LECTURE 16

Gradient Descent

Optimization methods to analytically and numerically minimize loss functions.

Today's Goal: Ordinary Least Squares Numerically

1. Choose a model	Multiple Linear Regression
2. Choose a loss function	L2 Loss Mean Squared Error (MSE)
3. Fit the model	Minimize average loss with ~~calculus~~ ~~geometry~~ **numerical methods**
4. Evaluate model performance	MSE

For each of our n datapoints:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p$$

$$\hat{\mathbb{Y}} = \mathbb{X}\boldsymbol{\theta}$$

Today's Roadmap

- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

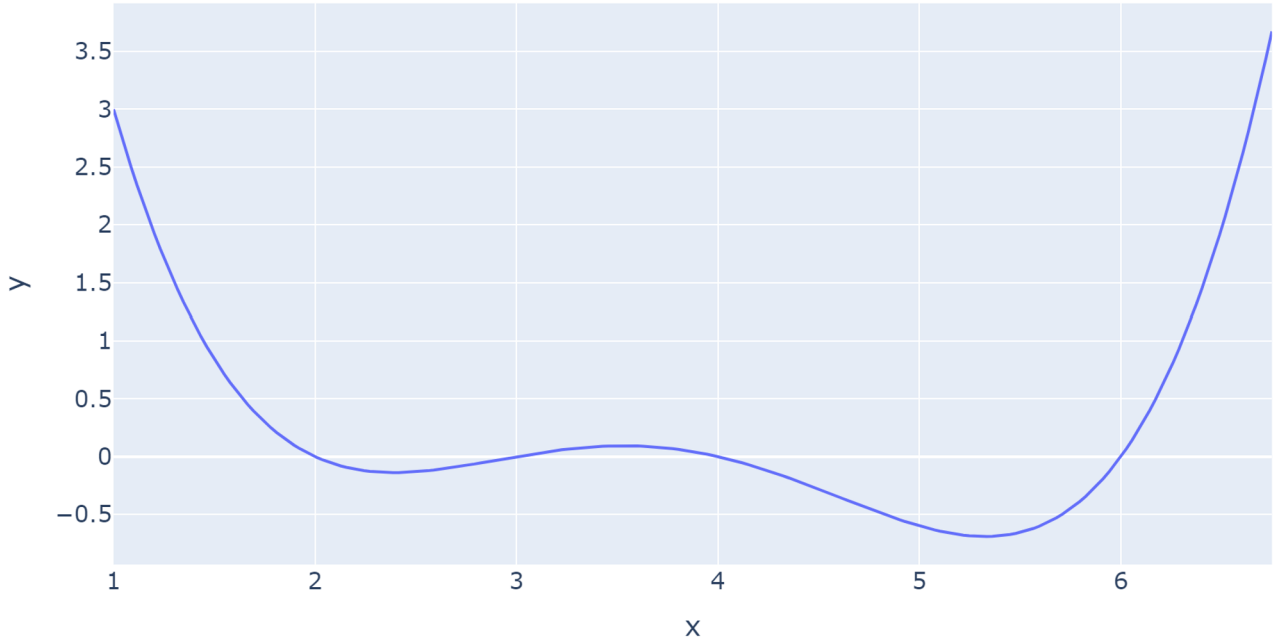
Minimizing an Arbitrary 1D Function

- **Minimizing an Arbitrary 1D Function**
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

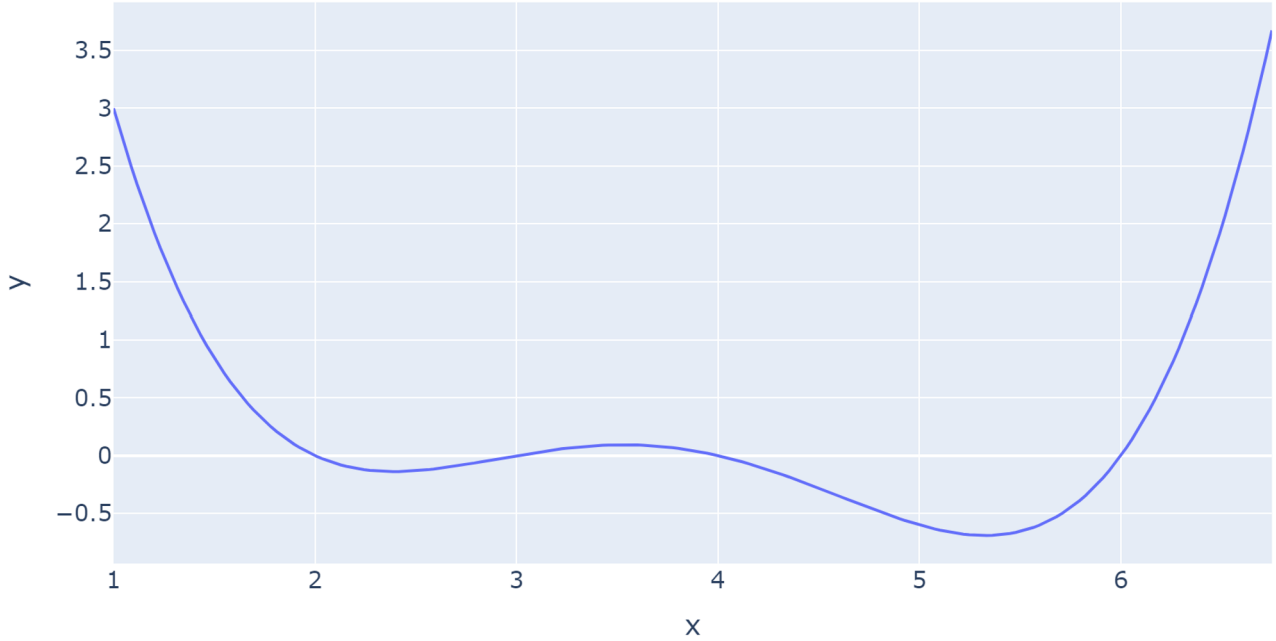
Arbitrary Function of Interest

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

x = np.linspace(1, 6.75, 200)
fig = px.line(y = arbitrary(x), x = x)
```



Minimizing this Function using scipy.optimize.minimize




```
from scipy.optimize import minimize

minimize(arbitrary, x0 = 6)
```

```
minimize(arbitrary, x0 = 1)
```

The choice of start point can affect the outcome

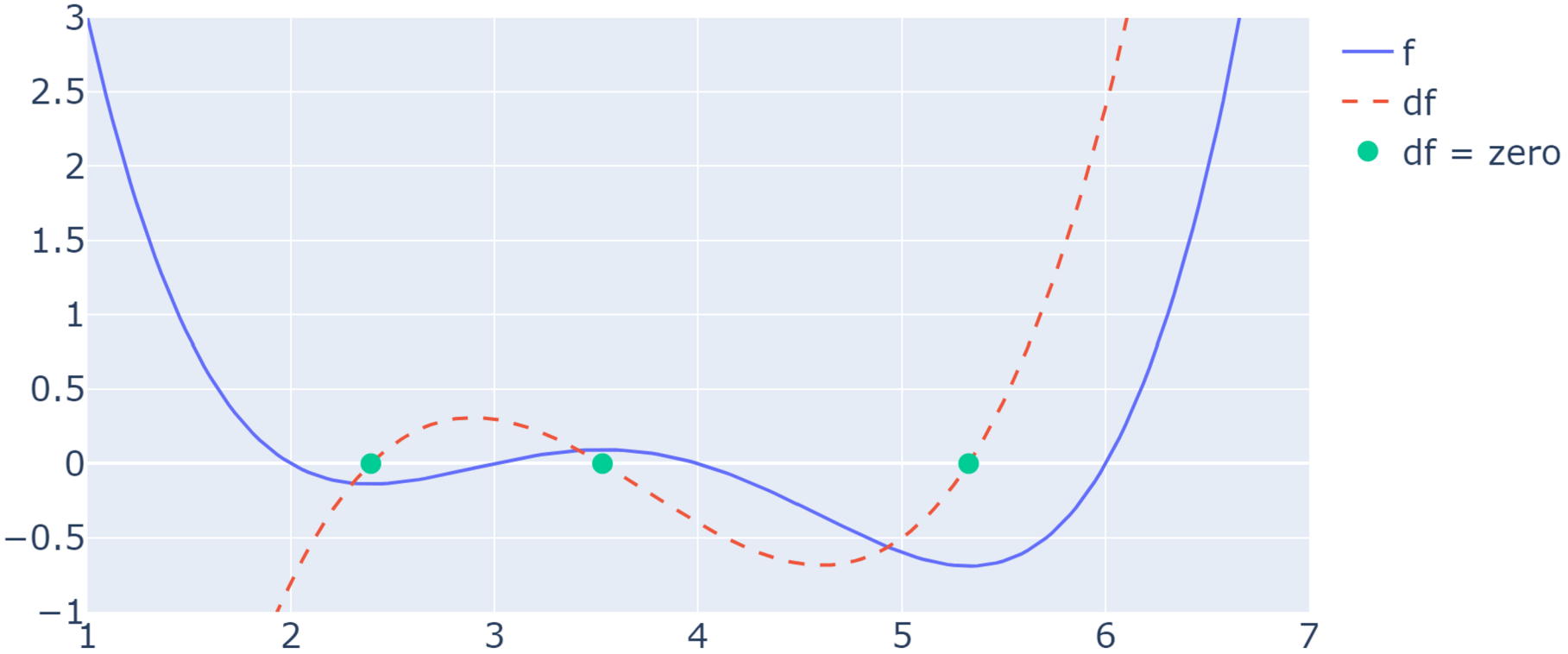
```
      fun: -0.6914096788729693
 hess_inv: array([[0.47465475]])
      jac: array([-3.05473804e-06])
  message: 'Optimization terminated successfully.'
     nfev: 16
      nit: 7
     njev: 8
   status: 0
  success: True
        x: array([5.3263436])
```

