Introduction to Neural Learning

CHAPTER 4

Predict, Compare and Learn

- How do we set weight values so the network predicts accurately?

What is Compare?

- Comparing gives a measurement of how much a prediction “missed” by.

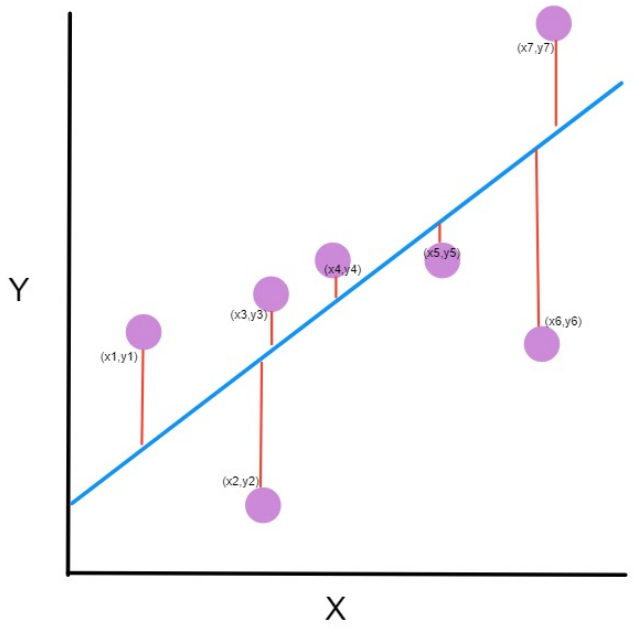
- Once you’ve made a prediction, the next step is to evaluate how well you did.
- However, to measure error is one of the most important and complicated subjects of deep learning.

Properties of Error

- Amplify bigger errors while ignoring very small ones.
- How to mathematically teach a network to do this.
- Error is always positive! - Analogy of an archer hitting a target. Whether the shot is too low by an inch or too high by an inch, the error is still just 1 inch.
- One simple way of measuring error: mean squared error.

Mean Squared Error

- Amplify bigger errors
- Reduce small errors
- It's but one of many ways to evaluate the accuracy of a neural network.
- This step will give you a sense for how much you missed, but that isn't enough to be able to learn.
- Given some prediction, it calculate an error measure that says either "a lot" or "a little."
- It won't tell you why you missed, what direction you missed, or what you should do to fix the error.
- It more or less says "big miss," "little miss," or "perfect prediction."

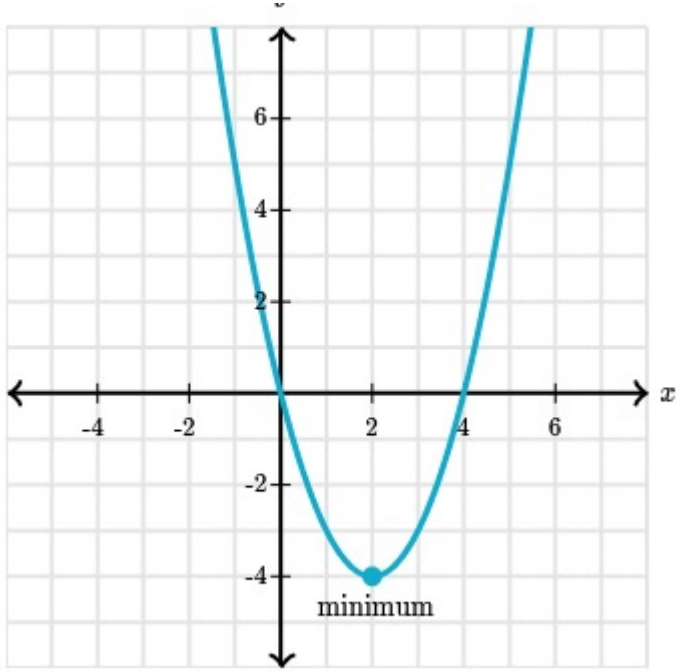


$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

$* n$ is the number of data points
$* Y_i$ represents observed values
$* \hat{Y}_i$ represents predicted values

Learning

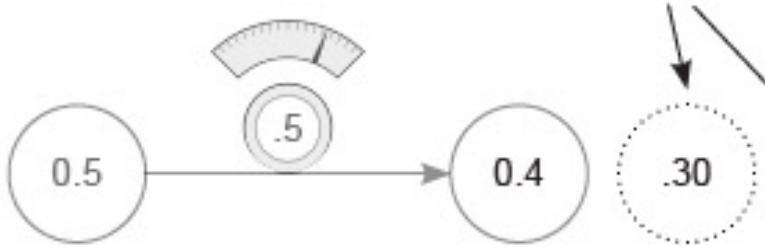
- Learning is all about error attribution, or the art of figuring out how each weight played its part in creating error.
- Algorithm: Gradient descent
- Gradient descent is a general-purpose algorithm that numerically finds minima of multivariable functions.
- It results in computing a number for each weight. That number represents how that weight should be higher or lower in order to reduce the error.
- Then
- you'll move the weight according to that number, and you'll be finished.



Compare: Does your network make good predictions?

Let's measure the error and find out!

Execute the following code in your Jupyter notebook. It should print `0.3025`:



The error is a way to measure how much you missed. There are multiple ways to calculate error, as you'll learn later. This one is *mean squared error*.

Error

```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight

error = (pred - goal_pred) ** 2

print(error)
```

Raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

What is the goal_pred variable?

Much like `input`, `goal_pred` is a number you recorded in the real world somewhere. But it's usually something hard to observe, like "the percentage of people who *did* wear sweatsuits," given the temperature; or "whether the batter *did* hit a home run," given his batting average.

Why is the error squared?

Think about an archer hitting a target. When the shot hits 2 inches too high, how much did the archer miss by? When the shot hits 2 inches too low, how much did the archer miss by? Both times, the archer missed by only 2 inches. The primary reason to *square* "how much you missed" is that it forces the output to be *positive*. `(pred - goal_pred)` could be negative in some situations, *unlike actual error*.

Doesn't squaring make big errors (>1) bigger and small errors (<1) smaller?

Yeah … It's kind of a weird way of measuring error, but it turns out that *amplifying* big errors and *reducing* small errors is OK. Later, you'll use this error to help the network learn, and you'd rather it *pay attention* to the big errors and not worry so much about the small ones. Good parents are like this, too: they practically ignore errors if they're small enough (breaking the lead on your pencil) but may go nuclear for big errors (crashing the car). See why squaring is valuable?

Why measure error?

Measuring error simplifies the problem.

The goal of training a neural network is to make correct predictions. That's what you want. And in the most pragmatic world (as mentioned in the preceding chapter), you want the network to take input that you can easily calculate (today's stock price) and predict things that are hard to calculate (tomorrow's stock price). That's what makes a neural network useful.

It turns out that changing `knob_weight` to make the network correctly predict `goal_prediction` is *slightly* more complicated than changing `knob_weight` to make `error == 0`. There's something more concise about looking at the problem this way. Ultimately, both statements say the same thing, but trying to *get the error to 0* seems more straightforward.

Different ways of measuring error *prioritize error differently.*

If this is a bit of a stretch right now, that's OK, but think back to what I said earlier: by *squaring* the error, numbers that are less than 1 get *smaller*, whereas numbers that are greater than 1 get *bigger*. You're going to change what I call *pure error* (`pred - goal_pred`) so that bigger errors become *very* big and smaller errors quickly become irrelevant.

By measuring error this way, you can *prioritize* big errors over smaller ones. When you have somewhat large pure errors (say, 10), you'll tell yourself that you have *very* large error (10**2 == 100); and in contrast, when you have small pure errors (say, 0.01), you'll tell yourself that you have *very* small error (0.01**2 == 0.0001). See what I mean about prioritizing? It's just modifying what you *consider to be error* so that you amplify big ones and largely ignore small ones.

In contrast, if you took the *absolute value* instead of squaring the error, you wouldn't have this type of prioritization. The error would just be the positive version of the pure error—which would be fine, but different. More on this later.

Why do you want only *positive* error?

Eventually, you'll be working with millions of `input -> goal_prediction` pairs, and we'll still want to make accurate predictions. So, you'll try to take the *average error* down to 0.

This presents a problem if the error can be positive and negative. Imagine if you were trying to get the neural network to correctly predict two datapoints—two `input -> goal_prediction` pairs. If the first had an error of 1,000 and the second had an error of –1,000, then the *average error* would be *zero*! You'd fool yourself into thinking you predicted perfectly, when you missed by 1,000 each time! That would be really bad. Thus, you want the error of *each prediction* to always be *positive* so they don't accidentally cancel each other out when you average them.

What's the simplest form of neural learning?

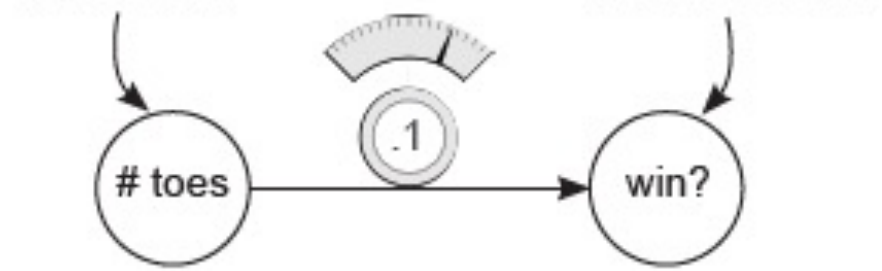
Learning using the hot and cold method.

At the end of the day, learning is really about one thing: adjusting `knob_weight` either up or down so the error is reduced. If you keep doing this and the error goes to 0, you're done learning! How do you know whether to turn the knob up or down? Well, you try *both up and down* and see which one reduces the error! Whichever one reduces the error is used to update `knob_weight`. It's simple but effective. After you do this over and over again, eventually error == 0, which means the neural network is predicting with perfect accuracy.

Hot and cold learning

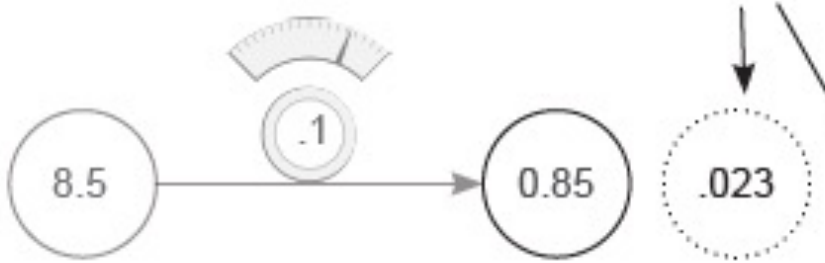
Hot and cold learning means wiggling the weights to see which direction reduces the error the most, moving the weights in that direction, and repeating until the error gets to 0.

