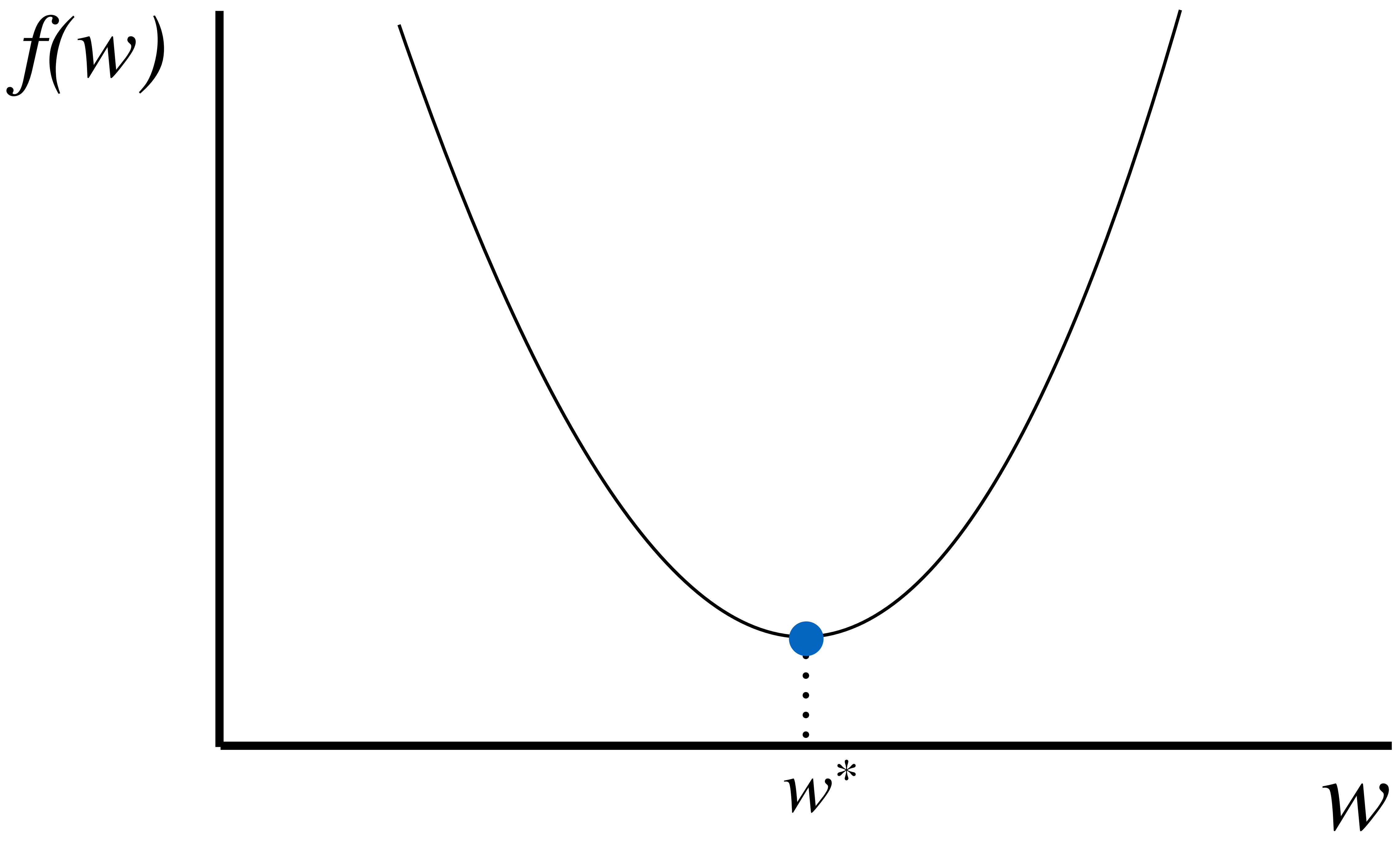
Gradient Descent

Linear Regression Optimization

Goal: Find $\mathbf{w}^*$ that minimizes

$$f(\mathbf{w}) = ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

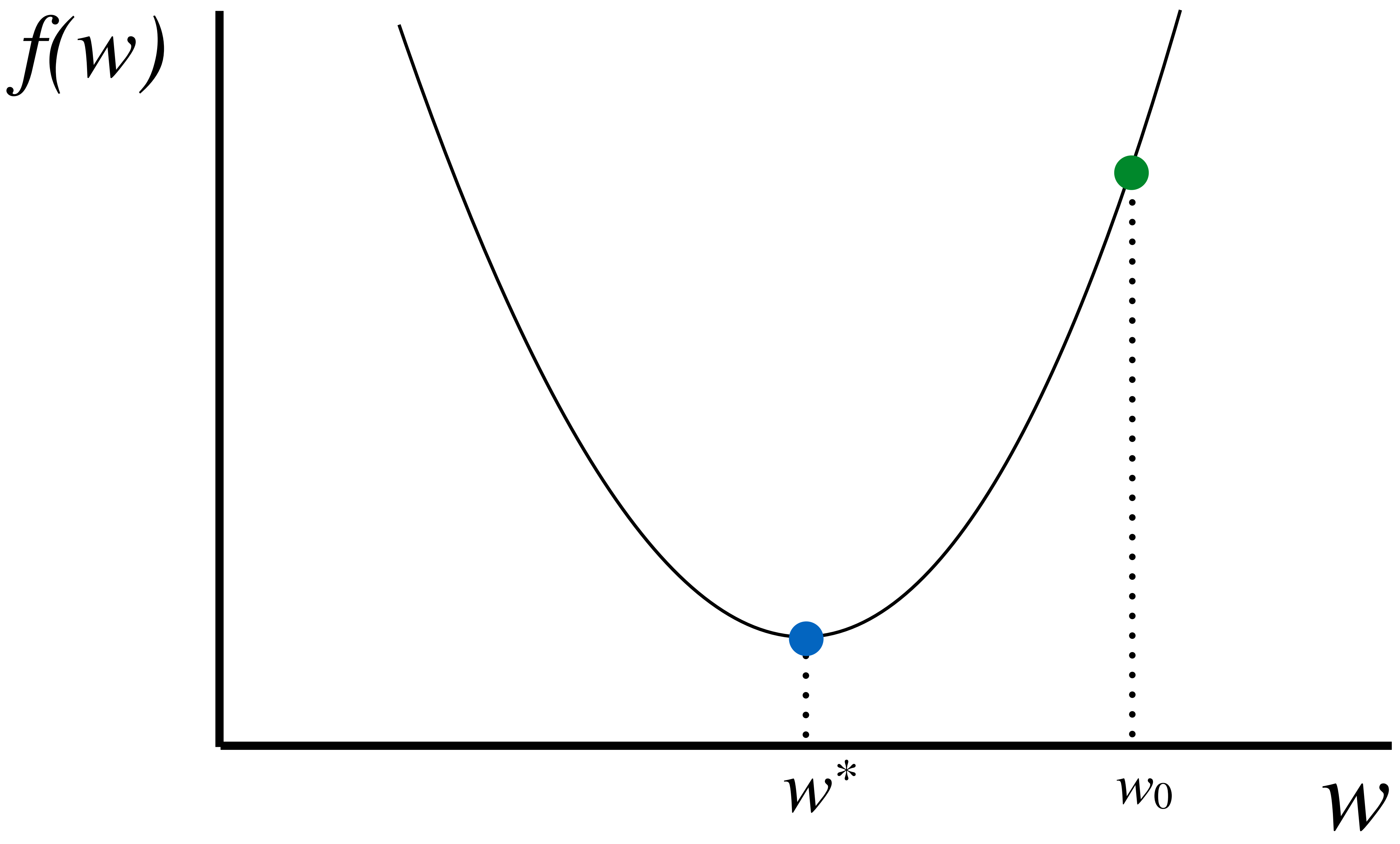
- Closed form solution exists
- Gradient Descent is iterative (Intuition: go downhill!)



Scalar objective: $f(w) = ||w\mathbf{x} - \mathbf{y}||_2^2 = \sum_{j=1}^{n} (wx^{(j)} - y^{(j)})^2$

Gradient Descent

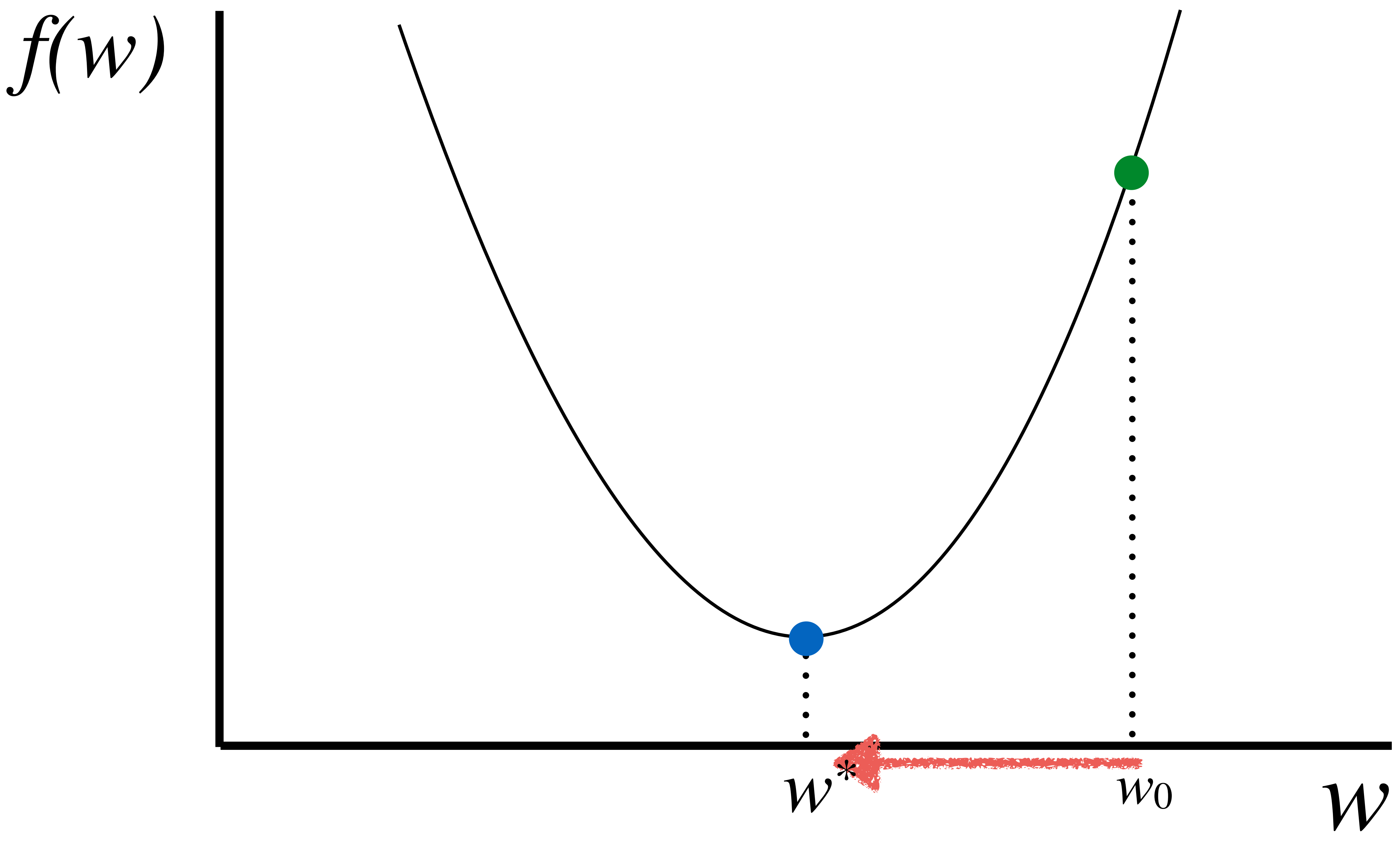
Start at a random point



Gradient Descent

Start at a random point

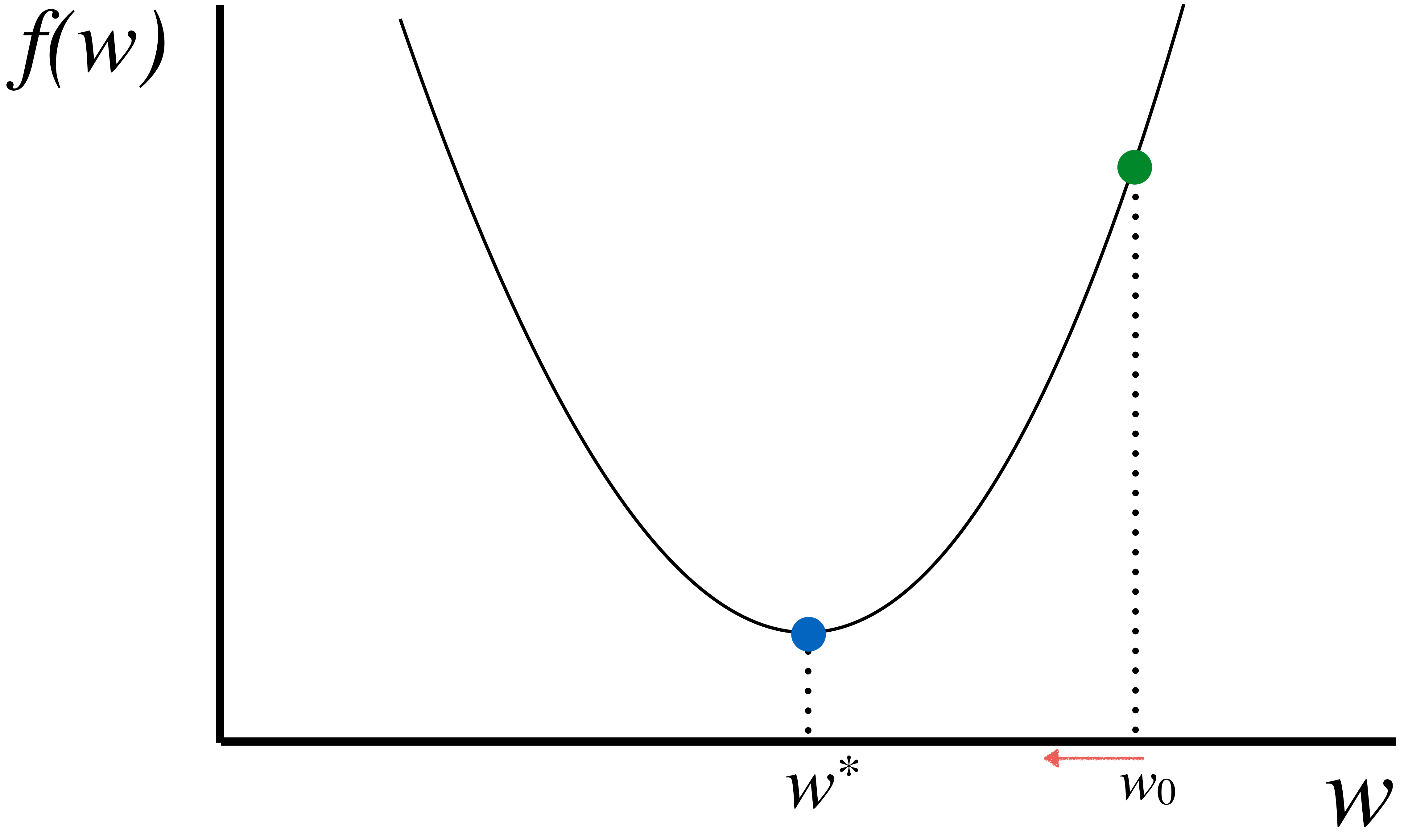
Determine a descent direction



Gradient Descent

Start at a random point

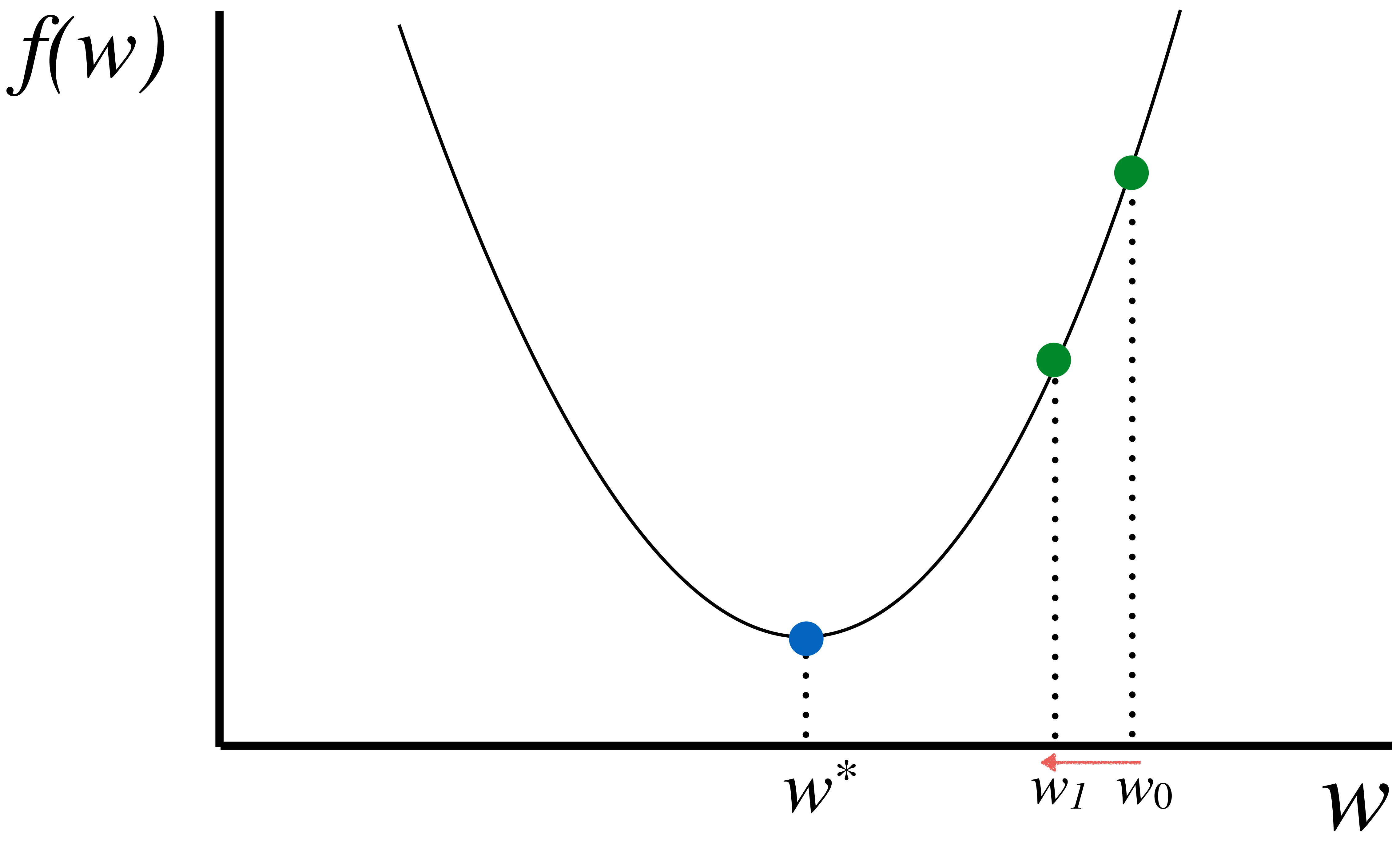
Determine a descent direction
Choose a step size



