

Deep Learning





Table of Contents

- 2 Fundamental concepts: how do machines learn?: 4
 - 3 Introduction to neural prediction: forward propagation: 23
 - 4 Introduction to neural learning: gradient descent: 83
 - 5 learning multiple weights at a time: generalizing gradient descent: 167
 - 6 building your first deep neural network: introduction to backpropagation: 209
 - 7 how to picture neural networks: in your head and on paper: 272
- 



Table of Contents

8 learning signal and ignoring noise: introduction to regularization and batching: 292

9 modeling probabilities and nonlinearities: activation functions: 319

10 neural learning about edges and corners: intro to convolutional neural networks: 355

11 neural networks that understand language: king – man + woman == ?: 371

Lesson 2 Fundamental concepts: how do machines learn?





How do machines learn?



In this lesson we learn

- What are deep learning, machine learning, and artificial intelligence?
- What are parametric models and nonparametric models?
- What are supervised learning and unsupervised learning?
- How can machines learn?



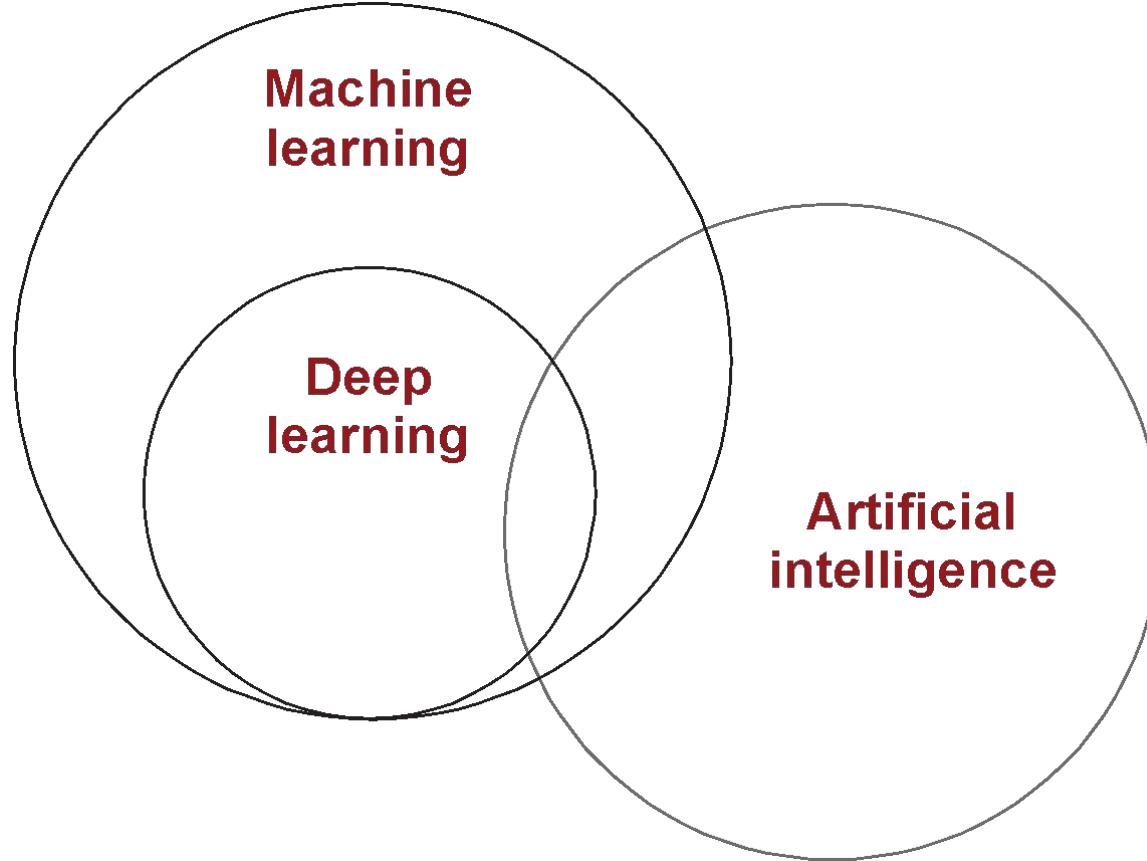
What is deep learning?



- Deep learning is a subset of machine learning, which is a field dedicated to the study and development of machines that can learn (sometimes with the goal of eventually attaining general artificial intelligence).
 - In industry, deep learning is used to solve practical tasks in a variety of fields such as computer vision (image), natural language processing (text), and automatic speech recognition (audio).
- 



What is deep learning?