❶ An empty network




```
weight = 0.1

lr = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

❷ PREDICT: Making a prediction and evaluating error



The error is a way to measure how much you missed. There are multiple ways to calculate error, as you'll learn later. This one is *mean squared error*.

Error

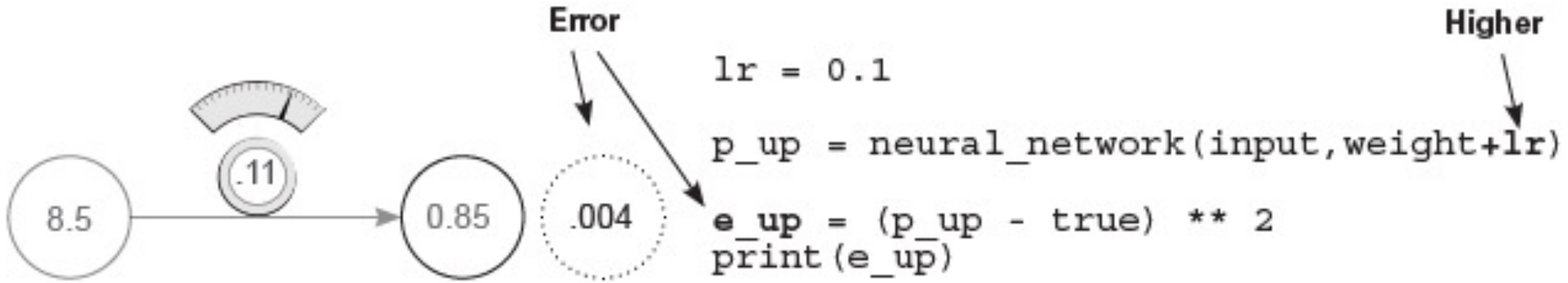
```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - true) ** 2
print(error)
```

Raw error

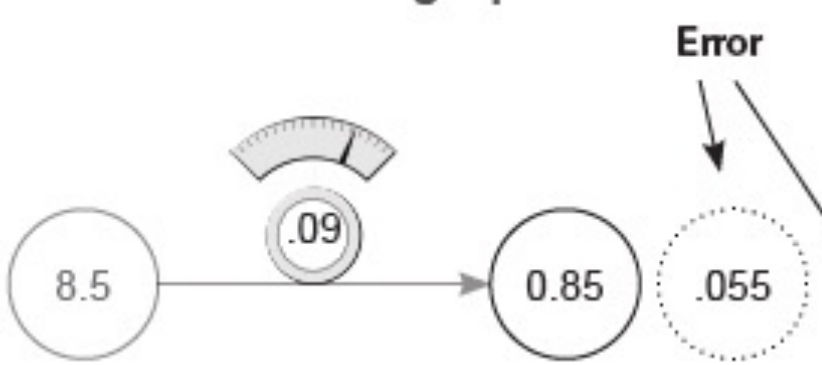
Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

❸ COMPARE: Making a prediction with a higher weight and evaluating error

We want to move the weight so the error goes downward. Let's try moving the weight up and down using `weight+lr` and `weight-lr`, to see which one has the lowest error.



❹ **COMPARE: Making a prediction with a lower weight and evaluating error**

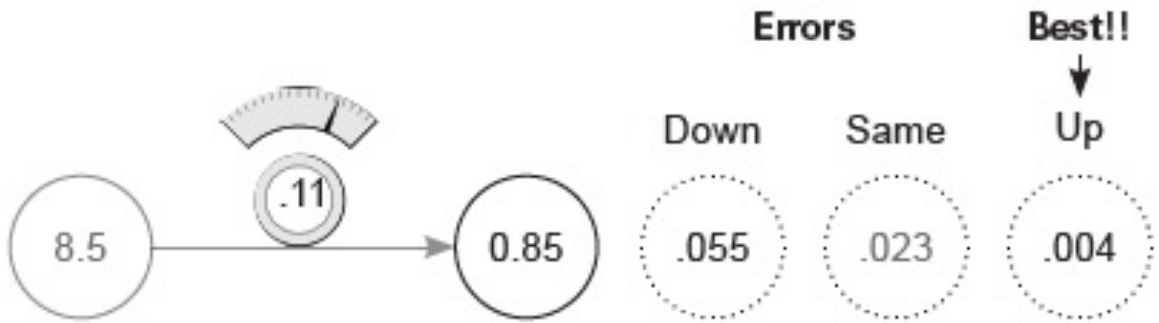


Error

Lower

```
lr = 0.01

p_dn = neural_network(input,weight-lr)

e_dn = (p_dn - true) ** 2
print(e_dn)
```

❺ COMPARE + LEARN: Comparing the errors and setting the new weight

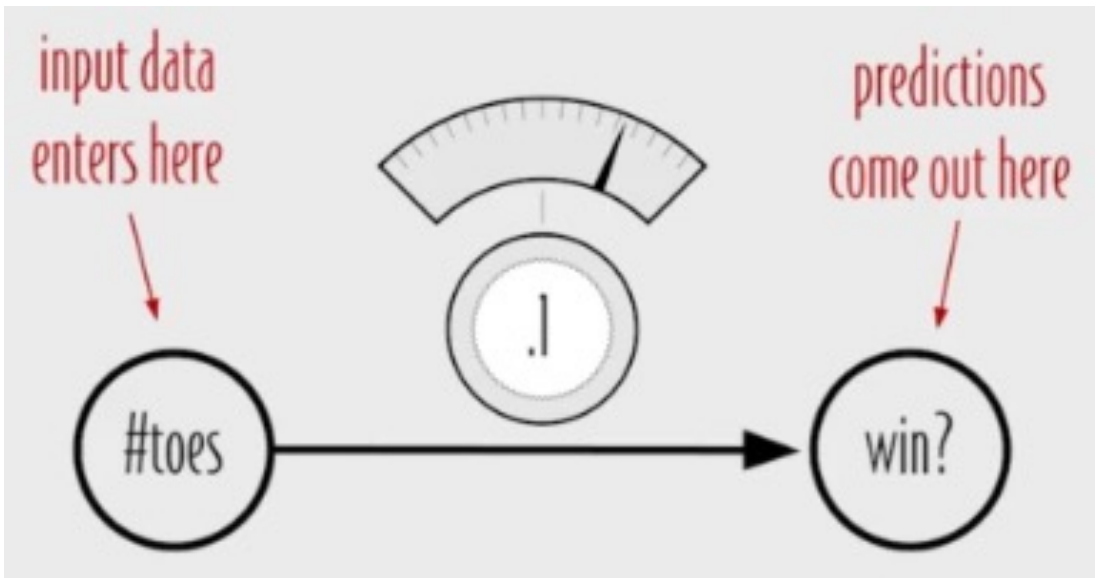



```
if(error > e_dn ||
   error > e_up):

  if(e_dn < e_up):
     weight -= lr

  if(e_up < e_up):
     weight += lr
```

These last five steps are one iteration of hot and cold learning. Fortunately, this iteration got us pretty close to the correct answer all by itself (the new error is only 0.004). But under normal circumstances, we'd have to repeat this process many times to find the correct weights. Some people have to train their networks for weeks or months before they find a good enough weight configuration.

This reveals what learning in neural networks really is: a *search problem*. You're *searching* for the best possible configuration of weights so the network's error falls to 0 (and predicts perfectly). As with all other forms of search, you might not find exactly what you're looking for, and even if you do, it may take some time. Next, we'll use hot and cold learning for a slightly more difficult prediction so you can see this searching in action!

```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction
```

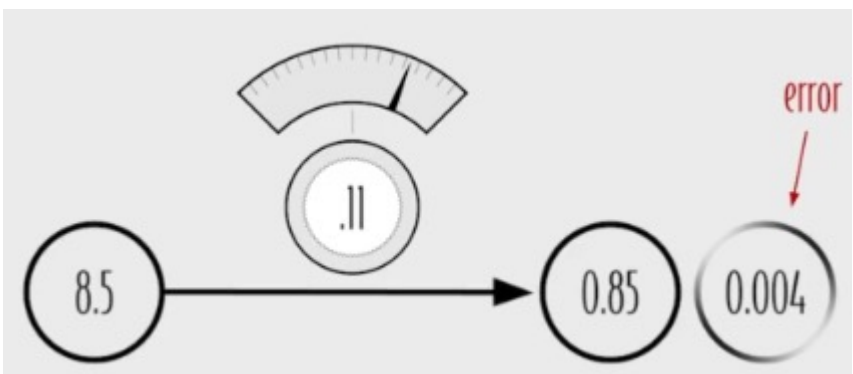


```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2
print(error)
```

```
0.02249999999999975
```

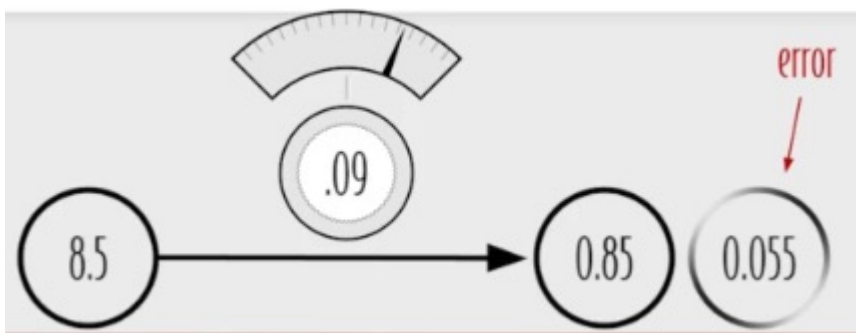
```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

print(error, up_error)
```

```
0.022499999999999975 0.00422499999999993
```



```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2
print(error, down_error)
```

```
0.022499999999999975 0.05522499999999994
```



```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

# Learning Steps
if(down_error < up_error):
    weight = weight - step_amount

if(down_error < up_error):
    weight = weight - step_amount

print(error, down_error, up_error)
```

Compare: Does our network make good predictions?

```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight
error = (pred - goal_pred) ** 2
print(error)
```

```
0.30250000000000005
```

Learning using the Hot and Cold Method

```
# 1) An Empty Network

weight = 0.1
lr = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction


# 2) PREDICT: Making A Prediction And Evaluating Error

number_of_toes = [8.5]
win_or_lose_binary = [1] #(won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

pred = neural_network(input,weight)
error = (pred - true) ** 2
print(error)
```

```
0.02249999999999975
```

```
# 3) COMPARE: Making A Prediction With a *Higher* Weight And Evaluating Error

weight = 0.1

def neural_network(input, weight):
    prediction = input * weight
    return prediction

number_of_toes = [8.5]
win_or_lose_binary = [1] #(won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

lr = 0.01
p_up = neural_network(input,weight+lr)
e_up = (p_up - true) ** 2
print(e_up)
```

```
0.00422499999999993
```

```
# 4) COMPARE: Making A Prediction With a *Lower* Weight And Evaluating Error

weight = 0.1

def neural_network(input, weight):
    prediction = input * weight
    return prediction

number_of_toes = [8.5]
win_or_lose_binary = [1] #(won!!!)

input = number_of_toes[0]
true = win_or_lose_binary[0]

lr = 0.01
p_dn = neural_network(input,weight-lr)
e_dn = (p_dn - true) ** 2
print(e_dn)
```

```
0.05524999999999994
```

```
weight = 0.5
input = 0.5
goal_prediction = 0.8

step_amount = 0.001

for iteration in range(1101):

    prediction = input * weight
    error = (prediction - goal_prediction) ** 2

    print("Error:" + str(error) + " Prediction:" + str(prediction))

    up_prediction = input * (weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2

    down_prediction = input * (weight - step_amount)
    down_error = (goal_prediction - down_prediction) ** 2

    if(down_error < up_error):
        weight = weight - step_amount

    if(down_error > up_error):
        weight = weight + step_amount
```

How much to move the weights each iteration

Repeat learning many times so the error can keep getting smaller.

Try up!

Try down!

If down is better, go down!

If up is better, go up!