Gradient Descent

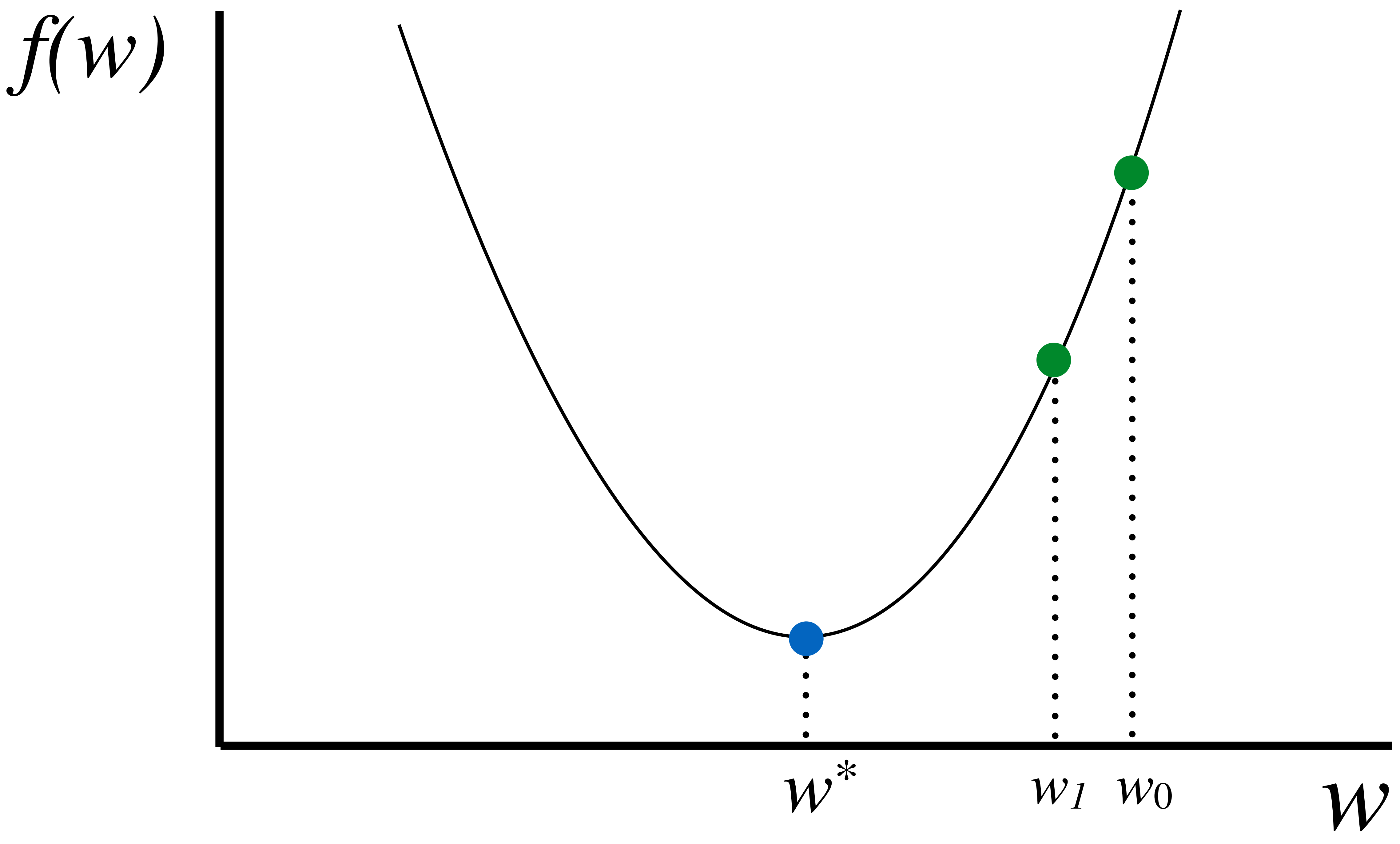
Start at a random point

Determine a descent direction
Choose a step size
Update



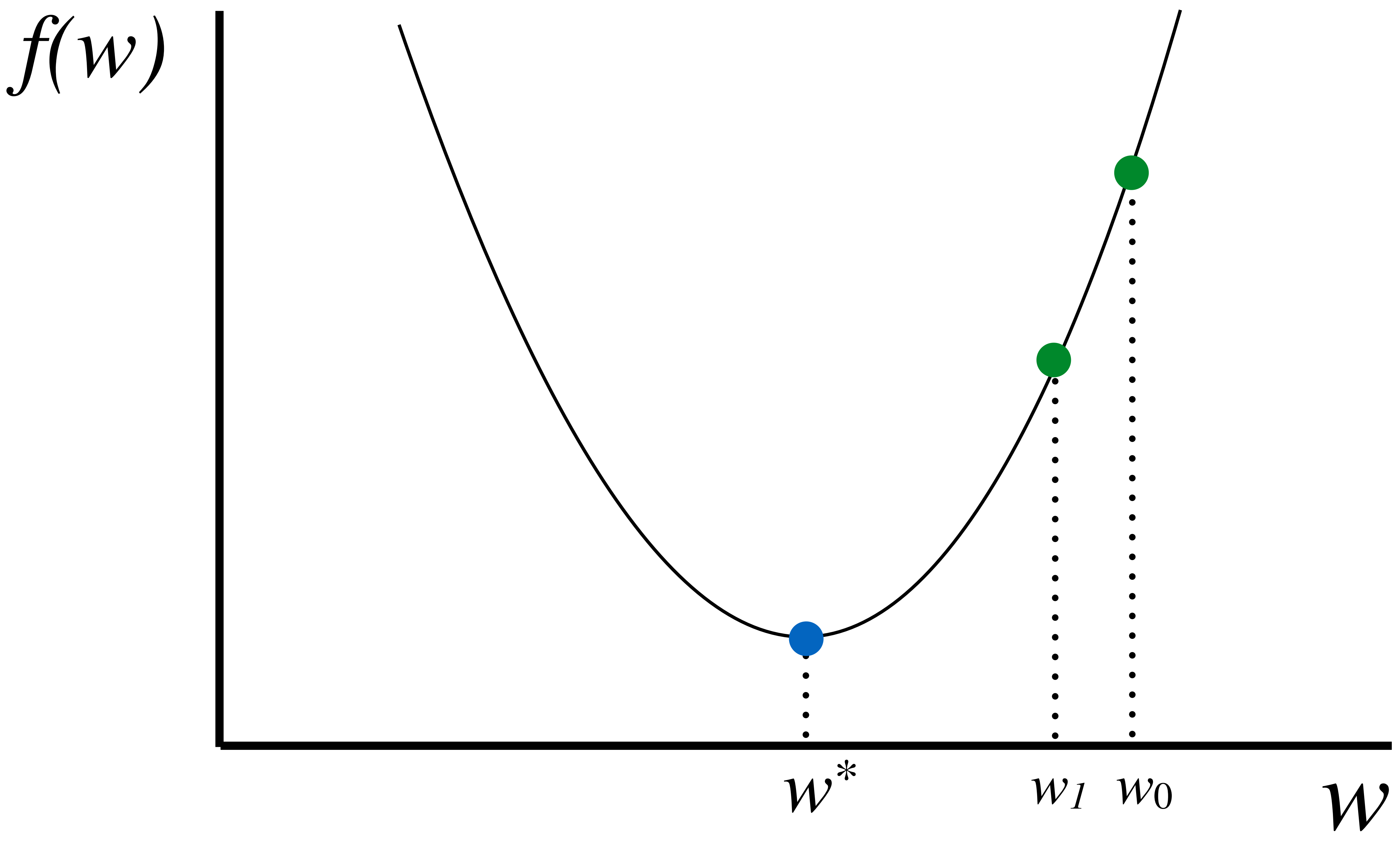
Gradient Descent

Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied



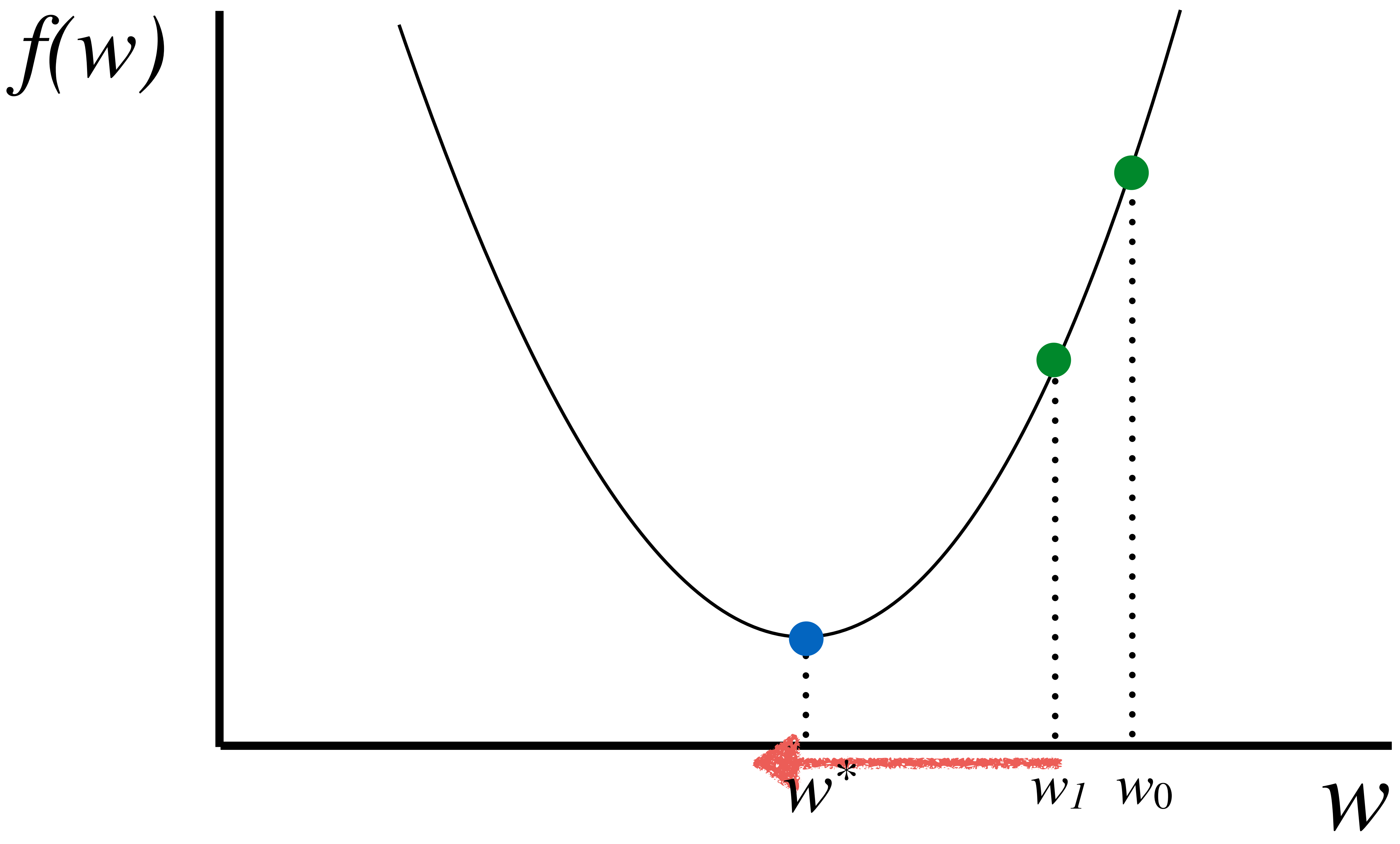
Gradient Descent

Start at a random point
Repeat
Determine a descent direction
Choose a step size
Update
Until stopping criterion is satisfied



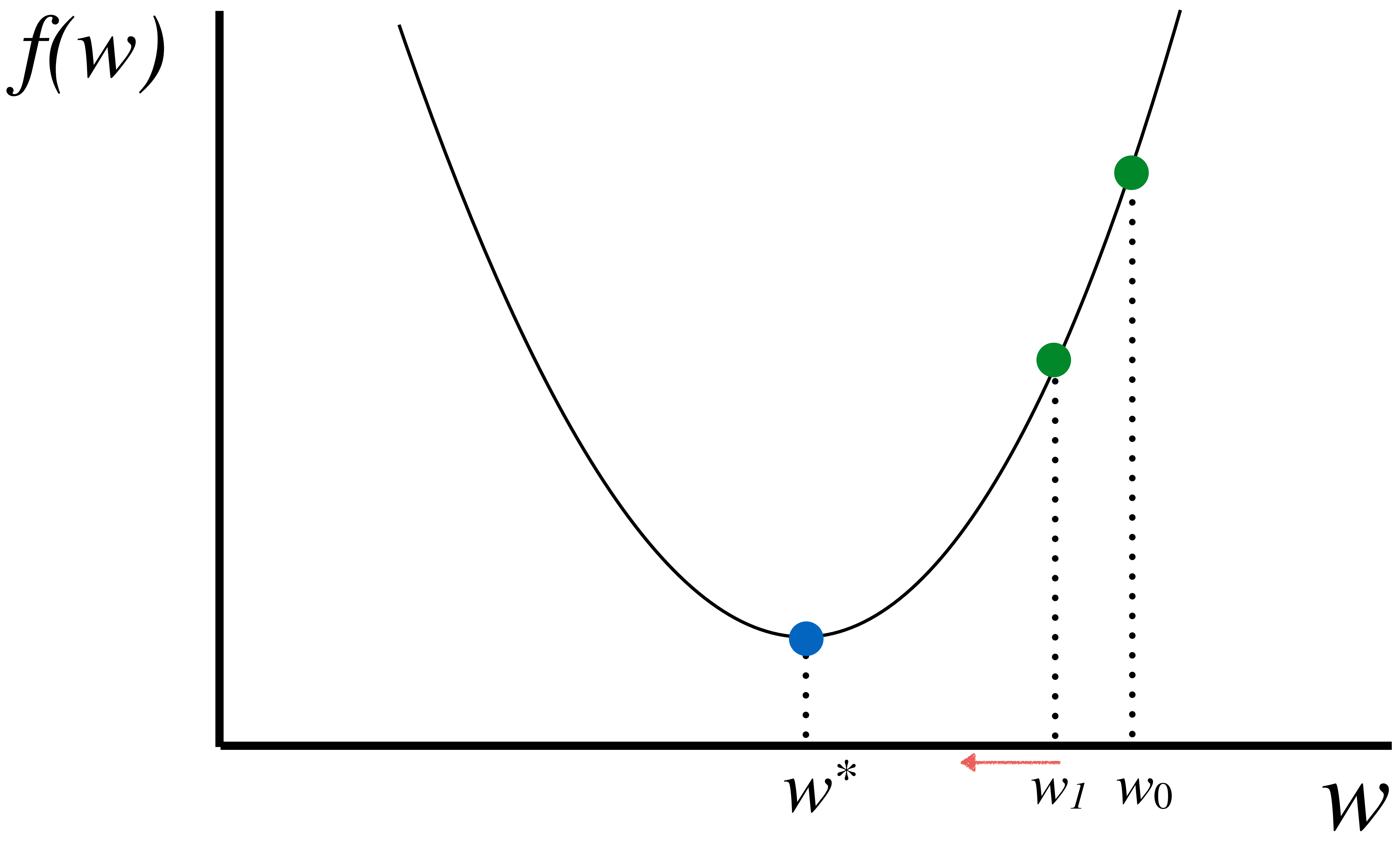
Gradient Descent

Start at a random point
Repeat
  Determine a descent direction
  Choose a step size
  Update
Until stopping criterion is satisfied



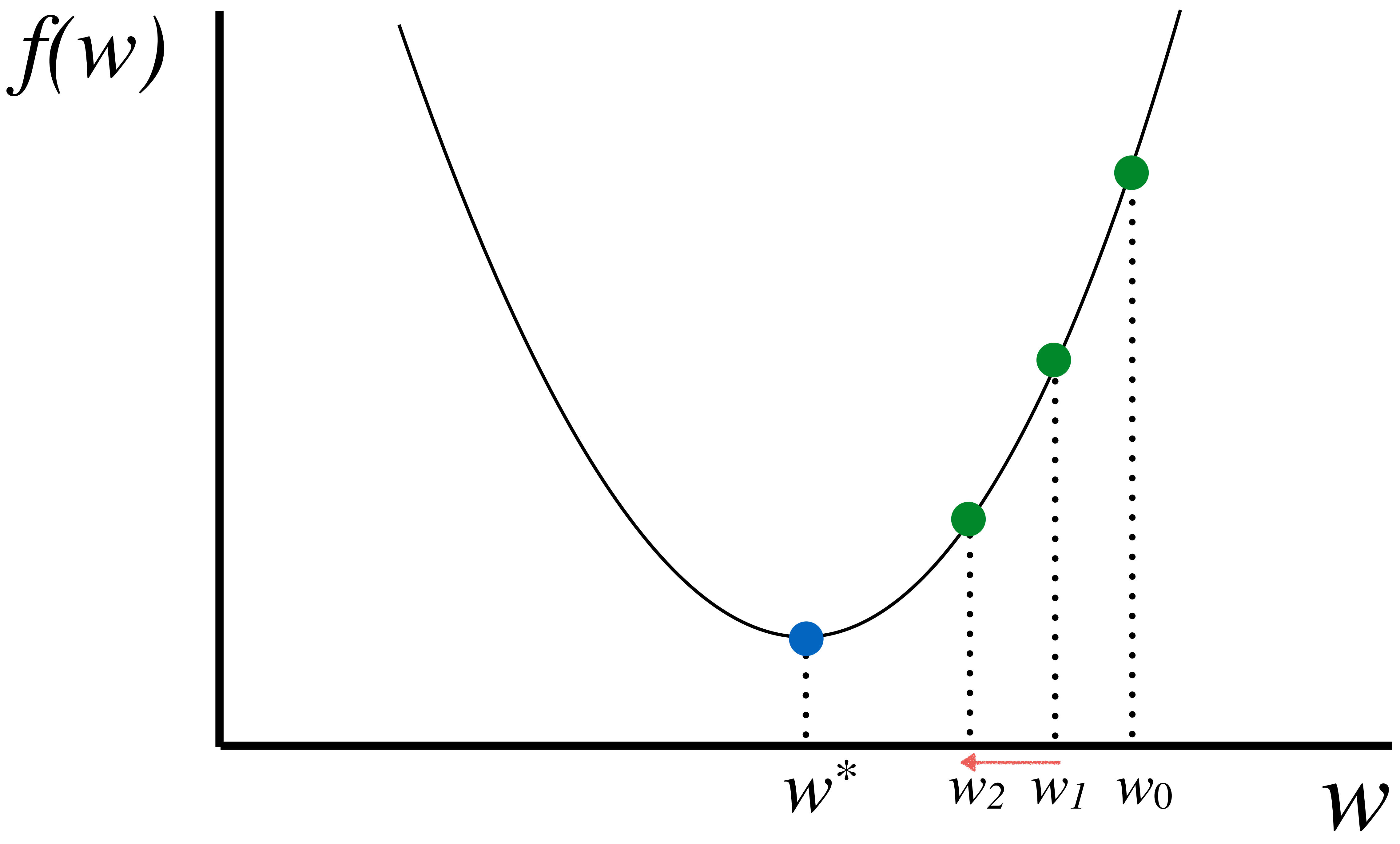
Gradient Descent

Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied



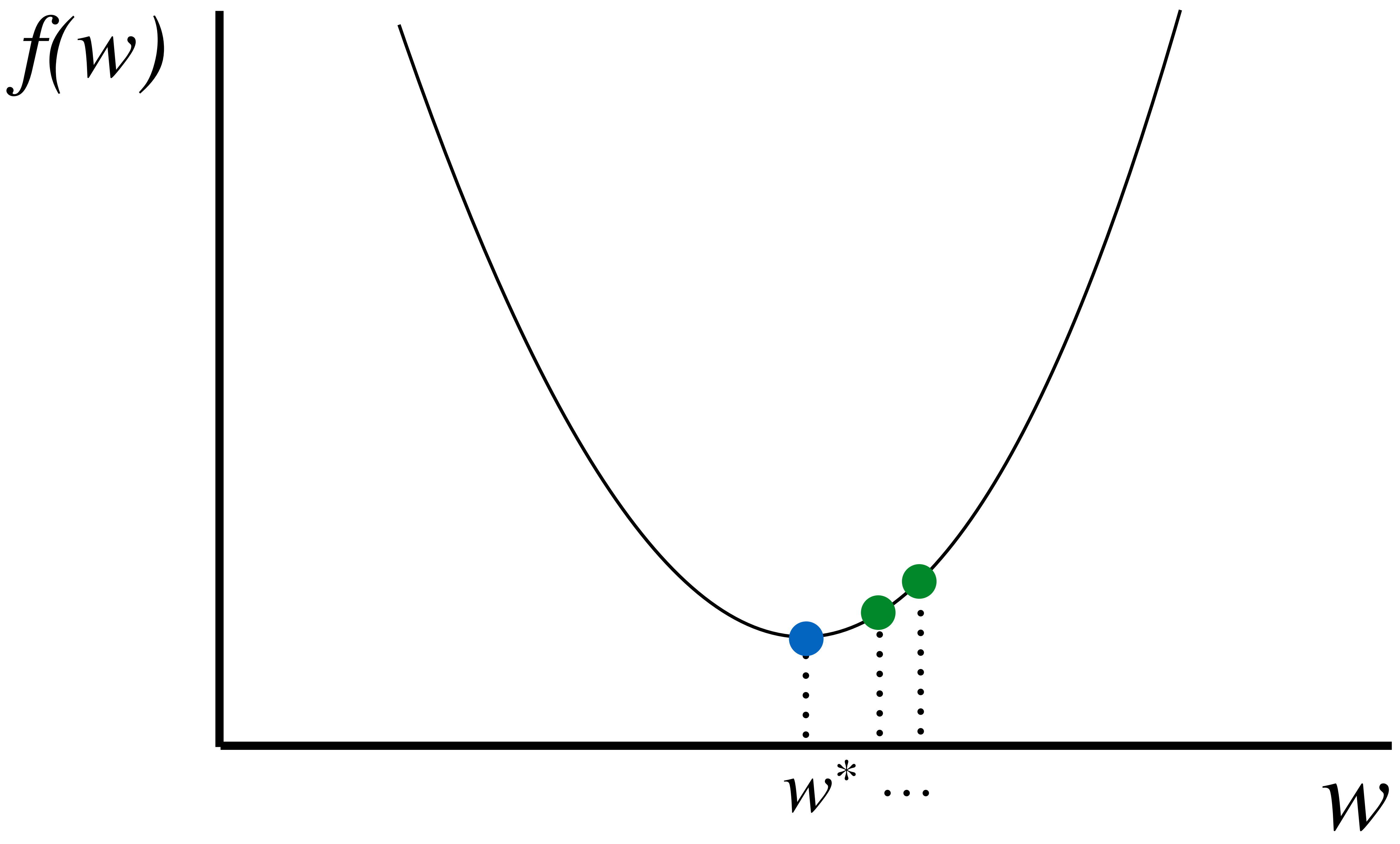
Gradient Descent

Start at a random point
Repeat
Determine a descent direction
Choose a step size
Update
Until stopping criterion is satisfied



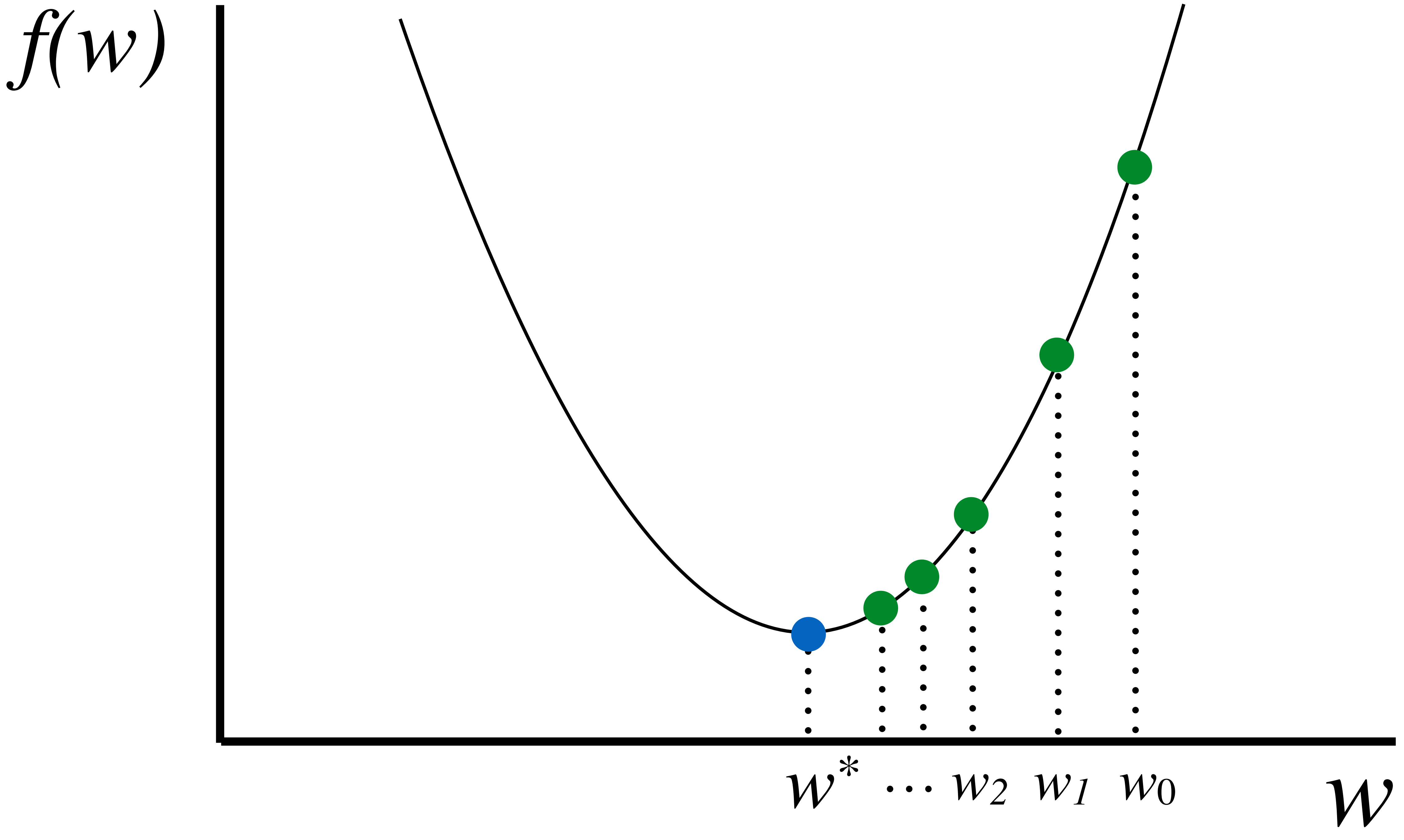
Gradient Descent

Start at a random point
Repeat
  Determine a descent direction
  Choose a step size
  Update
Until stopping criterion is satisfied

