

# Lesson 4 Introduction to neural learning: gradient descent





# Introduction to neural learning: gradient descent

---

In this lesson you will learn

- Do neural networks make accurate predictions?
- Why measure error?
- Hot and cold learning
- Calculating both direction and amount from error
- Gradient descent
- Learning is just reducing error
- Derivatives and how to use them to learn
- Divergence and alpha



# Predict, compare, and learn

---

- In lesson 3, you learned about the paradigm “predict, compare, learn,” and we dove deep into the first step: predict.
- In the process, you learned a myriad of things, including the major parts of neural networks (nodes and weights), how datasets fit into networks (matching the number of datapoints coming in at one time), and how to use a neural network to make a prediction.



# 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.
- This may seem like a simple concept, but you’ll find that coming up with a good way to measure error is one of the most important and complicated subjects of deep learning.
- There are many properties of measuring error that you’ve likely been doing your whole life without realizing it. Perhaps you (or someone you know) amplify bigger errors while ignoring very small ones.



# Compare

---

- As a heads-up, in this lesson we evaluate only one simple way of measuring error: mean squared error.
- 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.
- The output of the compare logic is a “hot or cold” type signal. Given some prediction, you'll calculate an error measure that says either “a lot” or “a little.”



# Learn

---

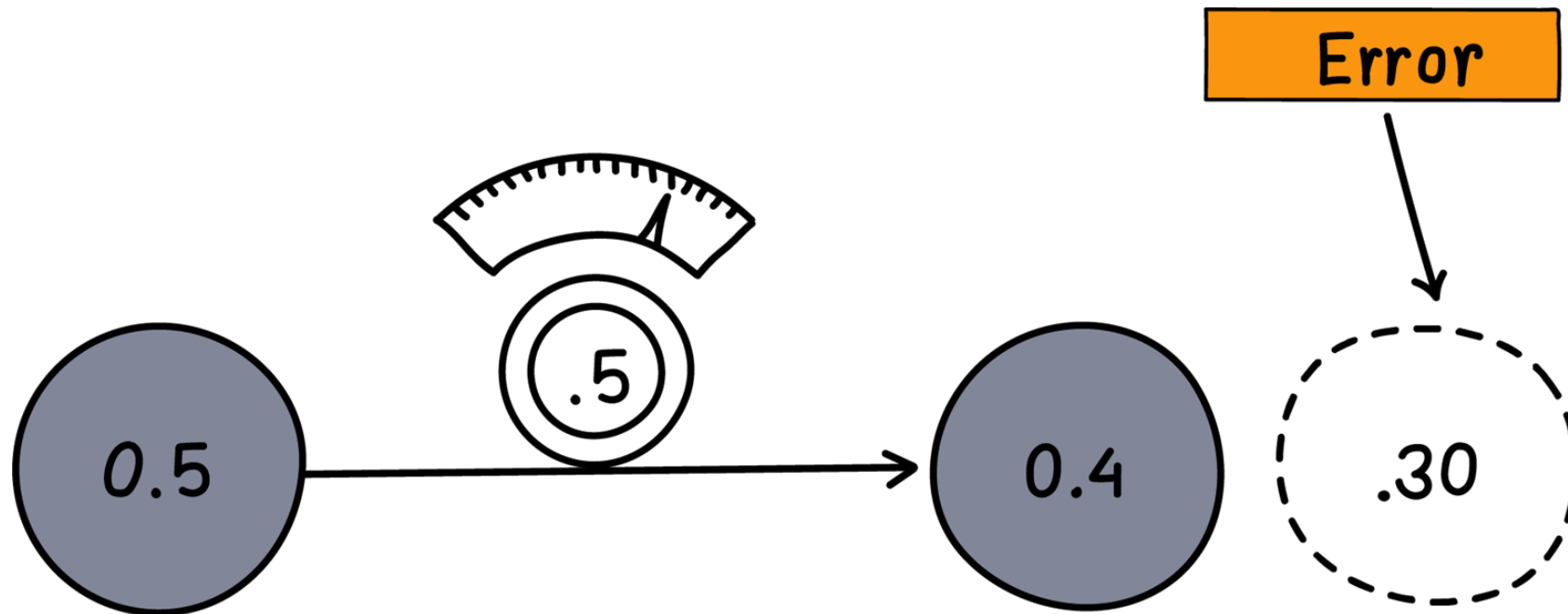
**Learning tells each weight how it can change to reduce the error.**

- Learning is all about error attribution, or the art of figuring out how each weight played its part in creating error.
- It's the blame game of deep learning.
- In this lesson, we'll spend many pages looking at the most popular version of the deep learning blame game: gradient descent.

# 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:





# 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 lesson), 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.





# Why measure error?

---

**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 ( $\text{pred} - \text{goal\_pred}$ ) so that bigger errors become very big and smaller errors quickly become irrelevant.



# Why measure 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.

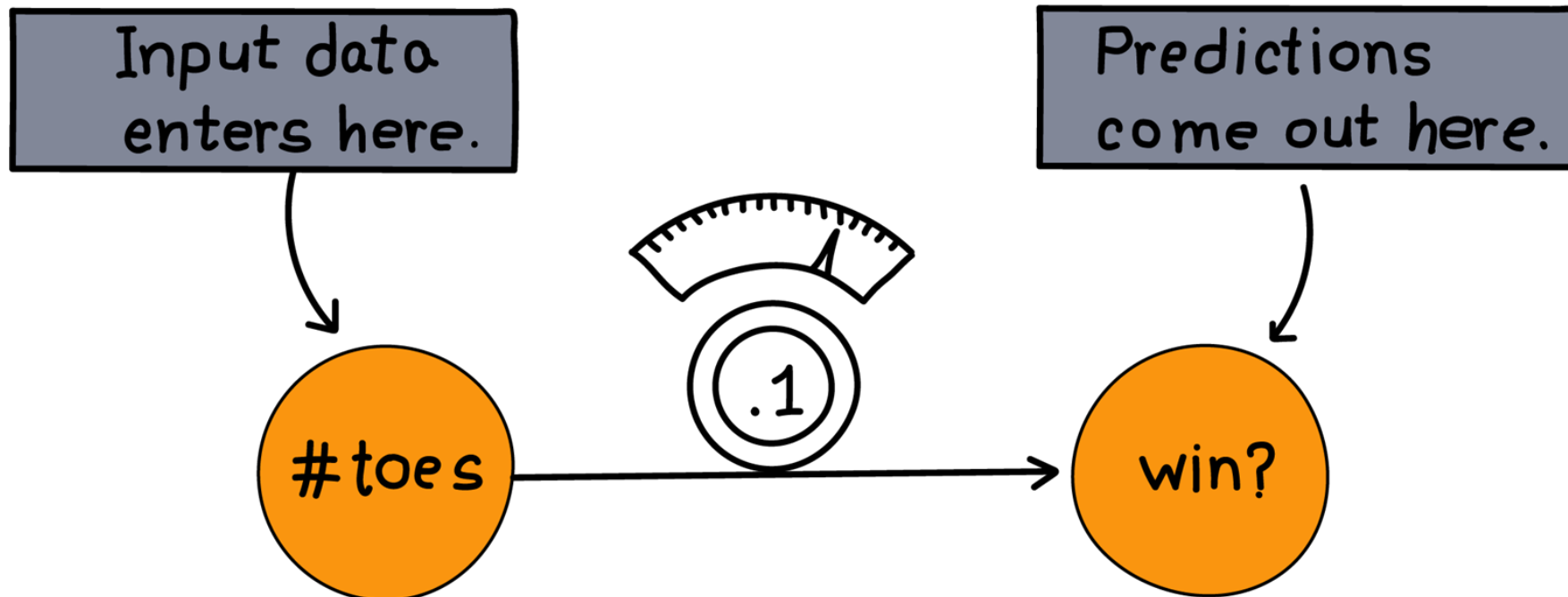
# What's the simplest form of neural learning?

## Learning using the hot and cold method.

- At the end of the day, learning is really about one thing: adjusting `dial_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 dial 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 `dial_weight`.

