Gradient Descent

LING 282/482: Deep Learning for Computational Linguistics

C.M. Downey

Fall 2025

Supervised Learning Basics

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
  - In math terms, learning a **function** $f(x) = y$

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
  - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
  - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples
  - $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
    - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples
    - $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
    - The dataset contains **pairs of inputs and outputs**

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
  - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples
  - $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
  - The dataset contains **pairs of inputs and outputs**
  - Ex: speed $\rightarrow$ whether you get a speeding ticket

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
    - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples
    - $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
    - The dataset contains **pairs of inputs and outputs**
    - Ex: speed $\rightarrow$ whether you get a speeding ticket
        - {(30, False), (33, False), (35, False), (37, True), (39, True)}

Supervised Learning Basics

- Overall idea: learn a **mapping** between **inputs** X and **outputs** Y
  - In math terms, learning a **function** $f(x) = y$
- The function is learned from a **dataset** of examples
  - $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$
  - The dataset contains **pairs of inputs and outputs**
  - Ex: speed $\rightarrow$ whether you get a speeding ticket
    - {(30, False), (33, False), (35, False), (37, True), (39, True)}
- Goal: learn the function that **best matches the dataset**

Learning a Function

Learning a Function

- We want to find a function $f : X \to Y$ such that...

Learning a Function

- We want to find a function $f : X \to Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$

Learning a Function

- We want to find a function $f : X \rightarrow Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y

Learning a Function

- We want to find a function $f : X \to Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y
    - $\in$ means "is an element of" or just "in"

Learning a Function

- We want to find a function $f : X \rightarrow Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y
    - $\in$ means "is an element of" or just "in"
  - The function f also **generalizes** well to **new data** (examples not in D)

Learning a Function

- We want to find a function $f : X \rightarrow Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y
    - $\in$ means "is an element of" or just "in"
  - The function f also **generalizes** well to **new data** (examples not in D)
- How do we know **what kind of function** to learn?

Learning a Function

- We want to find a function $f: X \to Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y
    - $\in$ means "is an element of" or just "in"
  - The function f also **generalizes** well to **new data** (examples not in D)
- How do we know **what kind of function** to learn?
  - Infinitely many to choose from

Learning a Function

- We want to find a function $f : X \to Y$ such that...
  - $\hat{y}_i = f(x_i)$ is **"close"** to the true y_i for all $(x_i, y_i) \in D$
    - $\hat{y}$, pronounced "y-hat", is the **predicted value** of y
    - $\in$ means "is an element of" or just "in"
  - The function f also **generalizes** well to **new data** (examples not in D)
- How do we know **what kind of function** to learn?
  - Infinitely many to choose from
  - Solution: learn the weights of a **parameterized function**

Parameterized Functions

Parameterized Functions

- A learning searches for a function f in a space of **possible functions**
- Parameters define a **family** of functions that share a common form
  - θ: general symbol for parameters/weights (usually represents **several**)
  - $\hat{y} = f(x; \theta)$: the function $f(x)$, **given parameters** θ
- Example: the **family of linear functions** $f(x) = mx + b$
  - $\theta = \{m, b\}$
  - This defines **all possible lines** (with different slopes and intercepts)
- Later: Neural Networks define their own family of functions

Loss Function

Loss Function

- We need a way to **measure how close** our parameterized function is to the "true" input/output mapping
  - In other words, we want to measure the **error** of our model

Loss Function

- We need a way to **measure how close** our parameterized function is to the "true" input/output mapping
  - In other words, we want to measure the **error** of our model
- "Loss Function": a measure of how much the **predicted output** $\hat{y}$ **diverges** from the **true output** y
  - $\ell(\hat{y}, y) = \ell(f(x, \theta), y)$
  - Common example: **squared error** $\ell(\hat{y}, y) = (\hat{y} - y)^2$ ((Q: why squared?))

Loss Function

- We need a way to **measure how close** our parameterized function is to the "true" input/output mapping
  - In other words, we want to measure the **error** of our model
- "Loss Function": a measure of how much the **predicted output** $\hat{y}$ **diverges** from the **true output** y
  - $\ell(\hat{y}, y) = \ell(f(x, \theta), y)$
  - Common example: **squared error** $\ell(\hat{y}, y) = (\hat{y} - y)^2$ ((Q: why squared?))
- We always want to **minimize the loss/error**
  - This is a type of **optimization problem**, which is a huge subfield of math

Loss Minimization

Loss Minimization

- Optimization problem: find the values of the **parameters** θ that **minimize the loss function**

Loss Minimization

- Optimization problem: find the values of the **parameters** θ that **minimize the loss function**
    - We will view loss as a **function of the parameters:** $\ell(\theta) := \ell(f(x, \theta), y)$

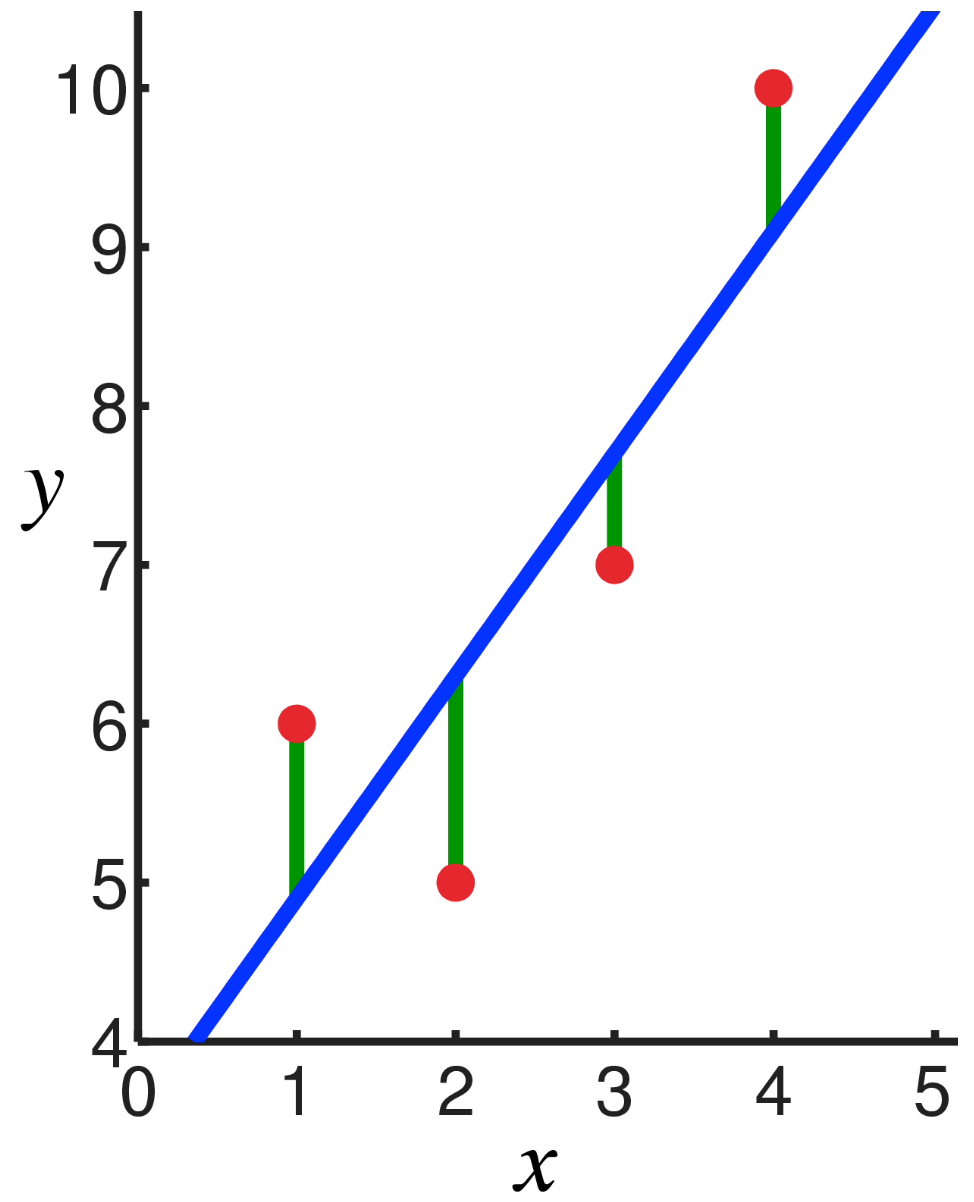
Loss Minimization

- Optimization problem: find the values of the **parameters** θ that **minimize the loss function**
  - We will view loss as a **function of the parameters:** $\ell(\theta) := \ell(f(x, \theta), y)$
  - In math terms, $\theta*$ are the **optimal parameters** $\theta* = \arg\min_{\theta} \ell(\theta)$

Loss Minimization

- Optimization problem: find the values of the **parameters** θ that **minimize the loss function**
  - We will view loss as a **function of the parameters:** $\ell(\theta) := \ell(f(x,\theta), y)$
  - In math terms, θ^* are the **optimal parameters** $\theta^* = \arg\min_{\theta} \ell(\theta)$
- Example: **Linear Regression** ("Least-Squares" method)

$$m^*, b^* = \arg\min_{m,b} \sum_i ((mx_i + b) - y_i)^2$$



Example: Secret Number Game

Guessing a number

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
  - You give me any **input number** (we'll call it x)

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
  - You give me any **input number** (we'll call it x)
  - I'll **add a secret number** to it (call it θ)

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
  - You give me any **input number** (we'll call it x)
  - I'll **add a secret number** to it (call it θ)
  - I'll tell you the **output number** (y)

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
  - You give me any **input number** (we'll call it x)
  - I'll **add a secret number** to it (call it θ)
  - I'll tell you the **output number** (y)
  - You have to **deduce the value of the secret number**

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
    - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
    - You give me any **input number** (we'll call it x)
    - I'll **add a secret number** to it (call it θ)
    - I'll tell you the **output number** (y)
    - You have to **deduce the value of the secret number**
- What is the **equation for the function** that we're applying?

Guessing a number

- We'll illustrate Gradient Descent with a **(very) simple number game**
  - (Trivially easy for humans, we **don't actually need** Gradient Descent to solve it)
- Idea:
  - You give me any **input number** (we'll call it x)
  - I'll **add a secret number** to it (call it θ)
  - I'll tell you the **output number** (y)
  - You have to **deduce the value of the secret number**
- What is the **equation for the function** that we're applying?
  - $\hat{y} = f(x) = x + \theta$

Process

Process

- Here are some **input-output pairs** that define our dataset:
  - $\{(2, 4), (3, 5), (5, 7), (8, 10)\}$

Process

- Here are some **input-output pairs** that define our dataset:
  - $\{(2, 4), (3, 5), (5, 7), (8, 10)\}$
- You can **see that $\theta = 2$,** but how would we **learn this algorithmically?**

Process

- Here are some **input-output pairs** that define our dataset:
  - $\{(2, 4), (3, 5), (5, 7), (8, 10)\}$
- You can **see that $\theta = 2$,** but how would we **learn this algorithmically?**
- First define a **parameterized function** $f(x, \theta)$
  - Model prediction: $\hat{y} = f(x, \theta) = x + \theta$

Process

- Here are some **input-output pairs** that define our dataset:
  - $\{(2, 4), (3, 5), (5, 7), (8, 10)\}$
- You can **see that $\theta = 2$,** but how would we **learn this algorithmically?**
- First define a **parameterized function** $f(x, \theta)$
  - Model prediction: $\hat{y} = f(x, \theta) = x + \theta$
- Then define a **loss/error function**
  - We'll use **Squared Error:** $\ell(\hat{y}, y) = (\hat{y} - y)^2 = (f(x, \theta) - y)^2$

Process

- Here are some **input-output pairs** that define our dataset:
  - $\{(2, 4), (3, 5), (5, 7), (8, 10)\}$
- You can **see that $\theta = 2$,** but how would we **learn this algorithmically?**
- First define a **parameterized function** $f(x, \theta)$
  - Model prediction: $\hat{y} = f(x, \theta) = x + \theta$
- Then define a **loss/error function**
  - We'll use **Squared Error:** $\ell(\hat{y}, y) = (\hat{y} - y)^2 = (f(x, \theta) - y)^2$
- Lastly **learn the optimal value of θ** (i.e. the value that **minimizes the loss**)

Loss function

Loss function

- We cast loss as a **function of the parameter(s)** θ

Loss function

- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x,\theta), y) = (f(x,\theta) - y)^2 = (x + \theta - y)^2$

Loss function

- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x, \theta), y) = (f(x, \theta) - y)^2 = (x + \theta - y)^2$
  - x, y are treated as **constants provided by the data**

Loss function

- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x,\theta),y) = (f(x,\theta) - y)^2 = (x + \theta - y)^2$
  - x, y are treated as **constants provided by the data**
- Plug in the datapoint $(x = 2,\, y = 4)$

Loss function

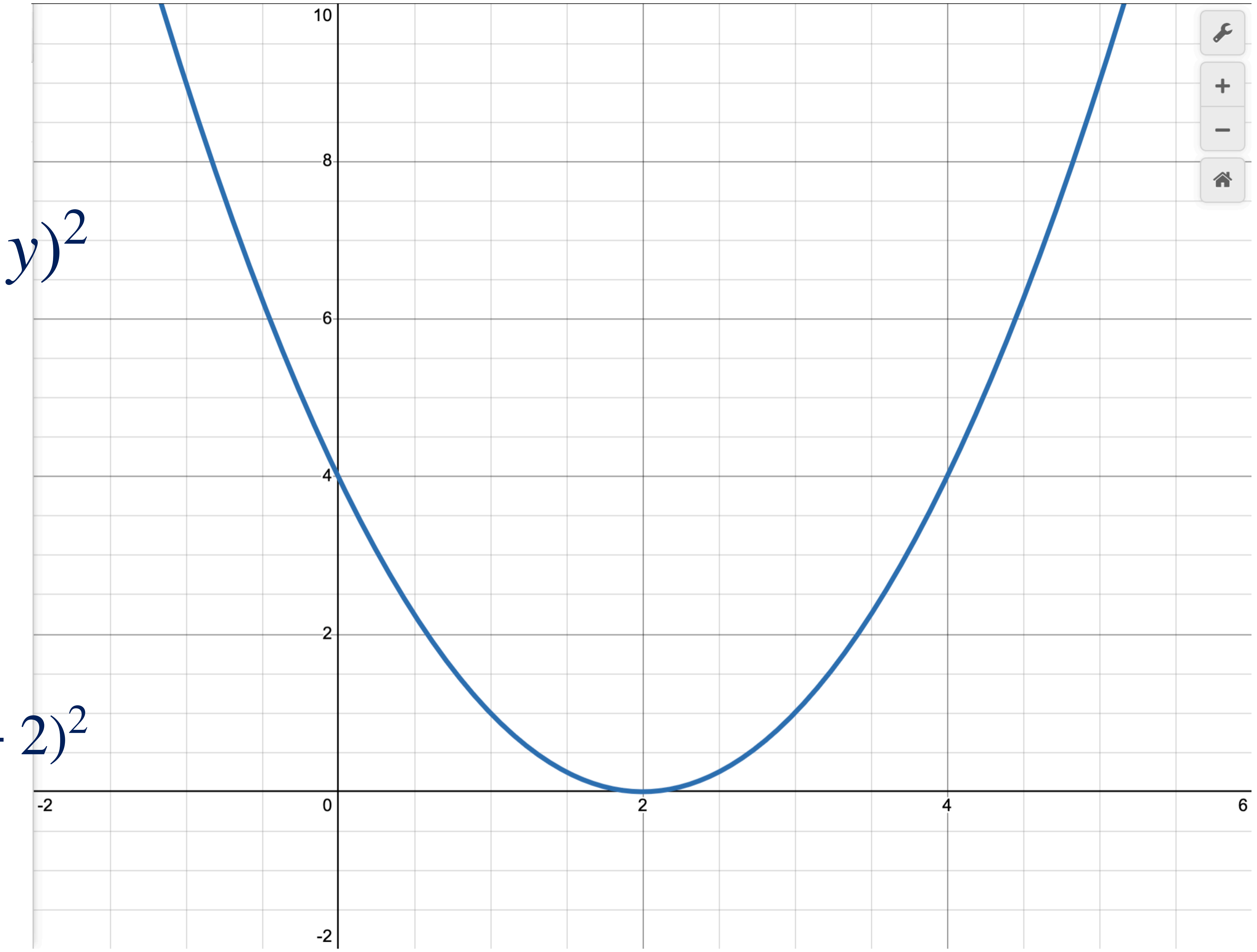
- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x, \theta), y) = (f(x, \theta) - y)^2 = (x + \theta - y)^2$
  - x, y are treated as **constants provided by the data**
- Plug in the datapoint $(x = 2,\, y = 4)$
  - $\ell(f(x, \theta), y) = (x + \theta - y)^2 = (2 + \theta - 4)^2 = (\theta - 2)^2$

Loss function

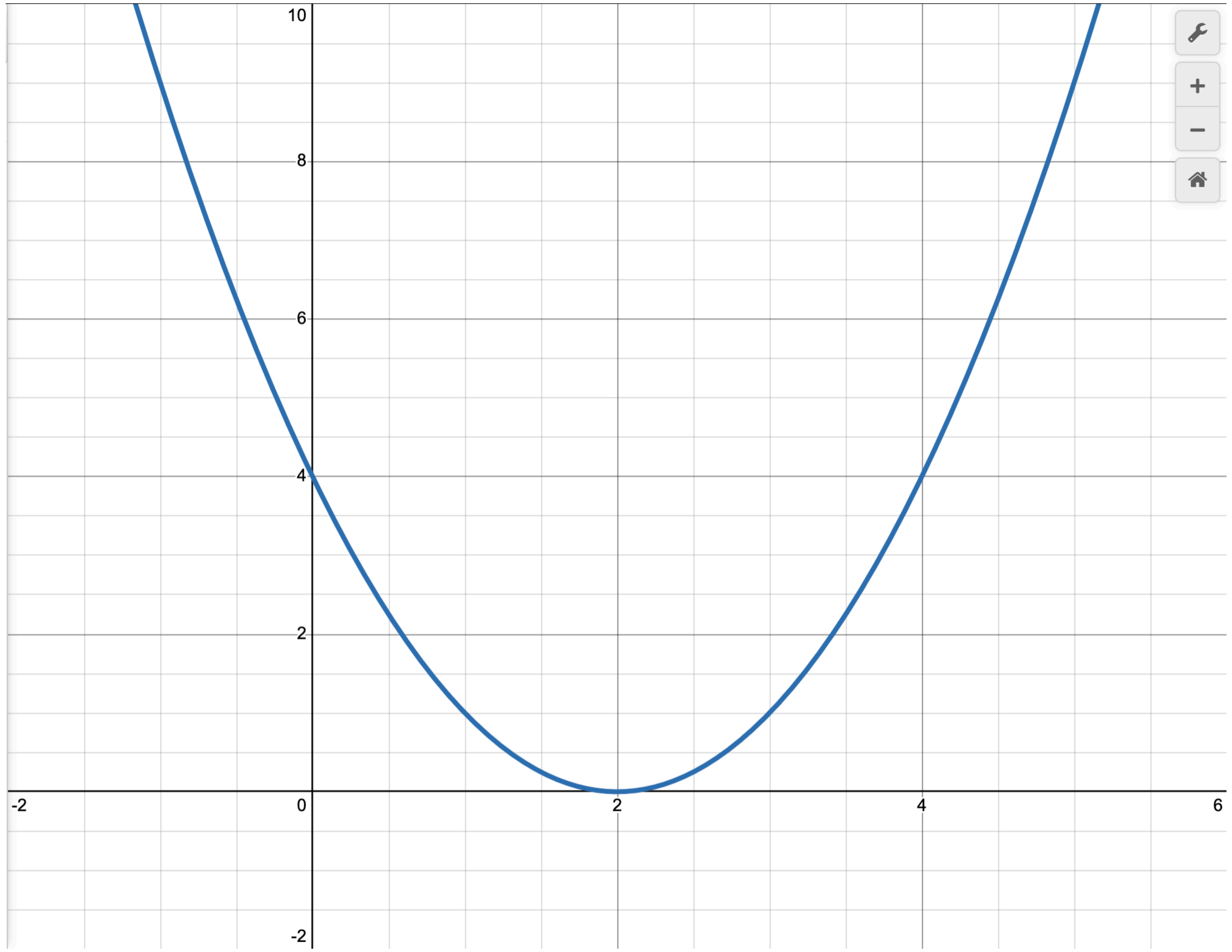
- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x, \theta), y) = (f(x, \theta) - y)^2 = (x + \theta - y)^2$
  - x, y are treated as **constants provided by the data**
- Plug in the datapoint $(x = 2, y = 4)$
  - $\ell(f(x, \theta), y) = (x + \theta - y)^2 = (2 + \theta - 4)^2 = (\theta - 2)^2$
  - We can **plot this loss curve!**

Loss function

- We cast loss as a **function of the parameter(s)** θ
  - $\ell(f(x,\theta),y) = (f(x,\theta) - y)^2 = (x + \theta - y)^2$
  - x, y are treated as **constants provided by the data**
- Plug in the datapoint $(x = 2,\ y = 4)$
  - $\ell(f(x,\theta),y) = (x + \theta - y)^2 = (2 + \theta - 4)^2 = (\theta - 2)^2$
  - We can **plot this loss curve!**

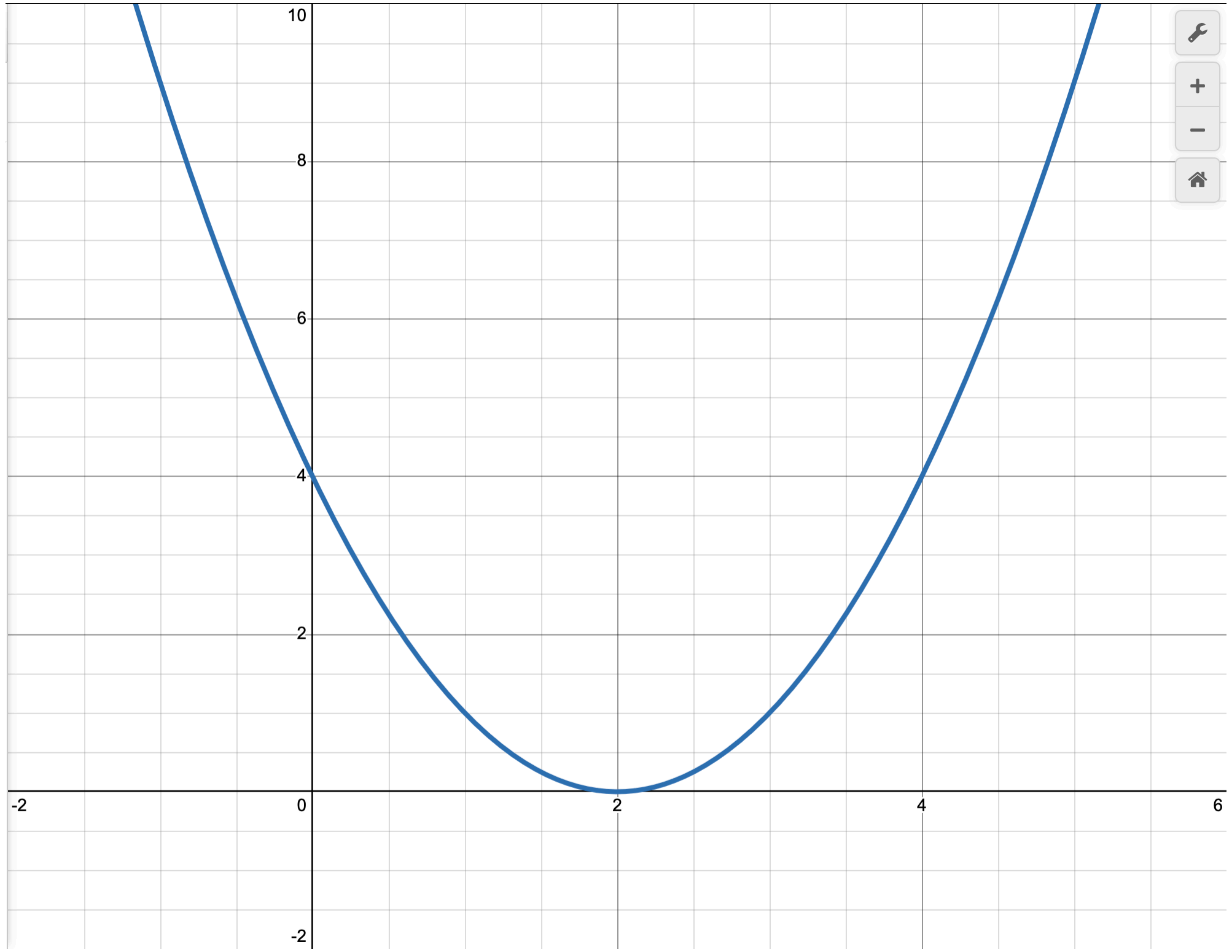


Loss function



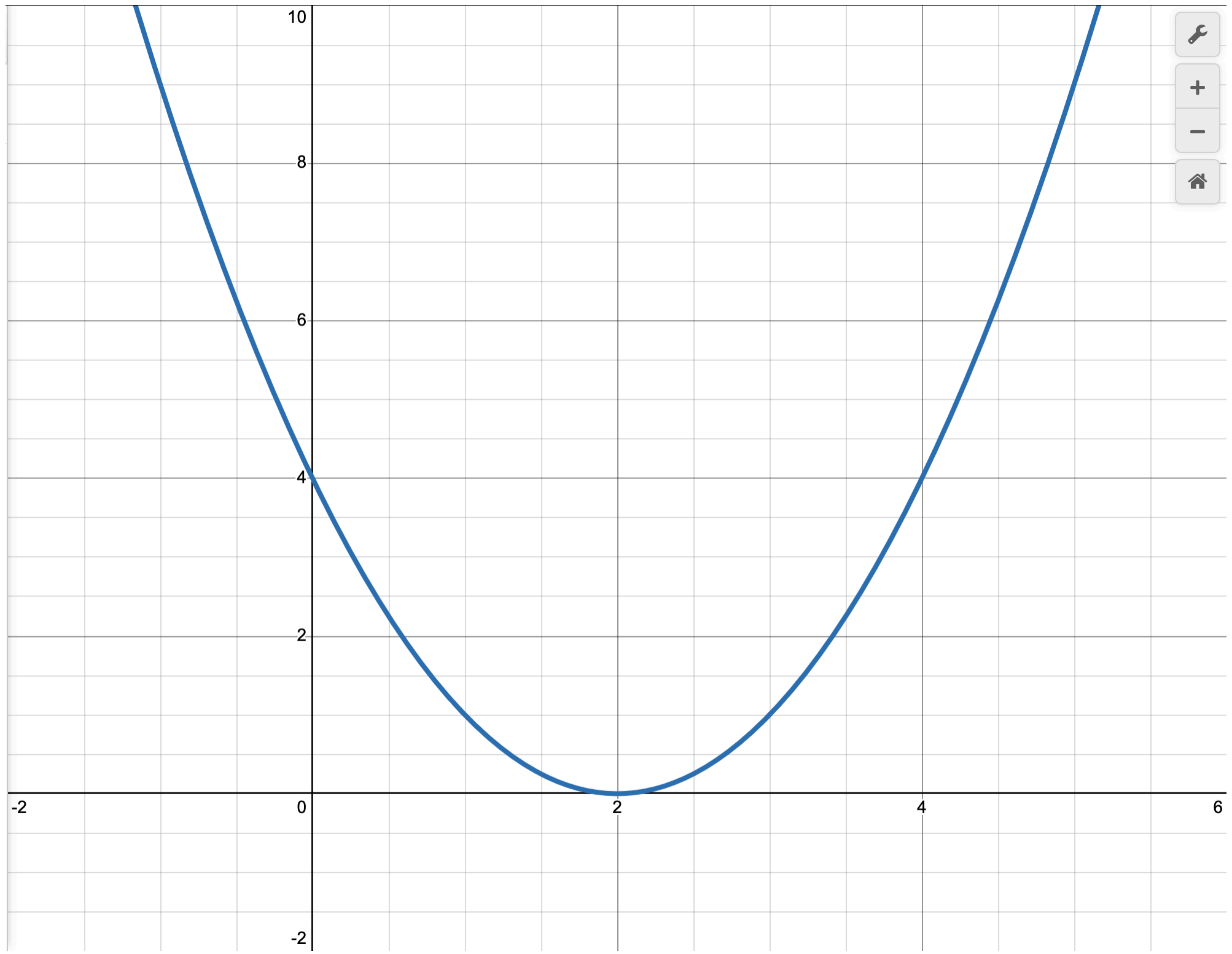
Loss function

- This curve shows the **properties we expect**
  - Loss is **minimized** where $\theta = 2$
  - Loss **grows large** the farther θ is from the true value



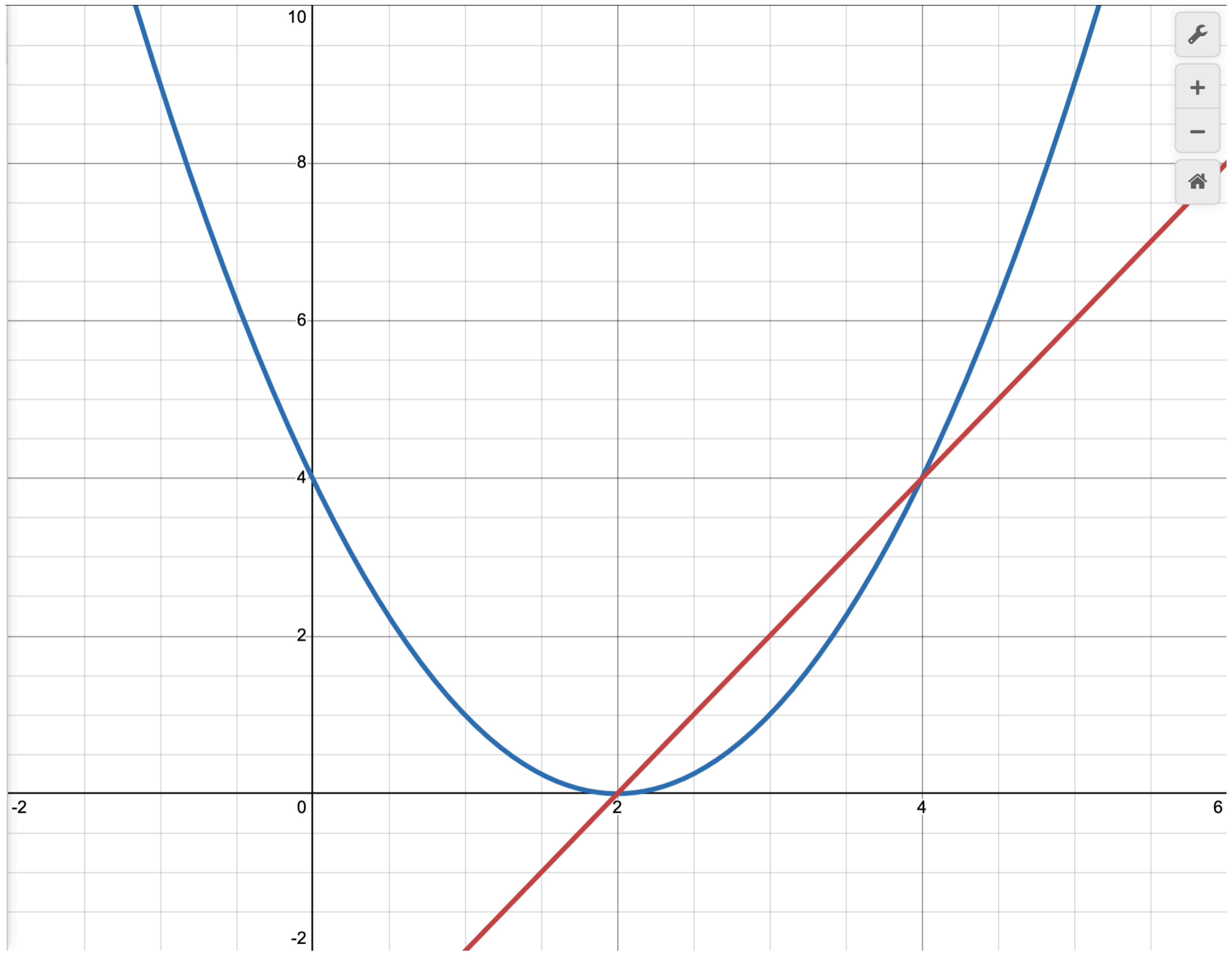
Loss function

- This curve shows the **properties we expect**
  - Loss is **minimized** where $\theta = 2$
  - Loss **grows large** the farther θ is from the true value
- Gradient Descent idea: minimize loss by **following the slope (i.e. "gradient") of the loss function**
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$

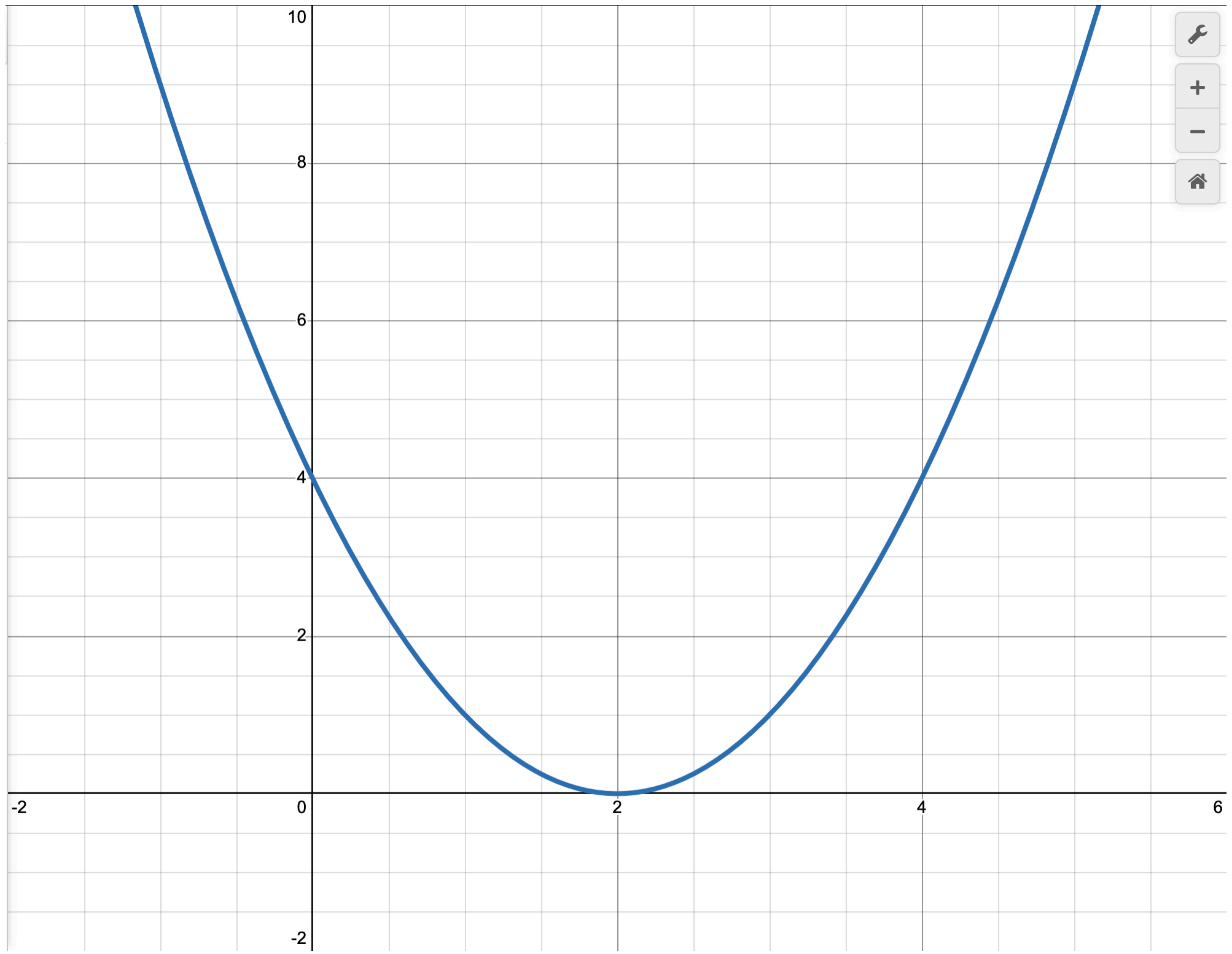


Loss function

- This curve shows the **properties we expect**
  - Loss is **minimized** where $\theta = 2$
  - Loss **grows large** the farther θ is from the true value
- Gradient Descent idea: minimize loss by **following the slope (i.e. "gradient") of the loss function**
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$