What is machine learning?

- Deep learning is a subset of machine learning, what is machine learning? Most generally, it is what its name implies.
- Machine learning is a subfield of computer science wherein machines learn to perform tasks for which they were not explicitly programmed.
- In short, machines observe a pattern and attempt to imitate it in some way that can be either direct or indirect.

Machine learning ~ = **Monkey see, monkey do**



Supervised machine learning



Supervised learning transforms datasets.

- Supervised learning is a method for transforming one dataset into another.
- For example, if you had a dataset called Monday Stock Prices that recorded the price of every stock on every Monday for the past 10 years, and a second dataset called Tuesday Stock Prices recorded over the same time period, a supervised learning algorithm might try to use one to predict the other.



Supervised machine learning



- The majority of work using machine learning results in the training of a supervised classifier of some kind.
- Even unsupervised machine learning (which you'll learn more about in a moment) is typically done to aid in the development of an accurate supervised machine learning algorithm.



Unsupervised machine learning

Unsupervised learning groups your data.

- Unsupervised learning shares a property in common with supervised learning: it transforms one dataset into another.
 - But the dataset that it transforms into is not previously known or understood.
 - Unlike supervised learning, there is no “right answer” that you’re trying to get the model to duplicate.
 - You just tell an unsupervised algorithm to “find patterns in this data and tell me about them.”
- 



Unsupervised machine learning



- I have good news! This idea of clustering is something you can reliably hold onto in your mind as the definition of unsupervised learning.
 - Even though there are many forms of unsupervised learning, all forms of unsupervised learning can be viewed as a form of clustering.
 - You'll discover more on this later in the course.
- 

Parametric vs. nonparametric learning

Oversimplified: Trial-and-error learning vs. counting and probability

- The last two pages divided all machine learning algorithms into two groups: supervised and unsupervised.
- Now, we're going to discuss another way to divide the same machine learning algorithms into two groups: **parametric and nonparametric**.





Supervised parametric learning



Oversimplified: Trial-and-error learning using knobs

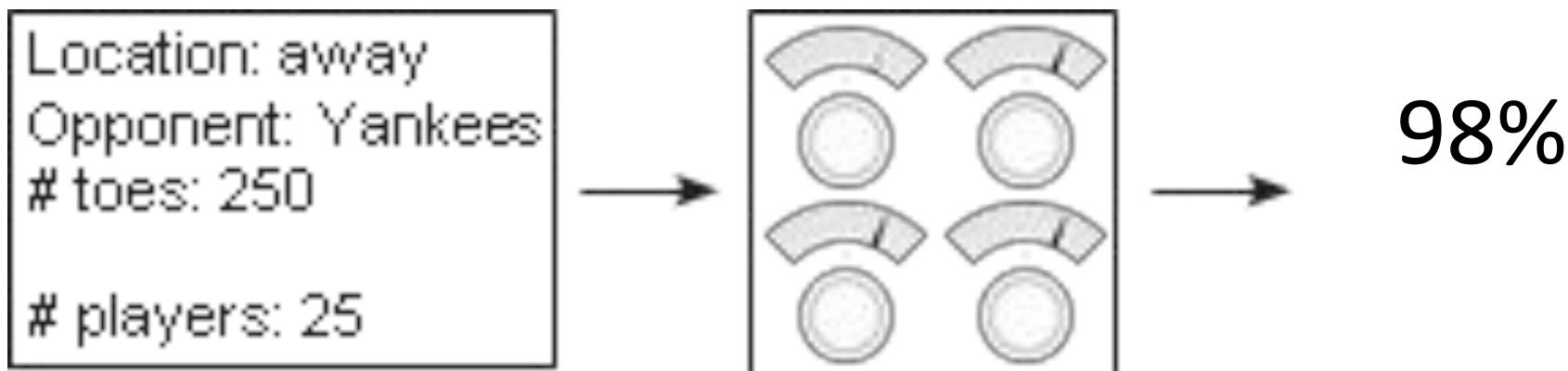
- Supervised parametric learning machines are machines with a fixed number of knobs (that's the parametric part), wherein learning occurs by turning the knobs.
- Input data comes in, is processed based on the angle of the knobs, and is transformed into a prediction.