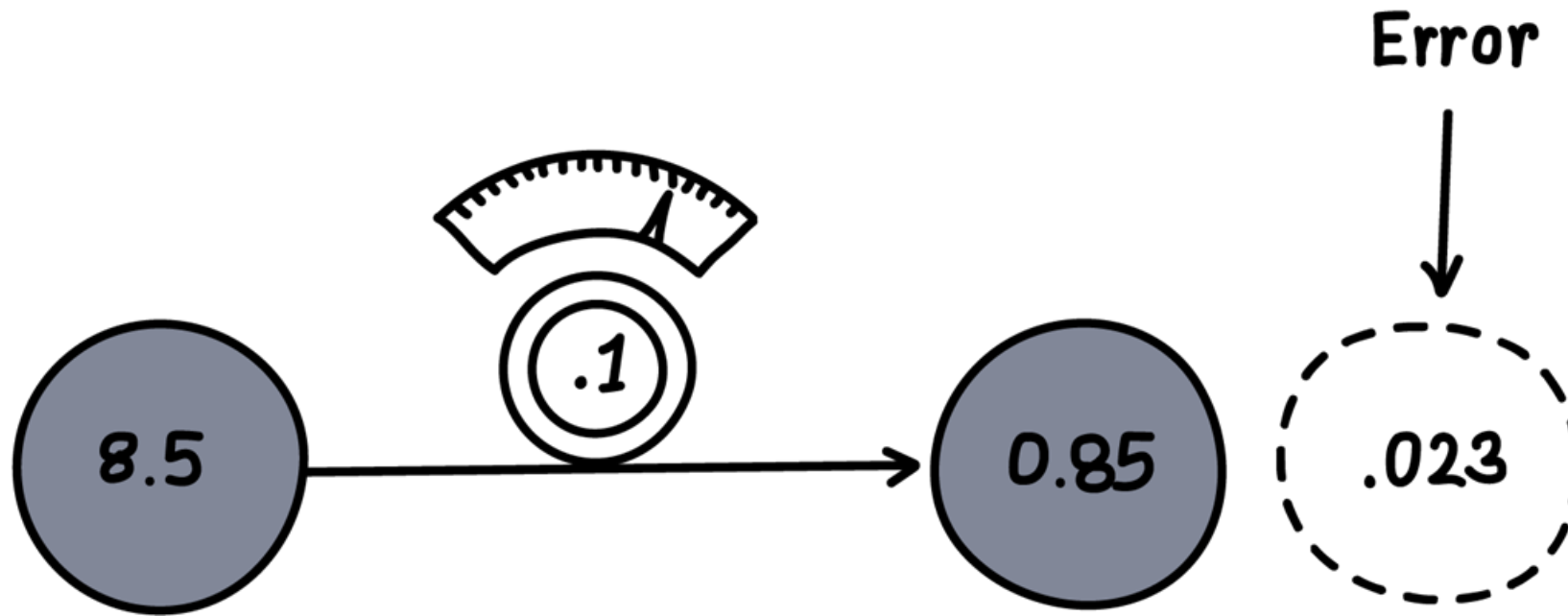
# What's the simplest form of neural learning?

① An empty network



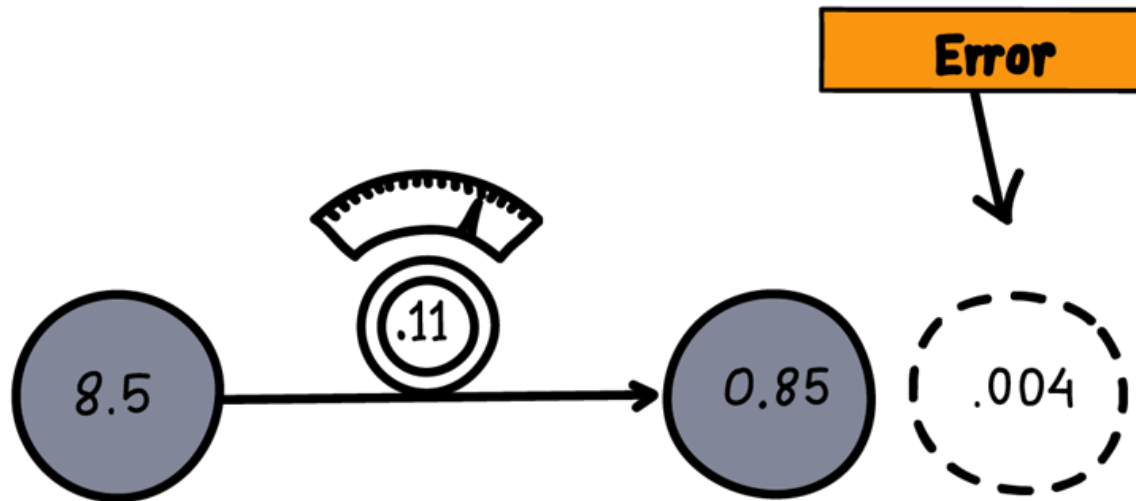
# What's the simplest form of neural learning?

② PREDICT: Making a prediction and evaluating error



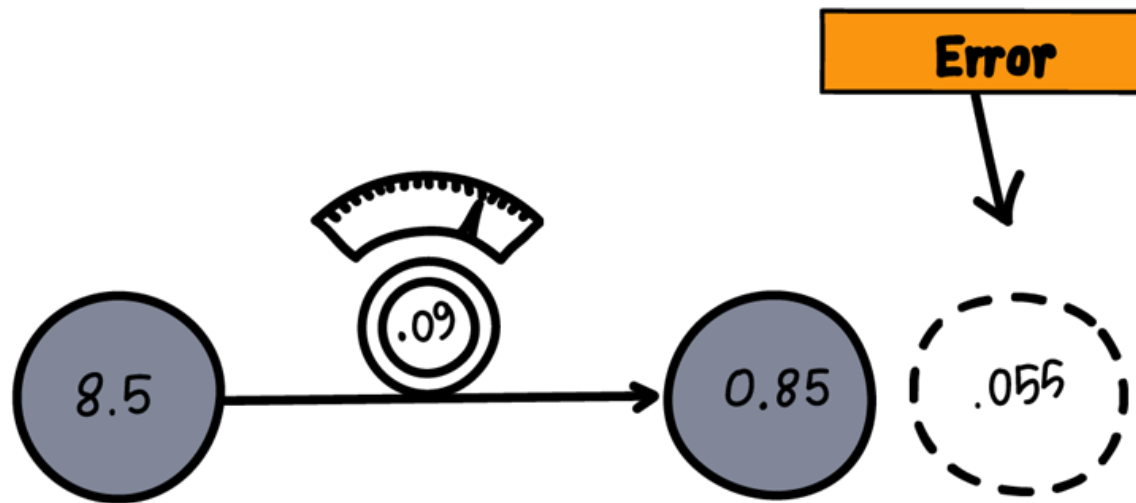
# What's the simplest form of neural learning?

③ COMPARE: Making a prediction with a higher weight and evaluating error



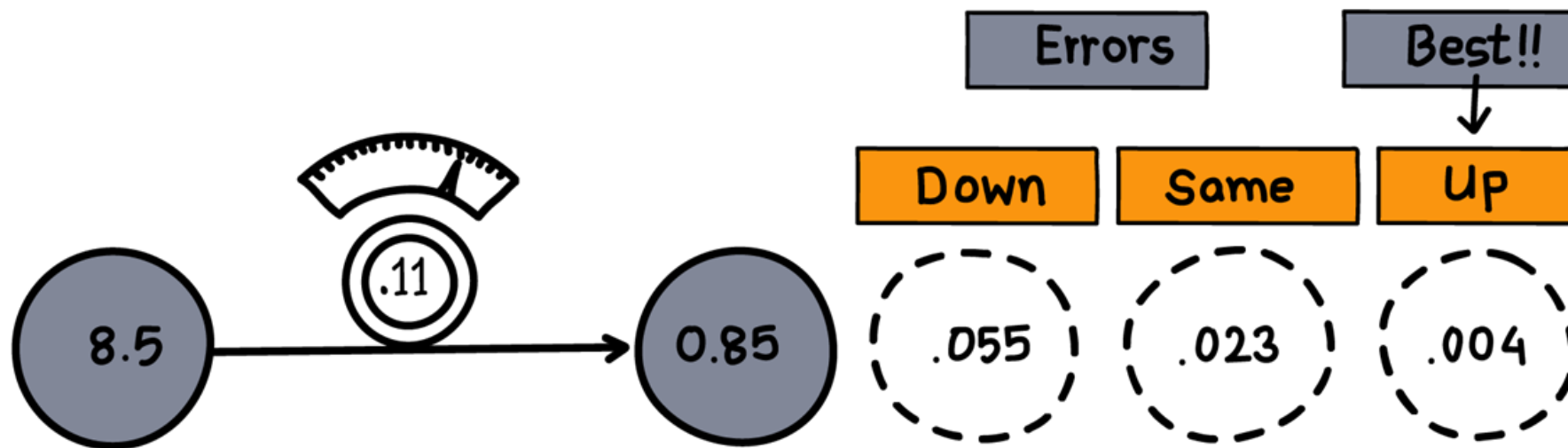
# What's the simplest form of neural learning?

③ COMPARE: Making a prediction with a lower weight and evaluating error



# What's the simplest form of neural learning?

⑤ COMPARE + LEARN: Comparing the errors and setting the new weight





# What's the simplest form of neural learning?

- 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.

# Hot and cold learning

This is perhaps the simplest form of learning.

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

How much to move  
the weights each  
iteration

```
step_amount = 0.001
```

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

```
for iteration in range(1101):
```

# Hot and cold learning

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

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

up_prediction = input * (weight + step_amount) ← Try up!
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = input * (weight - step_amount) ← Try down!
down_error = (goal_prediction - down_prediction) ** 2

if(down_error < up_error):
    weight = weight - step_amount ← If down is better,
                                   go down!

if(down_error > up_error):
    weight = weight + step_amount ← If up is better,
                                   go up!
```

# Hot and cold learning

- When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25  
Error:0.30195025 Prediction:0.2505  
.....  
Error:2.500000000033e-07 Prediction:0.7995  
Error:1.07995057925e-27 Prediction:0.8
```

← The last step correctly predicts 0.8!



# Characteristics of hot and cold learning

---

It's simple.

- Hot and cold learning is simple.
- After making a prediction, you predict two more times, once with a slightly higher weight and again with a slightly lower weight.
- You then move weight depending on which direction gave a smaller error. Repeating this enough times eventually reduces error to 0.