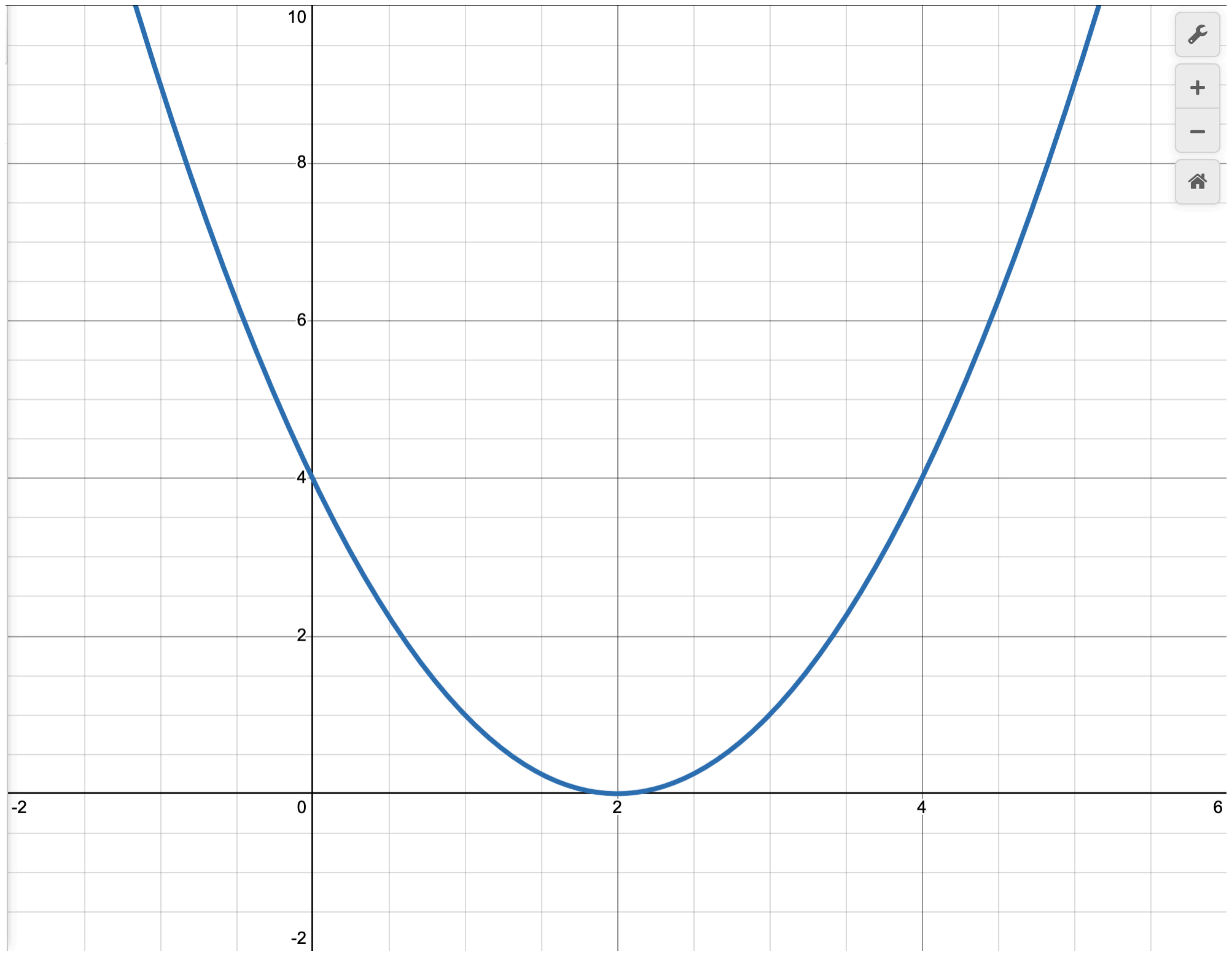


Loss function



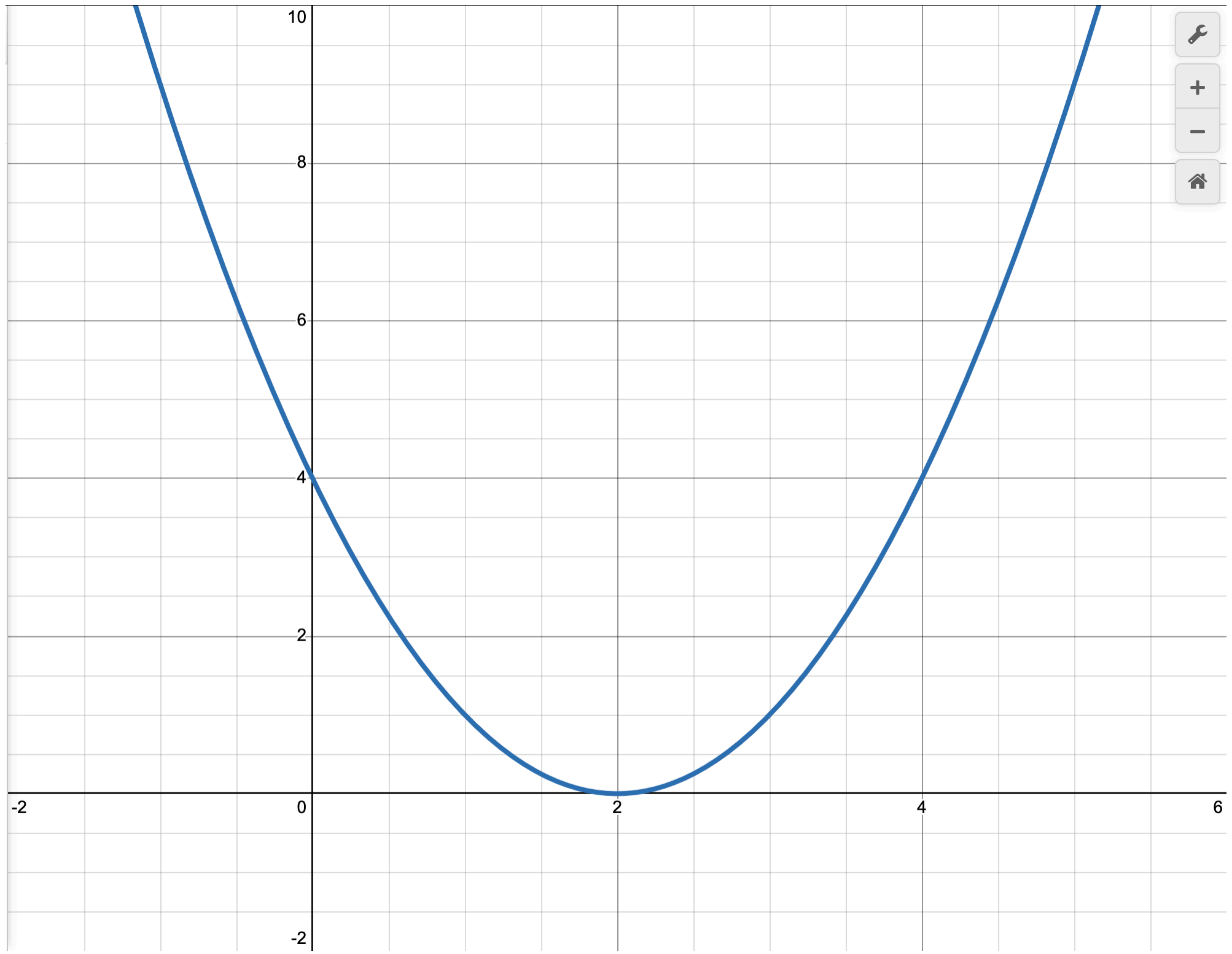
Loss function

- **NOTE**: for this example, **every datapoint** gives us the **exact same loss curve**
  - $(\theta - 2)^2 = (2 + \theta - 4)^2$
  - $= (3 + \theta - 5)^2$
  - $= (5 + \theta - 7)^2$
  - ...etc.



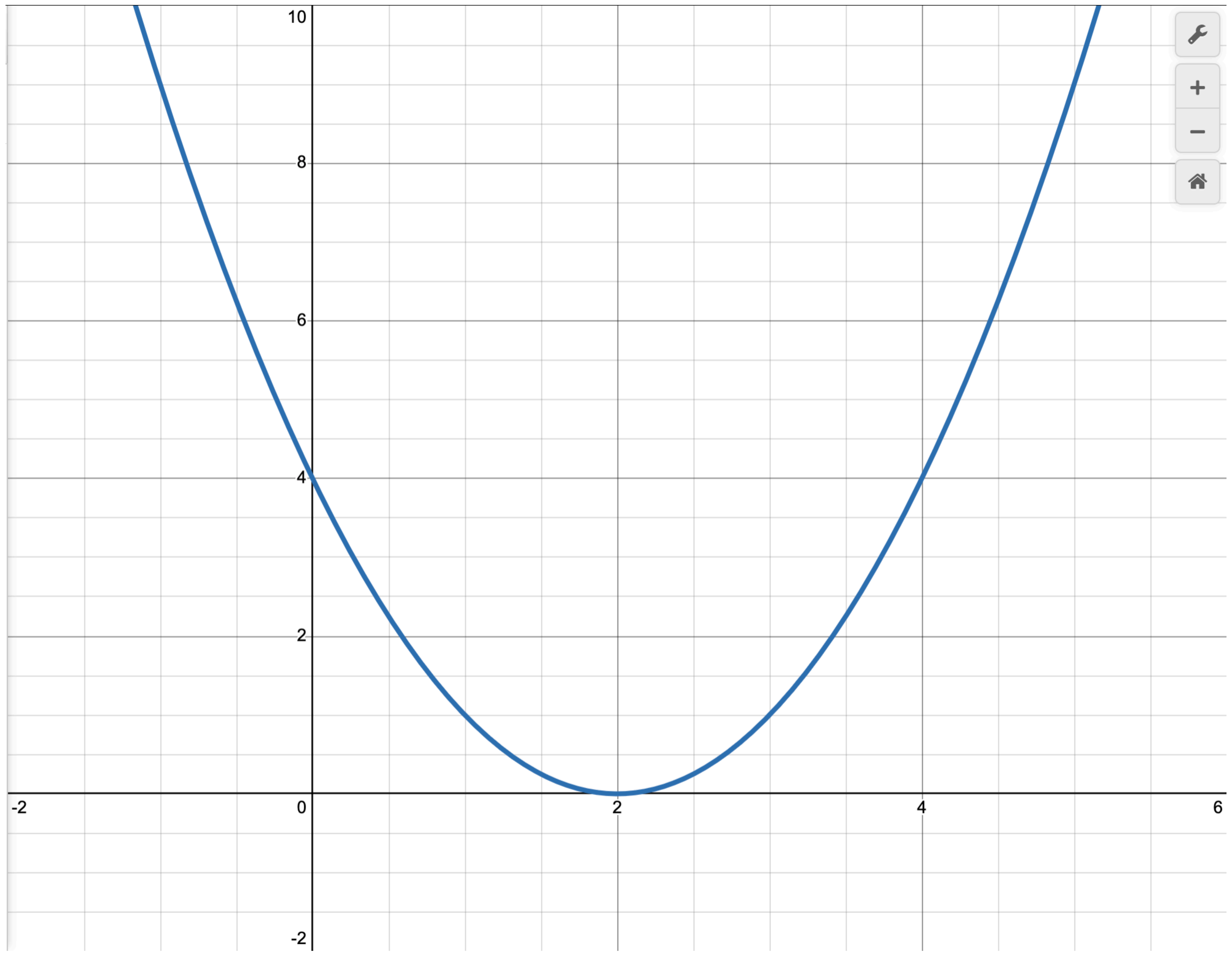
Loss function

- **NOTE**: for this example, **every datapoint** gives us the **exact same loss curve**
  - $(\theta - 2)^2 = (2 + \theta - 4)^2$
  - $= (3 + \theta - 5)^2$
  - $= (5 + \theta - 7)^2$
  - ...etc.
- This is **NOT always the case**
  - Will show an example later on

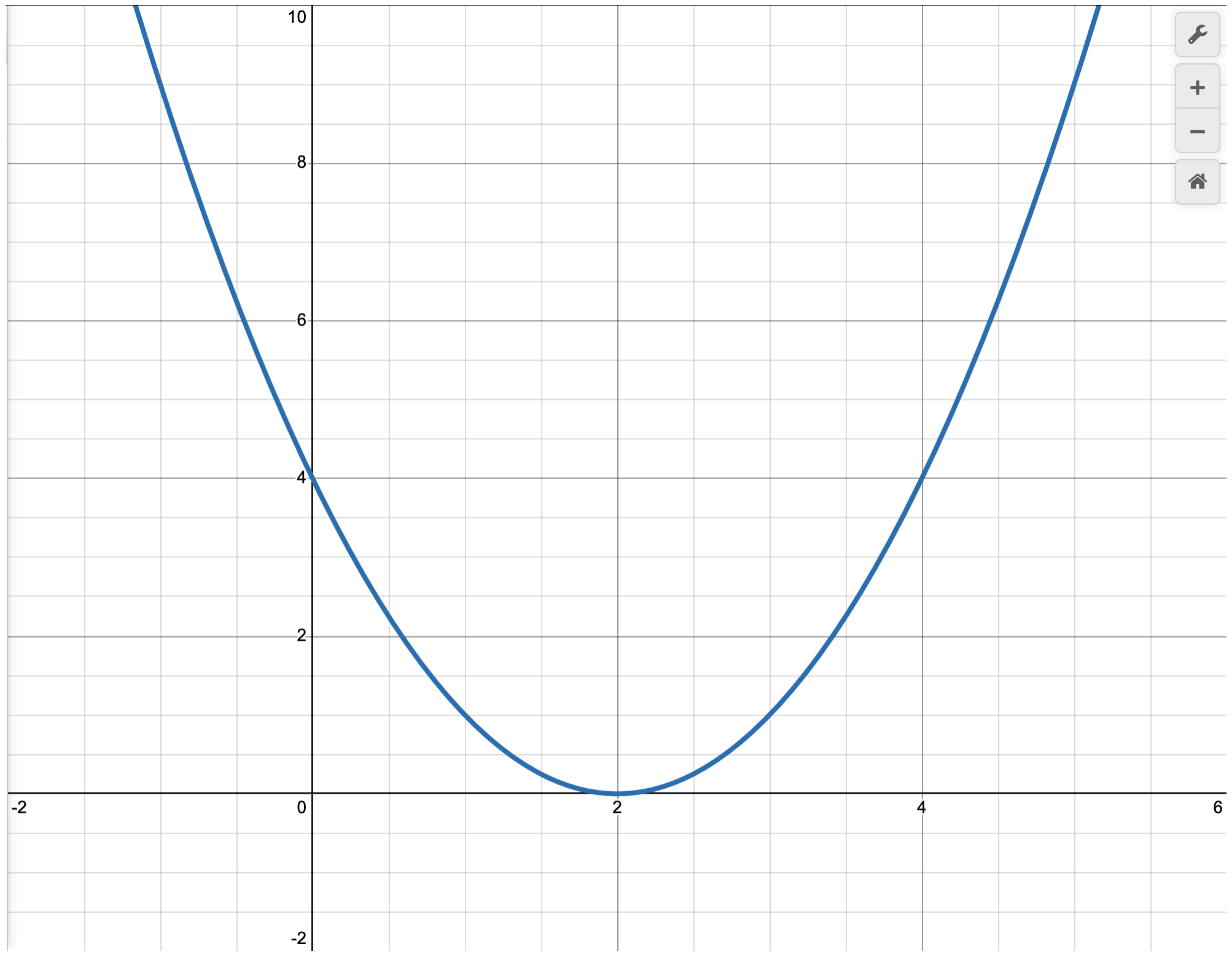


Loss function

- **NOTE**: for this example, **every datapoint** gives us the **exact same loss curve**
  - $(\theta - 2)^2 = (2 + \theta - 4)^2$
  - $= (3 + \theta - 5)^2$
  - $= (5 + \theta - 7)^2$
  - ...etc.
- This is **NOT always the case**
  - Will show an example later on
- For this example **ONLY**, solving for one datapoint solves the whole problem

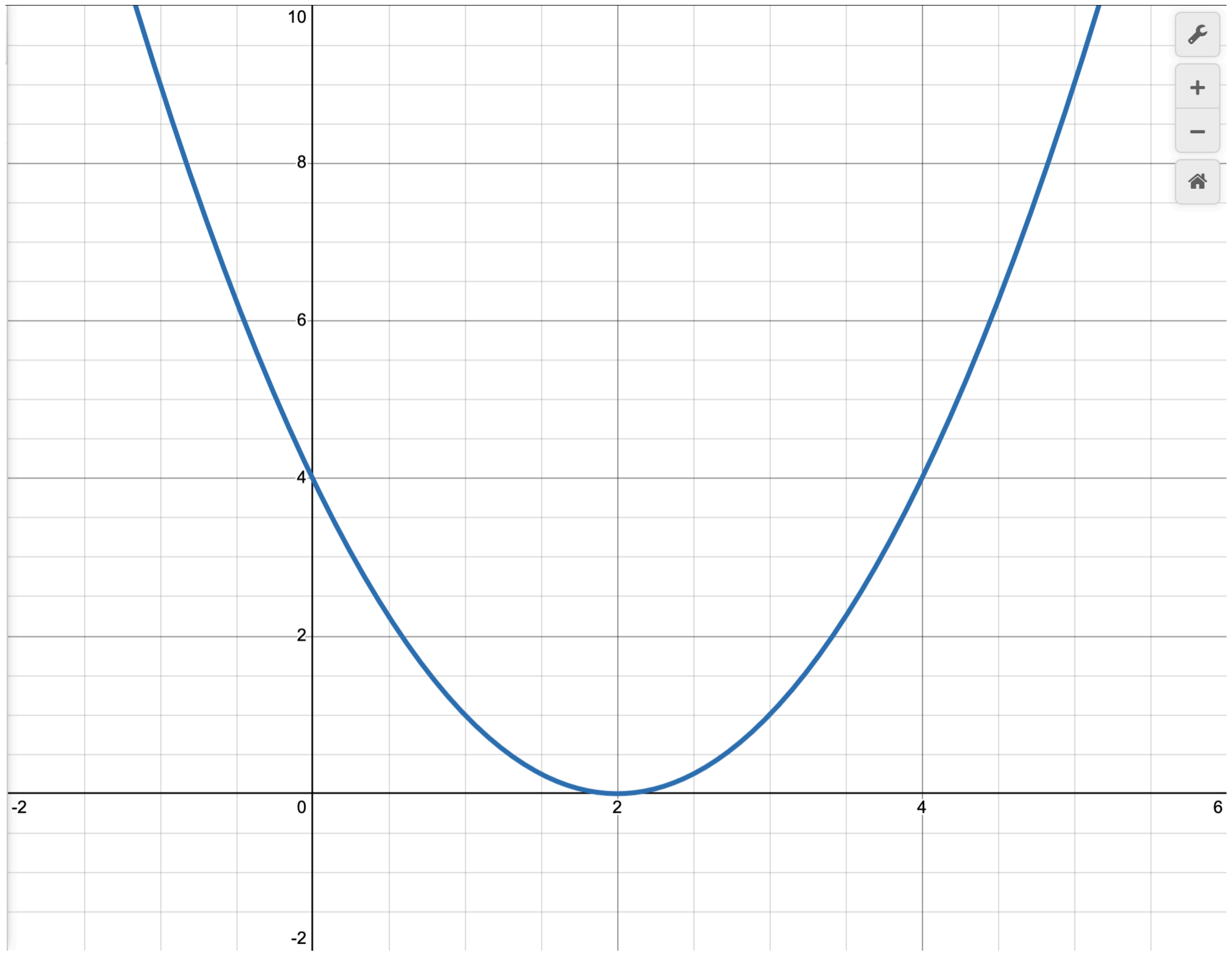


Gradient Descent (first try)



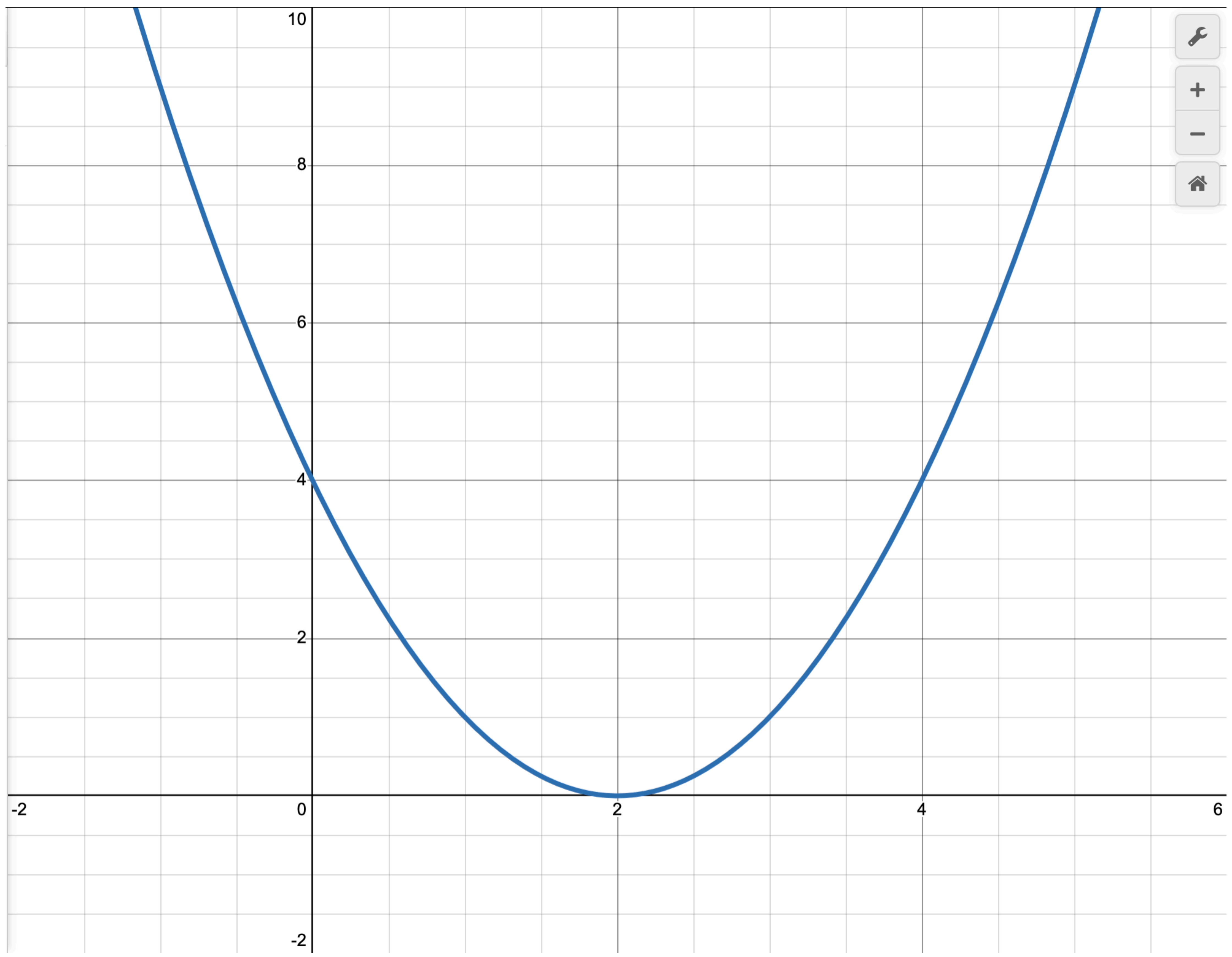
Gradient Descent (first try)

- Gradient Descent is an **iterative algorithm**
  - I.e. you **repeatedly adjust** θ until the loss is minimized



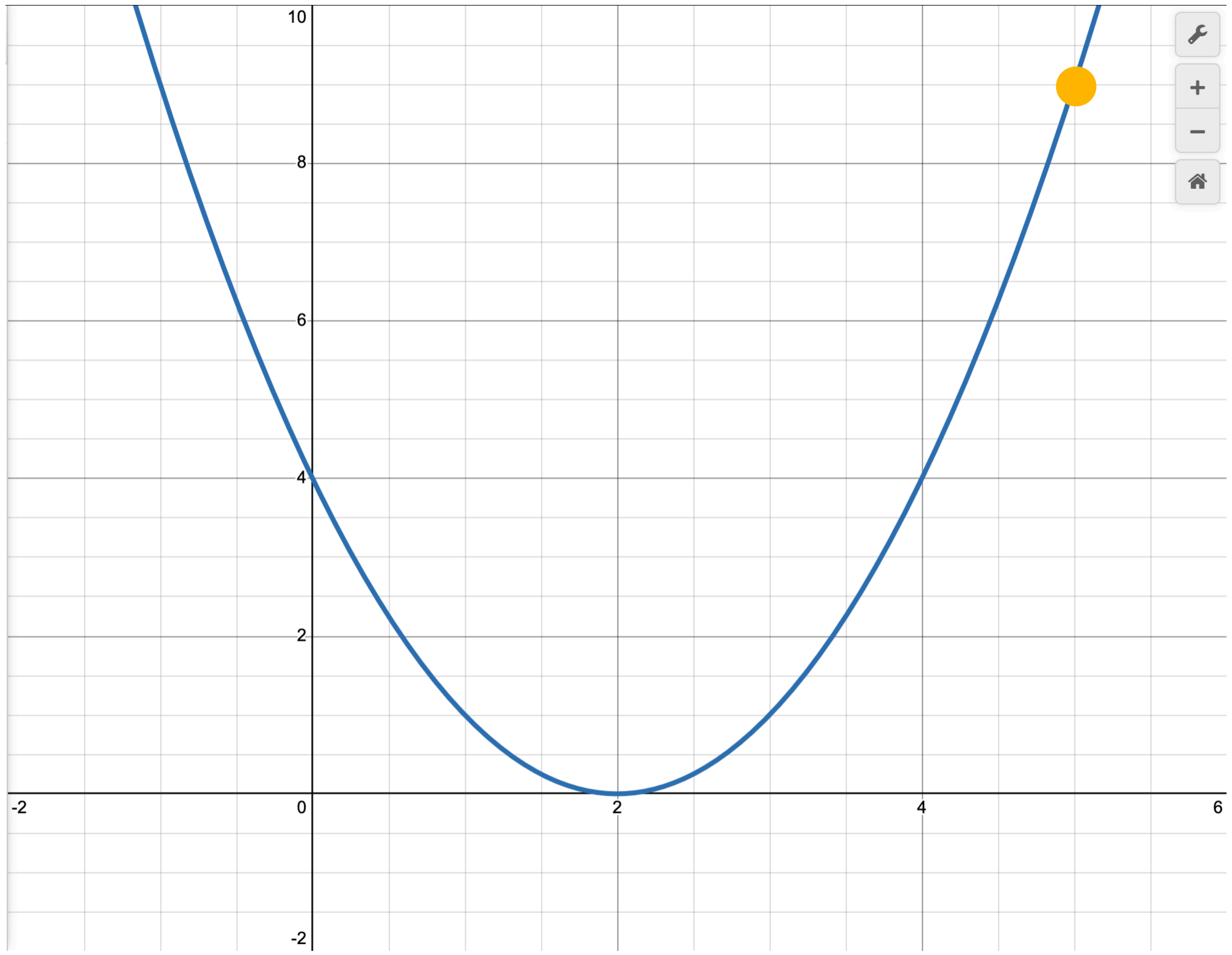
Gradient Descent (first try)

- Gradient Descent is an **iterative algorithm**
  - I.e. you **repeatedly adjust** θ until the loss is minimized
- The **initial value of** θ is a design choice
  - Sometimes **randomly initialized**
  - Sometimes **set to zero**
  - We'll start with $\theta = 5$

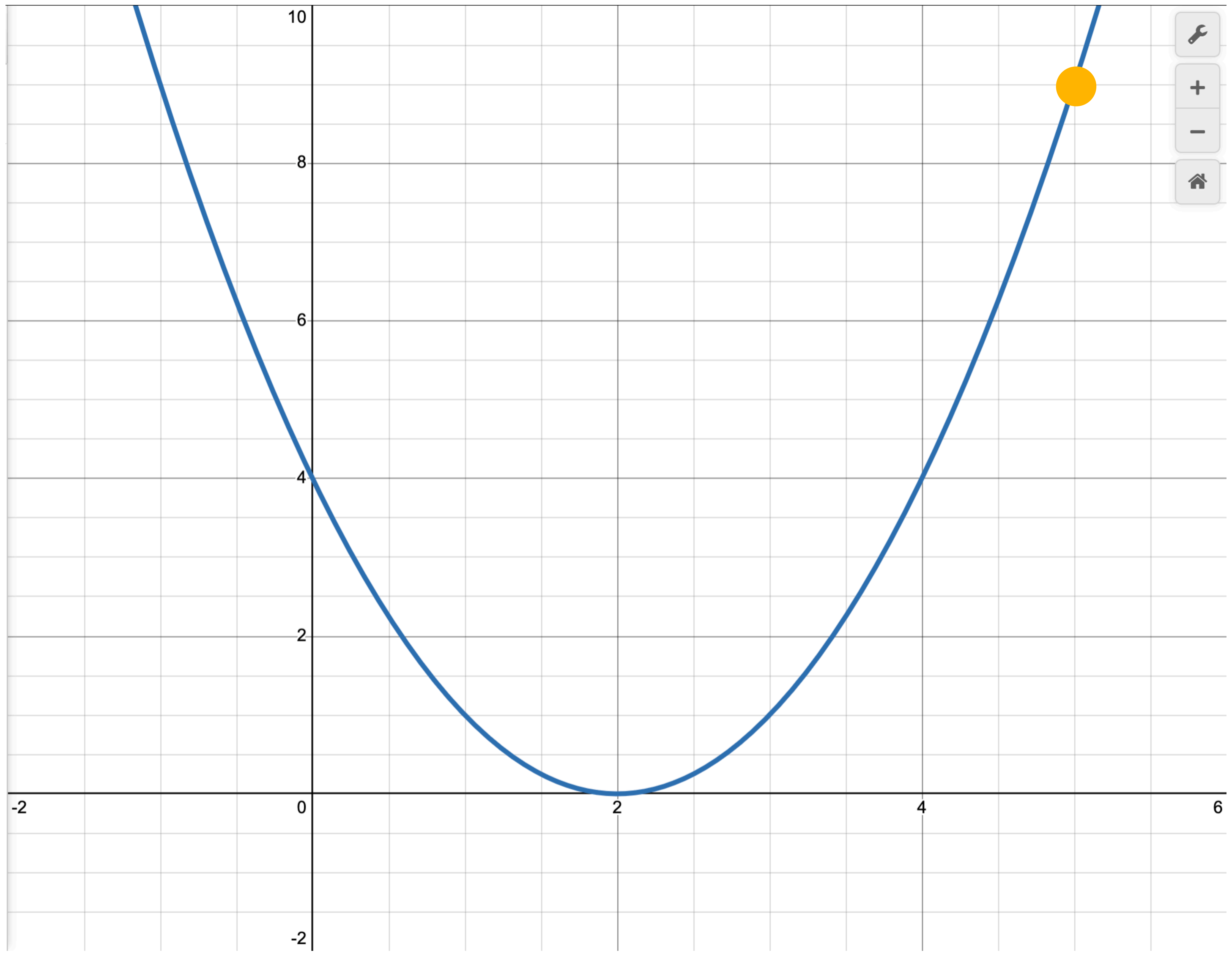


Gradient Descent (first try)

- Gradient Descent is an **iterative algorithm**
  - I.e. you **repeatedly adjust** θ until the loss is minimized
- The **initial value of** θ is a design choice
  - Sometimes **randomly initialized**
  - Sometimes **set to zero**
  - We'll start with $\theta = 5$

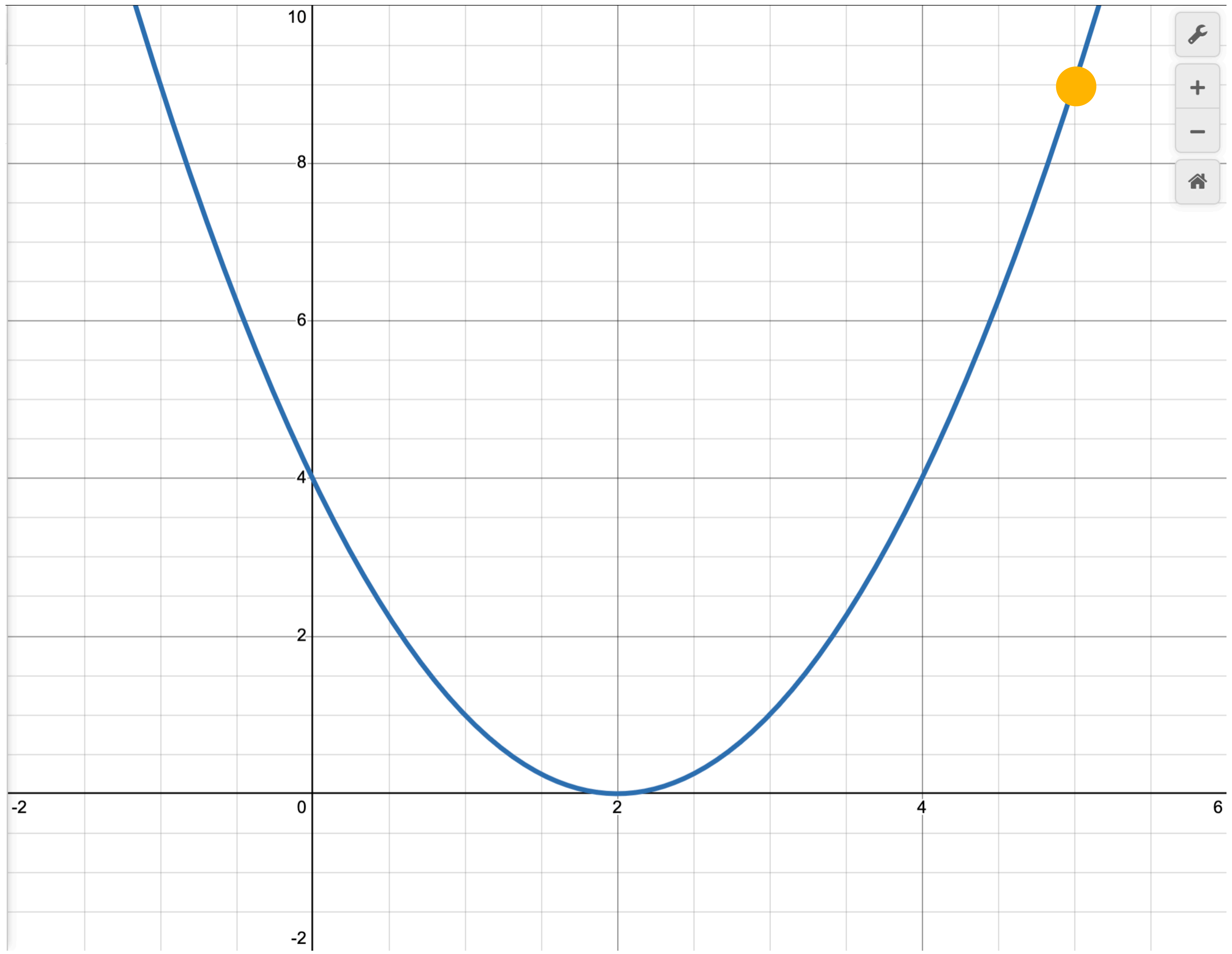


Gradient Descent (first try)



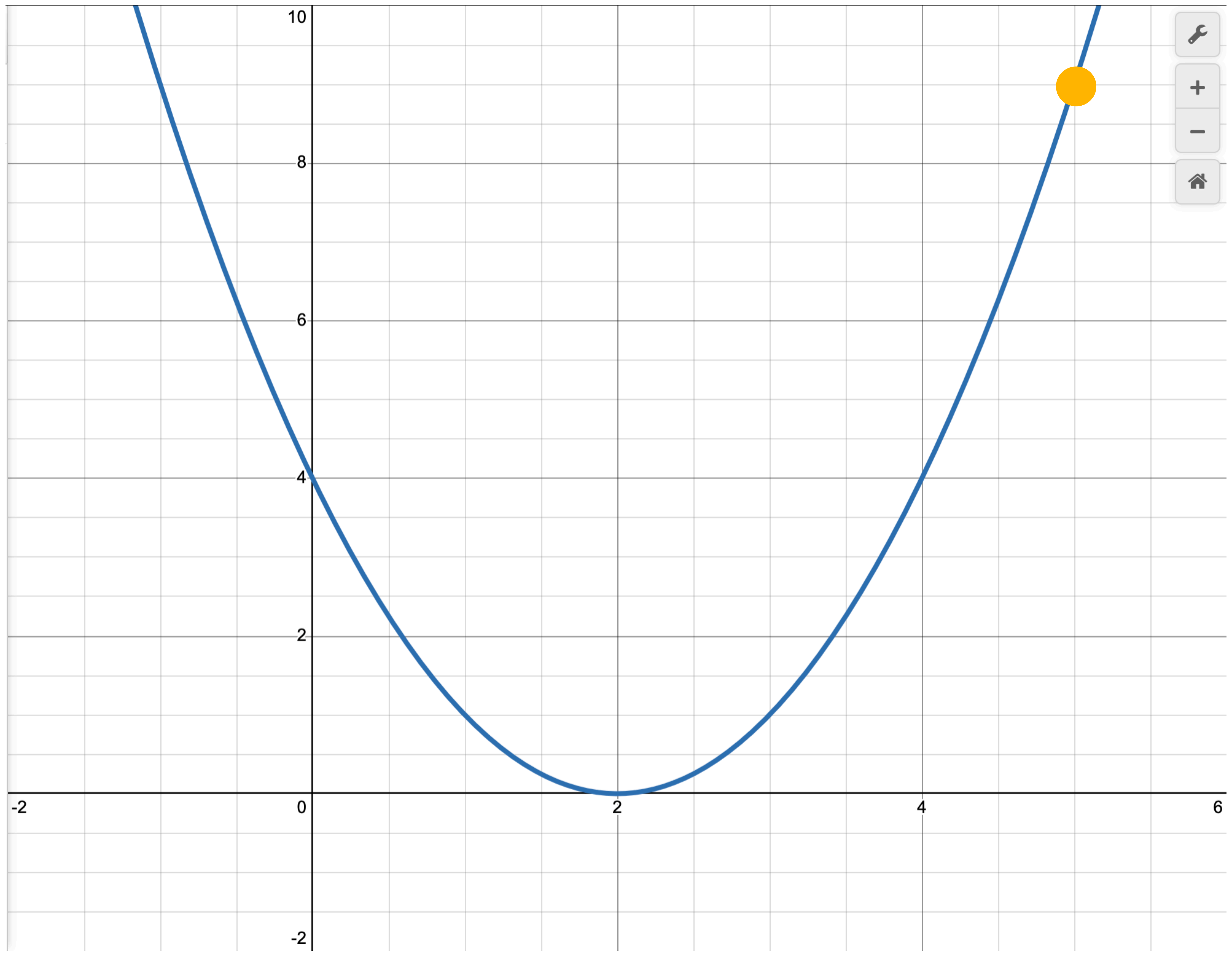
Gradient Descent (first try)

