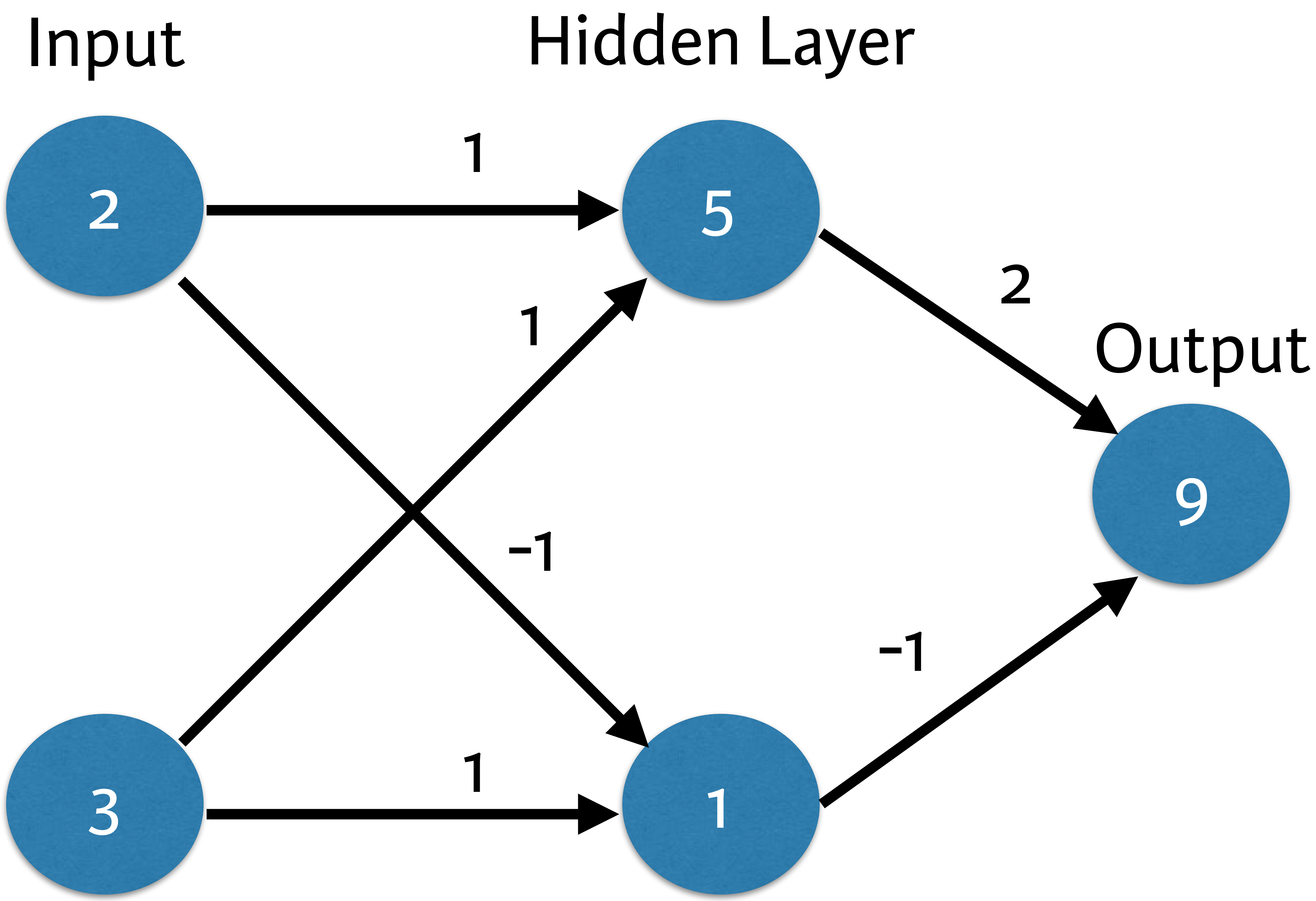
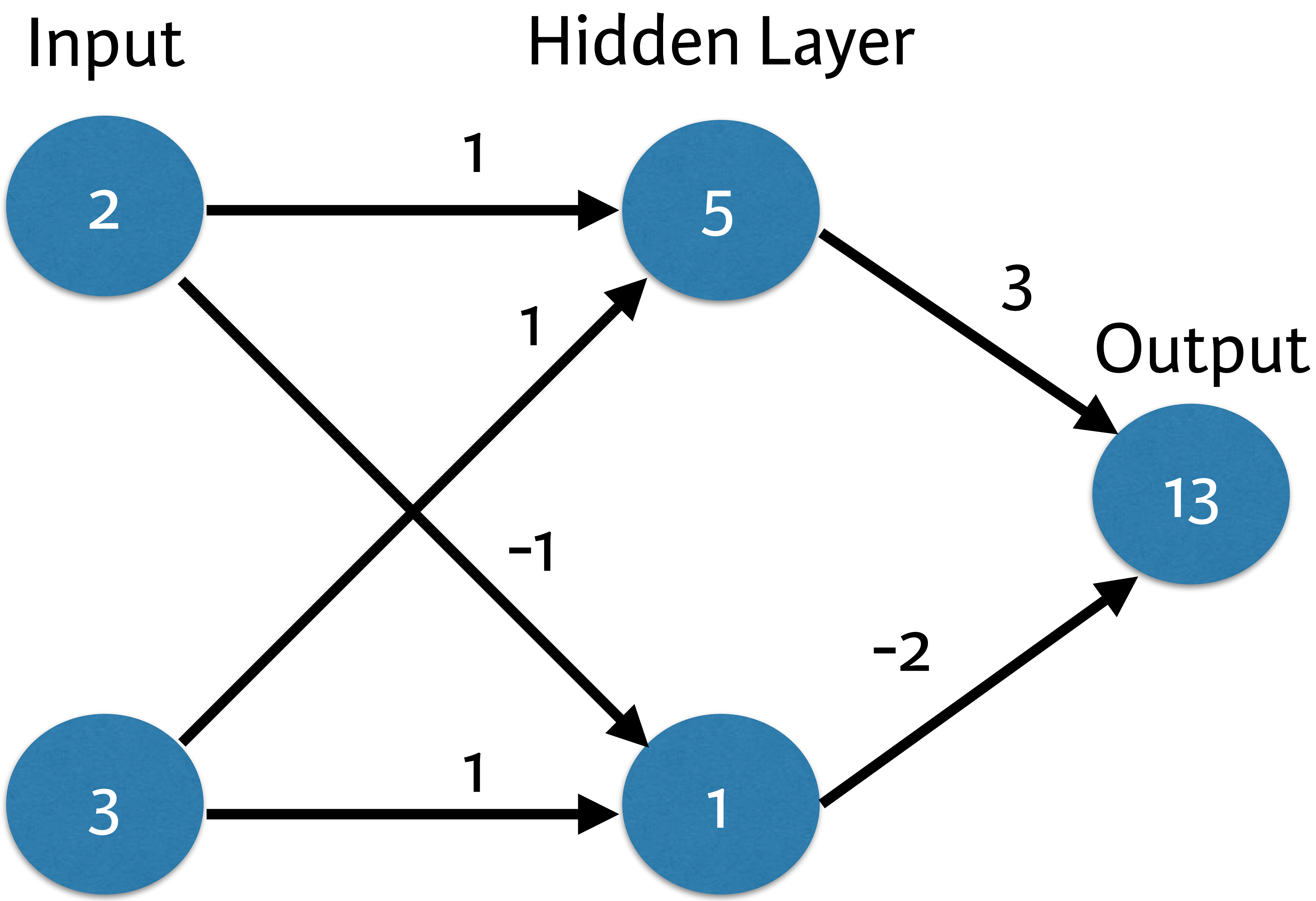
The need for optimization

A baseline neural network



- Actual Value of Target: 13
- Error: Predicted - Actual = -4

A baseline neural network



- Actual Value of Target: 13
- Error: Predicted - Actual = 0

Predictions with multiple points

- Making accurate predictions gets harder with more points
- At any set of weights, there are many values of the error
- ... corresponding to the many points we make predictions for