# Characteristics of hot and cold learning

---

## **Problem 1: It's inefficient.**

- You have to predict multiple times to make a single dial\_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  $n \times \text{step\_amount}$  away, the network will eventually overshoot by some number less than step\_amount

# Characteristics of hot and cold learning

- 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...
```

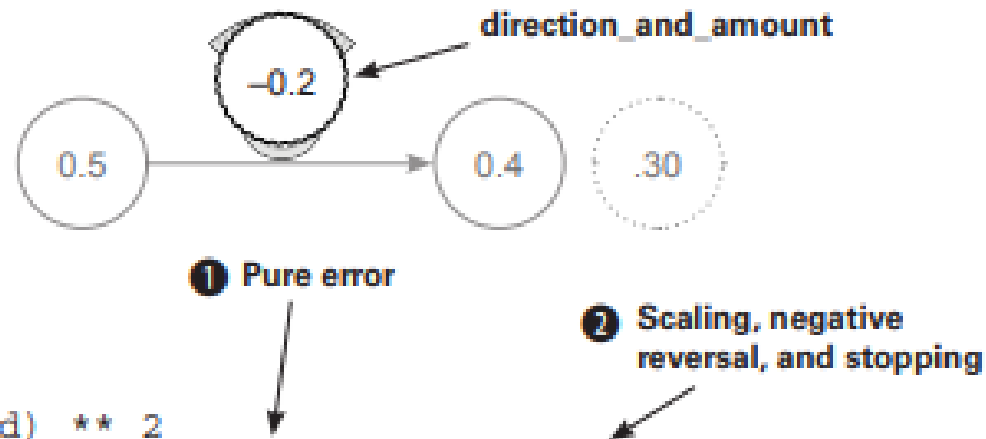
# Calculating both direction and amount from error

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))
```





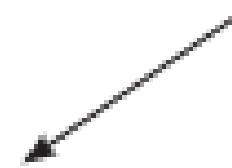
# Calculating both direction and amount from error

- When you run the previous code, you should see the following output:

```
Error:0.3025 Prediction:0.25  
Error:0.17015625 Prediction:0.3875  
Error:0.095712890625 Prediction:0.490625  
...
```

```
Error:1.7092608064e-05 Prediction:0.79586567925  
Error:9.61459203602e-06 Prediction:0.796899259437  
Error:5.40820802026e-06 Prediction:0.797674444578
```

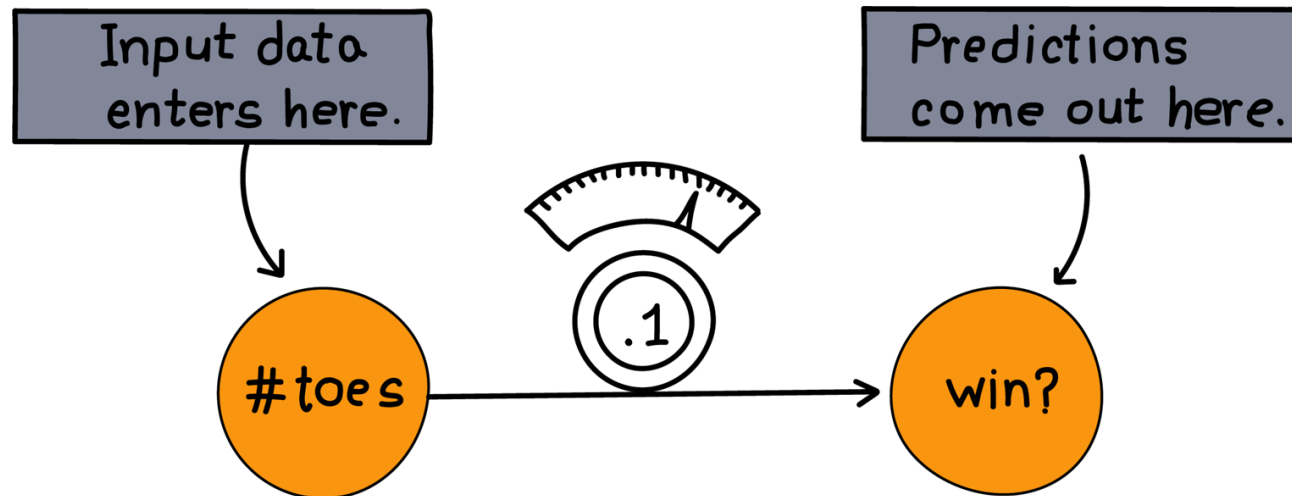
The last steps correctly approach 0.8!



# One iteration of gradient descent

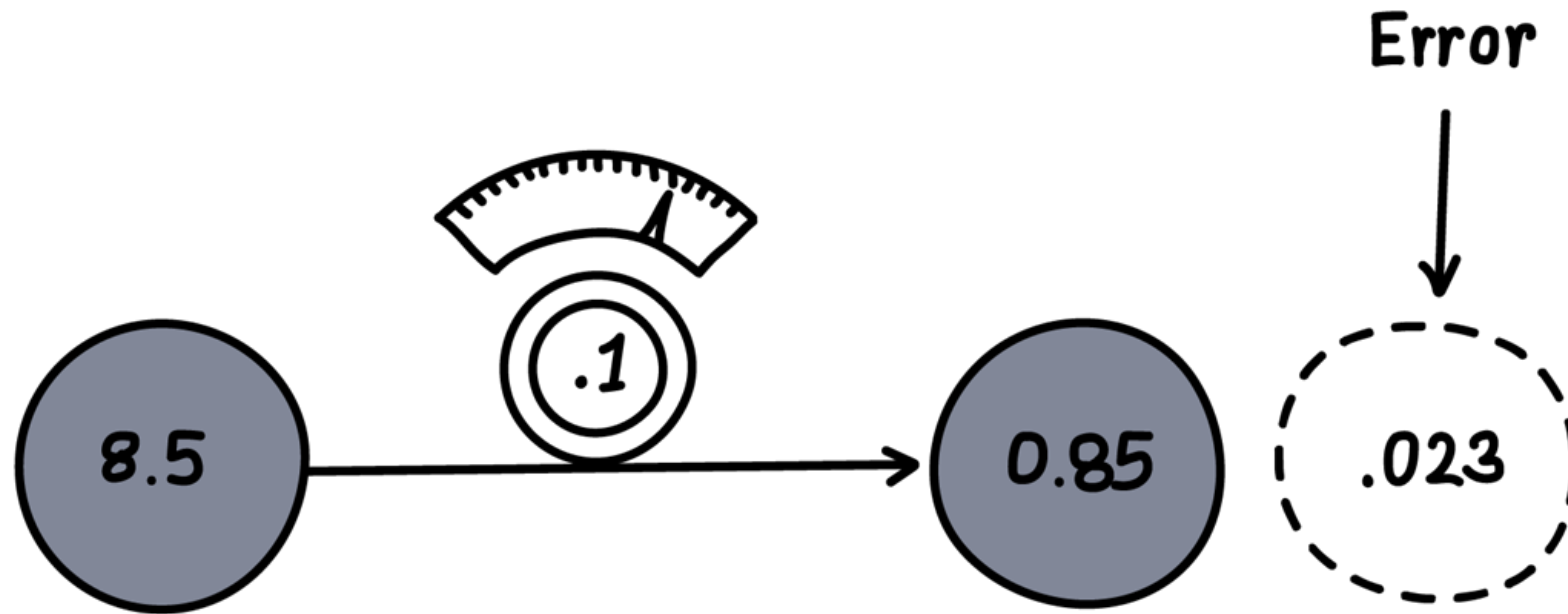
This performs a weight update on a single training example (input->>true) pair.