- At each step of the algorithm:



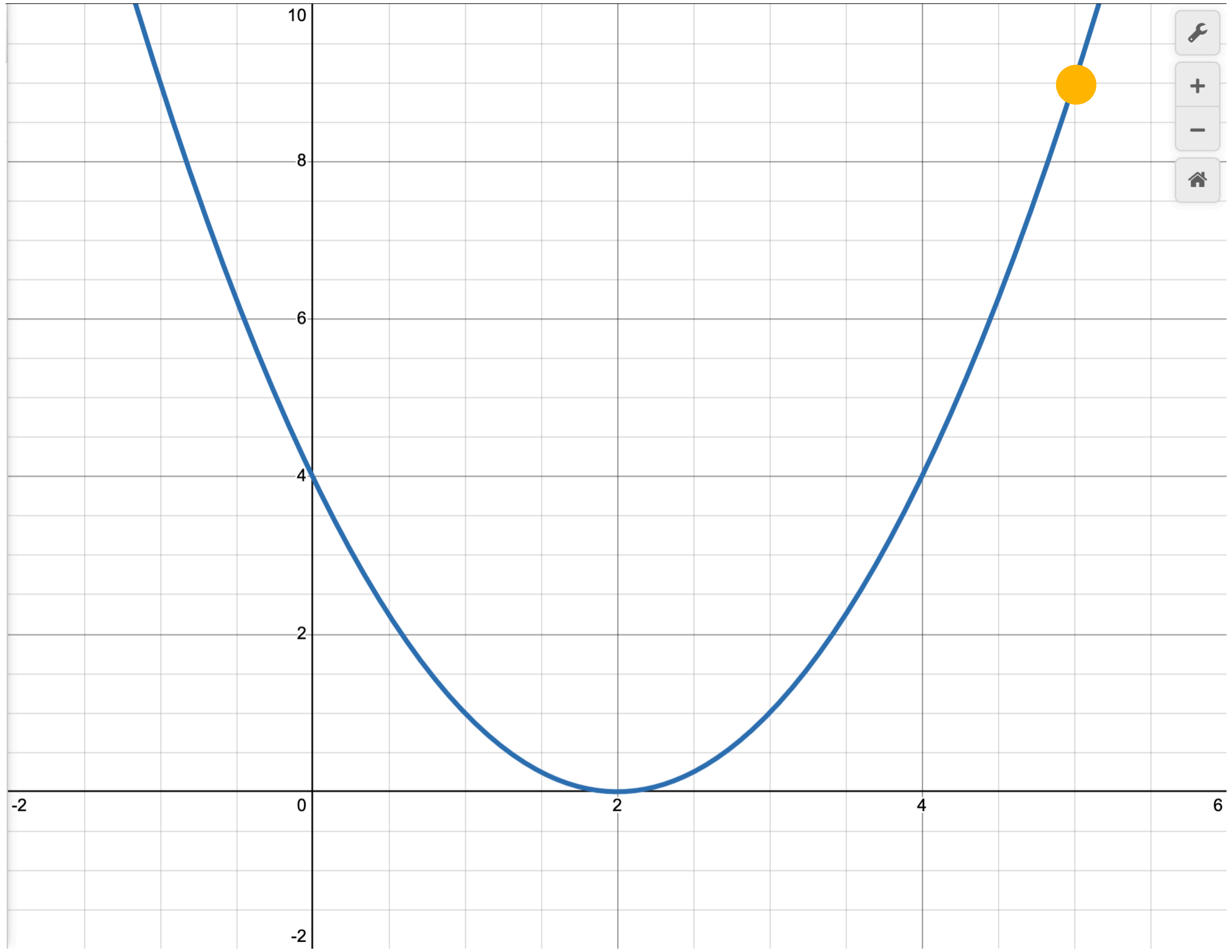
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**



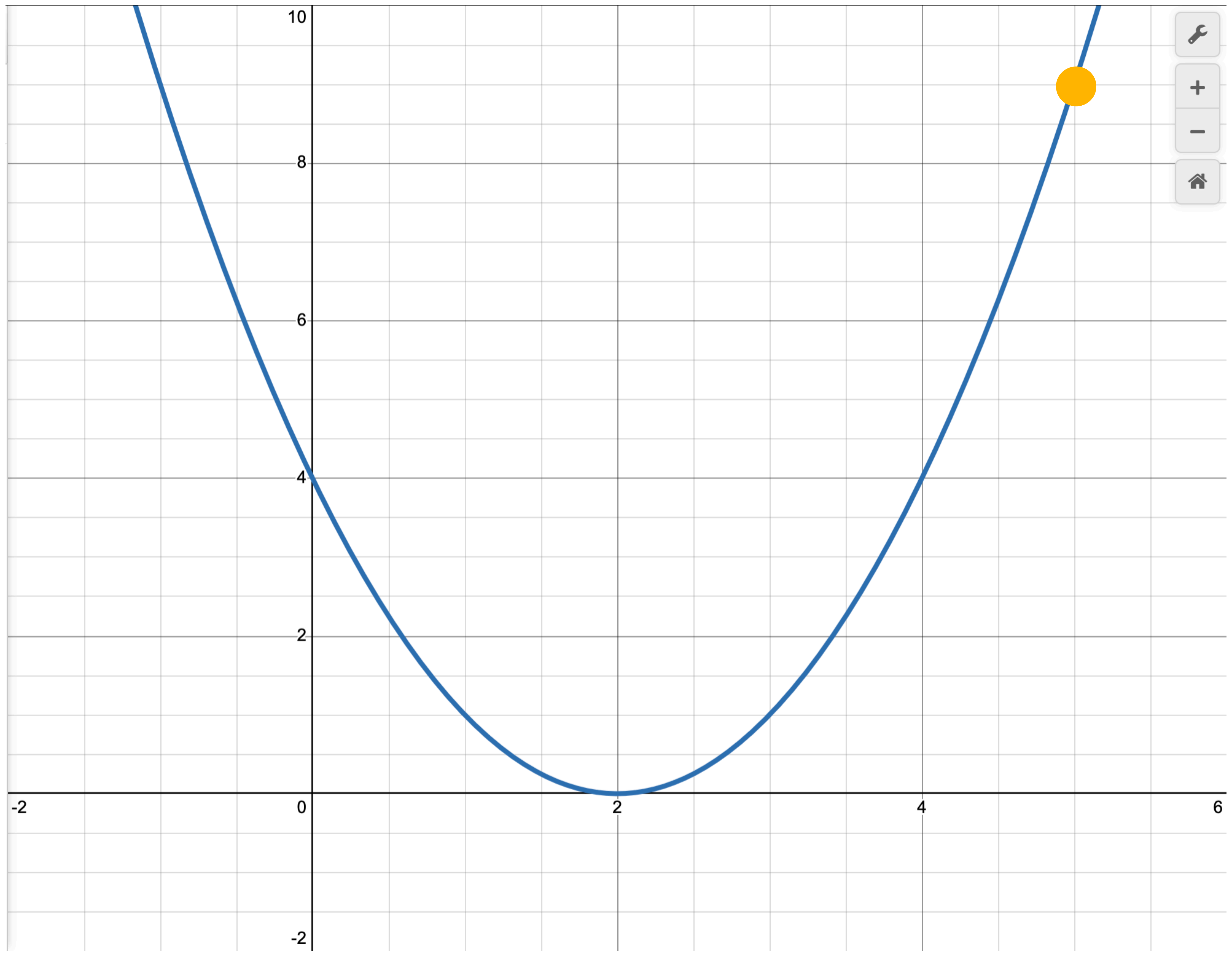
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)



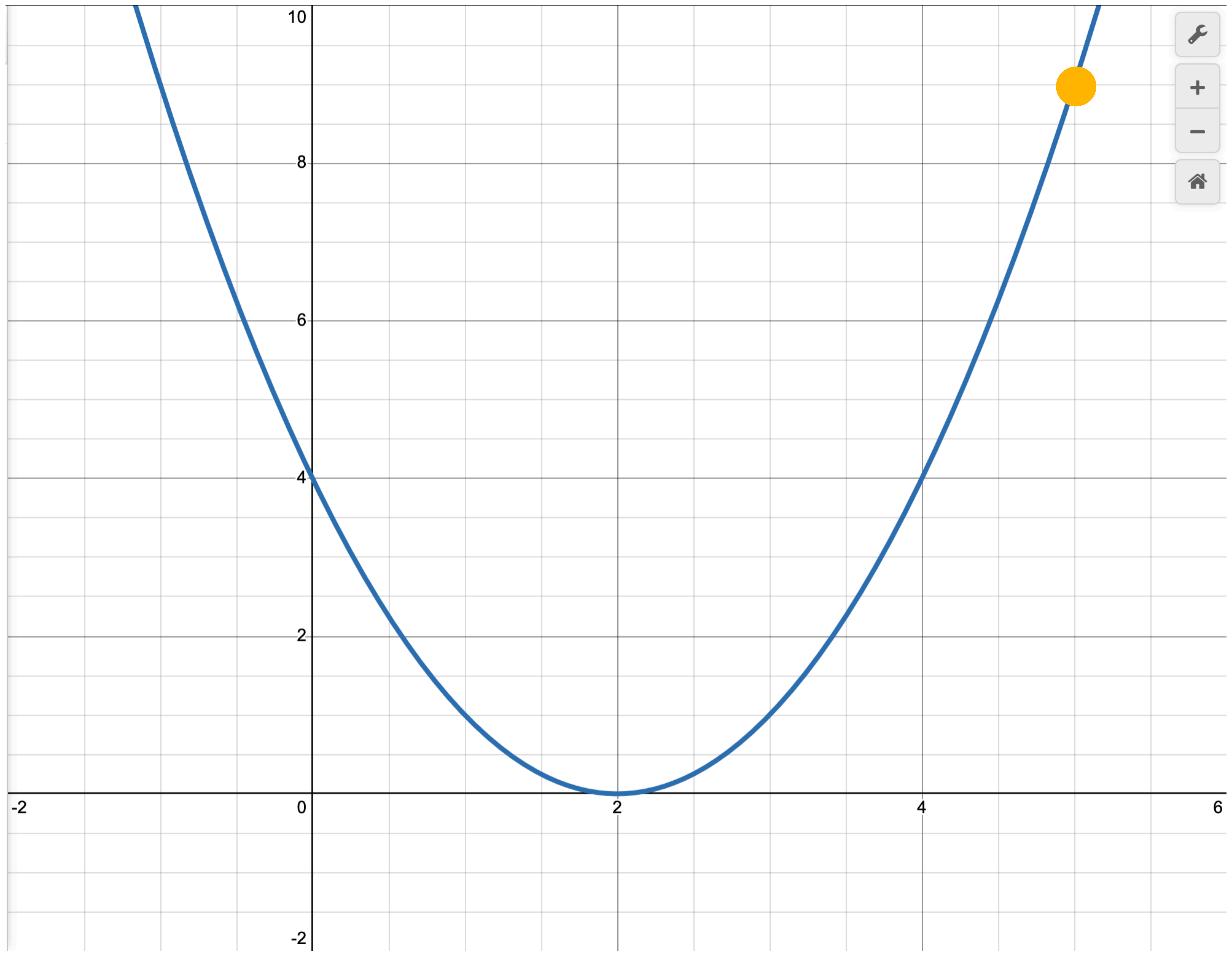
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:



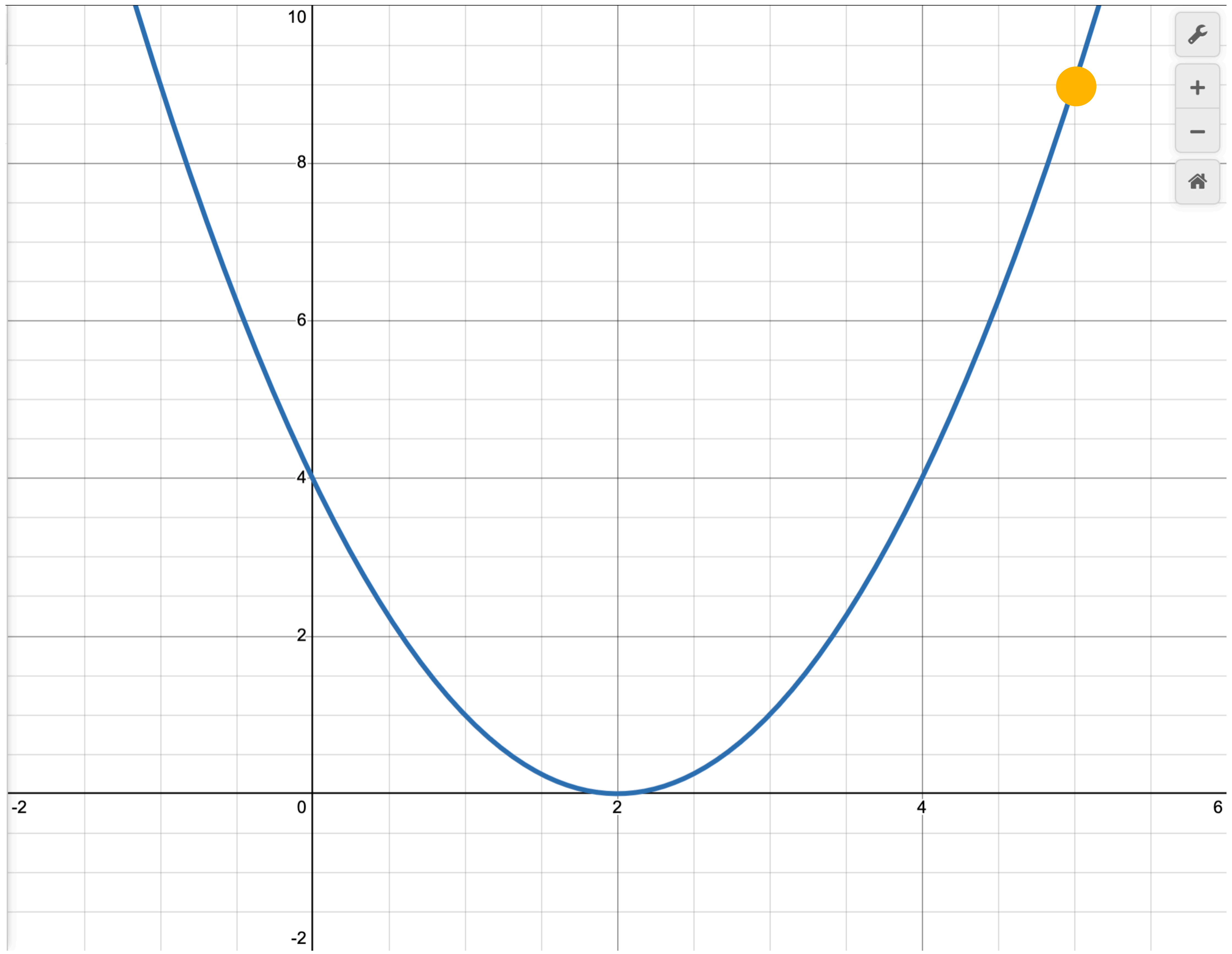
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$



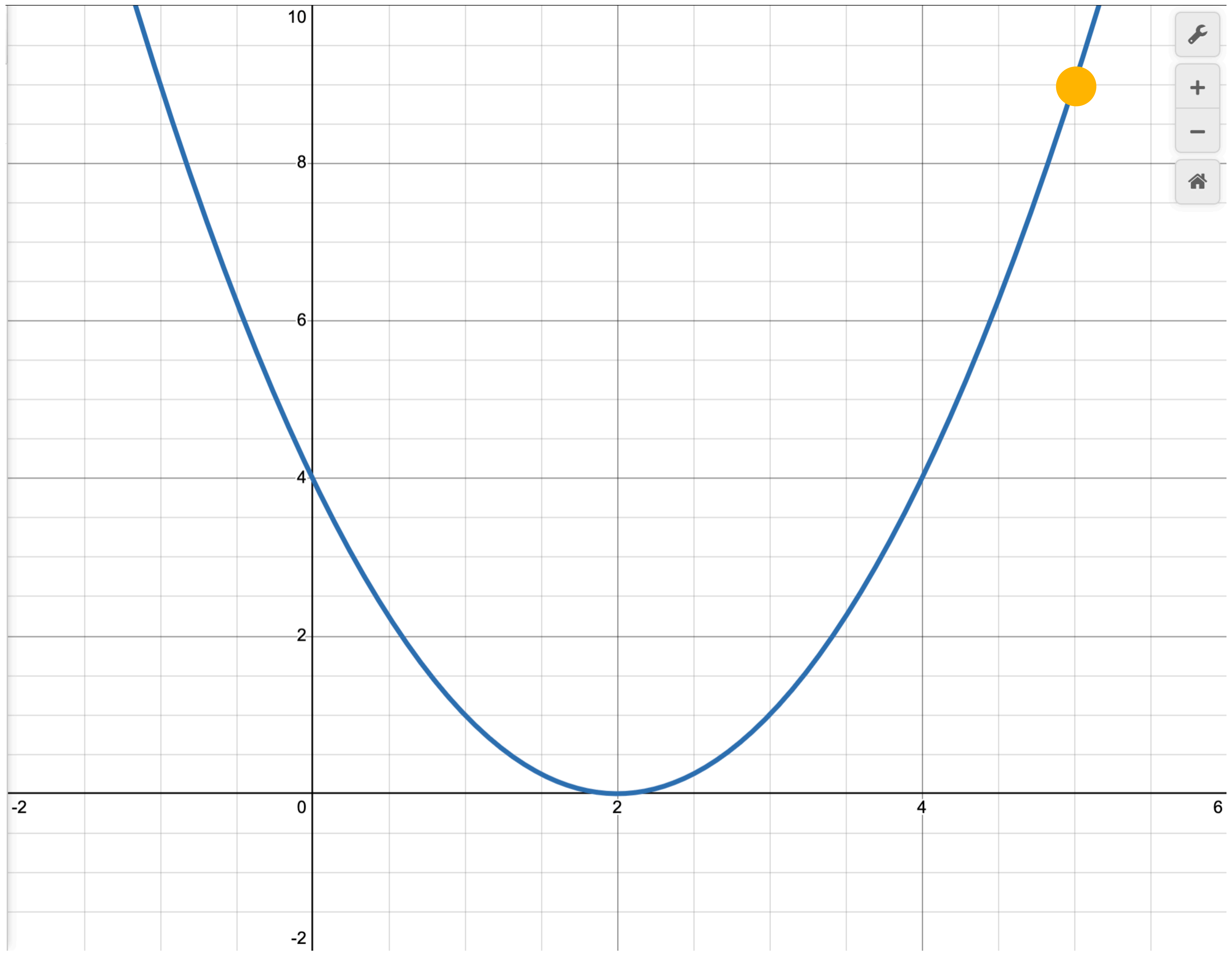
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ:



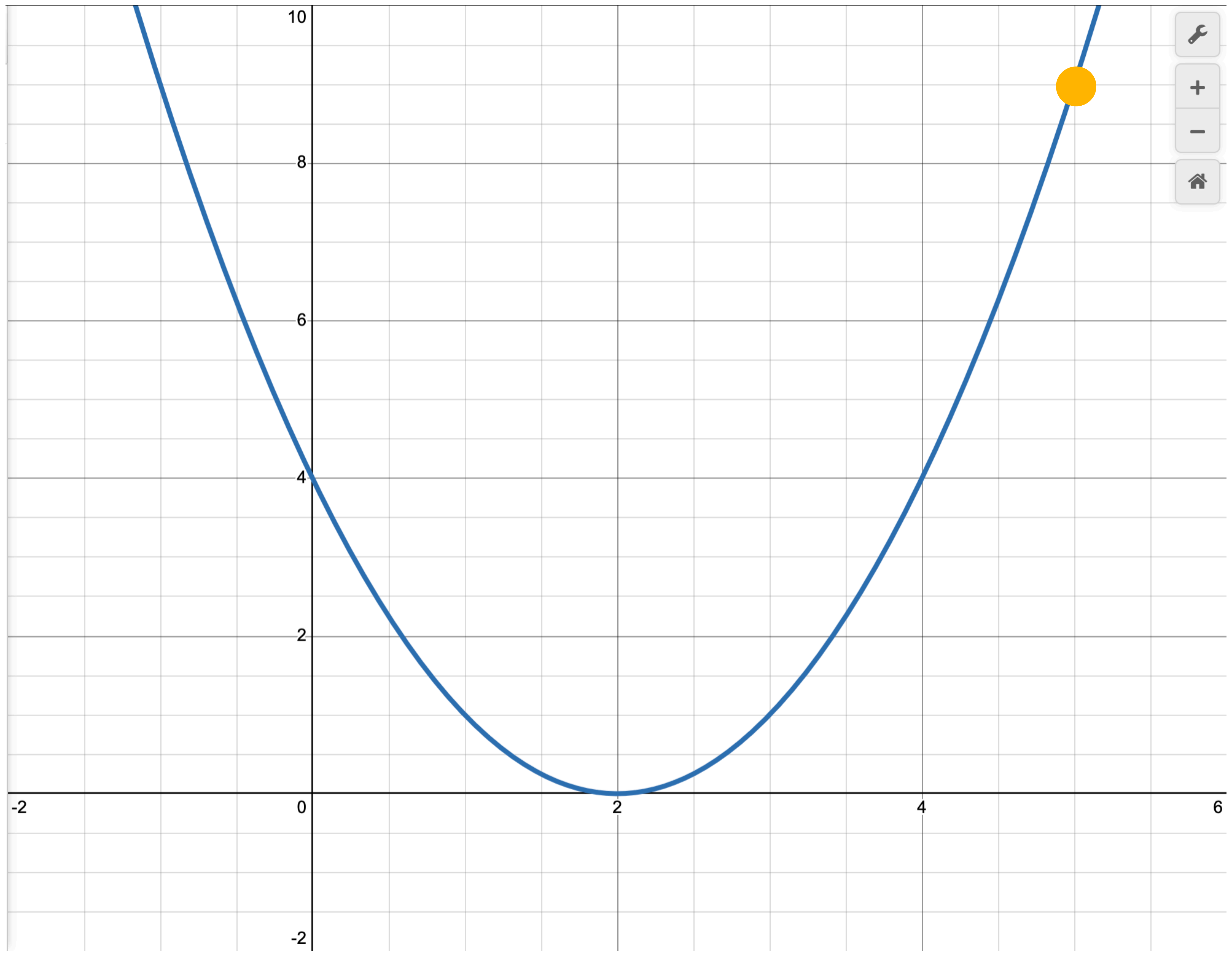
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ:
    - $2 \cdot 5 - 4 = 6$



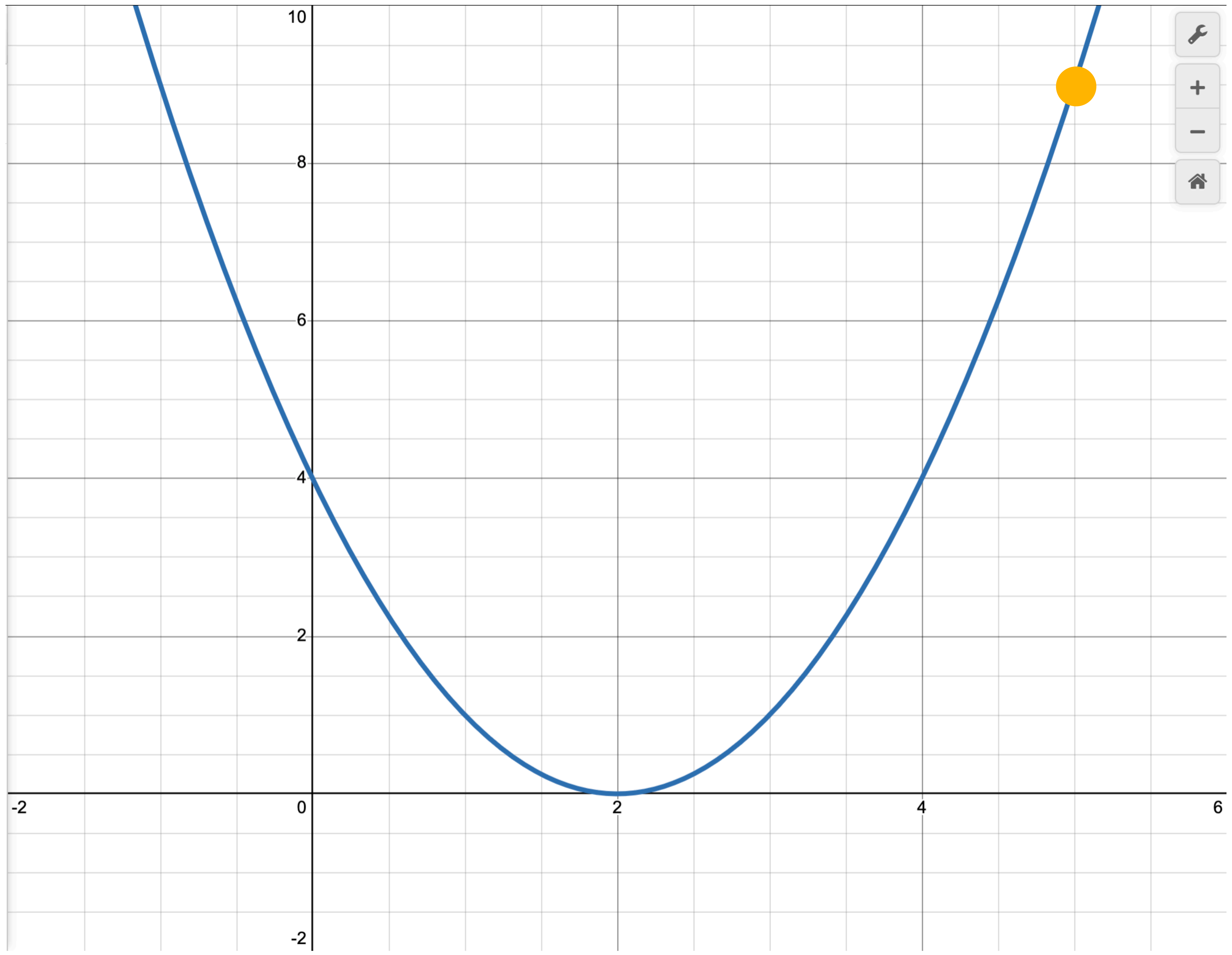
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:



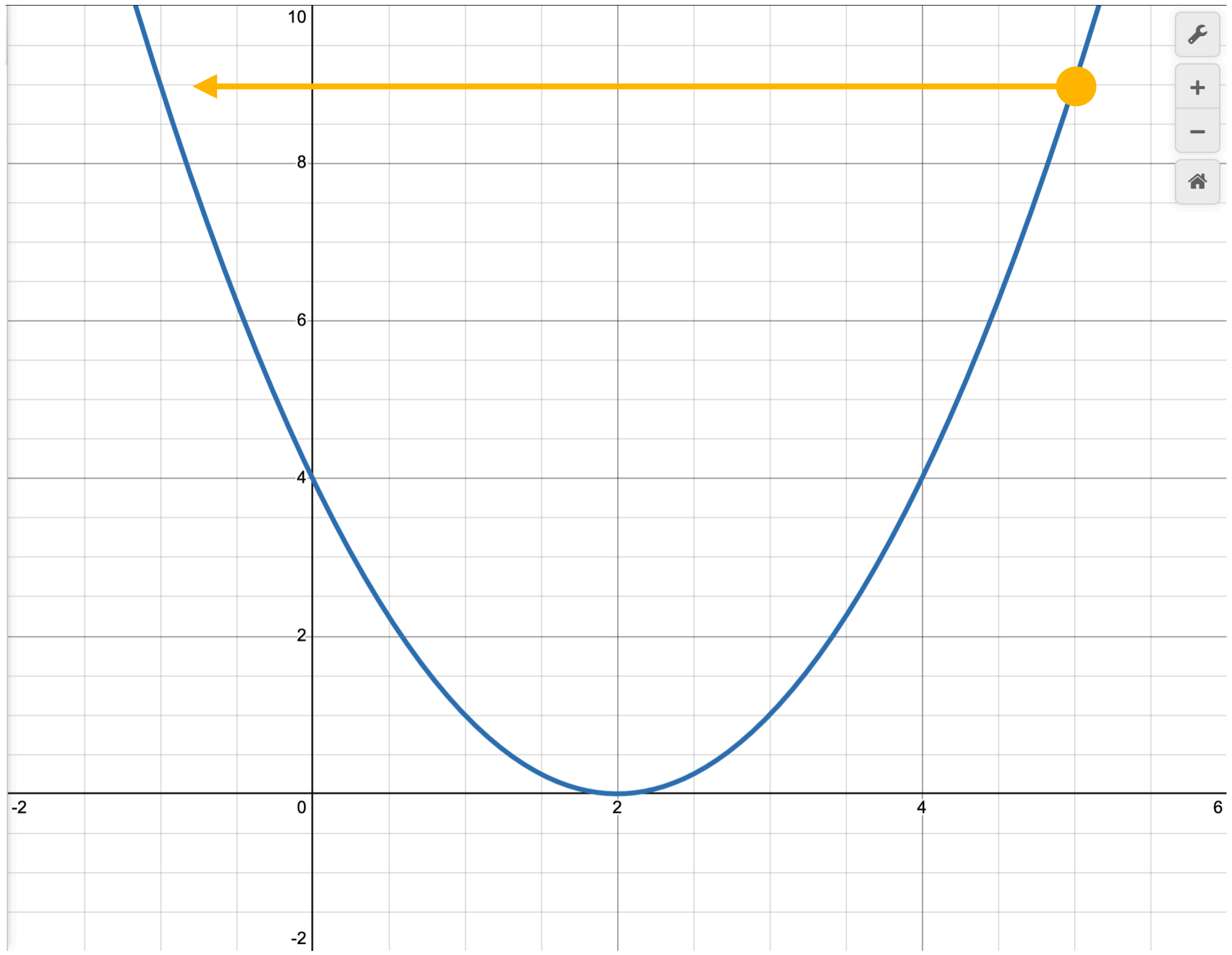
Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:
    - $\theta_1 = \theta_0 - 6 = -1$

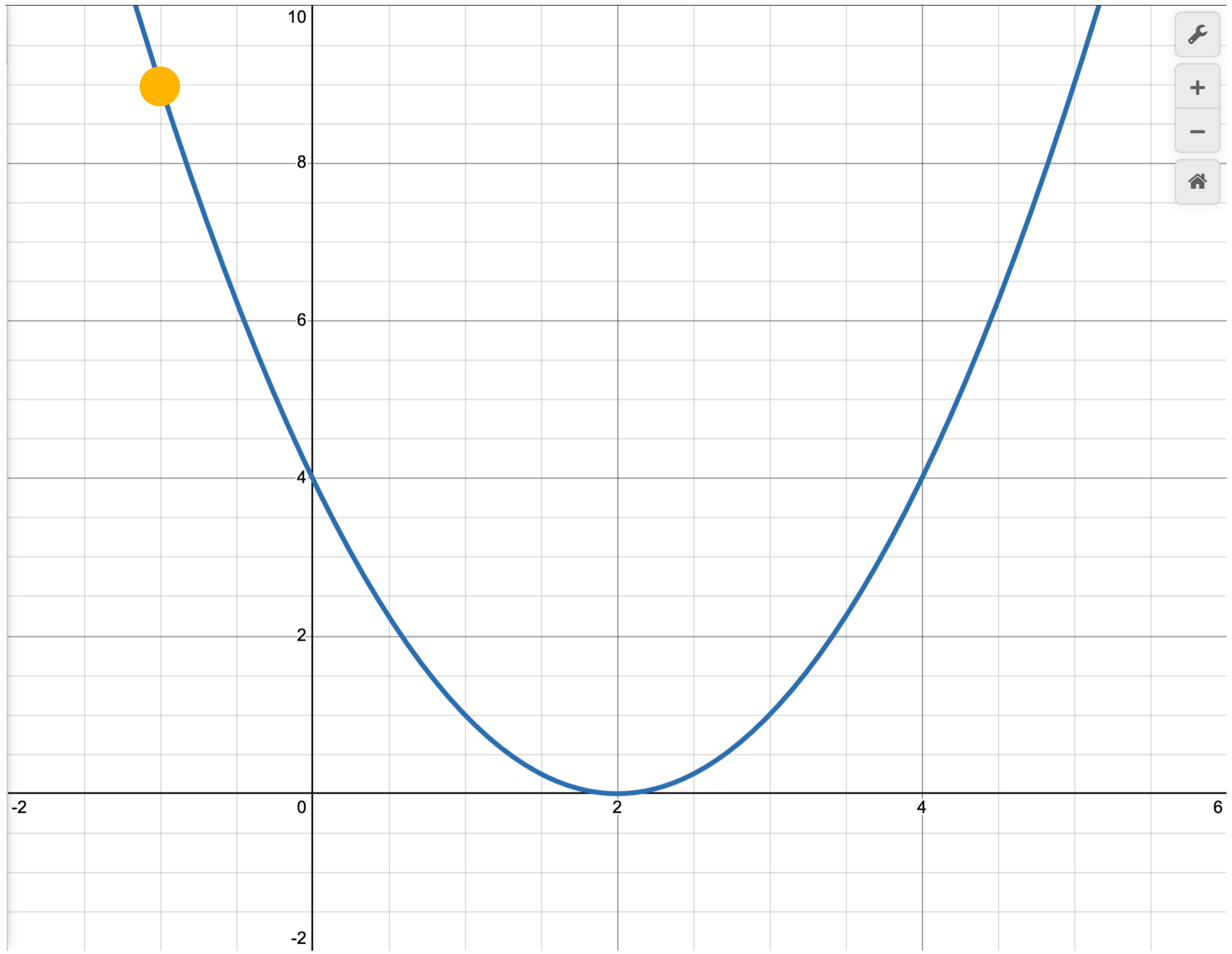


Gradient Descent (first try)

- At each step of the algorithm:
  - Calculate the **slope at the current point**
  - Adjust θ by the **negative of the slope** (because we want to **minimize** the function)
- First step:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:
    - $\theta_1 = \theta_0 - 6 = -1$

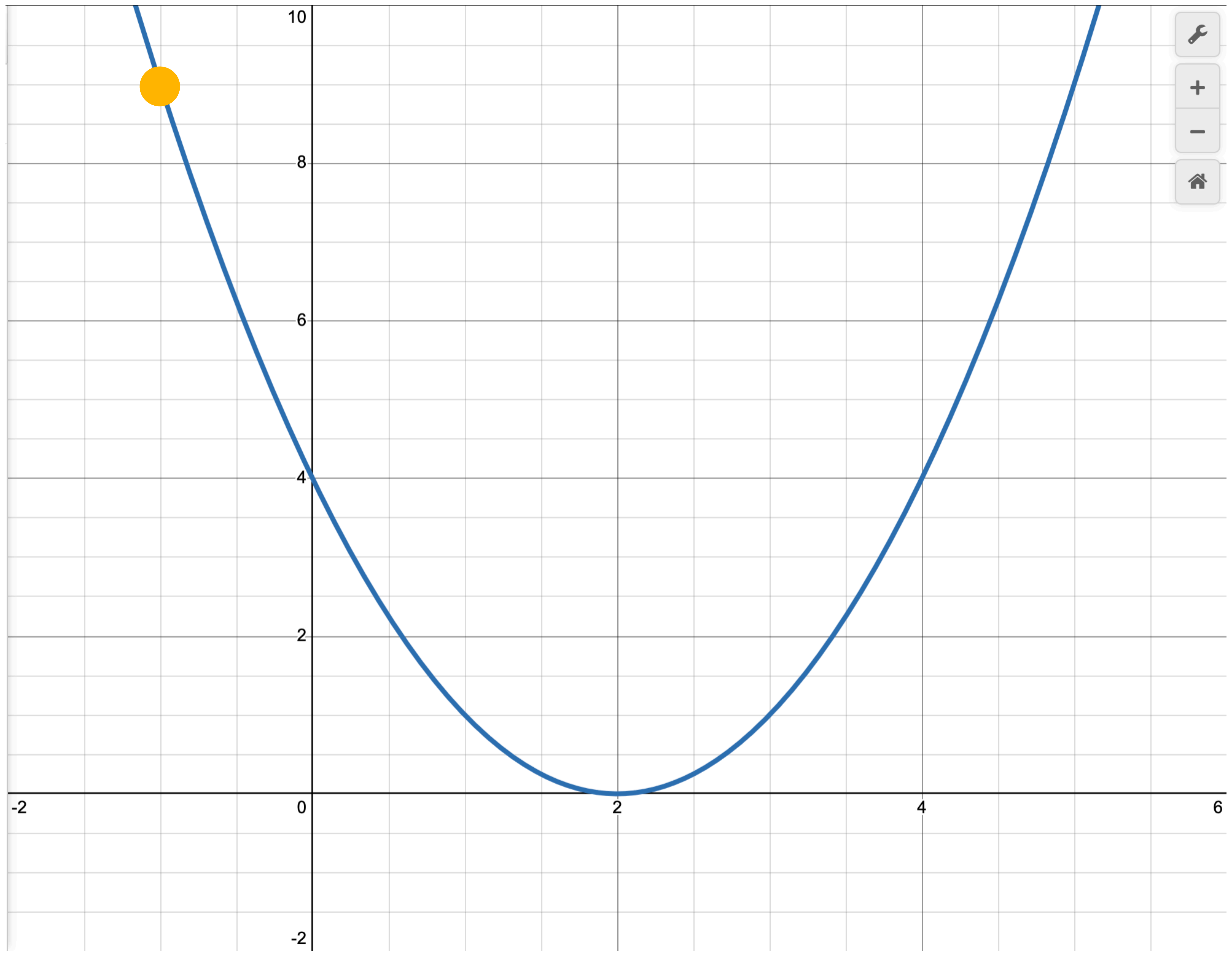


Gradient Descent (first try)