Loss function

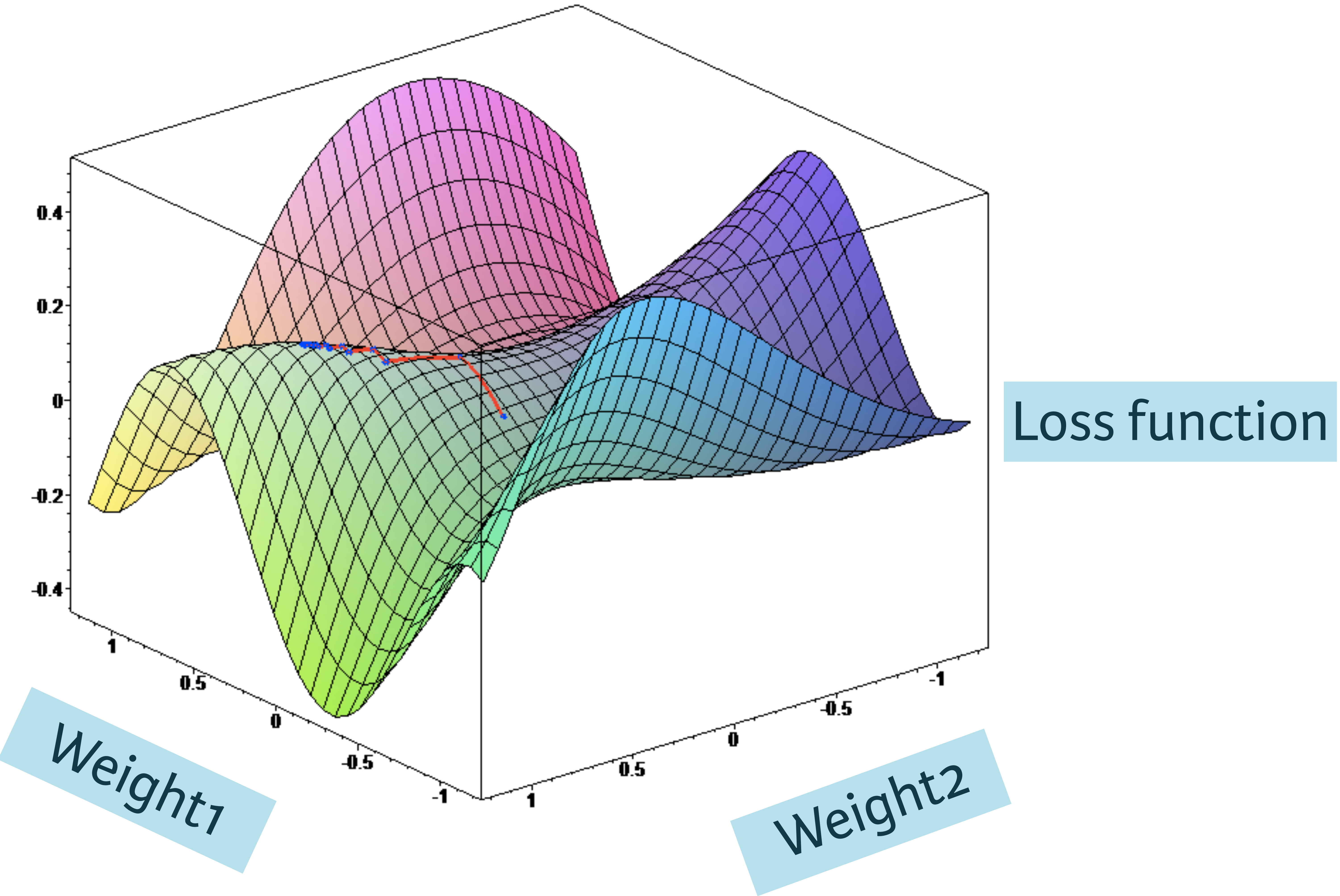
- Aggregates errors in predictions from many data points into single number
- Measure of model's predictive performance

Squared error loss function

Prediction	Actual	Error	Squared Error
10	20	-10	100
8	3	5	25
6	1	5	25

- Total Squared Error: 150
- Mean Squared Error: 50

Loss function



Loss function

- Lower loss function value means a better model
- Goal: Find the weights that give the lowest value for the loss function
- Gradient descent

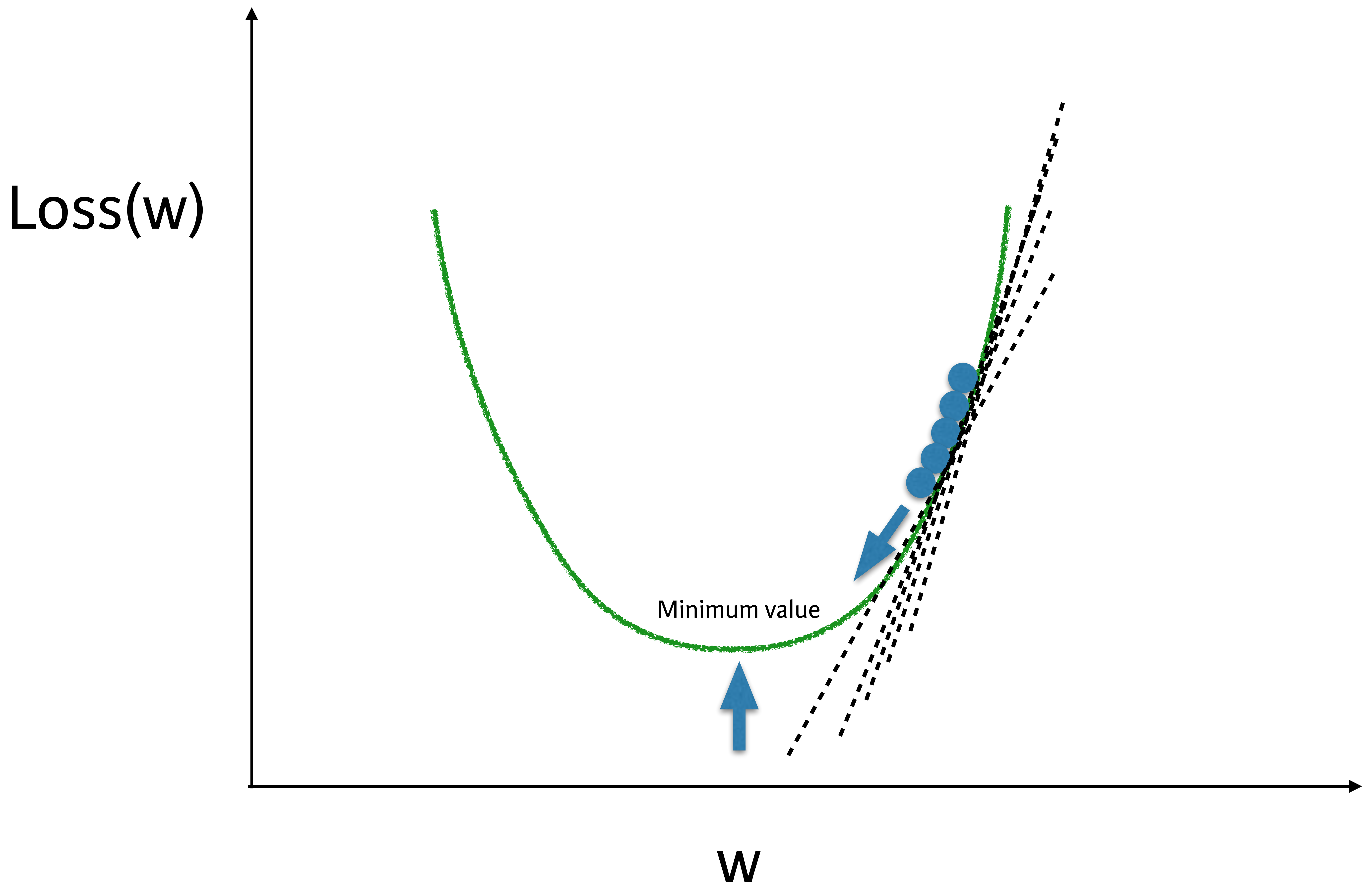
Gradient descent

- Imagine you are in a pitch dark field
- Want to find the lowest point
- Feel the ground to see how it slopes
- Take a small step downhill
- Repeat until it is uphill in every direction

Gradient descent steps

- Start at random point
- Until you are somewhere flat:
    - Find the slope
    - Take a step downhill

