LECTURE 20

Gradient Descent

Optimization methods to analytically and numerically minimize loss functions.

Data 100/Data 200, Fall 2021 @ UC Berkeley

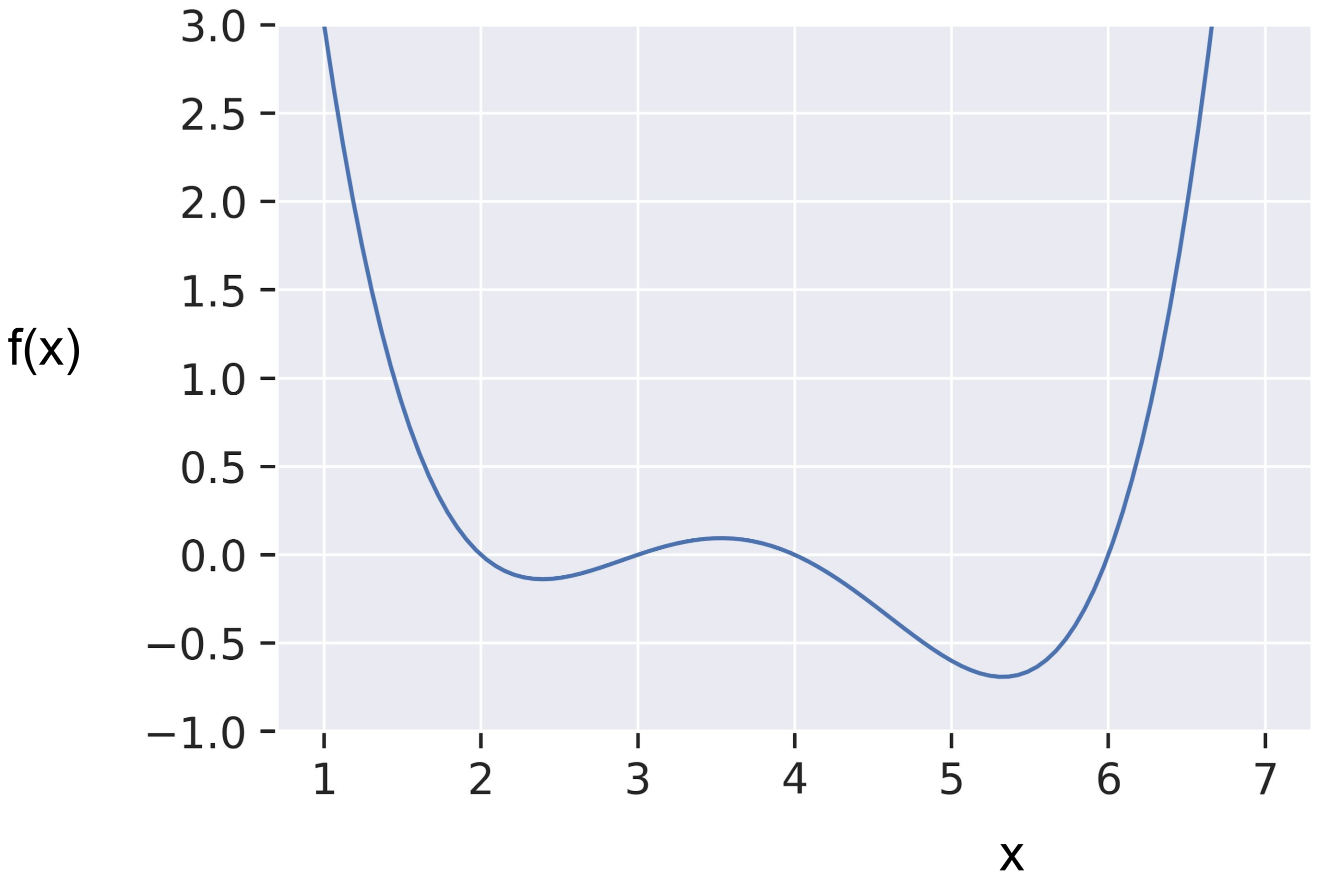
Fernando Pérez and Alvin Wan
(content by Josh Hug and Joseph Gonzalez)

Gradient Descent in 1D

Minimizing a Function

Suppose we want to minimize a function $f(x) = x^4 - 15x^3 + 80x^2 - 180x + 144$

- Many approaches for doing this.
- We'll discuss one approach today called "gradient descent".

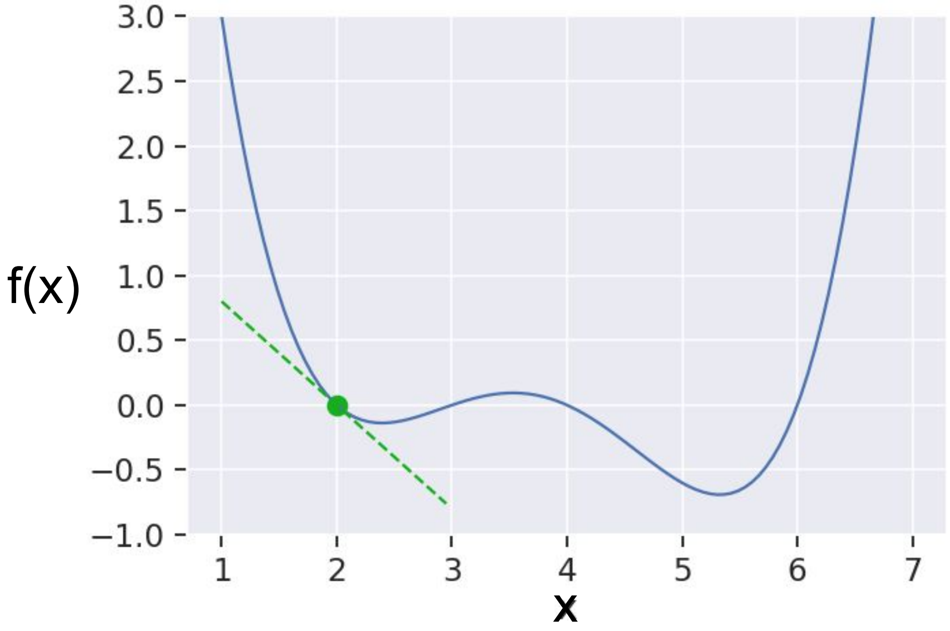


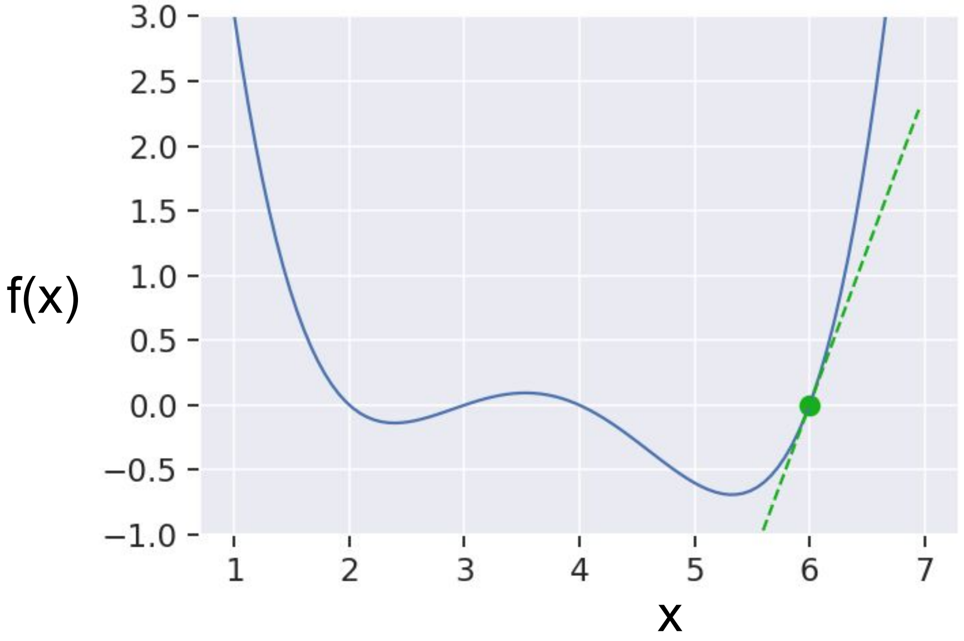
Gradient Descent Intuition

The intuition behind 1D gradient descent:

- To the left of a minimum, derivative is negative (going down).
- To the right of a minimum, derivative is positive (going up).
- Derivative tells you where and how far to go.

Let’s work from here and try to invent gradient descent.