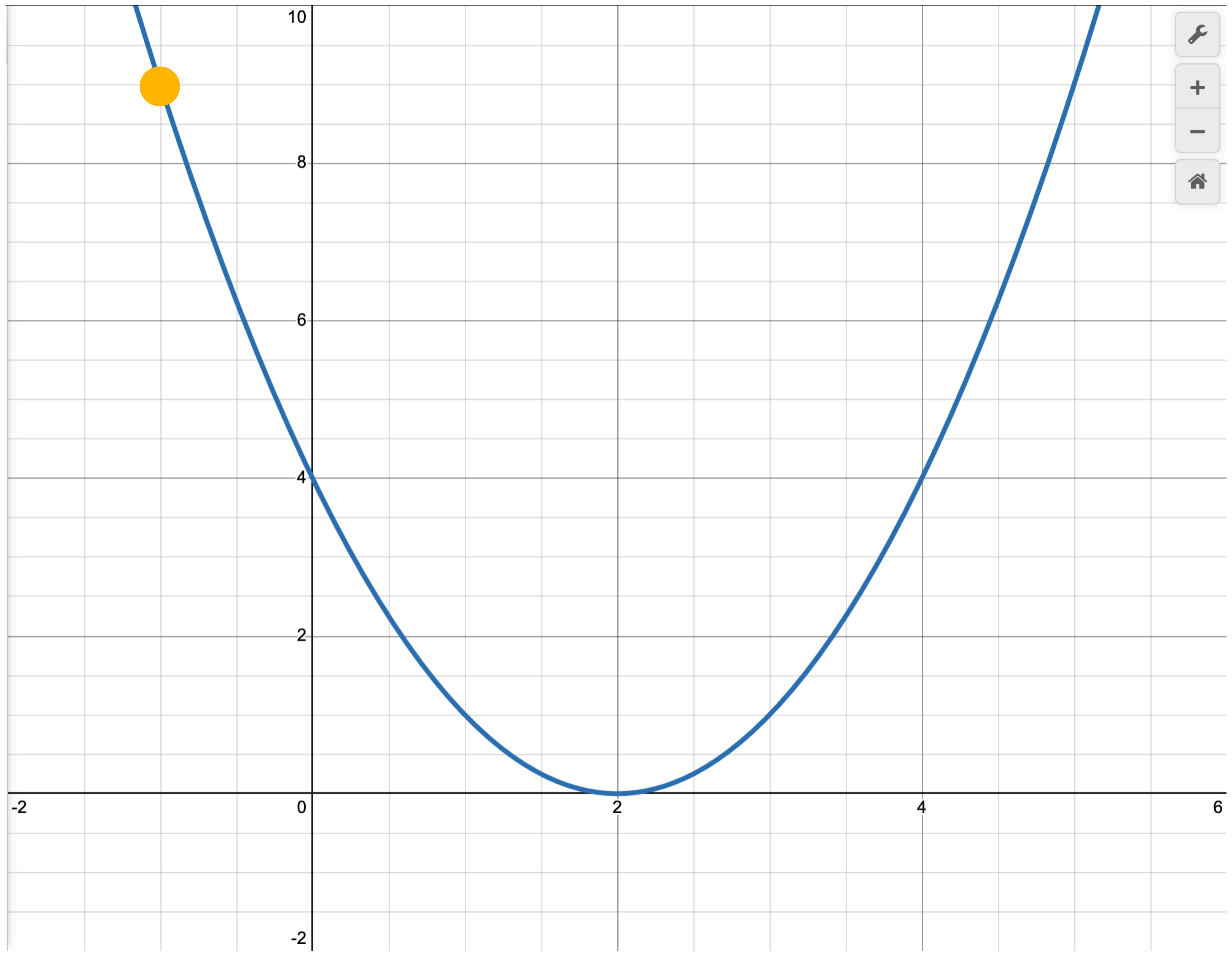
Gradient Descent (first try)

- Second step:



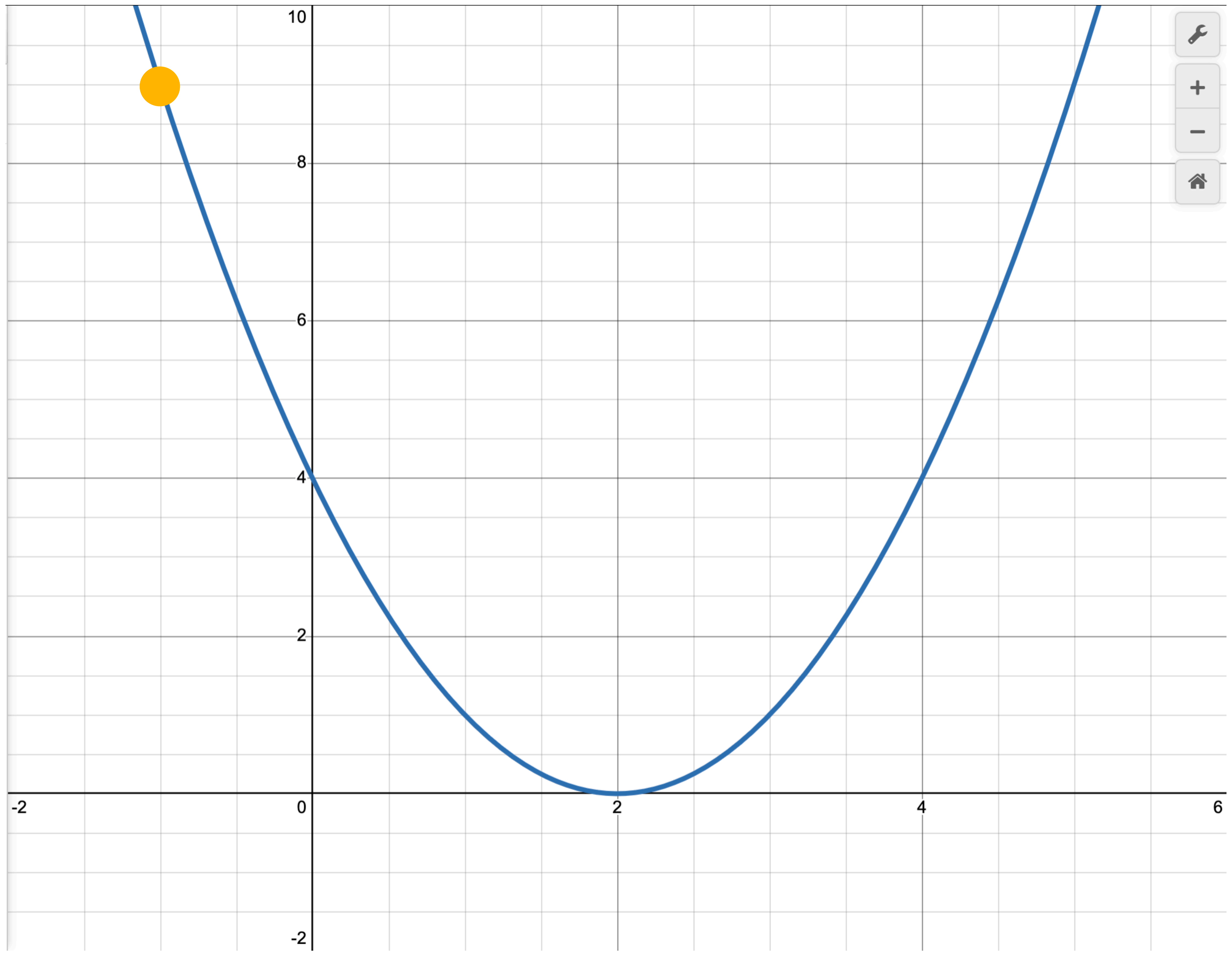
Gradient Descent (first try)

- Second step:
  - Same derivative function:



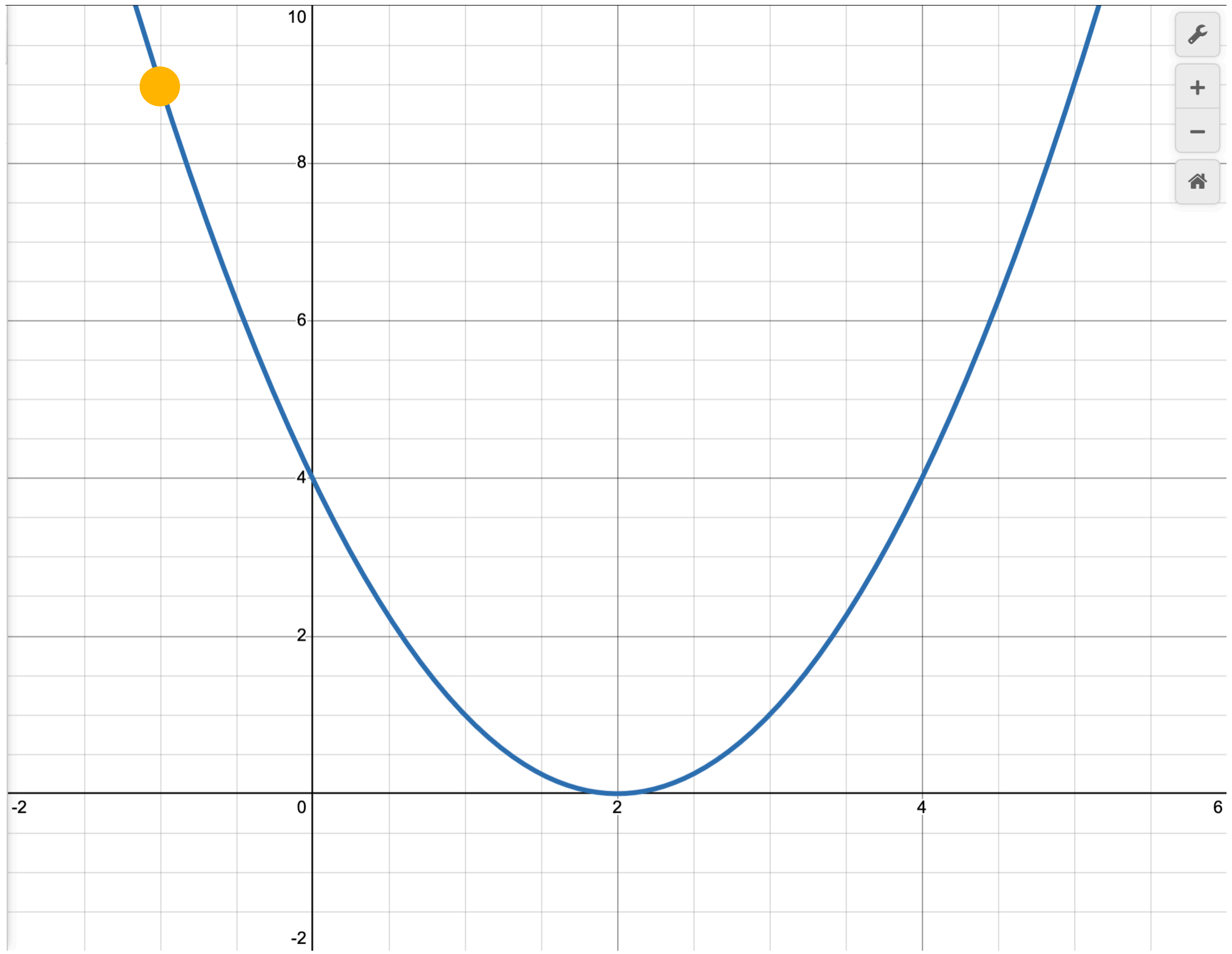
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$



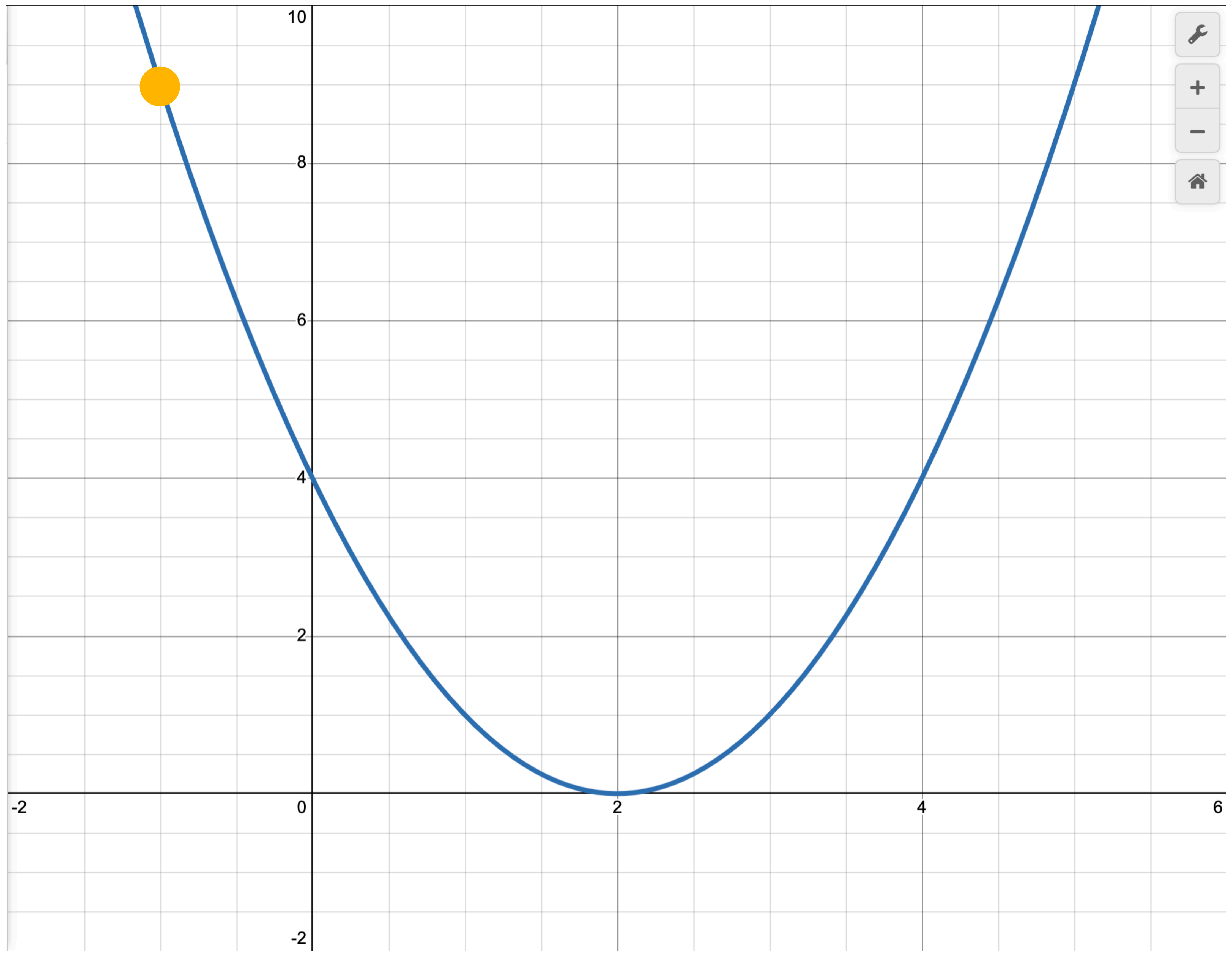
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:



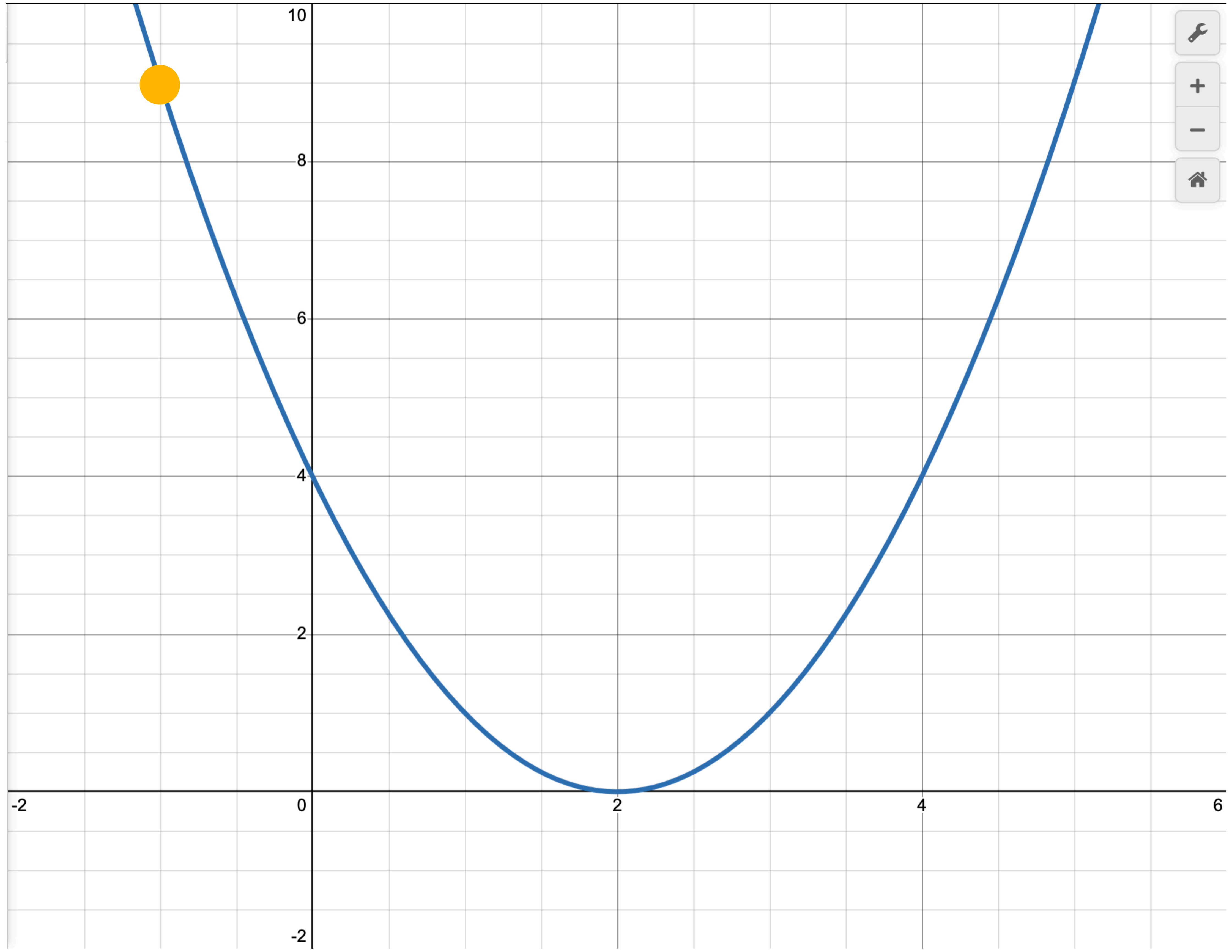
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$



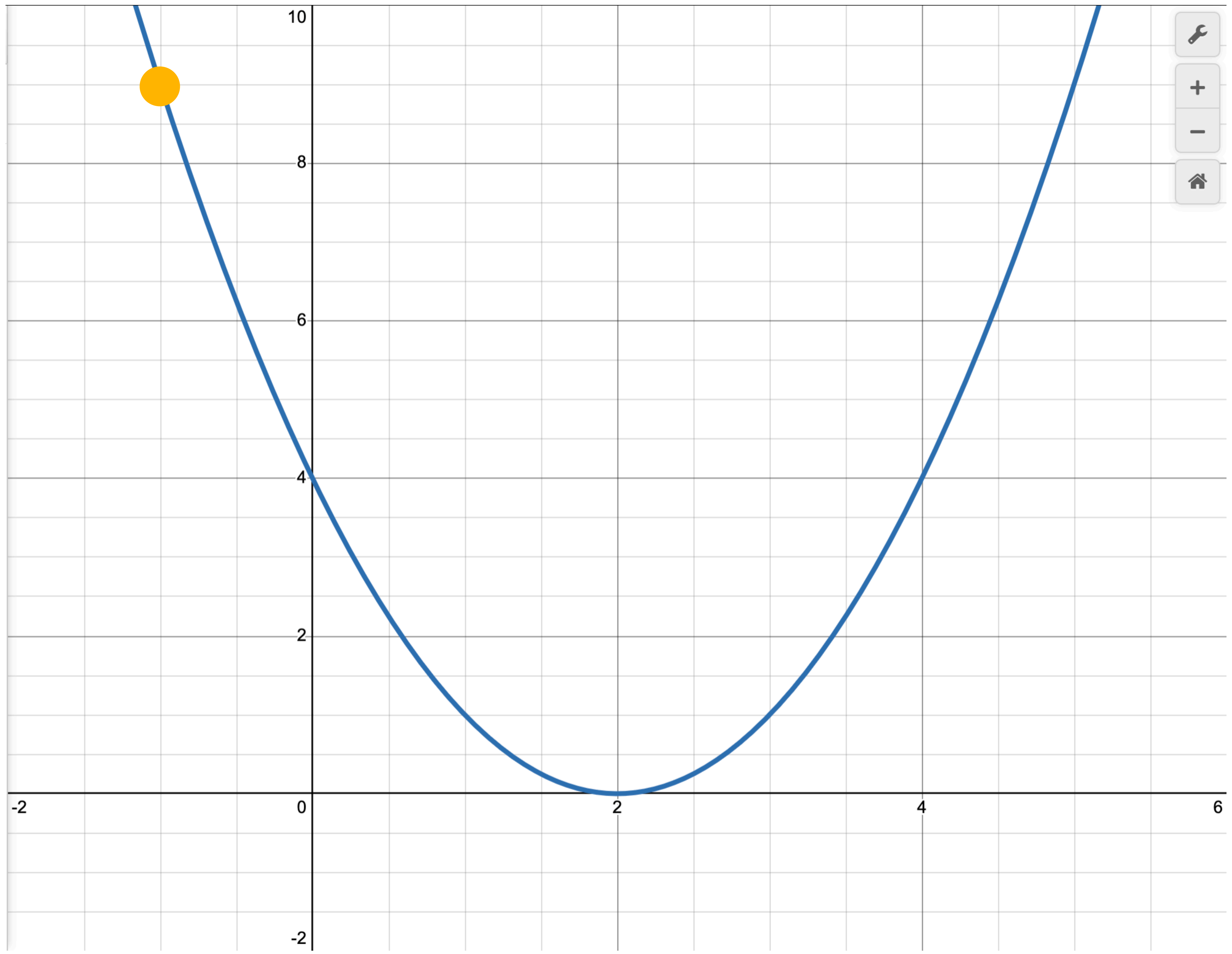
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$
  - Update θ:



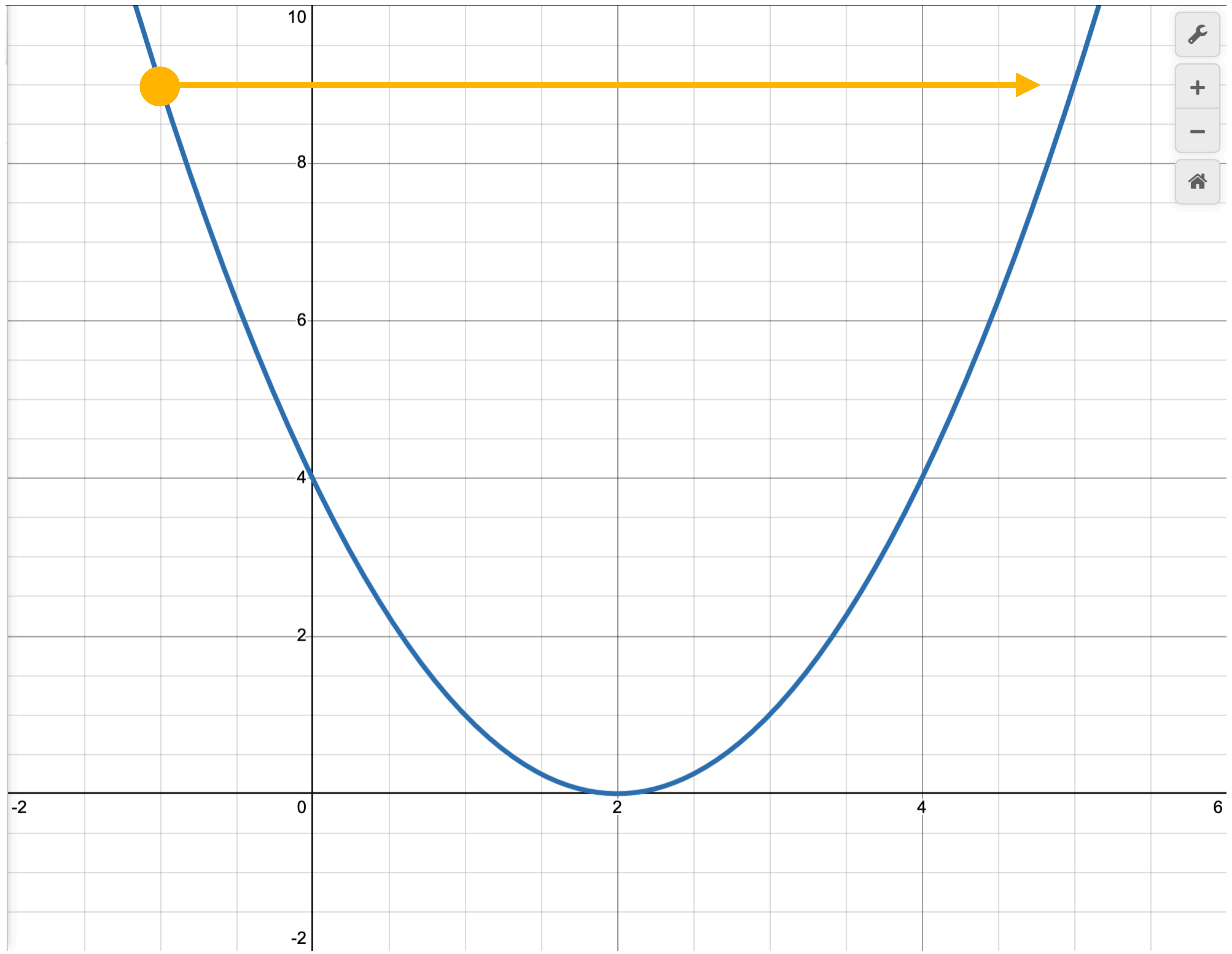
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$
  - Update θ:
    - $\theta_2 = \theta_1 - (-6) = 5$



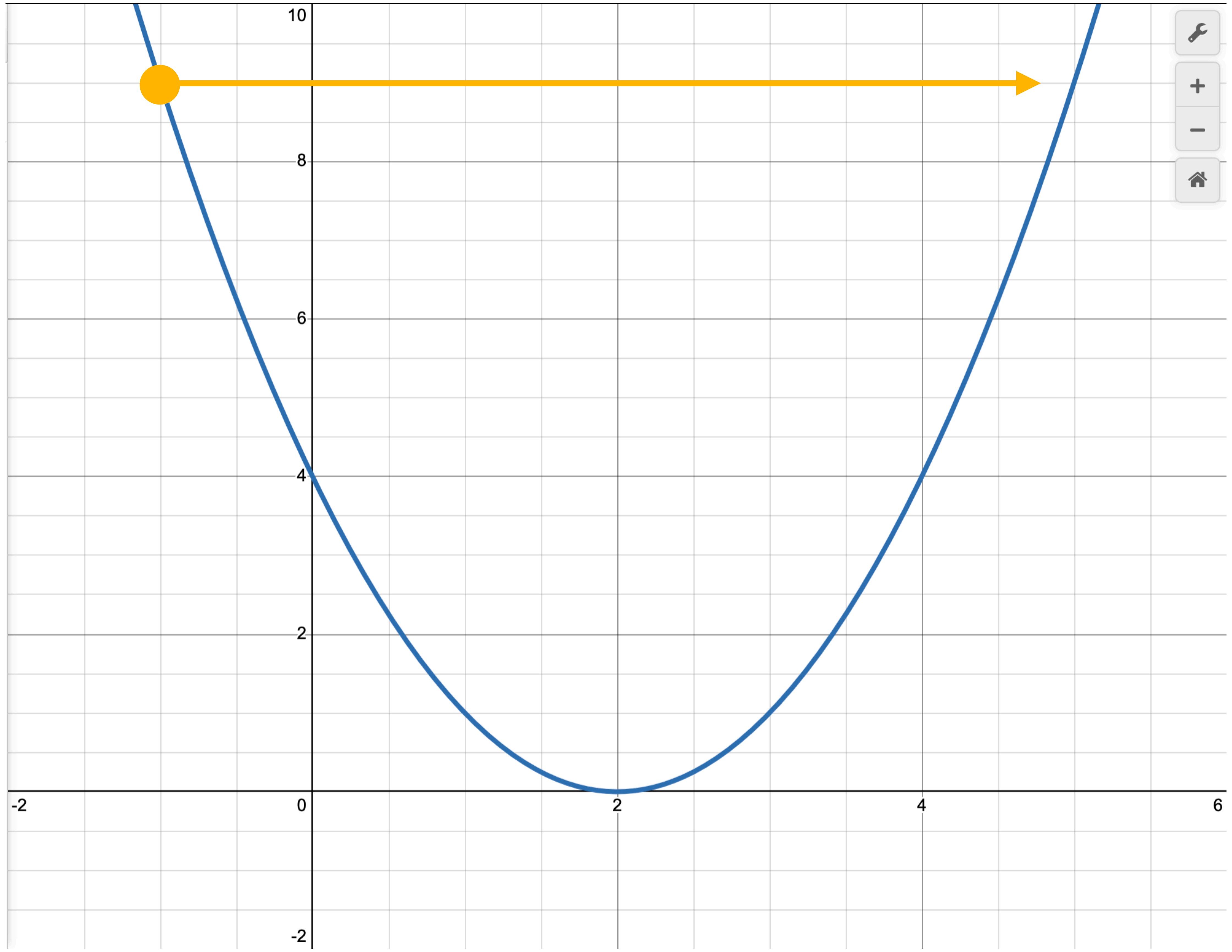
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$
  - Update θ:
    - $\theta_2 = \theta_1 - (-6) = 5$



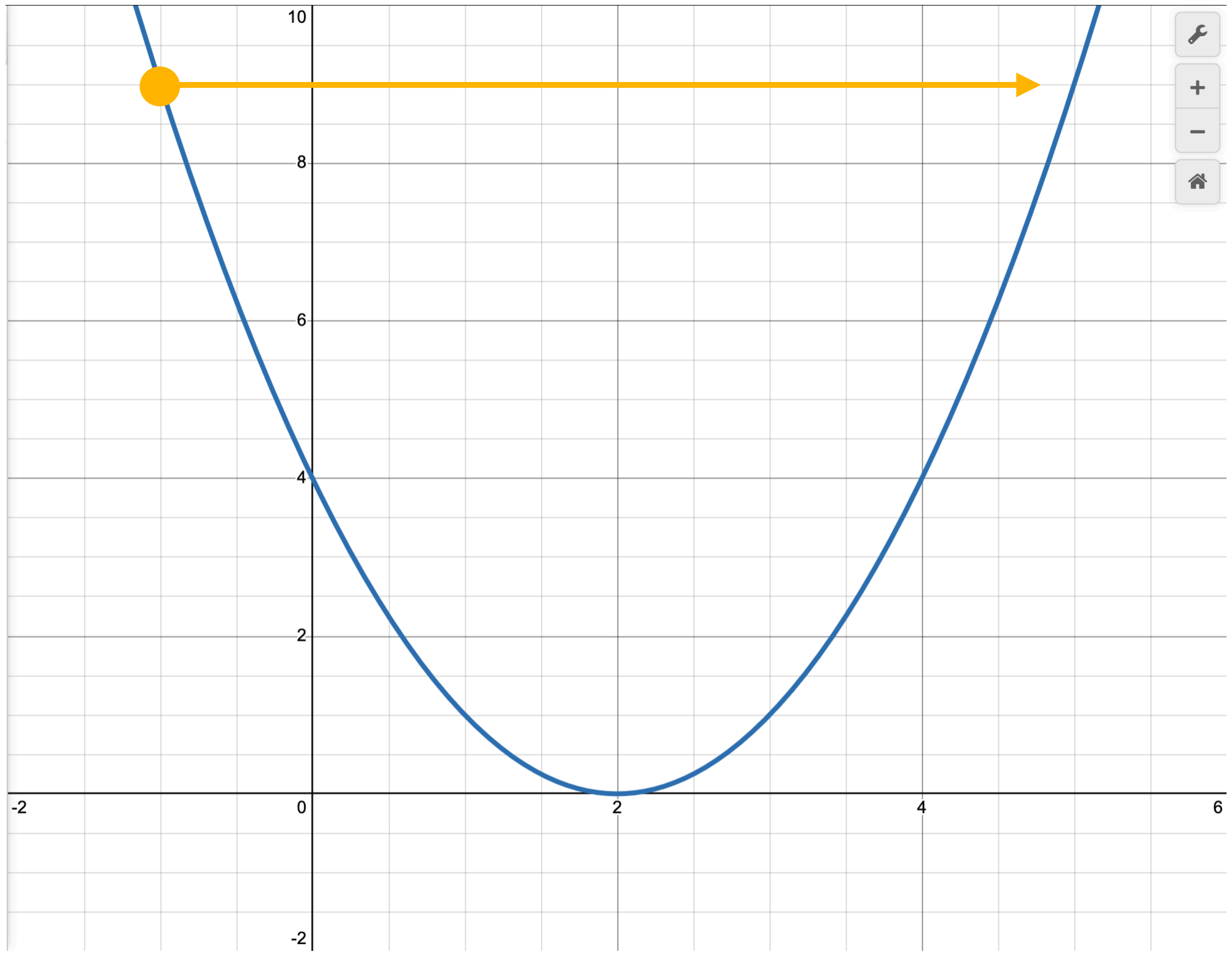
Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$
  - Update θ:
    - $\theta_2 = \theta_1 - (-6) = 5$
- Whoops! We're **back where we started!**

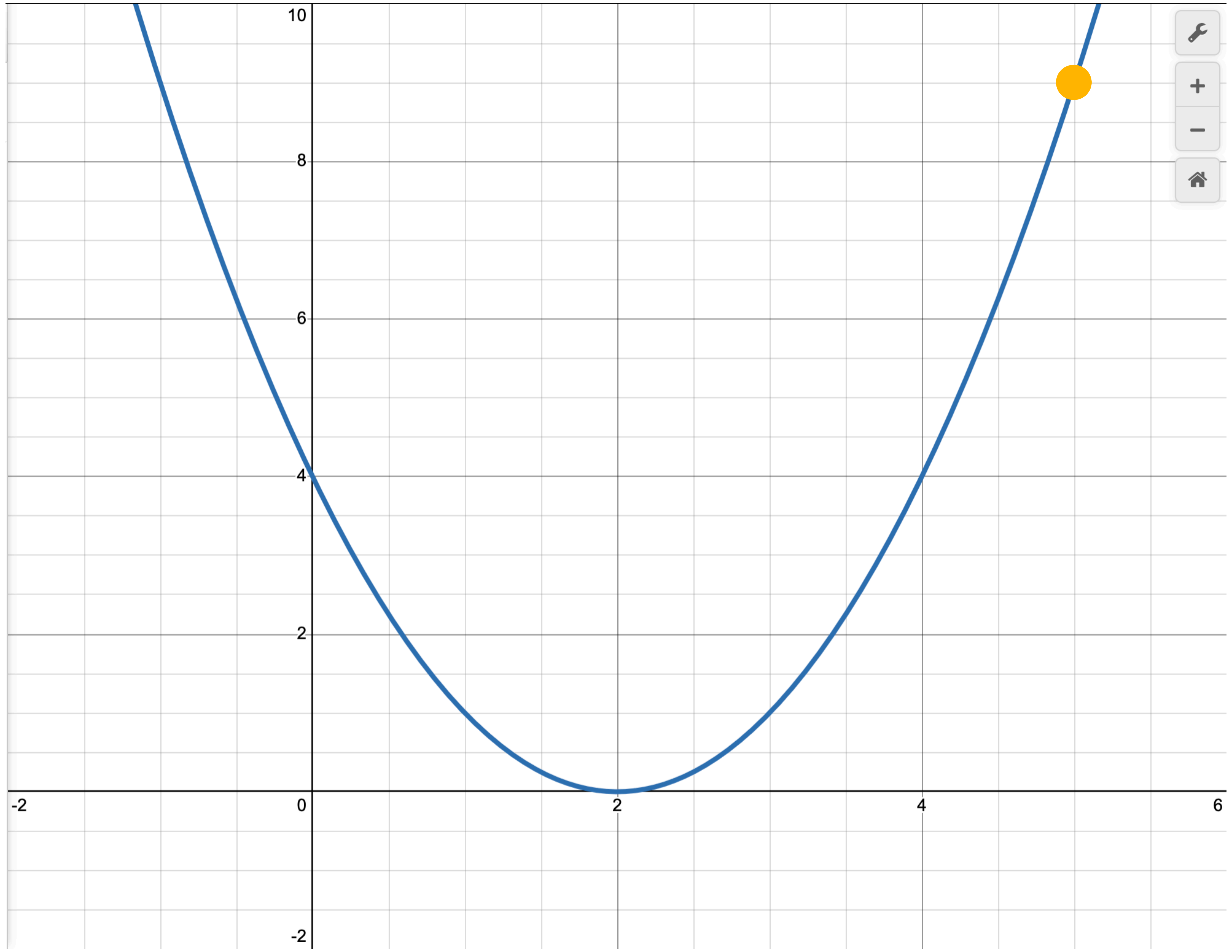


Gradient Descent (first try)

- Second step:
  - Same derivative function:
    - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in new θ_1:
    - $2 \cdot (-1) - 4 = -6$
  - Update θ:
    - $\theta_2 = \theta_1 - (-6) = 5$
- Whoops! We're **back where we started!**
  - This process would **"bounce back and forth"** forever!