```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

def neural_network(input,weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

# Learning Steps
if(down_error < up_error):
    weight = weight - step_amount

if(down_error < up_error):
    weight = weight - step_amount

print(error, down_error, up_error)
```

```
0.022499999999999975 0.05522499999999994 0.00422499999999993
```

Why did I iterate exactly 1,101 times?

The neural network in the example reaches 0.8 after exactly that many iterations. If you go past that, it wiggles back and forth between 0.8 and just above or below 0.8, making for a less pretty error log printed at the bottom of the left page. Feel free to try it.

Problem 1: It's inefficient.

You have to predict *multiple times* to make a single `knob_weight` update. This seems very inefficient.

Problem 2: Sometimes it's impossible to predict the exact goal prediction.

With a set `step_amount`, unless the perfect `weight` is exactly `n*step_amount` away, the network will eventually overshoot by some number less than `step_amount`. When it does, it will then start alternating back and forth between each side of `goal_prediction`. Set `step_amount` to 0.2 to see this in action. If you set `step_amount` to 10, you'll really break it. When I try this, I see the following output. It never remotely comes close to 0.8!

```
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
Error:19.8025 Prediction:5.25
Error:0.3025 Prediction:0.25
....
.... repeating infinitely...
```

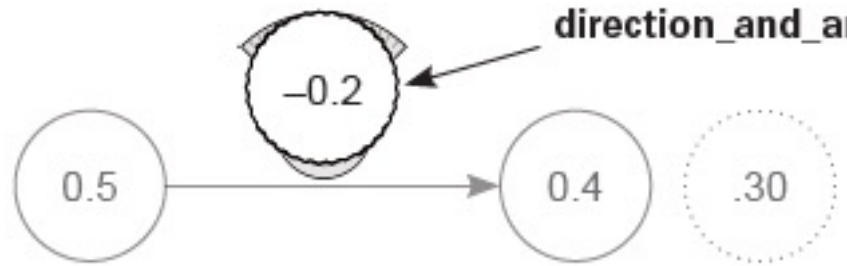
The real problem is that even though you know the correct *direction* to move `weight`, you don't know the correct *amount*. Instead, you pick a fixed one at random (`step_amount`). Furthermore, this amount has *nothing* to do with `error`. Whether `error` is big or tiny, `step_amount` is the same. So, hot and cold learning is kind of a bummer. It's inefficient because you predict three times for each `weight` update, and `step_ amount` is arbitrary, which can prevent you from learning the correct `weight` value.

What if you had a way to compute both direction and amount for each `weight` without having to repeatedly make predictions?

Calculating both direction and amount from error

Let's measure the error and find the direction and amount!

Execute this code in your Jupyter notebook:




```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print("Error:" + str(error) + " Prediction:" + str(pred))
```

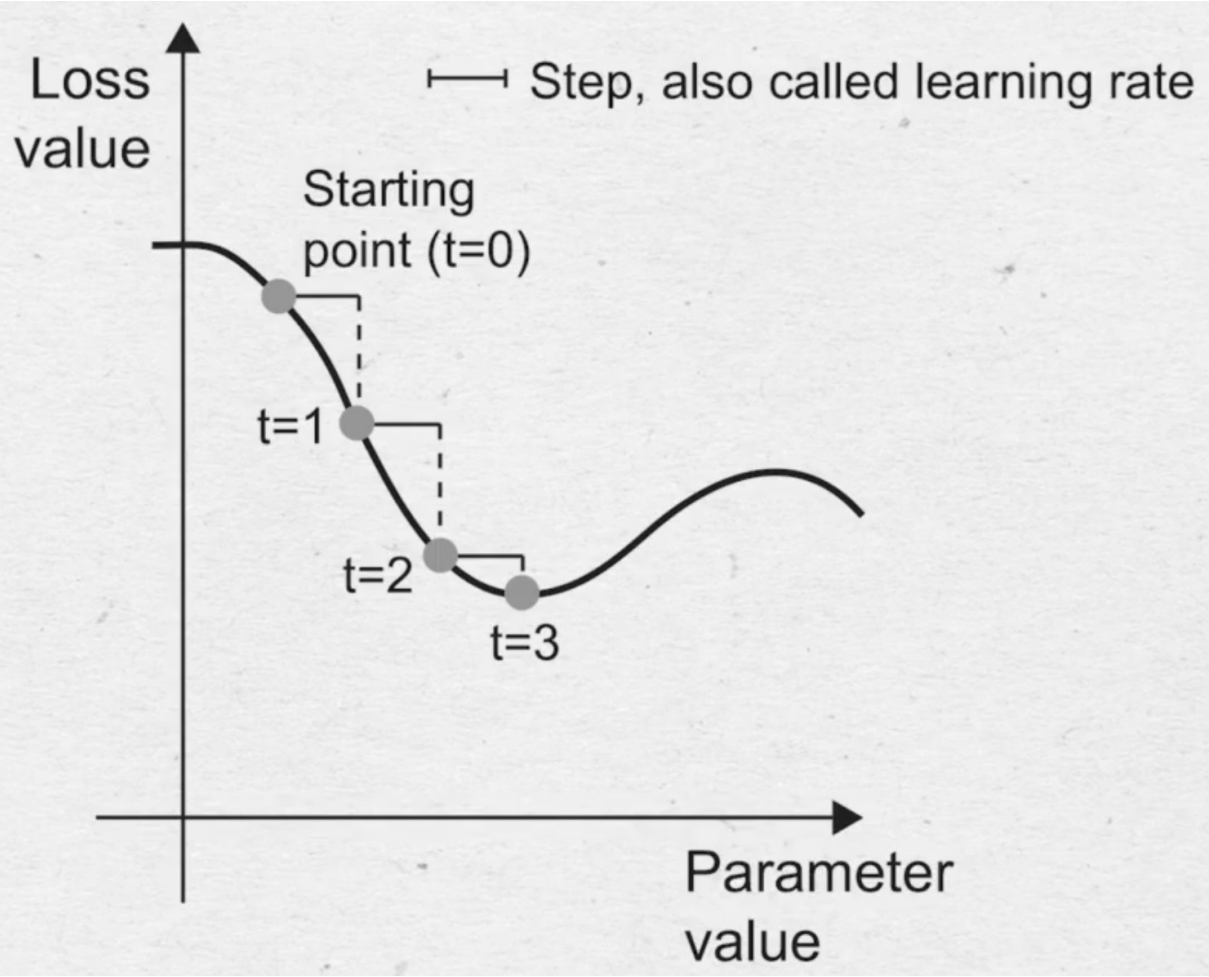
❶ Pure error

❷ Scaling, negative reversal, and stopping

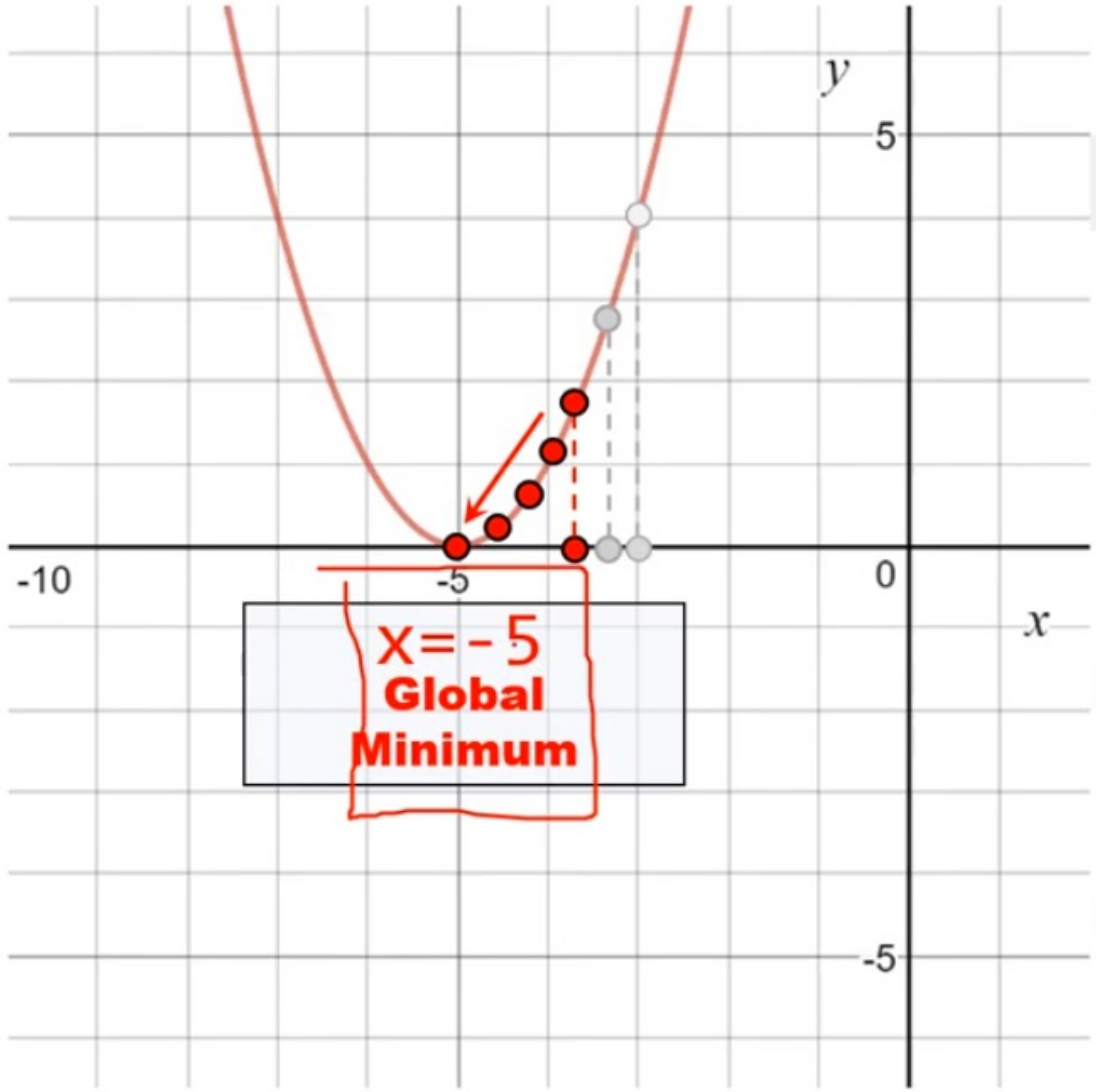
What you see here is a superior form of learning known as *gradient descent*. This method allows you to (in a single line of code, shown here in **bold**) calculate both the *direction* and the *amount* you should change `weight` to reduce `error`.

Gradient Desent

It is an optimization algorthim to find the minimum of a function.



In order to go down – we need to take steps opposite to direction and amount



$y=(x+5)^2$

#Step3

Perform **2** iterations of gradient descent.

 # Initialize Parameters

$X_0 = -3$

learning_rate = 0.01

$dy/dx = 2\times(x+5)$

 # Iteration 1

$X_1 = X_0 - (\text{learning\_rate}) \times (dy/dx)$

$X_1 = (-3) - (0.01) \times (2 \times ((-3)+5)) = -3.04$

 # Iteration 2

$X_2 = X_1 - (\text{learning\_rate}) \times (dy/dx)$

$X_1 = (-3.04) - (0.01) \times (2 \times ((-3.04)+5)) =$

$= -3.0792$