Supervised parametric learning

Step 1: Predict

- The first step, as mentioned, is to gather sports statistics, send them through the machine, and make a prediction about the probability that the Red Sox will win.





Supervised parametric learning



Step 2: Compare to the truth pattern

- The second step is to compare the prediction (98%) with the pattern you care about (whether the Red Sox won).
- Sadly, they lost, so the comparison is

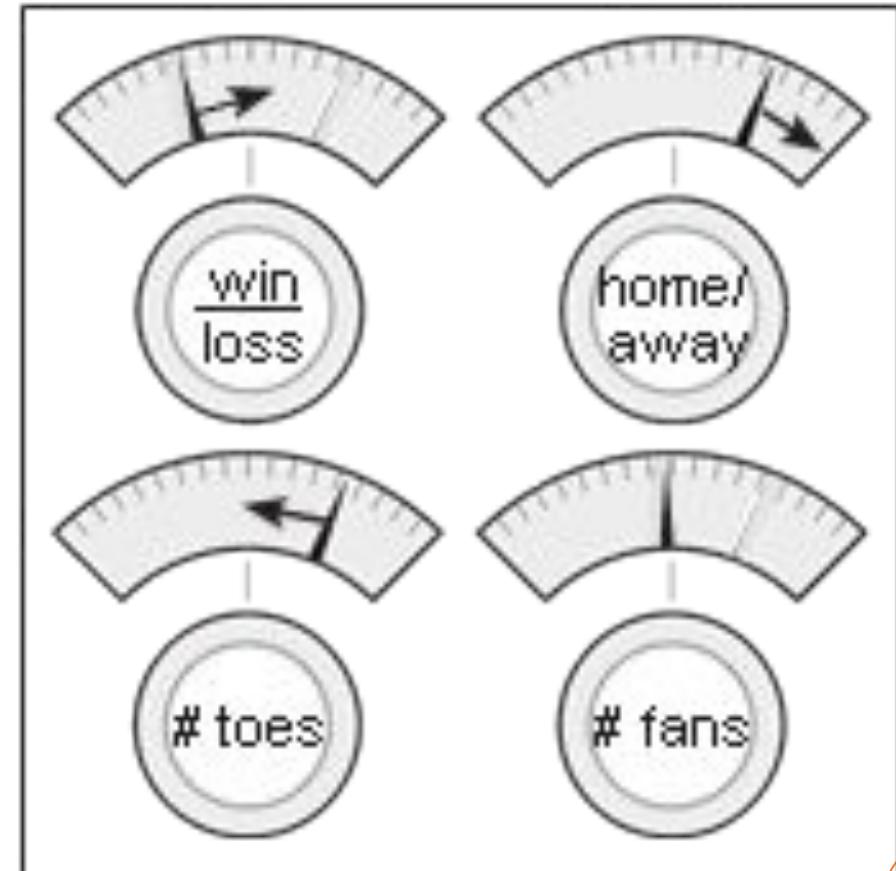
Pred: 98% > Truth: 0%

Supervised parametric learning

Step 3: Learn the pattern

- This step adjusts the knobs by studying both how much the model missed by (98%) and what the input data was (sports stats) at the time of prediction.