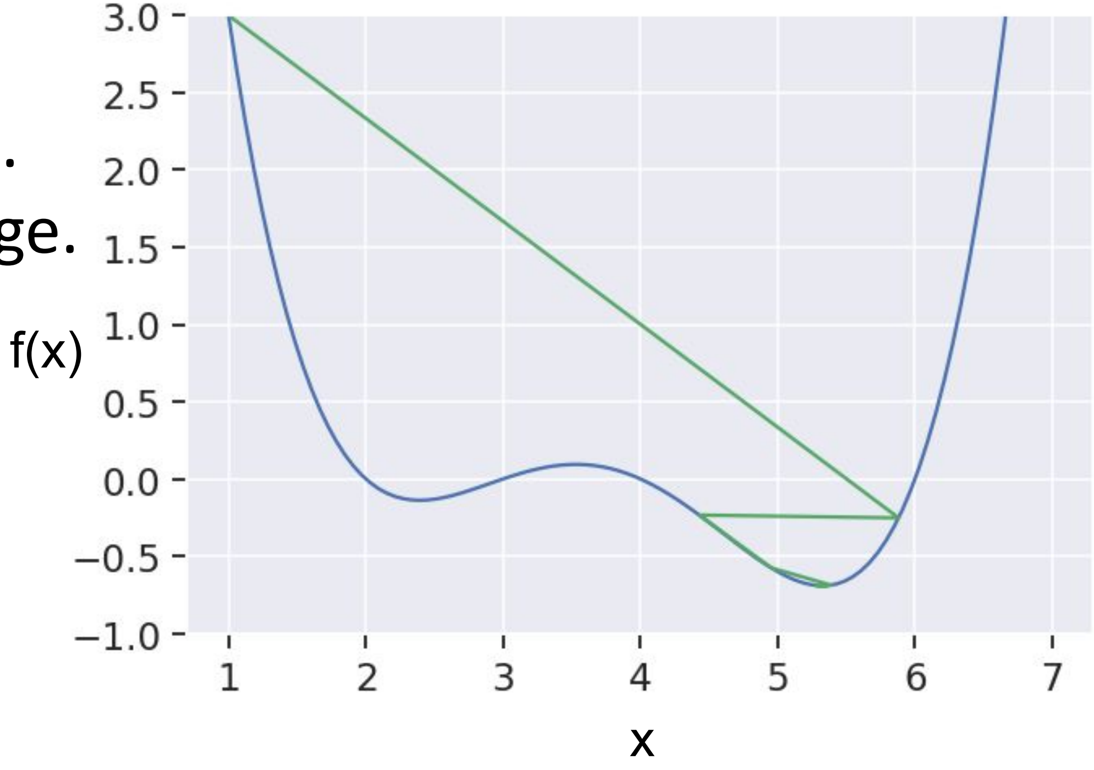
Gradient Descent Algorithm

The gradient descent algorithm is shown below:

- alpha is known as the "learning rate".
  - Too large and algorithm fails to converge.
  - Too small and it takes too long to converge.