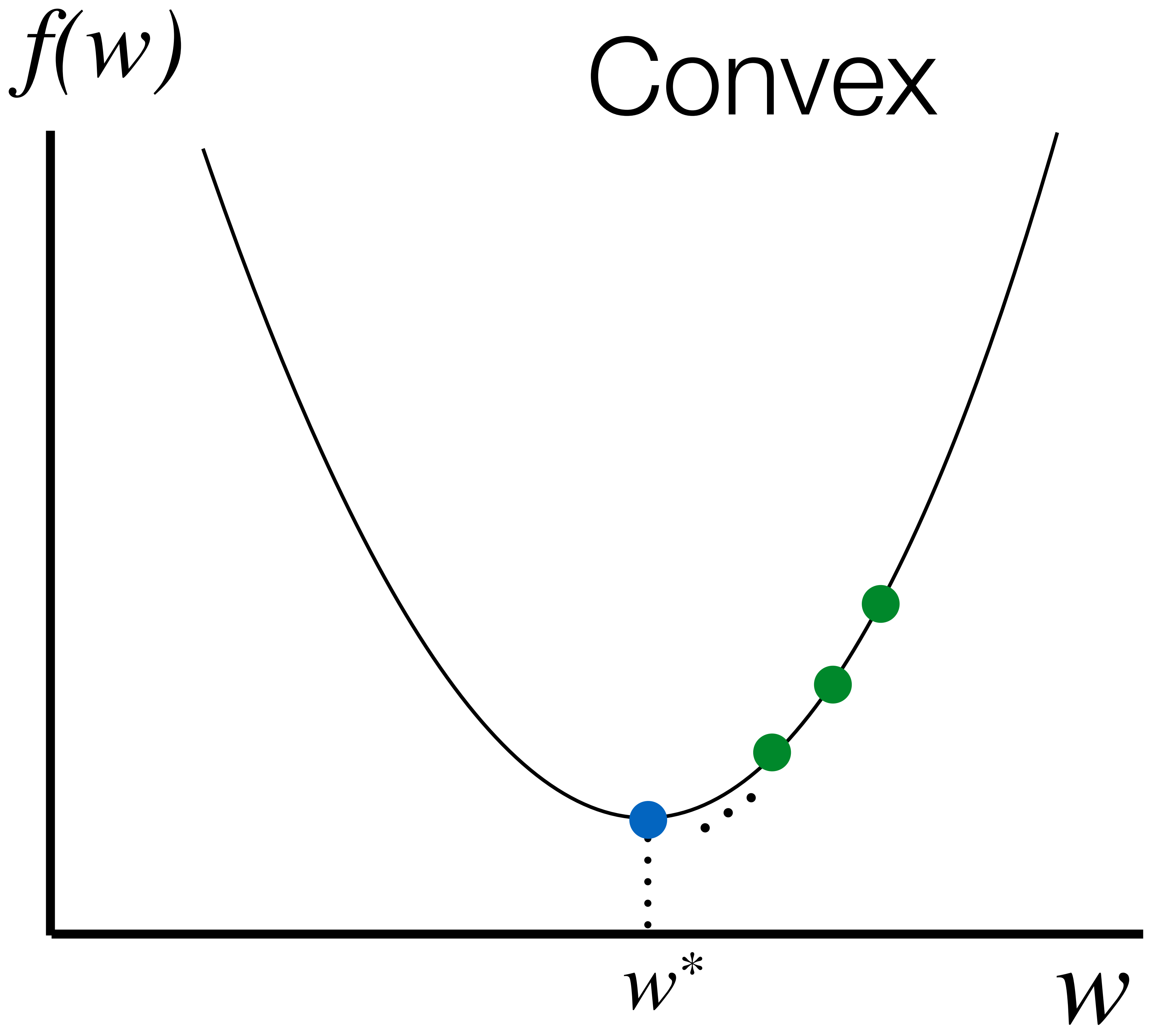


Gradient Descent

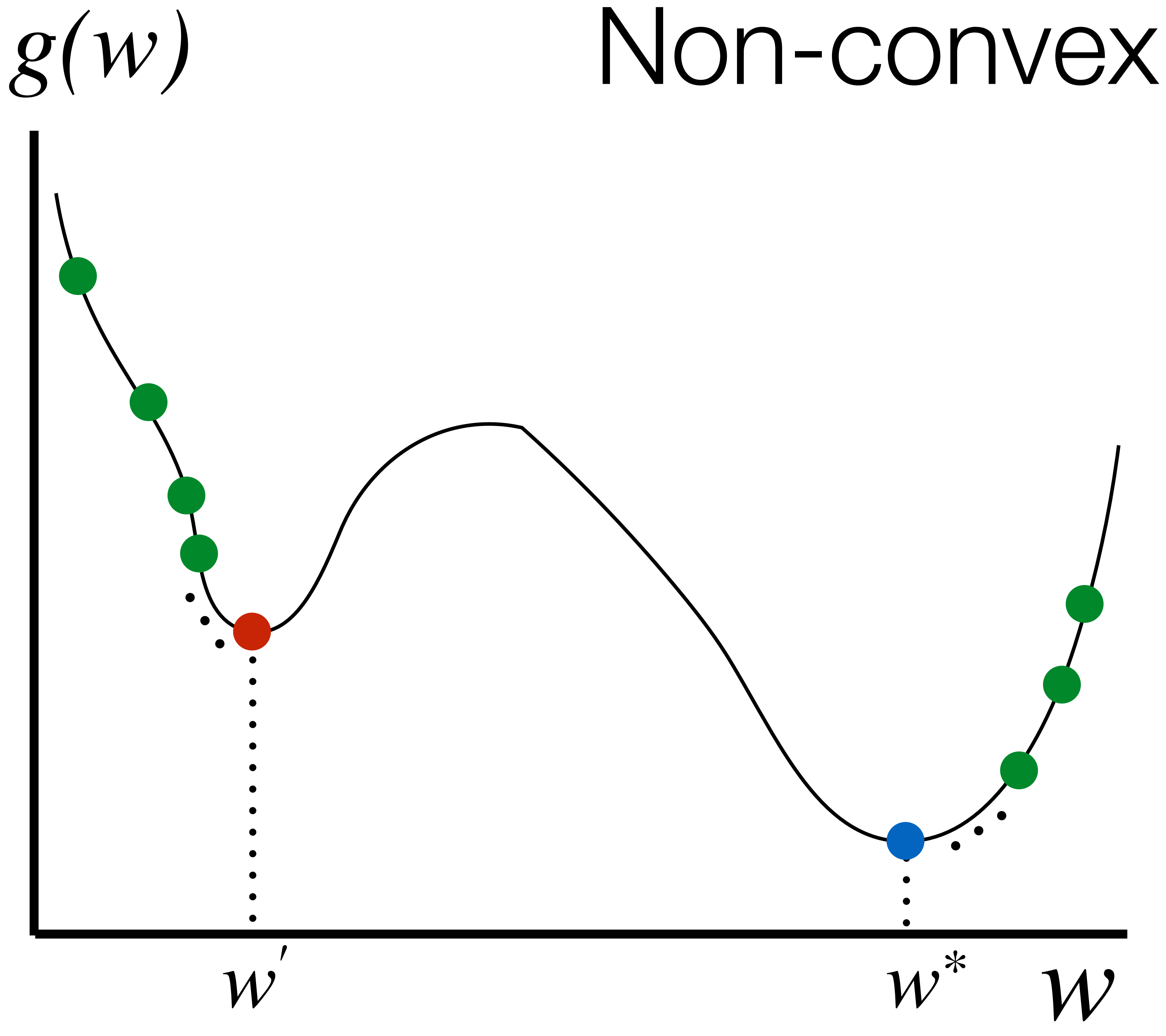
Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied



Where Will We Converge?



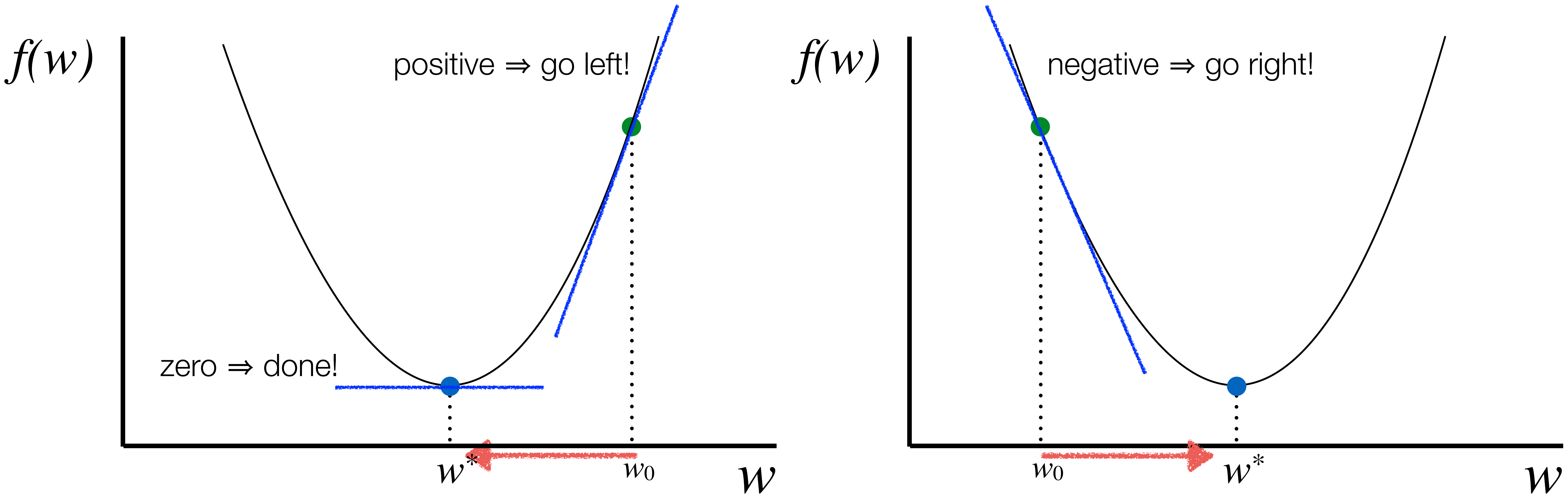
Any local minimum is a global minimum



Multiple local minima may exist

Least Squares, Ridge Regression and Logistic Regression are all convex!

Choosing Descent Direction (1D)



We can only move in two directions

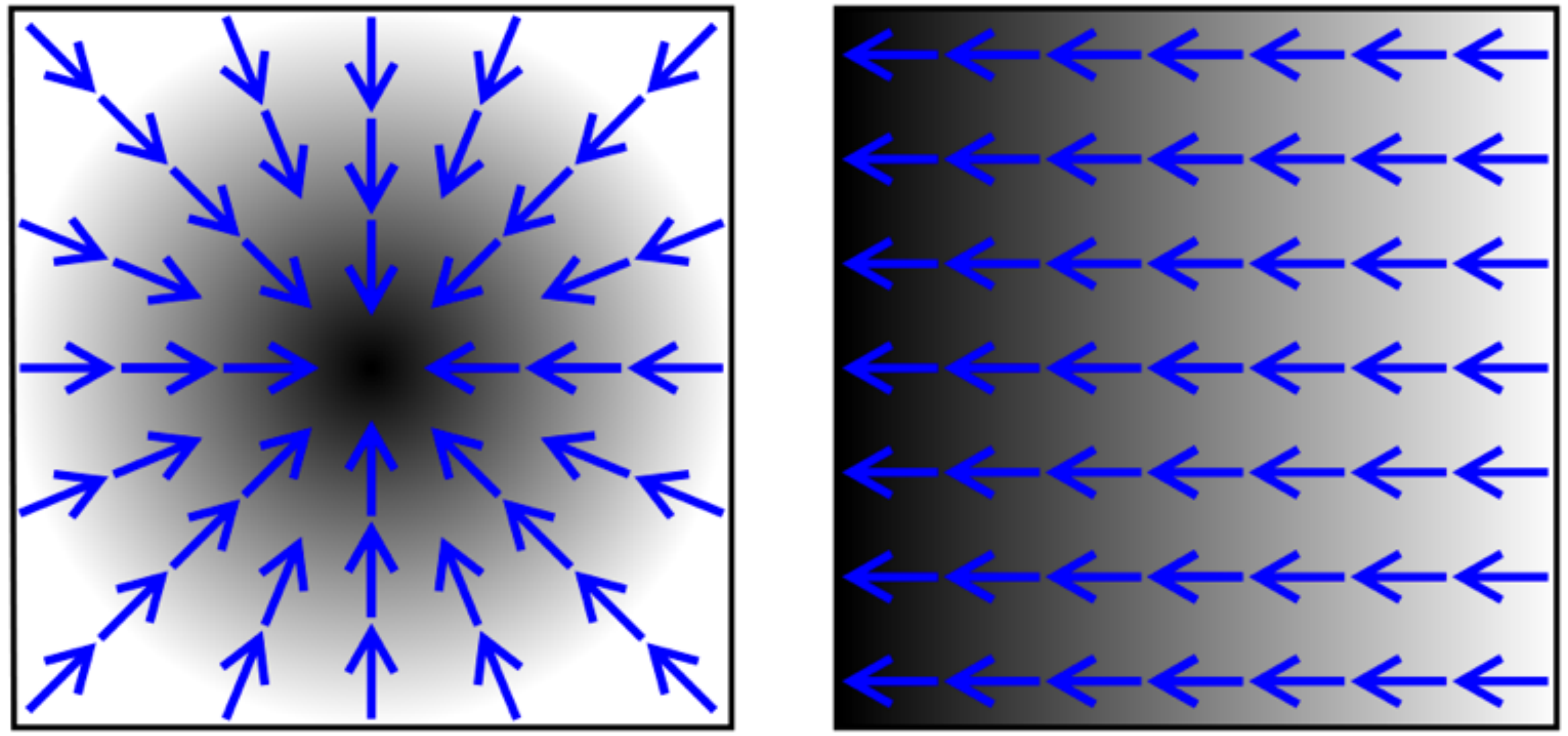
Negative slope is direction of descent!

Step Size

Update Rule: $w_{i+1} = w_i - \alpha_i \frac{df}{dw}(w_i)$

Negative Slope

Choosing Descent Direction



"Gradient2" by Sarang. Licensed under CC BY-SA 2.5 via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Gradient2.svg#/media/File:Gradient2.svg

2D Example:

- Function values are in black/white and black represents higher values
- Arrows are gradients

We can move anywhere in $\mathbb{R}^d$

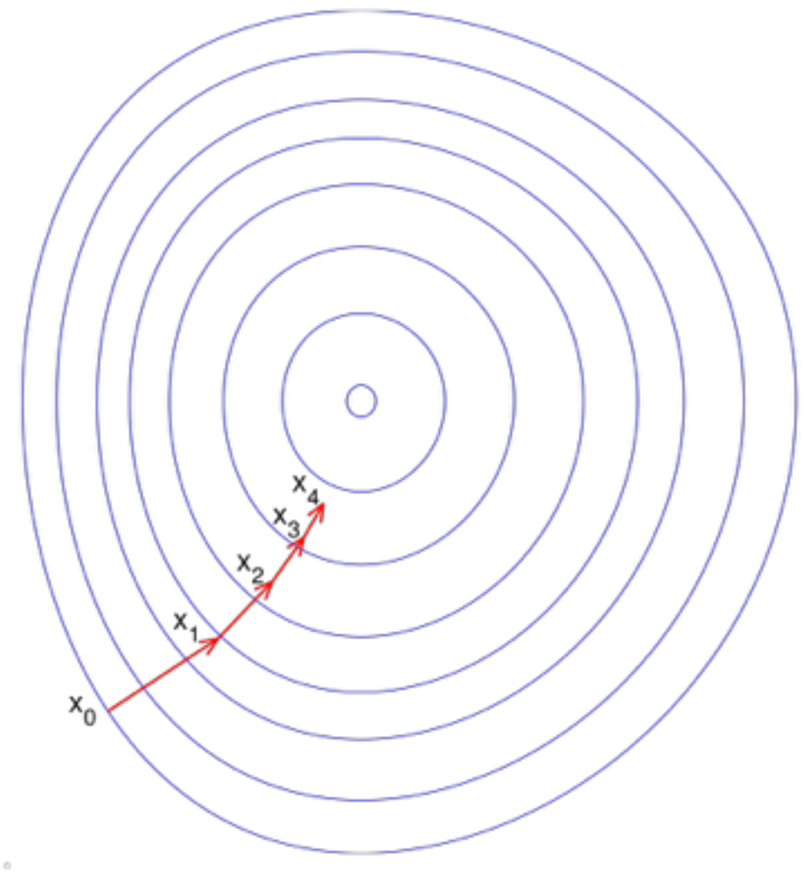
Negative gradient is direction of *steepest* descent!

Step Size

Update Rule: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$

Negative Slope

Gradient Descent for Least Squares



Update Rule: $w_{i+1} = w_i - \alpha_i \dfrac{df}{dw}(w_i)$

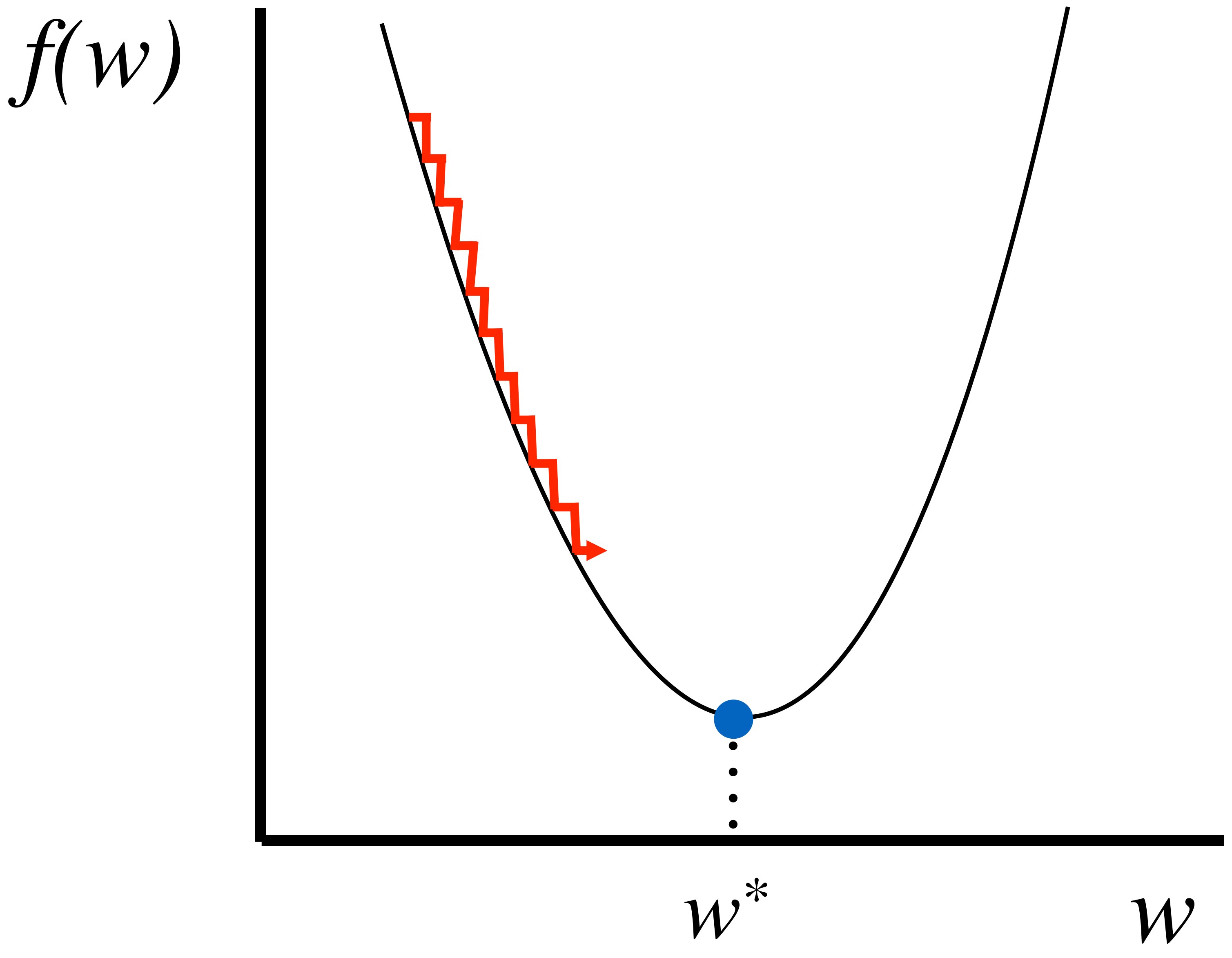
Scalar objective: $f(w) = ||w\mathbf{x} - \mathbf{y}||_2^2 = \sum_{j=1}^{n} (wx^{(j)} - y^{(j)})^2$

Derivative: $\dfrac{df}{dw}(w) = 2 \sum_{j=1}^{n} (wx^{(j)} - y^{(j)})x^{(j)}$
(chain rule)

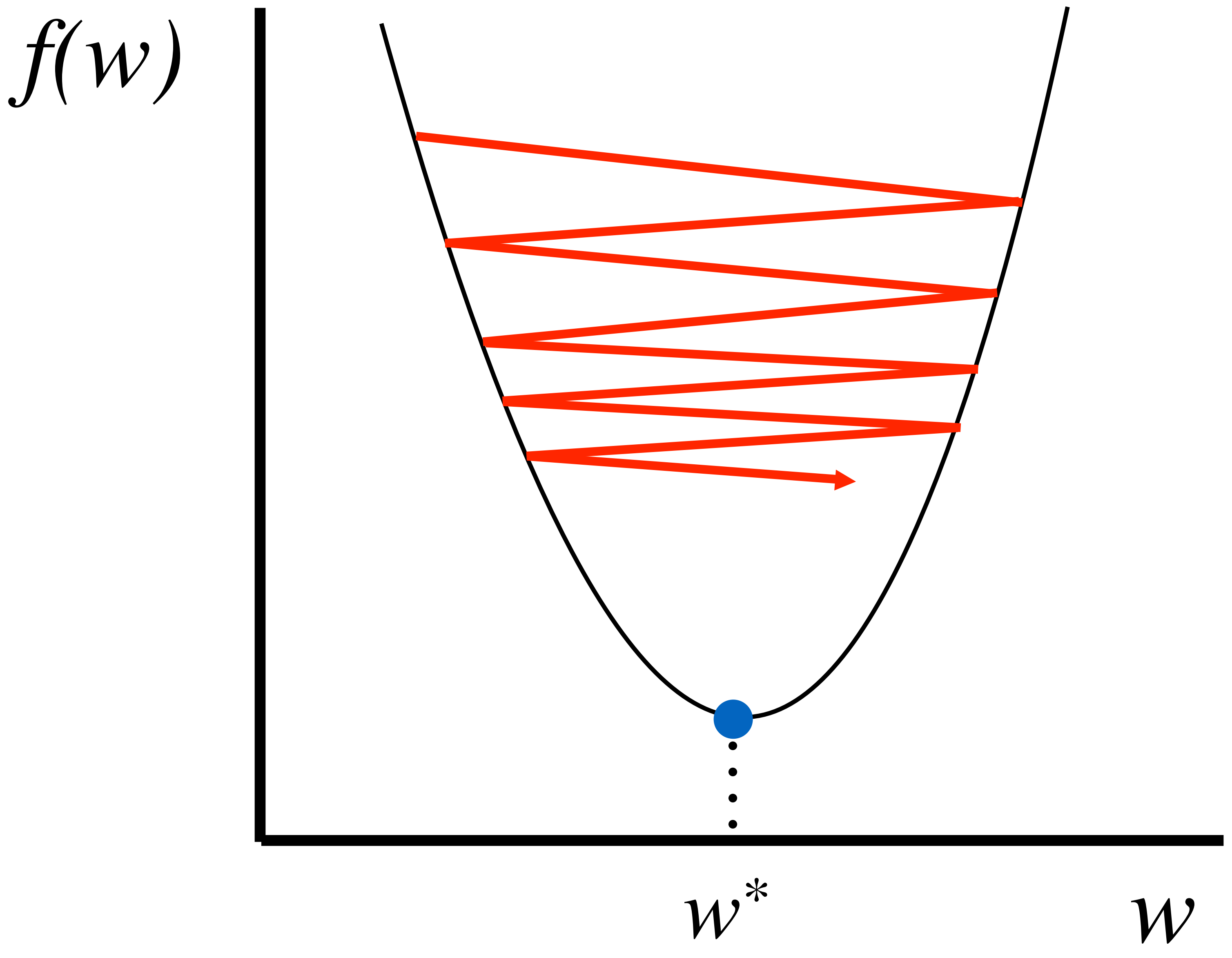
Scalar Update: $w_{i+1} = w_i - \alpha_i \sum_{j=1}^{n} (w_i x^{(j)} - y^{(j)})x^{(j)}$
(2 absorbed in α)