```
weight = 0.5
input = 0.5 # number of toes
goal_pred = 0.8 # represent a win
#step_amount = 0.001 # how much to move aour weights in each iteration


for iteration in range (20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print("Error: " + str(error) + " prediction: " + str(pred))
```

```
Error: 0.3025000000000005 prediction: 0.25
Error: 0.1701562500000004 prediction: 0.3875
Error: 0.095712890625 prediction: 0.4906250000000003
Error: 0.0538385097656251 prediction: 0.56796875
Error: 0.0302841567993164 2 prediction: 0.6259765625
Error: 0.017034838199615 5 prediction: 0.669482421875
Error: 0.009582096487283 72 prediction: 0.70211181640625
Error: 0.005389929274097089 prediction: 0.726583862304687 5
Error: 0.003031835216679615 3 prediction: 0.744937896728515 6
Error: 0.001705407309382288 2 prediction: 0.758703422546386 7
Error: 0.000959291611527537 1 prediction: 0.76902756690979
Error: 0.000539601531484238 4 prediction: 0.776770675182342 6
Error: 0.000303525861459885 prediction: 0.782578006386756 9
Error: 0.000170733297071186 78 prediction: 0.786933504790067 6
Error: 9.603747960254256e-05 prediction: 0.790200128592550 7
Error: 5.402108227642978e-05 prediction: 0.792650096444413 1
Error: 3.038685878049206e-05 prediction: 0.794487572333309 8
Error: 1.709260806402724 2e-05 prediction: 0.795865679249982 3
Error: 9.614592036015323e-06 prediction: 0.796899259437486 7
Error: 5.408208020258491e-06 prediction: 0.797674445781151
```

What is direction_and_amount?

`direction_and_amount` represents how you want to change `weight`. The first part ❶ is what I call *pure error*, which equals (`pred - goal_pred`). (More about this shortly.) The second part ❷ is the multiplication by the `input` that performs scaling, negative reversal, and stopping, modifying the pure error so it's ready to update `weight`.

What is the pure error?

The pure error is (`pred - goal_pred`), which indicates the raw direction and amount you missed. If this is a *positive* number, you predicted too *high*, and vice versa. If this is a *big* number, you missed by a *big* amount, and so on.

What are scaling, negative reversal, and stopping?

These three attributes have the combined effect of translating the pure error into the absolute amount you want to change `weight`. They do so by addressing three major edge cases where the pure error isn't sufficient to make a good modification to `weight`.

What is direction and amount?

This is how we can change our weight

1. Pure Error – (pred - goal_pred)

Pure Error – Raw direction and amount we missed.

If this is a +ve number – we predicted to high
If this is a -ve number – we predicted to low

If this is a big number – we missed by a big amount
If this is a small number – we missed by a small amount

2. weight – direction_amount

New weight is calculated by subtracting the direction_amount instead of adding it.

If direction_amount is a +ve number. It means we predicted too high. Therefore, weight needs to be smaller. So We subtract the -ve number.

If direction_amount is a -ve number. It means we predicted too low. Therefore, weight needs to be bigger. So We subtract the -ve number from weight which becomes bigger as –ve –ve add together.

3. * input

It performs the following.

1. Stopping
2. Negative reversal
3. Scaling

These three attributes have the combined effect of translating the pure error into the absolute amount you want to change weight. They do so by addressing three major edge cases where the pure error isn't sufficient to make a good modification to weight.

What is stopping?

Stopping is the first (and simplest) effect on the pure error caused by multiplying it by `input`. Imagine plugging a CD player into your stereo. If you turned the volume all the way up but the CD player was off, the volume change wouldn't matter. Stopping addresses this in a neural network. If `input` is 0, then it will force `direction_and_amount` to also be 0. You don't learn (change the volume) when `input` is 0, because there's nothing to learn. Every `weight` value has the same `error`, and moving it makes no difference because `pred` is always 0.

What is negative reversal?

This is probably the most difficult and important effect. Normally (when `input` is positive), moving `weight` upward makes the prediction move upward. But if `input` is negative, then all of a sudden `weight` changes directions! When `input` is negative, moving `weight` *up* makes the prediction go *down*. It's reversed! How do you address this? Well, multiplying the pure error by `input` will *reverse the sign* of `direction_and_amount` in the event that `input` is negative. This is *negative reversal*, ensuring that `weight` moves in the correct direction even if `input` is negative.

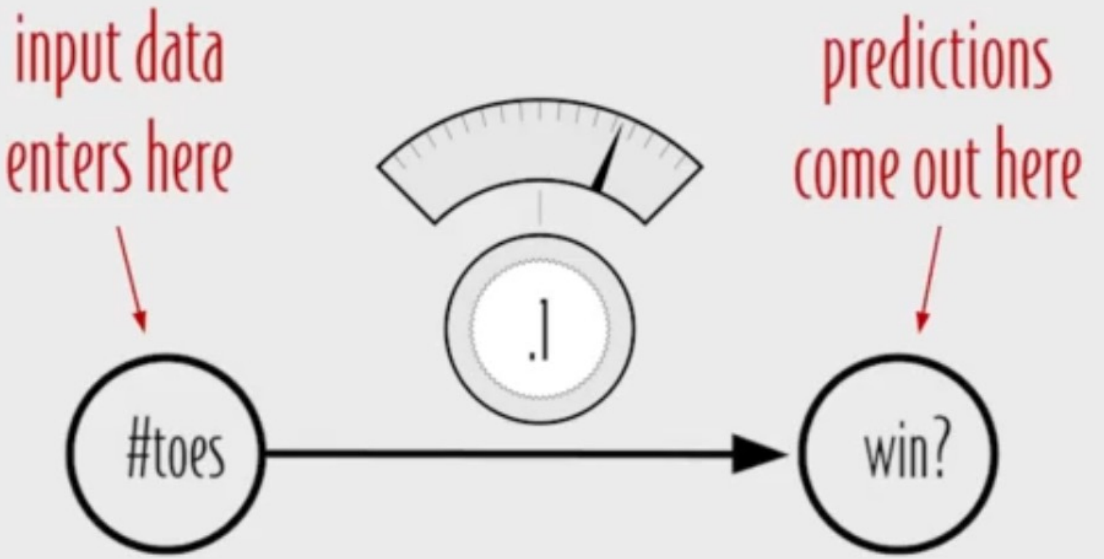
What is scaling?

Scaling is the third effect on the pure error caused by multiplying it by `input`. Logically, if `input` is big, your `weight` update should also be big. This is more of a side effect, because it often goes out of control. Later, you'll use *alpha* to address when that happens.

Gradient Desent – at any given time, it points to the best direction we need to move.

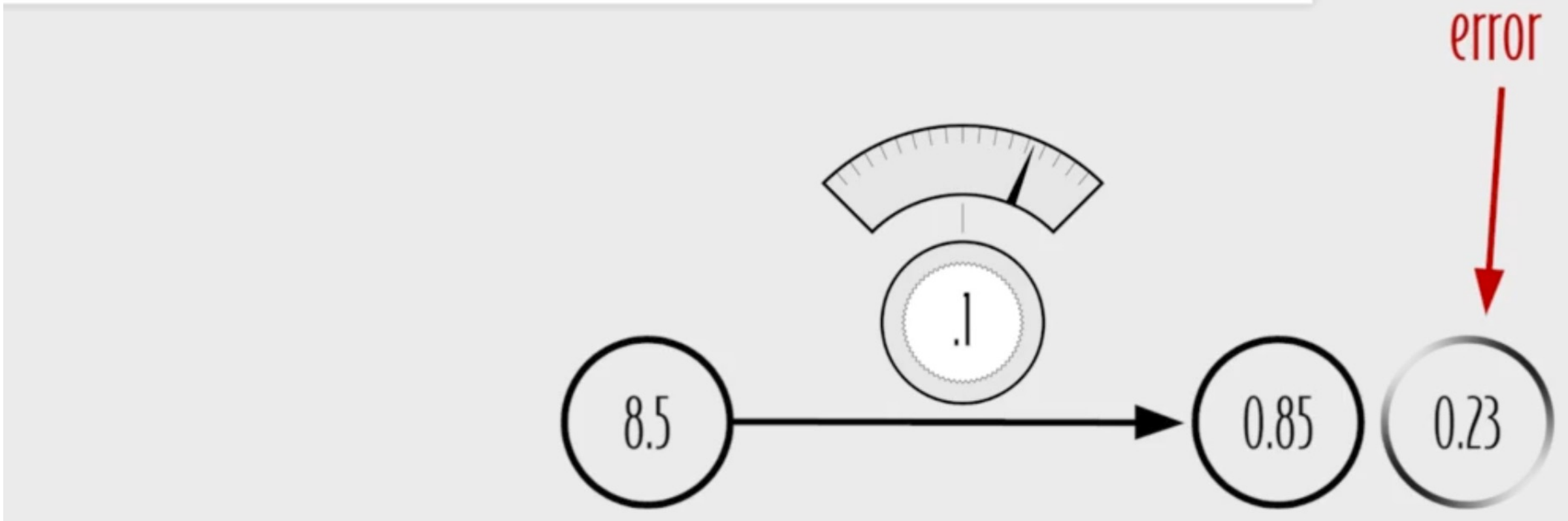
```
weight = 0.1
input = 8.5  # number of toes
goal_prediction = 1 # represents a win
step_amount = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input, weight)
error = (prediction - goal_prediction) ** 2

up_prediction = neural_network(input, weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input, weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

if(down_error < up_error):
    weight = weight - step_amount
if(down_error > up_error):
    weight = weight + step_amount

print error, down_error, up_error
```

```
weight = 0.1
input = 8.5  # number of toes
goal_prediction = 1 # represents a win
alpha = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input, weight)
error = (prediction - goal_prediction) ** 2

up_prediction = neural_network(input, weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input, weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

if(down_error < up_error):
    weight = weight - step_amount
if(down_error > up_error):
    weight = weight + step_amount