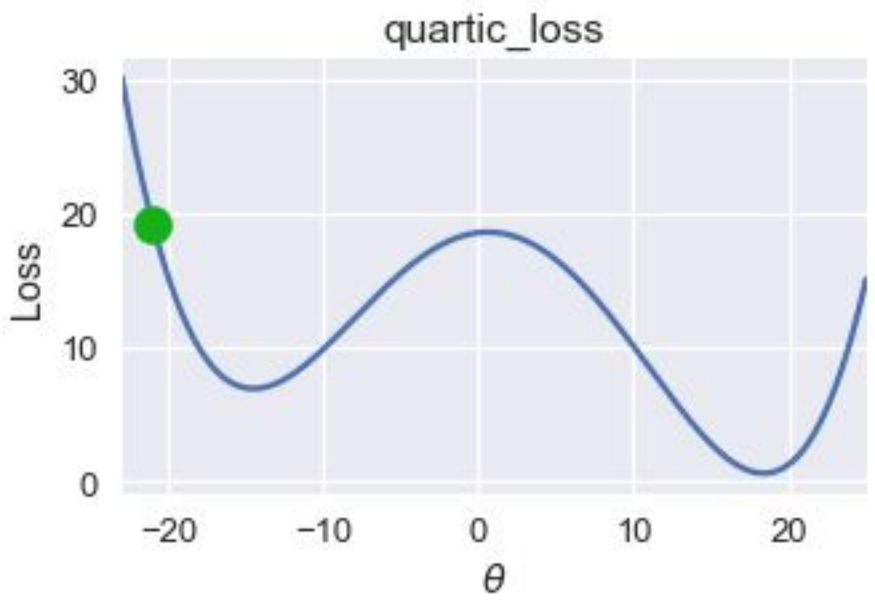
$$x^{(t+1)} = x^{(t)} - \alpha \frac{d}{dx} f(x)$$

```python
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```



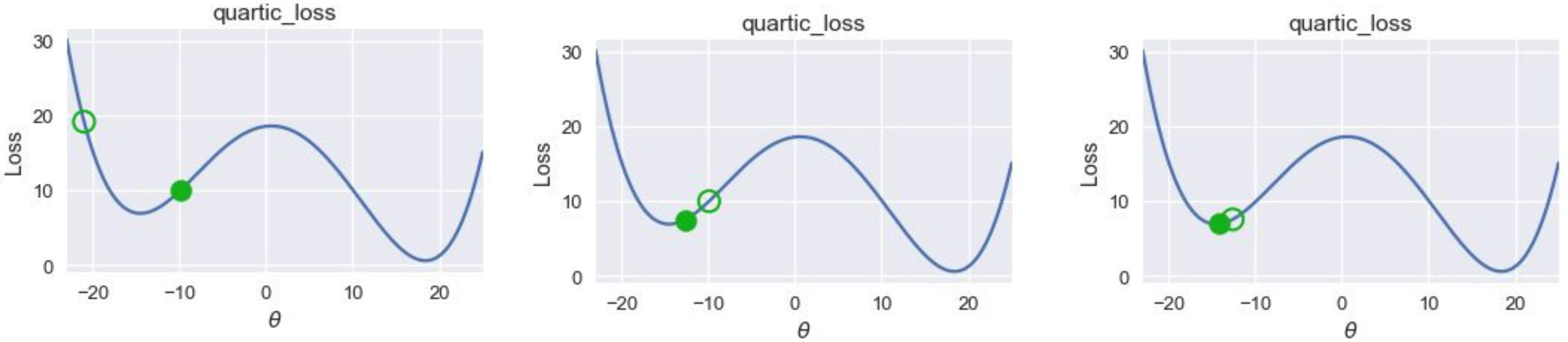
Gradient Descent Only Finds Local Minima

- If loss function has multiple local minima, GD is **not guaranteed** to find global minimum.
- Suppose we have this loss curve:



Gradient Descent Only Finds Local Minima

- Here’s how GD runs:



- GD can converge at -15 when global minimum is 18

Convexity

- For a **convex** function f, any local minimum is also a global minimum.
  - If loss function convex, gradient descent will always find the globally optimal minimizer.
- Formally, f is convex iff:

$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

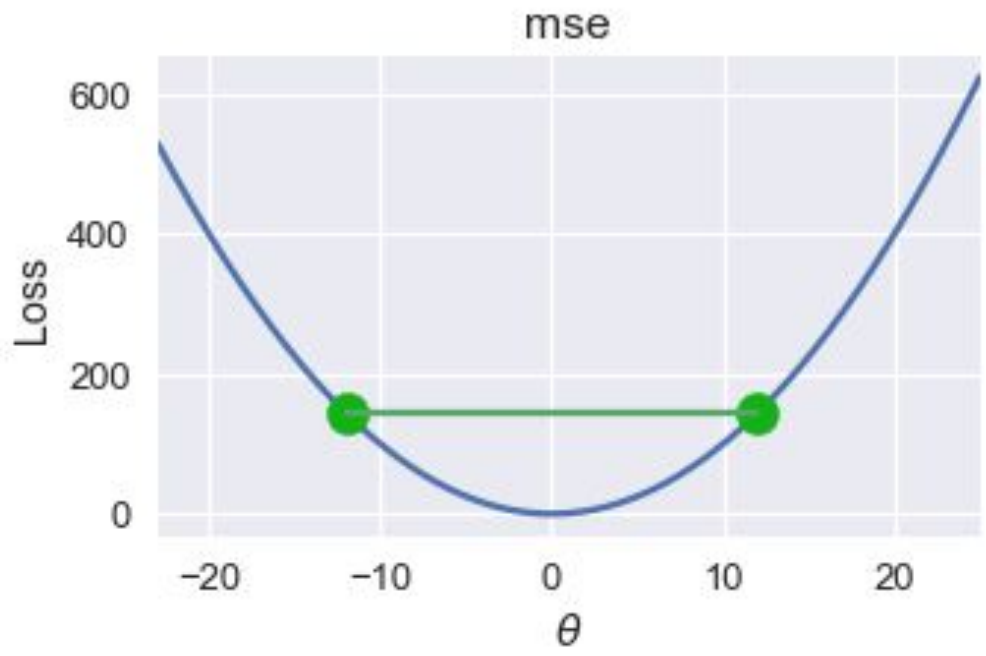
$$\text{For all } a, b \text{ in domain of } f \text{ and } t \in [0, 1]$$

Convexity

$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

$$\text{For all } a, b \text{ in domain of } f \text{ and } t \in [0,1]$$

- RTA: If I draw a line between two points on curve, all values on curve need to be on or below line.
- E.g. MSE loss is convex:

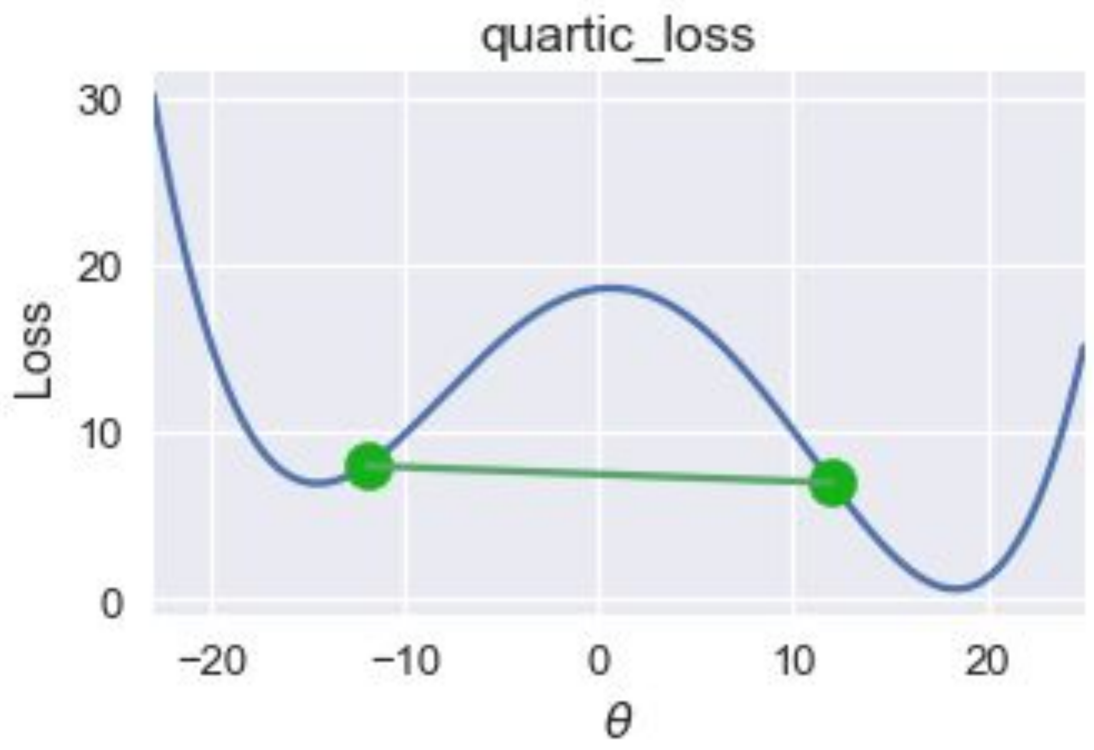


Convexity

$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

$$\text{For all } a, b \text{ in domain of } f \text{ and } t \in [0,1]$$

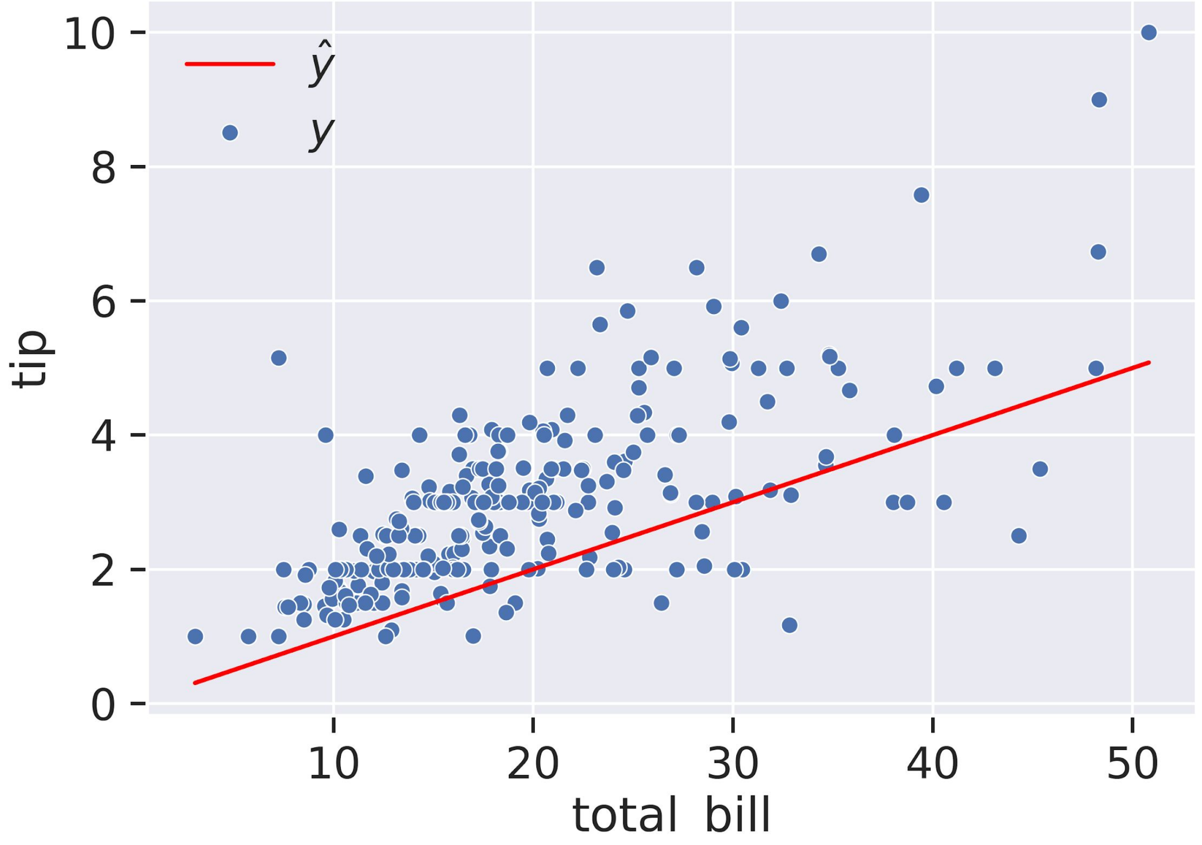
- But this loss function is not convex:



Optimizing Loss in 1D

Optimization Goal

Suppose we want to create a model that predicts the tip given the total bill for a table at a restaurant.



For this problem, we'll keep things simple and have only 1 parameter: gamma.

- $\hat{y} = f_{\hat{\gamma}(\vec{x})} = \hat{\gamma}\vec{x}$
- In other words, we are fitting a line with zero y-intercept.

See Notebook.

Optimization Goal

As discussed before, picking the best gamma is meaningless unless we pick:

- Loss function.
- Regularization term.

For this example, let's use the L2 loss and no regularization.

Solution Approach #1: Closed Form Solution

One approach is to use a closed form solution.

- On HW5 problem 3, you derived the closed form expression below:

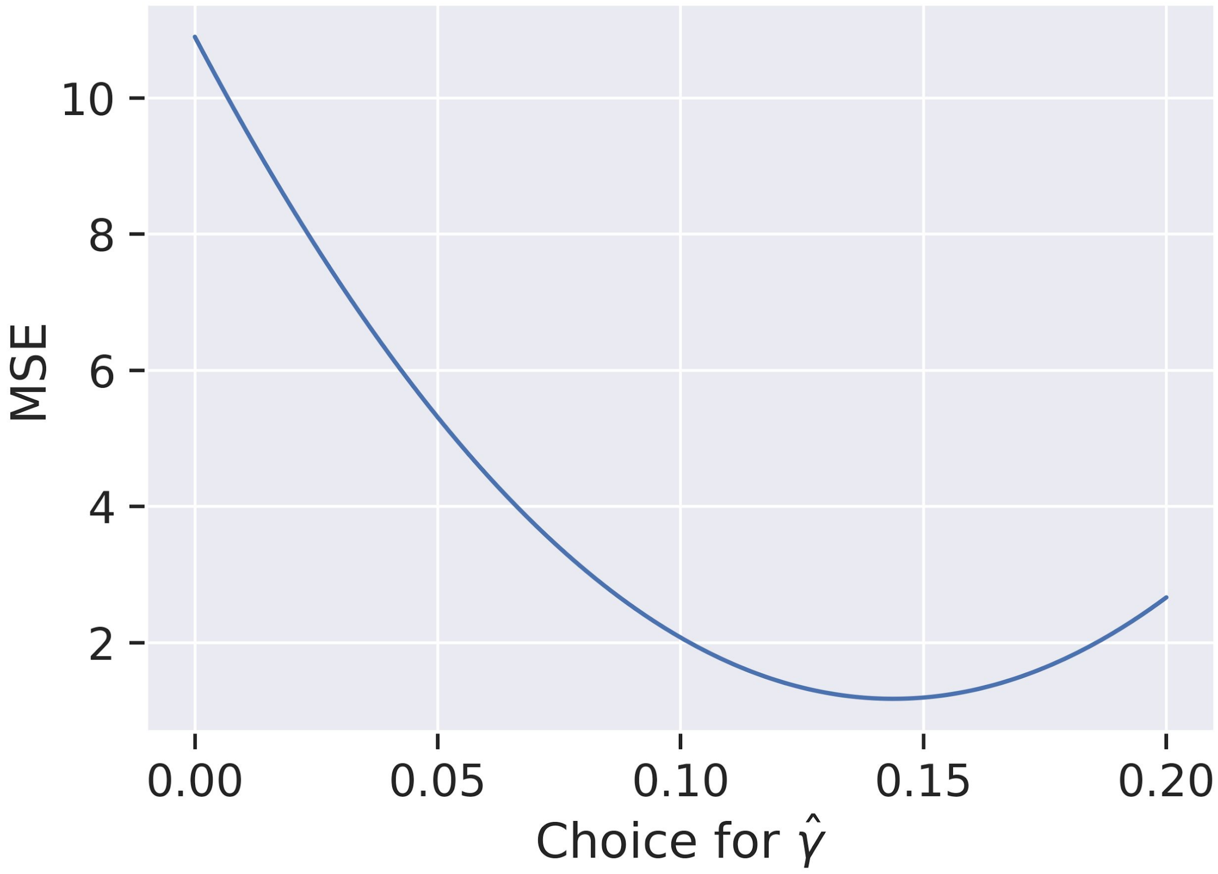
$$\hat{\gamma} = \frac{\sum x_i y_i}{\sum x_i^2}$$

Another closed form expression is just our standard normal equation:

$$\hat{\gamma} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T \vec{y}$$

Solution Approach #2A: Brute Force Plotting

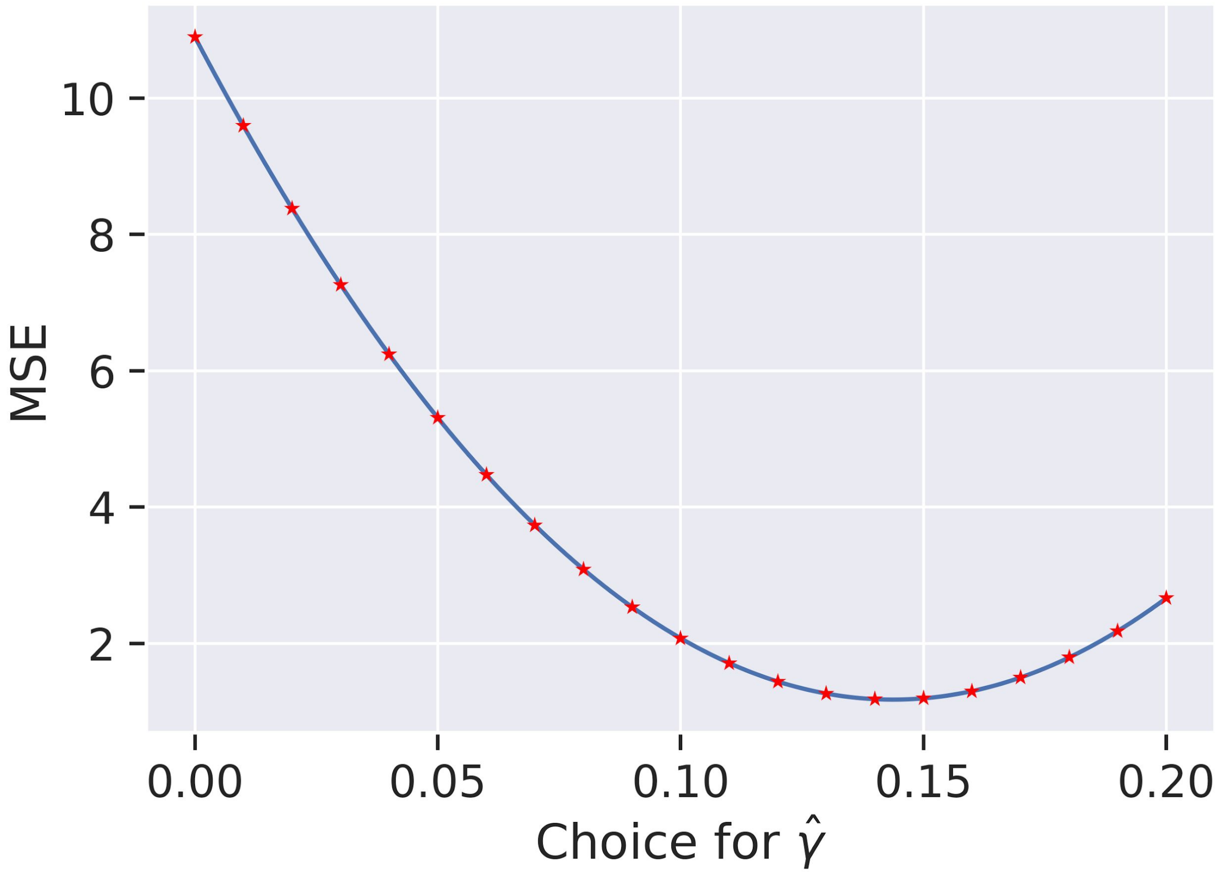
Another approach is to plot the loss and eyeball the minimum.