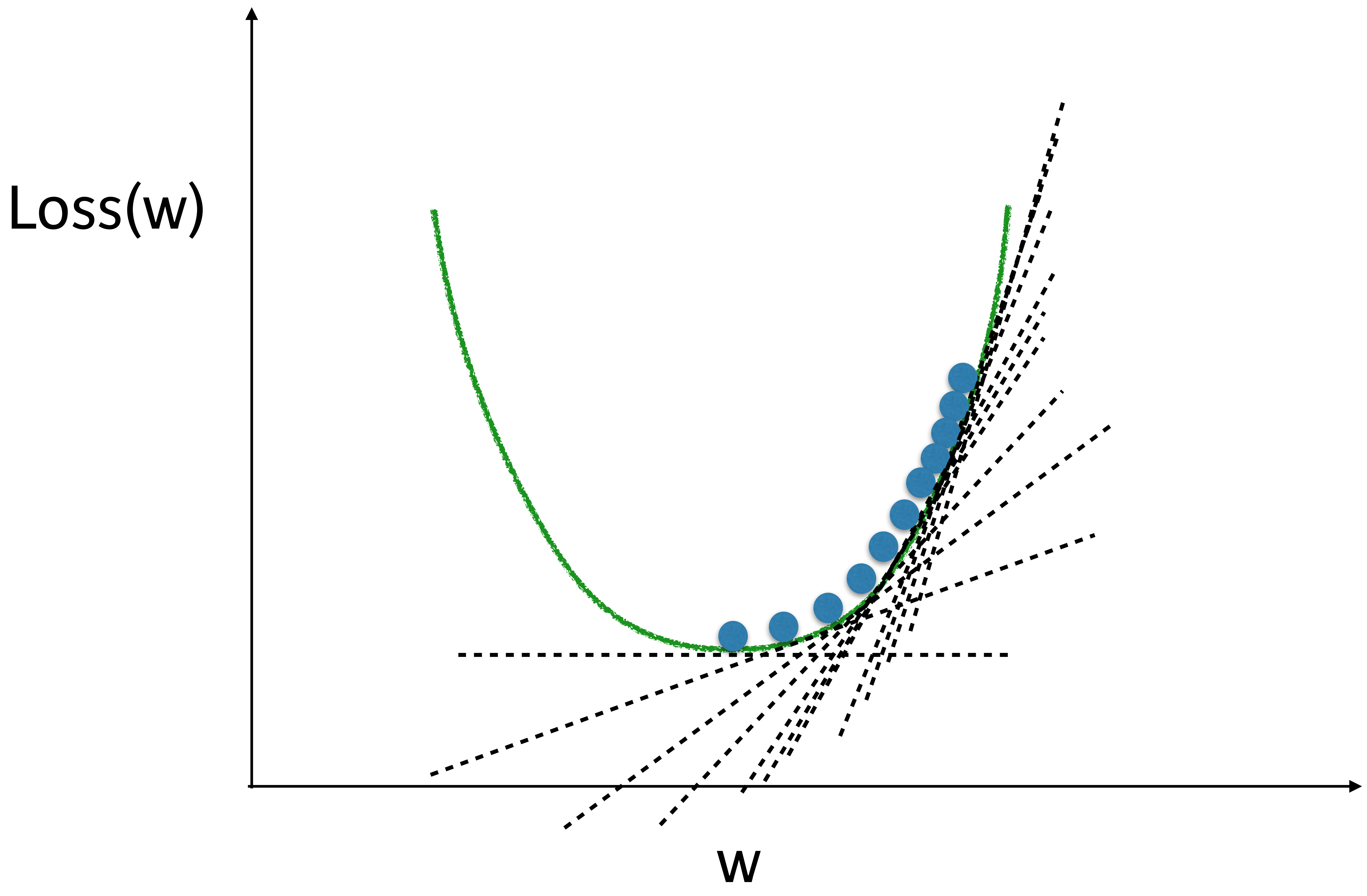
Optimizing a model with a single weight



Let's practice!

Gradient descent

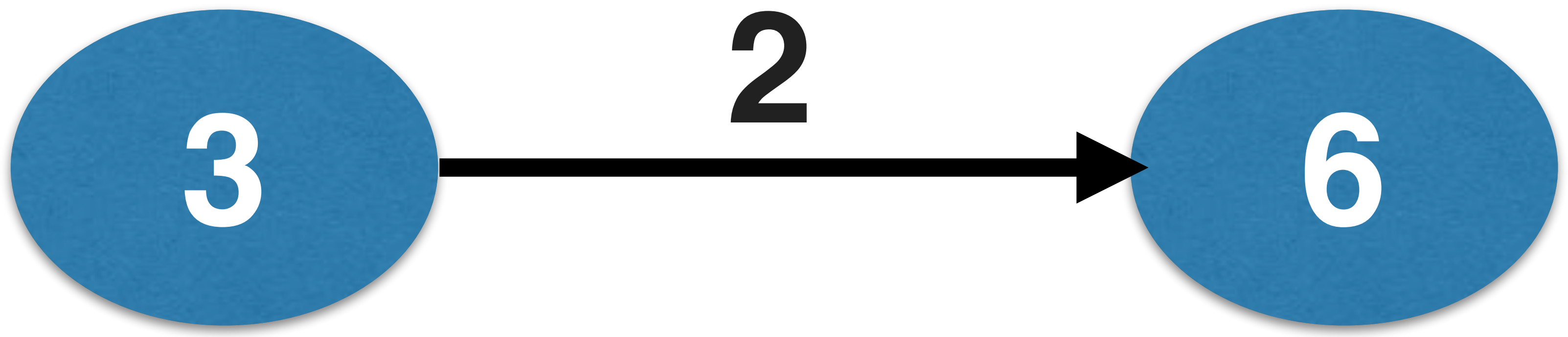
Gradient descent



Gradient descent

- If the slope is positive:
  - Going opposite the slope means moving to lower numbers
  - Subtract the slope from the current value
  - Too big a step might lead us astray
- Solution: learning rate
  - Update each weight by subtracting **learning rate * slope**

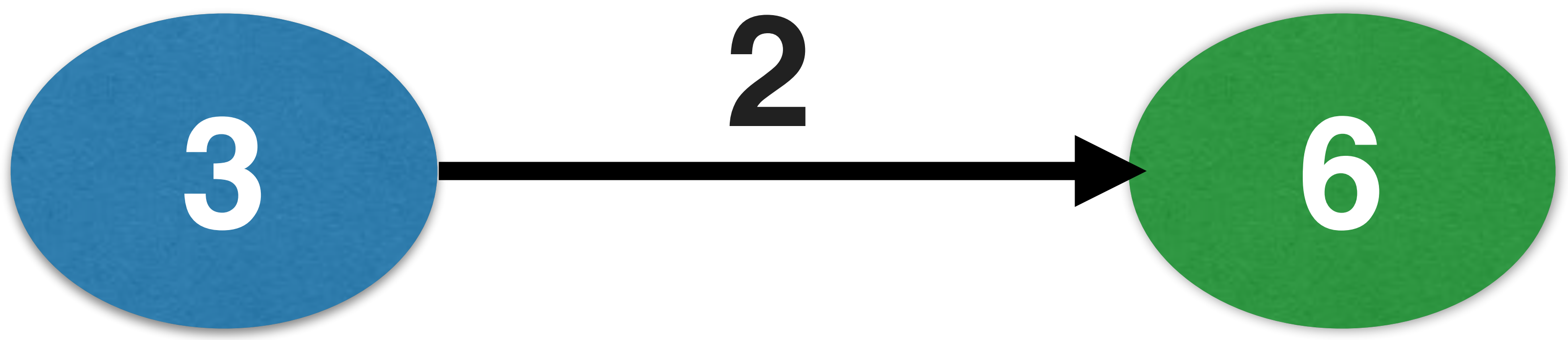
Slope calculation example



Actual Target Value = 10

- To calculate the slope for a weight, need to multiply:
    - Slope of the loss function w.r.t value at the node we feed into
    - The value of the node that feeds into our weight
    - Slope of the activation function w.r.t value we feed into