print error, down_error, up_error
```



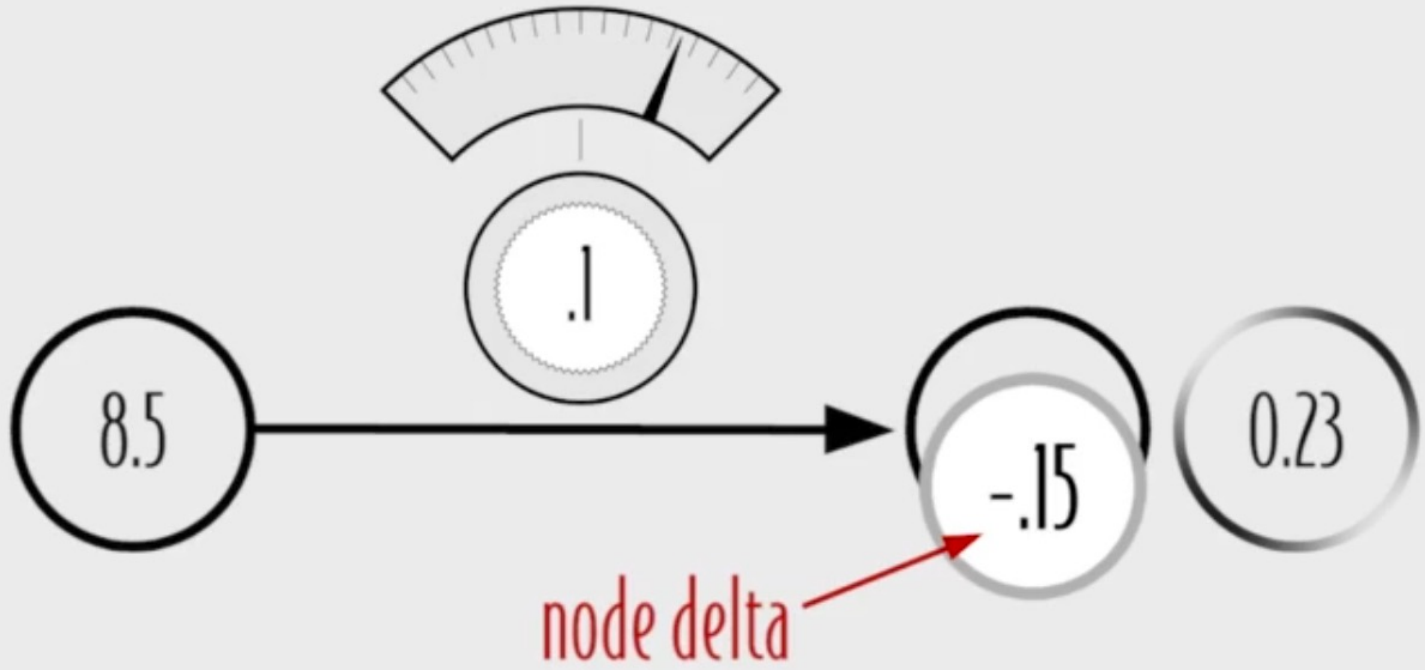
In [ ]:

```
weight = 0.1
input = 8.5  # number of toes
goal_prediction = 1 # represents a win
alpha = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input, weight)
error = (prediction - goal_prediction) ** 2
```



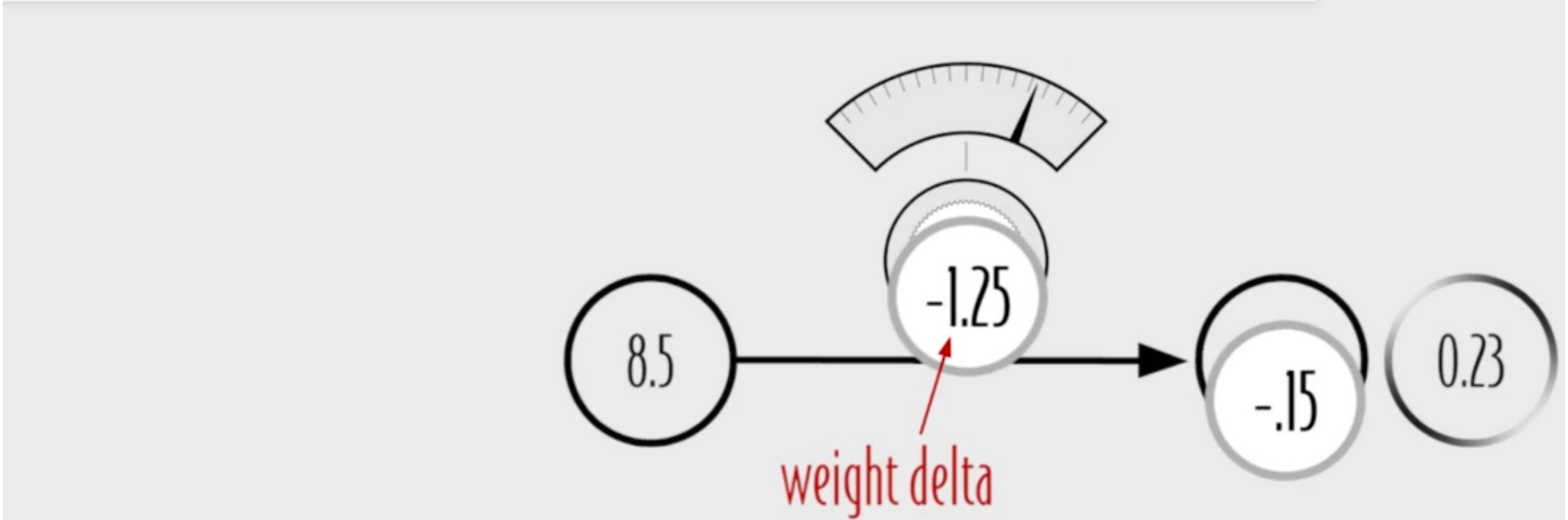
In [ ]:

```
weight = 0.1
input = 8.5  # number of toes
goal_prediction = 1 # represents a win
alpha = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input, weight)
error = (prediction - goal_prediction) ** 2

delta = prediction - goal_prediction
```



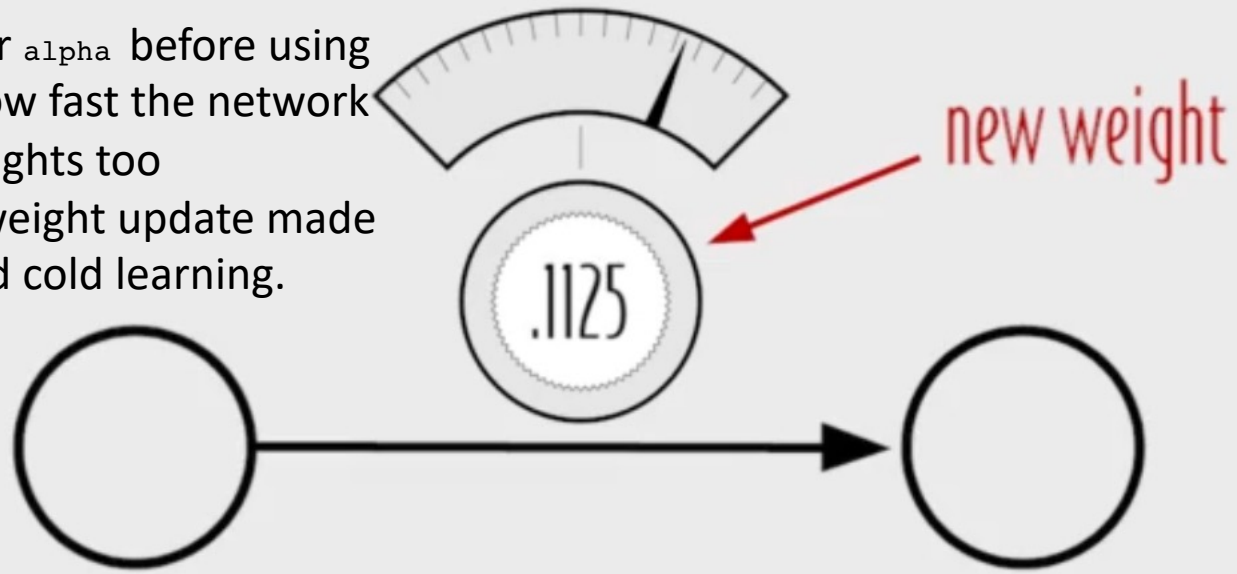
In [ ]:

```
weight = 0.1
input = 8.5  # number of toes
goal_prediction = 1 # represents a win
alpha = 0.01

def neural_network(input, weight):
    prediction = input * weight
    return prediction

prediction = neural_network(input, weight)
error = (prediction - goal_prediction) ** 2

delta = prediction - goal_prediction
weight_delta = input * delta
```



```
In [ ]: weight = 0.1
        input = 8.5  # number of toes
        goal_prediction = 1 # represents a win
        alpha = 0.01

        def neural_network(input, weight):
            prediction = input * weight
            return prediction

        prediction = neural_network(input, weight)
        error = (prediction - goal_prediction) ** 2

        delta = prediction - goal_prediction
        weight_delta = input * delta
        weight -+ weight_delta * alpha
```

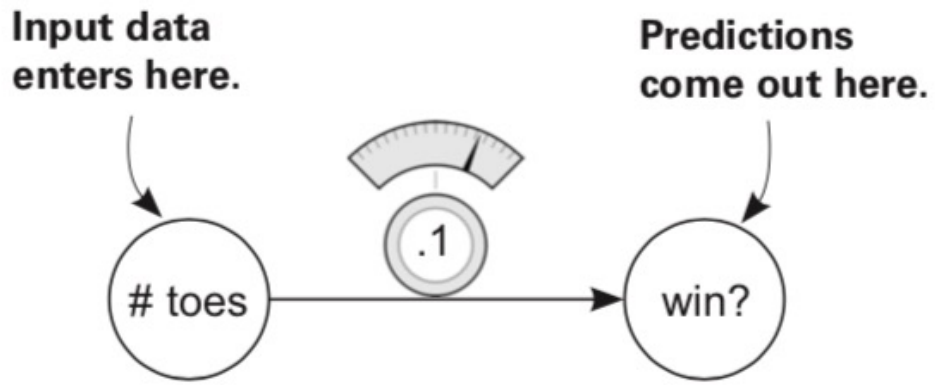
You multiply `weight_delta` by a small number `alpha` before using it to update `weight`. This lets us to control how fast the network learns. If it learns too fast, it can update weights too aggressively and overshoot. Note that the weight update made the same change (small increase) as hot and cold learning.



One iteration of gradient descent

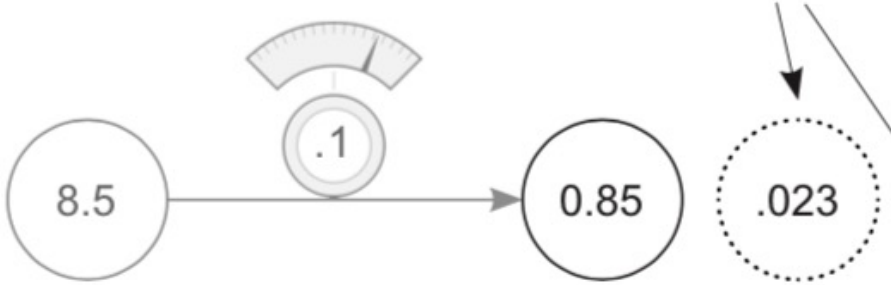
This performs a weight update on a single training example (input->true) pair.

❶ An empty network




```
weight = 0.1

alpha = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

❷ PREDICT: Making a prediction and evaluating error



Error

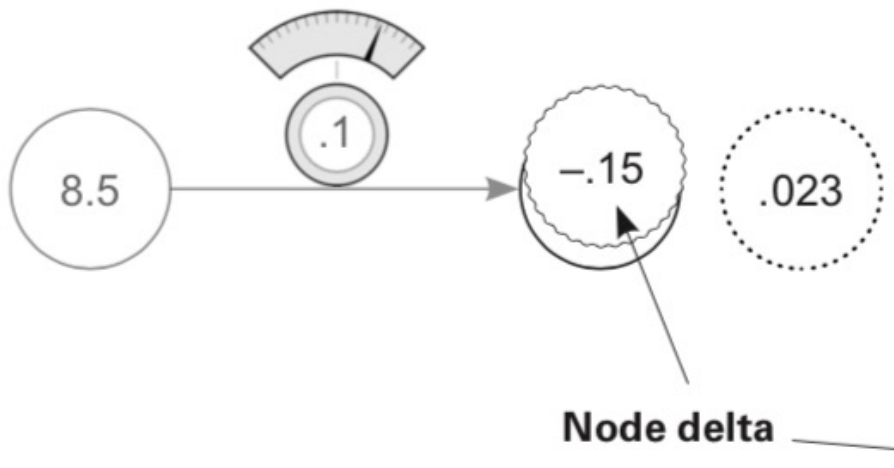
```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2
```

The error is a way to measure how much you missed. There are multiple ways to calculate error, as you'll learn later. This one is *mean squared error*.