```
      fun: -0.13827491294422317
 hess_inv: array([[0.74751575]])
      jac: array([-3.7997961e-07])
  message: 'Optimization terminated successfully.'
     nfev: 16
      nit: 7
     njev: 8
   status: 0
  success: True
        x: array([2.3927478])
```

Minimizing an Arbitrary 1D Function - Gradient Descent Example

- **Minimizing an Arbitrary 1D Function**
  - **Gradient Descent Example**
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

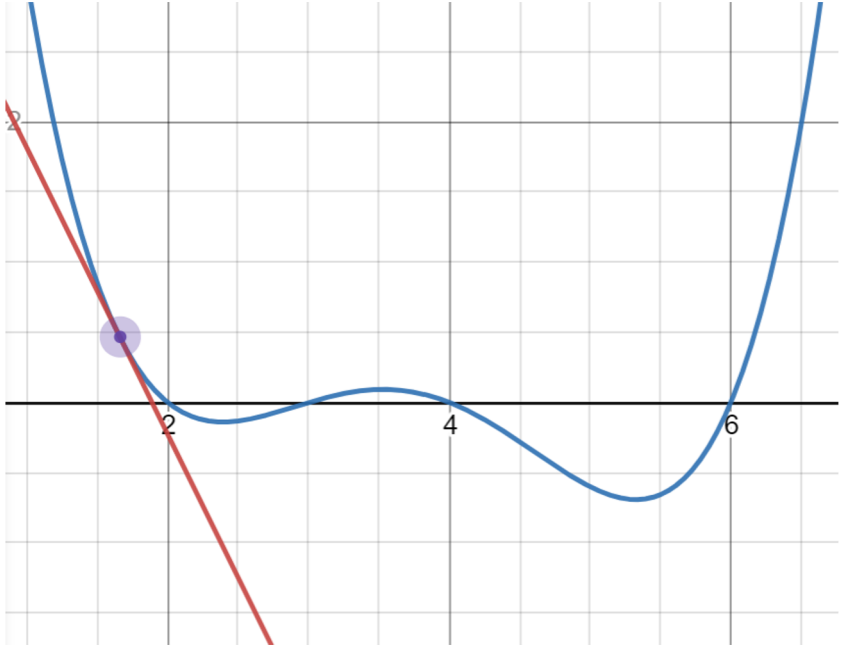
Function, Roots, and Derivatives



Derivative Tells Us Which Way to Go

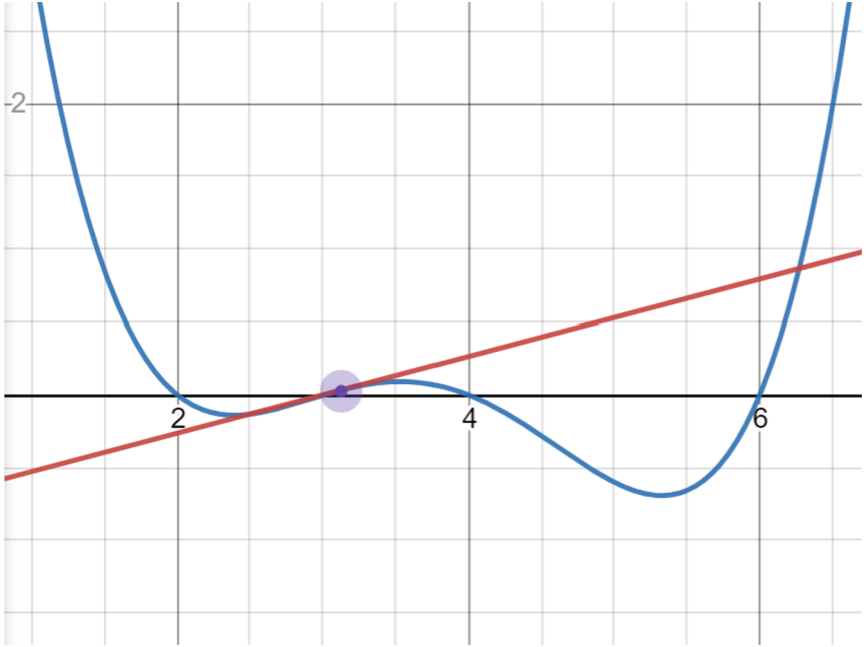
Derivative is negative, so go right.

- Follow the slope down.



Derivative is positive, so go left.

- Follow the slope down.

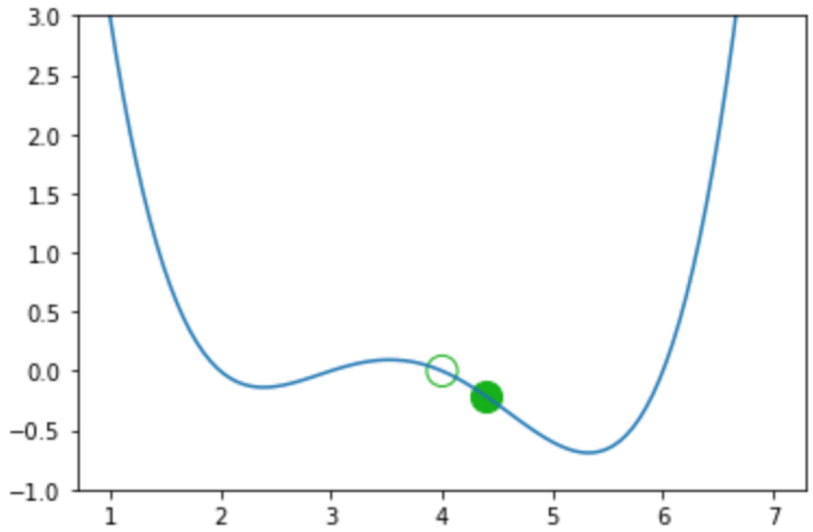


Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(4)
```

```
old x: 4
new x: 4.4
```




```
guess = 4
print(f"x: {guess}, f(x): {arbitrary(guess)}, derivative f'(x): {derivative_arbitrary(guess)}")
```