① An empty network



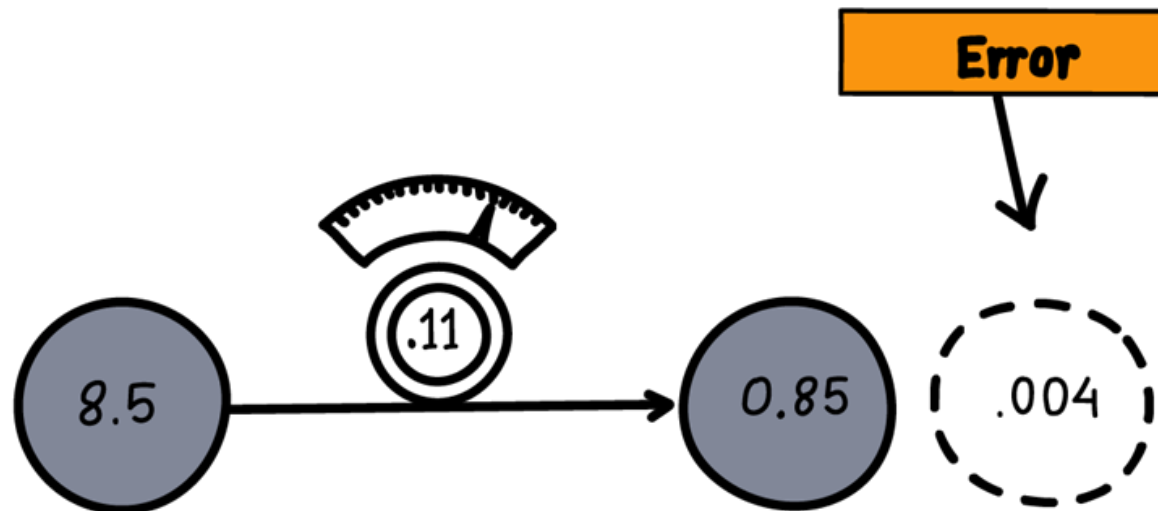
# One iteration of gradient descent

② PREDICT: Making a prediction and evaluating error



# One iteration of gradient descent

③ COMPARE: Making a prediction with a higher weight and evaluating error





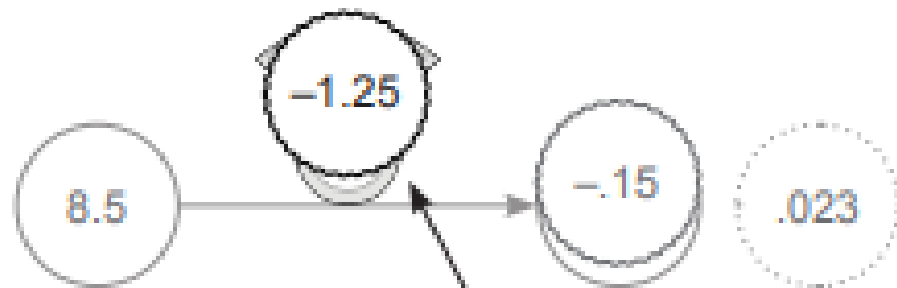
# One iteration of gradient descent

---

- The primary difference between gradient descent and this implementation is the new variable delta.
- It's the raw amount that the node was too high or too low.
- Instead of computing `direction_and_amount` directly, you first calculate how much you want the output node to be different.

# One iteration of gradient descent

## ④ LEARN: Calculating the weight delta and putting it on the weight

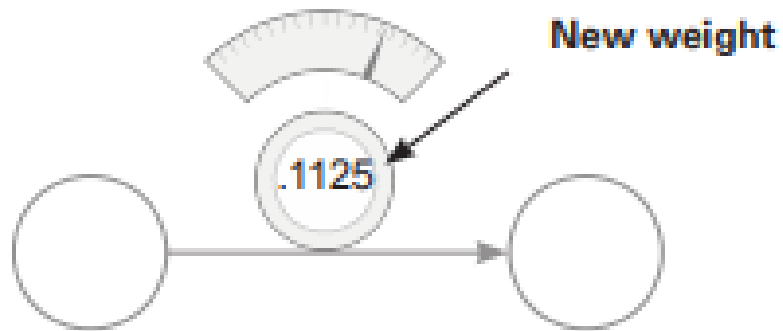


```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

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
```

# One iteration of gradient descent

## ⑤ 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
```

Fixed before training → `alpha = 0.01`

```
weight -= weight_delta * alpha
```

# 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))
```

These lines  
have a secret.





# Learning is just reducing error

---

- The secret lies in the pred and error calculations.
- Notice that you use pred inside the error calculation. Let's replace the pred variable with the code used to generate it:

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



# Learning is just reducing error

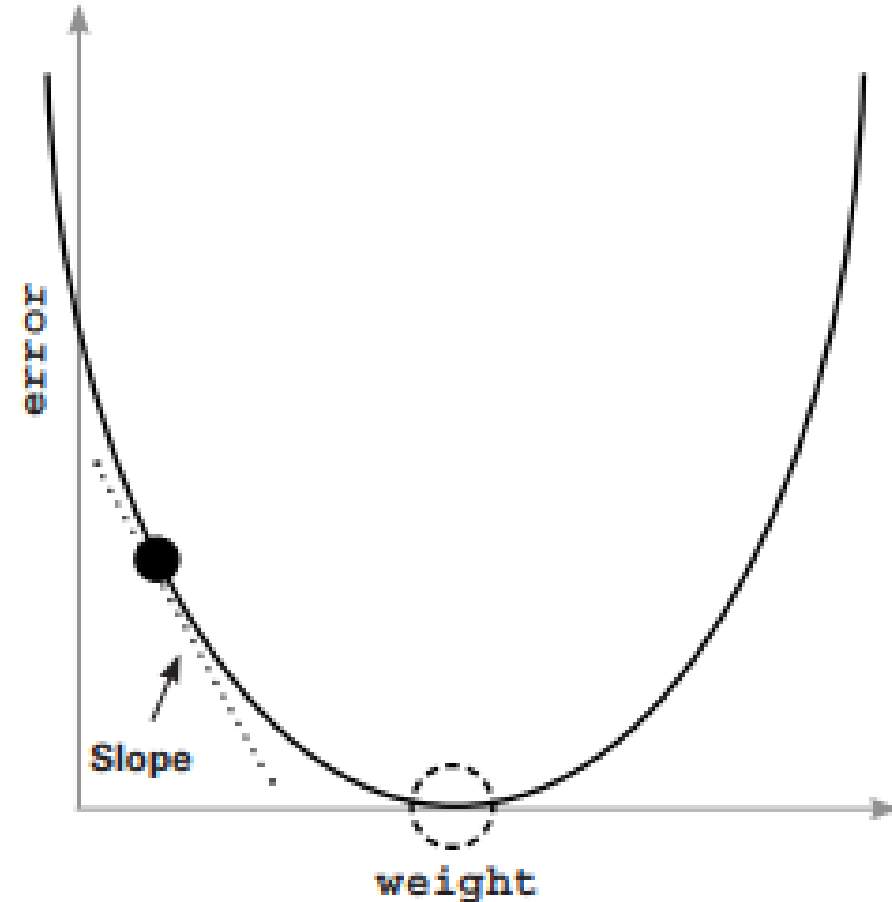
---

- This doesn't change the value of error at all! It just combines the two lines of code and computes error directly.
- Remember that input and goal\_prediction are fixed at 0.5 and 0.8, respectively (you set them before the network starts training).
- So, if you replace their variables names with the values, the secret becomes clear:

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

# Learning is just reducing error

- 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.





# Let's watch several steps of learning

---

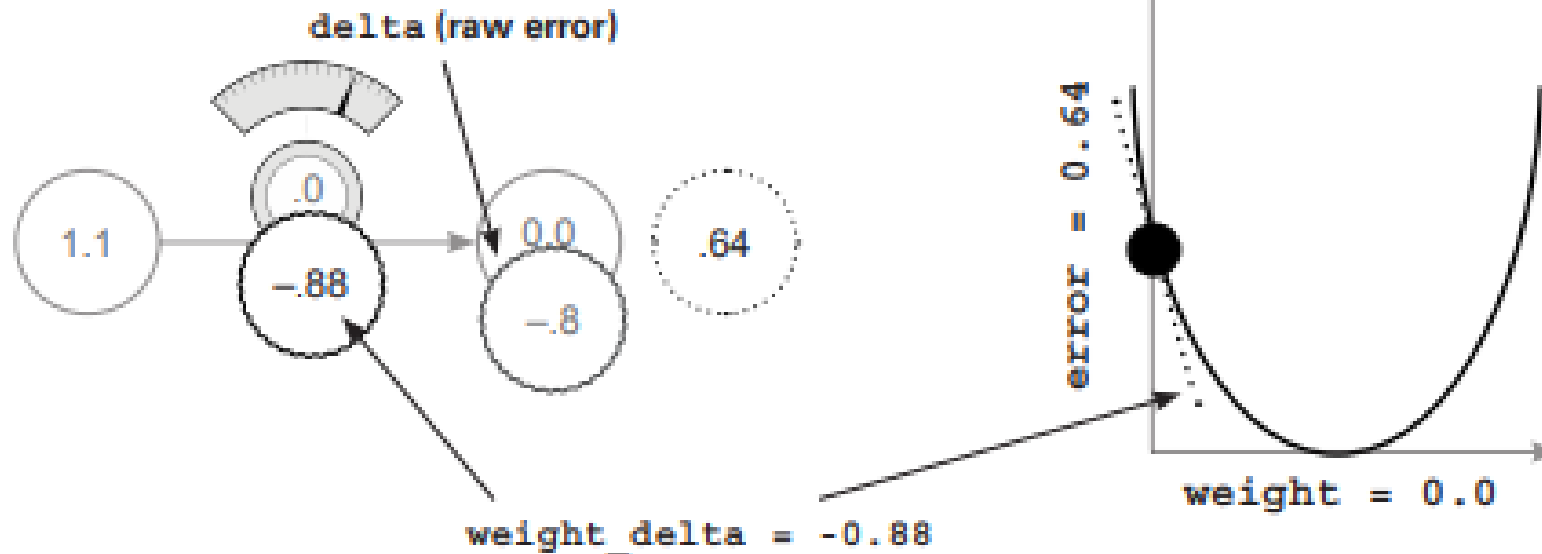
Will we eventually find the bottom of the bowl?

```
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))
```

