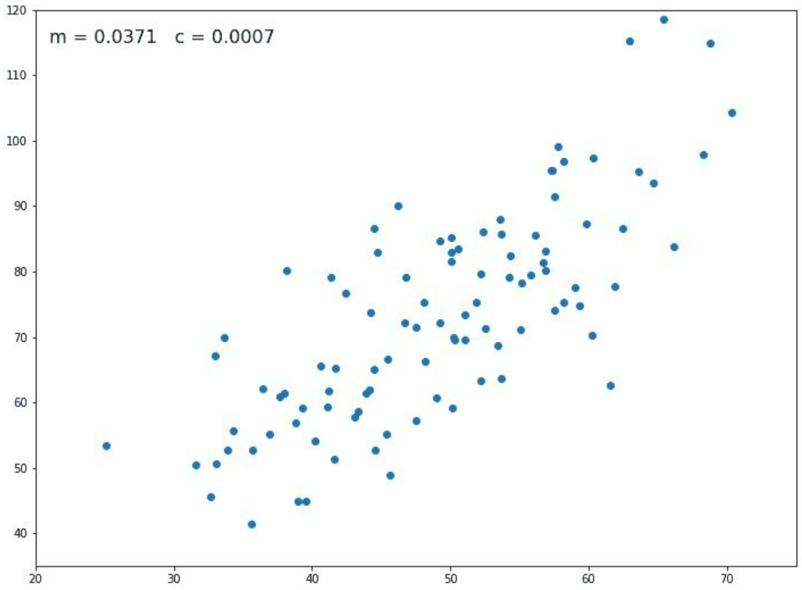
The Least Mean Squares (LMS) Update Rule

- Observe, the magnitude of the update is **proportional** to the error term
    - Error: prediction - target
- If the error is zero (perfect guess), there is no update!
    - This is great, as the good weights don't get changed!
- If the error is small, the change is small. The reverse is also true!

$$\frac{\partial cost(m,c)}{\partial m} = \frac{1}{N}\sum_{n=1}^{N}((mx^n + c) - t^n) * x^n$$