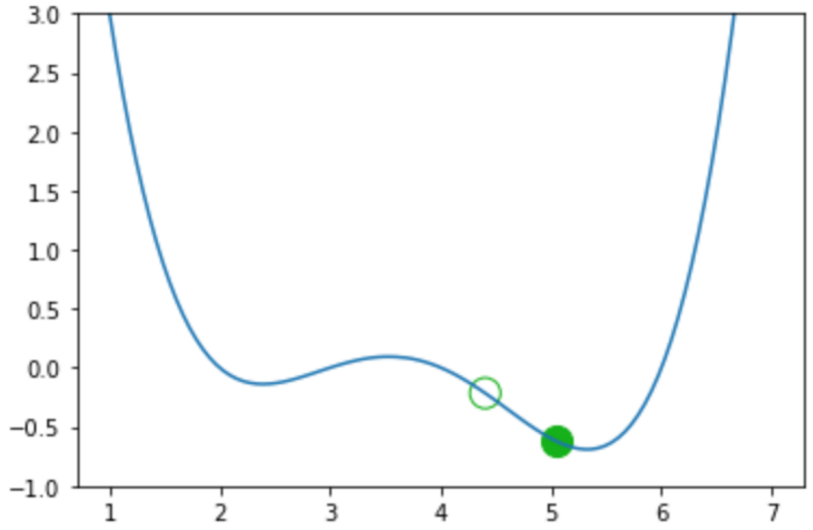
```
x: 4, f(x): 0.0, derivative f'(x): -0.4
```

Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(4.4)
```

```
old x: 4.4
new x: 5.0464000000000055
```




```
guess = 4 + 0.4
print(f"x: {guess}, f(x): {arbitrary(guess)}, derivative f'(x): {derivative_arbitrary(guess)}")
```

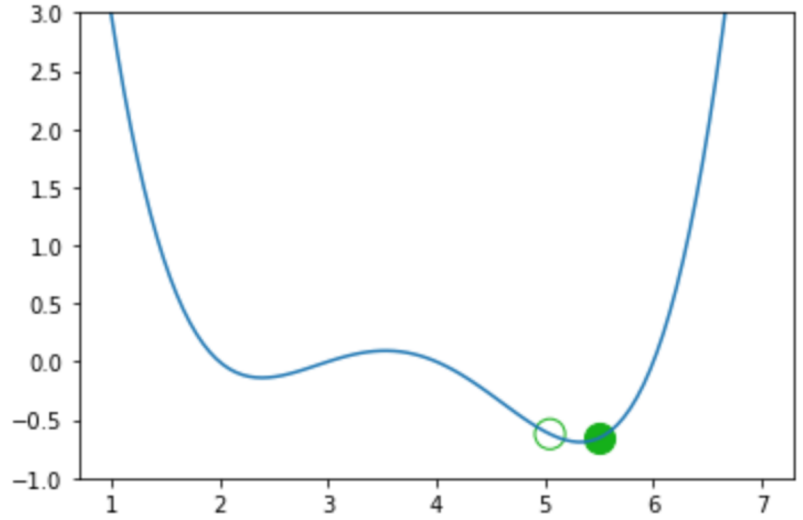
```
x: 4.4, f(x): -0.21504000000003315, derivative f'(x): -0.6464000000000055
```

Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(5.0464)
```

```
old x: 5.0464
new x: 5.49673060106241
```

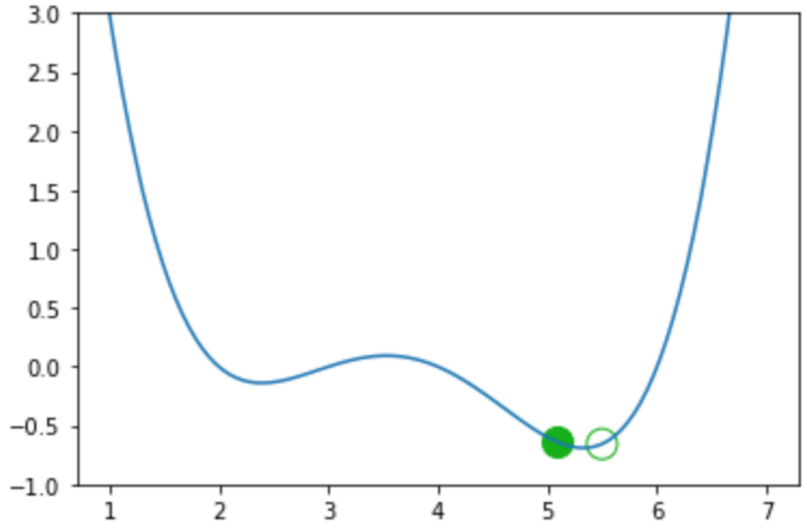


Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(5.4967)
```

```
old x: 5.4967
new x: 5.080917145374805
```

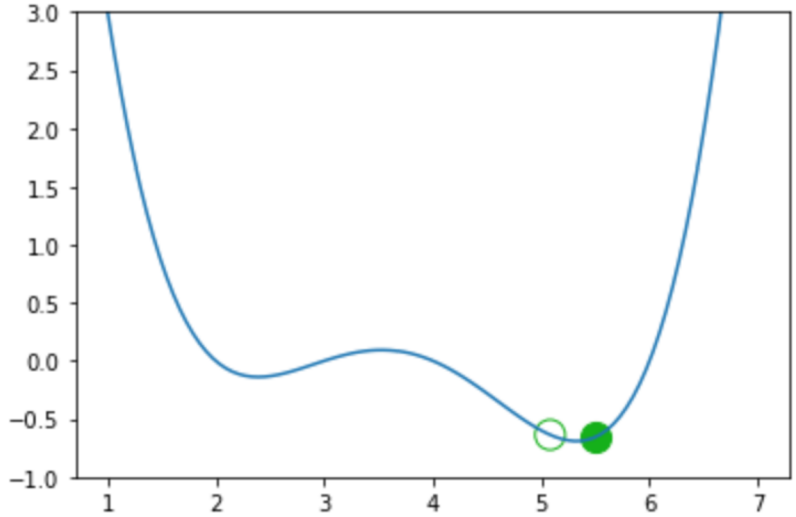


Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(5.080917145374805)
```

```
old x: 5.080917145374805
new x: 5.489966698640582
```

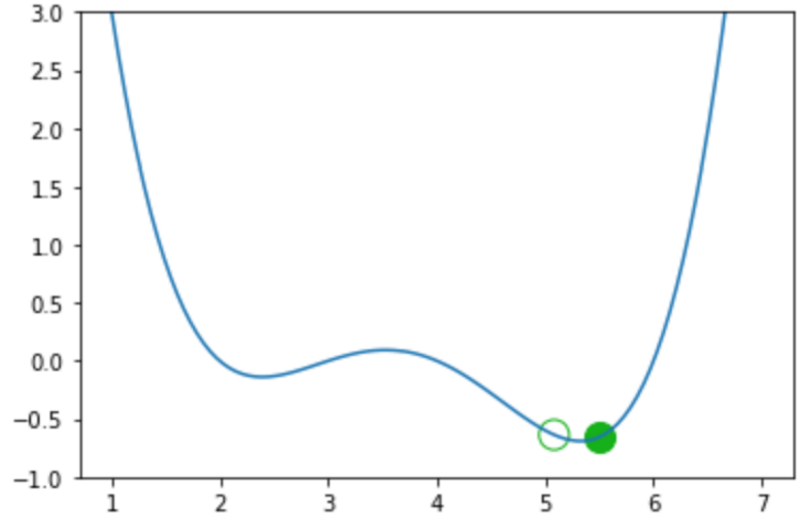


Manual Gradient Descent

```
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step(5.080917145374805)
```

```
old x: 5.080917145374805
new x: 5.489966698640582
```



We appear to be bouncing back and forth. Turns out we are stuck!