# Let's watch several steps of learning

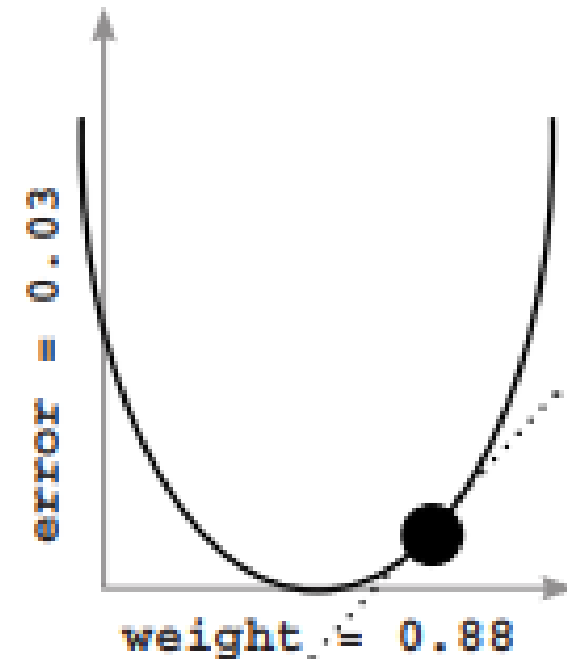
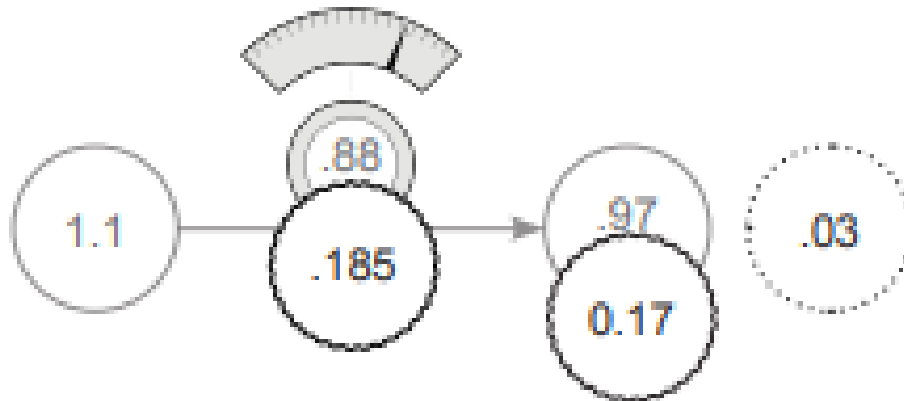
## ① A big weight increase



(Raw error modified for  
scaling, negative reversal,  
and stopping per this weight  
and input)

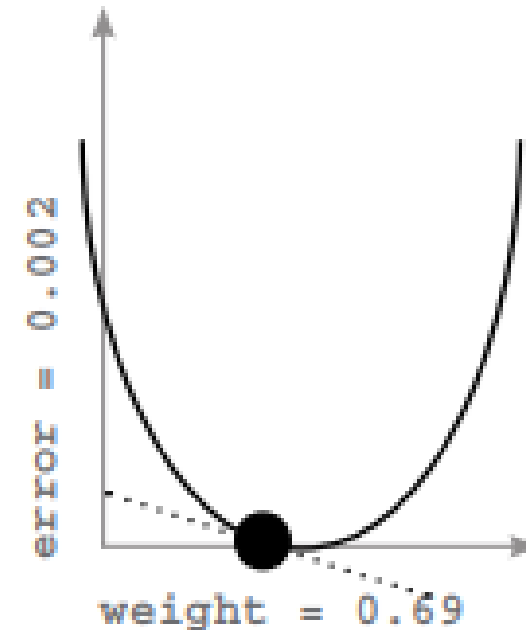
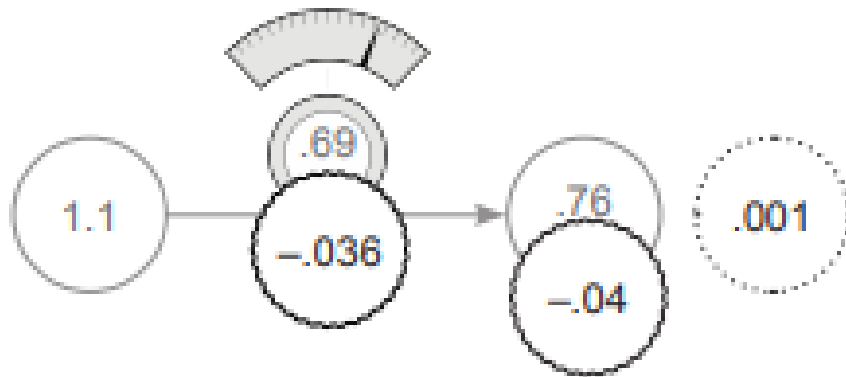
# Let's watch several steps of learning

② Overshot a bit; let's go back the other way.



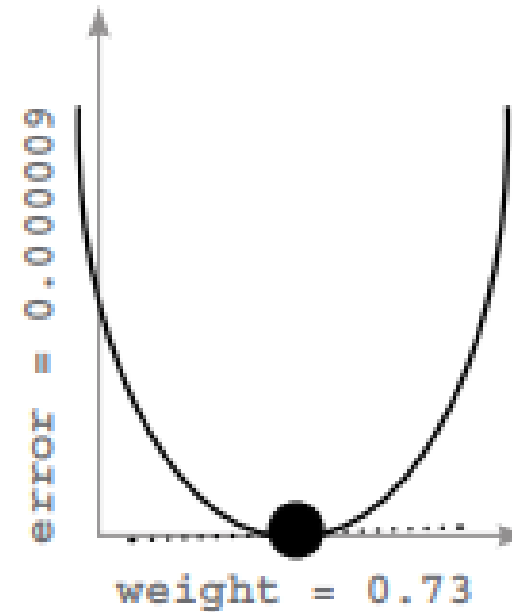
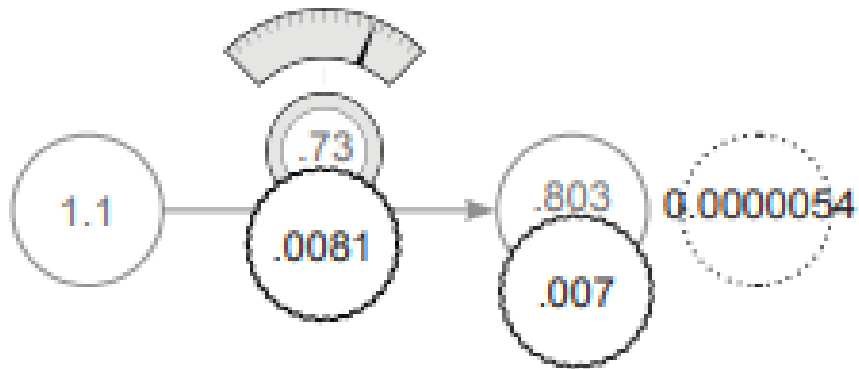
# Let's watch several steps of learning

③ Overshot again! Let's go back, but only a little.



# Let's watch several steps of learning

④ OK, we're pretty much there.





# Let's watch several steps of learning

## 5 Code output

```
-----  
Weight:0.0  
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
```

# 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.

# Why does this work? What is `weight_delta`, really?

- 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`**



# Why does this work? What is `weight_delta`, really?

---

- Now consider changing the 2, or the additions, subtractions, or multiplications.
- This is just changing how you calculate error in the first place.
- The error calculation is meaningless if it doesn't actually give a good measure of how much you missed (with the right properties mentioned a few pages ago).
- This won't do, either.