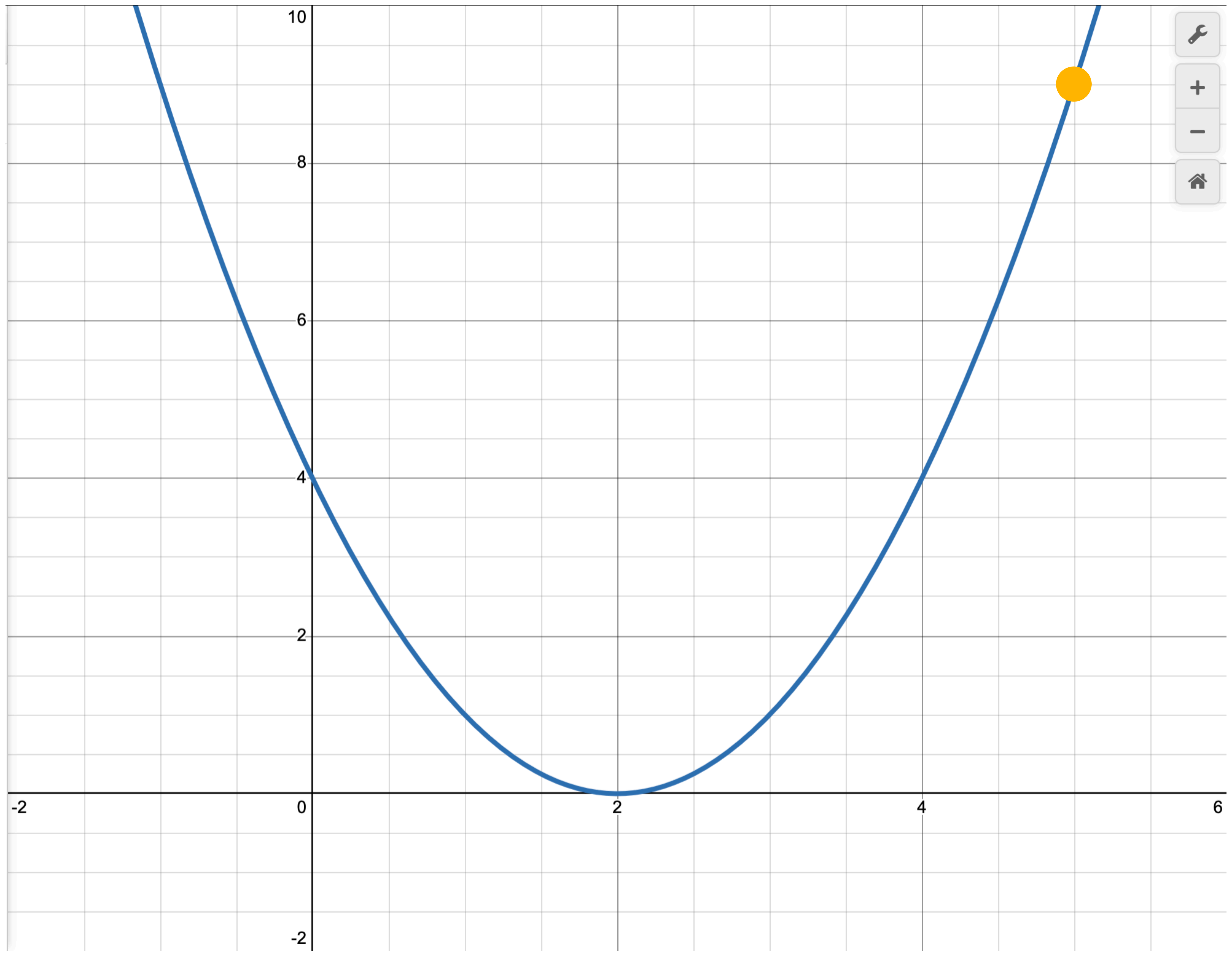


Gradient Descent (second try)



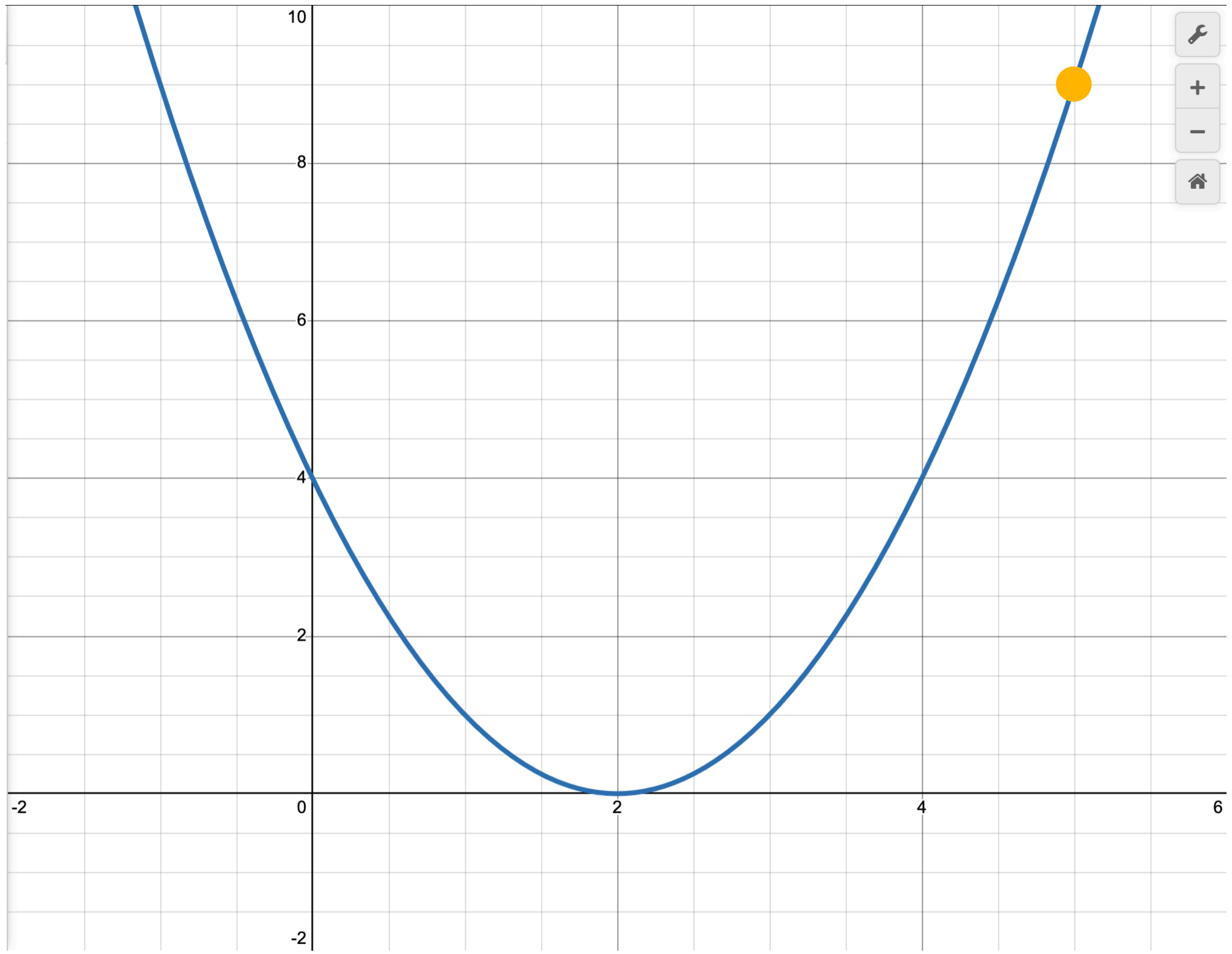
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)



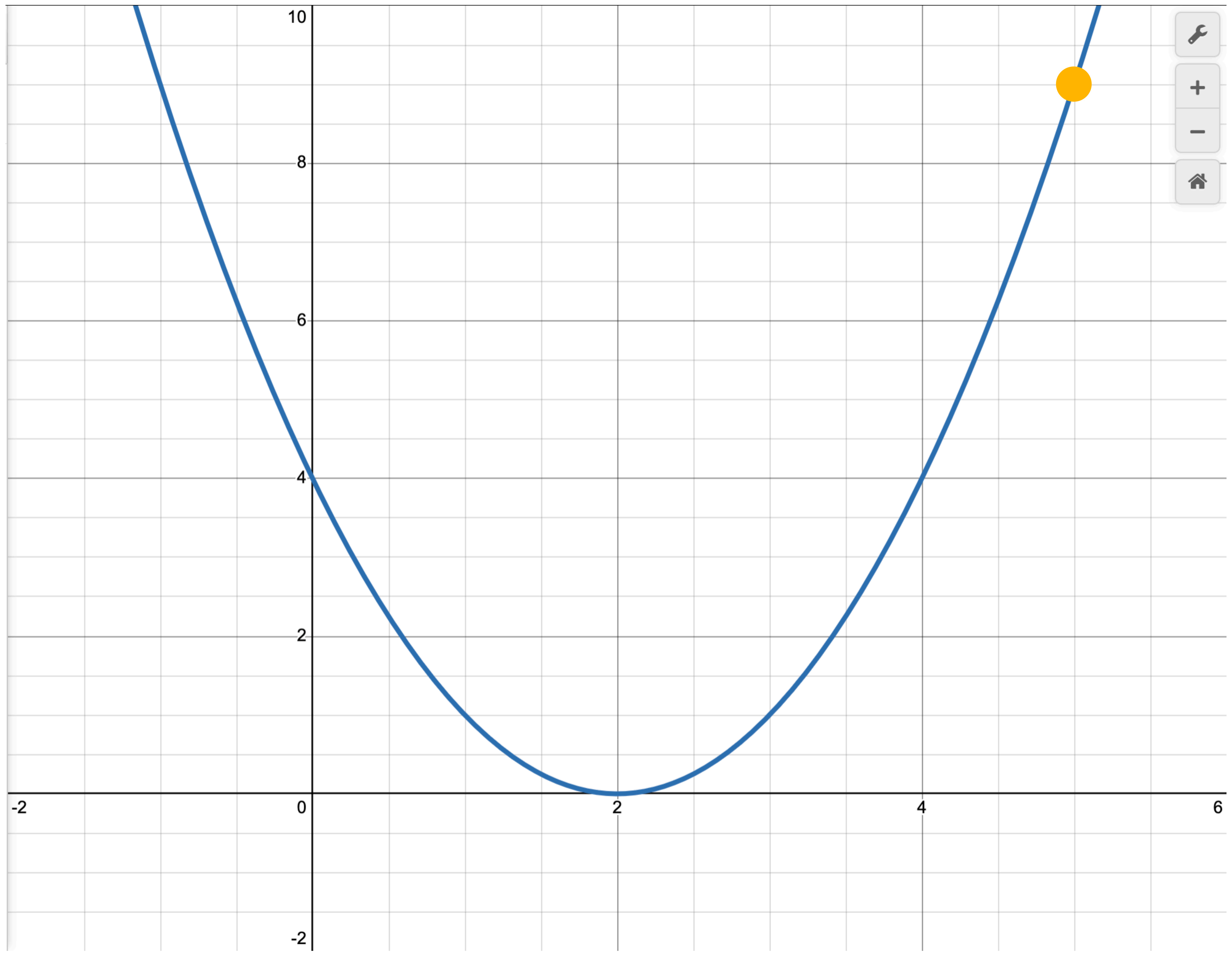
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:



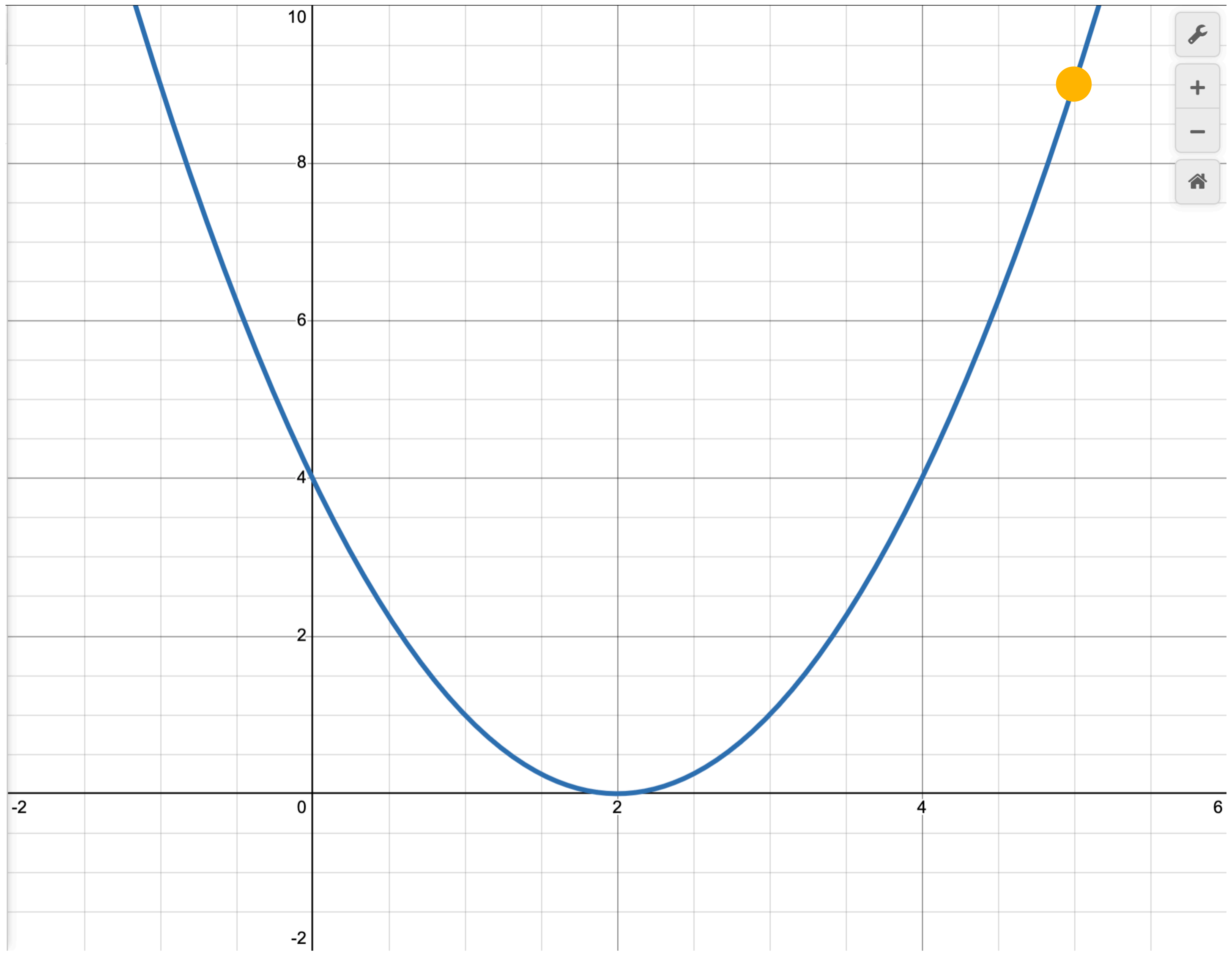
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$



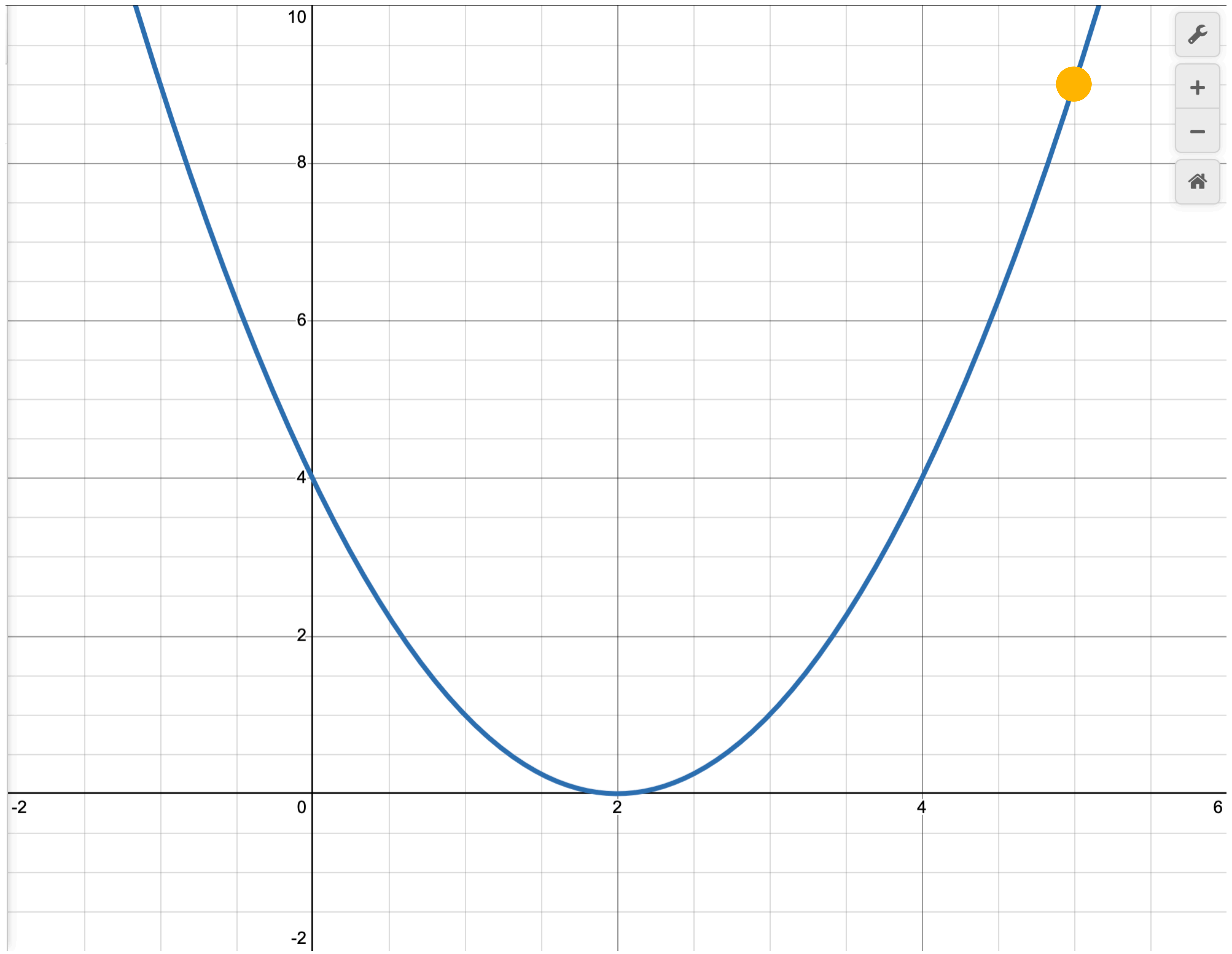
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ_0 to derivative:



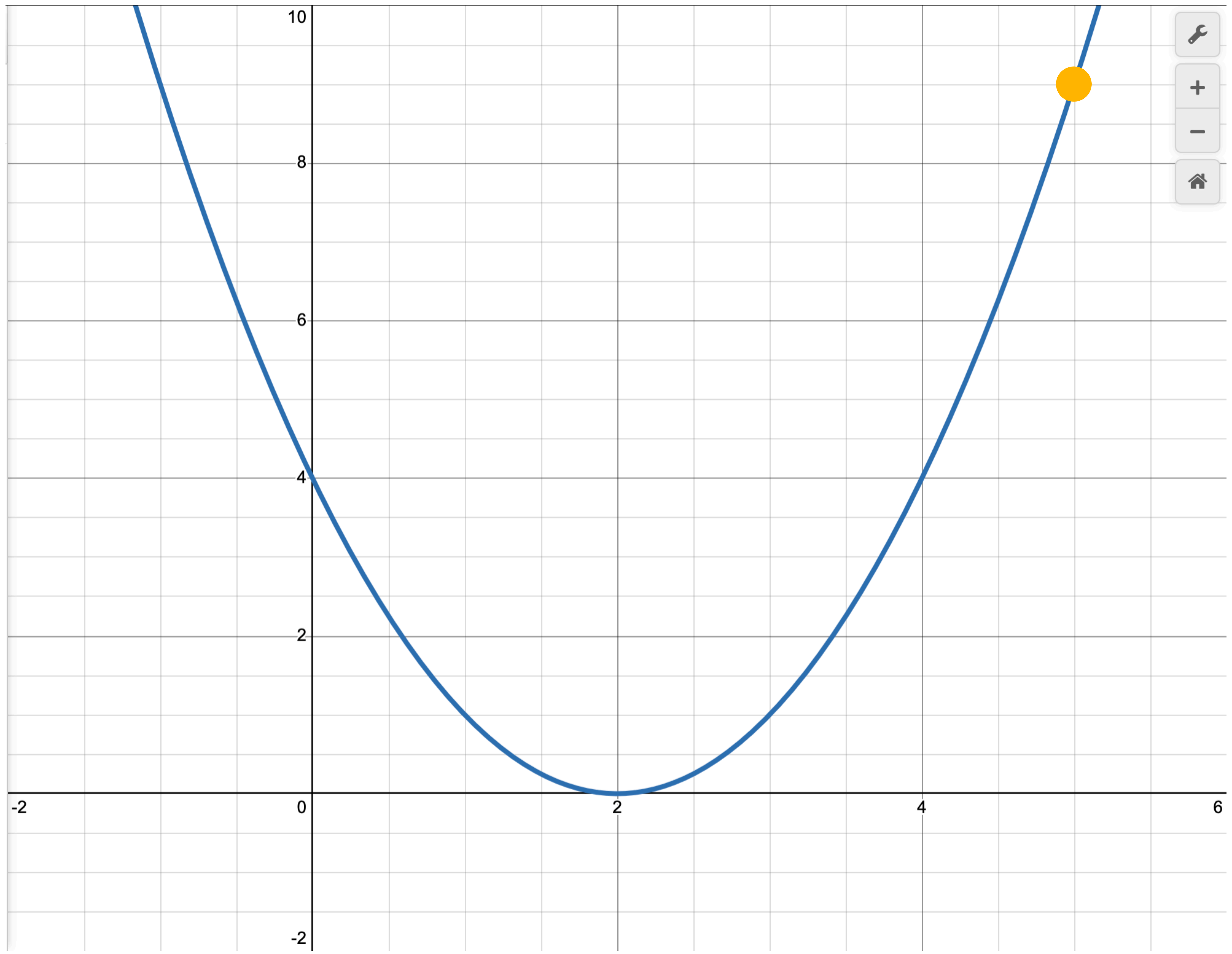
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ_0 to derivative:
    - $2 \cdot 5 - 4 = 6$



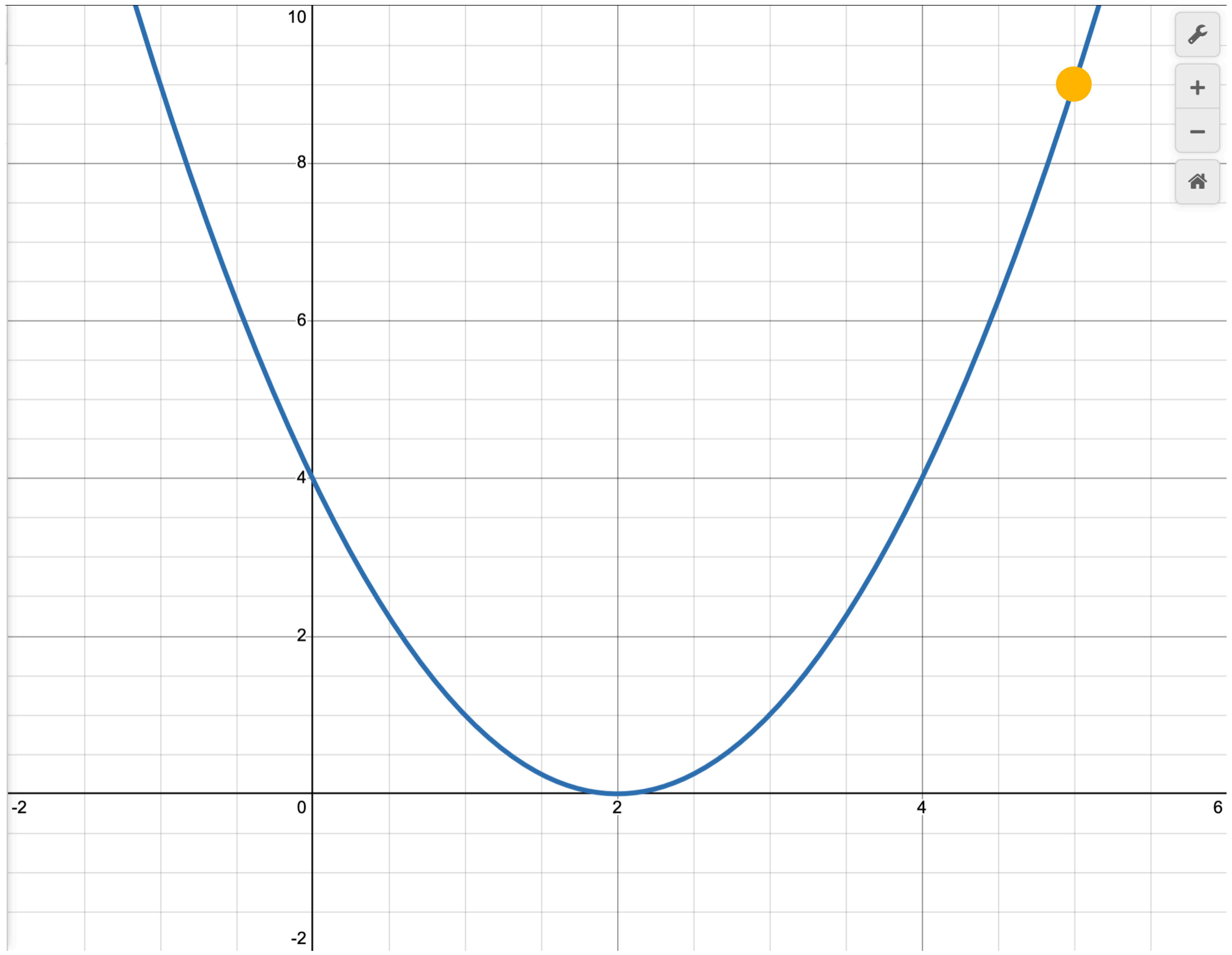
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ_0 to derivative:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:



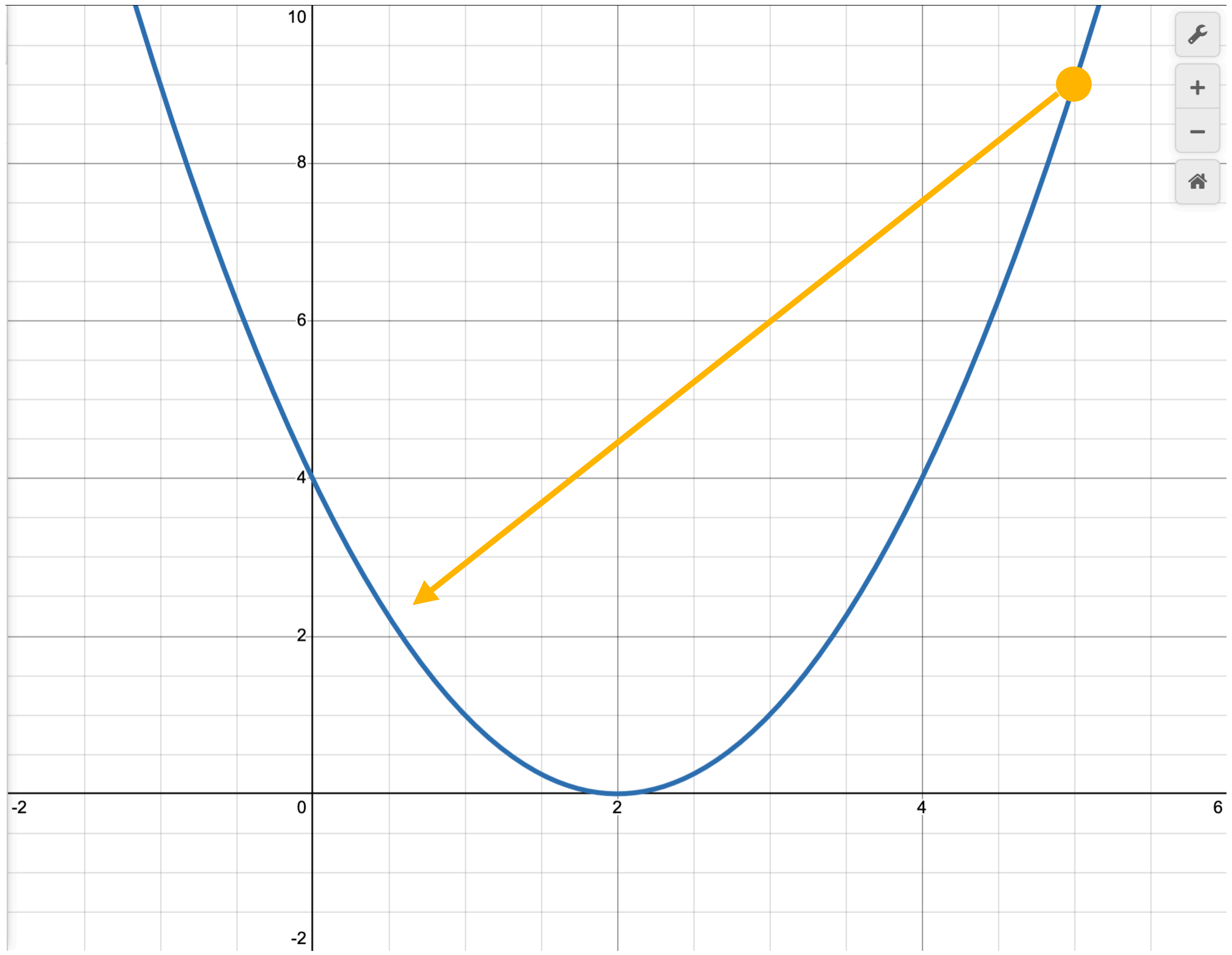
Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\dfrac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ_0 to derivative:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:
    - $\theta_1 = \theta_0 - 0.75 \cdot 6 = 0.5$

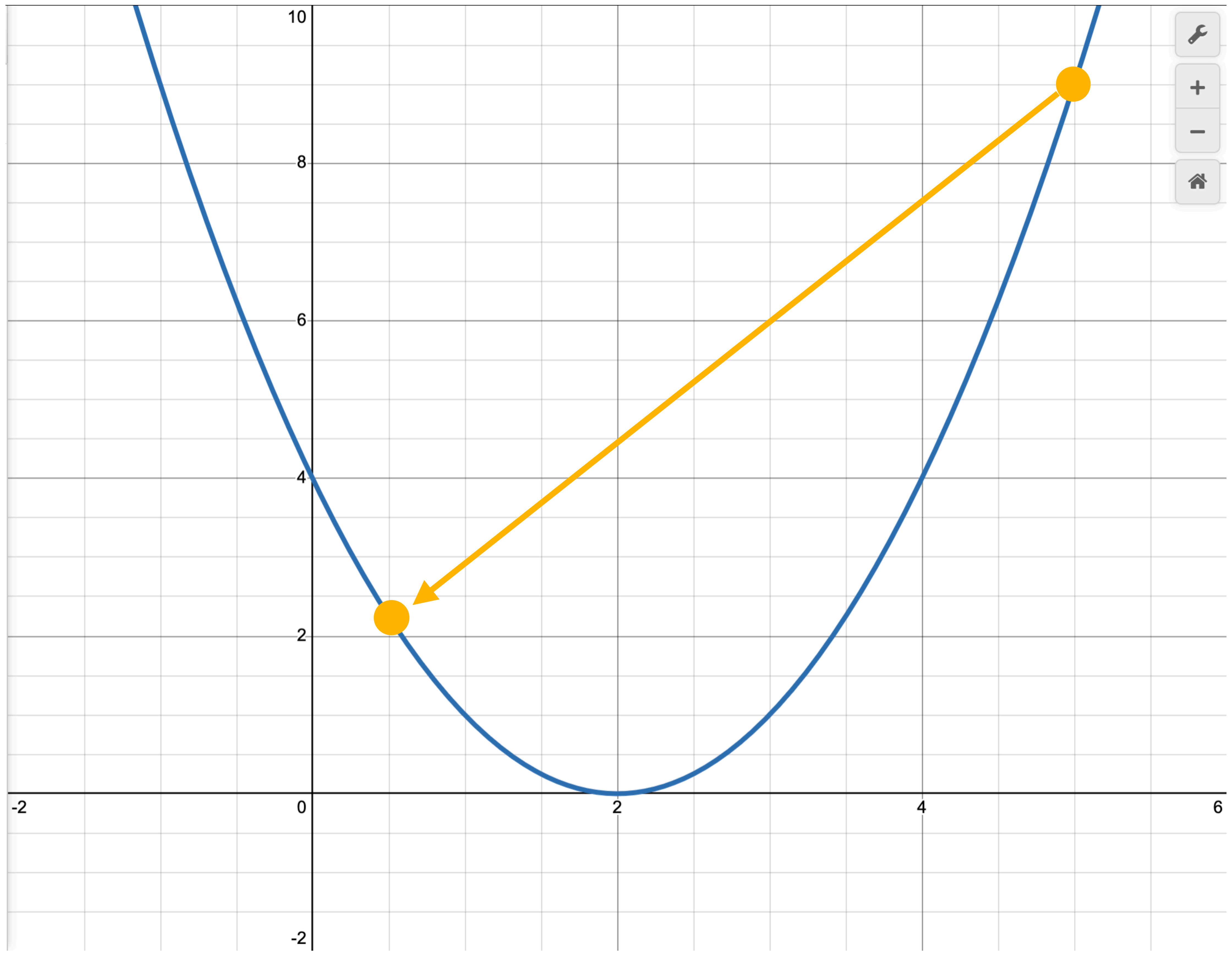


Gradient Descent (second try)

- What if we adjusted θ by a **fraction** of the slope? (Say, 0.75)
- Let's start again:
  - $\frac{d}{d\theta}(\theta - 2)^2 = 2\theta - 4$
  - Plug in θ_0 to derivative:
    - $2 \cdot 5 - 4 = 6$
  - Update θ:
    - $\theta_1 = \theta_0 - 0.75 \cdot 6 = 0.5$