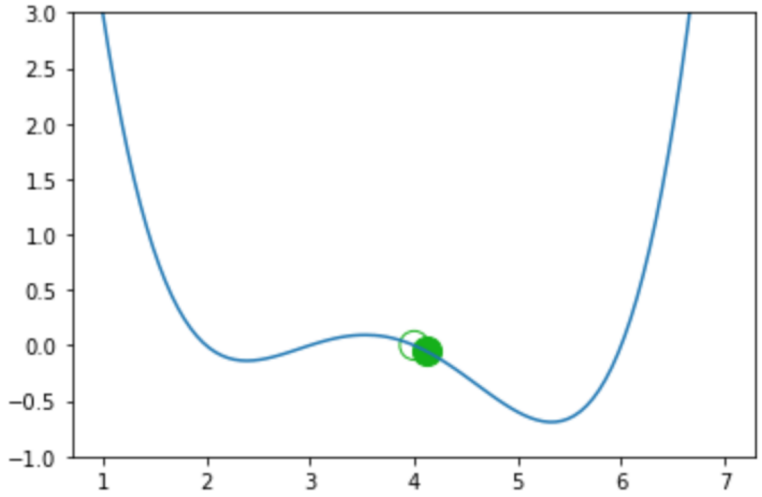
- Any suggestions for how we can avoid this issue?

Manual Gradient Descent with Slower “Learning Rate”

```
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step_better(4)
```

```
old x: 4
new x: 4.12
```

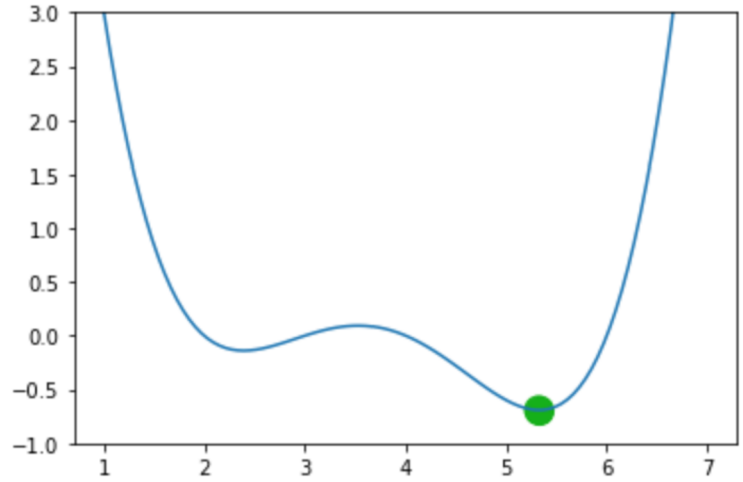


Manual Gradient Descent with Slower “Learning Rate” (many steps later)

```
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step_better(5.323)
```

```
old x: 5.323
new x: 5.325108157959999
```



Gradient Descent Implementation

- **Minimizing an Arbitrary 1D Function**
  - Gradient Descent Example
  - **Gradient Descent Implementation**
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

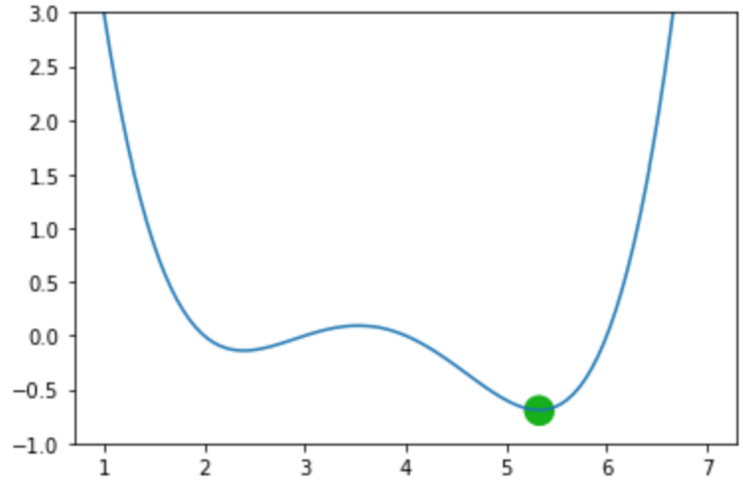
Gradient Descent as a Recurrence Relation

```
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

$$x^{(t+1)} = x^{(t)} - 0.3\frac{d}{dx}f(x)$$

```
plot_one_step_better(5.323)
```

```
old x: 5.323
new x: 5.325108157959999
```



Our Recurrence Relation as Iterative Code

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

```
def gradient_descent(df, initial_guess, alpha, n):
    """Performs n steps of gradient descent on df using learning rate alpha starting
       from initial_guess. Returns a numpy array of all guesses over time."""
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)
```

```
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
trajectory
```

```
array([4.        , 4.12      , 4.26729664, 4.44272584, 4.64092624,
       4.8461837 , 5.03211854, 5.17201478, 5.25648449, 5.29791149,
       5.31542718, 5.3222606 , 5.32483298, 5.32578765, 5.32614004,
       5.32626985, 5.32631764, 5.32633523, 5.3263417 , 5.32634408])
```

Our Recurrence Relation as Iterative Code

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

```
def gradient_descent(df, initial_guess, alpha, n):
    """Performs n steps of gradient descent on df using learning rate alpha starting
       from initial_guess. Returns a numpy array of all guesses over time."""
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)
```