# Why does this work? What is weight\_delta, really?

- 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))
```



# Tunnel vision on one concept

---

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

- So far in this lesson, we've been hammering on the idea that learning is really just about adjusting weight to reduce error to 0.
- This is the secret sauce. Truth be told, knowing 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.



# Tunnel vision on one concept

---

- When you were wiggling weight (hot and cold learning) and studying its effect on error, you were experimentally studying the relationship between these two variables.
- It's like walking into a room with 15 different unlabeled light switches.
- You start flipping them on and off to learn about their relationship to various lights in the room.



# Tunnel vision on one concept

---

- Once you knew the relationship, you could move weight in the right direction using two simple if statements:

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

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





# Tunnel vision on one concept

---

- Now, let's go back to the earlier formula that combined the pred and error logic.
- As mentioned, they quietly define an exact relationship between error and weight:

$$\text{error} = ((\text{input} * \text{weight}) - \text{goal\_pred}) ** 2$$



# A box with rods poking out of it

---

- Picture yourself sitting in front of a cardboard box that has two circular rods sticking through two little holes.
- The blue rod is sticking out of the box by 2 inches, and the red rod is sticking out of the box by 4 inches.
- Imagine that I tell you these rods were connected, but I won't tell you in what way. You have to experiment to figure it out.



# A box with rods poking out of it

---

- However much you move the blue rod, the red rod will move by twice as much.
- You might say the following is true:

$$\text{red\_length} = \text{blue\_length} * 2$$

- As it turns out, there's a formal definition for "When I tug on this part, how much does this other part move?"

# A box with rods poking out of it

- In the case of the red and blue rods, the derivative for “How much does red move when I tug on blue?” is 2. Just 2.
- Why is it 2? That’s the multiplicative relationship determined by the formula:

$$\text{red\_length} = \text{blue\_length} * 2$$

Derivative





# A box with rods poking out of it

---

- Consider a few examples. Because the derivative of `red_length` compared to `blue_length` is 2, both numbers move in the same direction.
- More specifically, red will move twice as much as blue in the same direction.
- If the derivative had been  $-1$ , red would move in the opposite direction by the same amount.



# Derivatives: Take two

---

**Still a little unsure about them? Let's take another perspective.**

- I've heard people explain derivatives two ways.
- 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.

# Derivatives: Take two

- Let me show you by plotting our favorite function:

$$\text{error} = ((\text{input} * \text{weight}) - \text{goal\_pred}) ** 2$$

- Remember, goal\_pred and input are fixed, so you can rewrite this function:

$$\text{error} = ((0.5 * \text{weight}) - 0.8) ** 2$$

- 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: Take two

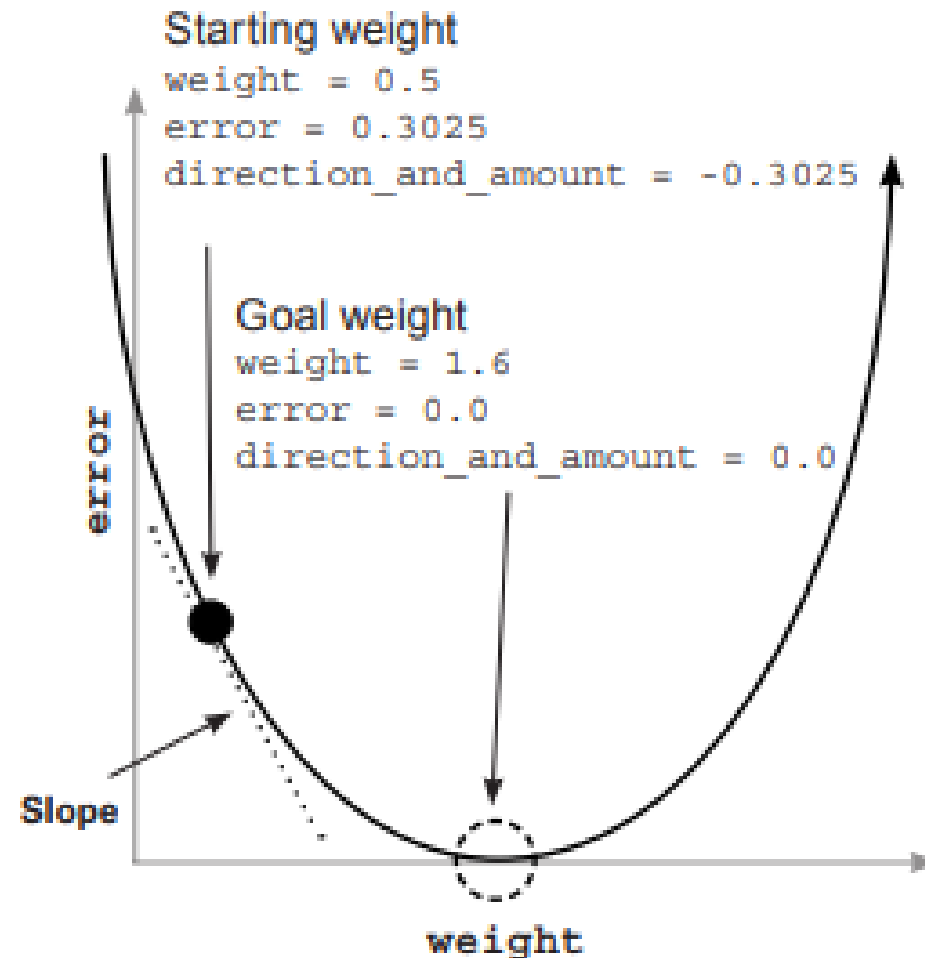
---

- 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.
- Even now, when I look at that curve, it's easy for me to lose track of what it represents.



# Derivatives: Take two

- And what's remarkable about derivatives is that they can see past the big formula for computing error (at the beginning of this section) and see this curve.
- You can compute the slope (derivative) of the line for any value of weight.





# What you really need to know

---

**With derivatives, you can pick any two variables in any formula, and know how they interact.**

- Take a look at this big whopper of a function:

$$y = (((\text{beta} * \text{gamma}) ** 2) + (\text{epsilon} + 22 - x)) ** (1/2)$$

- Here's what you need to know about derivatives.
- For any function (even this whopper), you can pick any two variables and understand their relationship with each other.
- For any function, you can pick two variables and plot them on an x-y graph as we did earlier.



# What you really need to know

---

- Bottom line: in this course, you're going to build neural networks.
- A neural network is really just one thing: a bunch of weights you use to compute an error function.
- And for any error function (no matter how complicated), you can compute the relationship between any weight and the final error of the network.



# What you don't really need to know

---

## Calculus

- Calculus turns out that learning all the methods for taking any two variables in any function and computing their relationship takes about three semesters of college.
- Truth be told, if you went through all three semesters so that you could learn how to do deep learning, you'd use only a very small subset of what you learned.



# What you don't really need to know

---

- In this course, I'm going to do what I typically do in real life (cuz I'm lazy—I mean, efficient): look up the derivative in a reference table. All you need to know is what the derivative represents.
- It's the relationship between two variables in a function so you can know how much one changes when you change the other.
- It's just the sensitivity between two variables.



# How to use a derivative to learn

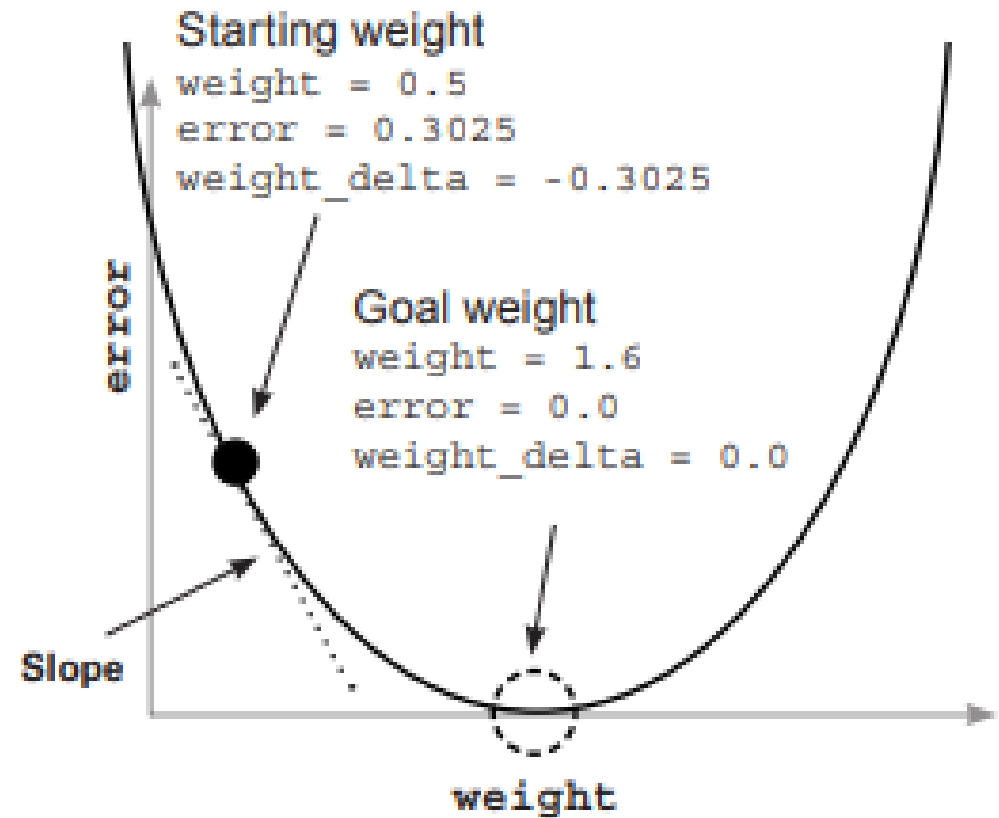
---

**weight\_delta is your derivative.**

- What's the difference between error and the derivative of error and weight? error is a measure of how much you missed.
- The derivative defines the relationship between each weight and how much you missed.

# How to use a derivative to learn

- You've learned the relationship between two variables in a function, but how do you exploit that relationship? As it turns out, this is incredibly visual and intuitive.
- Check out the error curve again.
- The black dot is where weight starts out: (0.5).





# How to use a derivative to learn

---

- The slope of a line or curve always points in the opposite direction of the lowest point of the line or curve.
- So, if you have a negative slope, you increase weight to find the minimum of error.
- Check it out. So, how do you use the derivative to find the error minimum (lowest point in the error graph)? You move the opposite direction of the slope—the opposite direction of the derivative.





# How to use a derivative to learn

---

- This method for learning (finding error minimums) is called gradient descent.
- This name should seem intuitive.
- You move the weight value opposite the gradient value, which reduces error to 0.


# Look familiar?