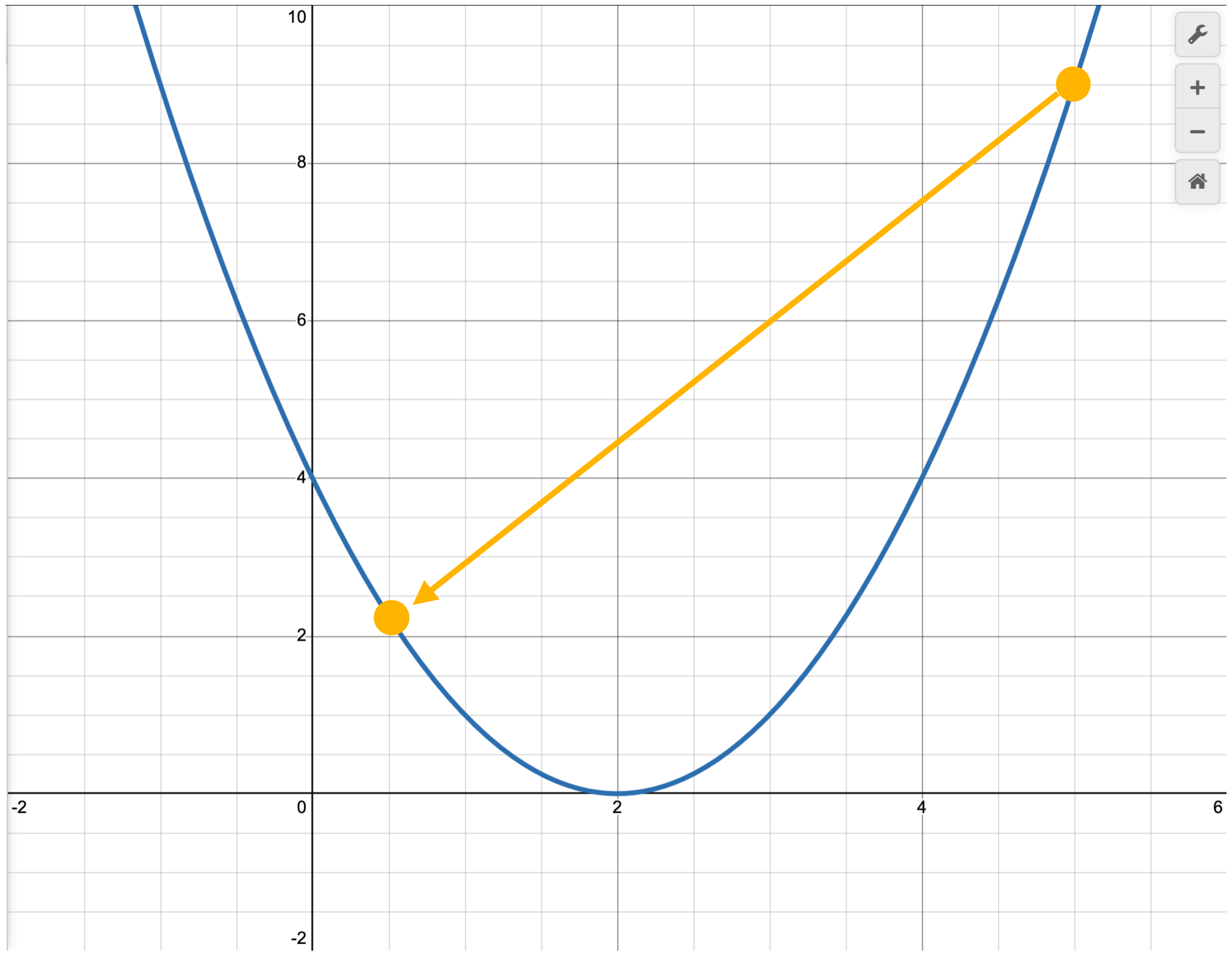


Gradient Descent (second try)



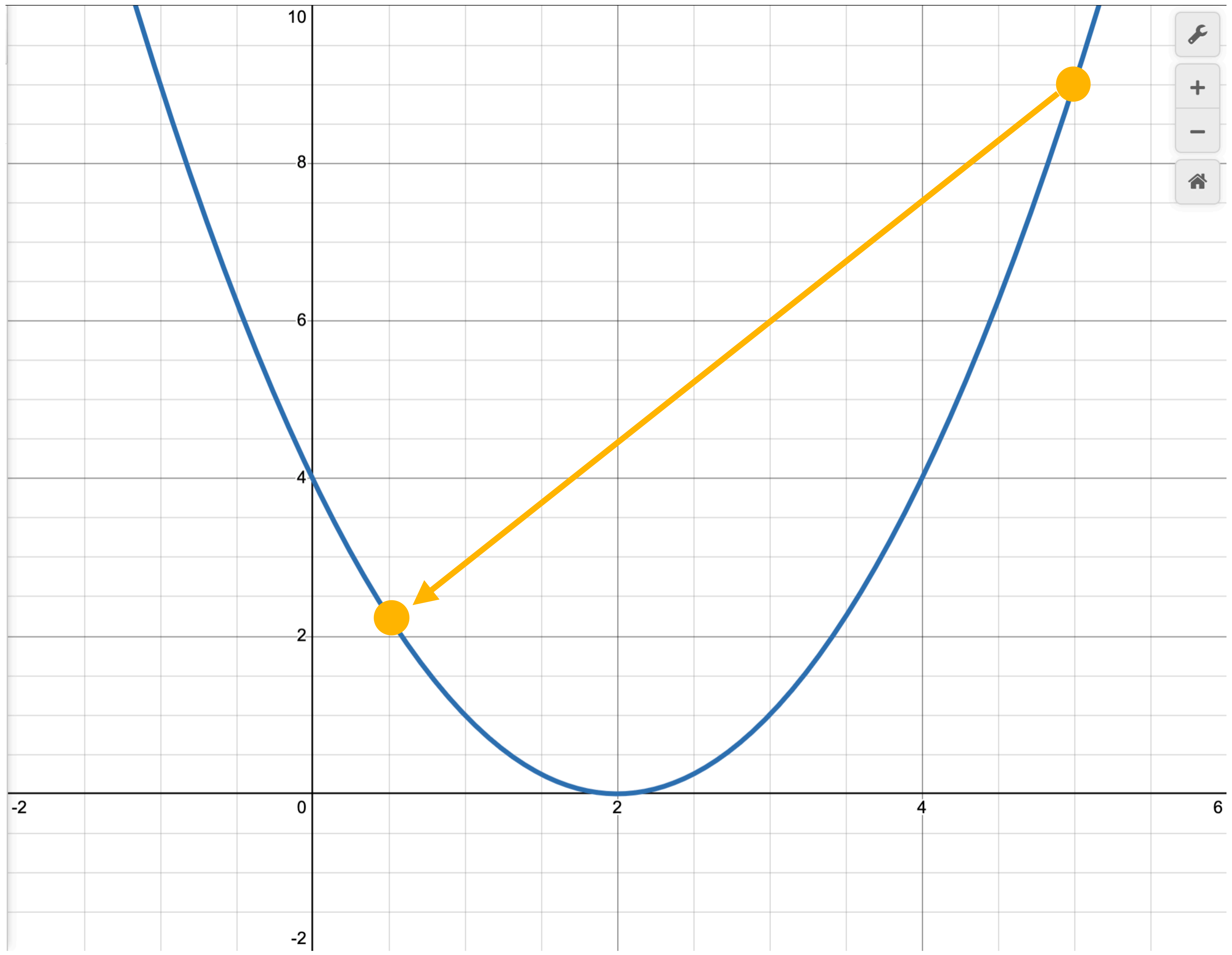
Gradient Descent (second try)

- Second step:



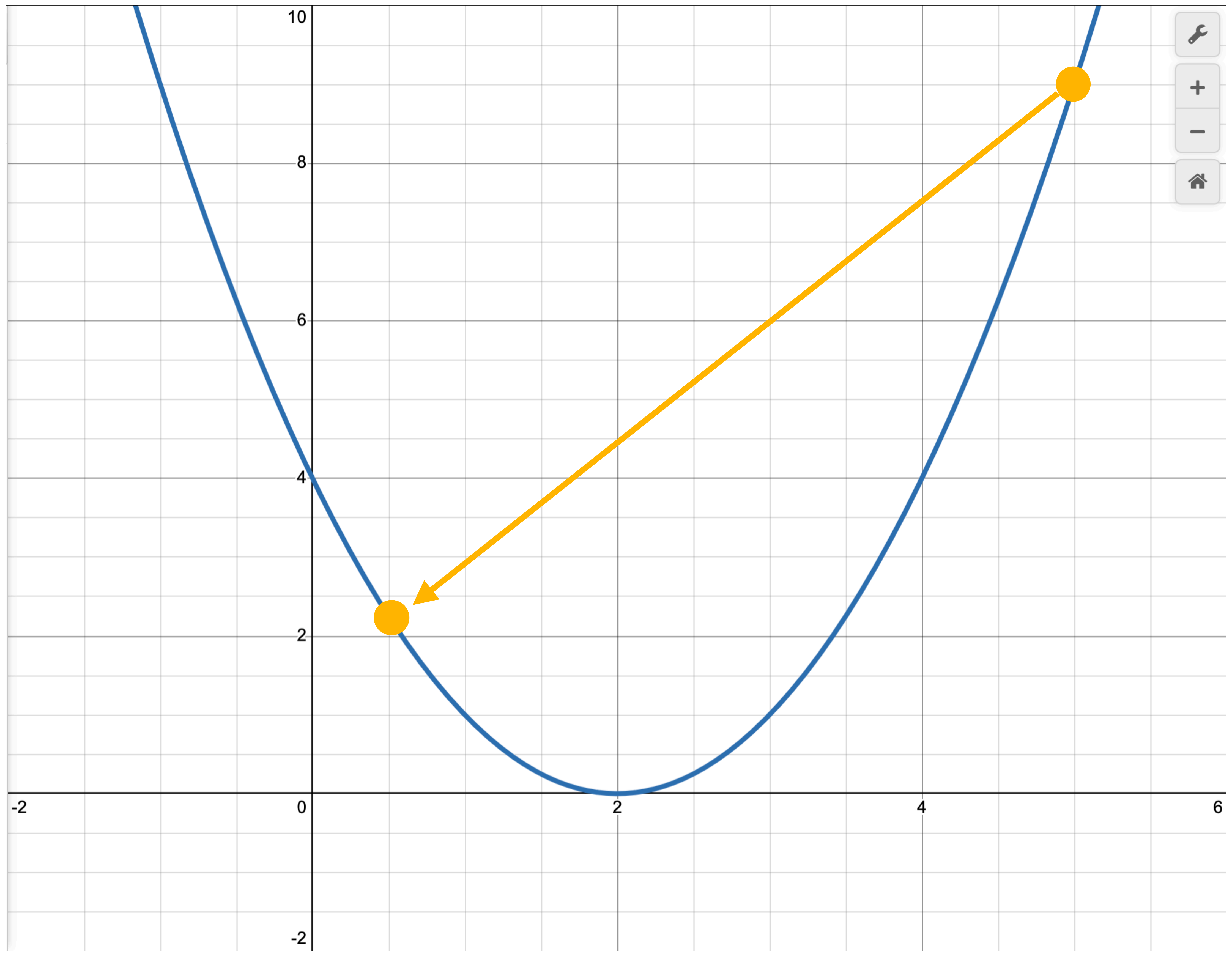
Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:



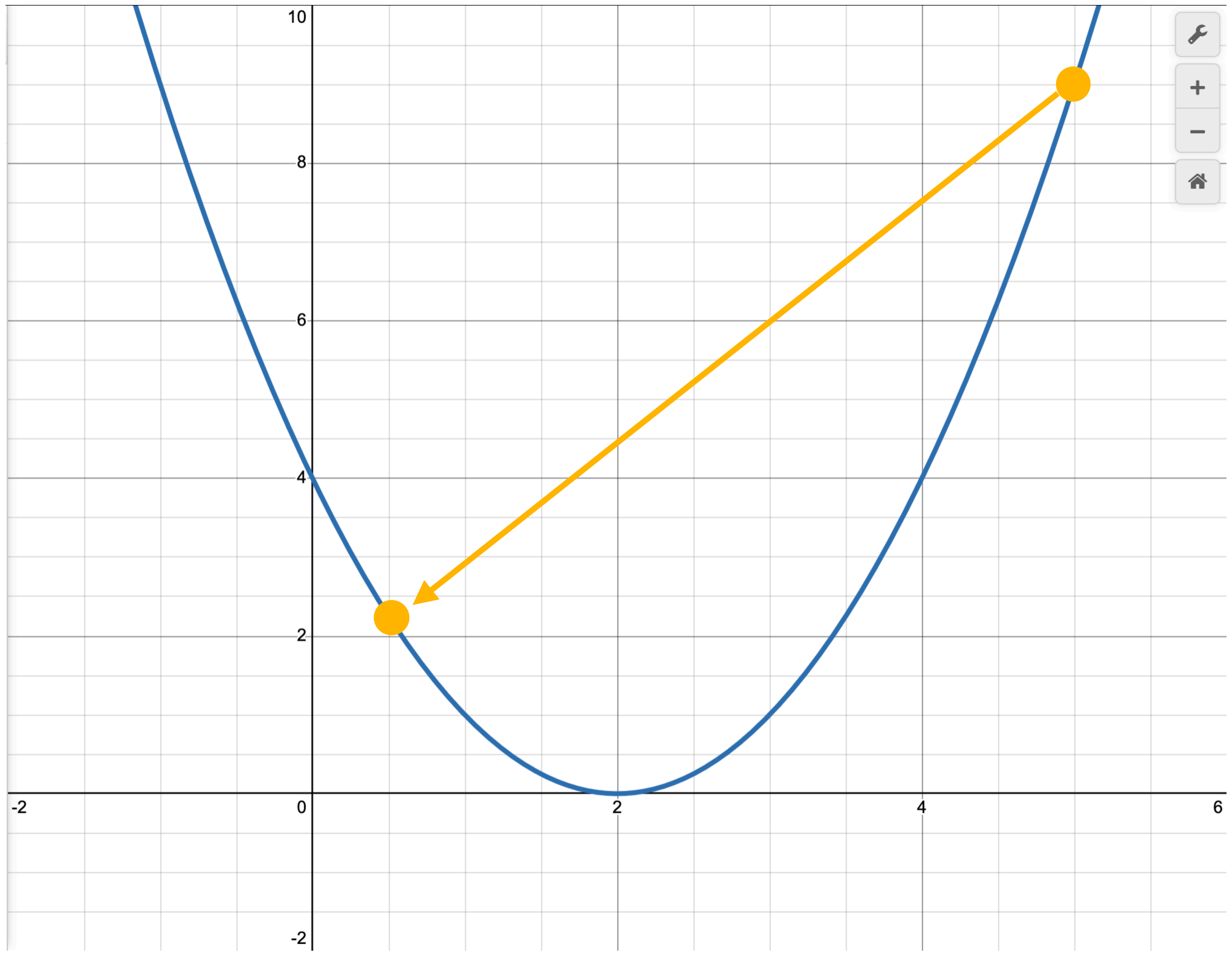
Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:
    - $2 \cdot 0.5 - 4 = -3$



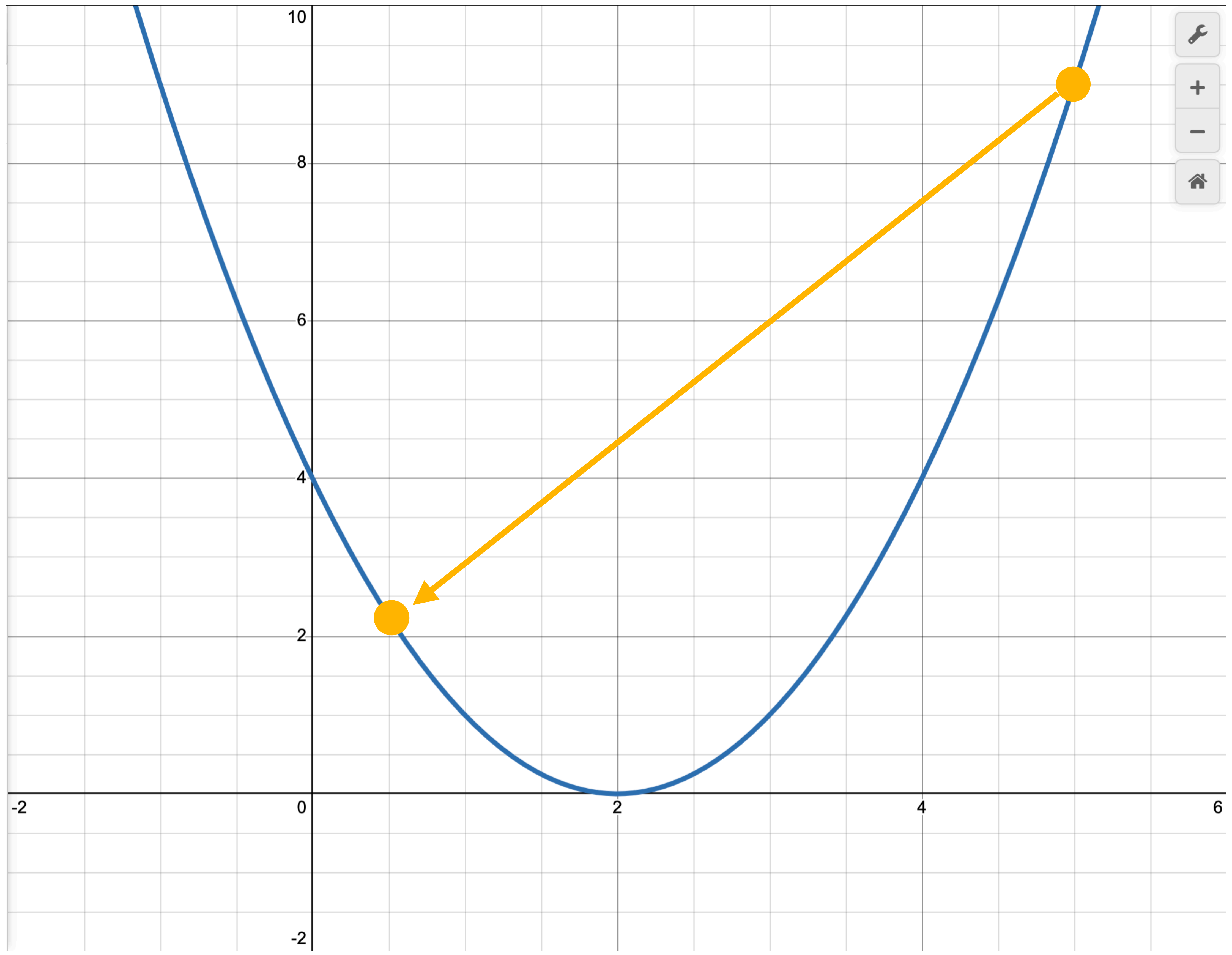
Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:
    - $2 \cdot 0.5 - 4 = -3$
  - Update θ:



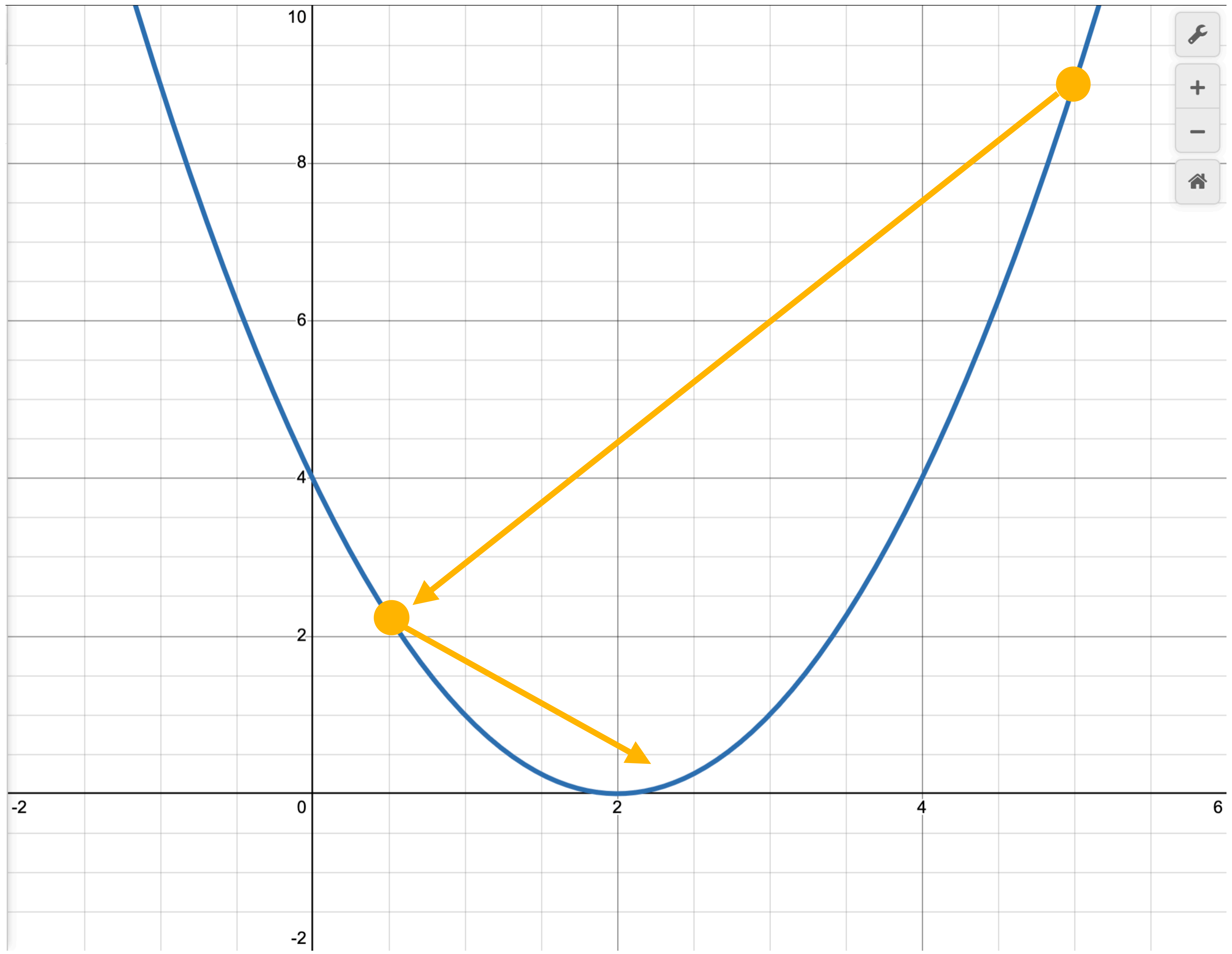
Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:
    - $2 \cdot 0.5 - 4 = -3$
  - Update θ:
    - $\theta_2 = \theta_1 - 0.75 \cdot (-3) = 2.25$



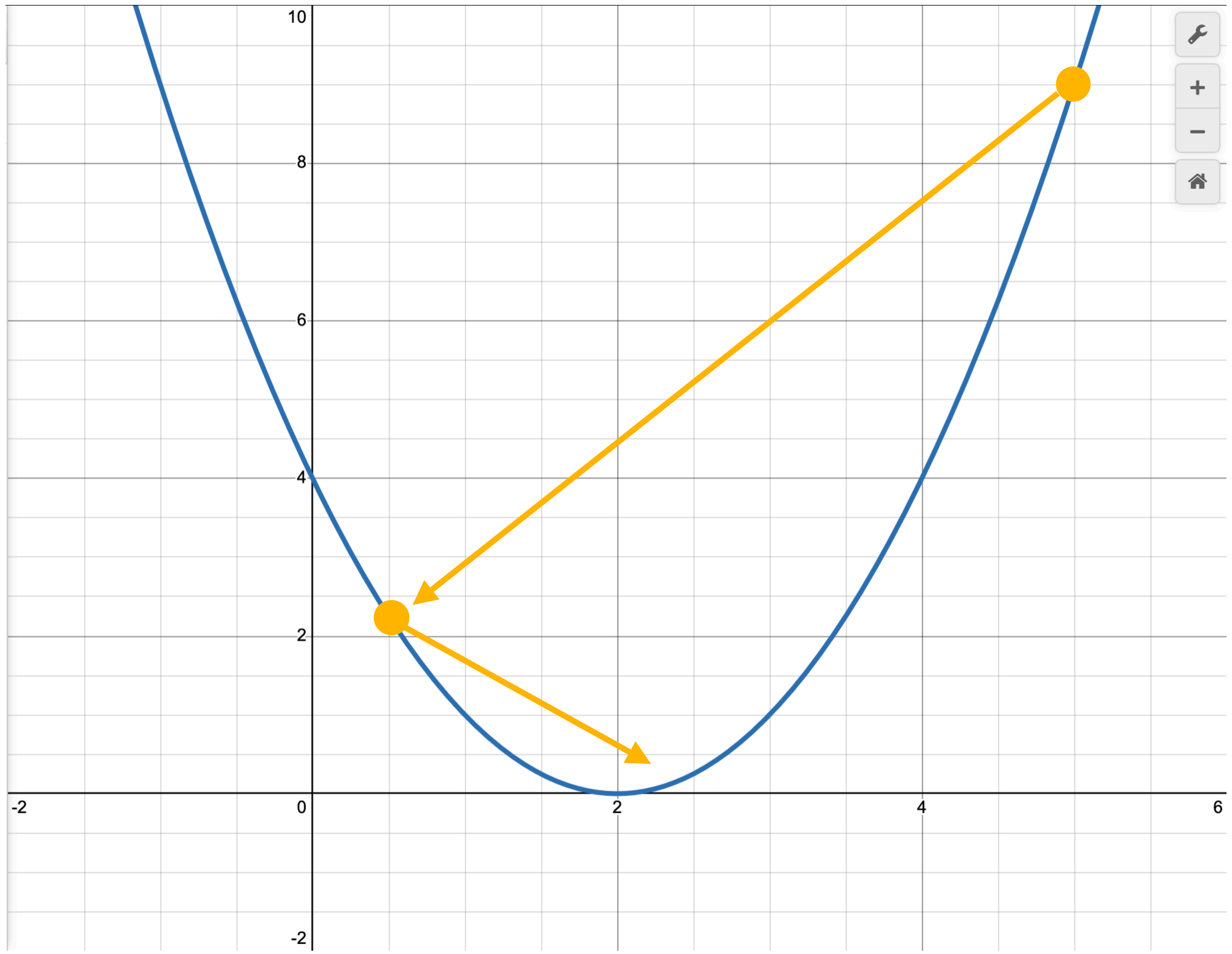
Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:
    - $2 \cdot 0.5 - 4 = -3$
  - Update θ:
    - $\theta_2 = \theta_1 - 0.75 \cdot (-3) = 2.25$

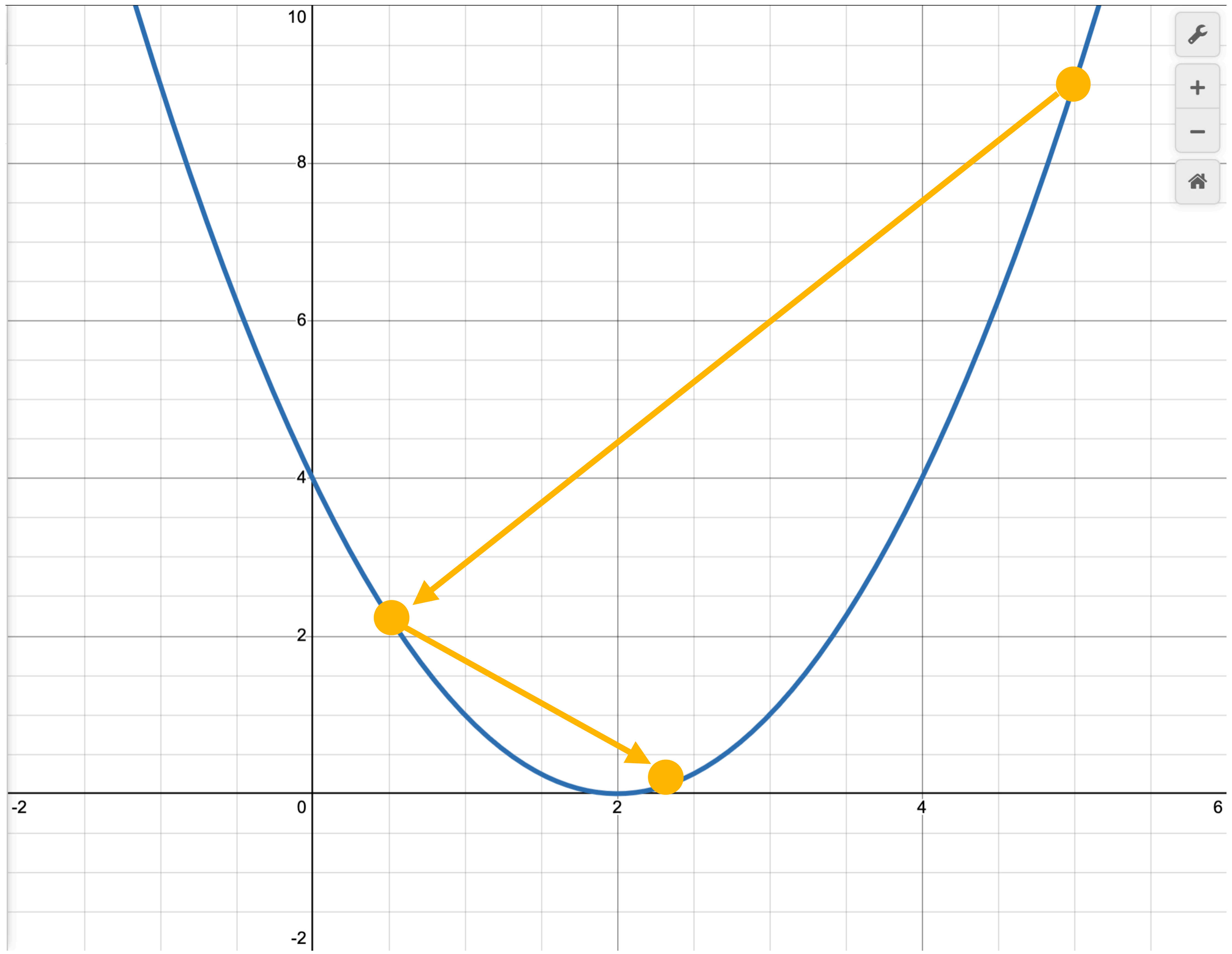


Gradient Descent (second try)

- Second step:
  - Plug in θ_1 to derivative:
    - $2 \cdot 0.5 - 4 = -3$
  - Update θ:
    - $\theta_2 = \theta_1 - 0.75 \cdot (-3) = 2.25$
- (This is looking better)

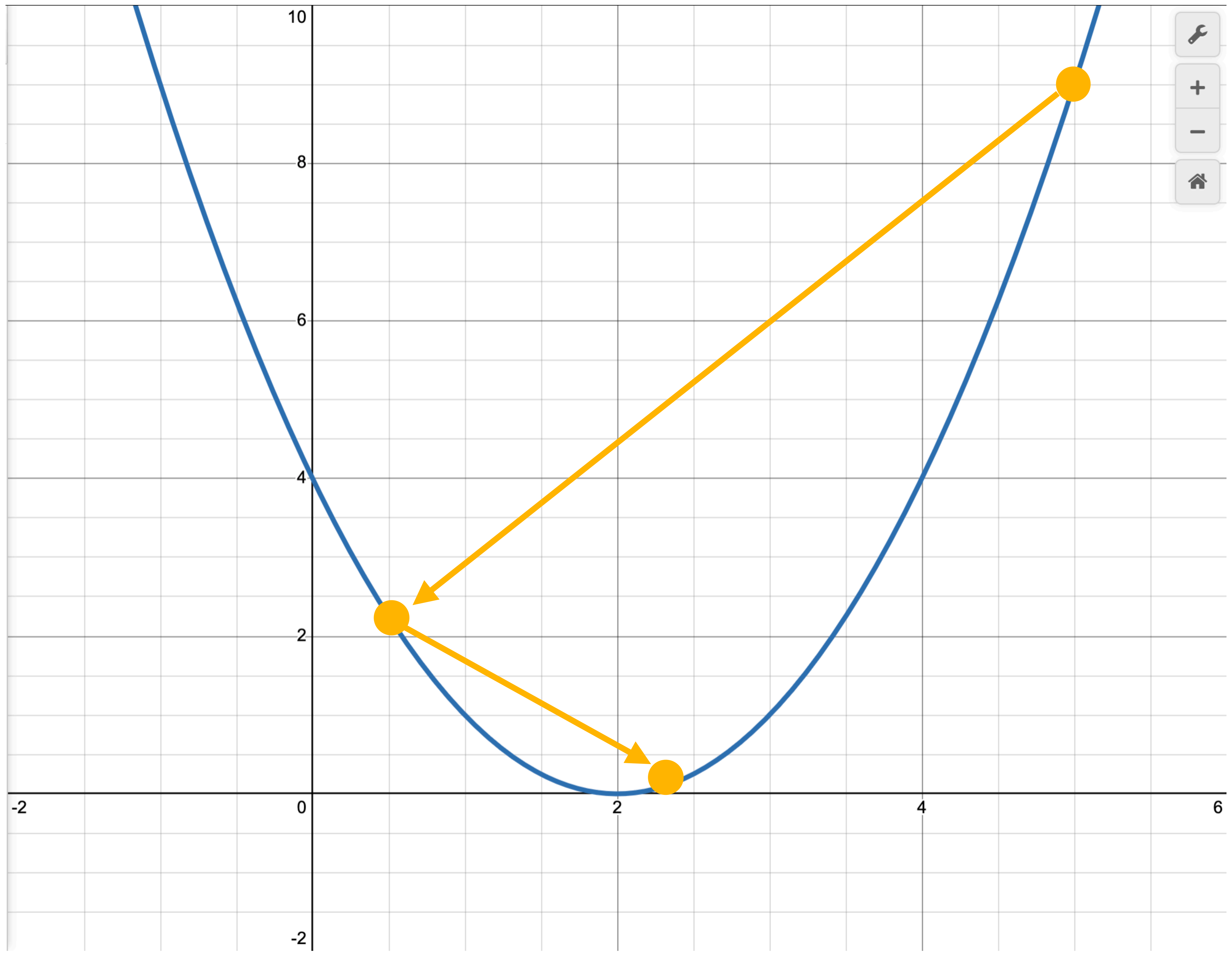


Gradient Descent (second try)



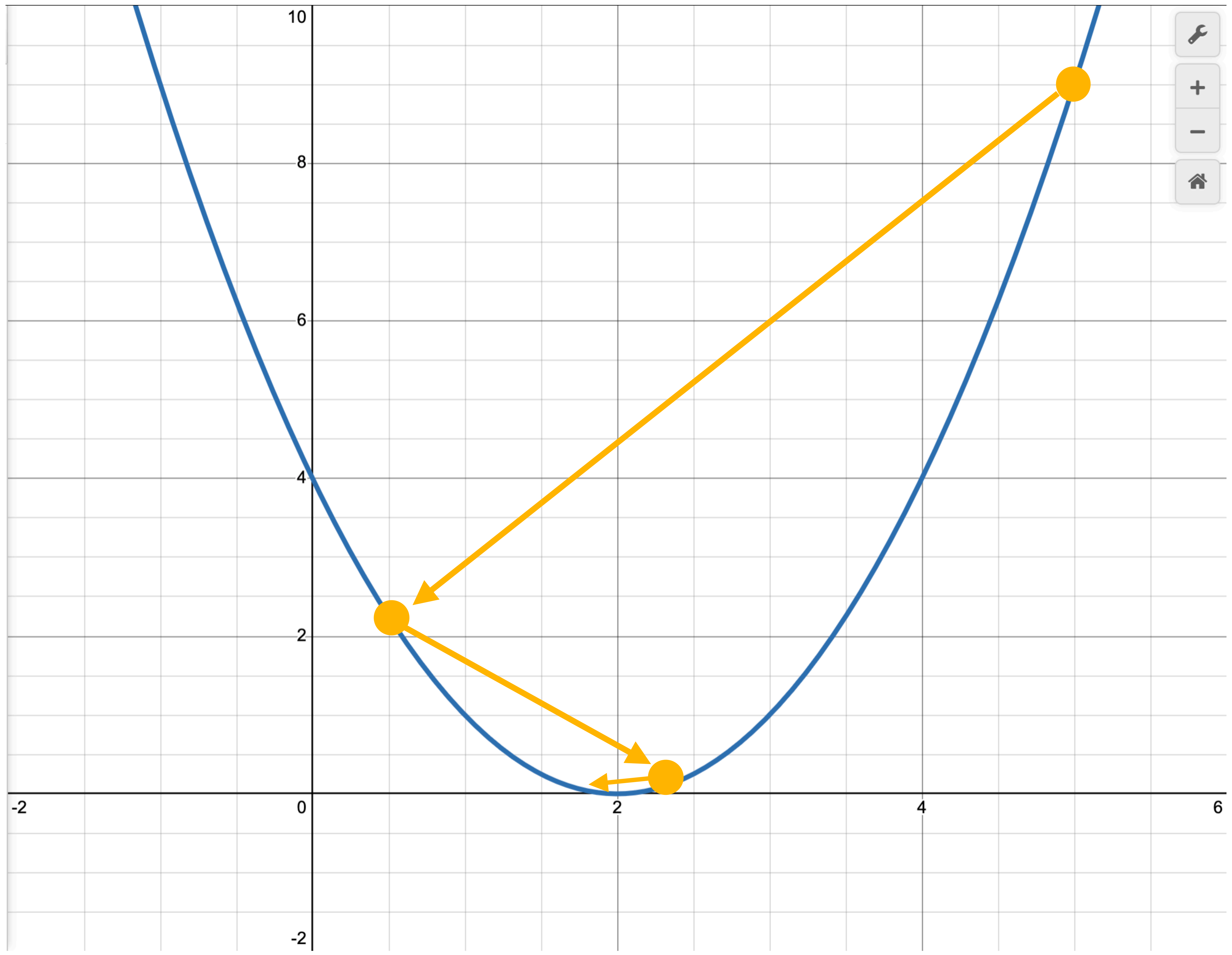
Gradient Descent (second try)

- Third step:
  - Plug in θ_2 to derivative:
    - $2 \cdot 2.25 - 4 = 0.5$
  - Update θ:
    - $\theta_3 = \theta_2 - 0.75 \cdot 0.5 = 1.875$