Raw error

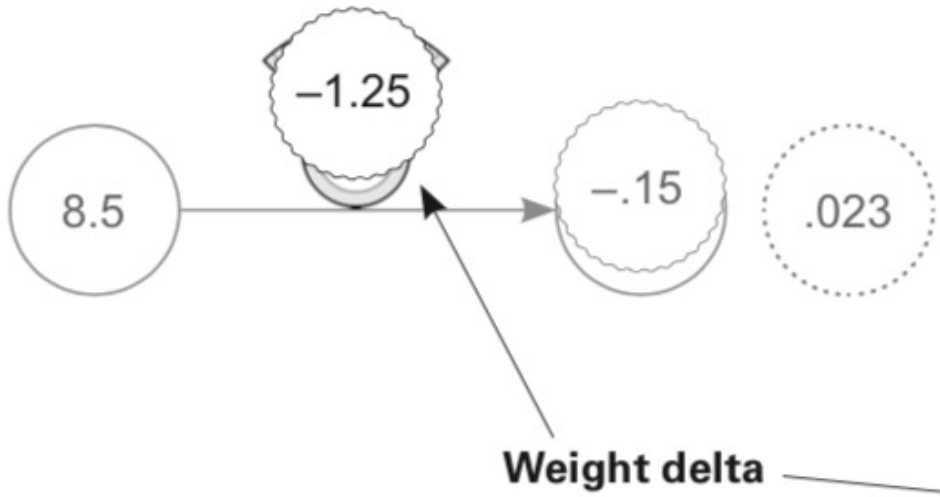
Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

❸ COMPARE: Calculating the node delta and putting it on the output node



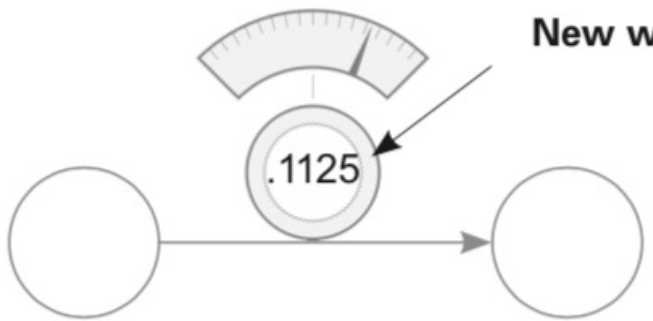

```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred
```

❹ LEARN: Calculating the weight delta and putting it on the weight




```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta
```

❺ LEARN: Updating the weight




```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta

alpha = 0.01

weight -= weight_delta * alpha
```

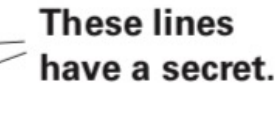
Fixed before training →

Learning is just reducing error

You can modify weight to reduce error.

Putting together the code from the previous pages, we now have the following:

```
weight, goal_pred, input = (0.0, 0.8, 0.5)


for iteration in range(4):

    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
```



The golden method for learning

This approach adjusts each `weight` in the correct direction and by the correct amount so that `error` reduces to 0.

```
weight = 0
input = .5
goal_prediction = .8
alpha = 1

def neural_network(input, weight):
    prediction = input * weight
    return prediction


for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    prediction = neural_network(input, weight)
    error = (prediction - goal_prediction) ** 2
    delta = prediction - goal_prediction
    weight_delta = input * delta
    weight -= weight_delta * alpha
    print "Error:" + str(error) + " Prediction:" + str(prediction)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

```
weight = 0
input = .5
goal_prediction = .8
alpha = 1

for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    prediction = input * weight
    error = (prediction - goal_prediction) ** 2
    delta = prediction - goal_prediction
    weight_delta = input * delta
    weight -= weight_delta * alpha
    print "Error:" + str(error) + " Prediction:" + str(prediction)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

```
weight = 0
input = .5
goal_prediction = .8
alpha = 1

for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    prediction = input * weight
    error = ((input * weight) - goal_prediction) ** 2
    delta = prediction - goal_prediction
    weight_delta = input * delta
    weight -= weight_delta * alpha
    print "Error:" + str(error) + " Prediction:" + str(prediction)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

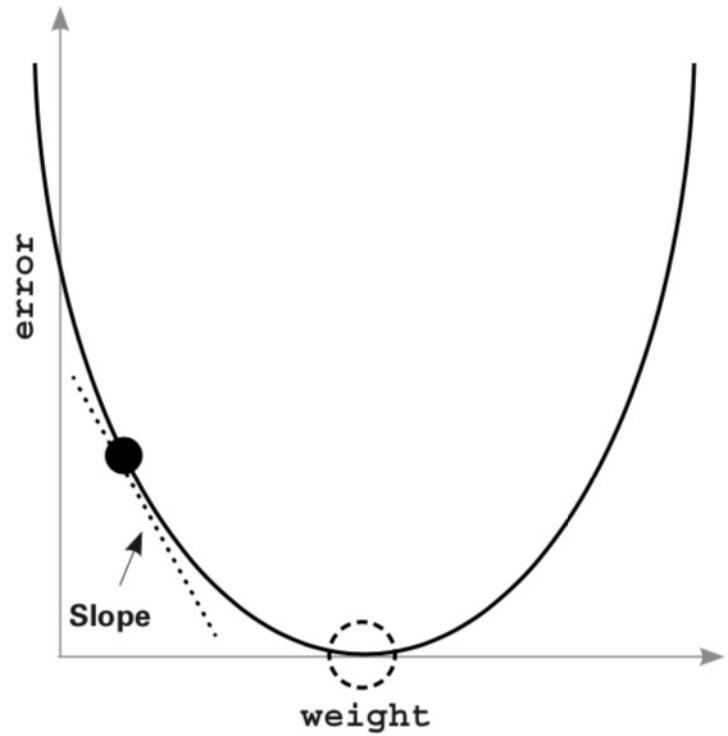
```
weight = 0
input = .5
goal_prediction = .8
alpha = 1

for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    prediction = input * weight
    error = ((0.5 * weight) - 0.8) ** 2
    delta = prediction - goal_prediction
    weight_delta = input * delta
    weight -= weight_delta * alpha
    print "Error:" + str(error) + " Prediction:" + str(prediction)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

The secret

For any `input` and `goal_pred`, an *exact relationship* is defined between `error` and `weight`, found by combining the `prediction` and `error` formulas. In this case:

```
error = ((0.5 * weight) - 0.8) ** 2
```

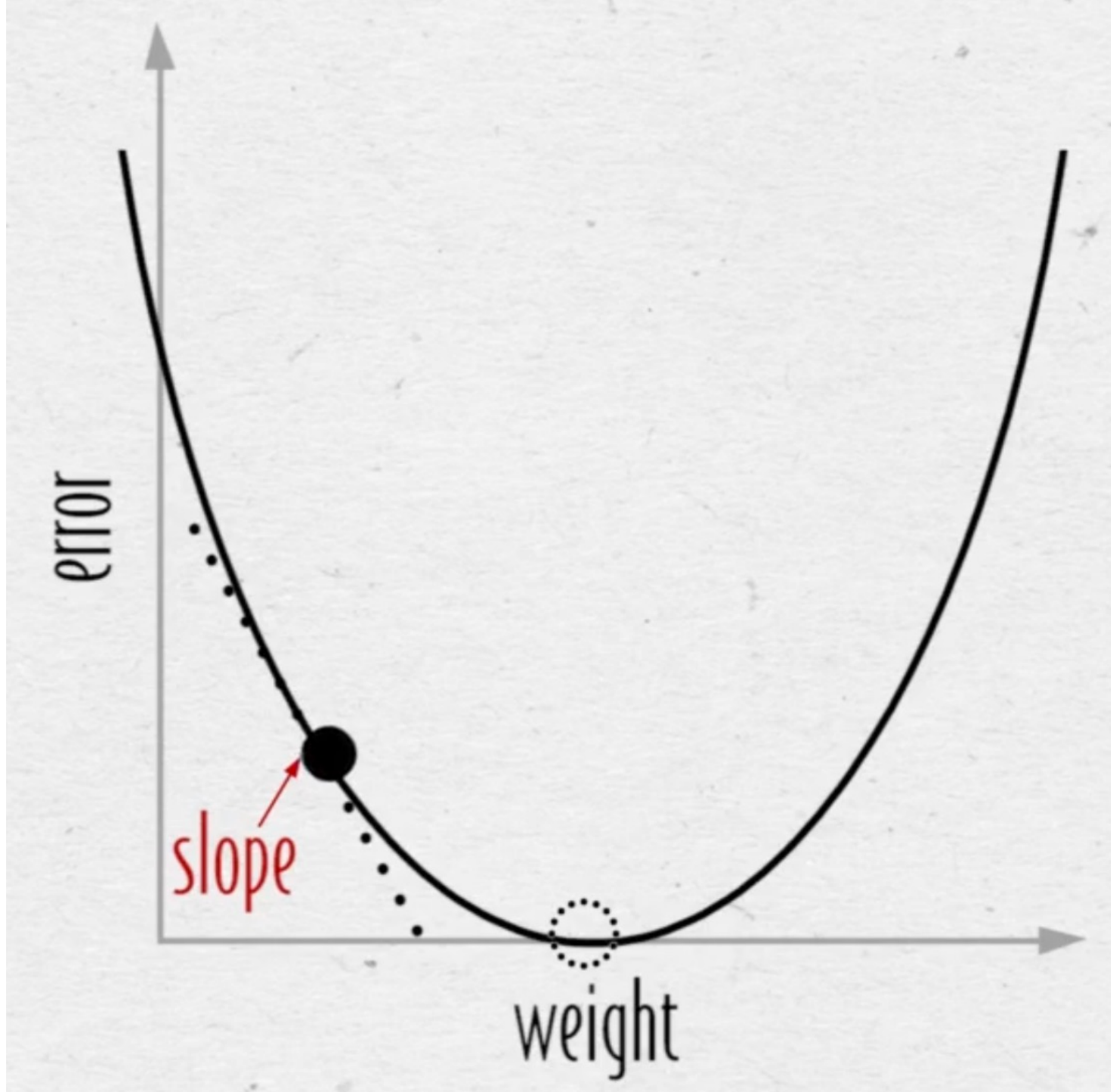
Let's say you increased `weight` by 0.5. If there's an exact relationship between `error` and `weight`, you should be able to calculate how much this also moves `error`. What if you wanted to move `error` in a specific direction? Could it be done?