```
weight = 0.0
goal_pred = 0.8
input = 1.1

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

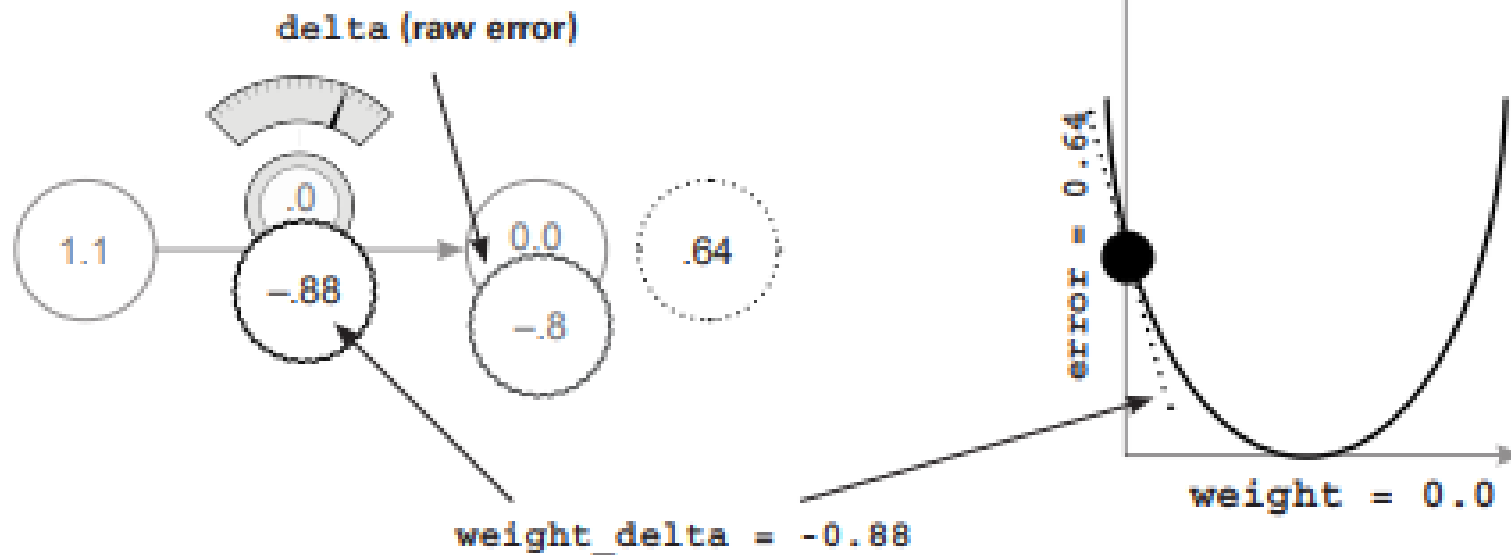
print("Error:" + str(error) + " Prediction:" + str(pred))
```

Derivative  
(how fast the error  
changes, given changes  
in the weight)



# Look familiar?

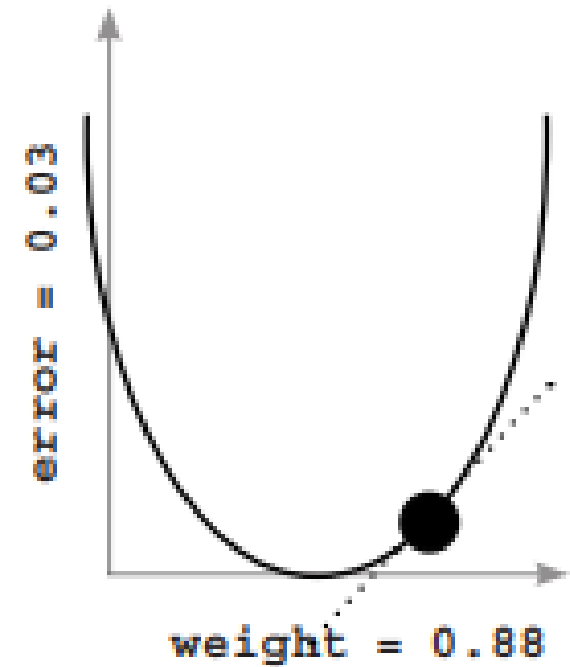
## ① A big weight increase



(Raw error modified for  
scaling, negative reversal,  
and stopping per this weight  
and input)

# Look familiar?

② Overshot a bit; let's go back the other way.



# Breaking gradient descent

- Just give me the code!

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

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = input * delta
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
```

# Breaking gradient descent

- When I run this code, I see the following output:

```
Error:0.3025 Prediction:0.25
Error:0.17015625 Prediction:0.3875
Error:0.095712890625 Prediction:0.490625
...
Error:1.7092608064e-05 Prediction:0.79586567925
Error:9.61459203602e-06 Prediction:0.796899259437
Error:5.40820802026e-06 Prediction:0.797674444578
```

# Breaking gradient descent

- Let's try setting input equal to 2, but still try to get the algorithm to predict 0.8.
- What happens? Take a look at the output:

```
Error:0.04 Prediction:1.0
```

```
Error:0.36 Prediction:0.2
```

```
Error:3.24 Prediction:2.6
```

```
...
```

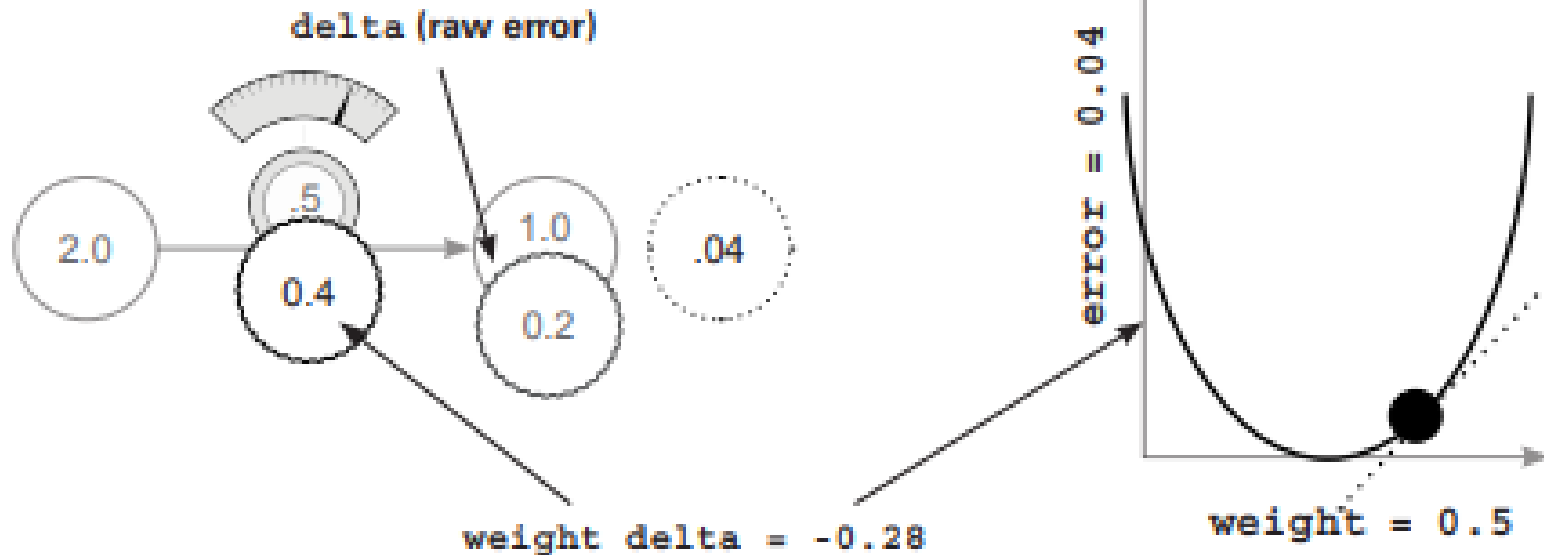
```
Error:6.67087267987e+14 Prediction:-25828031.8
```

```
Error:6.00378541188e+15 Prediction:77484098.6
```

```
Error:5.40340687069e+16 Prediction:-232452292.6
```

# Visualizing the overcorrections

## ① A big weight increase

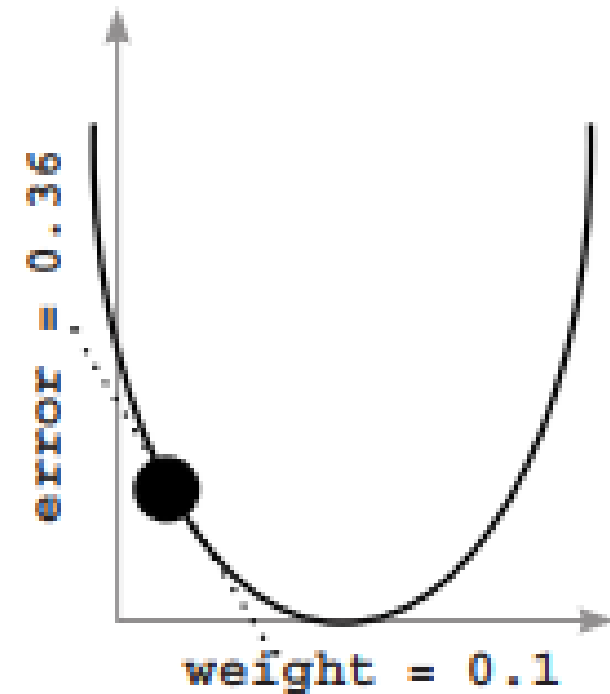
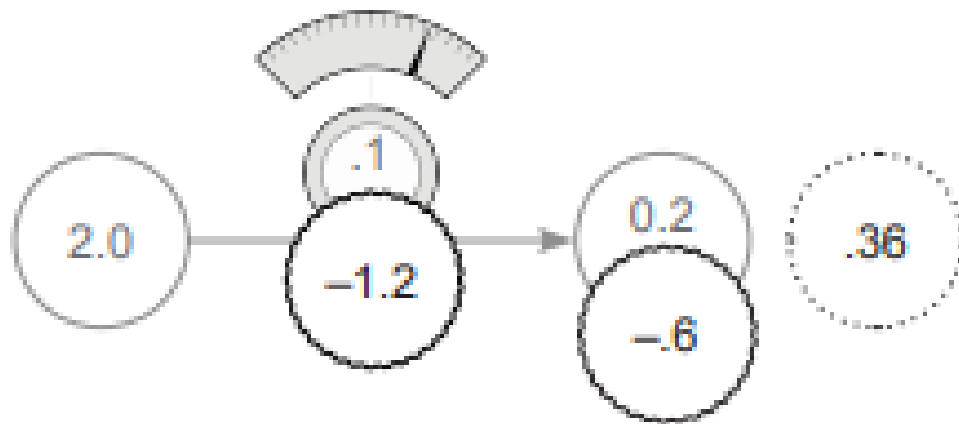