Vector Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$

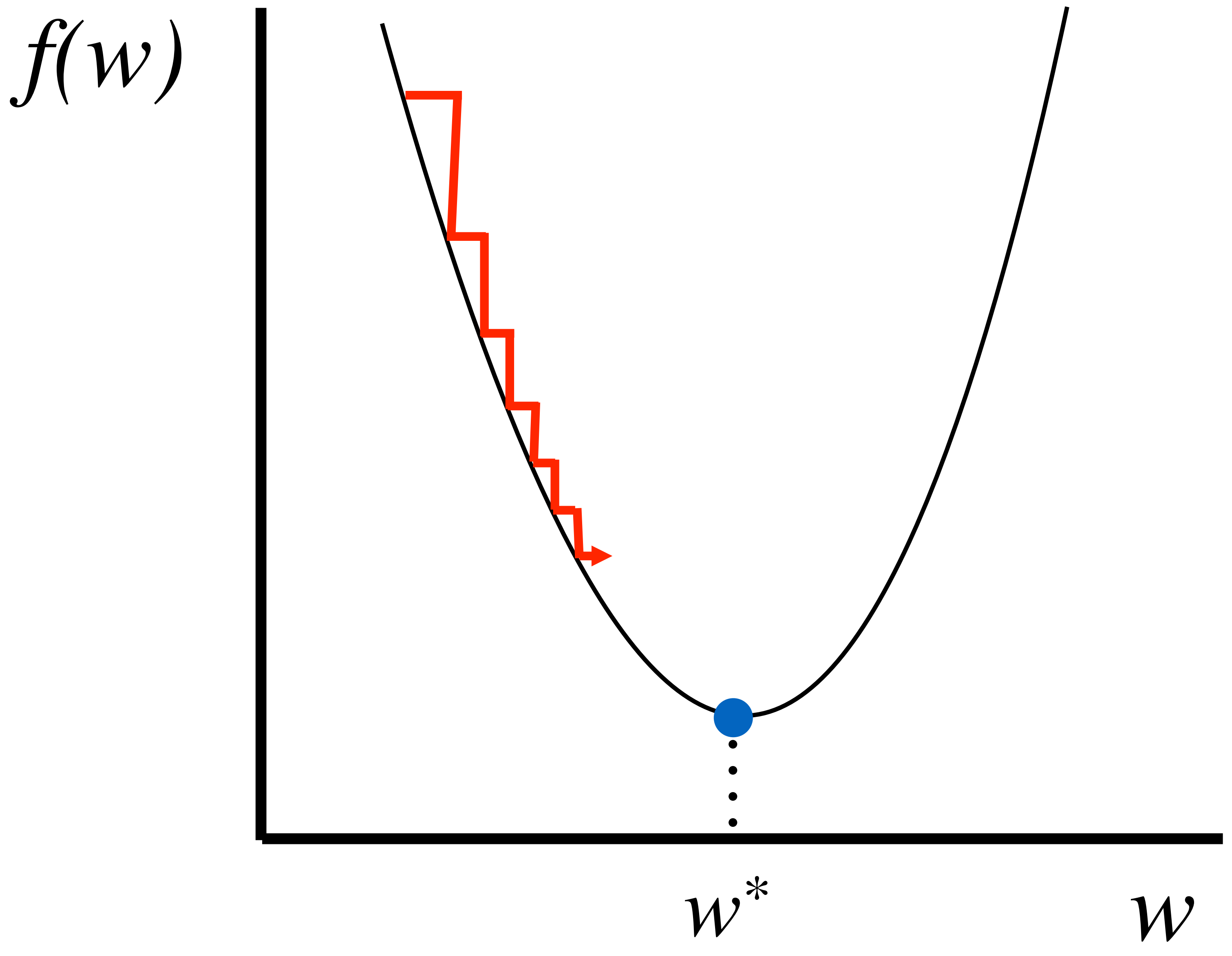
Choosing Step Size



Too small: converge very slowly



Too big: overshoot and even diverge



Reduce size over time

Theoretical convergence results for various step sizes

Constant

A common step size is $\alpha_i = \dfrac{\alpha}{n\sqrt{i}}$

Iteration #

Training Points

Parallel Gradient Descent for Least Squares

Vector Update: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} \underline{(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}}$

Compute summands in parallel!

note: workers must all have $\mathbf{w}_i$

Example: n = 6; 3 workers

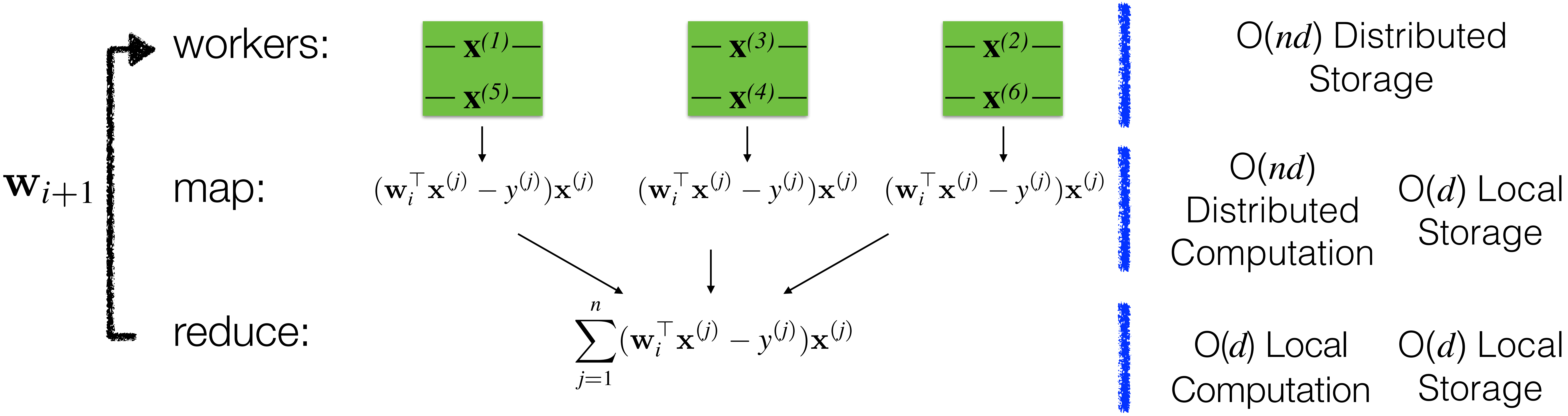
workers: $\mathbf{x}^{(1)}$, $\mathbf{x}^{(5)}$ | $\mathbf{x}^{(3)}$, $\mathbf{x}^{(4)}$ | $\mathbf{x}^{(2)}$, $\mathbf{x}^{(6)}$ — O(nd) Distributed Storage

map: $(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ | $(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ | $(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ — O(nd) Distributed Computation; O(d) Local Storage

reduce: $\sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ — O(d) Local Computation; O(d) Local Storage

$\mathbf{w}_{i+1}$

```python
> for i in range(numIters):
      alpha_i = alpha / (n * np.sqrt(i+1))
      gradient = train.map(lambda lp: gradientSummand(w, lp))
                      .sum()
      w -= alpha_i * gradient
  return w
```



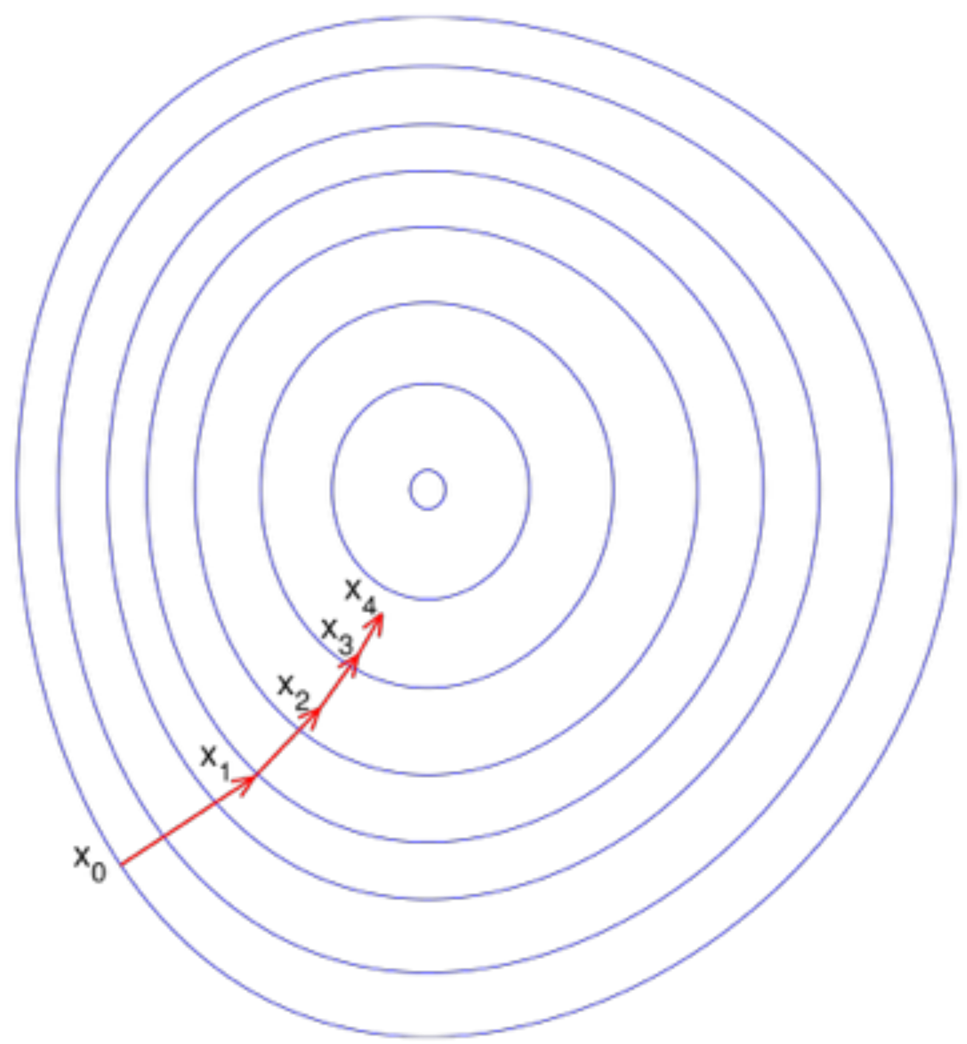
Gradient Descent Summary

Pros:

- Easily parallelized
- Cheap at each iteration
- Stochastic variants can make things even cheaper

Cons:

- Slow convergence (especially compared with closed-form)
- **Requires communication across nodes!**