```python
def mse_single_arg(gamma):
    """Returns the MSE on our data for the given gamma"""
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = gamma * x
    return mse_loss(gamma, x, y_obs)
```

Solution Approach #2B: Brute Force

A related approach: Try a bunch of gammas and simply keep the best one.




```python
def mse_single_arg(gamma):
    """Returns the MSE on our data for the given gamma"""
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = gamma * x
    return mse_loss(gamma, x, y_obs)
```

```python
def simple_minimize(f, xs):
    y = [f(x) for x in xs]
    return xs[np.argmin(y)]
```

```python
simple_minimize(mse_single_arg, np.linspace(0, 0.2, 21))
```

Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use this, we need to find the derivative of the function that we're trying to minimize.
- Earlier, we minimized an arbitrary 4th degree polynomial.

$$\frac{d}{dx}$$

```
def f(x):
    return (x**4 - 15*x**3 + 80*x**2 - 180*x + 144)/10
```

```
def df(x):
    return (4*x**3 - 45*x**2 + 160*x - 180)/10
```

Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use this, we need to find the derivative of the function that we're trying to minimize.
- Earlier, we minimized an arbitrary 4th degree polynomial.

```
def df(x):
    return (4*x**3 - 45*x**2 + 160*x - 180)/10
```

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```

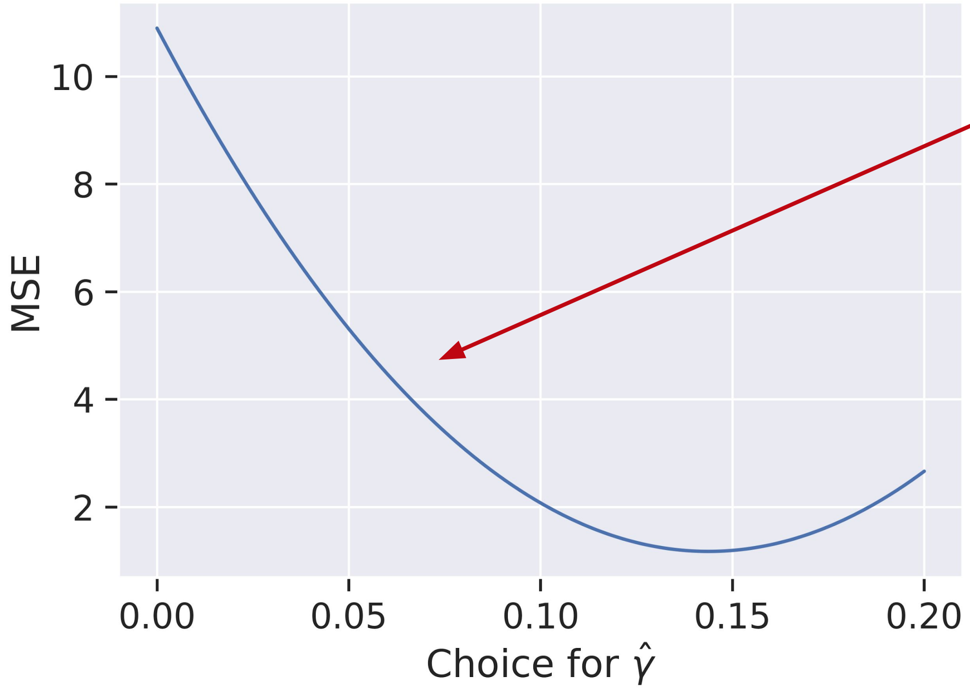
Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use GD on our linear regression problem, we need to find the derivative of the function that we're trying to minimize, namely `mse_loss`.

$$\frac{d}{d\hat{\gamma}}$$

```
def mse_loss(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean((y_hat - y_obs) ** 2)
```



Need to compute the derivative of this curve.

Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use GD on our linear regression problem, we need to find the derivative of the function that we're trying to minimize, namely `mse_loss`.

x comes from the fact that $\hat{y} = x\hat{\gamma}$

$$\frac{d}{d\hat{\gamma}}$$

```
def mse_loss(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean((y_hat - y_obs) ** 2)
```