Adjusting sensitivity by turning knobs



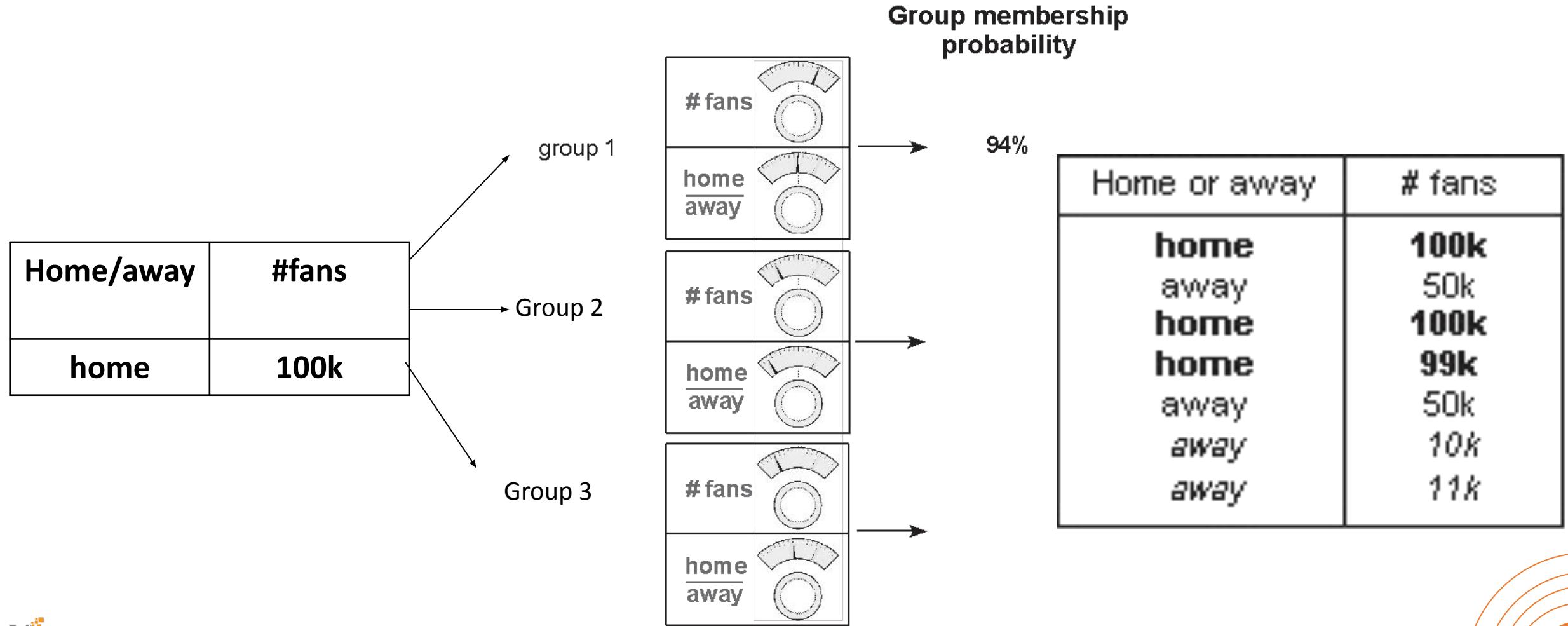


Unsupervised parametric learning



- Unsupervised parametric learning uses a very similar approach.
- Let's walk through the steps at a high level.
- Remember that unsupervised learning is all about grouping data.
- Unsupervised parametric learning uses knobs to group data.

Unsupervised parametric learning

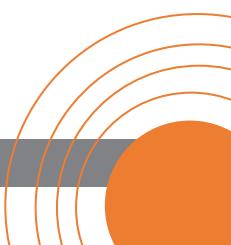




Nonparametric learning



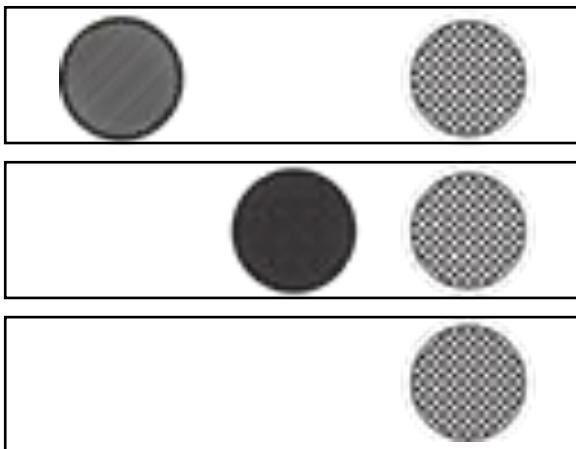
Oversimplified: Counting-based methods

- Nonparametric learning is a class of algorithm wherein the number of parameters is based on data (instead of predefined).
 - This lends itself to methods that generally count in one way or another, thus increasing the number of parameters based on the number of items being counted within the data.
- 



Nonparametric learning

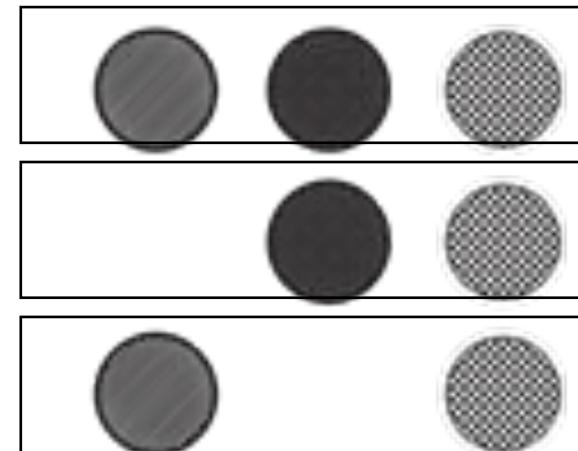
- After counting only a few examples, this model would then be able to predict that middle lights always (100%) cause cars to go, and right lights only sometimes (50%) cause cars to go