Communication Hierarchy

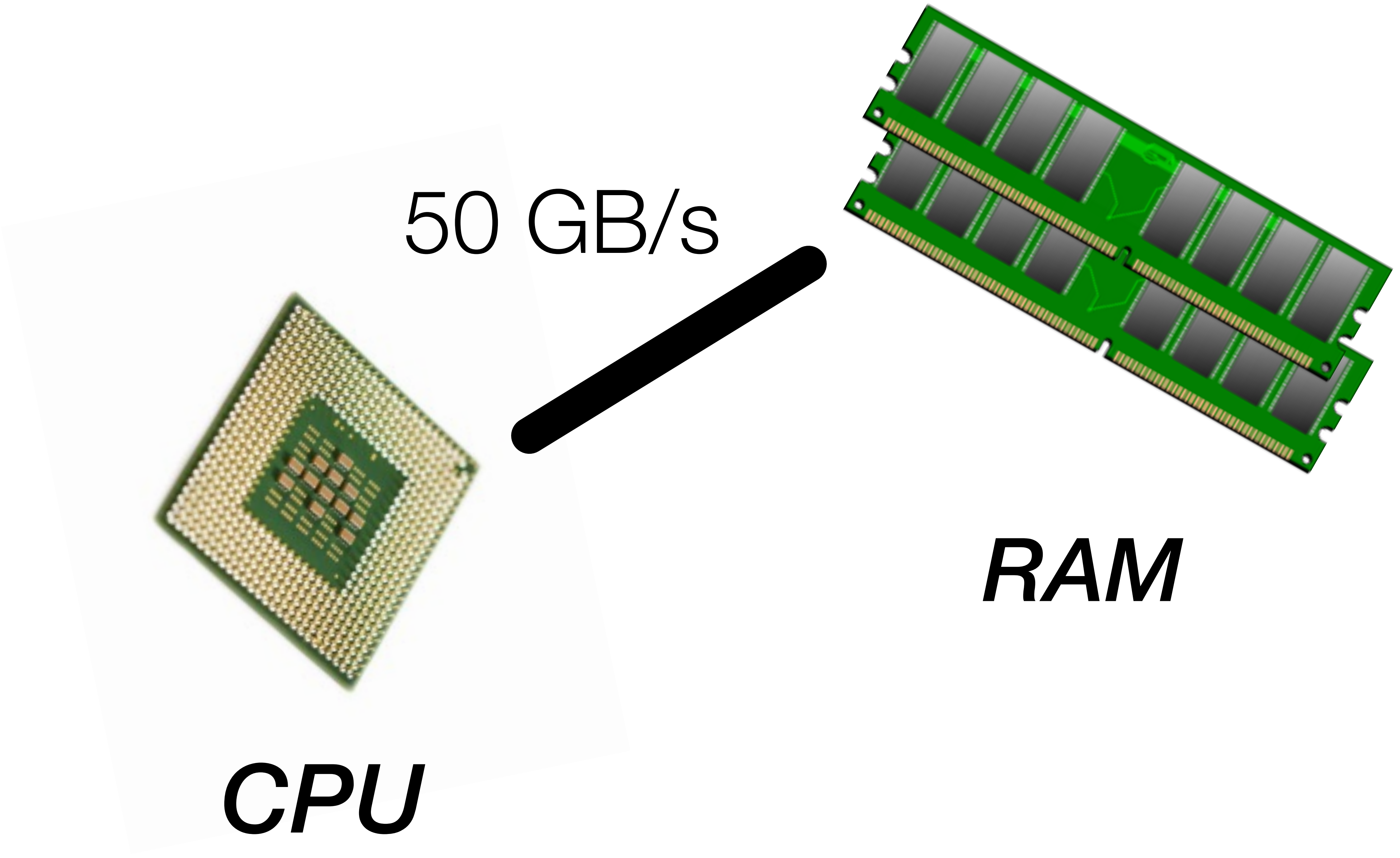
Communication Hierarchy

CPU

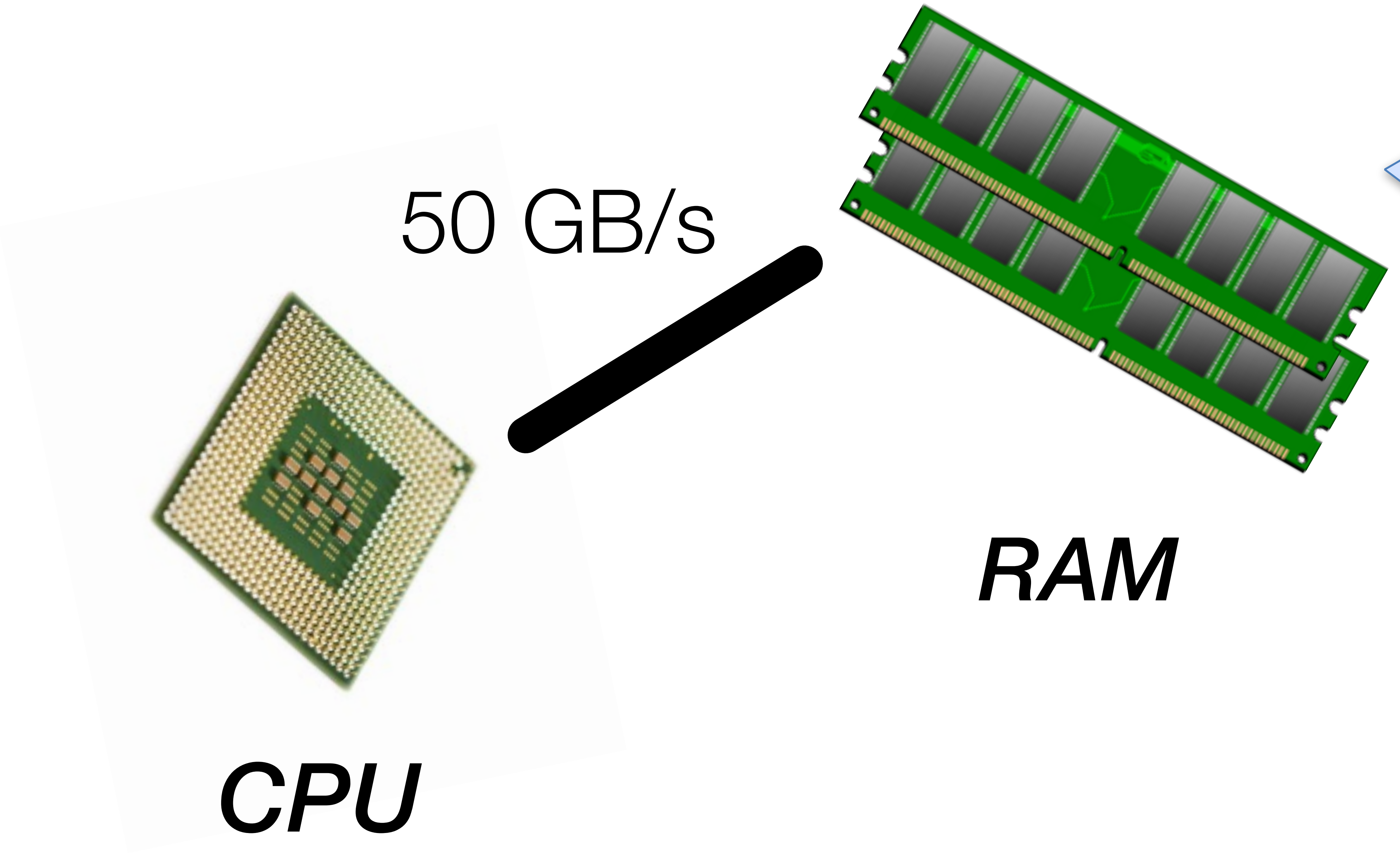
2 billion cycles/sec per core

Clock speeds not changing, but number of cores growing with Moore's Law

Communication Hierarchy



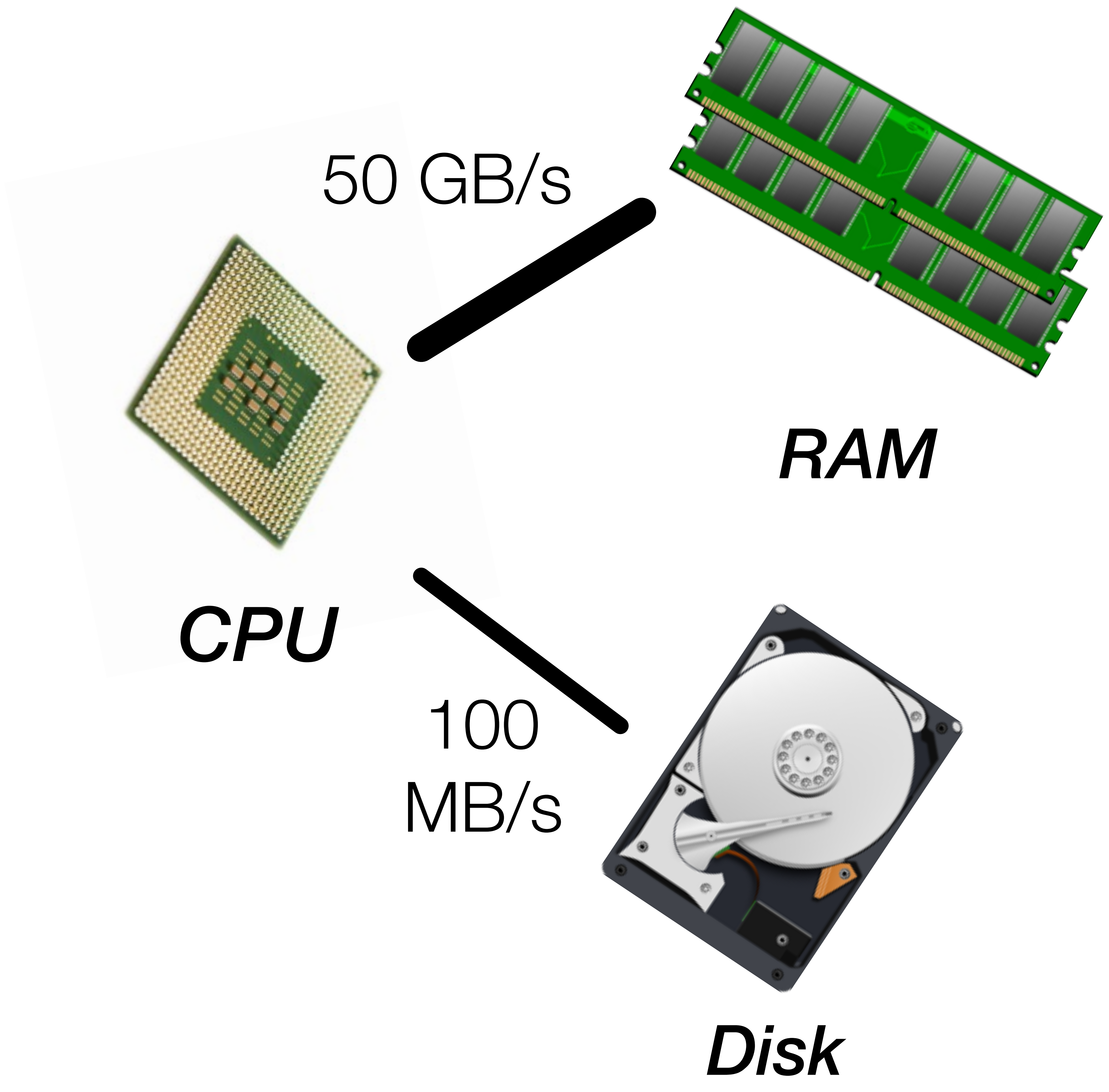
Communication Hierarchy



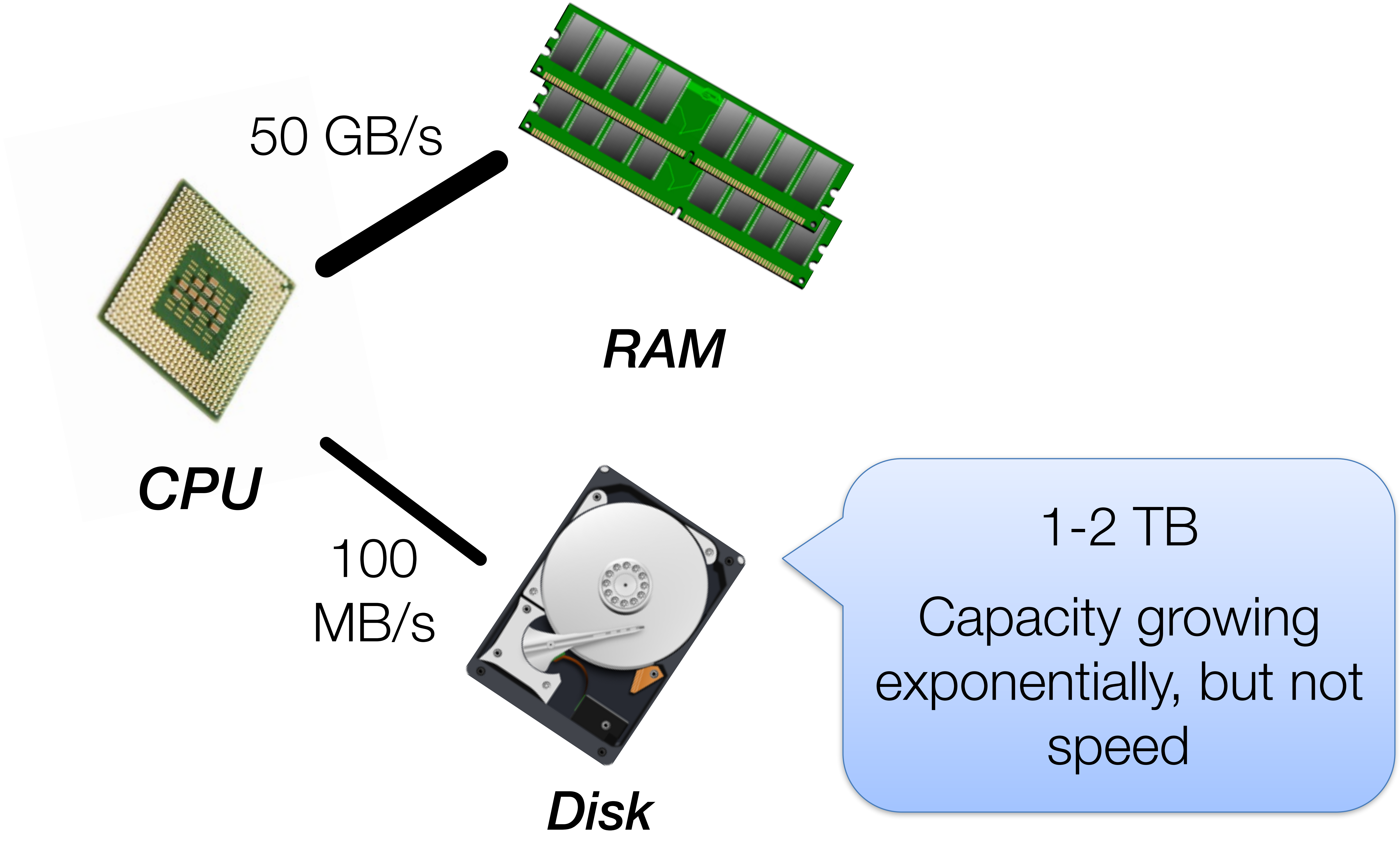
10-100 GB

Capacity growing with Moore's Law

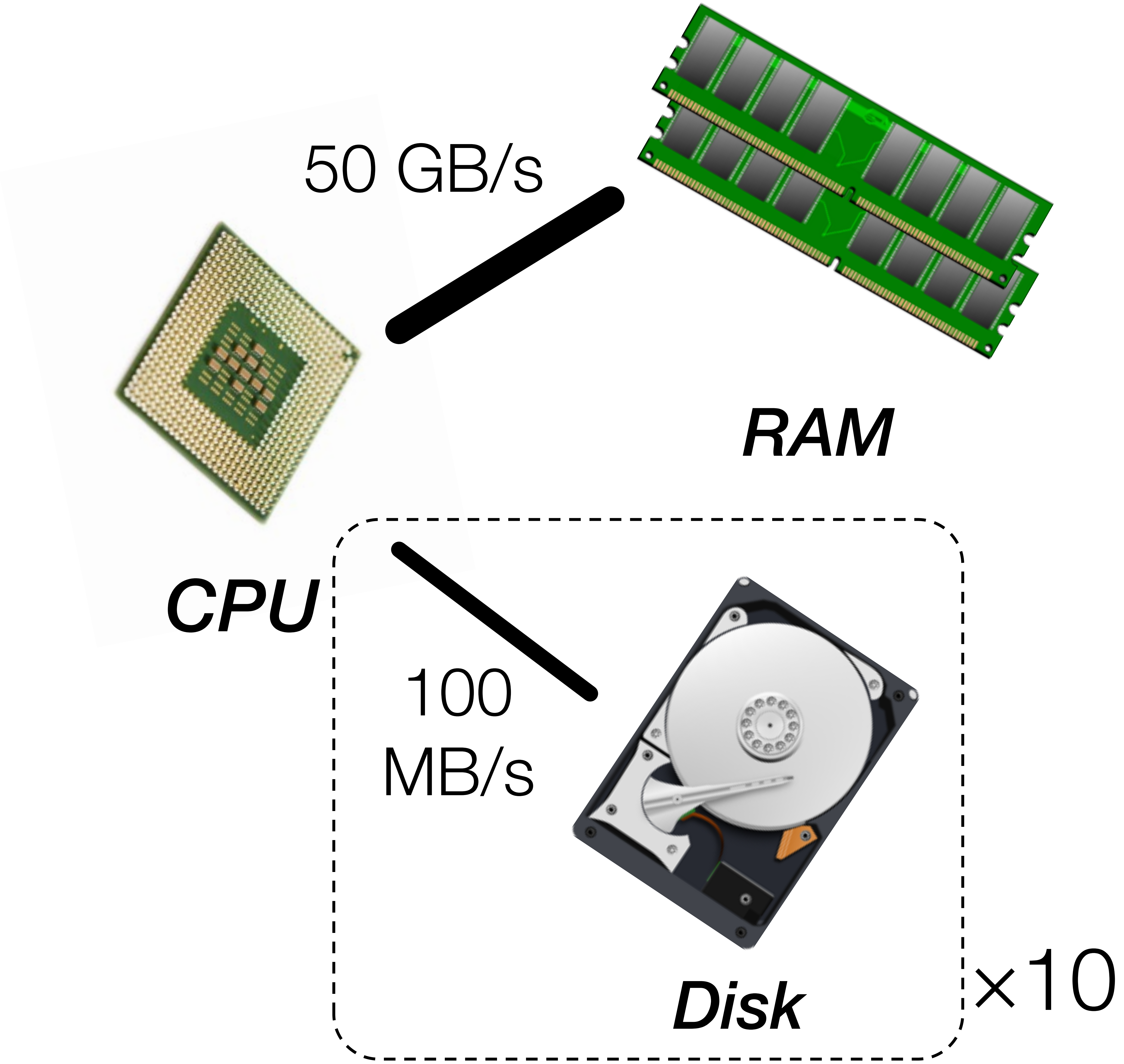
Communication Hierarchy



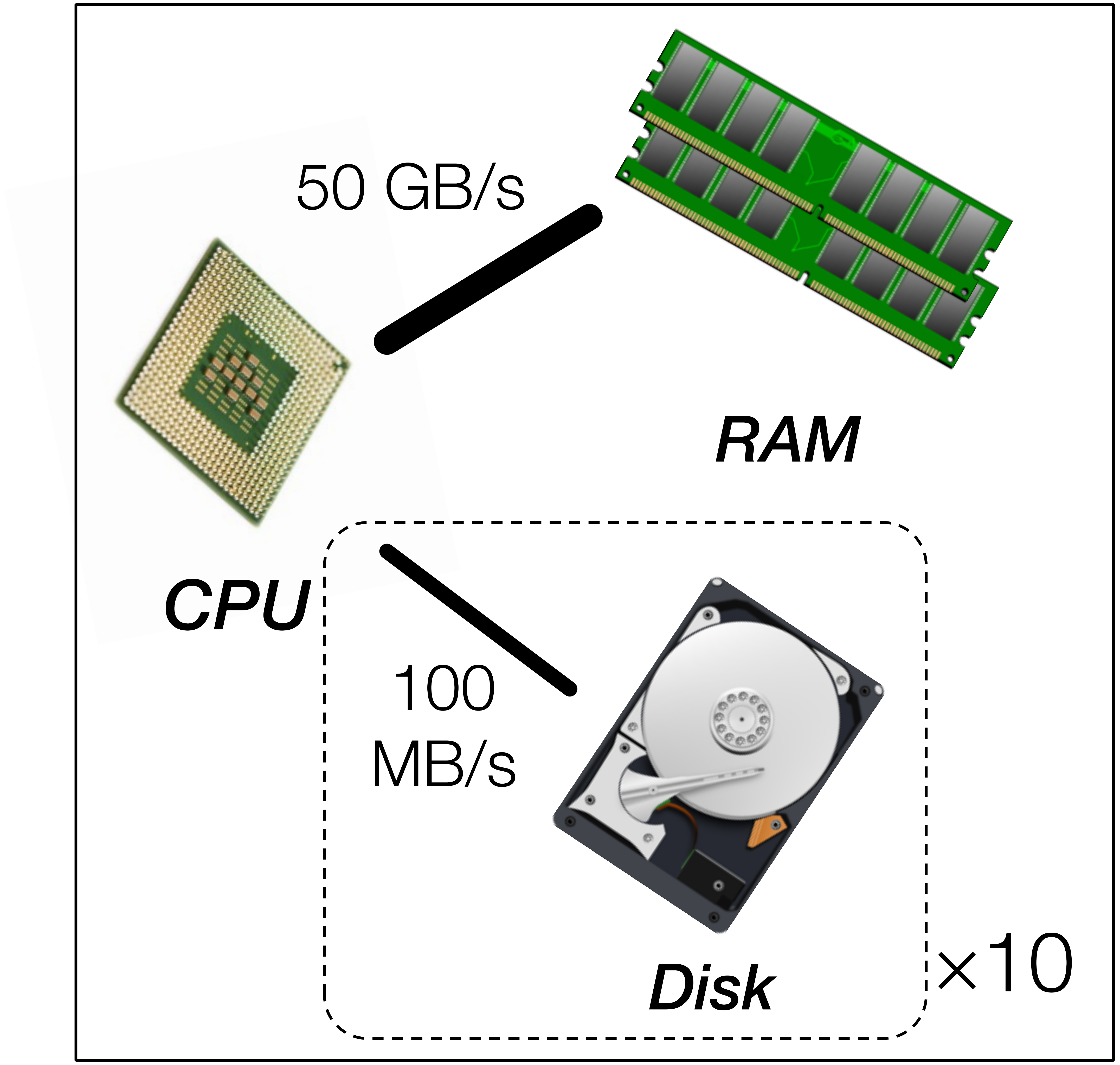
Communication Hierarchy



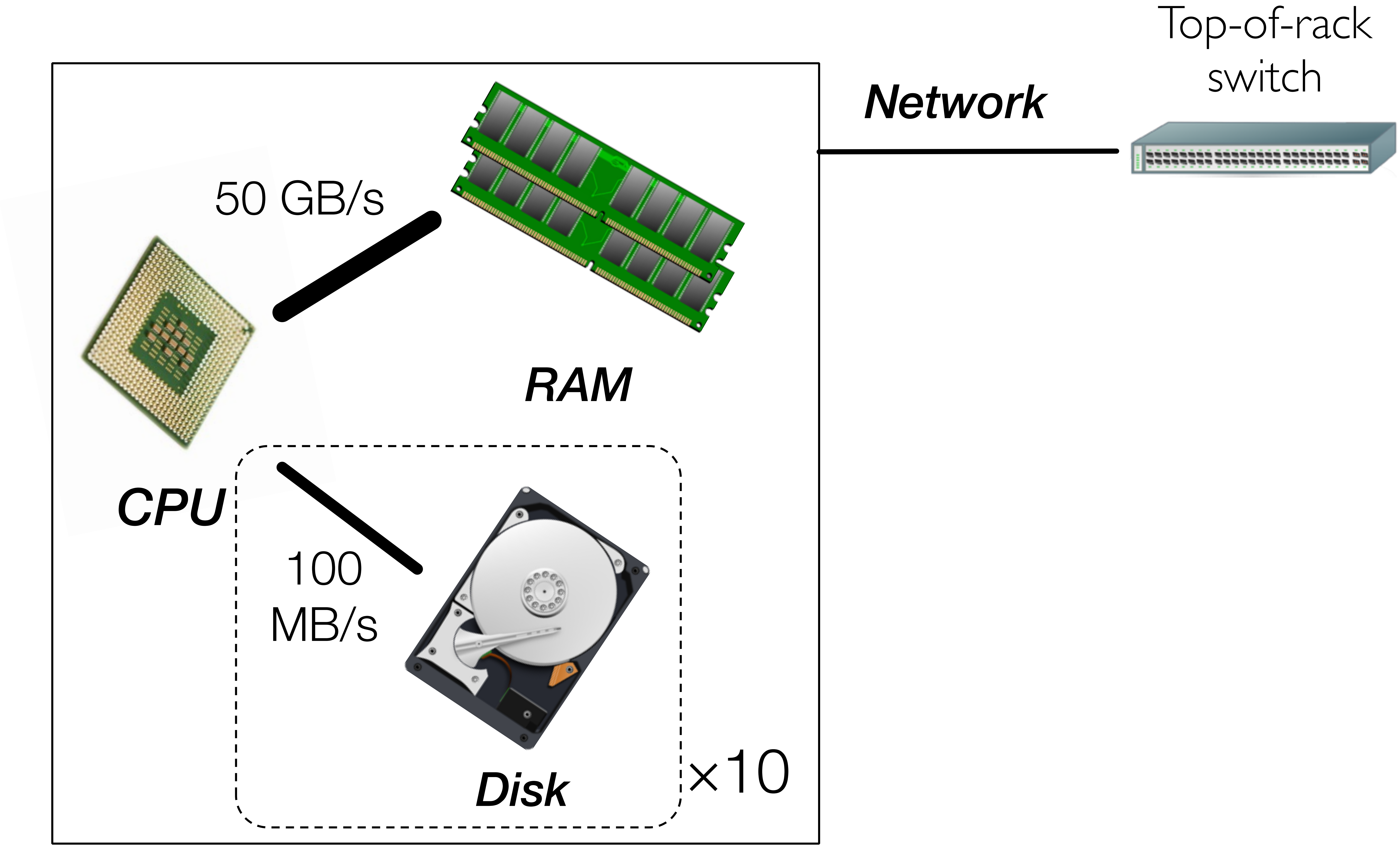
Communication Hierarchy



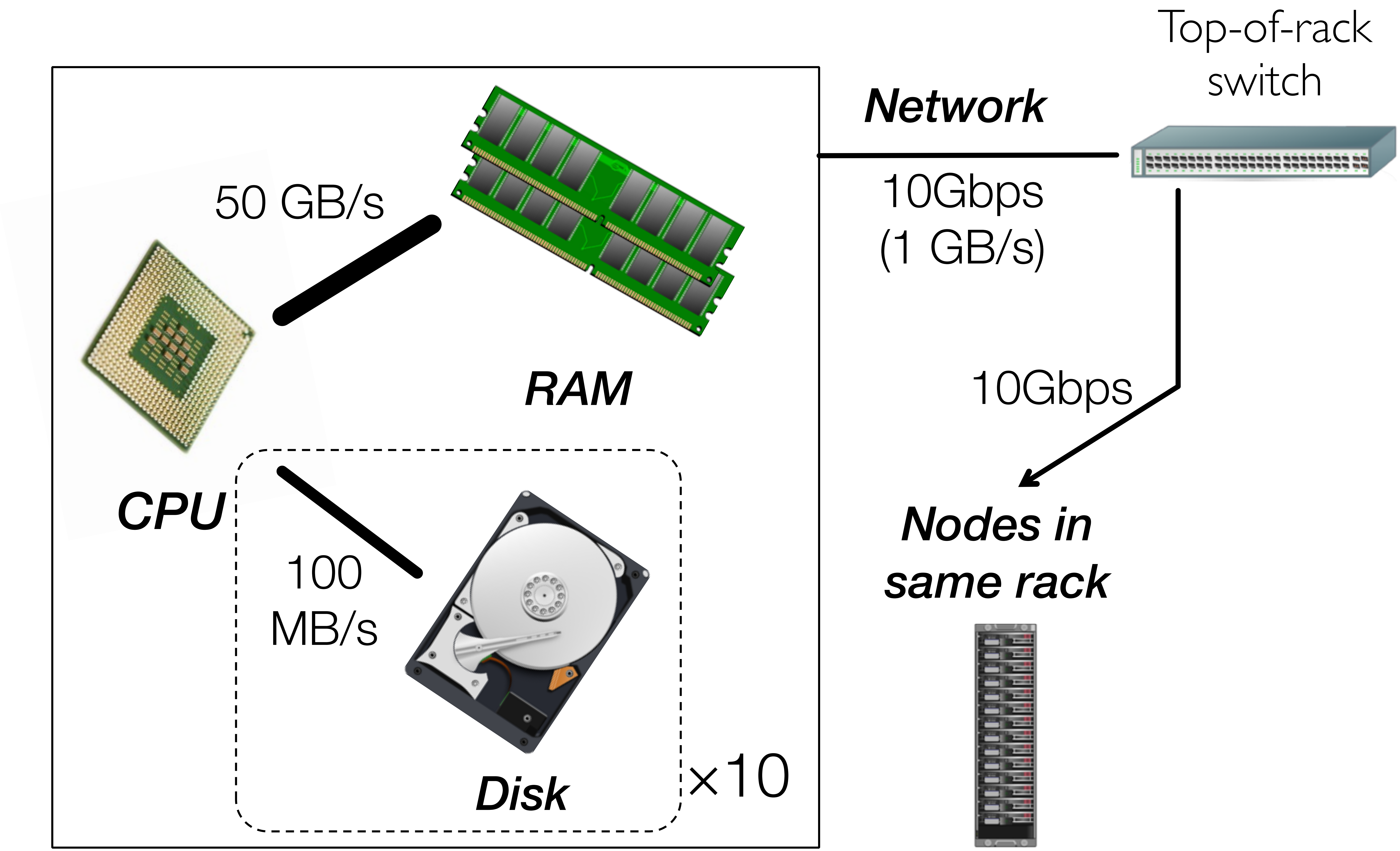
Communication Hierarchy



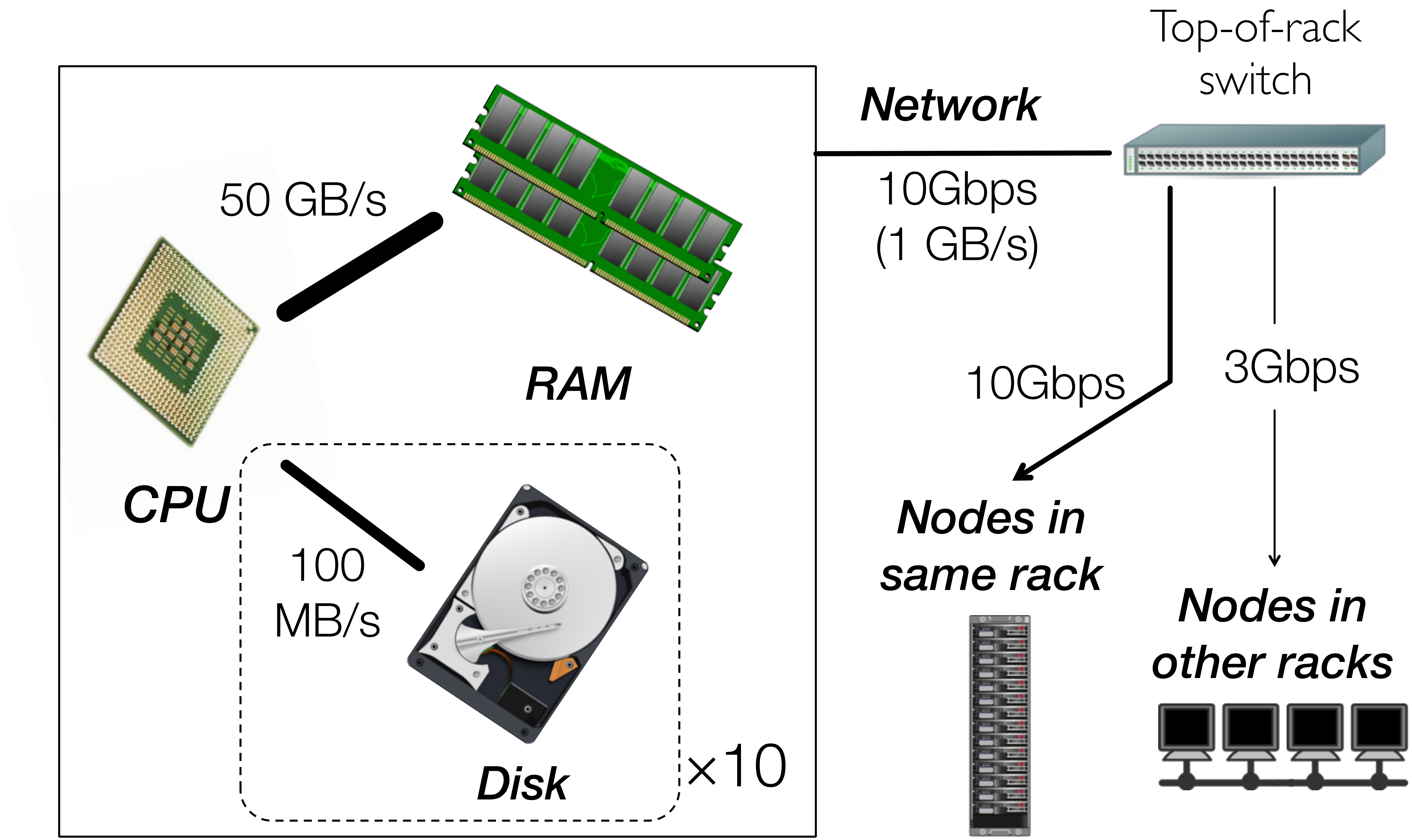
Communication Hierarchy



Communication Hierarchy



Communication Hierarchy



Summary

Access rates fall sharply with distance

50× gap between memory and network!

Must be mindful of this hierarchy when developing parallel algorithms!

Distributed ML: Communication Principles

Communication Hierarchy

Access rates fall sharply with distance

- Parallelism makes computation fast
- Network makes communication slow

Must be mindful of this hierarchy when developing parallel algorithms!