```
def mse_loss_derivative(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean(2 * (y_hat - y_obs) * x)
```

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```

Solutions #4/#5: scipy.optimize.minimize / scipy.linear_model

As before, we can also use the scipy.optimize.minimize or scipy.linear_model libraries. Because it's exactly the same as before, we omit the exact details from this lecture.

Ultimately, both of these approaches use a numerical method similar to gradient descent.

Gradient Descent in 2D

Loss Minimization Game

From Fall 2018:

- https://tinyurl.com/3dloss18
- Try playing until you get the “You Win!” message.

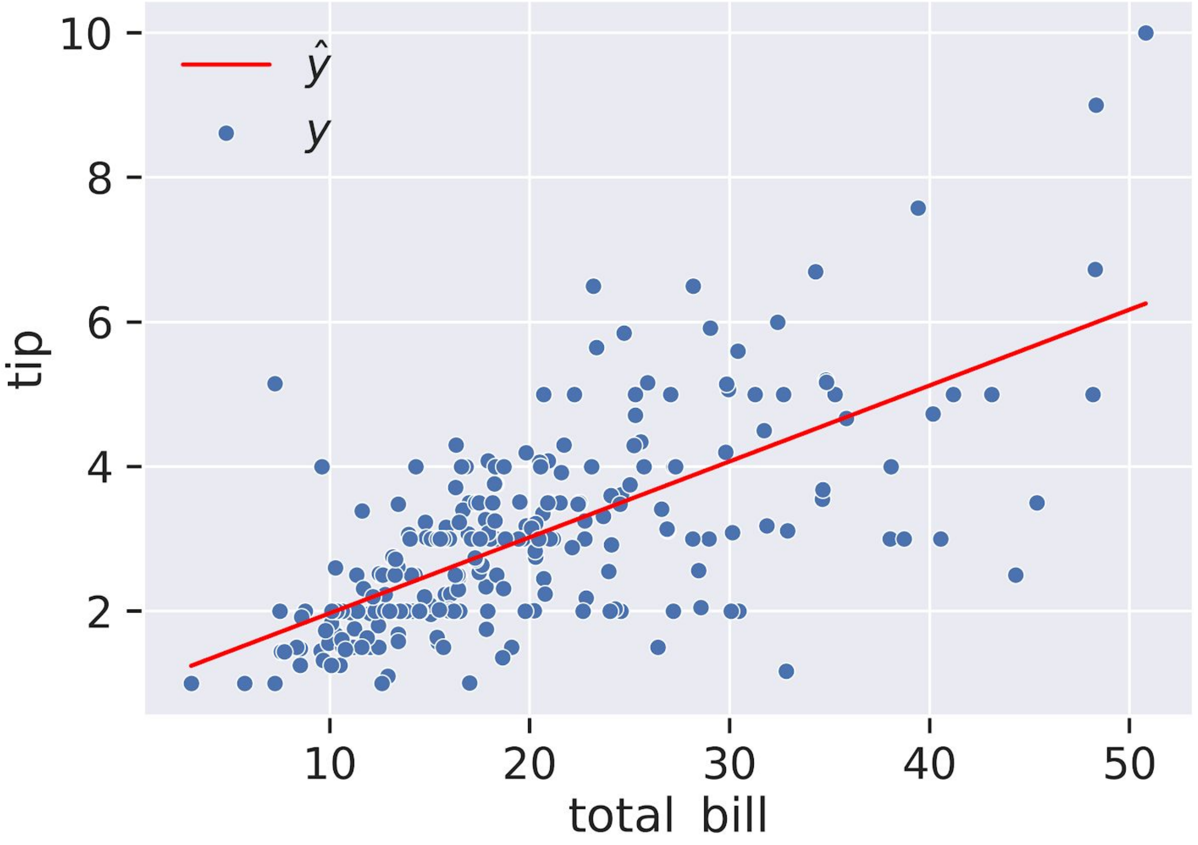
Optimization Goal

Now suppose we change our model so that it has two parameters θ_0 and θ_1.

- θ_0 is the y-intercept, and θ_1 is the slope.
- $$\text{tip} = \hat{\theta}_0 + \hat{\theta}_1 \text{bill}$$

$$\vec{\hat{y}} = f_{\vec{\hat{\theta}}}(\mathbb{X}) = \mathbb{X}\vec{\hat{\theta}}$$

$$\mathbb{X} = \begin{bmatrix} 1 & 16.99 \\ 1 & 10.34 \\ 1 & 21.01 \\ 1 & 23.68 \\ \vdots & \vdots \end{bmatrix}$$



Approach #1: Closed Form Solution

Since this is just a linear model, we can simply apply the normal equation.

$$\hat{\vec{y}} = f_{\hat{\vec{\theta}}}(\mathbb{X}) = \mathbb{X}\hat{\vec{\theta}} \qquad \hat{\vec{\theta}} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\vec{y}$$

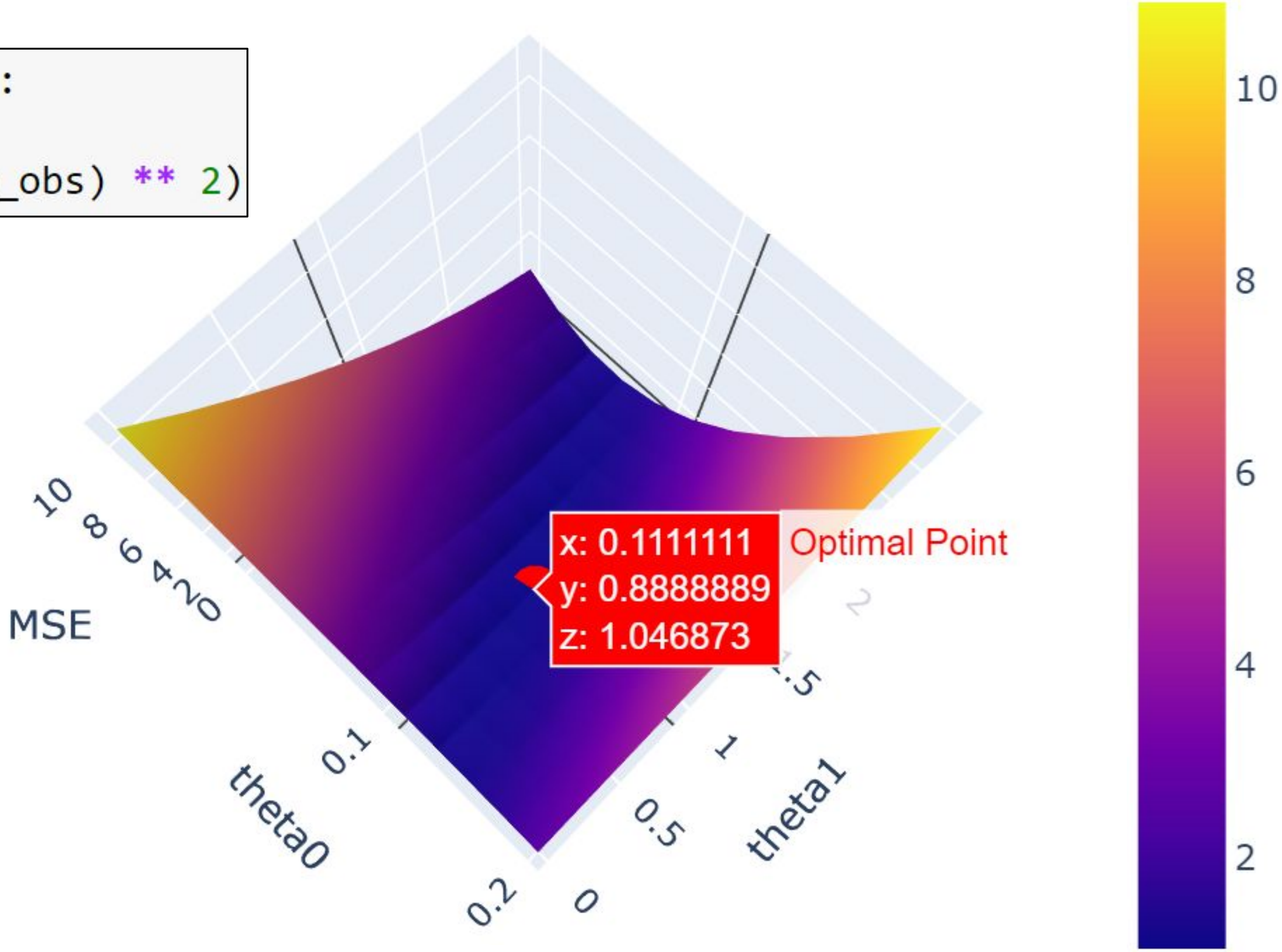
$$\mathbb{X} = \begin{bmatrix} 1 & 16.99 \\ 1 & 10.34 \\ 1 & 21.01 \\ 1 & 23.68 \\ \vdots & \vdots \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} 1.01 \\ 1.66 \\ 3.50 \\ 3.31 \\ \vdots \end{bmatrix}$$

For reasons we won't discuss, when calculating the closed form equation above, it's generally better to use np.linalg.solve instead of np.linalg.inv.

Approach #2: Brute Force / Plotting

As before, we could just plot the 2D loss surface and find the minimum that way (plot is easy to understand in the notebook).