```
trajectory = gradient_descent(derivative_arbitrary, 4, 1, 20)
trajectory
```

```
array([4.        , 4.4       , 5.0464    , 5.4967306 , 5.08086249,
       5.48998039, 5.09282487, 5.48675539, 5.09847285, 5.48507269,
       5.10140255, 5.48415922, 5.10298805, 5.48365325, 5.10386474,
       5.48336998, 5.1043551 , 5.48321045, 5.10463112, 5.48312031])
```

Gradient Descent on a 1D Model

- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- **Gradient Descent on a 1D Model**
- Gradient Descent in Higher Dimensions
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

Applying Gradient Descent to Our Tips Dataset

We've seen how to find the optimal parameters for a 1D linear model for the tips dataset:

- Using the derived equations.
- Using sklearn.
    - Uses gradient descent!

While in real practice in this course, you'll usually use sklearn, let's see how we can do the gradient descent ourselves.

We'll first fit a model that has no y-intercept, for maximum simplicity.

Let's try this out in our notebook.

Gradient Descent in Higher Dimensions

- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- **Gradient Descent in Higher Dimensions**
- Gradient Descent Extended
  - Stochastic Gradient Descent
  - Convexity

A Two Parameter Model

Suppose we now try simple linear regression, which has two parameters:

$$\text{tip} = \theta_0 + \theta_1 \text{bill}$$

We'll use gradient descent to minimize the function below:

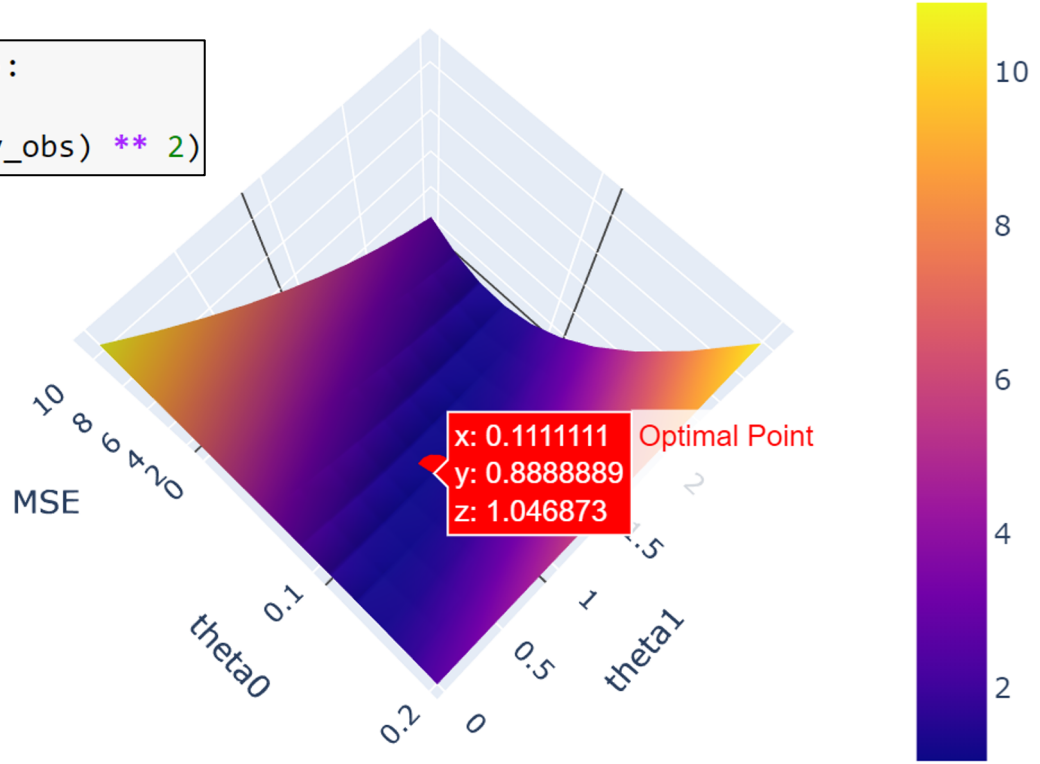
- Here, theta is a two dimensional vector!

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

A 2D Loss Function

Here, we see the loss of our model as a function of our two parameters.

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



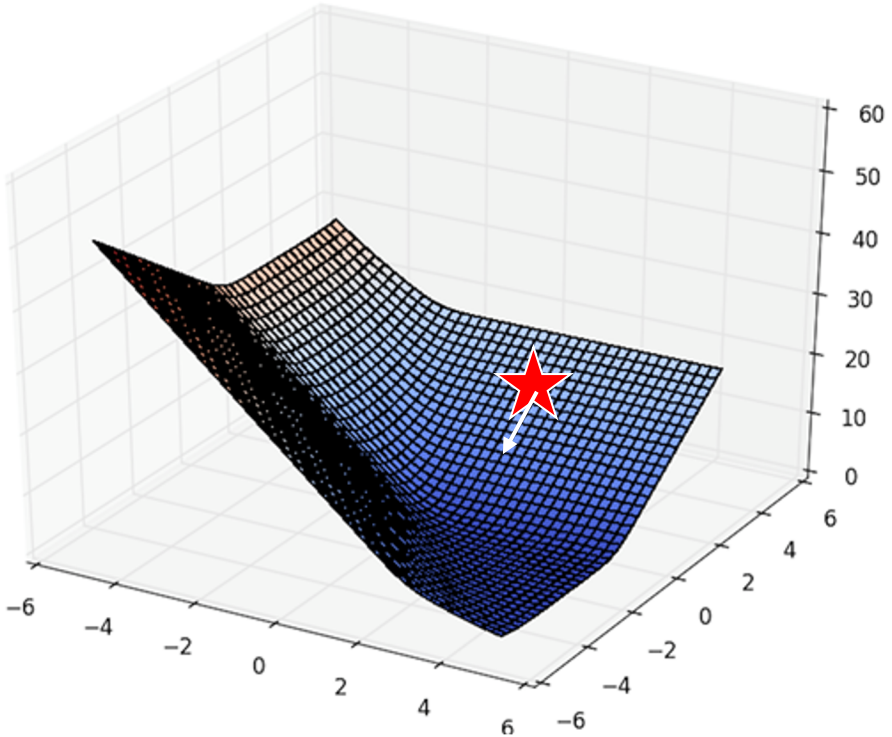
Gradient Descent

Just like earlier, we can follow the slope of our 2D function.

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

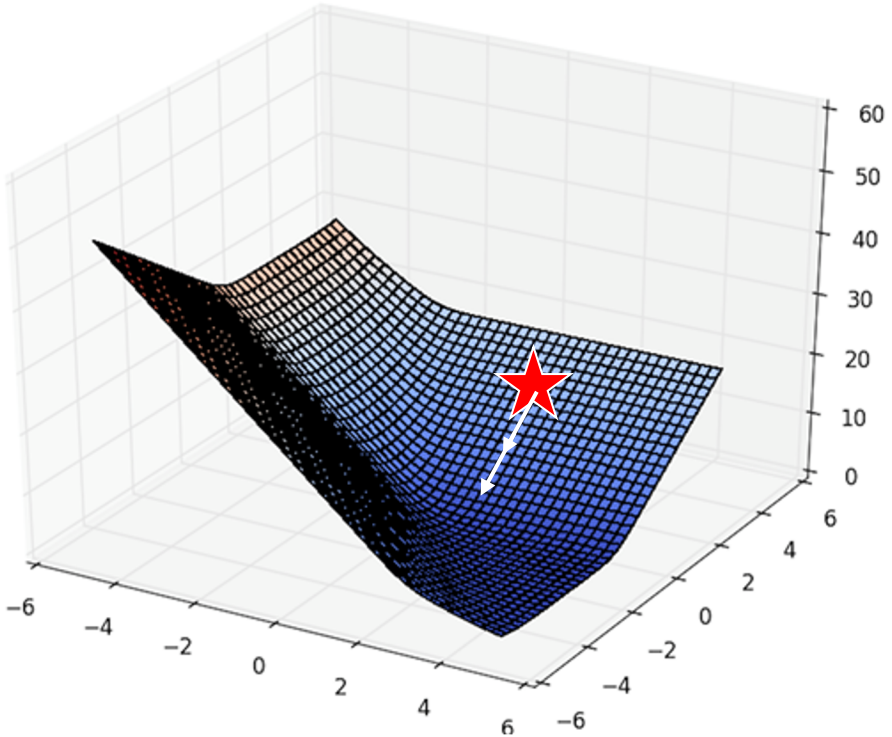
Approach #3: Gradient Descent

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



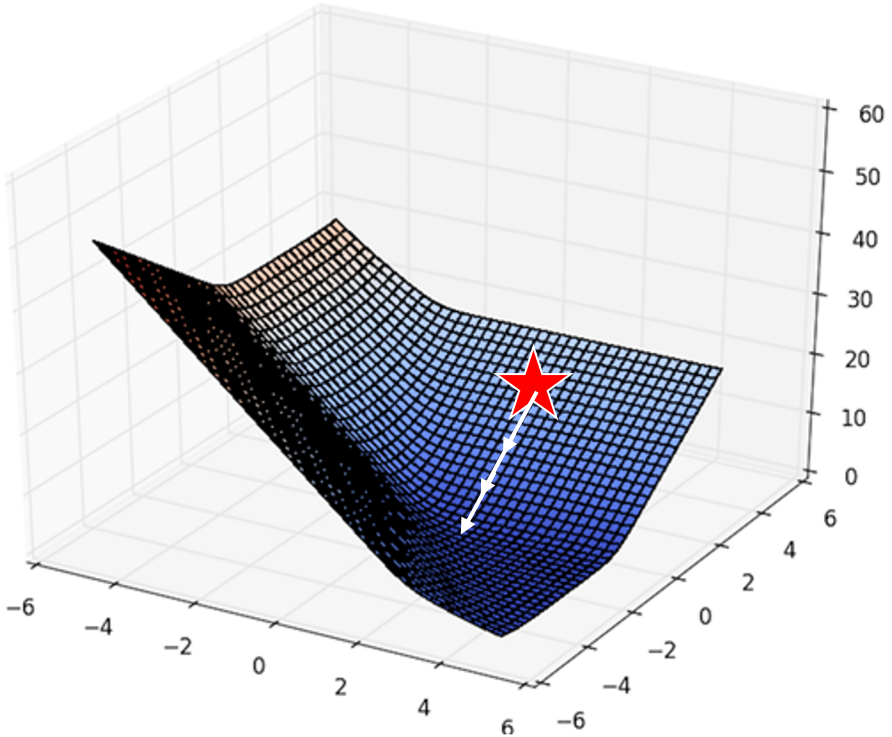
Approach #3: Gradient Descent

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



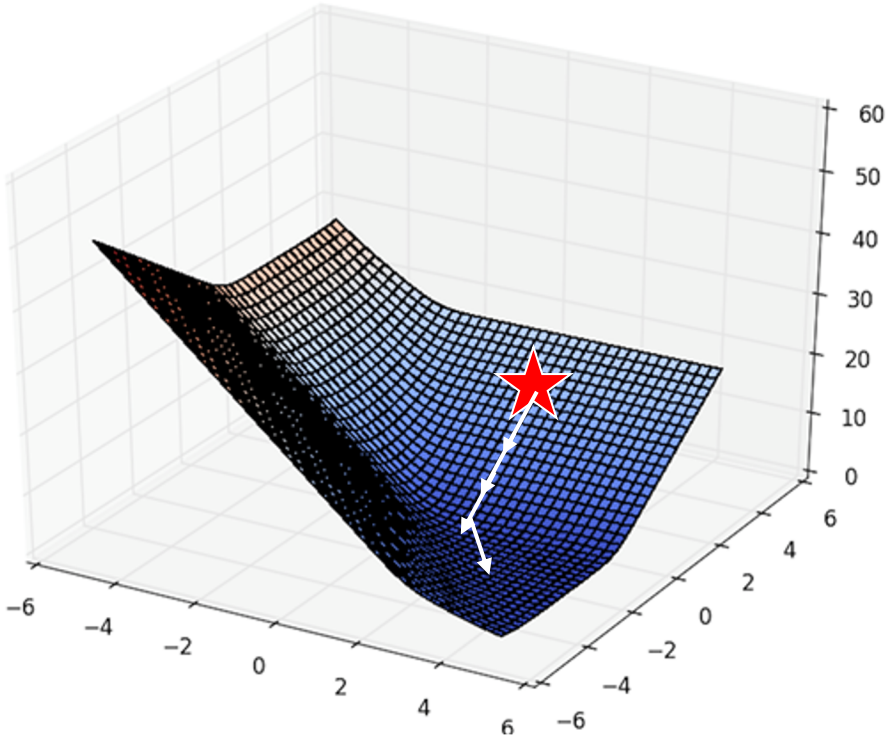
Approach #3: Gradient Descent

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



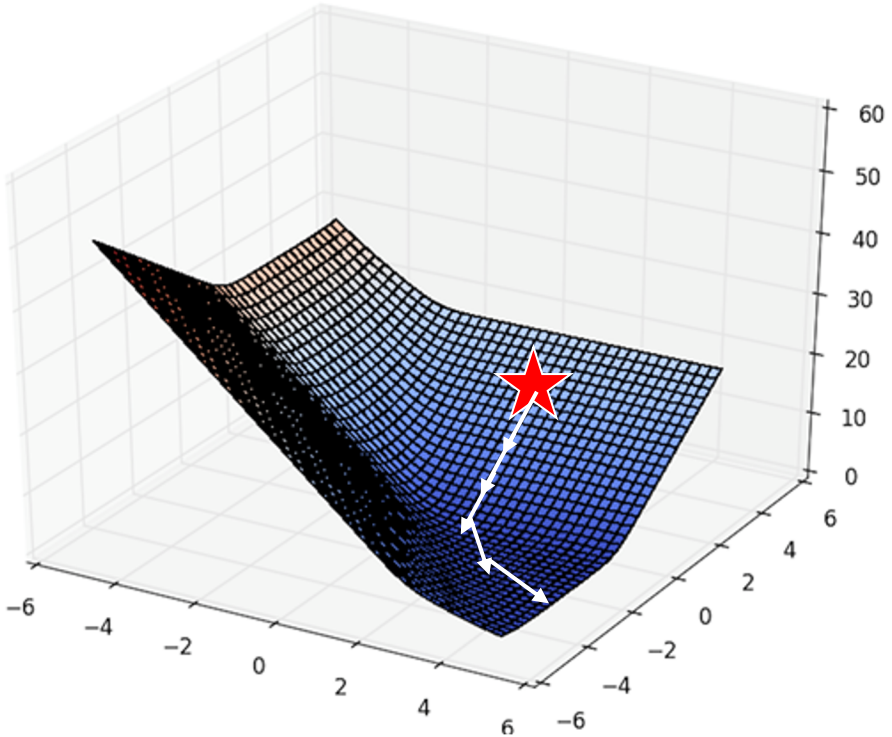
Approach #3: Gradient Descent

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



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.

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

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

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

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.

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 data).

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