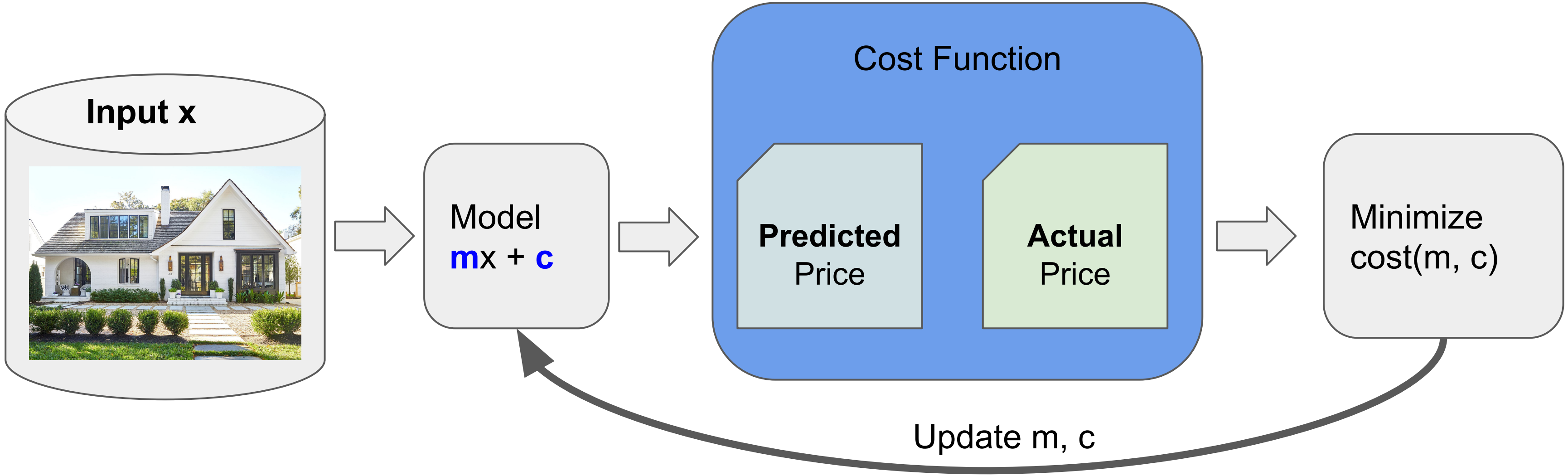
Parameters (m, c) and their line

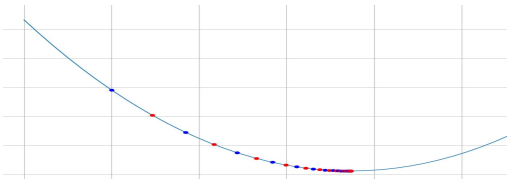


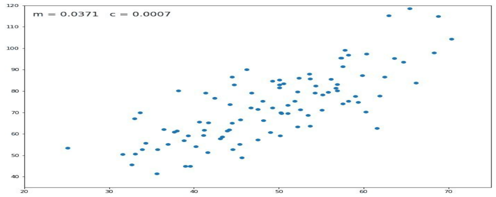


Img src

Supervised learning







ML Gradient Descent Variants: Comparison

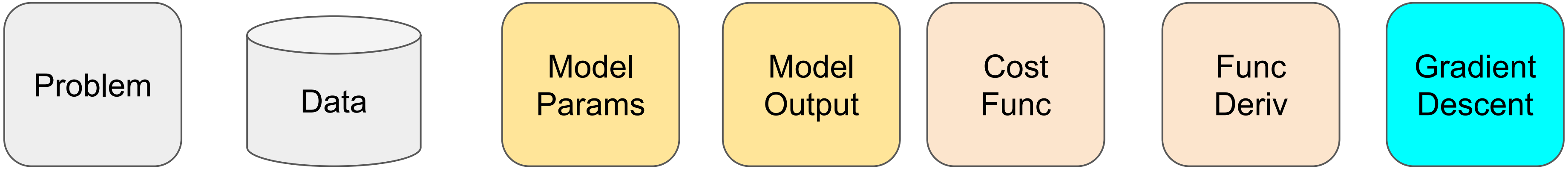
- Batch Gradient Descent
  - As we did, in each iteration, **use all N training examples** to make a single update to the parameters
  - Update weights based on all N examples at once
- Stochastic Gradient Descent
  - Shuffle your data
  - For **each single** training example:
    - Update weights based on this example only
  - Stochastic? This means that each training example is randomly sampled
- Mini-Batch Gradient Descent
  - Divide the N examples to mini-batches; each batch has S random training examples
  - For each mini-batch
    - Update weights based on all S examples of the batch