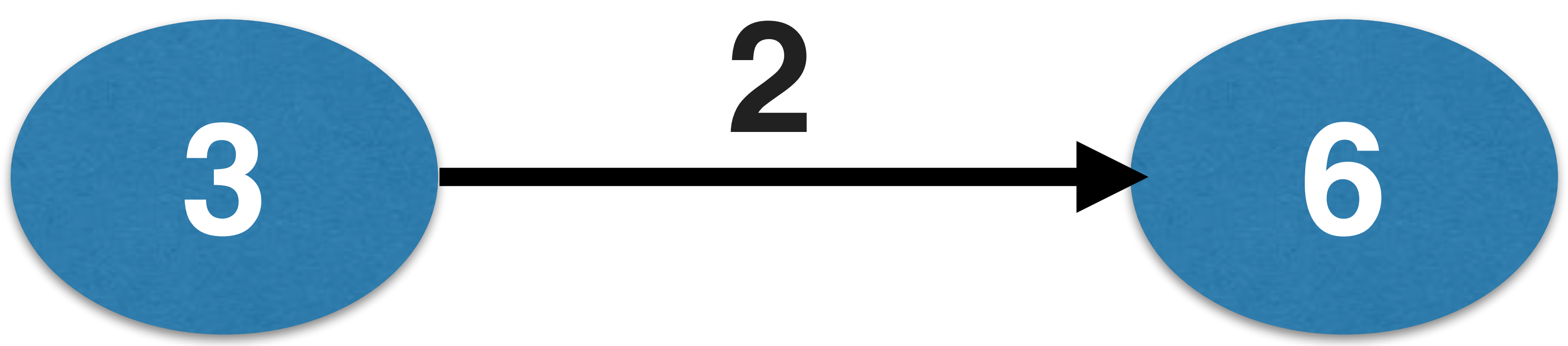
Slope calculation example



Actual Target Value = 10

- To calculate the slope for a weight, need to multiply:
  - Slope of the loss function w.r.t value at the node we feed into
  - The value of the node that feeds into our weight
  - Slope of the activation function w.r.t value we feed into

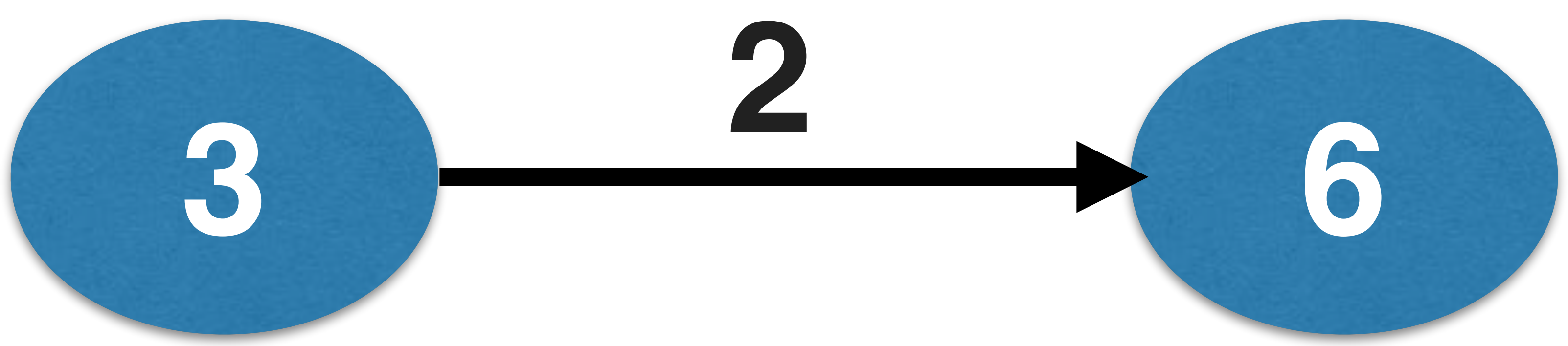
Slope calculation example



Actual Target Value = 10

- Slope of mean-squared loss function w.r.t prediction:
  - 2 * (Predicted Value - Actual Value) = 2 * Error
  - 2 * -4

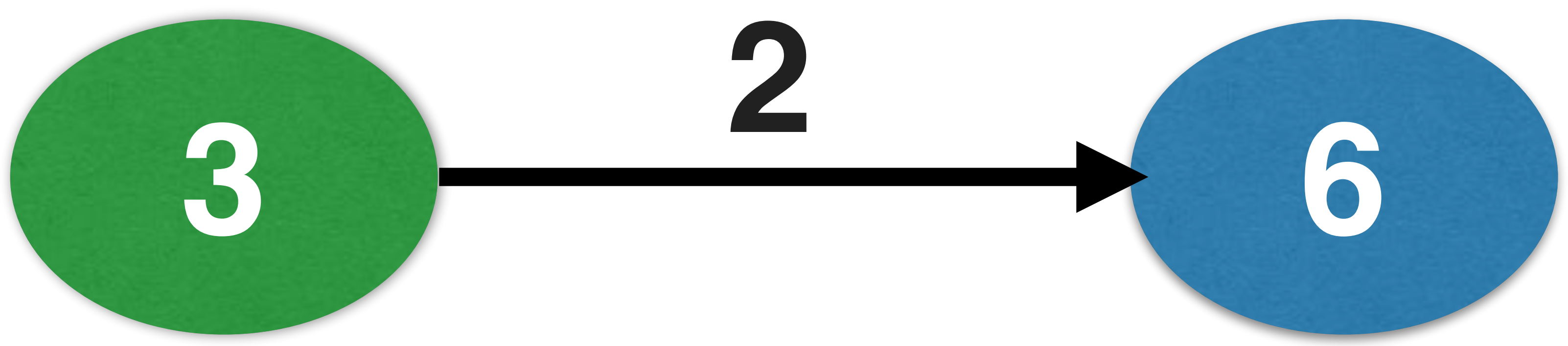
Slope calculation example



Actual Target Value = 10

- To calculate the slope for a weight, need to multiply:
    - Slope of the loss function w.r.t value at the node we feed into
    - The value of the node that feeds into our weight
    - Slope of the activation function w.r.t value we feed into

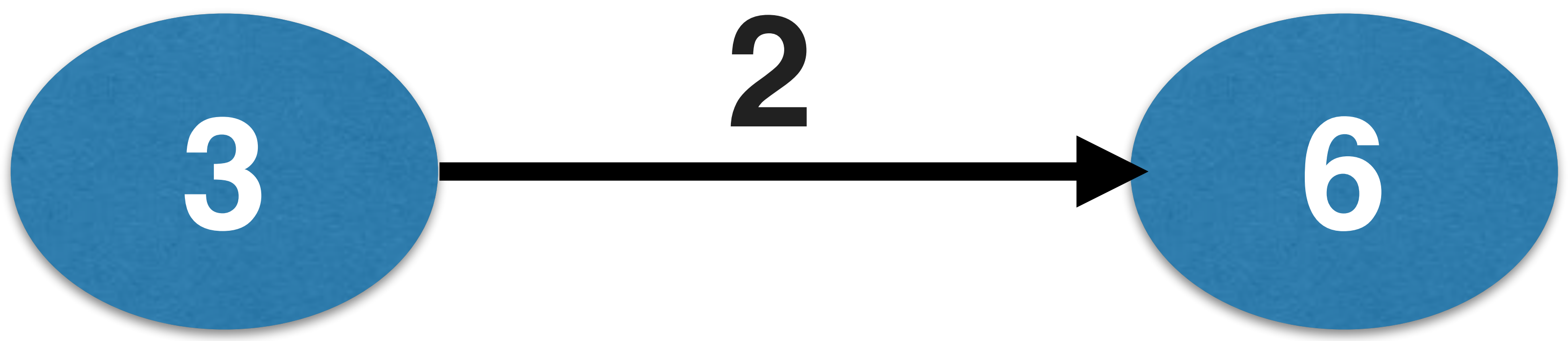
Slope calculation example



Actual Target Value = 10

- To calculate the slope for a weight, need to multiply:
    - Slope of the loss function w.r.t value at the node we feed into
    - The value of the node that feeds into our weight
    - Slope of the activation function w.r.t value we feed into