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

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

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

You Try:

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

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

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

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

Gradient Descent Wrap up

- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- **Gradient Descent Wrap up:**
  - Stochastic Gradient Descent
  - Convexity

Gradient Descent

Gradient descent algorithm: nudge θ in negative gradient direction until θ converges.

For a model with one parameter (equivalently: input data is one dimensional):

gradient of the loss function evaluated at current θ

$$\theta^{t+1} = \theta^t - \alpha \frac{d}{d\theta} L(\theta, \vec{x}, \vec{y})$$

Next value for θ

θ: Model weights L: loss function

α: Learning rate (ours was constant, but other techniques have α decrease over time)

y: True values from training data

Gradient Descent

Gradient descent algorithm: nudge θ in negative gradient direction until θ converges.

For a model with multiple parameters:

gradient of the loss function evaluated at current θ

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

Next value for θ

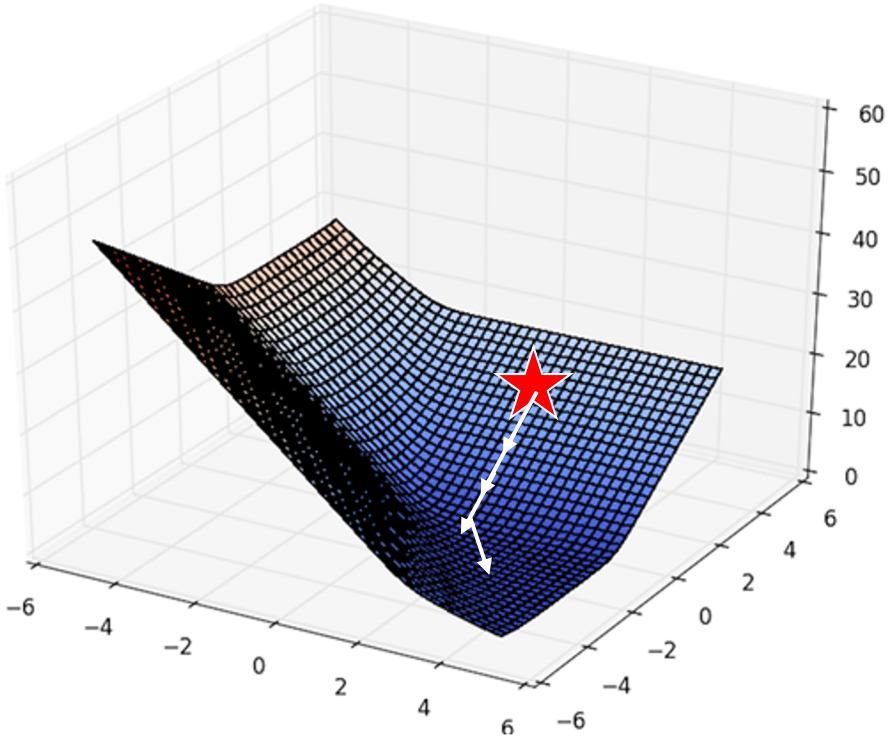
θ: Model weights L: loss function

α: Learning rate (ours was constant, but other techniques have α decrease over time)

y: True values from training data

Gradient Descent

By repeating this process over and over, you can find a local minimum of the function being optimized.



Stochastic Gradient Descent

- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- **Gradient Descent Wrap up:**
  - **Stochastic Gradient Descent**
  - Convexity

Batch Gradient Descent

The algorithm we derived in the last class is more verbosely known as “batch gradient descent”.

- Uses the entire batch of available data to compute the gradient.

gradient of the loss function evaluated at current θ

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

Next value for θ

Impractical in some circumstances. Imagine you have a billions of data points.

- Computing the gradient would require computing the loss for a prediction for EVERY data point, then computing the mean loss across all several billion.

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters.
- Then compute gradient on next 10% of the data. Adjust parameters.
- Then compute gradient on third 10% of the data. Adjust parameters.
- …
- Then compute gradient on last 10% of the data. Adjust parameters.

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters.
- Then compute gradient on next 10% of the data. Adjust parameters.
- Then compute gradient on third 10% of the data. Adjust parameters.
- …
- Then compute gradient on last 10% of the data. Adjust parameters.