ML Gradient Descent Variants: Comparison

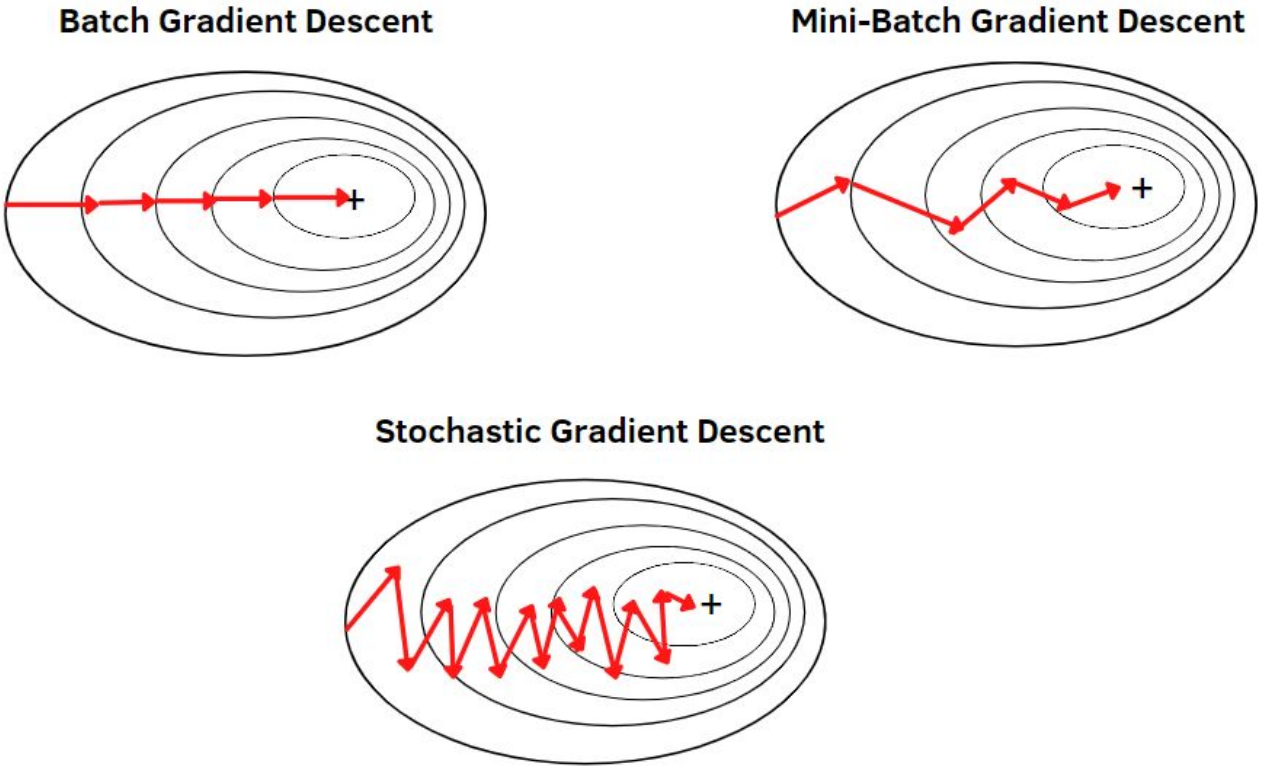
- Gradient Descent
  - It's not suitable for use with large-scale datasets...as we need to process all data at once
- Stochastic Gradient Descent
  - Pros: better suited to a large-scale dataset
  - Pros: Faster to converge (less data to make a decision)
  - Pros: Faster for visualization / debugging problems
  - Cons: As a gradient approximation (entire dataset vs a single random example), the updates have a higher variance (the convergence path of SGD is **noisier**)
    - Might have a higher error than gradient descent
- Mini-Batch Gradient Descent
  - A 'meet in the middle' solution between the two approaches (**practical**)
  - Cons: hyperparameter to tune: batch-size, but in practice not an issue (e.g. 64-256)

Modeling Style So Far

- We will see such treatment a lot
- Define a problem and collect its data
- Select a suitable model
- Given data and the model, we get the model output
- Compute cost function and its derivative
- Apply gradient descent (generic technique)



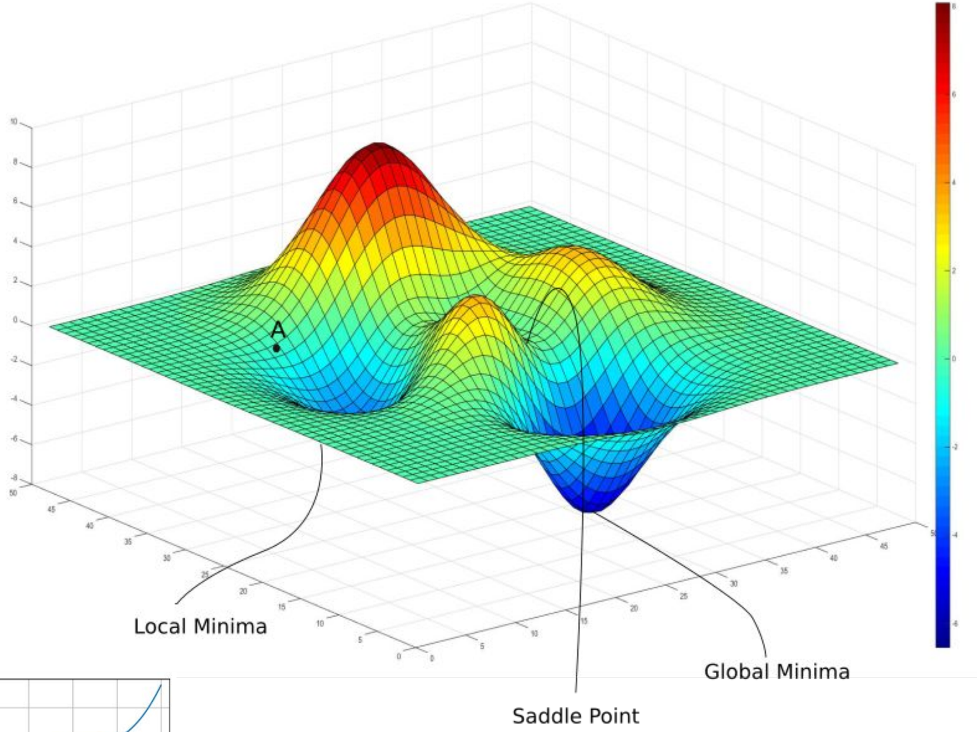
ML Gradient Descent Variants: Visualization