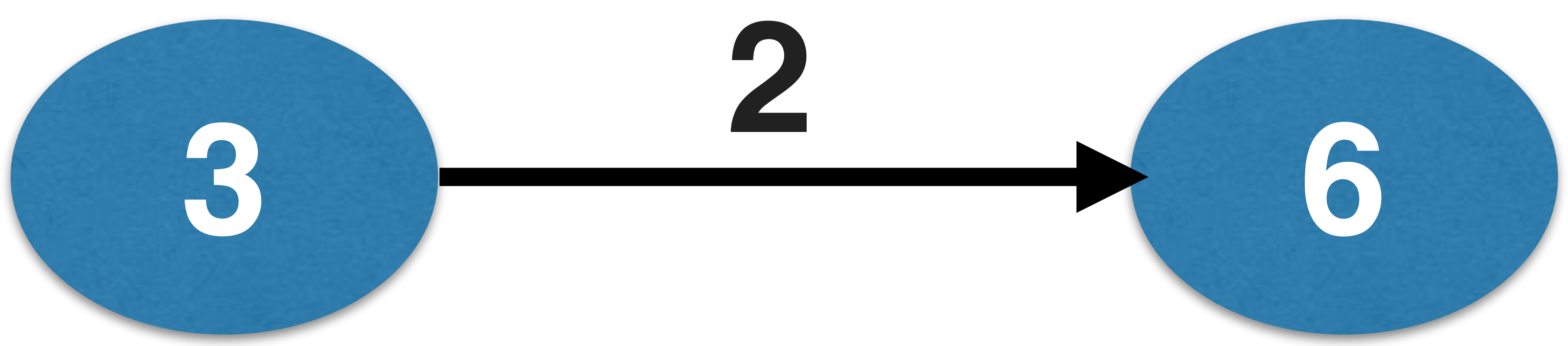
Slope calculation example



Actual Target Value = 10

- To calculate the slope for a weight, need to multiply:
  - Slope of the loss function w.r.t value at the node we feed into
  - The value of the node that feeds into our weight
  - ~~Slope of the activation function w.r.t value we feed into~~

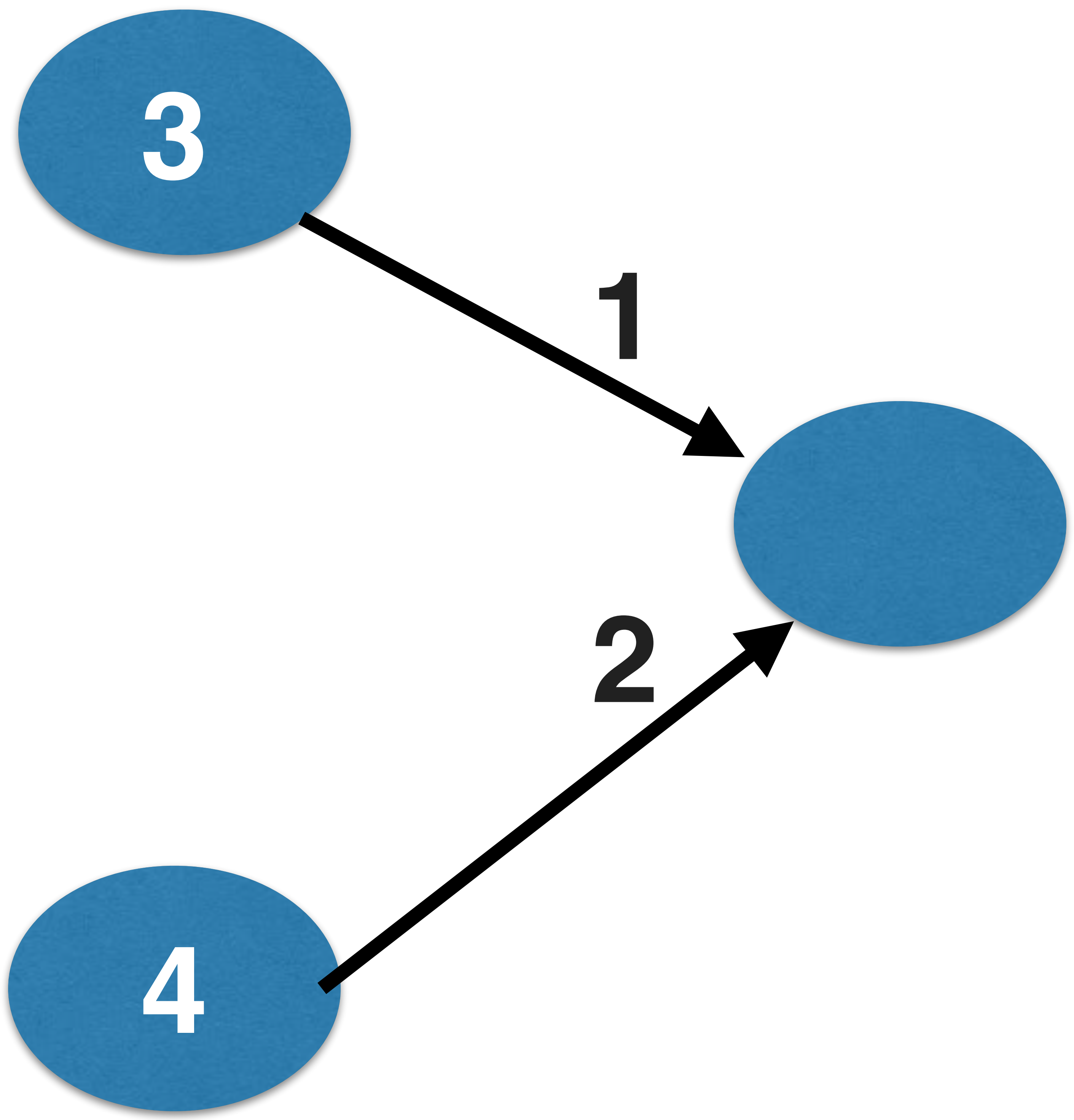
Slope calculation example



Actual Target Value = 10

- 2 * -4 * 3
- -24
- If learning rate is 0.01, the new weight would be
- 2 - 0.01(-24) = 2.24

Network with two inputs affecting prediction



Code to calculate slopes and update weights

```
In [1]: import numpy as np

In [2]: weights = np.array([1, 2])

In [3]: input_data = np.array([3, 4])

In [4]: target = 6

In [5]: learning_rate = 0.01

In [6]: preds = (weights * input_data).sum()

In [7]: error = preds - target

In [8]: print(error)
5
```

Code to calculate slopes and update weights

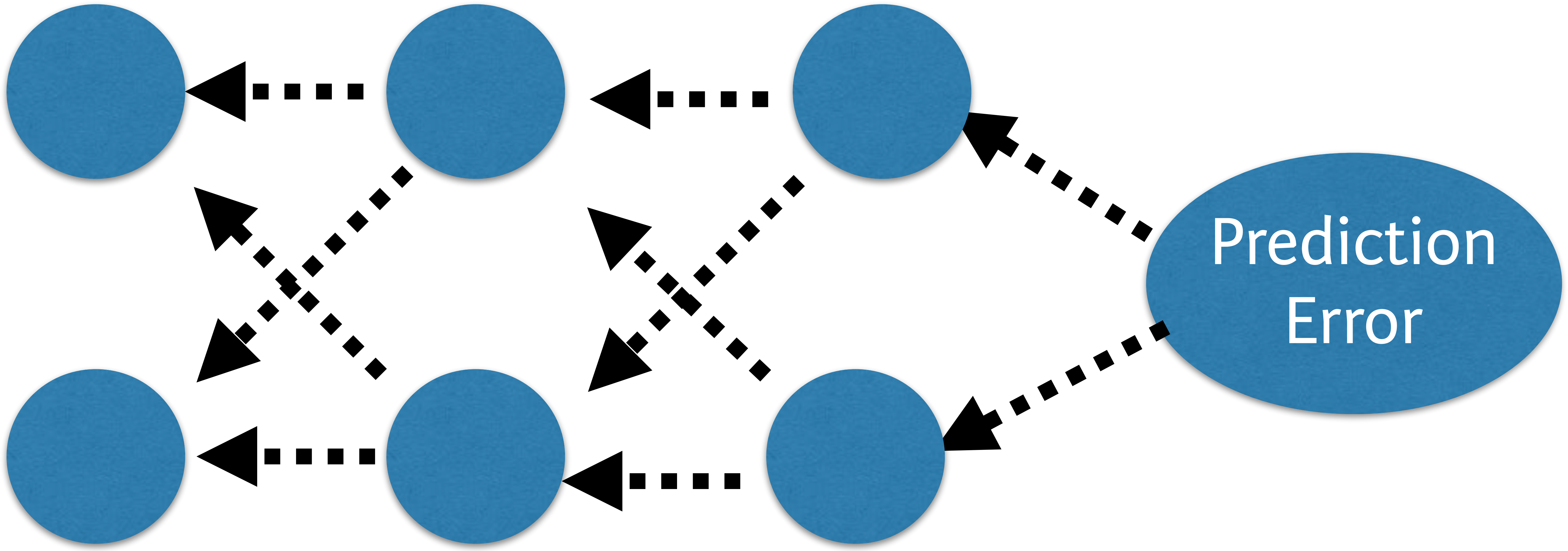
```
In [9]: gradient = 2 * input_data * error

In [10]: gradient
Out[10]: array([30, 40])

In [11]: weights_updated = weights - learning_rate * gradient

In [12]: preds_updated = (weights_updated * input_data).sum()

In [13]: error_updated = preds_updated - target

In [14]: print(error_updated)
-2.5
```

Let's practice!