Question: Are we done now?

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters.
- Then compute gradient on next 10% of the data. Adjust parameters.
- Then compute gradient on third 10% of the data. Adjust parameters.
- …
- Then compute gradient on last 10% of the data. Adjust parameters.

Question: Are we done now? **Not unless we were lucky!**

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters.
- Then compute gradient on next 10% of the data. Adjust parameters.
- Then compute gradient on third 10% of the data. Adjust parameters.
- …
- Then compute gradient on last 10% of the data. Adjust parameters.

Question: So what should we do next?

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters.
- Then compute gradient on next 10% of the data. Adjust parameters.
- Then compute gradient on third 10% of the data. Adjust parameters.
- …
- Then compute gradient on last 10% of the data. Adjust parameters.

Question: So what should we do next? **Go through data again.**

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-Batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- Compute gradient on first 10% of the data. Adjust parameters. Then compute gradient on next 10% of the data … Then compute gradient on final 10% of the data. Adjust parameters.
- Repeat the process above until we we either hit some max number of iterations or our error is below some desired threshold.

gradient of the loss function evaluated at current θ

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

Next value for θ

Mini-batch Gradient Descent

In mini-batch gradient descent, we only use a subset of the data when computing the gradient. Example:

- **Compute gradient on first 10% of the data. Adjust parameters. Then compute gradient on next 10% of the data … Then compute gradient on final 10% of the data. Adjust parameters.**
- Repeat the process above until we we either hit some max number of iterations or our error is below some desired threshold.

Each pass in bold is called a **training epoch**.

gradient of the loss function evaluated at current θ

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

Next value for θ

Interpreting a Gradient Computed from a Mini-Batch

Of note: The gradient we compute using only 10% of our data is not the true gradient!

- It's merely an approximation. May not be the absolutely best way down the true loss surface.
- Works well in practice.

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

Batch Size and Sampling

In our example, we used 10% as the size of each mini-batch.

- Interestingly, in real world practice, the size of a mini-batch is usually just a fixed number that is independent of the size of the data set.
- Size of the batch represents the quality of the gradient approximation.
  - All of the data (batch gradient descent): True gradient.