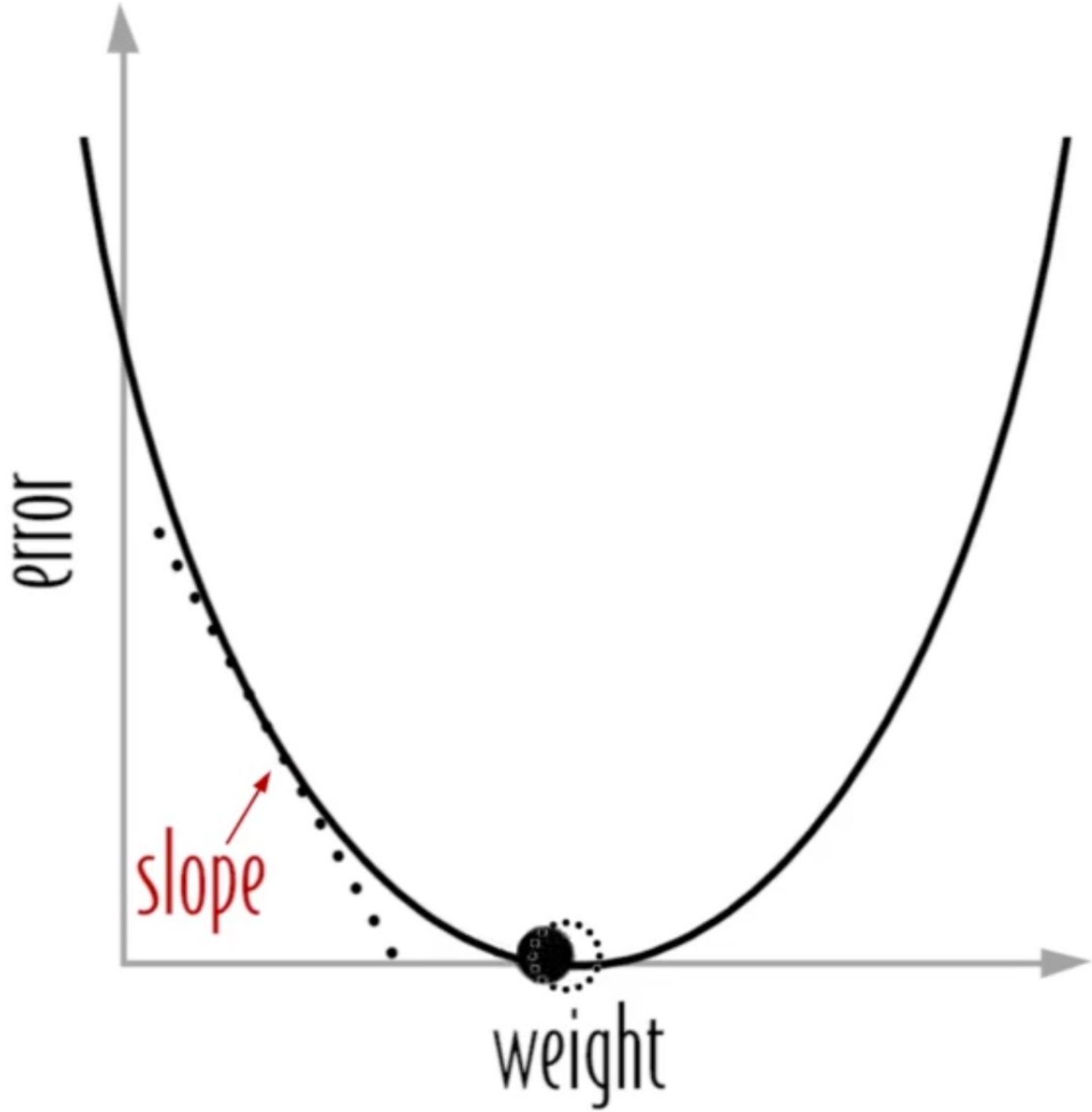


This graph represents every value of error for every weight according to the relationship in the previous formula. Notice it makes a nice bowl shape. The black dot is at the point of *both* the current `weight` and `error`. The dotted circle is where you want to be (`error` == 0).

Key takeaway

The slope points to the *bottom* of the bowl (lowest `error`) no matter where you are in the bowl. You can use this slope to help the neural network reduce the error.


error
slope
weight


error
slope
weight

Let's watch several steps of learning

Will we eventually find the bottom of the bowl?

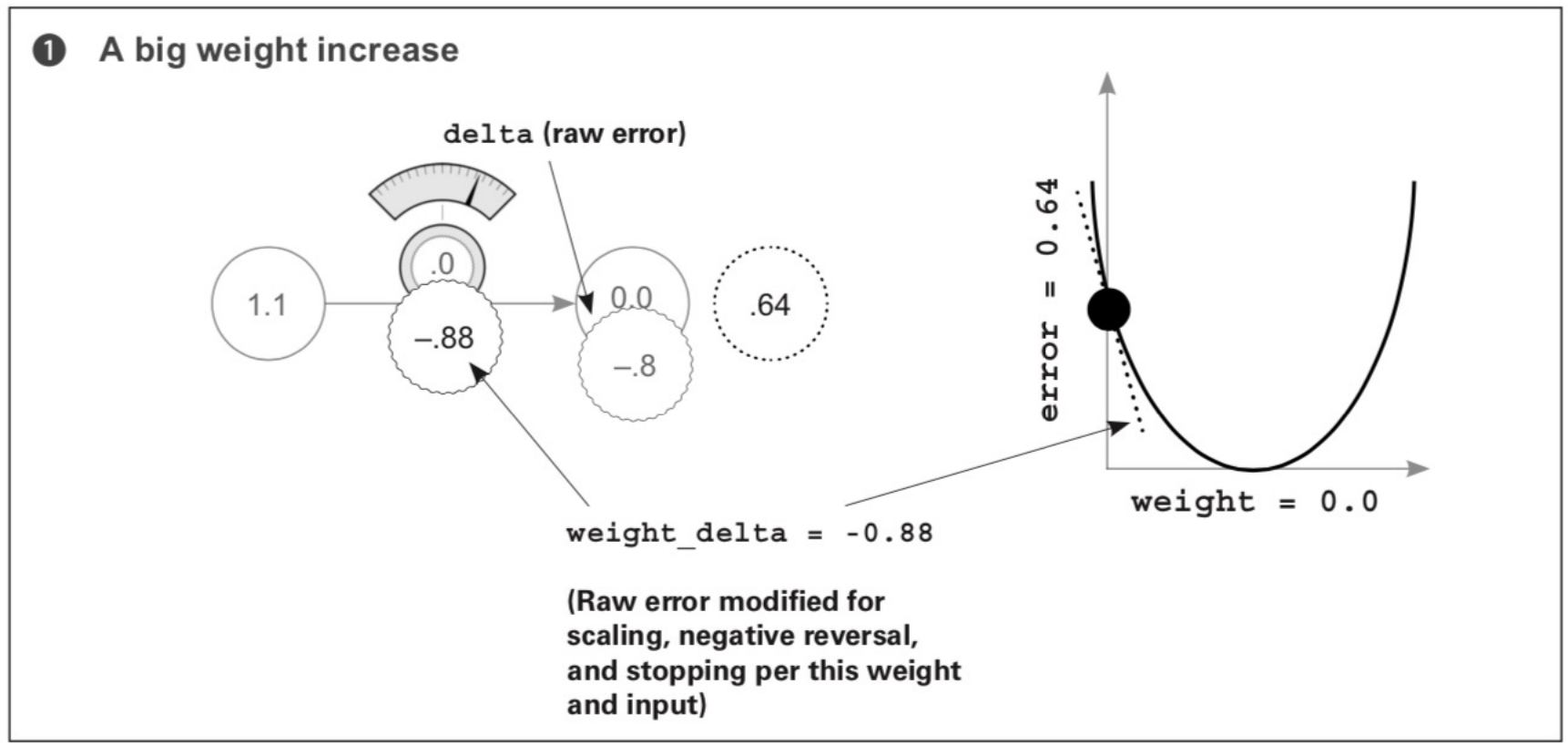
```
weight, goal_pred, input = (0.0, 0.8, 1.1)

for iteration in range(4):
    print("-----\nWeight:" + str(weight))
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
    print("Delta:" + str(delta) + " Weight Delta:" + str(weight_delta))
```

```
weight = 0
input = 1.1
goal_prediction = .8
alpha = 1

def neural_network(input, weight):
    prediction = input * weight
    return prediction


for iteration in range(4):
    print "-----\nWeight:" + str(weight)
    prediction = neural_network(input, weight)
    error = (prediction - goal_prediction) ** 2
    delta = prediction - goal_prediction
    weight_delta = input * delta
    weight -= weight_delta * alpha
    print "Error:" + str(error) + " Prediction:" + str(prediction)
    print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta)
```

```
-----
Weight:0
Error:0.64 Prediction:0.0
Delta:-0.8 Weight Delta:-0.88
-----
Weight:0.88
Error:0.028224 Prediction:0.968
Delta:0.168 Weight Delta:0.1848
-----
Weight:0.6952
Error:0.0012446784 Prediction:0.76472
Delta:-0.03528 Weight Delta:-0.038808
-----
Weight:0.734008
Error:5.489031744e-05 Prediction:0.8074088
Delta:0.0074088 Weight Delta:0.00814968
```

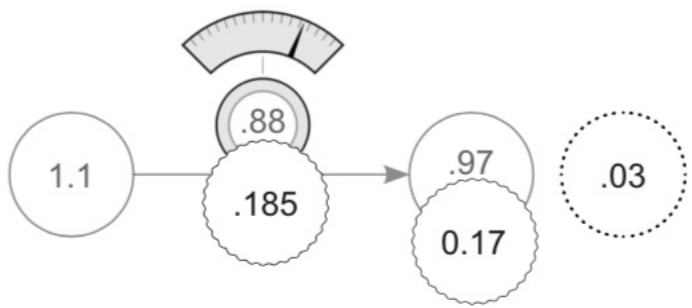
```
weight = 0
input = 1.1
goal_prediction = .8
alpha = 1
```

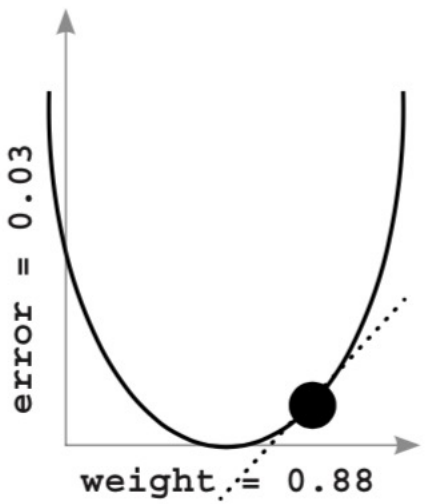
```
-----
Weight:0
Error:0.64 Prediction:0.0
Delta:-0.8 Weight Delta:-0.88
```



```
-----
Weight:0.88
Error:0.028224 Prediction:0.968
Delta:0.168 Weight Delta:0.1848
```

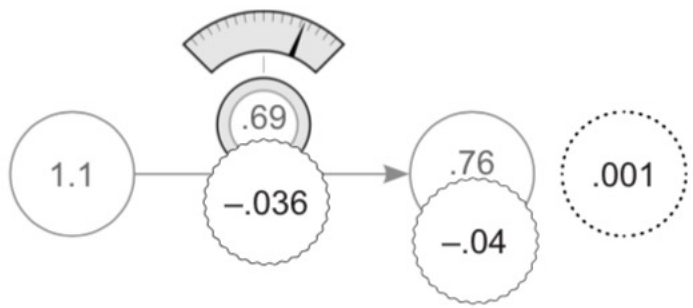
❷ Overshot a bit; let's go back the other way.