Backpropagation

Backpropagation

- Allows gradient descent to update all weights in neural network (by getting gradients for all weights)
- Comes from chain rule of calculus
- Important to understand the process, but you will generally use a library that implements this



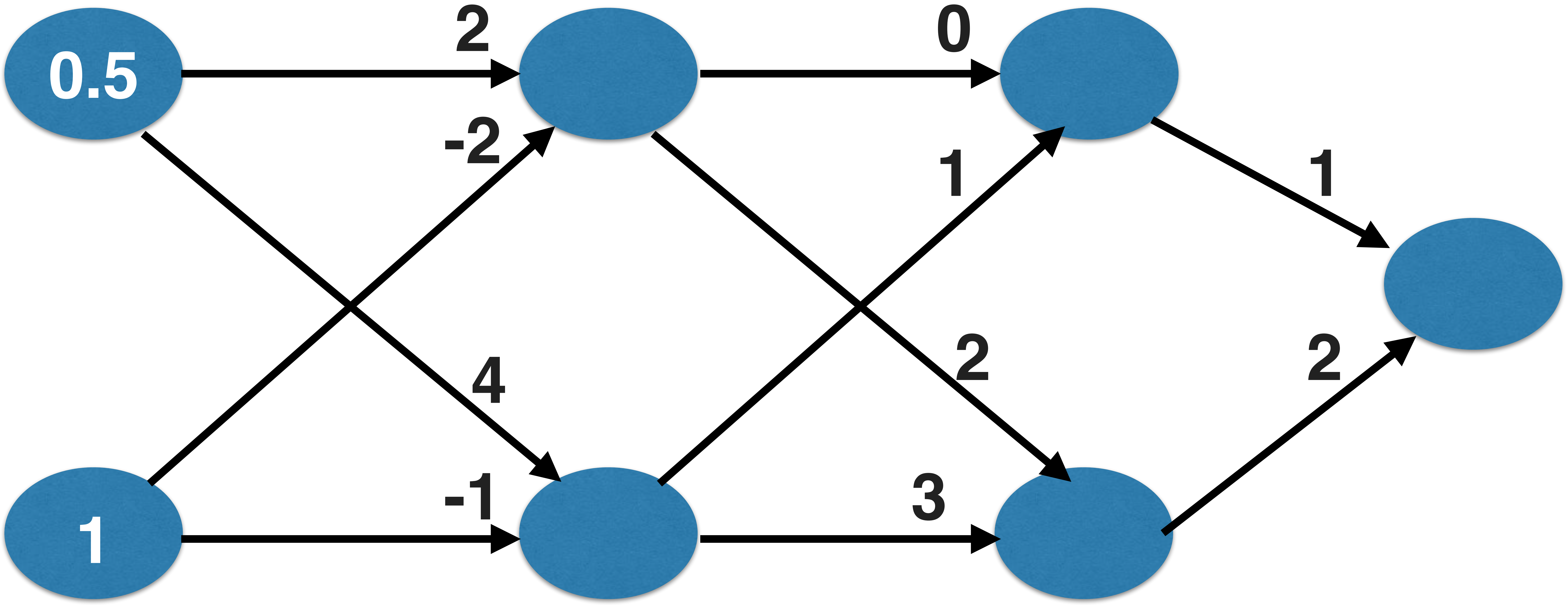
Backpropagation process

- Trying to estimate the slope of the loss function w.r.t each weight
- Do forward propagation to calculate predictions and errors