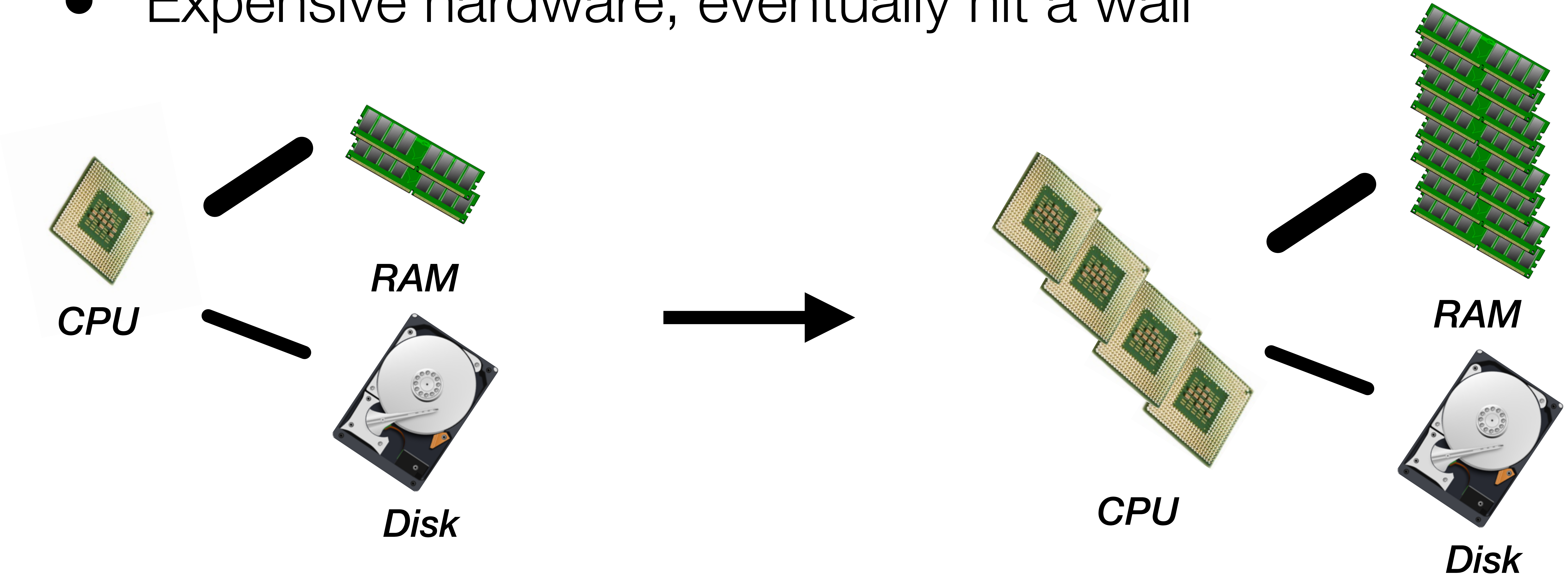
2nd Rule of thumb
Perform parallel and in-memory computation

Persisting in memory reduces communication

- Especially for iterative computation (gradient descent)

Scale-up (powerful multicore machine)

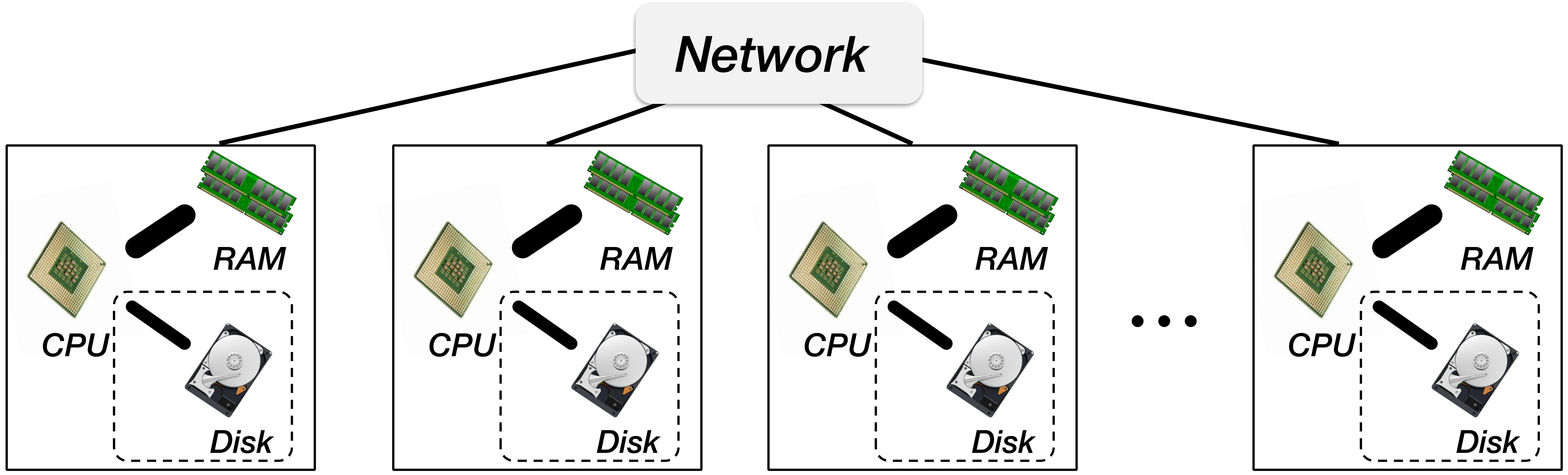
- No network communication
- Expensive hardware, eventually hit a wall



2nd Rule of thumb
Perform parallel and in-memory computation

Persisting in memory reduces communication
- Especially for iterative computation (gradient descent)

Scale-out (distributed, e.g., cloud-based)
- Need to deal with network communication
- Commodity hardware, scales to massive problems



2nd Rule of thumb
Perform parallel and in-memory computation

Persisting in memory reduces communication

- Especially for iterative computation (gradient descent)

Scale-out (distributed, e.g., cloud-based)

- Need to deal with network communication
- Commodity hardware, scales to massive problems

```
> train.cache()   ← Persist training data across iterations
  for i in range(numIters):
      alpha_i = alpha / (n * np.sqrt(i+1))
      gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
      w -= alpha_i * gradient
```

3rd Rule of thumb

Minimize Network Communication

Q: How should we leverage distributed computing while mitigating network communication?

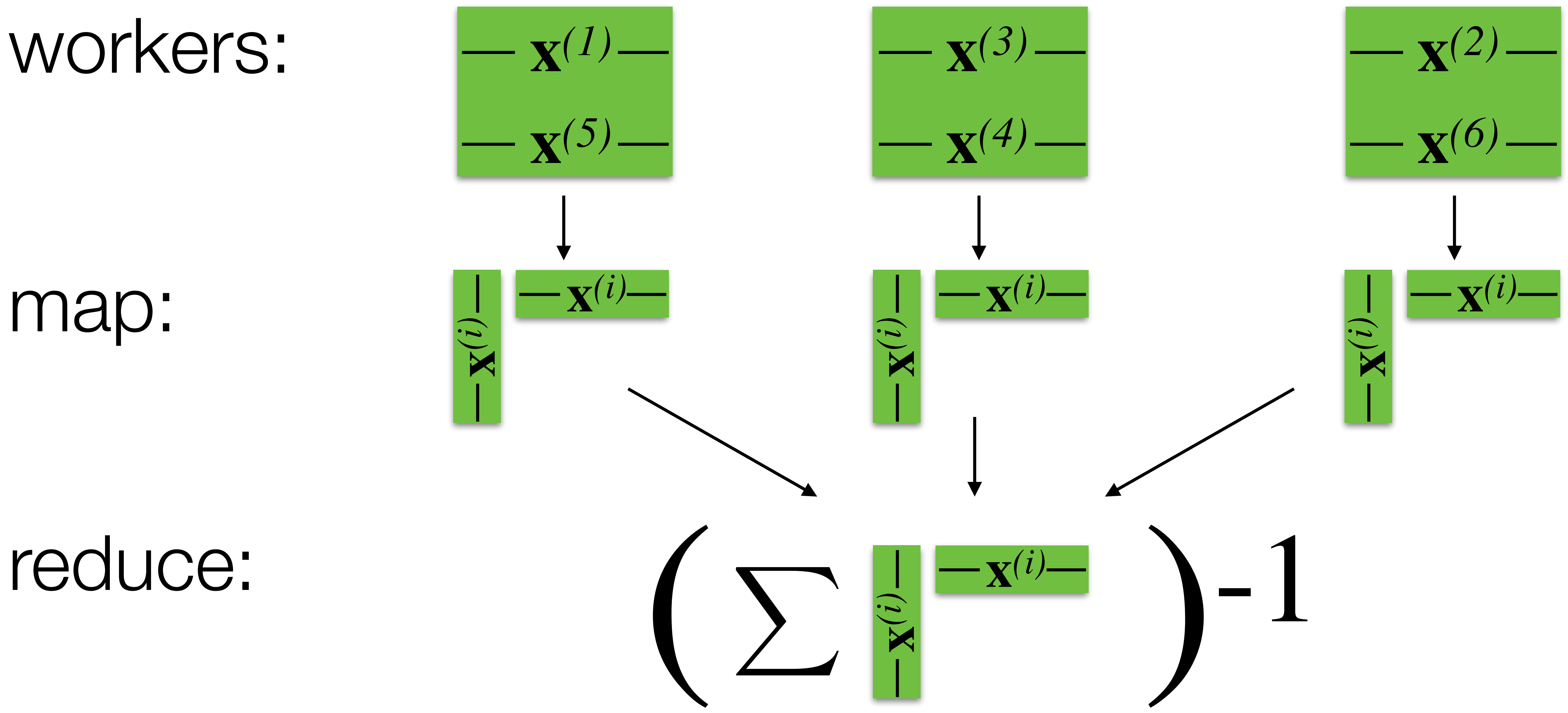
First Observation: We need to store and potentially communicate Data, Model and Intermediate objects

- **A**: Keep large objects local

3rd Rule of thumb
Minimize Network Communication - Stay Local

Example: Linear regression, big n and small d

- Solve via closed form (not iterative!)
- Communicate O(d^2) intermediate data



3rd Rule of thumb

Minimize Network Communication - Stay Local

$\mathbf{w}_{i+1}$

workers:

— $\mathbf{x}^{(1)}$ —	— $\mathbf{x}^{(3)}$ —	— $\mathbf{x}^{(2)}$ —
— $\mathbf{x}^{(5)}$ —	— $\mathbf{x}^{(4)}$ —	— $\mathbf{x}^{(6)}$ —

map:

$$(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)} \qquad (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)} \qquad (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$$

reduce:

$$\sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$$

3rd Rule of thumb
Minimize Network Communication - Stay Local

Example: Linear regression, big n and big d

- Gradient descent, communicate $\mathbf{w}_i$
- O(d) communication OK for fairly large d
- Compute locally on data (*Data Parallel*)

workers:

$\mathbf{x}^{(1)}$, $\mathbf{x}^{(5)}$	$\mathbf{x}^{(3)}$, $\mathbf{x}^{(4)}$	$\mathbf{x}^{(2)}$, $\mathbf{x}^{(6)}$

map: $(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ $\quad(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$ $\quad(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$

reduce:

$$\sum_{j=1}^{n}(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$$

$\mathbf{w}_{i+1}$

3rd Rule of thumb

Minimize Network Communication - Stay Local

Example: Hyperparameter tuning for ridge regression with small n and small d

- Data is small, so can communicate it
- ‘Model’ is collection of regression models corresponding to different hyperparameters
- Train each model locally (*Model Parallel*)

3rd Rule of thumb

Minimize Network Communication - Stay Local

Example: Linear regression, big n and huge d

- Gradient descent
- O(d) communication slow with hundreds of millions parameters
- Distribute data and model (*Data and Model Parallel*)
- Often rely on sparsity to reduce communication

3rd Rule of thumb
Minimize Network Communication

Q: How should we leverage distributed computing while mitigating network communication?

First Observation: We need to store and potentially communicate Data, Model and Intermediate objects

- **A**: Keep large objects local

Second Observation: ML methods are typically iterative

- **A**: Reduce # iterations

3rd Rule of thumb

Minimize Network Communication - Reduce Iterations

Distributed iterative algorithms must compute and communicate

- In Bulk Synchronous Parallel (BSP) systems, e.g., Apache Spark, we strictly alternate between the two

Distributed Computing Properties

- Parallelism makes computation fast
- Network makes communication slow

Idea: Design algorithms that **compute more, communicate less**

- Do more computation at each iteration
- Reduce total number of iterations

3rd Rule of thumb
Minimize Network Communication - Reduce Iterations

Extreme: **Divide-and-conquer**

- Fully process each partition locally, communicate final result
- Single iteration; minimal communication
- Approximate results

```
> w = train.mapPartitions(localLinearRegression)
                 .reduce(combineLocalRegressionResults)
```

```
> for i in range(numIters):
      alpha_i = alpha / (n * np.sqrt(i+1))
      gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
      w -= alpha_i * gradient
```

3rd Rule of thumb

Minimize Network Communication - Reduce Iterations

Less extreme: **Mini-batch**

- Do more work locally than gradient descent before communicating
- Exact solution, but diminishing returns with larger batch sizes

```
for i in range(fewerIters):
    update = train.mapPartitions(doSomeLocalGradientUpdates)
                  .reduce(combineLocalUpdates)
    w += update
```

```
for i in range(numIters):
    alpha_i = alpha / (n * np.sqrt(i+1))
    gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
    w -= alpha_i * gradient
```

3rd Rule of thumb

Minimize Network Communication - Reduce Iterations

Throughput: How many bytes per second can be read

Latency: Cost to send message (independent of size)

Latency	
Memory	1e-4 ms
Hard Disk	10 ms
Network (same datacenter)	.25 ms
Network (US to Europe)	>5 ms

We can amortize latency!

- Send larger messages
- *Batch* their communication
- E.g., Train multiple models together

1st Rule of thumb

Computation and storage should be linear (in n, d)

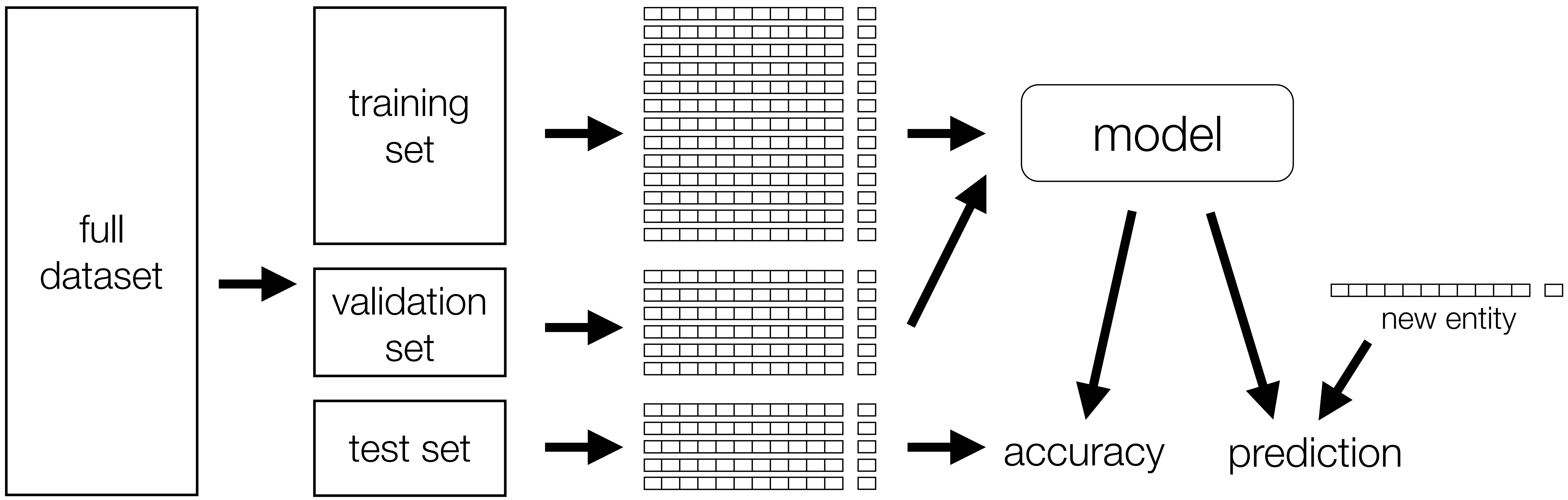
2nd Rule of thumb

Perform parallel and in-memory computation

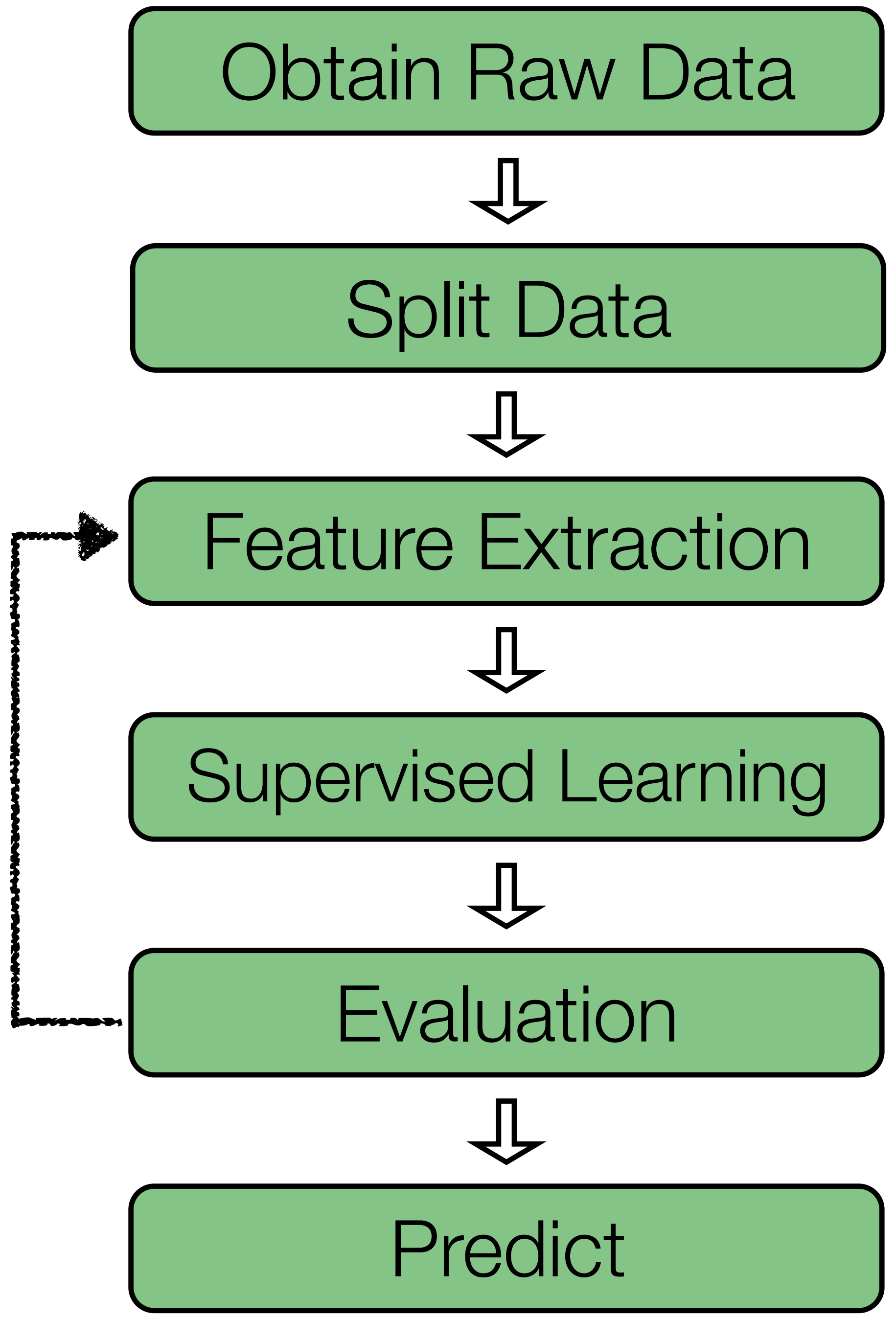
3rd Rule of thumb

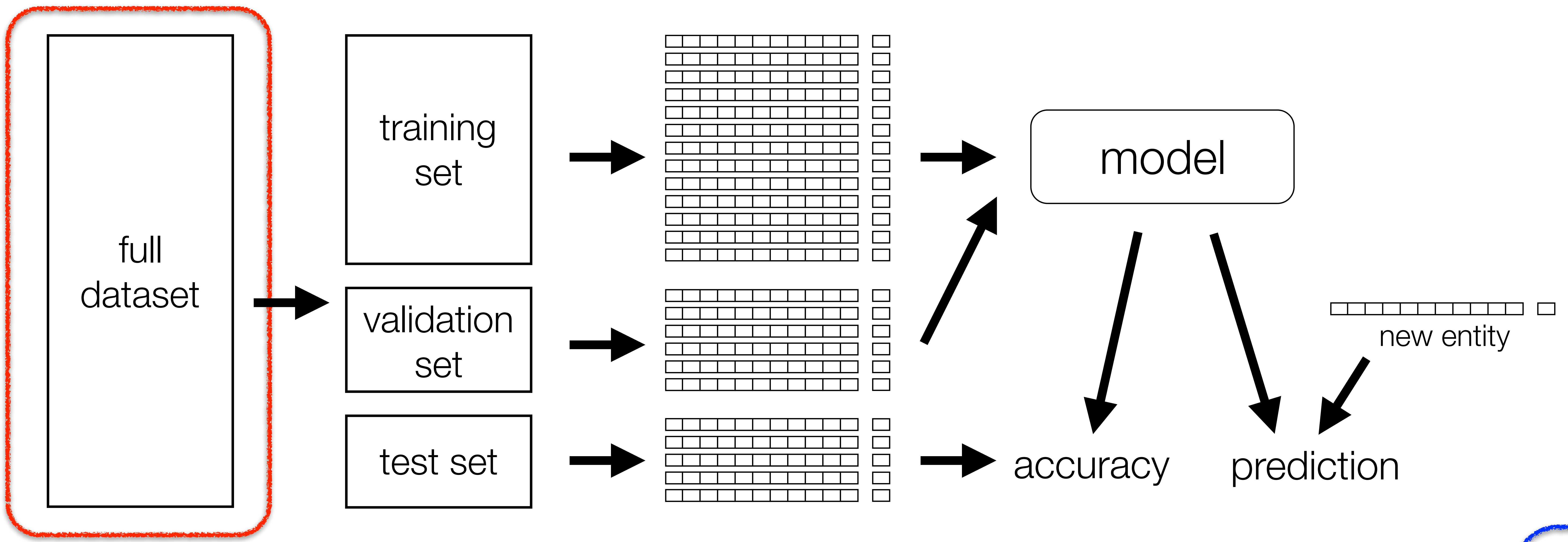
Minimize Network Communication

Lab Preview



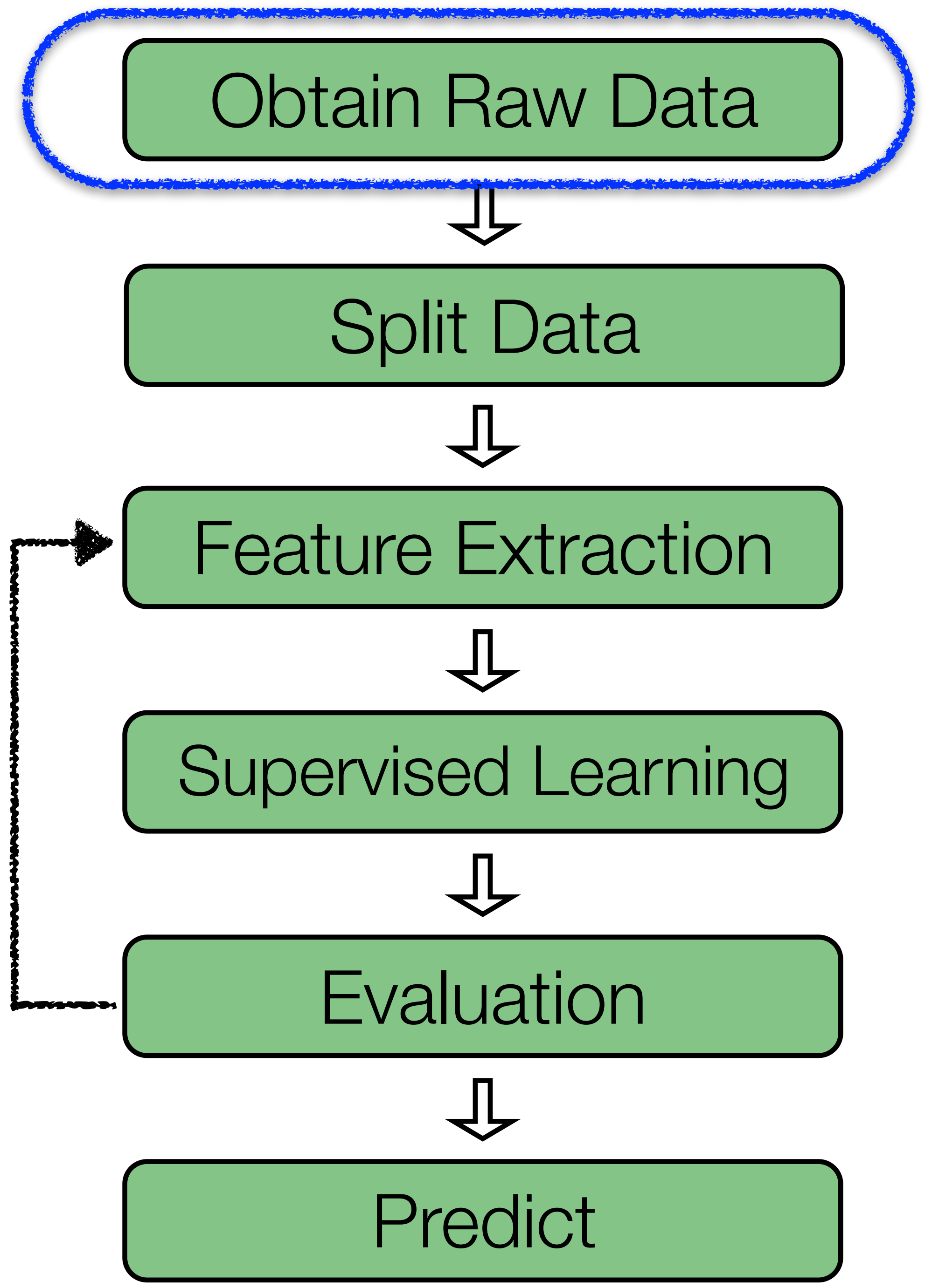
Goal: Predict song’s release year from audio features