Stop

Go

Stop



Go

Go

Stop



Summary



- In this lesson, we've gone a level deeper into the various flavors of machine learning.
 - You learned that a machine learning algorithm is either supervised or unsupervised and either parametric or nonparametric.
 - Furthermore, we explored exactly what makes these four different groups of algorithms distinct.
- 

Lesson 3 Introduction to neural prediction: forward propagation





Forward propagation



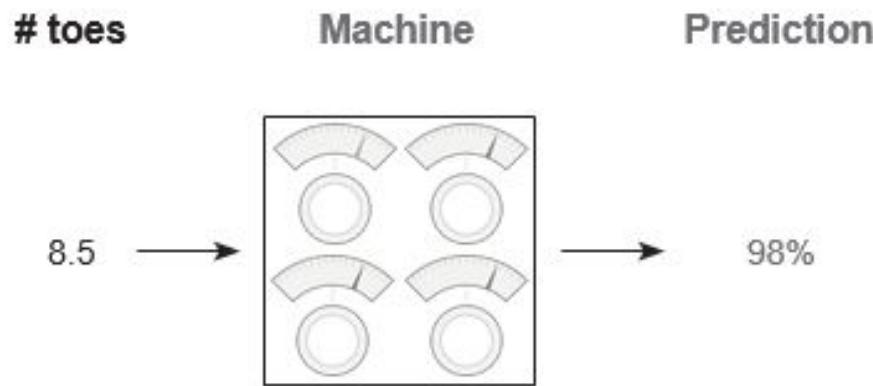
In this lesson we learn

- A simple network making a prediction
- What is a neural network, and what does it do?
- Making a prediction with multiple inputs
- Making a prediction with multiple outputs
- Making a prediction with multiple inputs and outputs
- Predicting on predictions

Forward propagation

Step 1: Predict

- In this lesson, you'll learn more about what these three different parts of a neural network prediction look like under the hood.
- Let's start with the first one: the data.





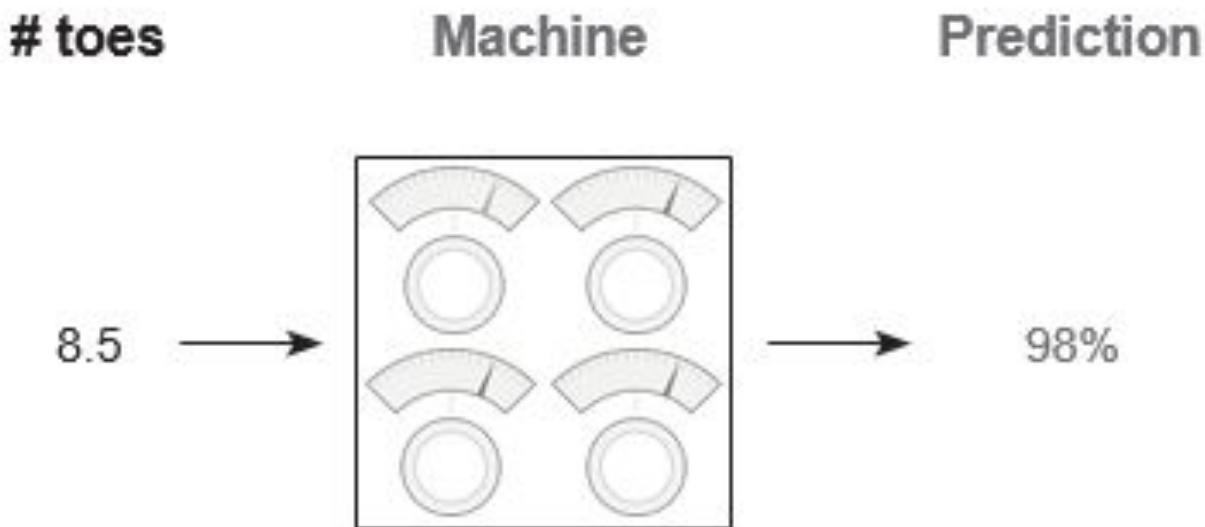
Forward propagation



- Later, you'll find that the number of datapoints you process at a time has a significant impact on what a network looks like.
- You might be wondering, "How do I choose how many datapoints to propagate at a time?" The answer is based on whether you think the neural network can be accurate with the data you give it.