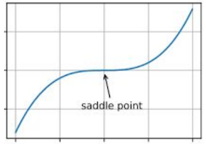


Img src

Escaping Saddle Points

- In later complex ML models, we will have several **saddle point!**
- Stochastic/mini-batch seem to have a good chance of skipping them due to their *noisy gradients*
- 2) Run the optimization algorithm from several initial random points
- **Red**: max - **Blue**: min

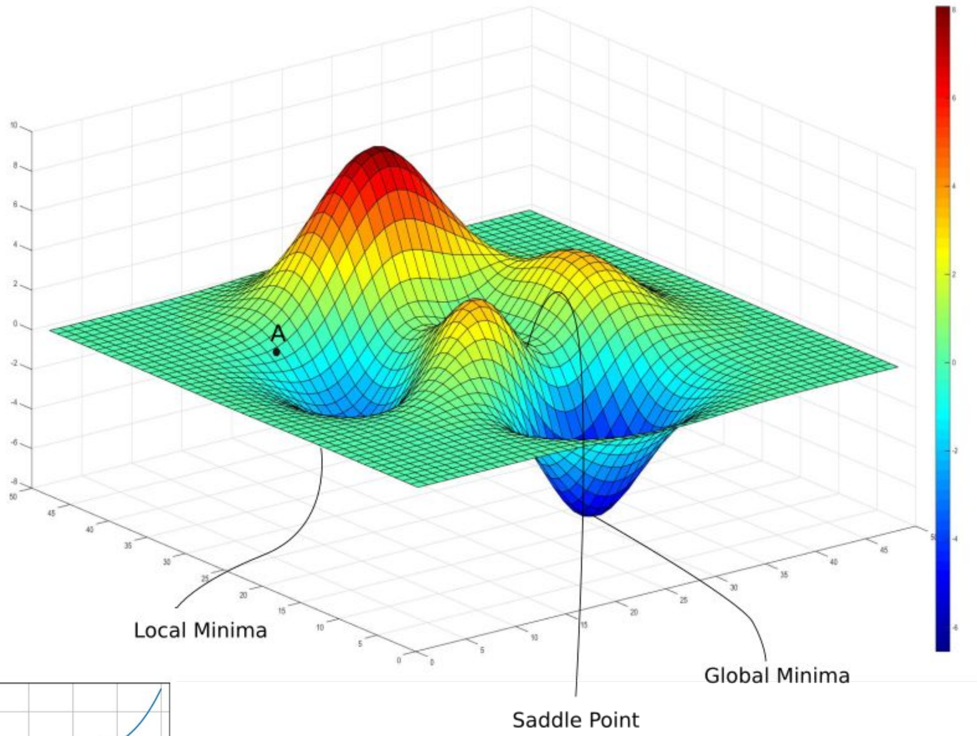


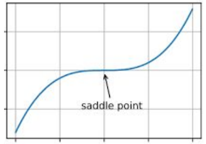


Img src

Escaping **some** of the local minima Points

- Interestingly, due to the noise nature of SGD gradients, it can even **jump out** some of the local minima and reach a better minima
    - Research Question
    - In practice, it can be better than even GD itself (for non-convex functions)
- However, still with a huge research space for the cost function, we still ends in some local minima