```python
def mse_loss(theta, X, y_obs):
    y_hat = X @ theta
    return np.mean((y_hat - y_obs) ** 2)
```



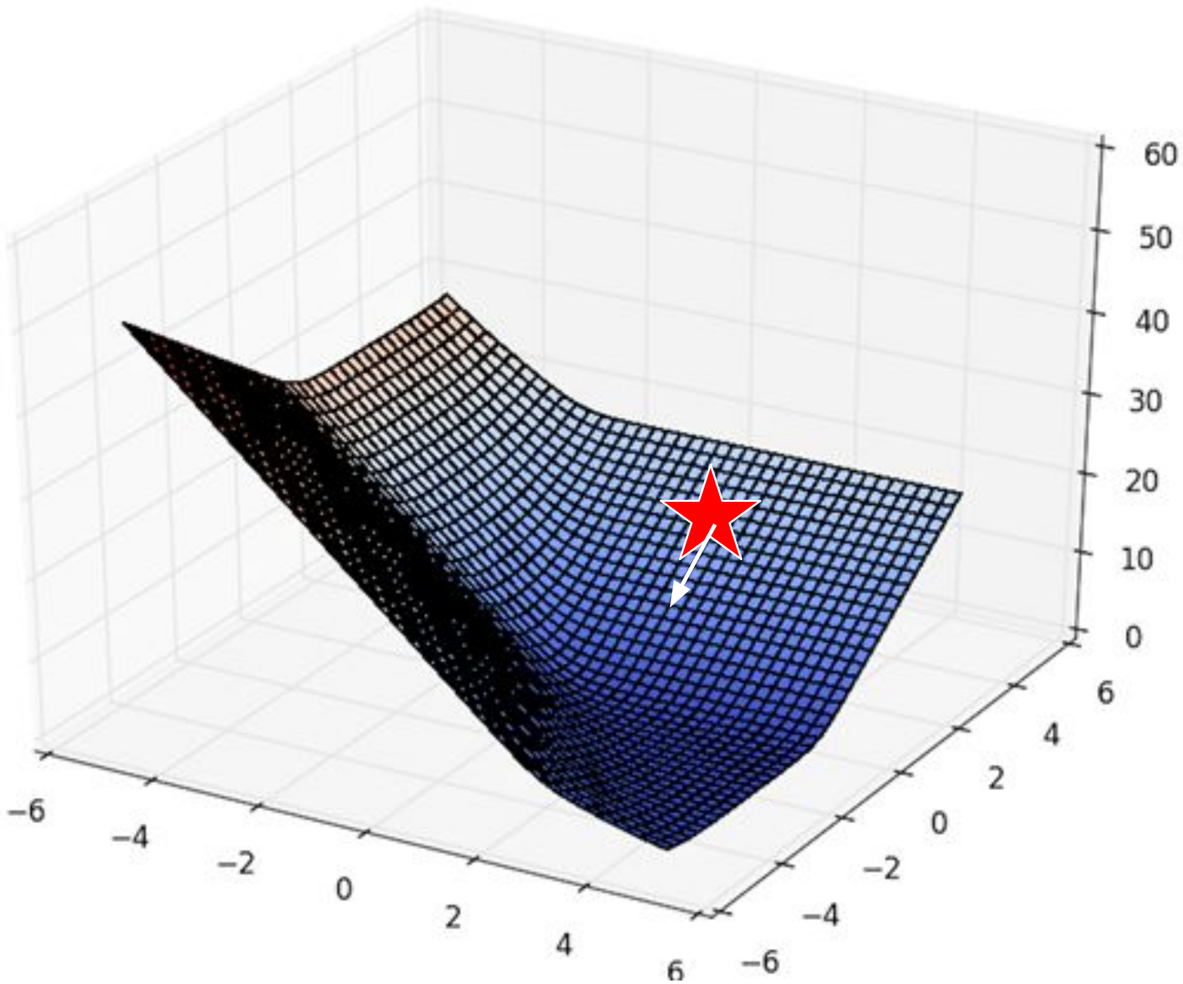
Approach #3: Gradient Descent

Another approach is to pick a starting point on our loss surface and follow the slope to the bottom.

- On a 2D surface, the best way to go down is described by a 2D vector.

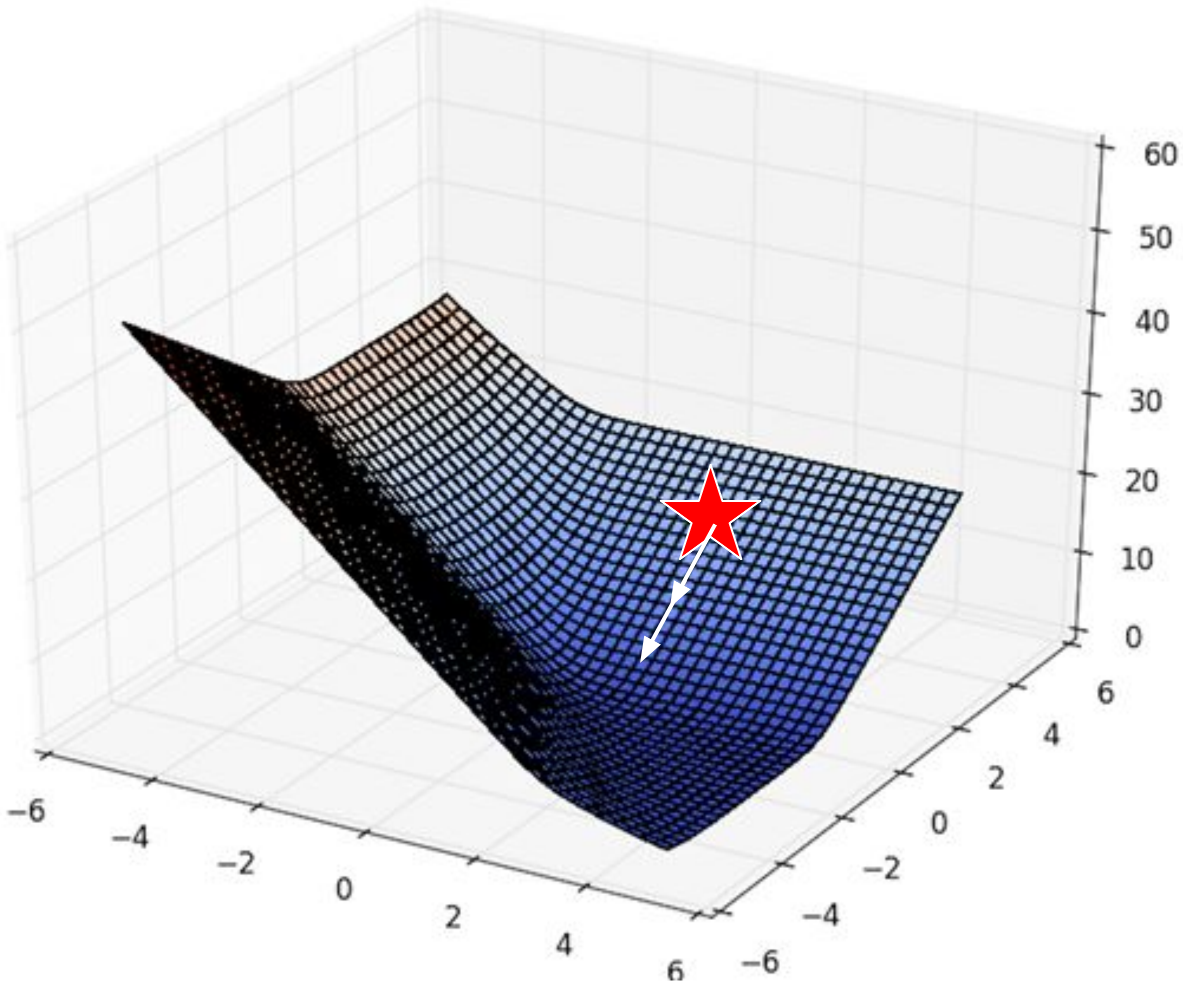
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



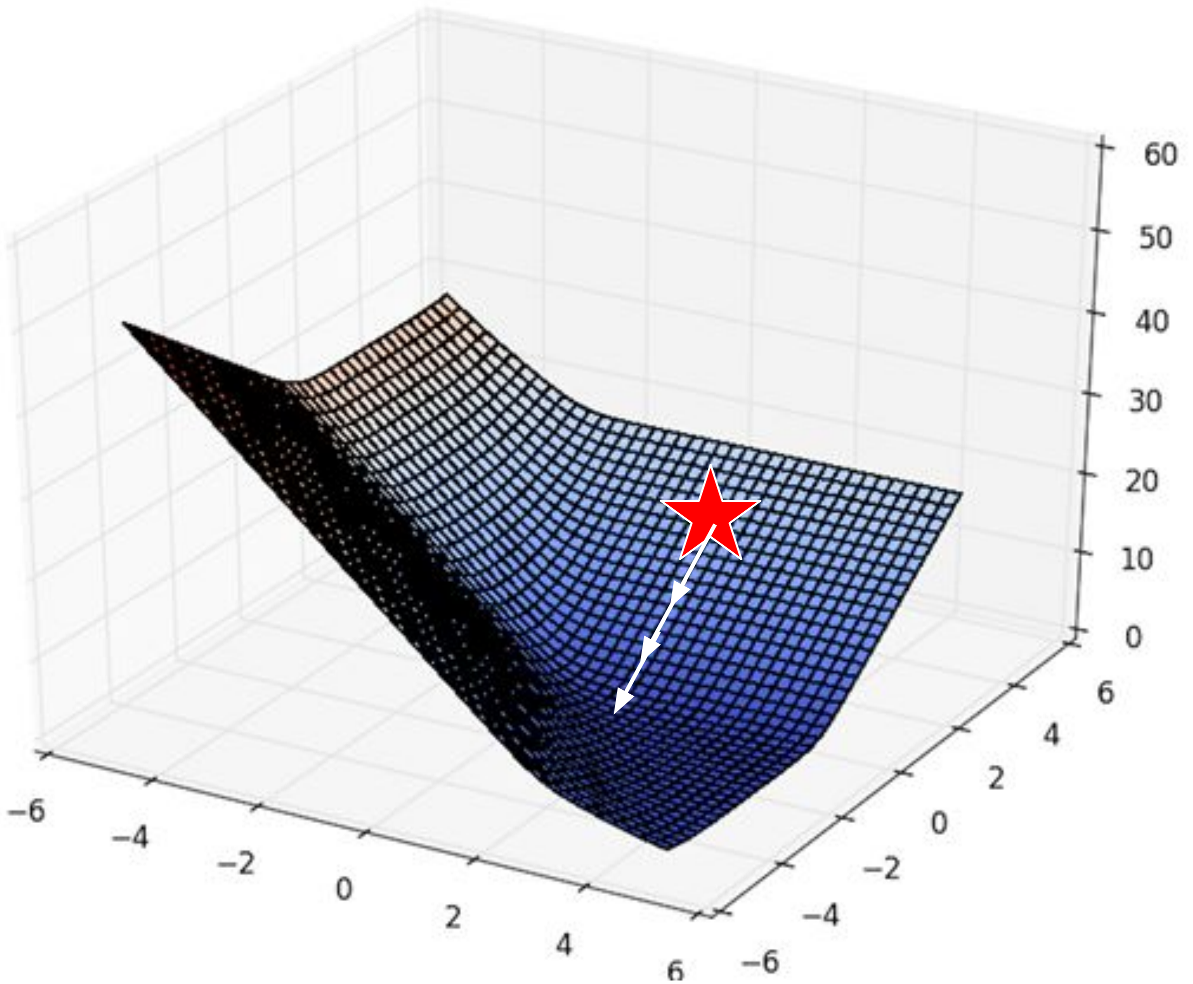
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



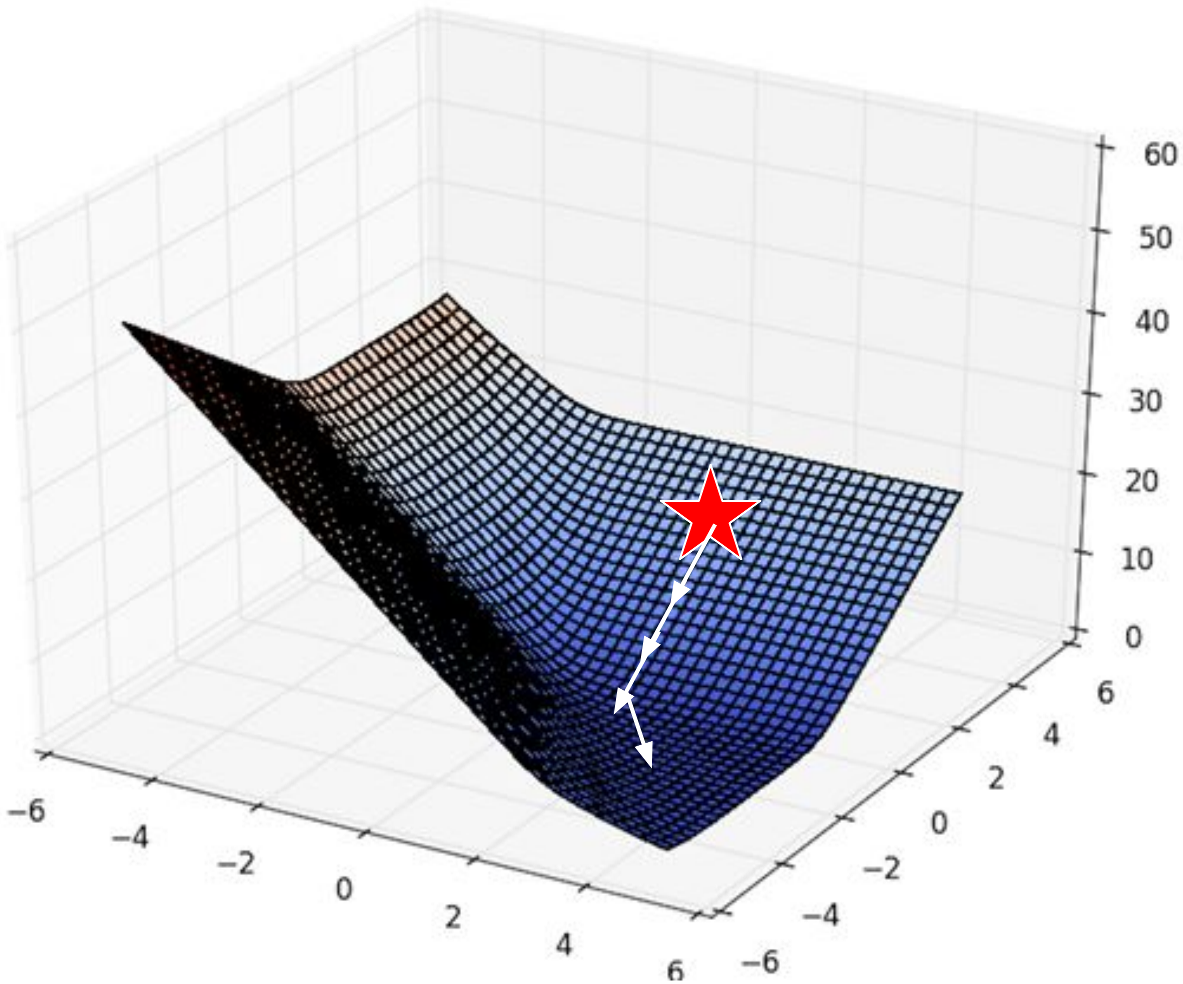
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



Example: Gradient of a 2D Function

Consider the 2D function: $f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$

For a function of 2 variables, $f(\theta_0, \theta_1)$ we define the gradient $\nabla_{\vec{\theta}} f = \frac{\partial f}{\partial \theta_0}\vec{i} + \frac{\partial f}{\partial \theta_1}\vec{j}$, where $\vec{i}$ and $\vec{j}$ are the unit vectors in the θ_0 and θ_1 directions.

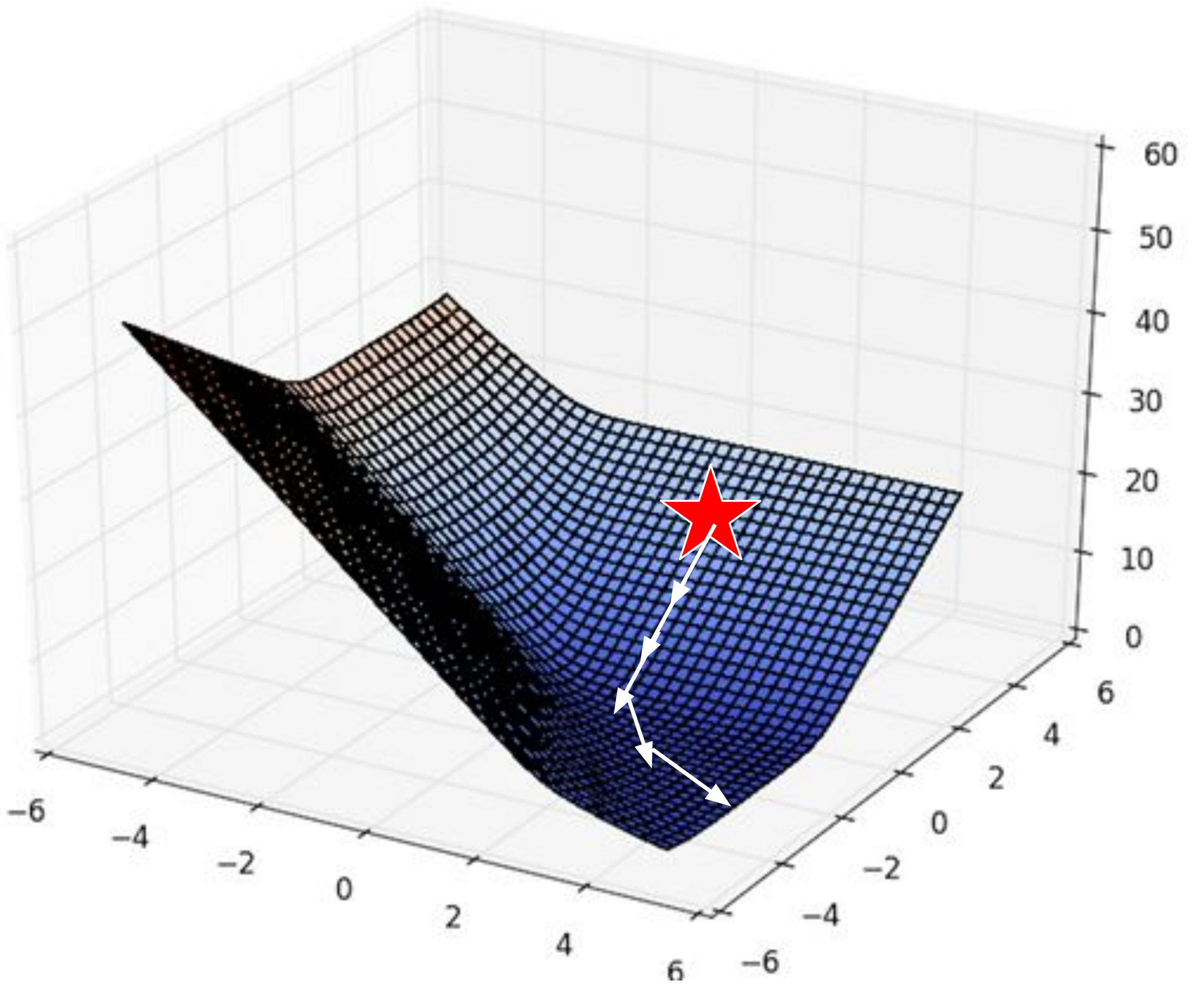
$$\frac{\partial f}{\partial \theta_0} = 16\theta_0 + 3\theta_1$$

$$\frac{\partial f}{\partial \theta_1} = 3\theta_0$$

$$\nabla_{\vec{\theta}} f = (16\theta_0 + 3\theta_1)\vec{i} + 3\theta_0\vec{j}$$

Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



Example: Gradient of a 2D Function in Column Vector Notation

Consider the 2D function: $f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$

Gradients are also often written in column vector notation.

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} 16\theta_0 + 3\theta_1 \\ 3\theta_0 \end{bmatrix}$$

Example: Gradient of a Function in Column Vector Notation

For a generic function of p + 1 variables.

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} \frac{\partial}{\partial \theta_0}(f) \\ \frac{\partial}{\partial \theta_1}(f) \\ \vdots \\ \frac{\partial}{\partial \theta_p}(f) \end{bmatrix}$$

How to Interpret Gradients

- You should read these gradients as:
  - If I nudge the 1st model weight, what happens to loss?
  - If I nudge the 2nd, what happens to loss?
  - Etc.