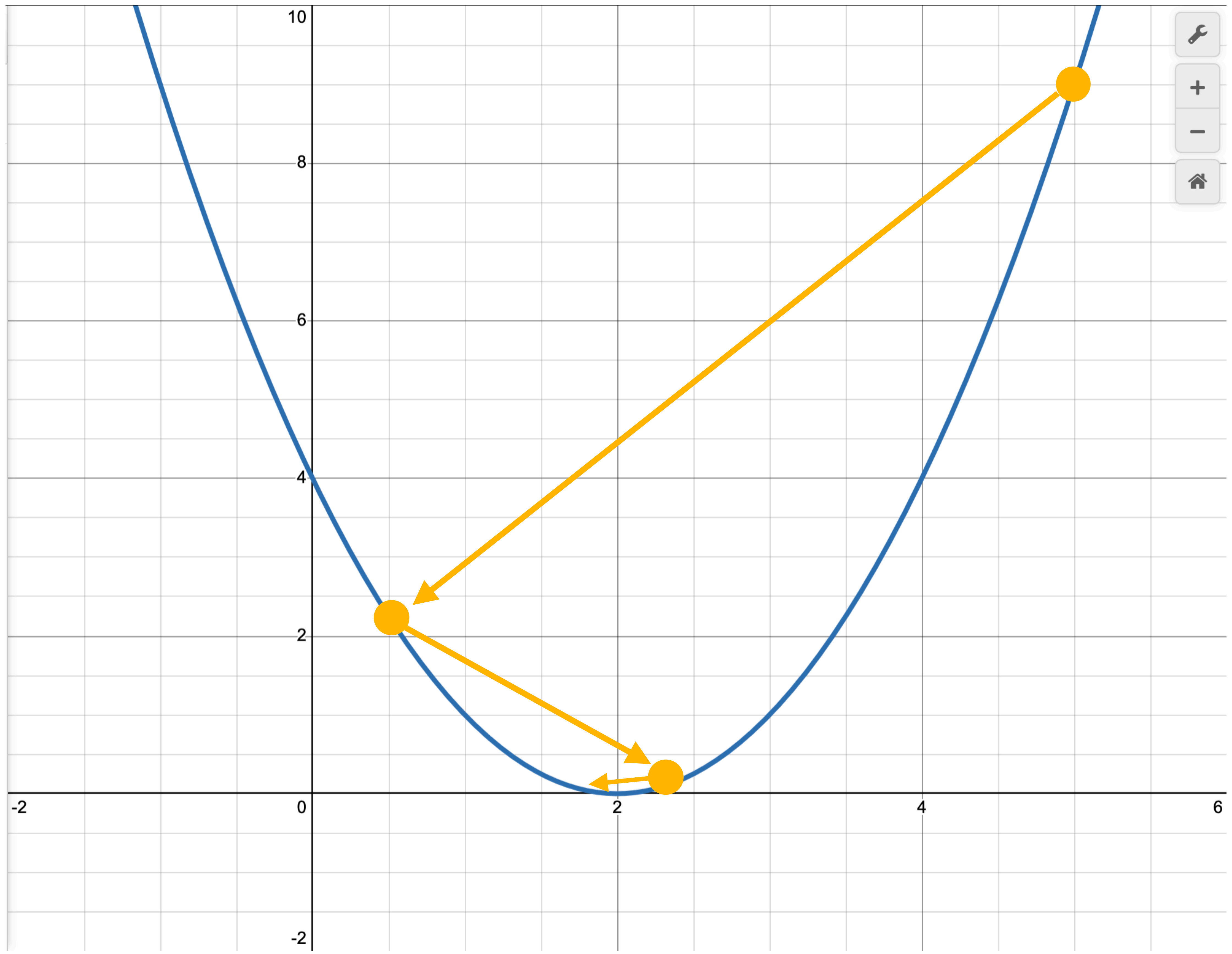
Gradient Descent (second try)

- Third step:
  - Plug in θ_2 to derivative:
    - $2 \cdot 2.25 - 4 = 0.5$
  - Update θ:
    - $\theta_3 = \theta_2 - 0.75 \cdot 0.5 = 1.875$



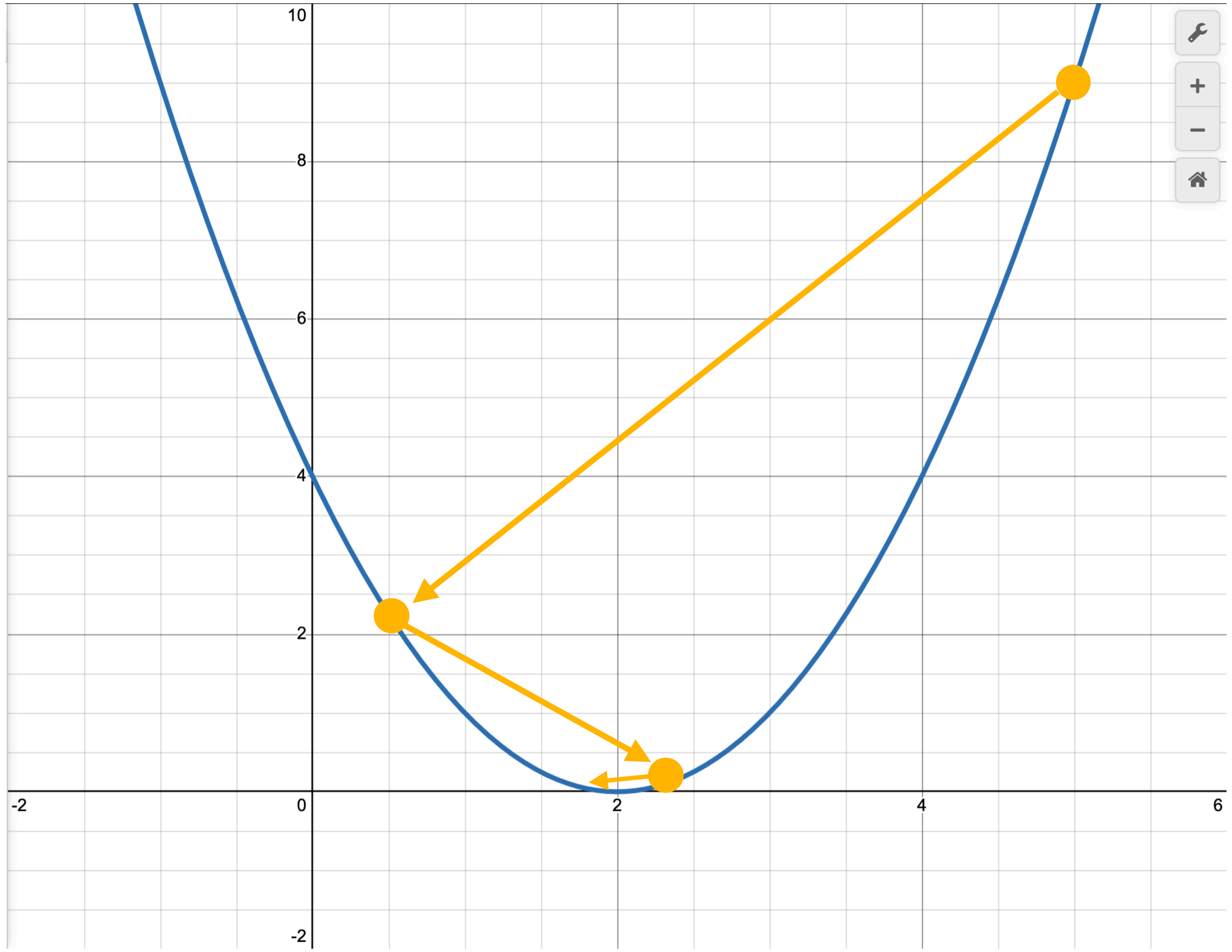
Gradient Descent (second try)

- Third step:
  - Plug in θ_2 to derivative:
    - $2 \cdot 2.25 - 4 = 0.5$
  - Update θ:
    - $\theta_3 = \theta_2 - 0.75 \cdot 0.5 = 1.875$
- **Loss gets lower with every step!**

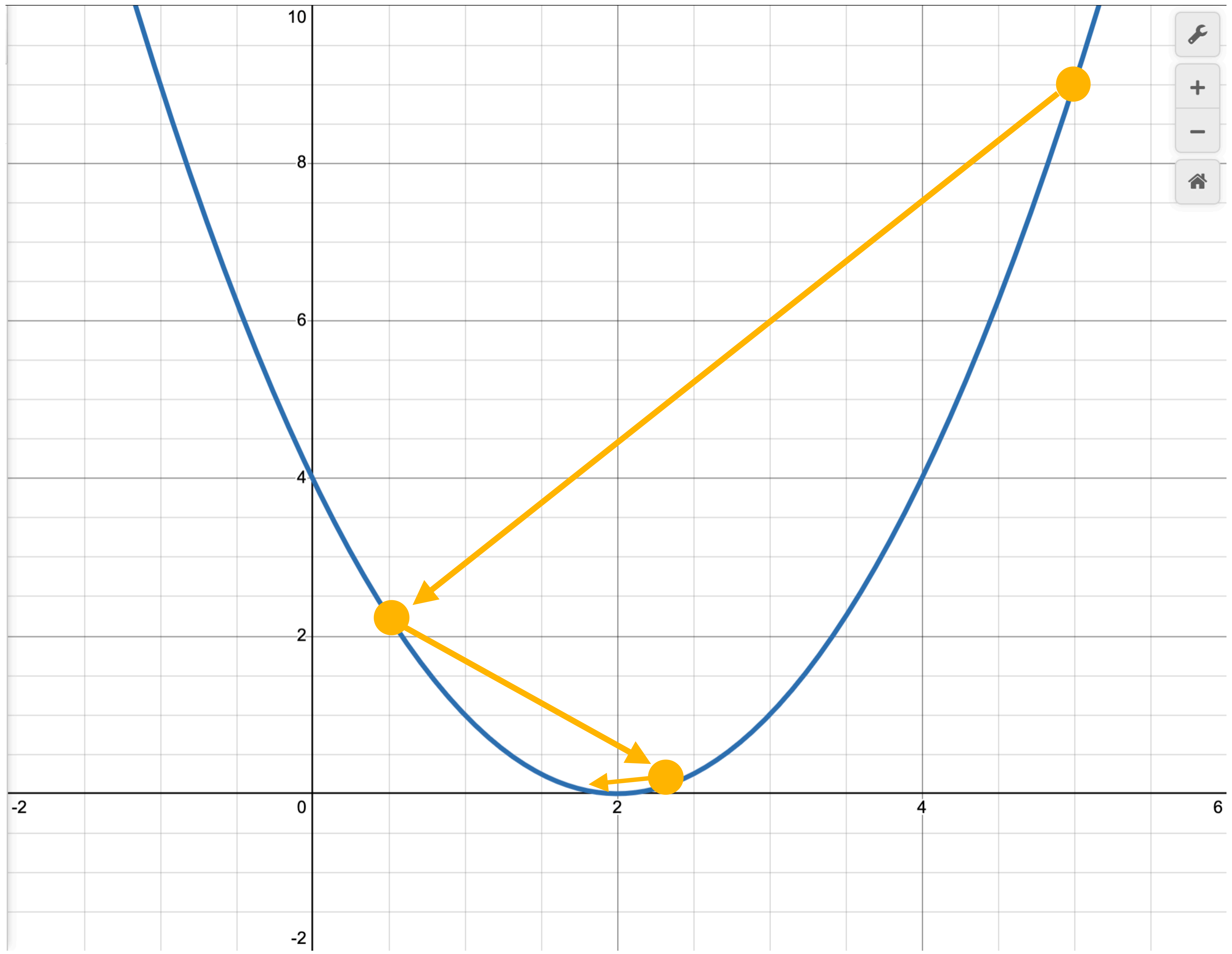


Gradient Descent (second try)

- Third step:
  - Plug in θ_2 to derivative:
    - $2 \cdot 2.25 - 4 = 0.5$
  - Update θ:
    - $\theta_3 = \theta_2 - 0.75 \cdot 0.5 = 1.875$
- **Loss gets lower with every step!**
- θ gets **arbitrarily close to the optimal value** with more steps

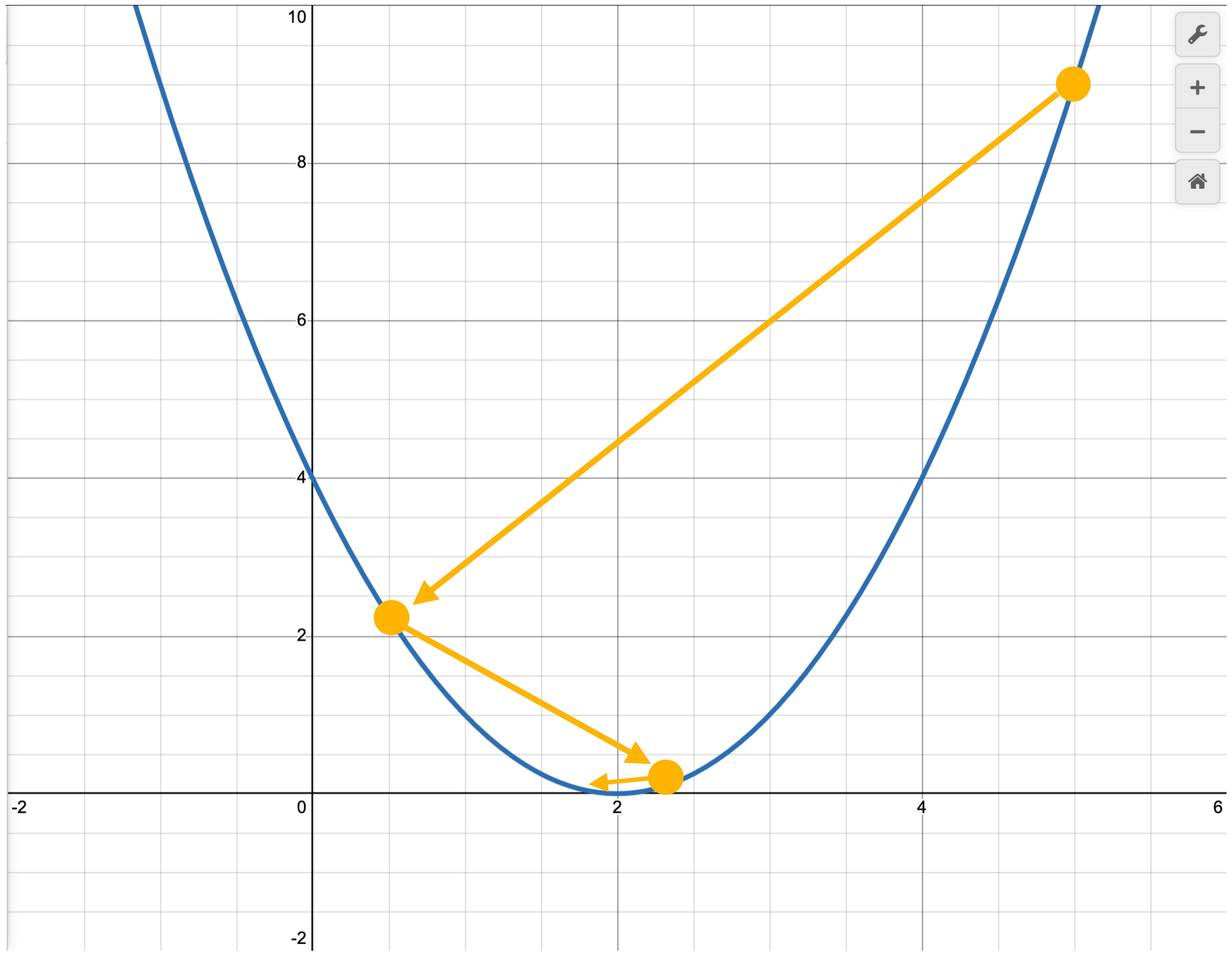


Learning Rate



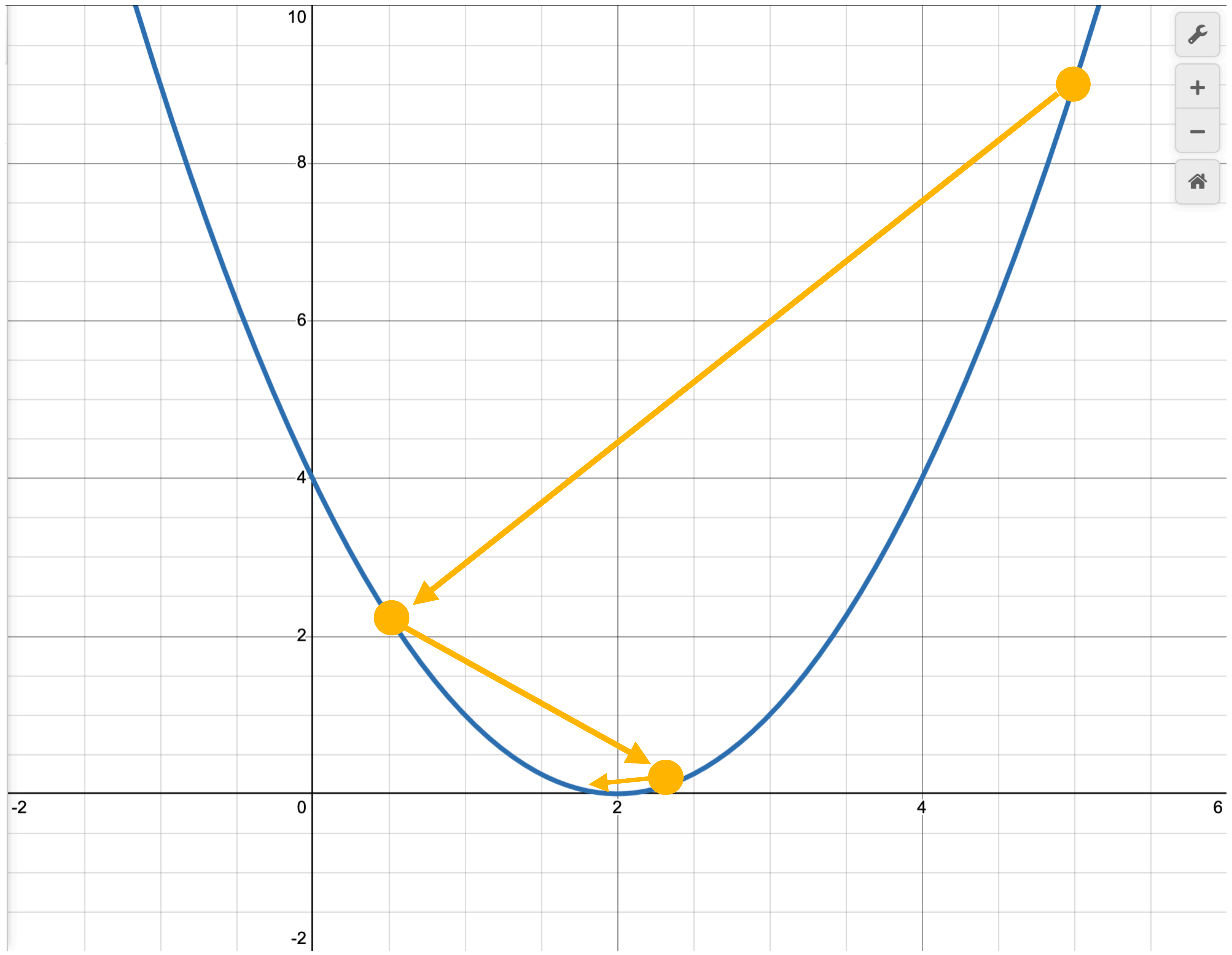
Learning Rate

- The fraction by which we multiplied our adjustment is called the **Learning Rate**



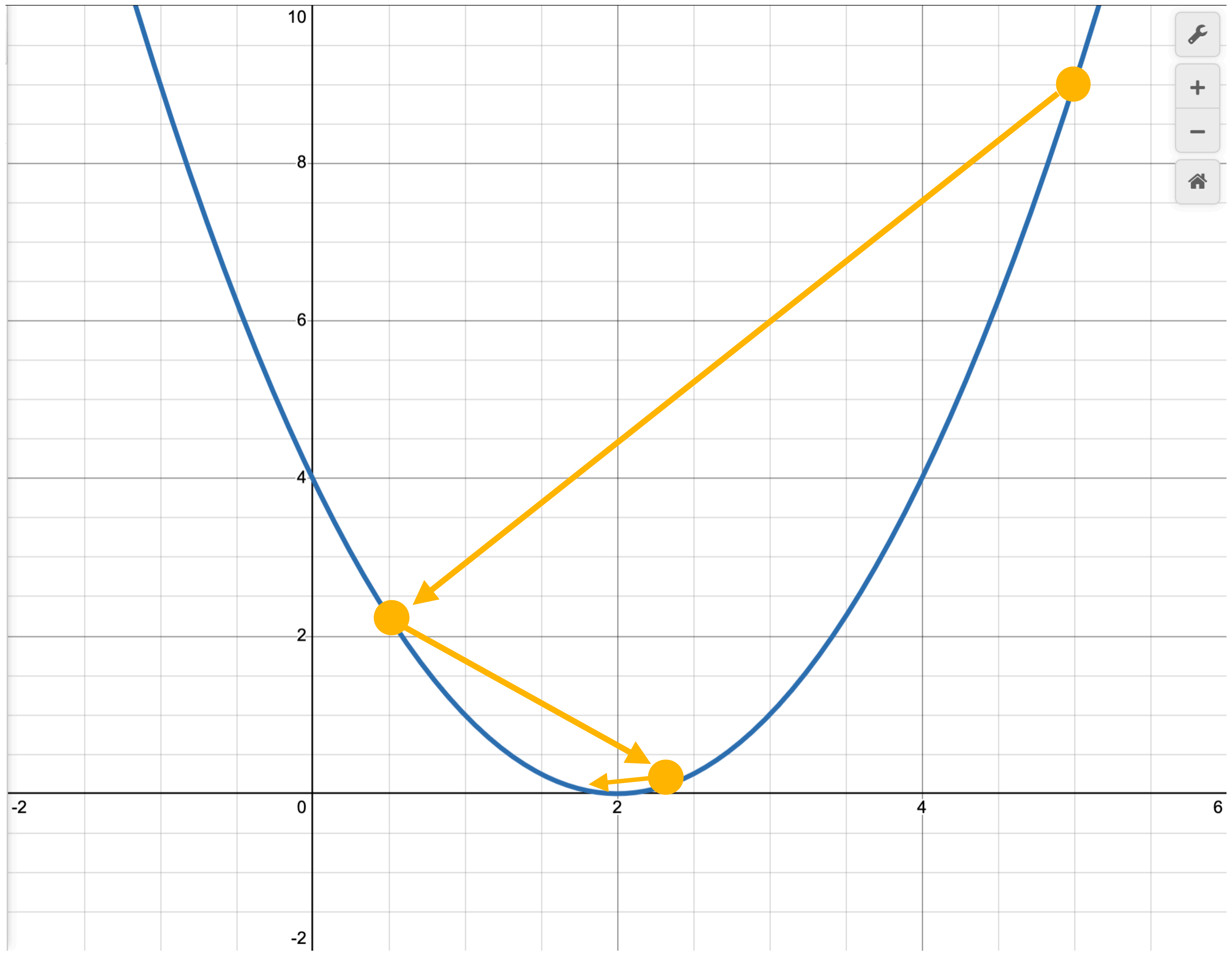
Learning Rate

- The fraction by which we multiplied our adjustment is called the **Learning Rate**
- In practice, LR is chosen by **trial and error** (called "tuning")



Learning Rate

- The fraction by which we multiplied our adjustment is called the **Learning Rate**
- In practice, LR is chosen by **trial and error** (called "tuning")
- Risks of different values
  - Too high → **"bouncing around"** and missing an optimum
  - Too low → taking **many steps** to reach the optimum



Noisy Secret Number Game

Adding noise to the game

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?
- $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?
- $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$
  - Is there a θ that **exactly solves** this secret number dataset?

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?
- $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$
  - Is there a θ that **exactly solves** this secret number dataset?
  - **No!** (At least assuming our function is still $f(x, \theta) = x + \theta$)

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?
- $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$
  - Is there a θ that **exactly solves** this secret number dataset?
  - **No!** (At least assuming our function is still $f(x, \theta) = x + \theta$)
- Gradient Descent allows us to make an **optimal estimate** given the data

Adding noise to the game

- Let's do a **slightly more complicated** version of the number game
- What if our dataset encodes a **noisy relationship** between input/output?
- $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$
  - Is there a θ that **exactly solves** this secret number dataset?
  - **No!** (At least assuming our function is still $f(x, \theta) = x + \theta$)
- Gradient Descent allows us to make an **optimal estimate** given the data
- Unlike the previous example, we need to **define the loss function** over the **entire dataset**

Global loss function

$$\mathscr{L}(f(X,\theta),Y) = \frac{1}{N}\sum_{i=1}^{N} \ell(f(x_i,\theta),y_i)$$

loss over **all datapoints**

average

loss for a **single datapoint**

Batch Gradient Descent

Batch Gradient Descent

- Optimizing the loss function over **all datapoints at once** is known as **Batch Gradient Descent**

Batch Gradient Descent

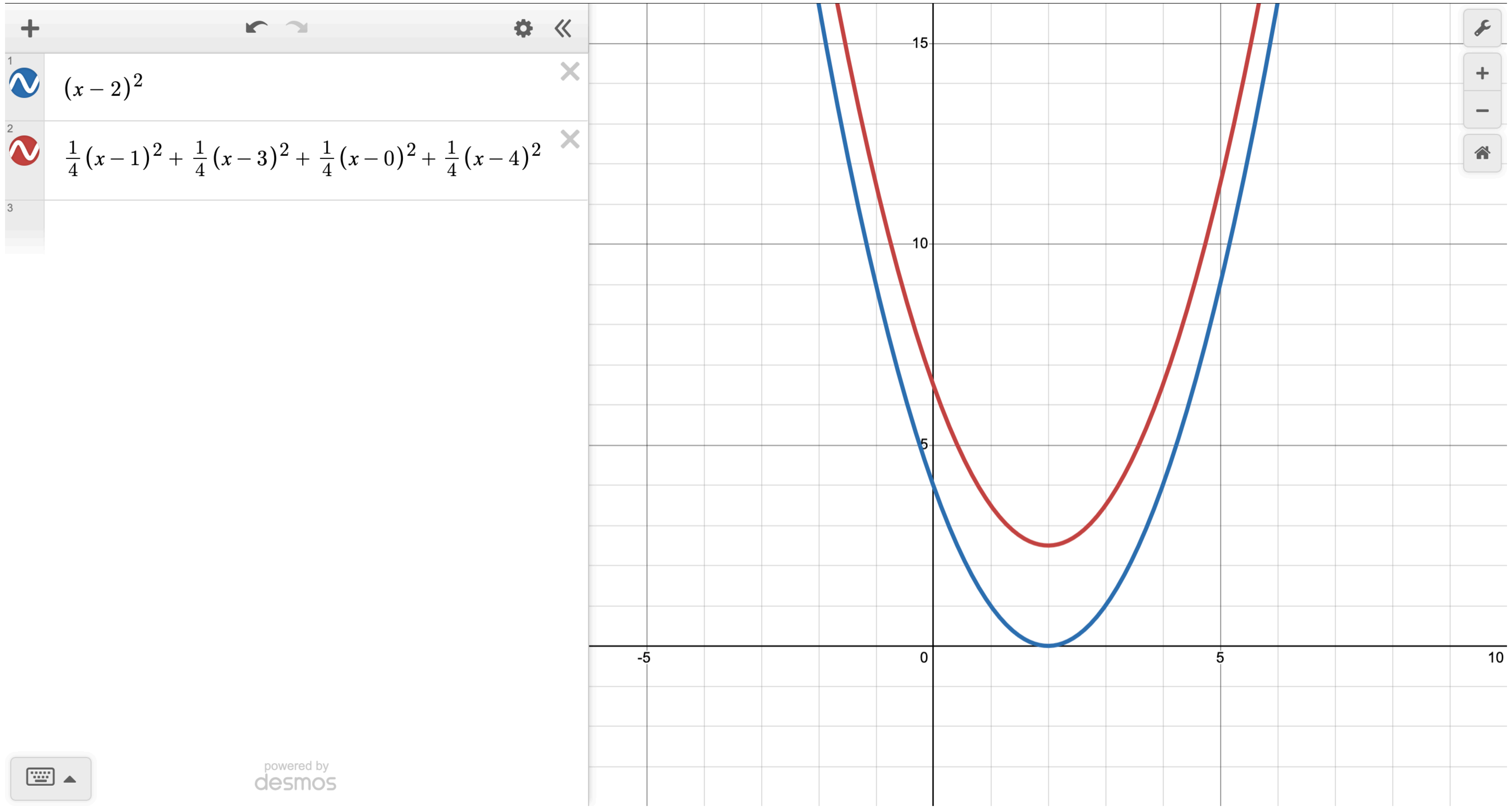
- Optimizing the loss function over **all datapoints at once** is known as **Batch Gradient Descent**
- This ensures our learned parameter(s) θ are **optimized for the dataset as a whole**, rather than for any single example

Batch Gradient Descent

- Optimizing the loss function over **all datapoints at once** is known as **Batch Gradient Descent**
- This ensures our learned parameter(s) θ are **optimized for the dataset as a whole**, rather than for any single example
- **Any guesses** what the optimal value of θ is for our **noisy dataset?**
  - $D = \{(2, 3), (3, 6), (5, 5), (8, 12)\}$
  - Hint, look at the **difference between each pair**

Batch Gradient Descent

- The optimal θ is still 2! (The **average** input/output difference)
- Note that the **optimal loss is > 0** (i.e. there is some error left over!)



Summary so far

Summary

Summary

- Gradient Descent finds the **ideal parameter(s)** θ for a **parameterized function** $f(x, \theta)$, in order to model a **dataset** $D = \{X, Y\}$

Summary

- Gradient Descent finds the **ideal parameter(s)** θ for a **parameterized function** $f(x, \theta)$, in order to model a **dataset** $D = \{X, Y\}$
- The optimal parameters **minimize error/loss** between the **predicted output** $\hat{y} = f(x, \theta)$ and the **true output** y

Summary

- Gradient Descent finds the **ideal parameter(s)** θ for a **parameterized function** $f(x, \theta)$, in order to model a **dataset** $D = \{X, Y\}$
- The optimal parameters **minimize error/loss** between the **predicted output** $\hat{y} = f(x, \theta)$ and the **true output** y
    - We define and minimize a **loss function** $\mathscr{L}(f(X, \theta), Y)$

Summary

- Gradient Descent finds the **ideal parameter(s)** θ for a **parameterized function** $f(x, \theta)$, in order to model a **dataset** $D = \{X, Y\}$
- The optimal parameters **minimize error/loss** between the **predicted output** $\hat{y} = f(x, \theta)$ and the **true output** y
  - We define and minimize a **loss function** $\mathscr{L}(f(X, \theta), Y)$
- Gradient Descent uses the **derivative of the loss function** $\frac{d}{d\theta}\mathscr{L}(f(X, \theta), Y)$ to **follow the slope to the minimal point**

Summary

- Gradient Descent finds the **ideal parameter(s)** θ for a **parameterized function** $f(x, \theta)$, in order to model a **dataset** $D = \{X, Y\}$
- The optimal parameters **minimize error/loss** between the **predicted output** $\hat{y} = f(x, \theta)$ and the **true output** y
  - We define and minimize a **loss function** $\mathscr{L}(f(X, \theta), Y)$
- Gradient Descent uses the **derivative of the loss function** $\frac{d}{d\theta}\mathscr{L}(f(X, \theta), Y)$ to **follow the slope to the minimal point**
  - This is an **iterative process** that sometimes needs a tuned **learning rate**

Caveats

Caveats

- The loss functions we've seen so far are a **special type** called **convex**

Caveats

- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)

Caveats

- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)
  - This is because convex functions only have **one minimum**, which is always the **global minimum** (i.e. the function is **shaped like a bowl**)

Caveats

- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)
  - This is because convex functions only have **one minimum**, which is always the **global minimum** (i.e. the function is **shaped like a bowl**)
  - Gradient Descent is **guaranteed** to converge to the **optimum solution** for convex functions (as long is the learning rate isn't too high)

Caveats

- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)
  - This is because convex functions only have **one minimum**, which is always the **global minimum** (i.e. the function is **shaped like a bowl**)
  - Gradient Descent is **guaranteed** to converge to the **optimum solution** for convex functions (as long is the learning rate isn't too high)
  - GD is **NOT** guaranteed to converge for **non-convex functions**

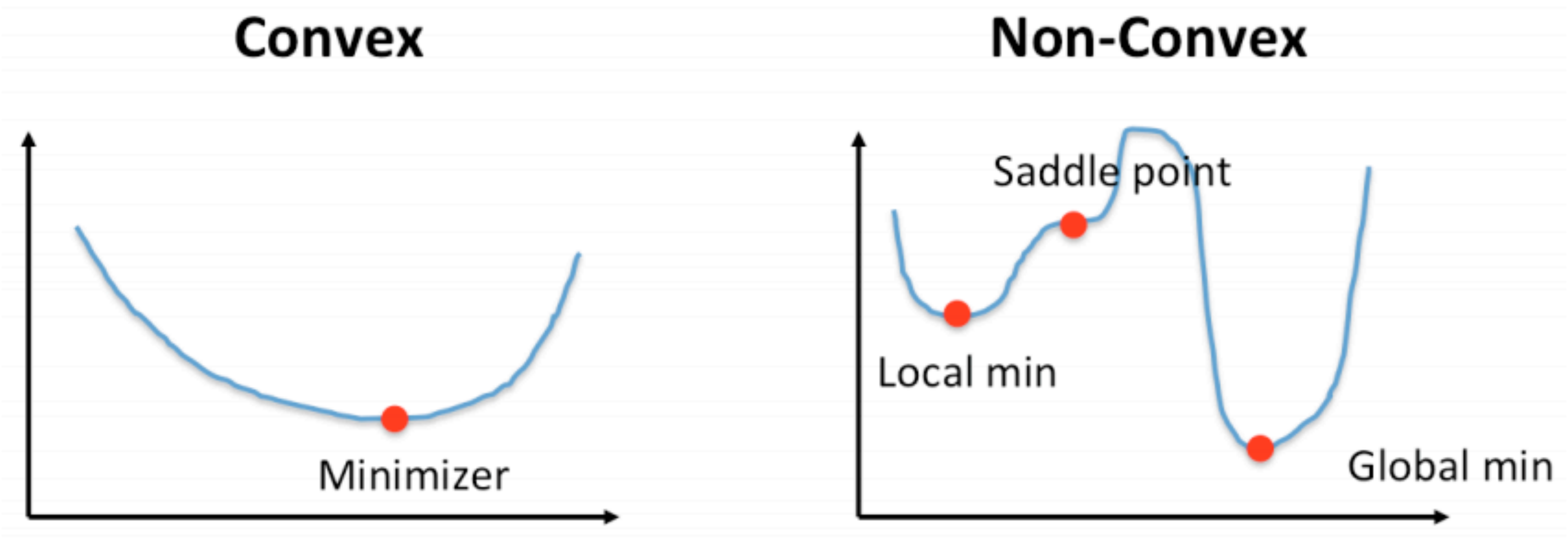
Caveats

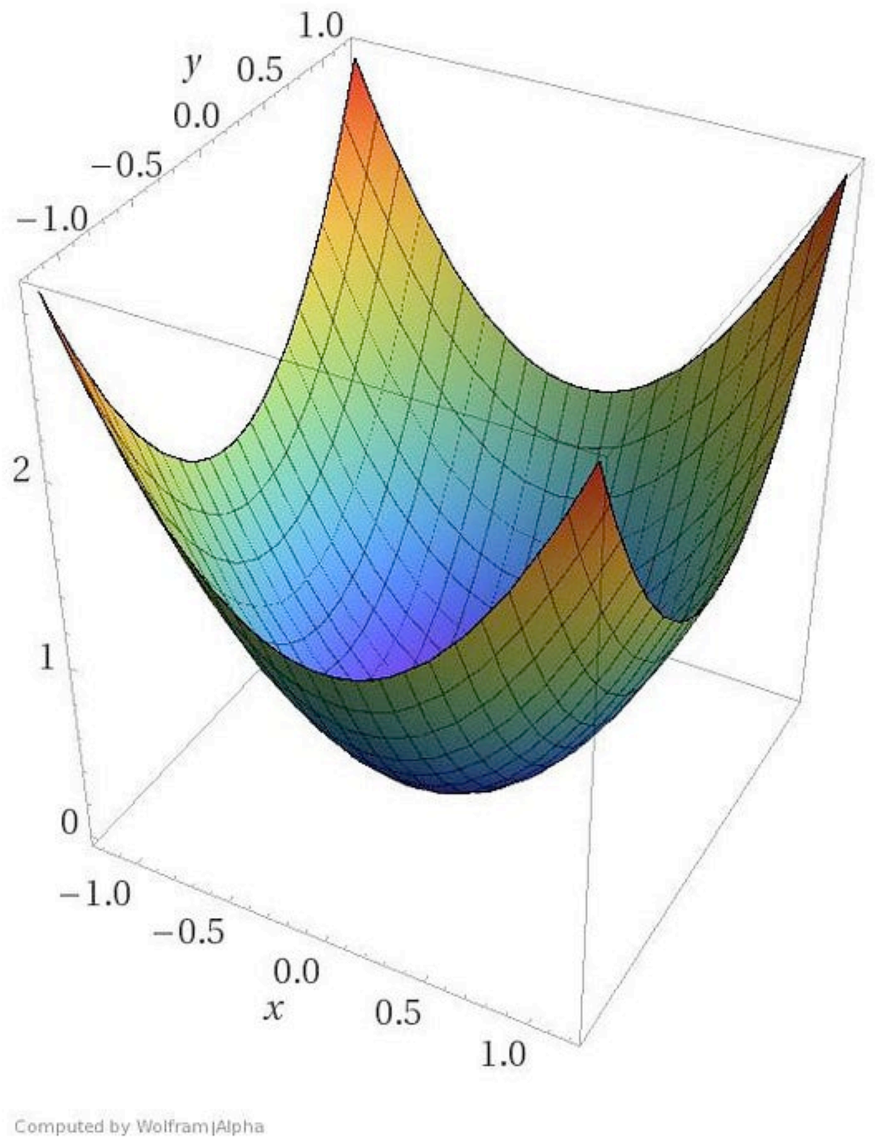
- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)
  - This is because convex functions only have **one minimum**, which is always the **global minimum** (i.e. the function is **shaped like a bowl**)
  - Gradient Descent is **guaranteed** to converge to the **optimum solution** for convex functions (as long is the learning rate isn't too high)
  - GD is **NOT** guaranteed to converge for **non-convex functions**
- We've only looked at functions with a **single parameter**