  - 10% of the data: Approximation of the gradient (may not be fastest way down)
- Typical choice for mini-batch size: 32 points.
  - Used regardless of how many billions of data points you may have.
  - See ML literature for more on why.
  - Results in a total of N/32 passes per epoch.

Additionally, rather than going in the order of our original dataset, we typically shuffle the data in between training epochs.

- Details are beyond the scope of our class.

Stochastic Gradient Descent

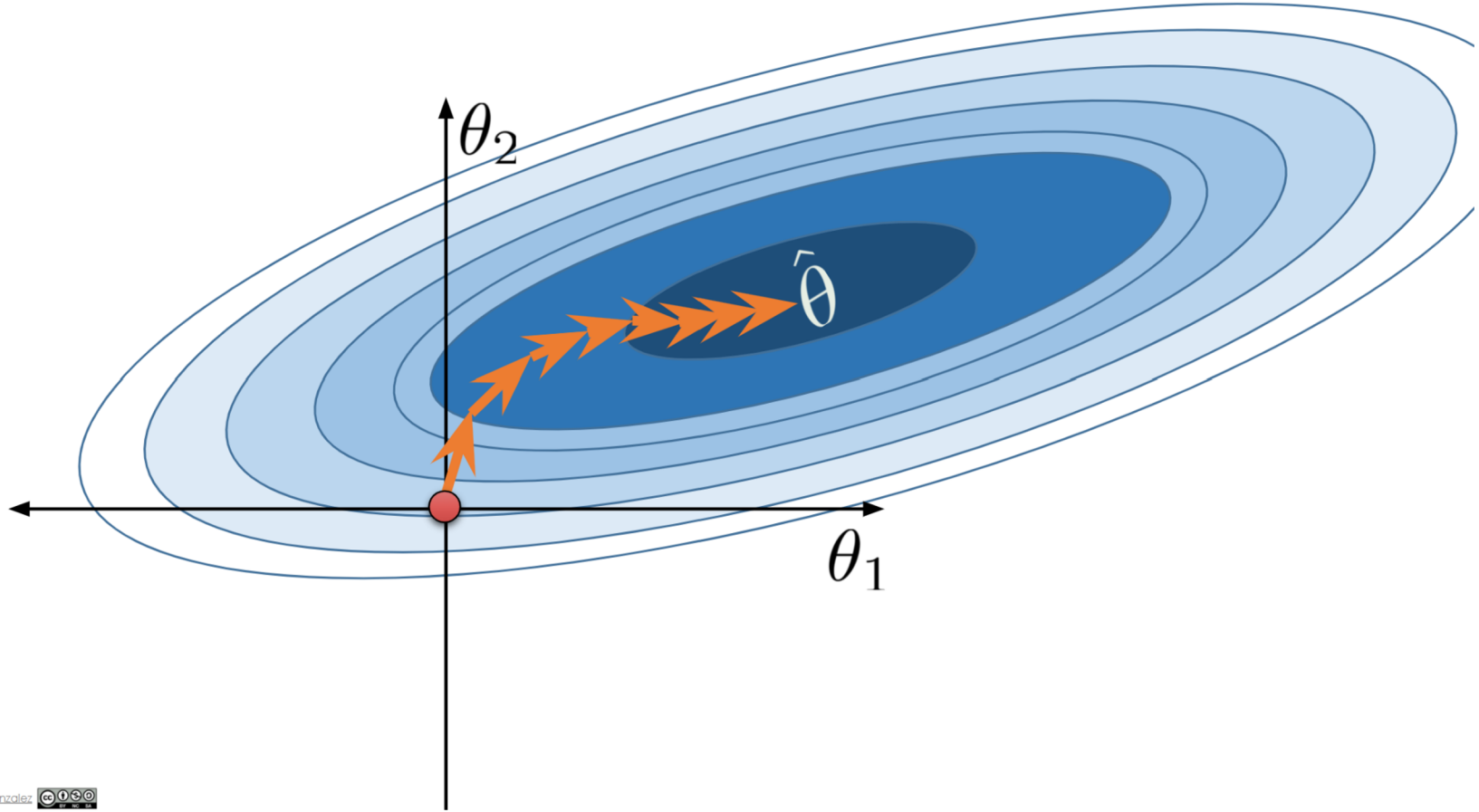
In the most extreme case, we choose a batch size of 1.

- Gradient is computed using only a single data point!
- Number of passes per epoch is therefore the number of data points.
- It may surprise you but this actually works on real world datasets.
    - Why it's surprising: Imagine training an algorithm that recognizes pictures of dogs. Training based on only one dog image at a time means updating potentially millions of parameters based on a single image.
    - Why it works (intuitively): If we average across many epochs across the entire dataset, effect is similar to if we simply compute the true gradient based on the entire dataset.

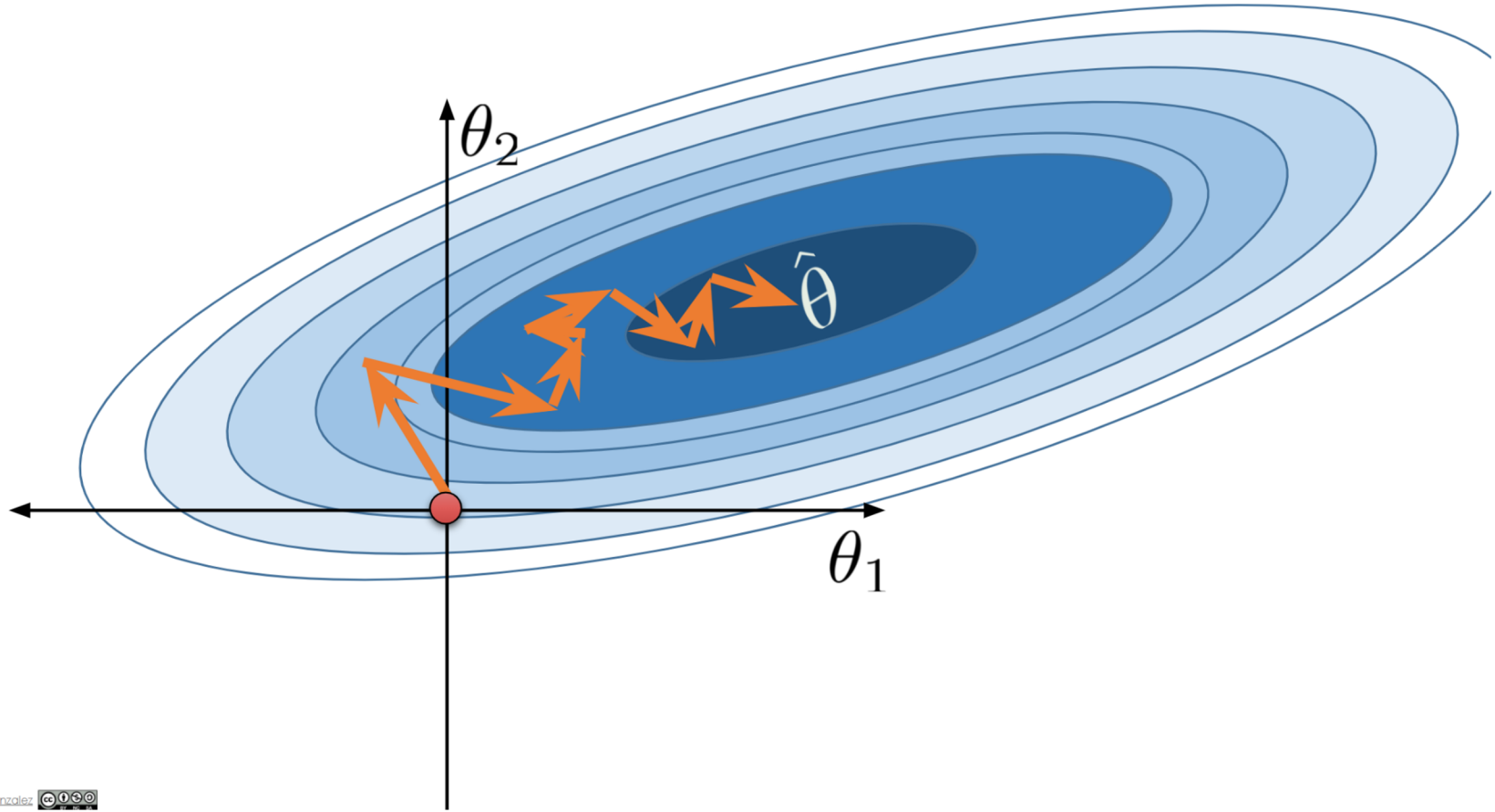
A batch size of 1 is called "stochastic gradient descent".

- Some practicioners use the terms "stochastic gradient descent" and "mini-batch gradient descent" interchangeably.

Gradient Descent



Stochtastic Gradient Descent

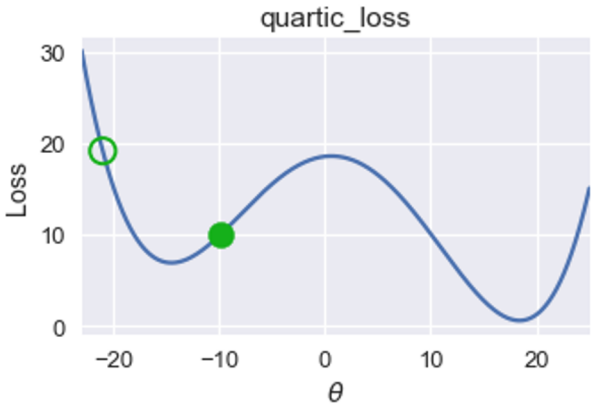


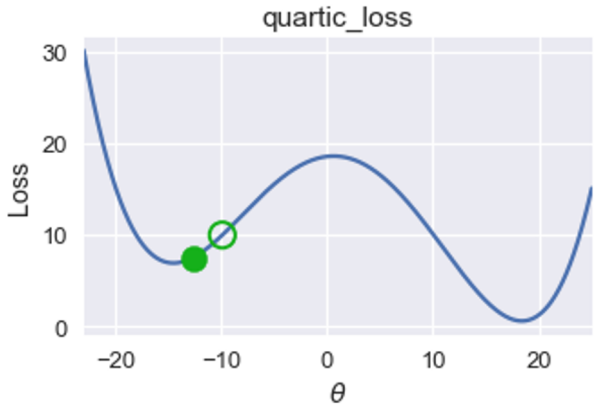
Convexity

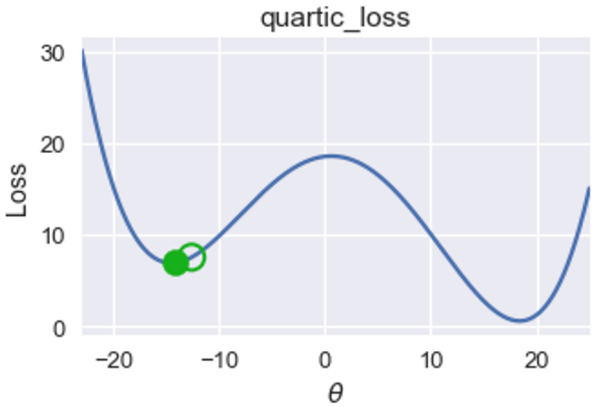
- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions
- Gradient Descent Wrap up:
  - Stochastic Gradient Descent
  - **Convexity**

Gradient Descent Only Finds Local Minima

As we saw, the gradient descent procedure can get stuck in a local minimum.







If a function has a special property called “convexity”, then gradient descent is guaranteed to find the global minimum.

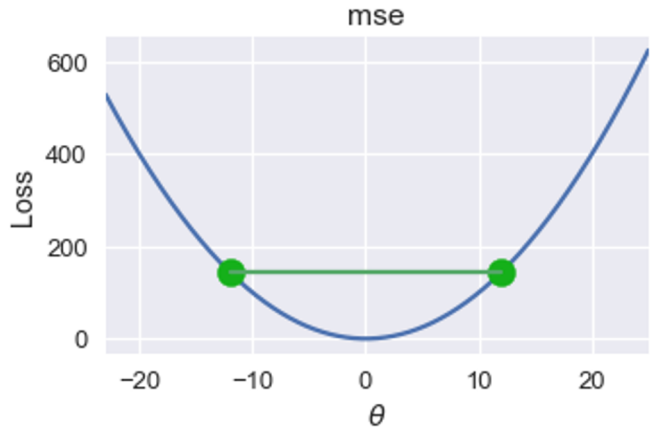
Convexity

Formally, f is convex iff:

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

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

- Or in plain English: If I draw a line between two points on the curve, all values on the curve must be on or below the line.
- Good news, MSE loss is convex! So gradient descent is always going to do a good job minimizing the MSE, and will always find the global minimum.

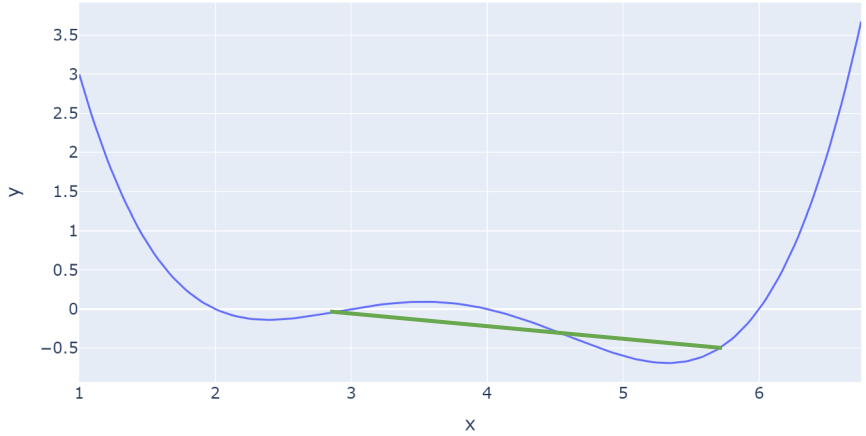


Convexity and Avoidance of Local Minima

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 parameters.

Our arbitrary curve from before is not convex:



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

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

Not all point are below the line!

Convexity and Optimization Difficulty