This is similar to what you were doing when playing the loss game.

Batch Gradient Descent

- **Gradient descent** algorithm: nudge θ in negative gradient direction until θ converges.
- Batch gradient descent update rule:

Next value for θ

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

Gradient of loss wrt θ

Learning rate

θ: Model weights L: loss function

α: Learning rate, usually a small constant

y: True values from the training data

Gradient Descent Algorithm

- Initialize model weights to all zero
  - Also common: initialize using small random numbers
- Update model weights using update rule:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

- Repeat until model weights don’t change (convergence).
  - At this point, we have θ̂, our minimizing model weights

The Gradient Descent Algorithm

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

- α is the learning rate
- Converges when gradient is ≈ 0 (or we run out of patience)

You Try:

Derive the gradient descent rule for a linear model with two model weights and MSE loss.

- Below we'll consider just one observation (i.e. one row of our design matrix).

$$f_{\vec{\theta}}(\vec{x}) = \vec{x}^T\vec{\theta} = \theta_0 x_0 + \theta_1 x_1$$

Squared loss for a single prediction of our linear regression model.

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = ?$$

You Try:

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

$$\frac{\partial}{\partial \theta_0} \ell(\vec{\theta}, \vec{x}, y_i) = 2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_0)$$

$$\frac{\partial}{\partial \theta_1} \ell(\vec{\theta}, \vec{x}, y_i) = 2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_1)$$

The gradient for the entire dataset is the average of the gradients for each point, so we can run GD as-is.

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = \begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$$

You Try:

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = \begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$$

The gradient for the entire dataset is the average of the gradients for each point, so we use np.mean to compute that average.