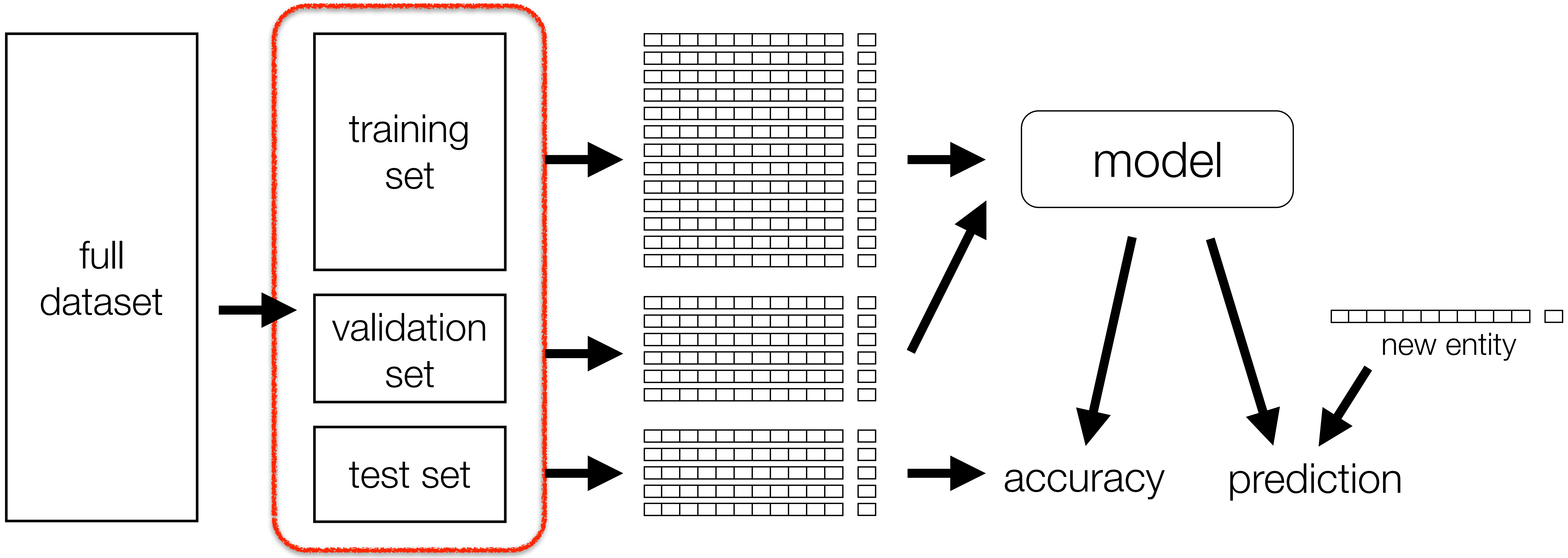




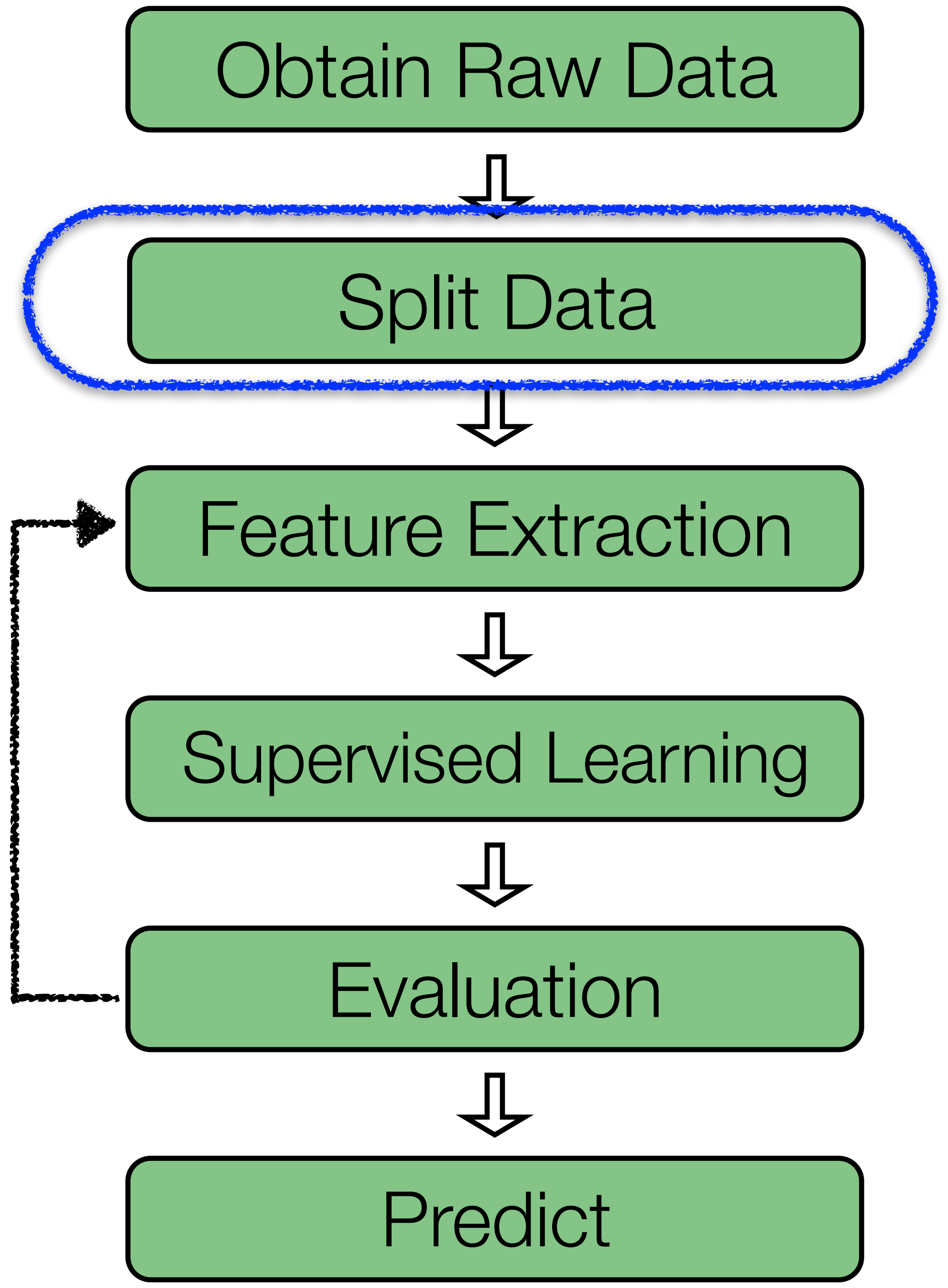
Raw Data: Millionsong Dataset from UCI ML Repository

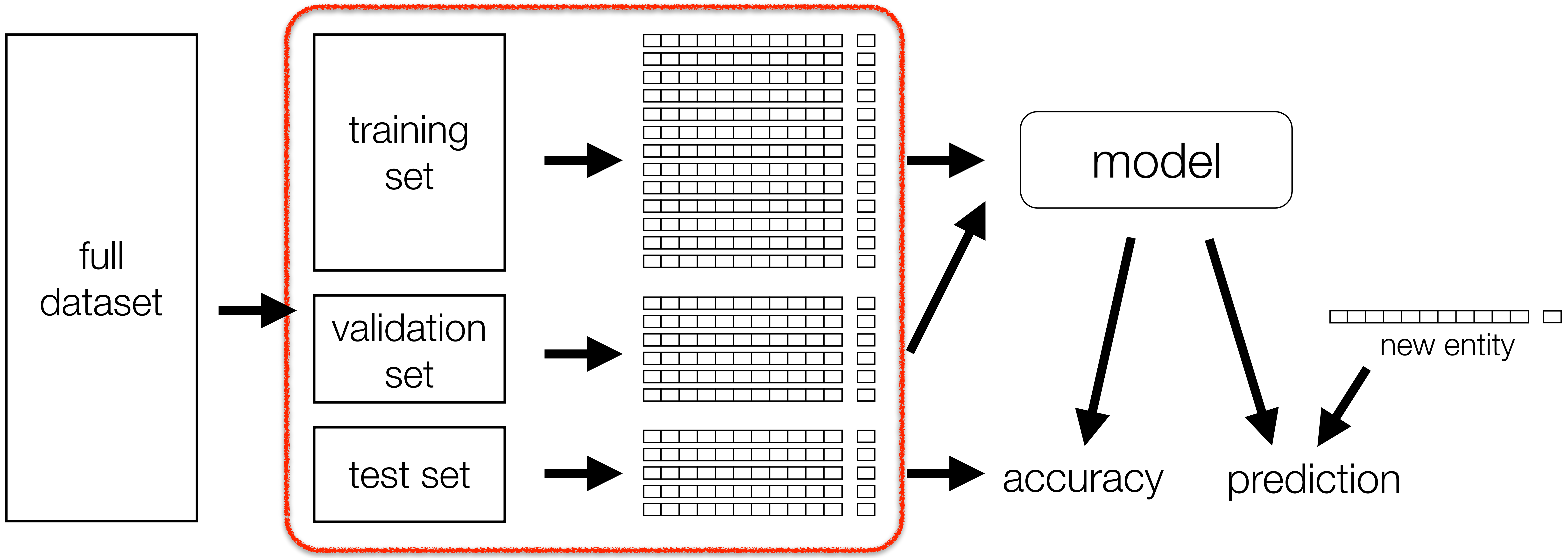
- Explore features
- Shift labels so that they start at 0 (for interpretability)
- Visualize data





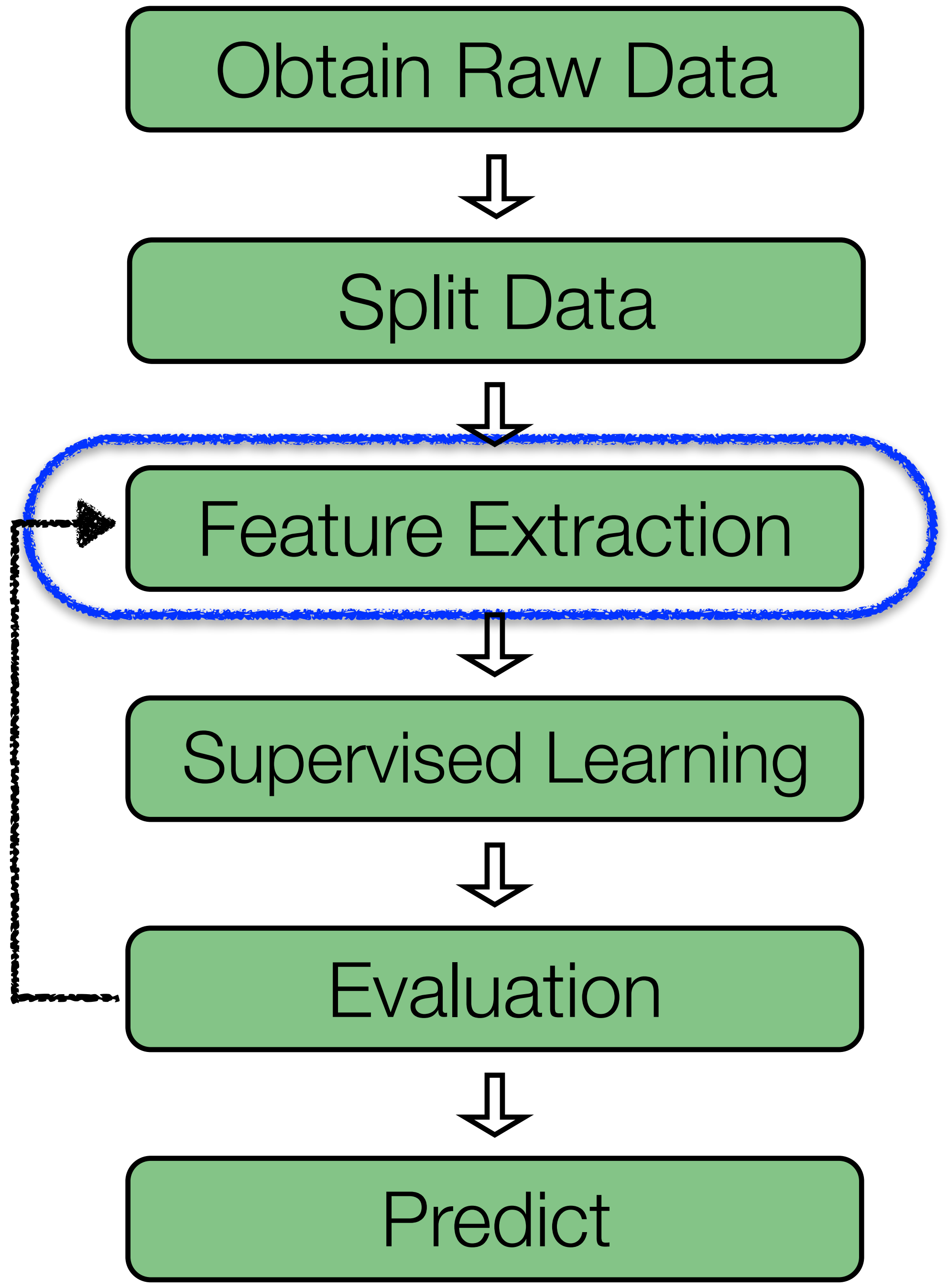
Split Data: Create training, validation, and test sets

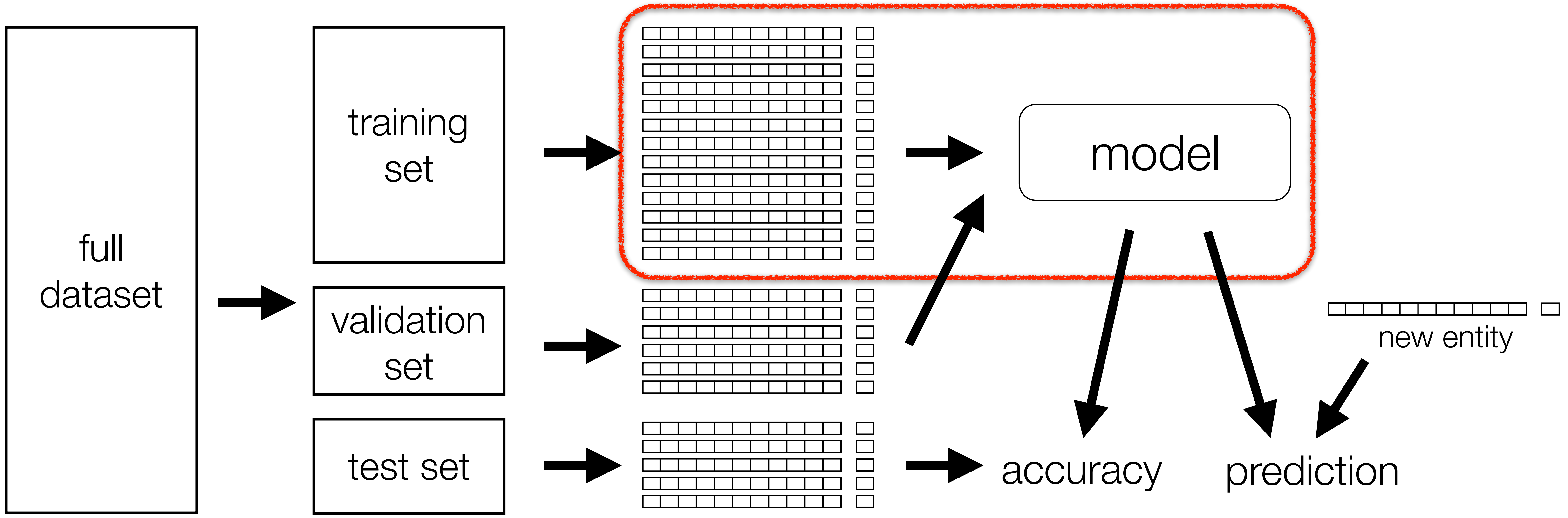




Feature Extraction:

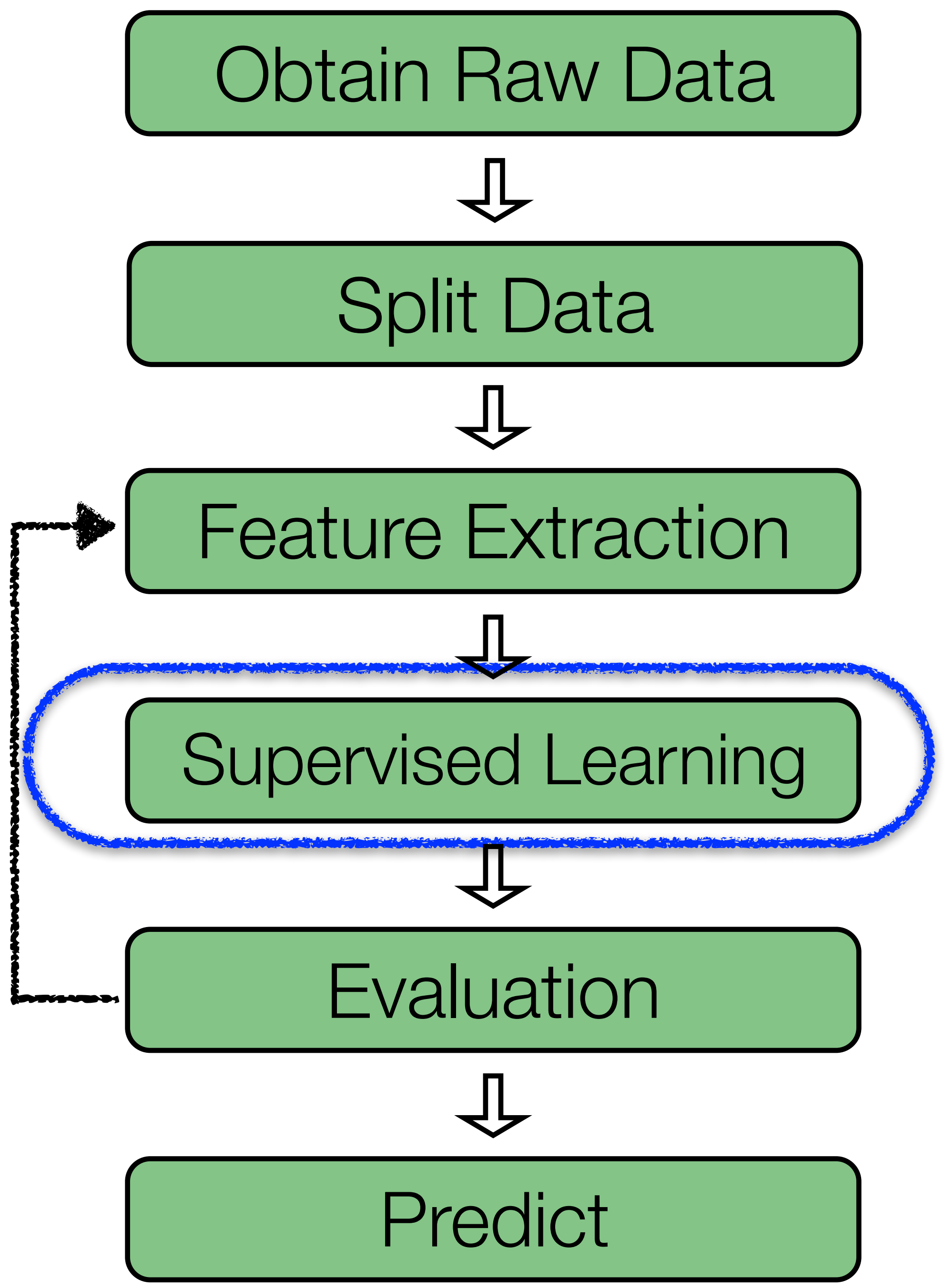
- Initially use raw features
- Subsequently compare with quadratic features