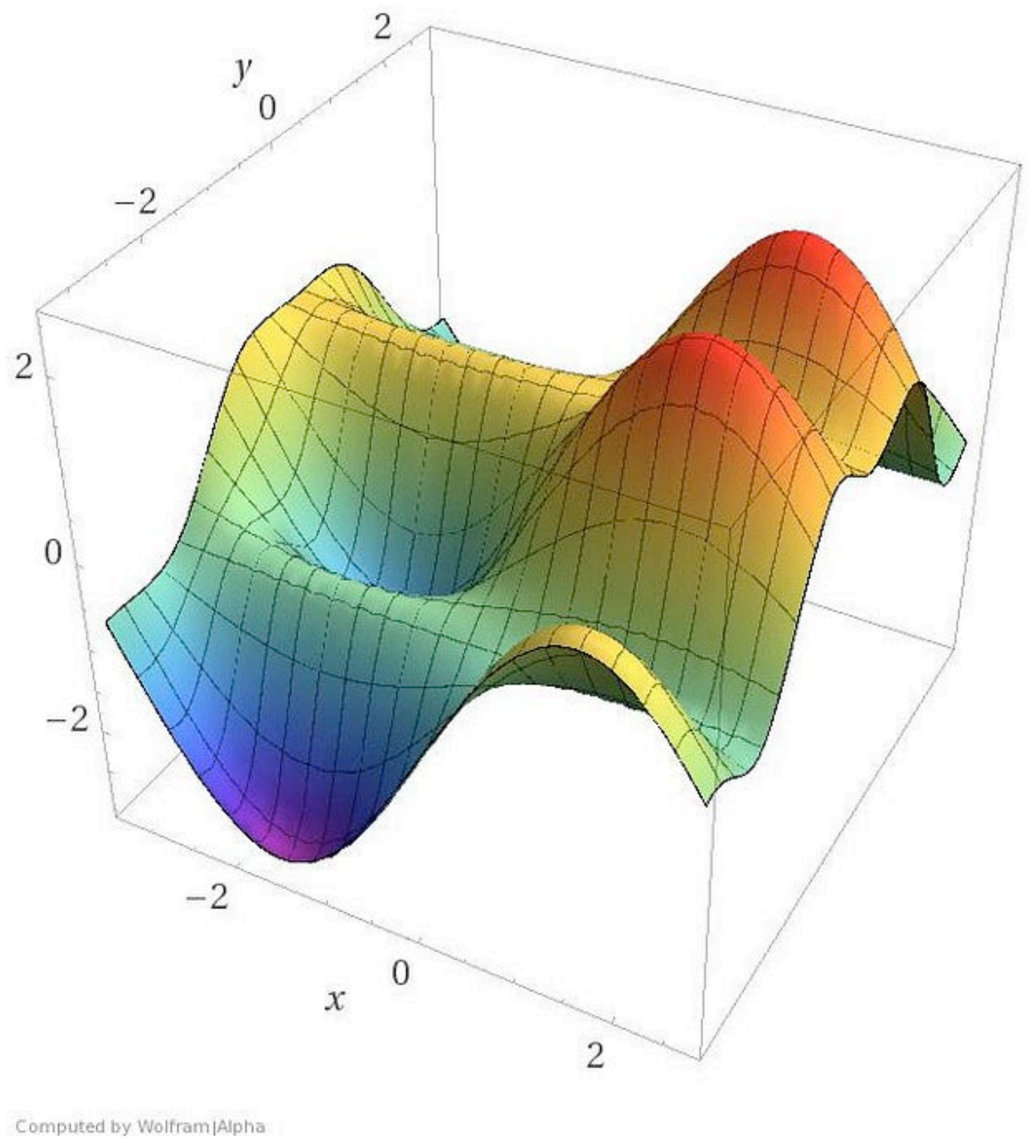
Caveats

- The loss functions we've seen so far are a **special type** called **convex**
  - Convex functions can be **optimized WITHOUT gradient descent** (sometimes just by noting **where the derivative is 0**)
  - This is because convex functions only have **one minimum**, which is always the **global minimum** (i.e. the function is **shaped like a bowl**)
  - Gradient Descent is **guaranteed** to converge to the **optimum solution** for convex functions (as long is the learning rate isn't too high)
  - GD is **NOT** guaranteed to converge for **non-convex functions**
- We've only looked at functions with a **single parameter**
  - You can have **as many parameters as you want!**

Convex vs. Non-convex

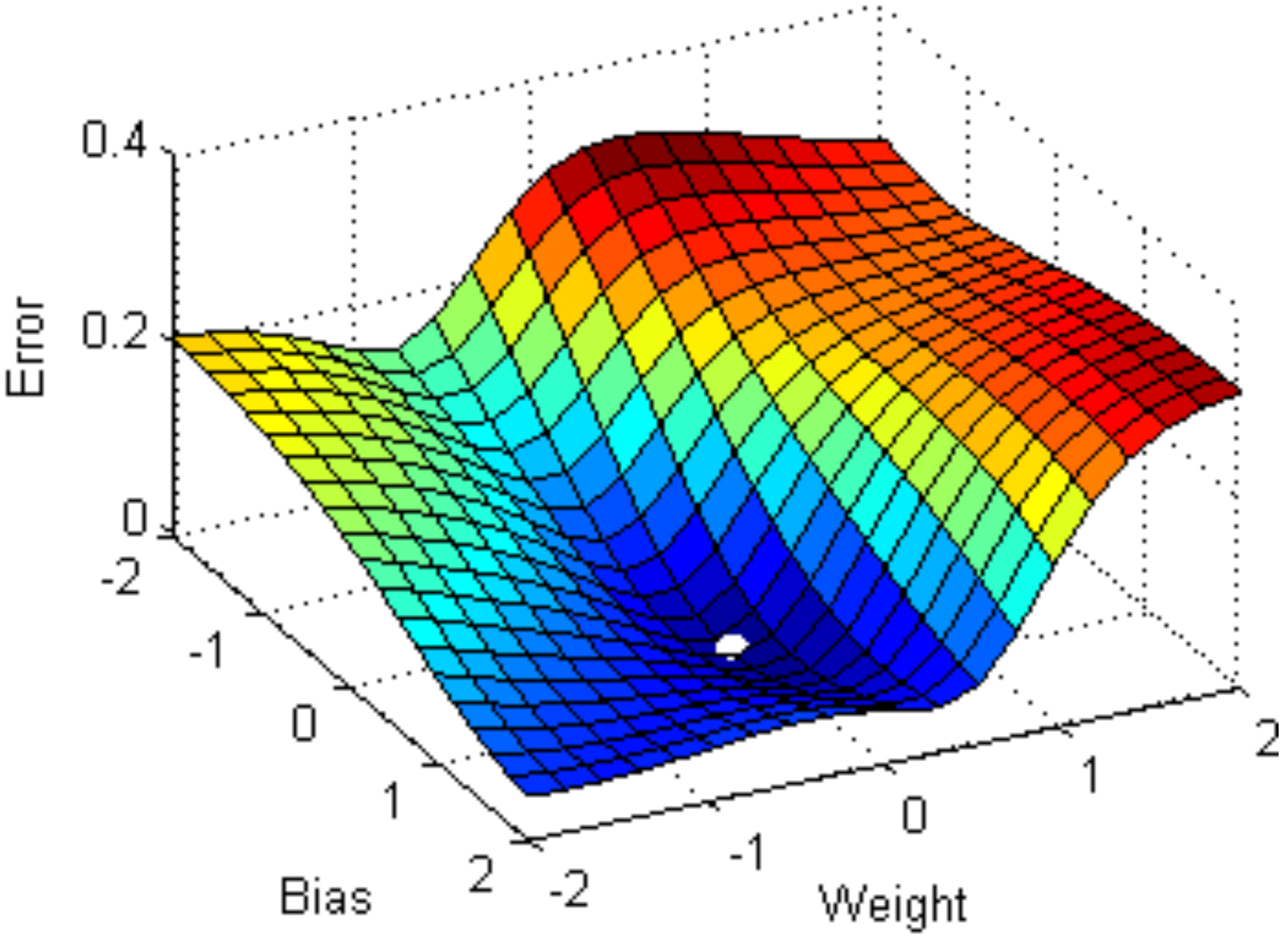






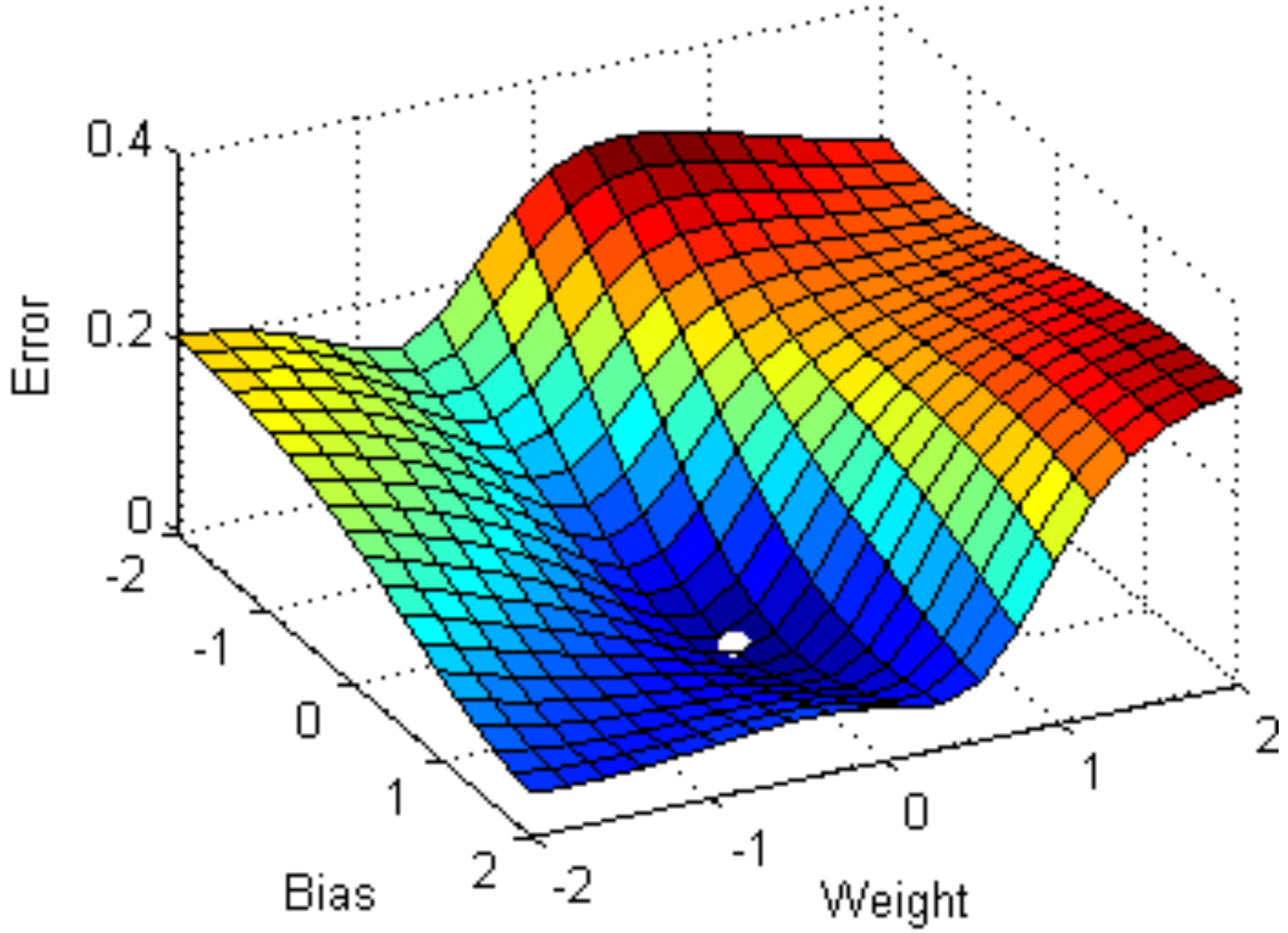
Multi-variable Functions / Gradients

Gradient Descent: Intuition



source

Gradient Descent: Intuition



source

Partial Derivatives

- A **partial derivative** is needed for a function with **multiple input variables**
- Each measures slope respect **one variable**, with the others held constant

Partial Derivatives

- A **partial derivative** is needed for a function with **multiple input variables**
- Each measures slope respect **one variable**, with the others held constant

$$f(x, y) = 10x^3y^2 + 5xy^3 + 4x + y$$

Partial Derivatives

- A **partial derivative** is needed for a function with **multiple input variables**
- Each measures slope respect **one variable**, with the others held constant

$$f(x, y) = 10x^3y^2 + 5xy^3 + 4x + y$$

$$\frac{\partial f}{\partial x} = 30x^2y^2 + 5y^3 + 4$$

Partial Derivatives

- A **partial derivative** is needed for a function with **multiple input variables**
- Each measures slope respect **one variable**, with the others held constant

$$f(x, y) = 10x^3y^2 + 5xy^3 + 4x + y$$

$$\frac{\partial f}{\partial x} = 30x^2y^2 + 5y^3 + 4$$

$$\frac{\partial f}{\partial y} = 20x^3y + 15xy^2 + 1$$

Gradient

Gradient

- The gradient of a function $f(x_1, x_2, \ldots x_n)$ is a **vector**, consisting of **all partial derivatives**

Gradient

- The gradient of a function $f(x_1, x_2, \ldots x_n)$ is a **vector**, consisting of **all partial derivatives**

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$$

Gradient

- The gradient of a function $f(x_1, x_2, ... x_n)$ is a **vector**, consisting of **all partial derivatives**

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$$

$$f(x, y) = 4x^2 + y^2$$
$$\nabla f = \langle 8x, 2y \rangle$$

Gradient

- The gradient of a function $f(x_1, x_2, \ldots x_n)$ is a **vector**, consisting of **all partial derivatives**

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$$

$$f(x, y) = 4x^2 + y^2$$
$$\nabla f = \langle 8x, 2y \rangle$$

- The gradient is perpendicular to the *level curve* at a point (next slide)

Gradient

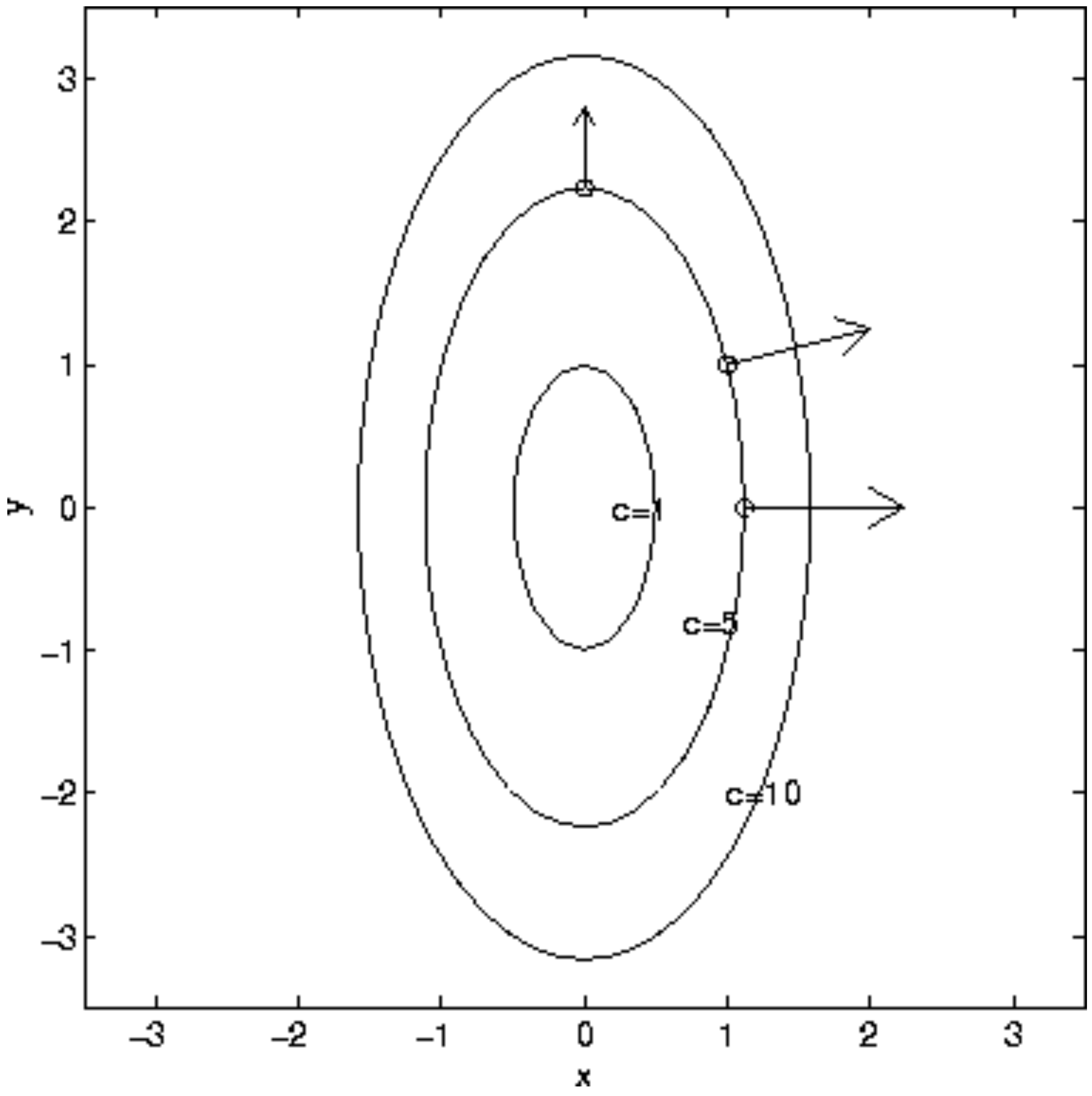
- The gradient of a function $f(x_1, x_2, \ldots x_n)$ is a **vector**, consisting of **all partial derivatives**

$$\nabla f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right\rangle$$

$$f(x, y) = 4x^2 + y^2$$
$$\nabla f = \langle 8x, 2y \rangle$$

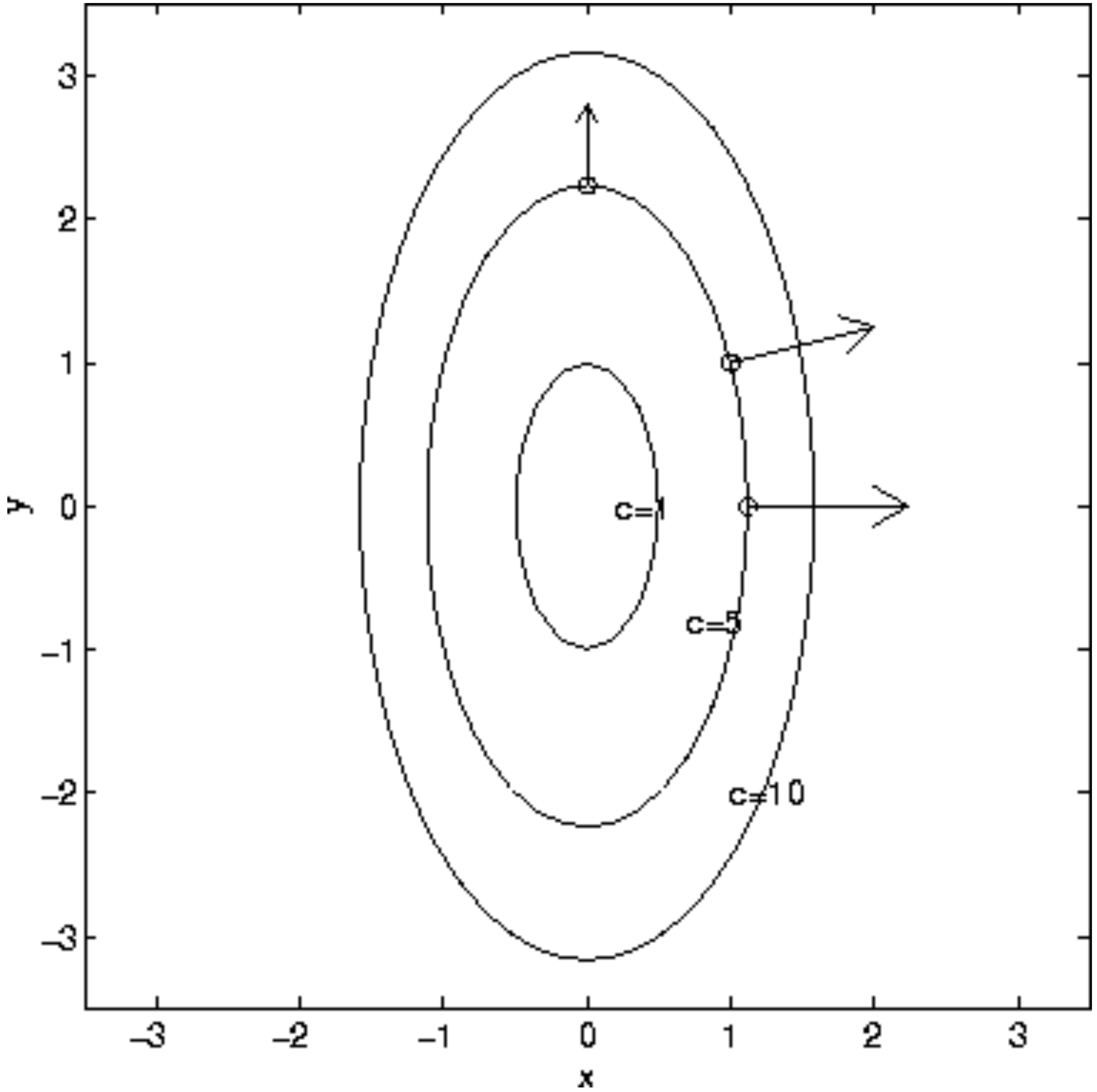
- The gradient is perpendicular to the *level curve* at a point (next slide)
- The gradient points in the direction of **greatest increase** of f

Gradient and Level Curves



Level curves: $f(x, y) = c$

Gradient and Level Curves

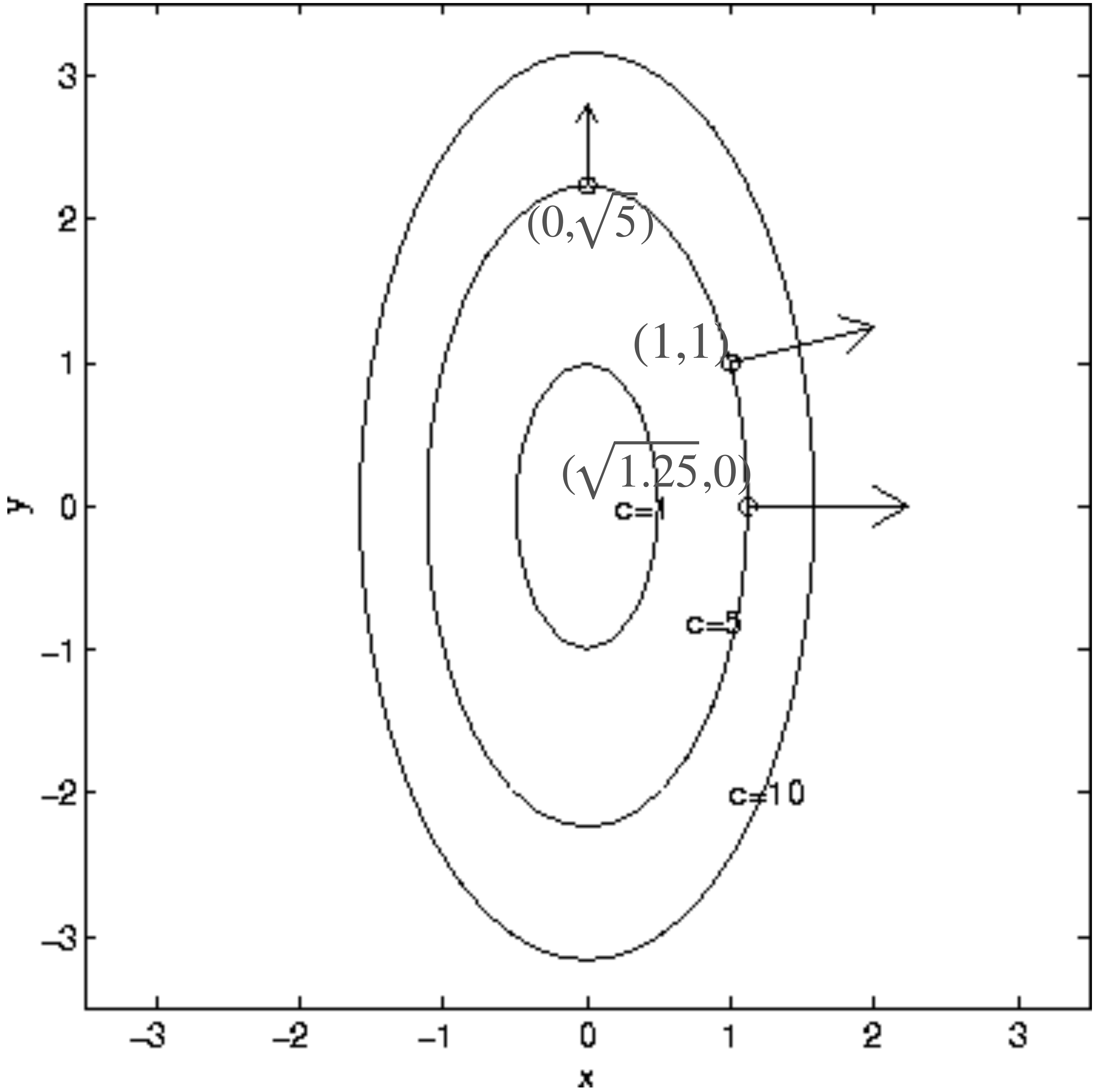


$$f(x, y) = 4x^2 + y^2$$

$$\nabla f = \langle 8x, 2y \rangle$$

Level curves: $f(x, y) = c$

Gradient and Level Curves

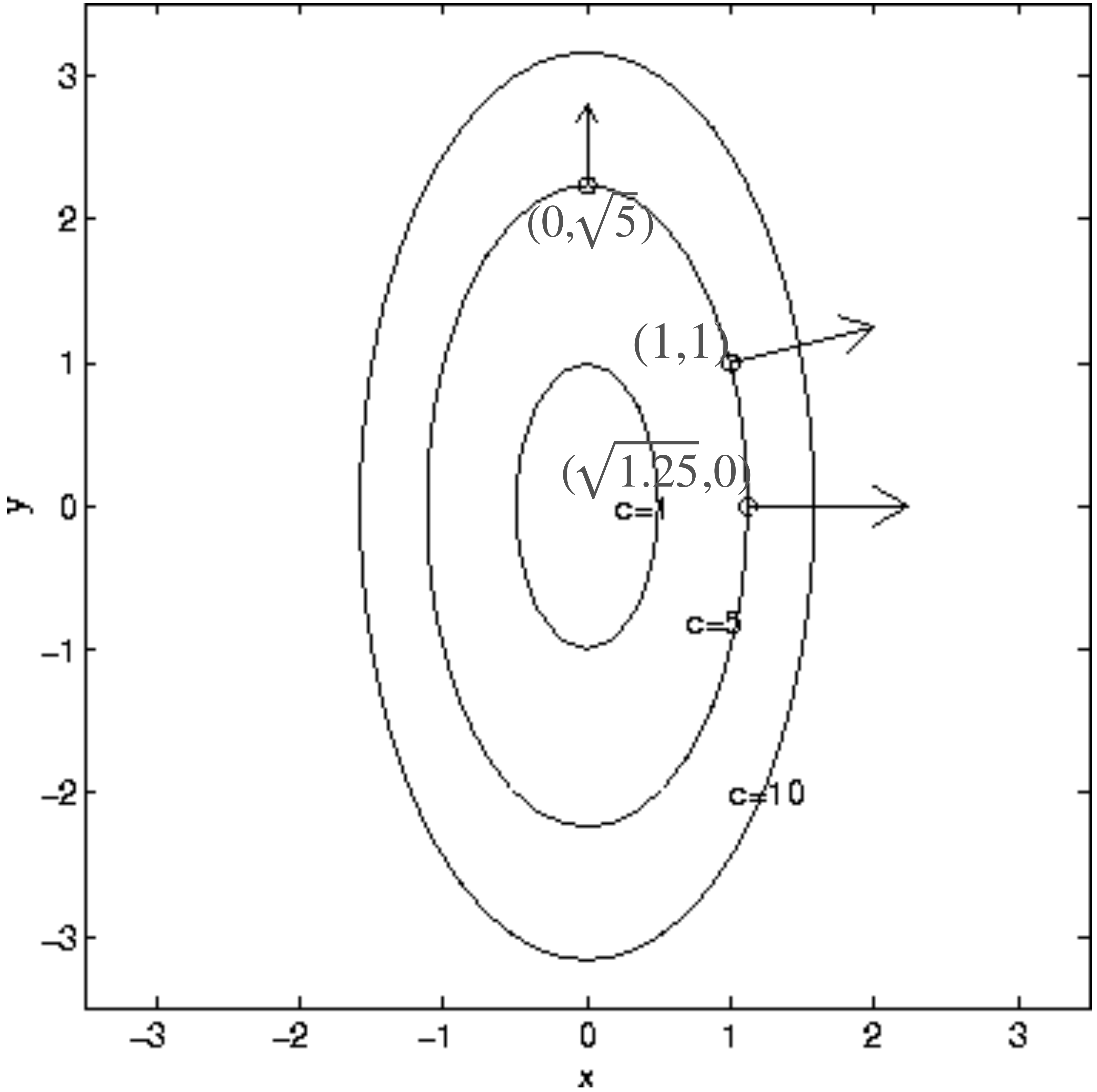


$$f(x, y) = 4x^2 + y^2$$

$$\nabla f = \langle 8x, 2y \rangle$$

Level curves: $f(x, y) = c$

Gradient and Level Curves

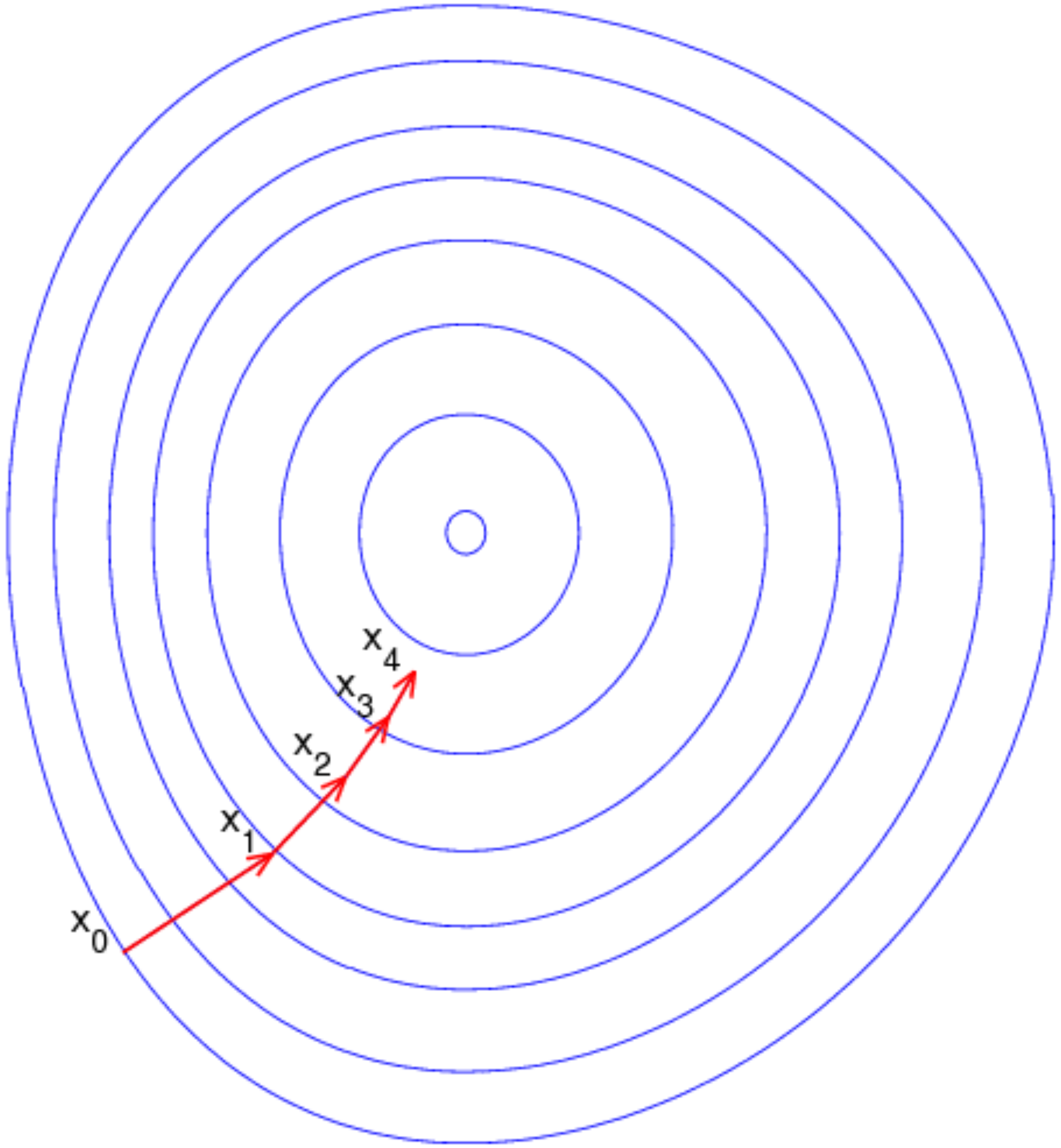


$$f(x,y) = 4x^2 + y^2$$
$$\nabla f = \langle 8x, 2y \rangle$$

Level curves: $f(x,y) = c$

Q: what are the actual gradients at those points?

Gradient Descent and Level Curves



source

Gradient Descent Algorithm

- Initialize θ_0
- Repeat until convergence:

$$\theta_{n+1} = \theta_n - \alpha \nabla \mathscr{L}(\hat{Y}(\theta_n), Y)$$

- **High learning rate**: big steps, may bounce and **“overshoot”** the target
- **Low learning rate**: small steps, smoother minimization of loss, but can be slow or **get stuck**

Gradient Descent Algorithm

- Initialize θ_0
- Repeat until convergence:

$$\theta_{n+1} = \theta_n - \underset{\text{Learning rate}}{\alpha} \nabla \mathscr{L}(\hat{Y}(\theta_n), Y)$$

- **High learning rate**: big steps, may bounce and **"overshoot"** the target
- **Low learning rate**: small steps, smoother minimization of loss, but can be slow or **get stuck**

Stochastic Gradient Descent

Stochastic Gradient Descent

- The above is called **batch gradient descent**
  - Updates **based on entire dataset at once**
  - Expensive, and slow; **does not scale well**

Stochastic Gradient Descent

- The above is called **batch gradient descent**
    - Updates **based on entire dataset at once**
    - Expensive, and slow; **does not scale well**
- Stochastic gradient descent: **single example at a time**; very noisy estimate of true gradient

Stochastic Gradient Descent

- The above is called **batch gradient descent**
  - Updates **based on entire dataset at once**
  - Expensive, and slow; **does not scale well**
- Stochastic gradient descent: **single example at a time**; very noisy estimate of true gradient
- **Mini-batch** gradient descent:
  - Break the data into “mini-batches”: **small chunks** of the data
  - Compute gradients and update parameters for each batch
  - Mini-batch of size 1 = single example = stochastic gradient descent
  - A noisy estimate of the true gradient, but works well in practice; more parameter updates

Stochastic Gradient Descent

- The above is called **batch gradient descent**
  - Updates **based on entire dataset at once**
  - Expensive, and slow; **does not scale well**
- Stochastic gradient descent: **single example at a time**; very noisy estimate of true gradient
- **Mini-batch** gradient descent:
  - Break the data into "mini-batches": **small chunks** of the data
  - Compute gradients and update parameters for each batch
  - Mini-batch of size 1 = single example = stochastic gradient descent
  - A noisy estimate of the true gradient, but works well in practice; more parameter updates
- **Epoch**: one pass through the whole training data

Stochastic Gradient Descent

```
initialize parameters / build model

for each epoch:

 data = shuffle(data)
 batches = make_batches(data)

 for each batch in batches:

  outputs = model(batch)
  loss = loss_fn(outputs, true_outputs)
  compute gradients
  update parameters
```