(Raw error modified for scaling,  
negative reversal, and stopping  
per this weight and input)



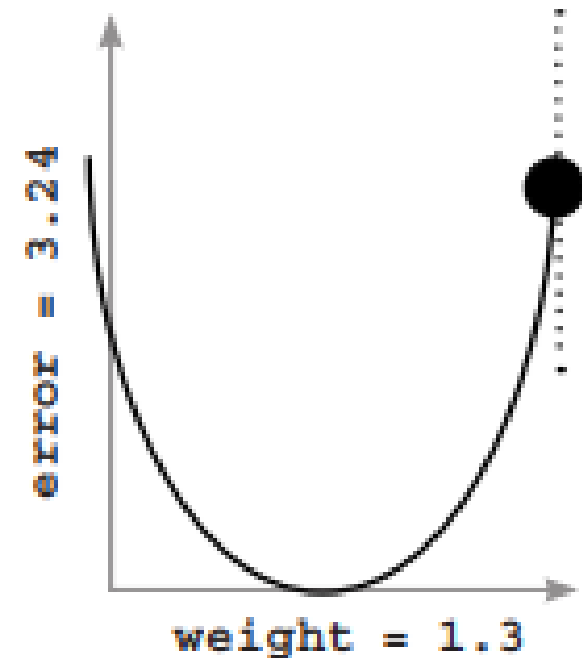
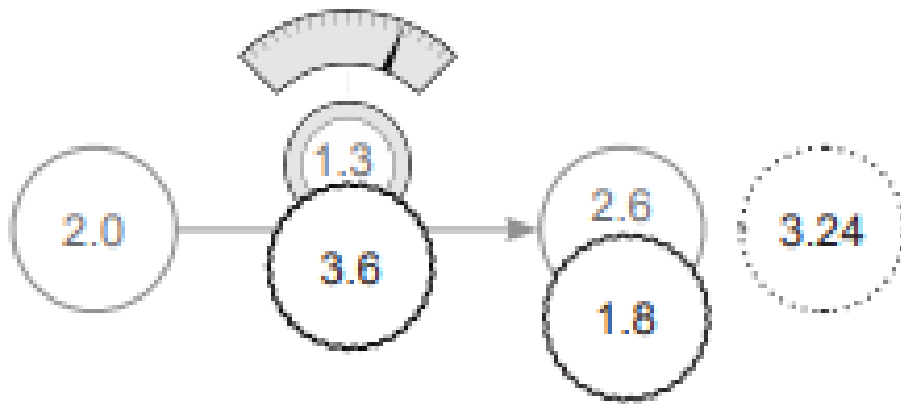
# Visualizing the overcorrections

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



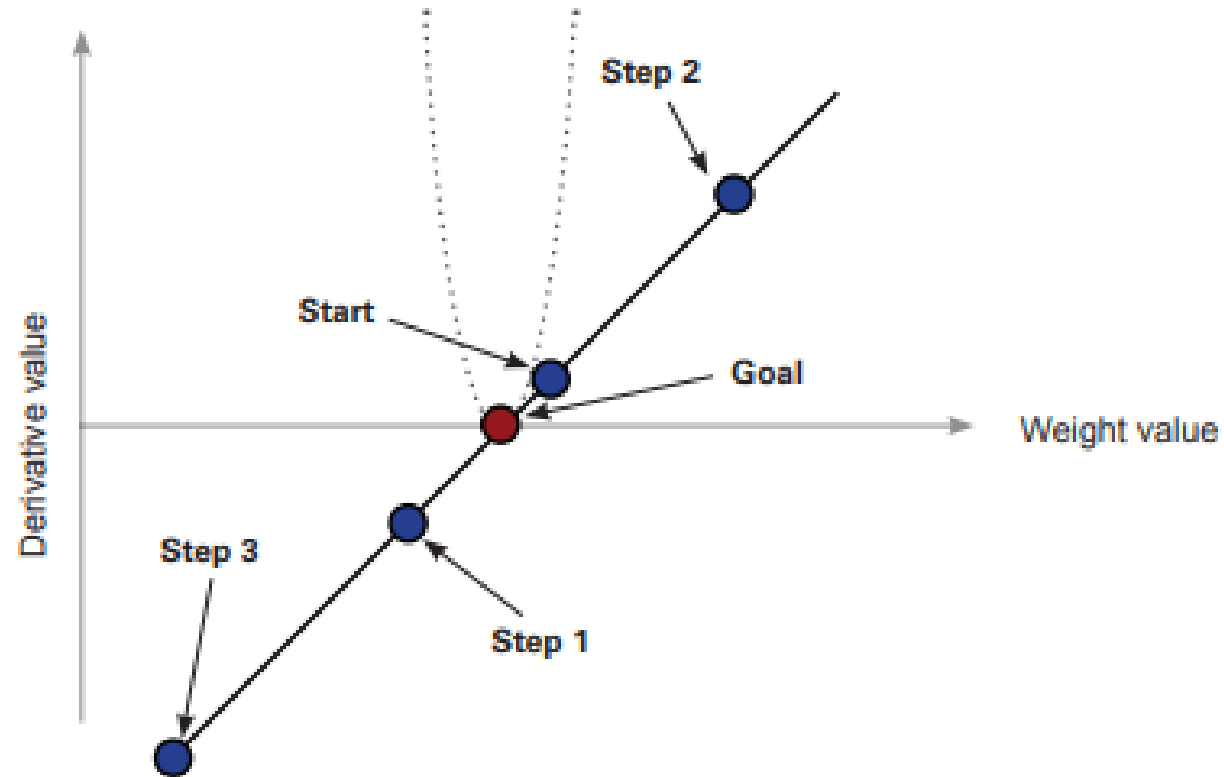
# Visualizing the overcorrections

③ Overshot again! Let's go back, but only a little.



# Divergence

Sometimes neural networks explode in value. Oops?





# Divergence

---

- What really happened? The explosion in the error was caused by the fact that you made the input larger.
- Consider how you're updating the weight:

$$\text{weight} = \text{weight} - (\text{input} * (\text{pred} - \text{goal\_pred}))$$



# Divergence

---

- If you have a big input, the prediction is very sensitive to changes in the weight (**because  $\text{pred} = \text{input} * \text{weight}$** ).
- This can cause the network to overcorrect.
- In other words, even though the weight is still starting at 0.5, the derivative at that point is very steep.
- See how tight the U-shaped error curve is in the graph?



# Introducing alpha

---

**It's the simplest way to prevent overcorrecting weight updates.**

- What's the problem you're trying to solve? That if the input is too big, then the weight update can overcorrect.
- What's the symptom? That when you overcorrect, the new derivative is even larger in magnitude than when you started (although the sign will be the opposite).



# Introducing alpha

---

- The symptom is this overshooting.
- The solution is to multiply the weight update by a fraction to make it smaller.
- In most cases, this involves multiplying the weight update by a single real-valued number between 0 and 1, known as alpha.



# Alpha in code

---

## Where does our “alpha” parameter come into play?

- You just learned that alpha reduces the weight update so it doesn't overshoot.
- How does this affect the code? Well, you were updating the weights according to the following formula:

$$\text{weight} = \text{weight} - \text{derivative}$$





# Alpha in code

---

- Accounting for alpha is a rather small change, as shown next.
- Notice that if alpha is small (say, 0.01), it will reduce the weight update considerably, thus preventing it from overshooting:

**$\text{weight} = \text{weight} - (\text{alpha} * \text{derivative})$**

# Alpha in code

- That was easy. Let's install alpha into the tiny implementation from the beginning of this lesson and run it where input = 2 (which previously didn't work):

```
weight = 0.5
goal_pred = 0.8
input = 2
alpha = 0.1
```

What happens when you make alpha crazy small or big? What about making it negative?

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

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

# Alpha in code

```
Error:0.04 Prediction:1.0  
Error:0.0144 Prediction:0.92  
Error:0.005184 Prediction:0.872
```

...

```
Error:1.14604719983e-09 Prediction:0.800033853319  
Error:4.12576991939e-10 Prediction:0.800020311991  
Error:1.48527717099e-10 Prediction:0.800012187195
```



# Memorizing

---

**It's time to really learn this stuff.**

- This may sound a bit intense, but I can't stress enough the value I've found from this exercise: see if you can build the code from the previous section in a Jupyter notebook (or a .py file, if you must) from memory.
- I know that might seem like overkill, but I (personally) didn't have my "click" moment with neural networks until I was able to perform this task.