Backpropagation process

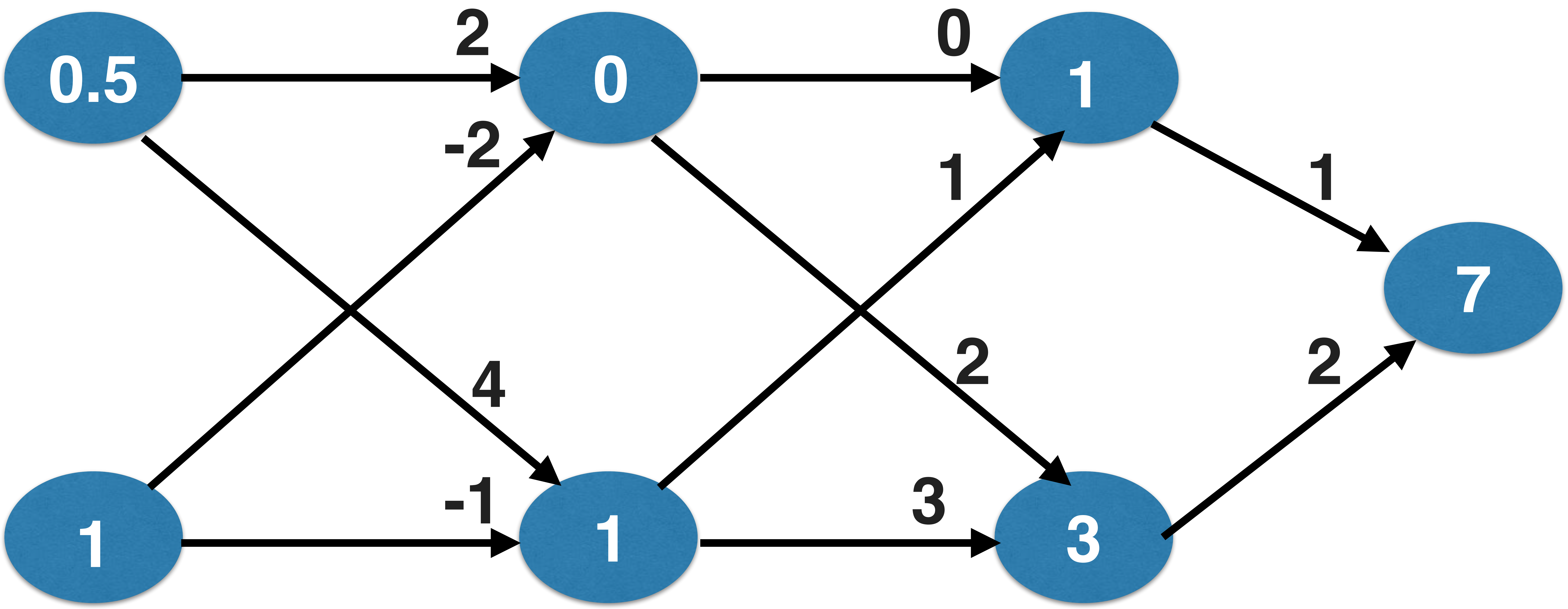
ReLU Activation Function

Actual Target Value = 4



Backpropagation process

ReLU Activation Function
Actual Target Value = 4
Error = 3

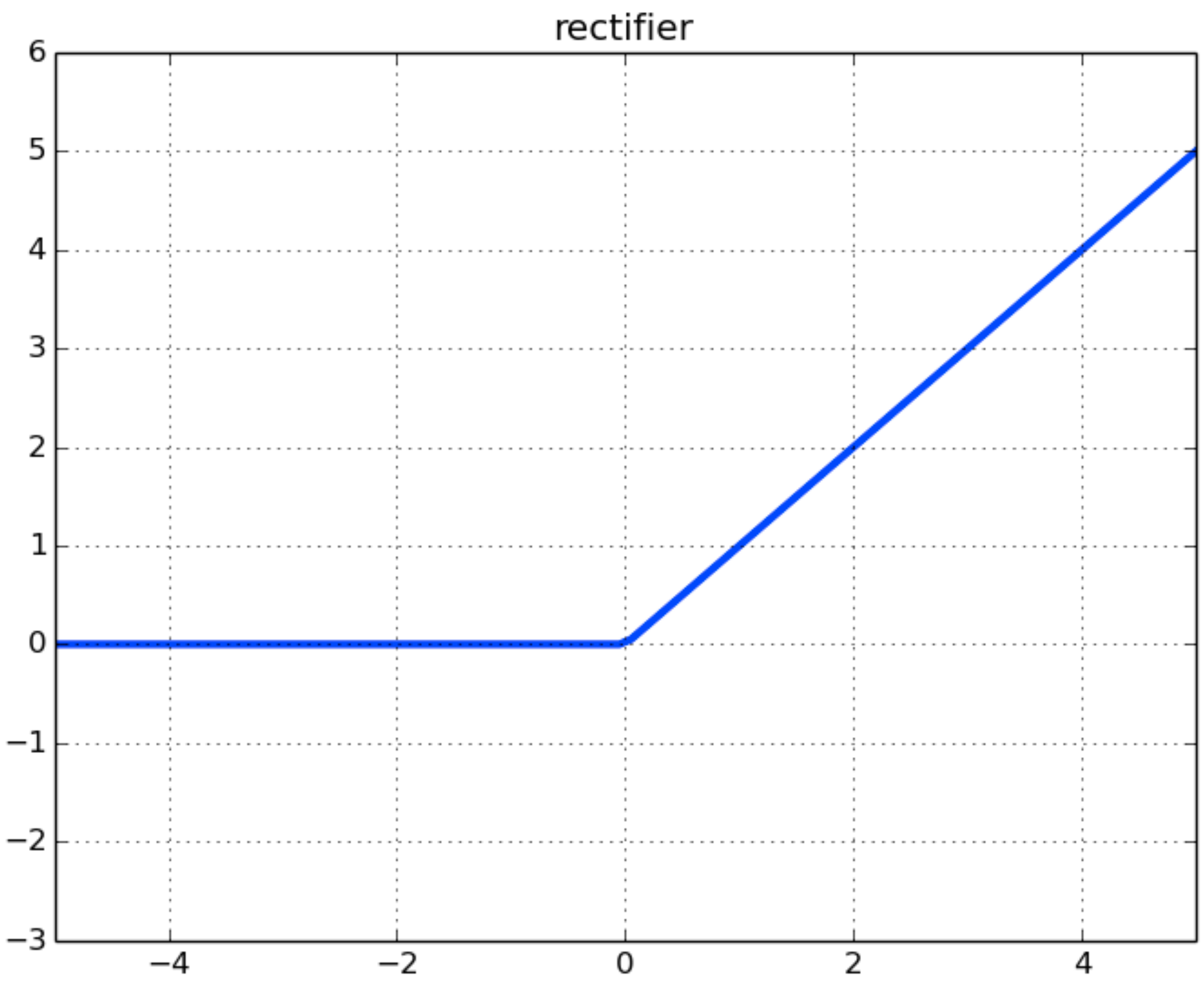


Backpropagation process

- Go back one layer at a time
- Gradients for weight is product of:

1. Node value feeding into that weight
2. Slope of loss function w.r.t node it feeds into
3. Slope of activation function at the node it feeds into

ReLU Activation Function



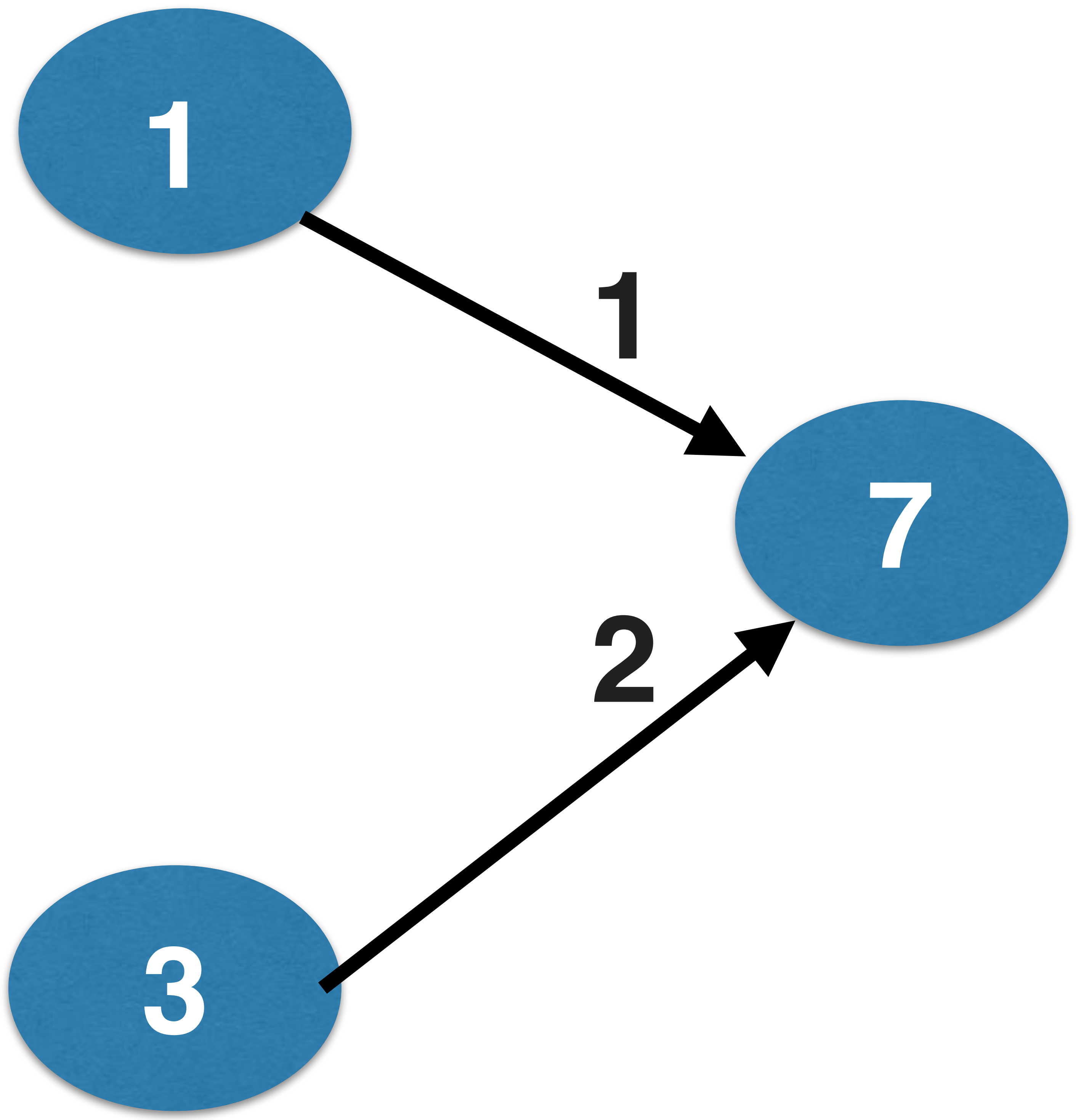
Backpropagation process

- Need to also keep track of the slopes of the loss function w.r.t node values
- Slope of node values are the sum of the slopes for all weights that come out of them

Let's practice!