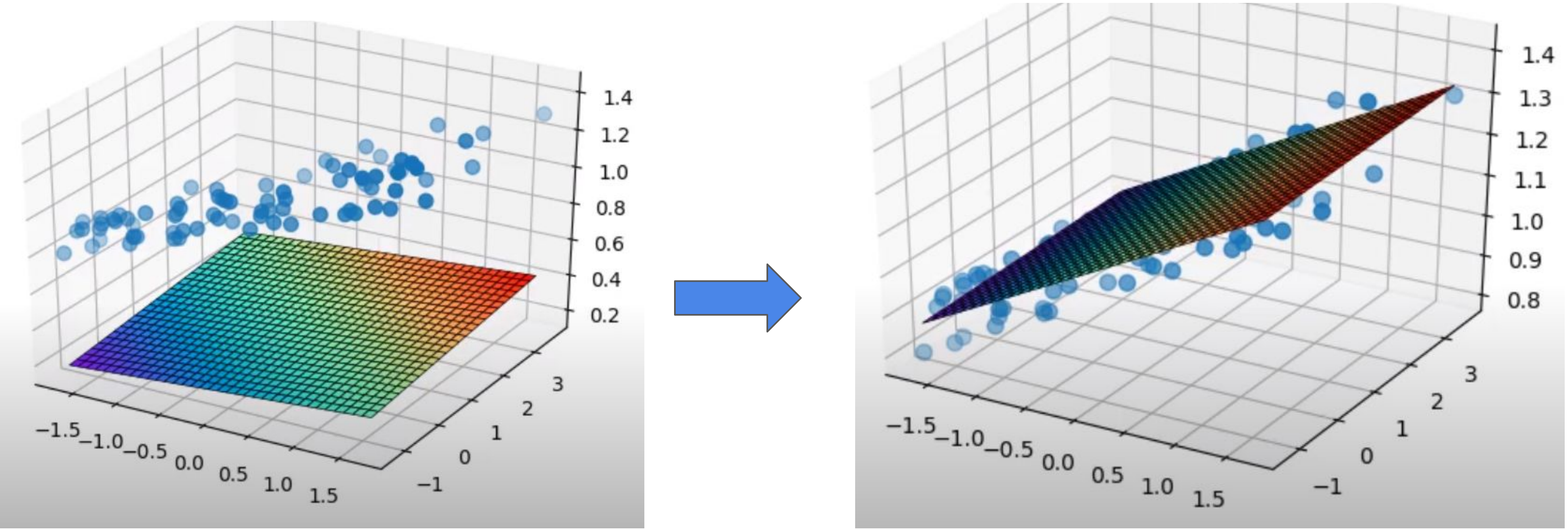




Img src

Multivariable Regression

- So far we were modeling an input vector that has a **single feature**
  - For example, determining the price of a property from its size
- For 1 feature, we need 2 parameters (representing a **line**: m & c)
  - C: The intercept helps the line freedom (don't just pass with the origin)
- In practice, we can have many input features (multivariable input)
- For D features, we need D+1 parameters (representing a **hyperplane**)
  - +1 for the intercept to allow the hyperplane freedom
    - We call it the **bias** in machine learning
  - Then, we do partial derivative to **D+1 variables**
- Overall, it is the same treatment!
  - Just more variables and hence more corresponding parameters
  - Harder to visualize!


1.4
1.2
1.0
0.8
0.6
0.4
0.2
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
−1 0 1 2 3
1.4
1.3
1.2
1.1
1.0
0.9
0.8
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
−1 0 1 2 3

Img src

Prediction **Function** Generalization

- Let's rewrite the new prediction function of **D+1 weights** and vectorize it for the **nth training** example

$$y(X^n, W) = 1 * W0 + X_1^n * W_1 + X_2^n * W_2 + ... + +X_d^n * W_d$$

$$y(X^n, W) = \sum_{j=0}^{d} X_j^n W_j$$

$$y(X^n, W) = W^T * X^n$$

- n is the nth example
- X is 2D matrix: X[N][D+1]
- Transpose for matrix multiplication correctness