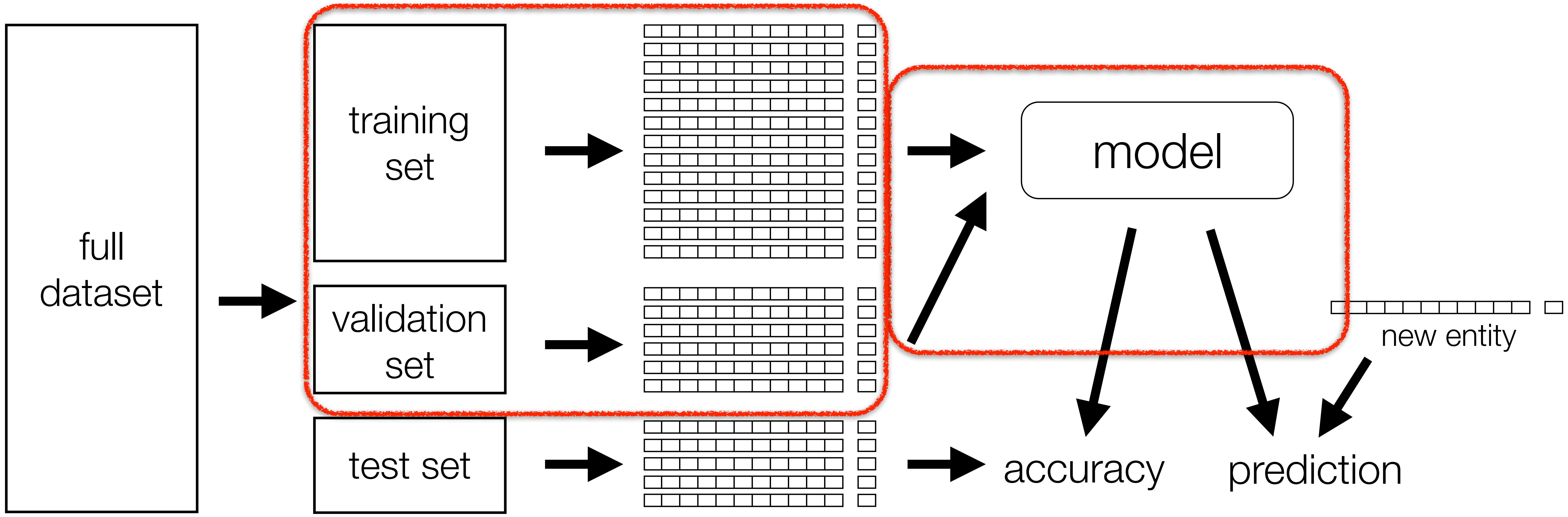


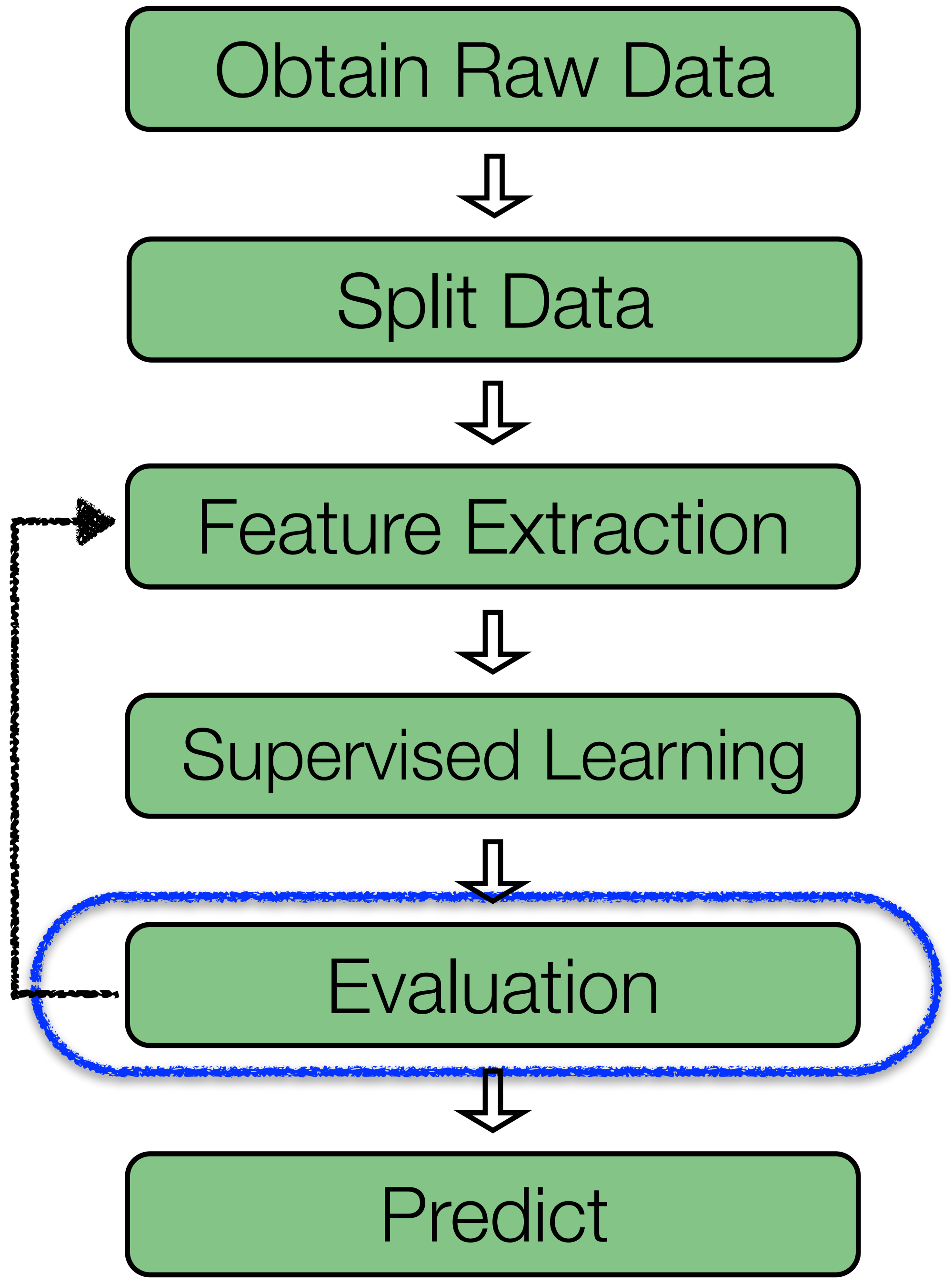


Supervised Learning: Least Squares Regression

- First implement gradient descent from scratch
- Then use MLlib implementation
- Visualize performance by iteration

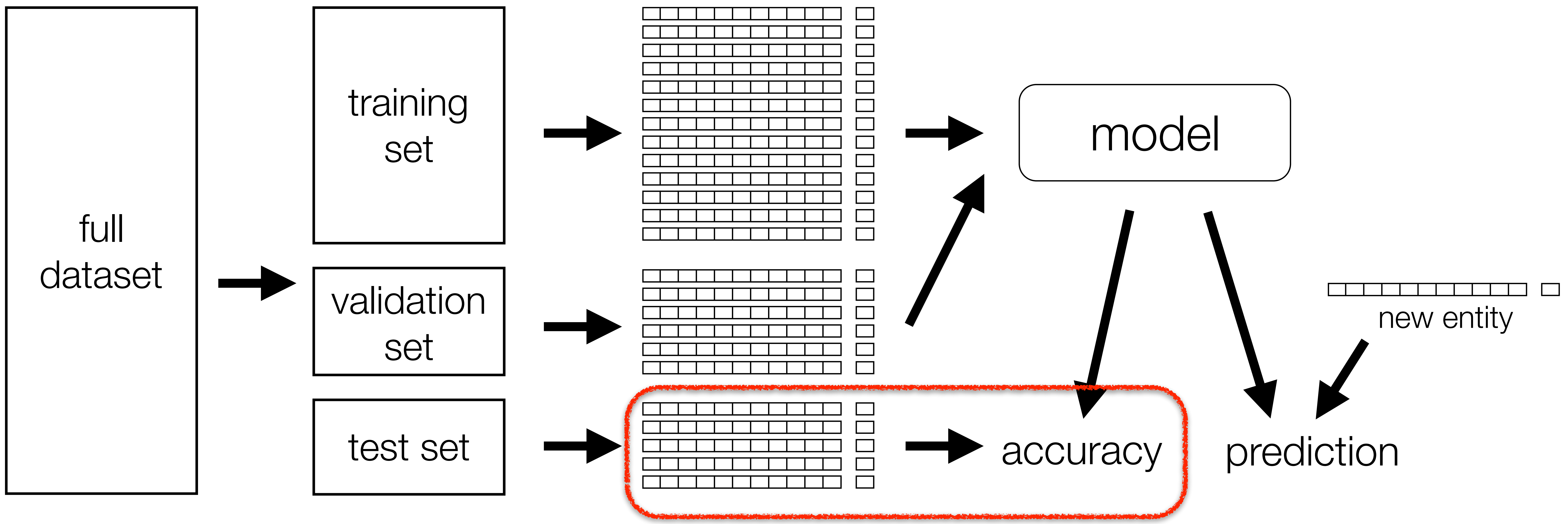






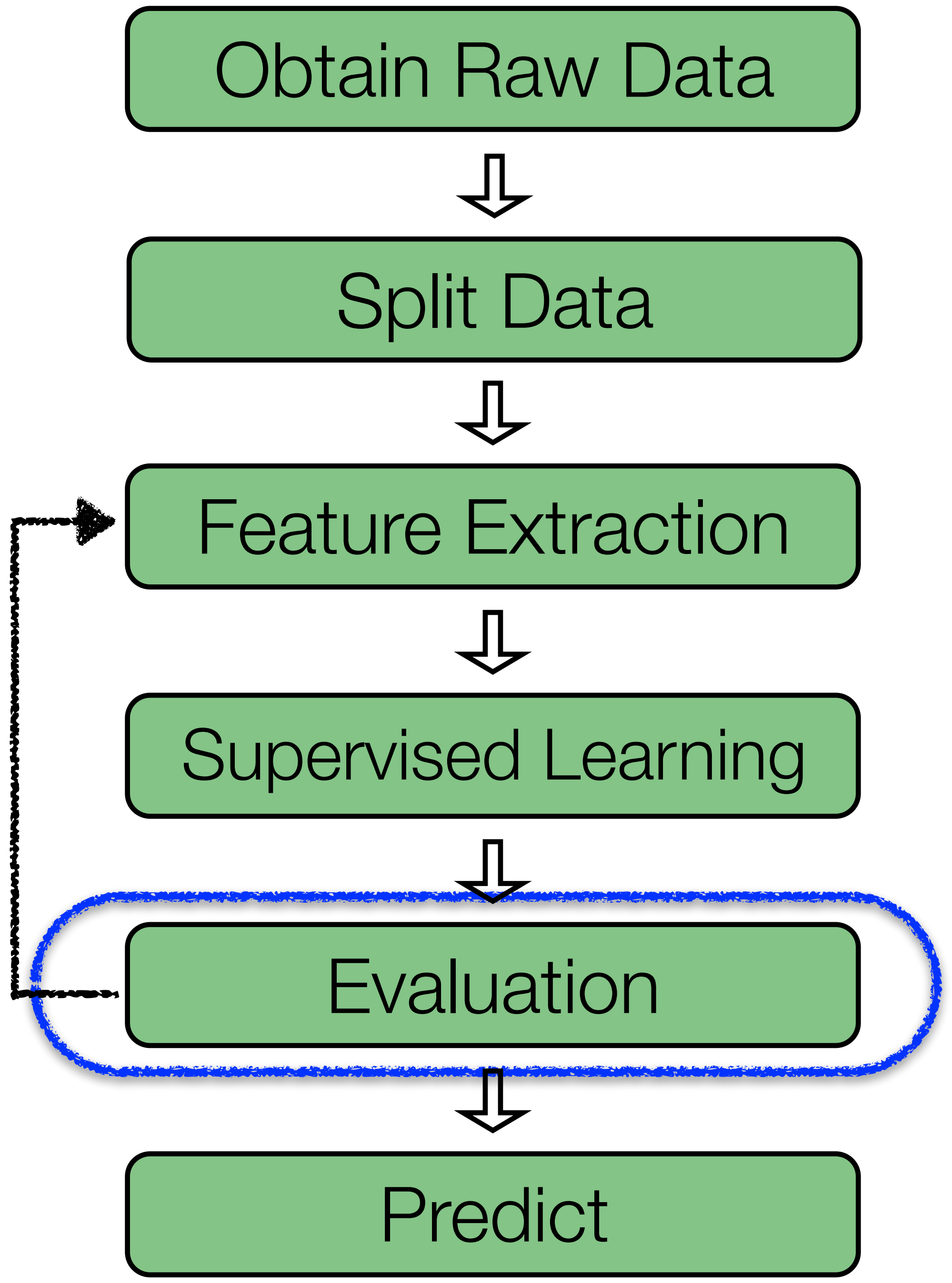
Evaluation (Part 1): Hyperparameter tuning

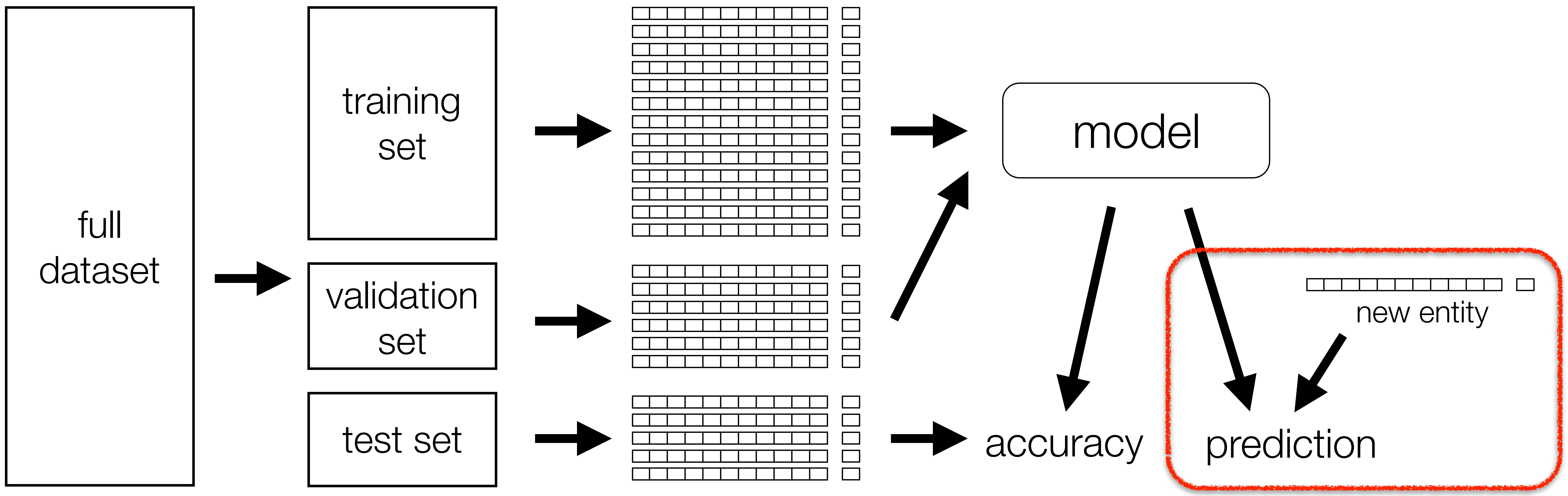
- Use grid search to find good values for regularization and step size hyperparameters
- Evaluate using RMSE
- Visualize grid search

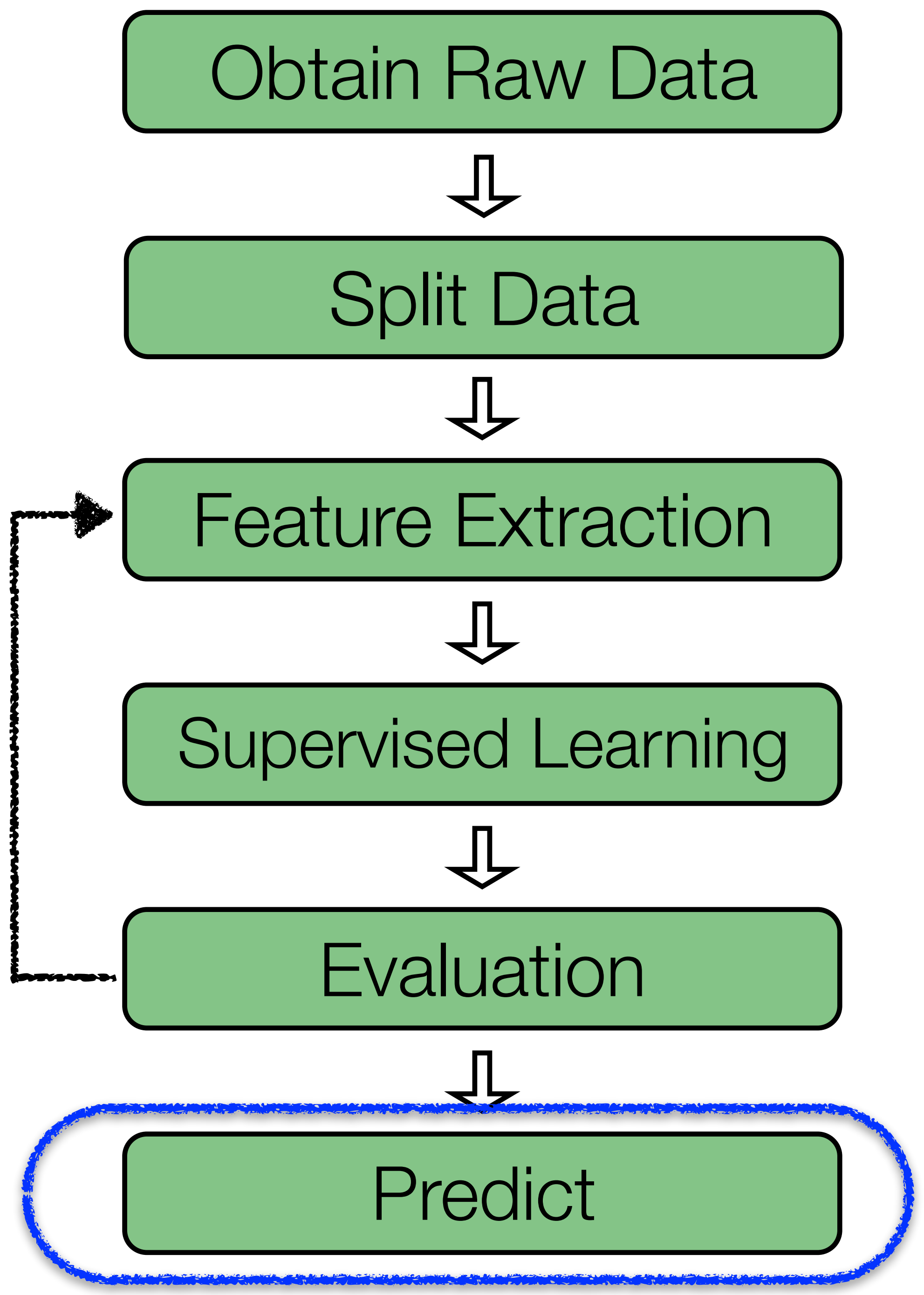


Evaluation (Part 2): Evaluate final model

- Evaluate using RMSE
- Compare to baseline model that returns average song year in training data







Predict: Final model could be used to predict song year for new songs (we won't do this though)

MLlib and Pipelines

Spark's Machine Learning Library (MLlib)

- Consists of common learning algorithms and utilities
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering
  - Dimensionality reduction

- Two packages:
  - spark.mllib
  - spark.ml

ML: Transformer

- A *Transformer* is a class which can transform one DataFrame into another DataFrame
- A Transformer implements `transform()`

- Examples
  - HashingTF
  - LogisticRegressionModel
  - Binarizer

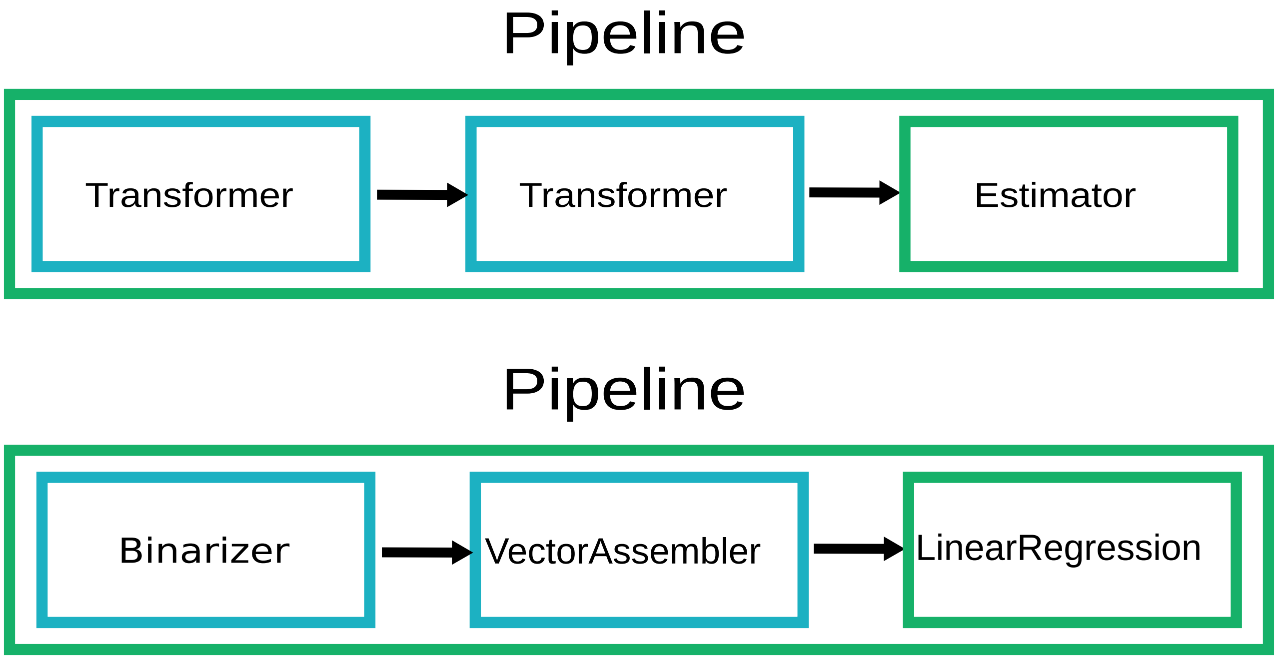
ML: Estimator

- An *Estimator* is a class which can take a DataFrame and produce a Transformer
- An Estimator implements `fit()`

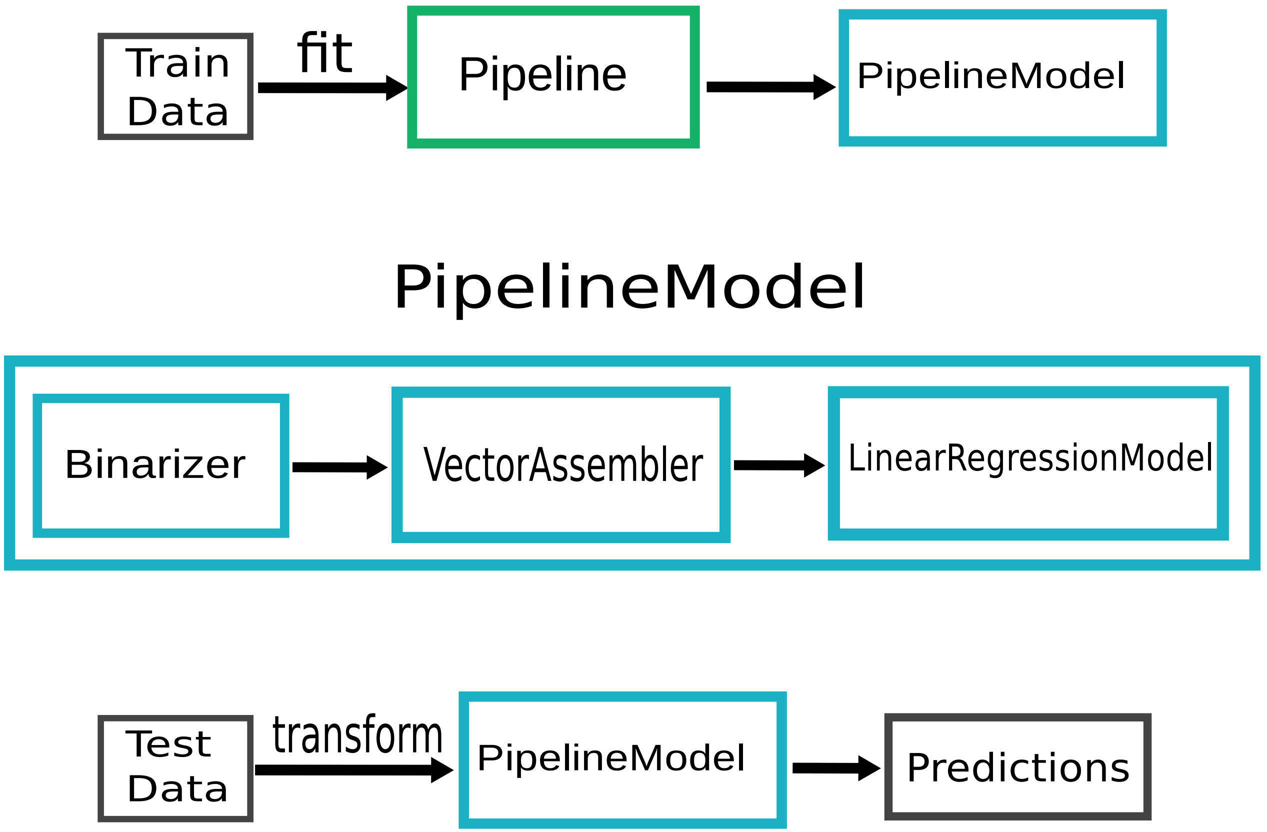
- Examples
  - LogisticRegression
  - StandardScaler
  - Pipeline

ML: Pipelines

A *Pipeline* is an estimator that contains stages representing a resusable workflow. Pipeline stages can be either estimators or transformers.



ML: PipelineModel



ML: Standard Scaler Pipeline