Backpropagation in practice

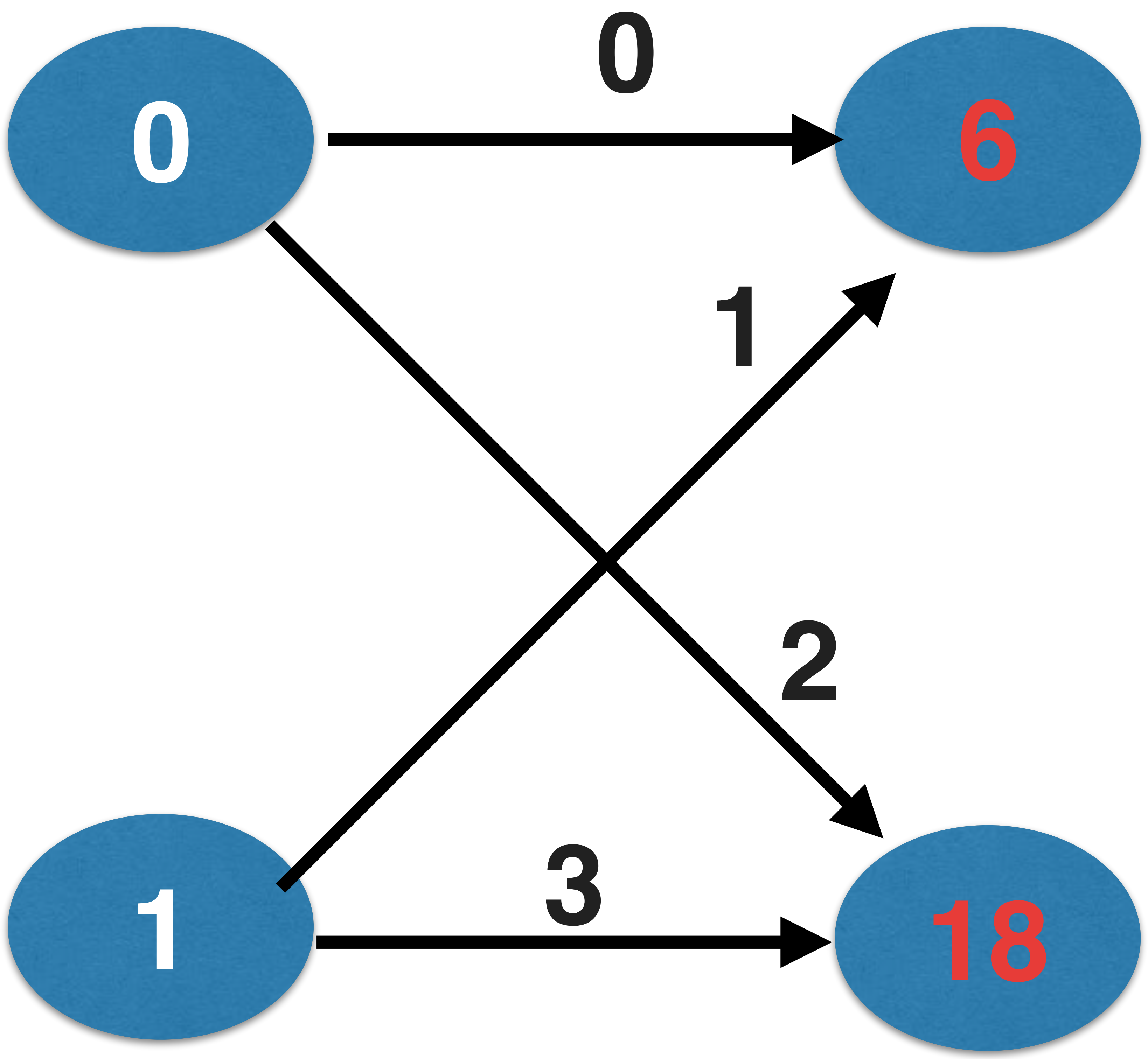
Backpropagation



ReLU Activation Function
Actual Target Value = 4
Error = 3

- Top weight's slope = 1 * 6
- Bottom weight's slope = 3 * 6

Backpropagation

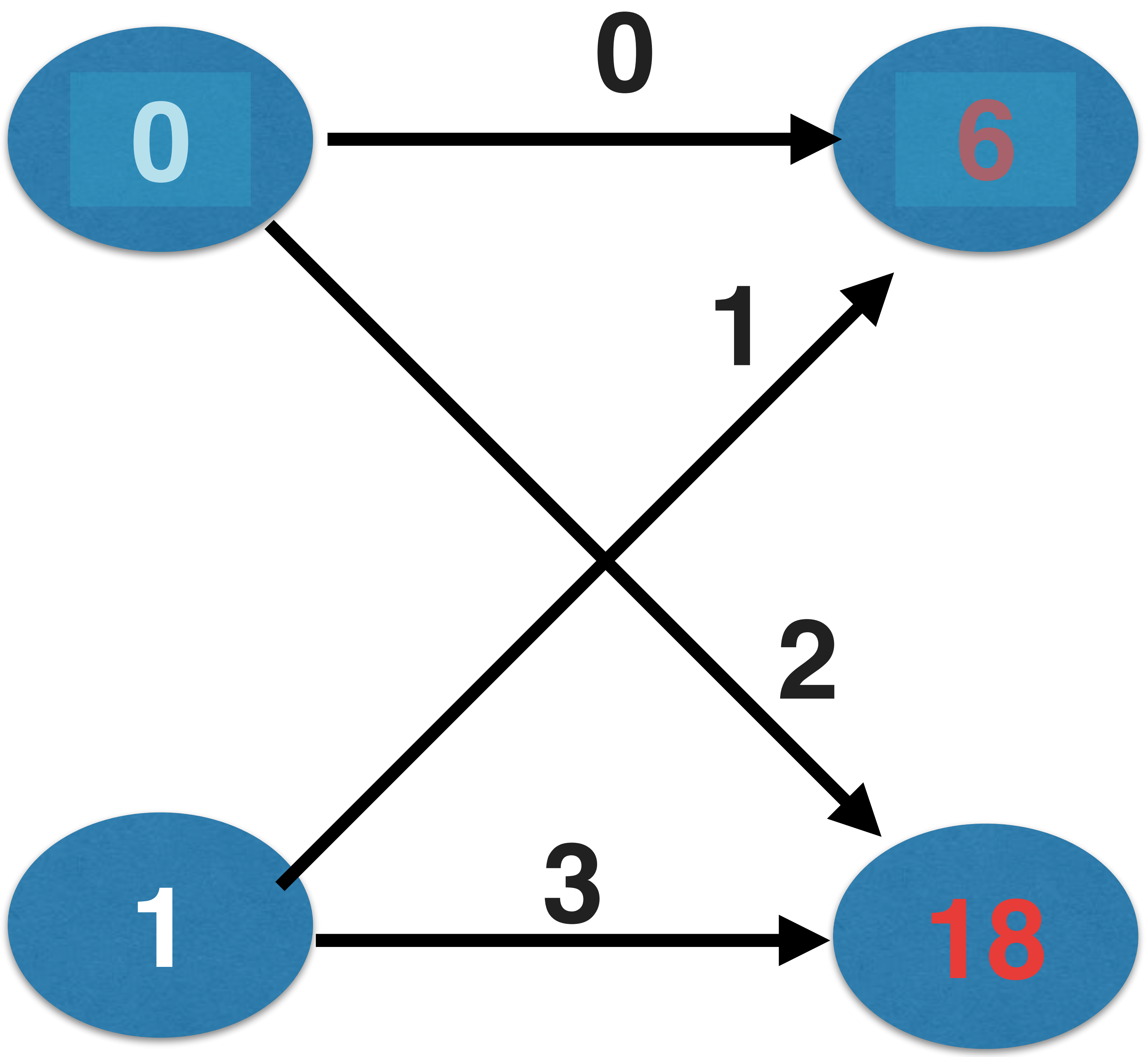


Calculating slopes associated with any weight

- Gradients for weight is product of:

1. Node value feeding into that weight
2. Slope of activation function for the node being fed into
3. Slope of loss function w.r.t output node

Backpropagation



Current Weight Value	Gradient
0	0
1	6
2	0
3	18

Backpropagation: Recap

- Start at some random set of weights
- Use forward propagation to make a prediction
- Use backward propagation to calculate the slope of the loss function w.r.t each weight
- Multiply that slope by the learning rate, and subtract from the current weights
- Keep going with that cycle until we get to a flat part

Stochastic gradient descent

- It is common to calculate slopes on only a subset of the data ('batch')
- Use a different batch of data to calculate the next update
- Start over from the beginning once all data is used
- Each time through the training data is called an epoch
- When slopes are calculated on one batch at a time: stochastic gradient descent

Let's practice