- Observe, the partial derivative of $y(X^n, W)$ relative to Wj = X_{nj}
- This can be implemented as: np.dot(W, X)

Cost Function Generalization

- Tip: **Cost = Error = Loss** ⇒ something to **minimize**
- Now, just plugin the cost function
  - Notice how the derivative is still in a similar format to mx+c partial derivatives

$$cost(W) = \frac{1}{2N} \sum_{n=1}^{N} (y(X^n, W) - t^n)^2$$

$$\frac{\partial cost(W)}{\partial W_j} = \frac{1}{N} \sum_{n=1}^{N} (y(X^n, W) - t^n) * X_j^n$$

Vectorization

- Culture in C++/Java/C# and others are very iterative
- On the other hand, Python follows Pythonic coding
- If you are looping a lot, ask yourself if we can vectorize things
- Why vectorization:
    - Readable code: np.dot(W, X)
    - Very fast (optimized linear algebra libraries)
    - Matrix multiplication is speedy on a Graphics Processing Unit (GPU)
        - GPUs play a huge role in the success of Deep Learning

Using Scikit-Learn library

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error

# Create linear regression object
model = linear_model.LinearRegression()

# Train the model using the training sets
model.fit(x, t)
optimal_weights = model.coef_
print(f'Sikit parameters: {optimal_weights}')

# Make predictions using the testing set
pred_t = model.predict(x)
# divide by 2 to MATCH our cost function
error = mean_squared_error(t, pred_t) / 2