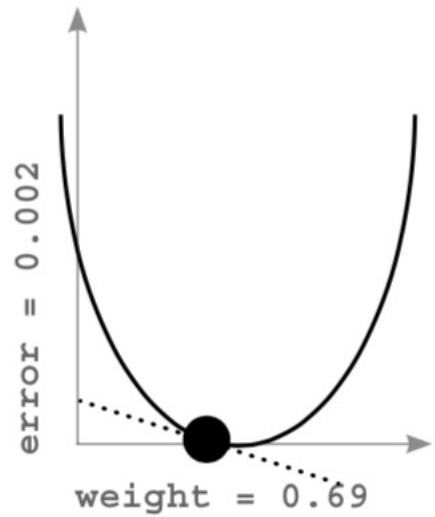




```
-----
Weight:0.6952
Error:0.0012446784 Prediction:0.76472
Delta:-0.03528 Weight Delta:-0.038808
```

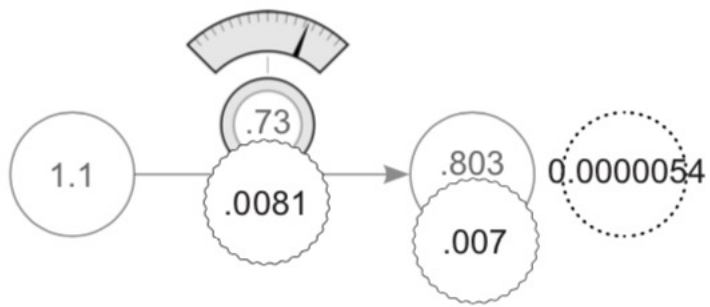
❸ **Overshot again! Let's go back, but only a little.**

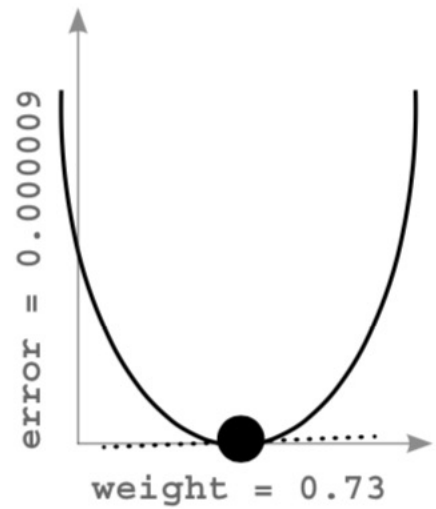




```
-----
Weight:0.734008
Error:5.489031744e-05 Prediction:0.8074088
Delta:0.0074088 Weight Delta:0.00814968
```

❹ **OK, we're pretty much there.**





Why does this work? What is weight_delta, really?

Let's back up and talk about functions. What is a function? How do you understand one?

Consider this function:

```
def my_function(x):
    return x * 2
```

A function takes some numbers as input and gives you another number as output. As you can imagine, this means the function defines some sort of relationship between the input number(s) and the output number(s). Perhaps you can also see why the ability to learn a function is so powerful: it lets you take some numbers (say, image pixels) and convert them into other numbers (say, the probability that the image contains a cat).

Every function has what you might call *moving parts*: pieces you can tweak or change to make the output the function generates different. Consider `my_function` in the previous example. Ask yourself, "What's controlling the relationship between the input and the output of this function?" The answer is, the 2. Ask the same question about the following function:

```
error = ((input * weight) - goal_pred) ** 2
```

To sum up: you modify specific parts of an error function until the `error` value goes to 0. This error function is calculated using a combination of variables, some of which you can change (weights) and some of which you can't (input data, output data, and the error logic):

```
weight = 0.5
goal_pred = 0.8
input = 0.5

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print("Error:" + str(error) + " Prediction:" + str(pred))
```

Key takeaway

You can modify *anything* in the `pred` calculation except `input`.

We'll spend the rest of this book (and many deep learning researchers will spend the rest of their lives) trying everything you can imagine on that `pred` calculation so that it can make good predictions. Learning is all about automatically changing the prediction function so that it makes good predictions—aka, so that the subsequent `error` goes down to 0.

Now that you know what you're allowed to change, how do you go about doing the changing? That's the good stuff. That's the machine learning, right? In the next section, we're going to talk about exactly that.

Tunnel vision on one concept

Concept: Learning is adjusting the weight to reduce the error to 0.

How to do this is all about understanding the *relationship* between `weight` and `error`. If you understand this relationship, you can know how to adjust `weight` to reduce `error`.

To understand the relationship between two variables is to understand *how changing one variable changes the other*.

In this case, what you're really after is the ***sensitivity*** between these two variables. Sensitivity is another name for direction and amount. You want to know how sensitive `error` is to `weight`. You want to know the direction and the amount that `error` changes when you change `weight`. This is the goal.

```
if(down_error < up_error):
    weight = weight - step_amount

if(down_error > up_error):
    weight = weight + step_amount
```

This is the relationship between `error` and `weight`. This relationship is exact. It's computable. It's universal. It is and will always be.

```
error = ((input * weight) - goal_pred) ** 2
```

Derivatives

A box with rods poking out of it

Notice that you always have the derivative *between two variables*. You're always looking to know how one variable moves when you change another one. If the derivative is positive, then when you change one variable, the other will move in the *same* direction. If the derivative is *negative*, then when you change one variable, the other will move in the *opposite* direction.

Derivatives

One way is all about understanding how one variable in a function changes when you move another variable. The other way says that a derivative is the slope at a point on a line or curve. As it turns out, if you take a function and plot it (draw it), the slope of the line you plot is the *same thing* as "how much one variable changes when you change the other."

```
error = ((input * weight) - goal_pred) ** 2
```

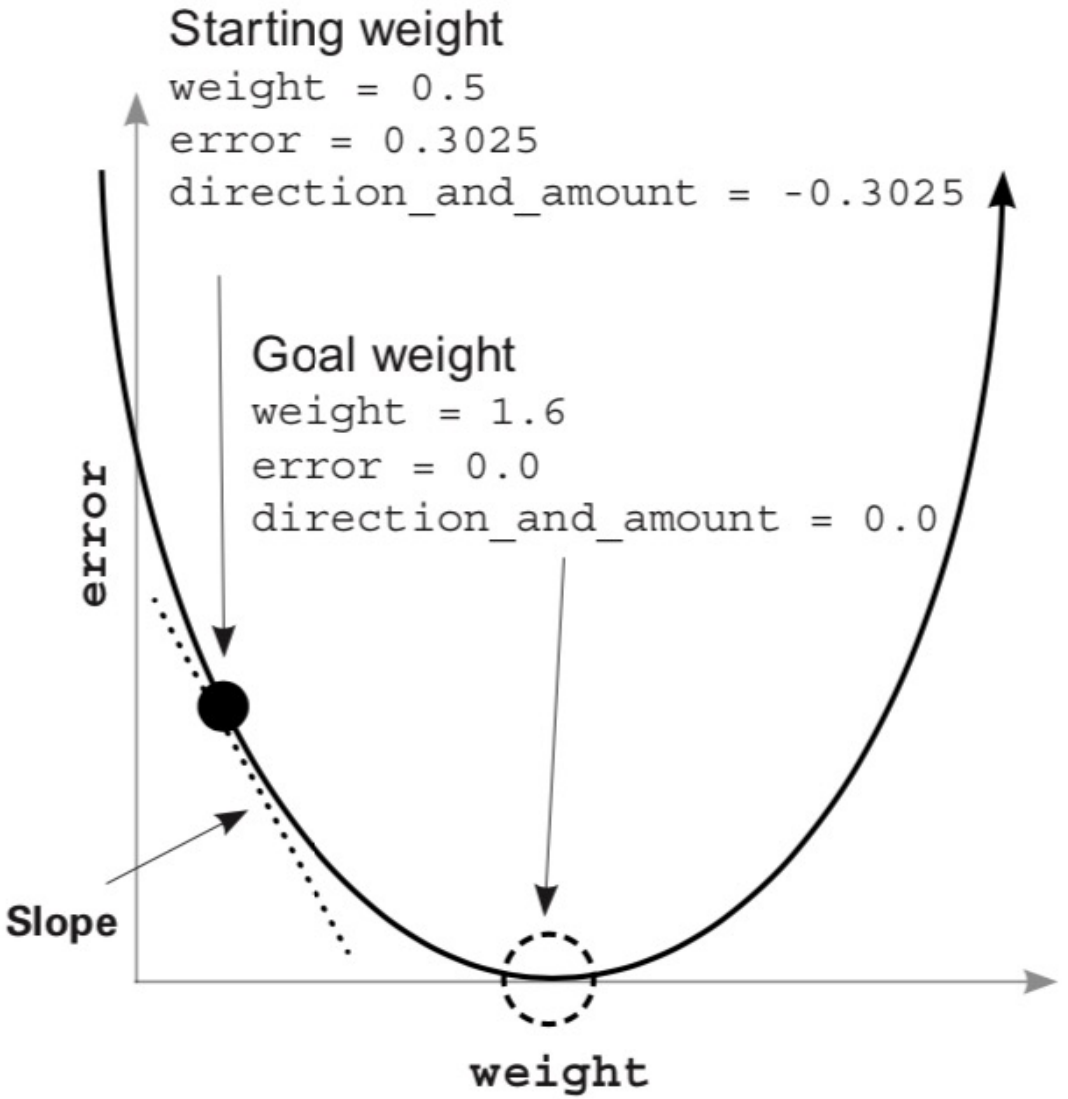
Remember, `goal_pred` and `input` are fixed, so you can rewrite this function:

```
error = ((0.5 * weight) - 0.8) ** 2
```

Because there are only two variables left that change (all the rest of them are fixed), you can take every `weight` and compute the `error` that goes with it. Let's plot them.

Derivatives

The plot looks like a big U-shaped curve. Notice that there's also a point in the middle where `error` == 0. Also notice that to the right of that point, the slope of the line is positive, and to the left of that point, the slope of the line is negative. Perhaps even more interesting, the farther away from the *goal weight* you move, the steeper the slope gets.



These are useful properties. The slope's sign gives you direction, and the slope's steepness gives you amount. You can use both of these to help find the goal `weight`.

How do we use our derivative to find the error minimum?

How to use a derivative to learn

weight_delta is your derivative.

Error vs Derivatives

How much of the error change occurred by weight change.
Later, how to use derivates to reduce errors.