```
def mse_gradient(theta, X, y_obs):
    """Returns the gradient of the MSE on our data for the given theta"""
    x0 = X.iloc[:, 0]
    x1 = X.iloc[:, 1]
    dth0 = np.mean(-2 * (y_obs - theta[0] * x0 - theta[1] * x1) * x0)
    dth1 = np.mean(-2 * (y_obs - theta[0] * x0 - theta[1] * x1) * x1)
    return np.array([dth0, dth1])
```

(demo)

Stochastic Gradient Descent

The Gradient Descent Algorithm

$\theta^{(0)} \leftarrow$ initial vector (random, zeros ...)

For τ from 0 to convergence:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

- α is the learning rate
- Converges when gradient is ≈ 0 (or we run out of patience)

Which Step in This Algorithm is Most Time Consuming?

Gradient Descent Algorithm

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \boxed{\nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})}$$

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \boxed{\nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})}$$

Typically the loss function is really the **average loss** over a **large dataset**.

$$\nabla_\theta \mathbf{L}(\theta)\Big|_{\theta=\theta^{(\tau)}} = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \text{loss}(y, f_\theta(x))\Big|_{\theta=\theta^{(\tau)}}$$

- **Loading** and **computing** on all the data is **expensive** ☹.
- What do we do when accessing the "**population**" is prohibitively expensive?

Stochastic Gradient Descent

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

$$\mathcal{B} \sim \text{ Random subset of indices}$$

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \alpha \left(\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \mathbf{L}_i(\theta) \Big|_{\theta=\theta^{(\tau)}} \right)$$

1. Draw a simple random sample of data indices
   - Often called a **batch** or **mini-batch**
   - Choice of **batch size** trade-off **gradient quality** and **speed**

2. Compute **gradient estimate** and uses as **gradient**

Stochastic Gradient Descent

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

$$\mathcal{B} \sim \text{ Random subset of indices}$$

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \alpha \left(\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \mathbf{L}_i(\theta) \Big|_{\theta=\theta^{(\tau)}} \right)$$

Decomposable Loss

$$\mathbf{L}(\theta) = \sum_{i=1}^{n} \mathbf{L}_i(\theta) = \sum_{i=1}^{n} \mathbf{L}(\theta, x_i, y_i)$$

Loss can be written as a sum of the loss on each record.

Gradient Descent

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \alpha \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathbf{L}_i(\theta) \Big|_{\theta=\theta^{(\tau)}} \right)$$

Stochastic Gradient Descent

$\theta^{(0)} \leftarrow$ initial vector (random, zeros …)

For τ from 0 to convergence:

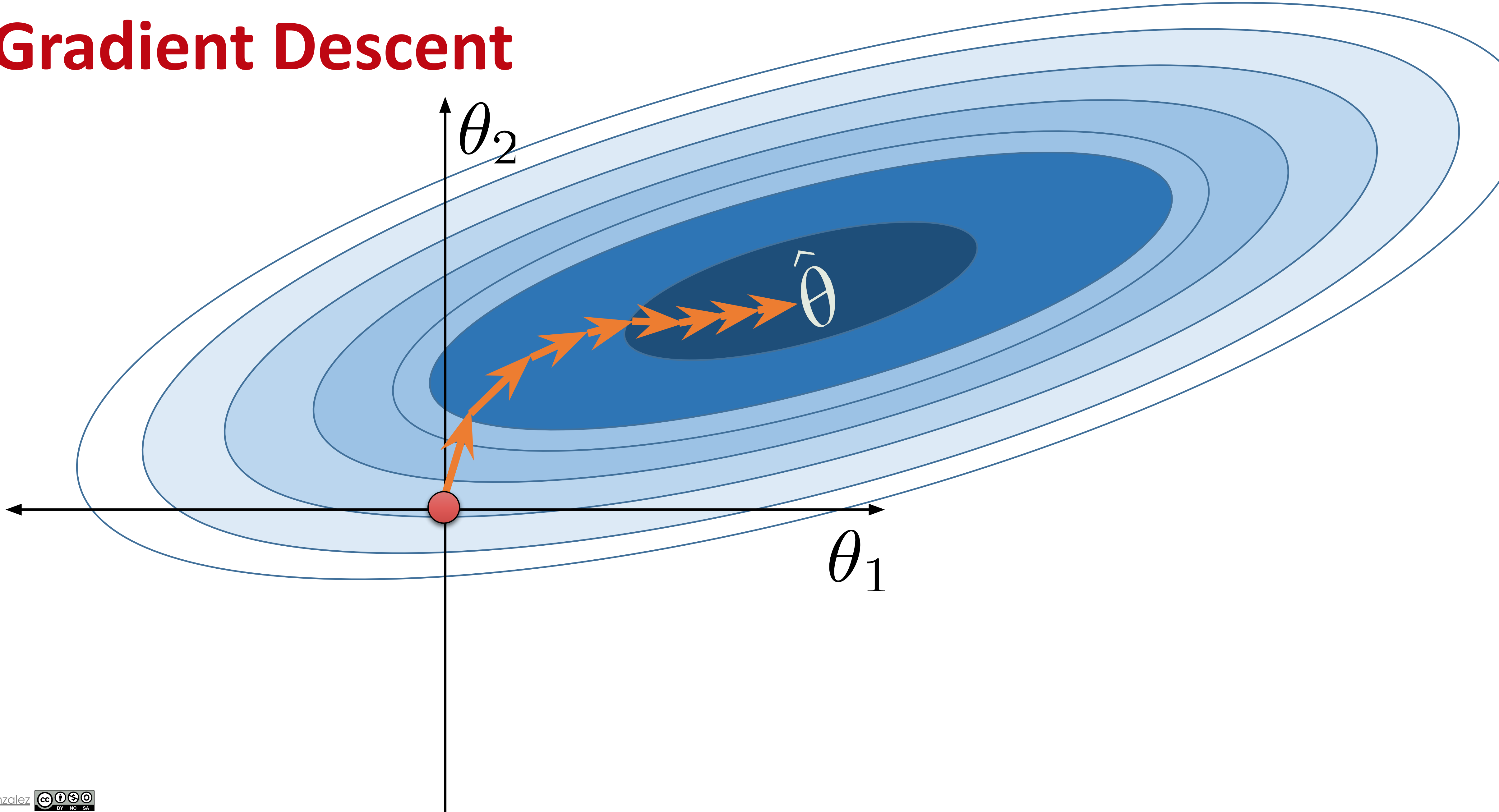
$$\mathcal{B} \sim \text{Random subset of indices}$$

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \alpha \left(\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \mathbf{L}_i(\theta) \Big|_{\theta=\theta^{(\tau)}} \right)$$

Very Similar Algorithms

Assuming Decomposable Loss Functions

Gradient Descent



<u>Stochastic</u> Gradient Descent