print(f'Error: {error}')
```

There is also model.**intercept_**

MSE as evaluation Metrics

- We used MSE metric for optimizing the function
- It is also a good metric to evaluate the dataset
    - Error = 0: most optimal. No limit on max
- So after finding the best parameters, compute the MSE
- This tells you how good is the model
    - But outliers can be a problem (learn soon)

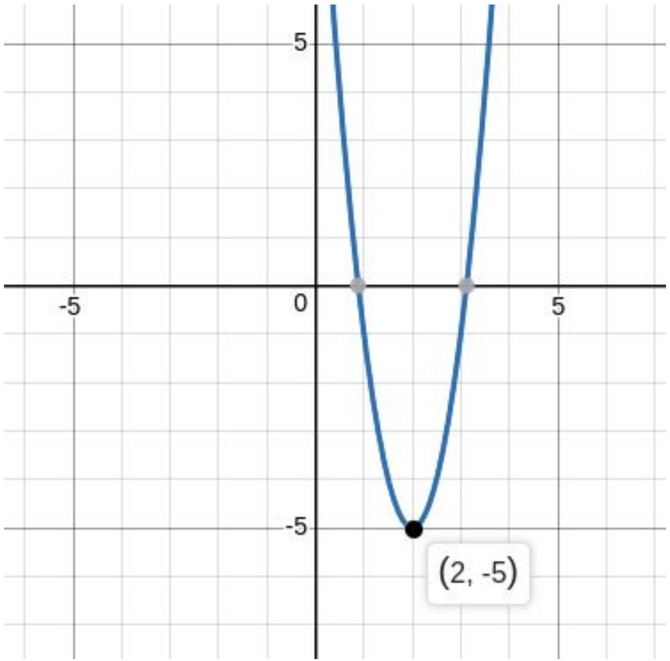
MSE vs R-Squared (after course)

- There is another **evaluation** metric (still learn with MSE)
- It is called R-squared score (see scikit)
  - Represents the fraction of **variance** captured by the regression model
  - Less sensitive to outliers. MSE is sensitive
  - R-squared values range from 0 to 1, where 1 is perfect
  - Gives better interpretability comparison between models
  - Some people find it useless / confusing
- Stat Quest video
- Article: MSE vs R-squared
- Also Adjusted R Squared Formula

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}$$

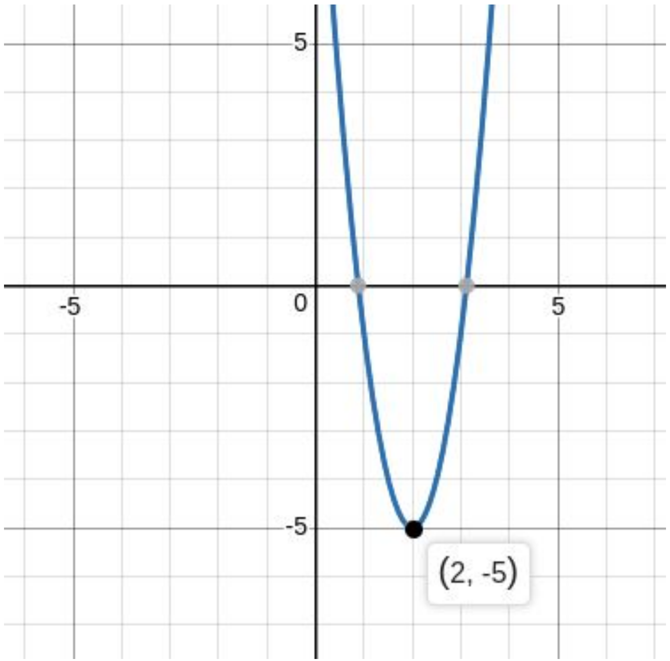
Question!

- What is the minimum Y of this quadratic equation: $Y=(2X-4)^2 - 5$
- Find 2 ways



Recall Closed Formula

- What is the minimum Y of this quadratic equation: $Y=(2X-4)^2 - 5$
- **Set derivative = 0**
- $F' = (2X-4) * 2 = 0$
- $X = 4/2 = 2$
- Then at $X = 2$, the minimum is $Y = -5$



Applying Closed Formula

- Why didn't we just apply the derivative for the **quadratic convex cost** function?
- In fact, we can. We can also end up with a closed formula (**rare** in ML)
- Simple linear regression (single feature: mx+c) has a formula
  - In the homework, we will derive it. Set derivative = 0
- Multivariate linear regression
  - Again, set the derivative to 0 and end up with the formula (**Normal Equations / OLS**)
  - But this requires more math skills / **matrix calculus** for multivariates

$$\Theta = (X^T X)^{-1} X^T y$$

- X is the training matrix, y is the ground truth and theta offers the best weights/parameters

Matrix Calculus

- Matrix calculus is a specialized **notation** for doing multivariable calculus for partial derivatives of a single function with respect to **many variables**, and/or of a **multivariate function** with respect to a single variable, into vectors and matrices that can be treated as **single** entities.
- This greatly **simplifies** operations
- There are sheet cheats for the derivatives
    - It is good to learn how to derive
- There are cheatbooks for the rules

Scalar derivative $f(x) \to \frac{\mathrm{d}f}{\mathrm{d}x}$	Vector derivative $f(\mathbf{x}) \to \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}$
$bx \to b$	$\mathbf{x}^T\mathbf{B} \to \mathbf{B}$
$bx \to b$	$\mathbf{x}^T\mathbf{b} \to \mathbf{b}$
$x^2 \to 2x$	$\mathbf{x}^T\mathbf{x} \to 2\mathbf{x}$
$bx^2 \to 2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x} \to 2\mathbf{B}\mathbf{x}$

Normal Equations vs Gradient Descent

- So, overall, there are **two approaches**: iterative and formula!
    - Normal Equation is an **analytical approach** (formula)
    - **Pros**
        - No hyperparameters (learning rate). No iterations
        - No data preprocessing
        - Good for small datasets
    - **Cons**
        - Requires ALL data in memory, which is impractical (min-batch wins)
        - **Impractical**: for **high** D-dimension inputs, we need $O(D^3)$ for the matrix inversion
        - Gradient descent in return is a **general technique**
- See homework

$$\Theta = (X^T X)^{-1} X^T y$$

Relevant Resources

- Prof Andrew Ng: Video
- Linear Regression - Eng Hesham Asem: Video
- Normal Equation - Prof Andrew Ng - Video
- Normal Equation Non Invertibility - Prof Andrew Ng - Video
- Formulation of Normal Equation method
- Normal equation solution of the least-squares problem - Video
- Linear Regression - Playlist
- Matrix Calculus: link - Matrix Cookbook - cheat sheet

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."