Forward propagation

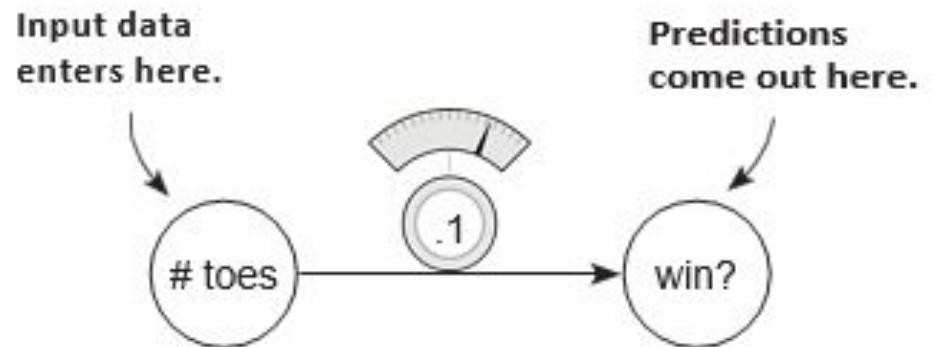
- Let's stick with a single prediction of the likelihood that the baseball team will win:



Forward propagation

- (Abstractly, these “knobs” are actually called weights, and I’ll refer to them as such from here on out.)
- So, without further ado, here’s your first neural network, with a single weight mapping from the input “# toes” to the output “win?”:

b an empty network

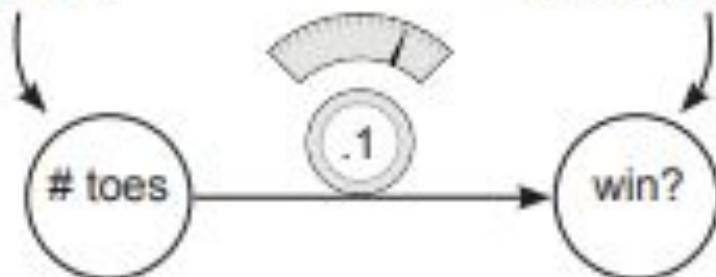


A simple neural network making a prediction

Let's start with the simplest neural network possible.

1 An empty network

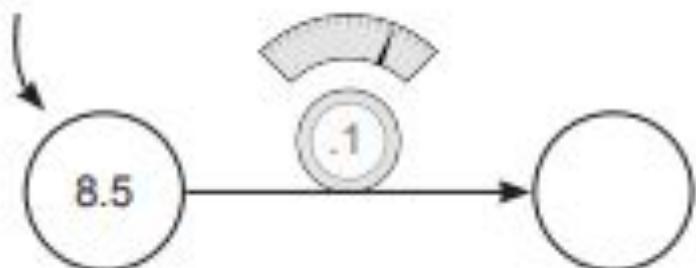
Input data
enters here.



```
weight = 0.1  
  
def neural_network(input, weight):  
  
    prediction = input * weight  
  
    return prediction
```

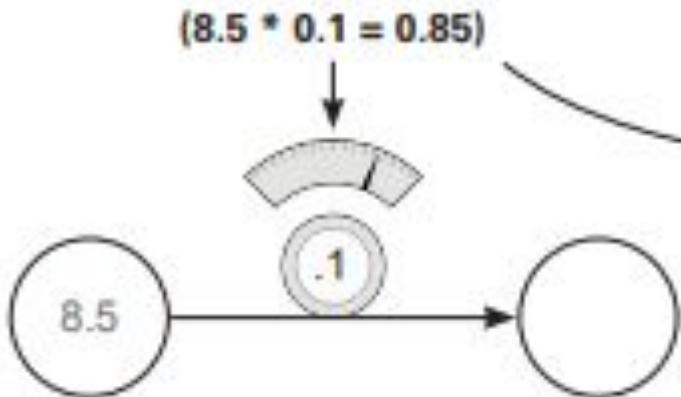
② Inserting one input datapoint

Input data
(# toes)



```
number_of_toes = [8.5, 9.5, 10, 9]  
input = number_of_toes[0]  
pred = neural_network(input, weight)  
print(pred)
```

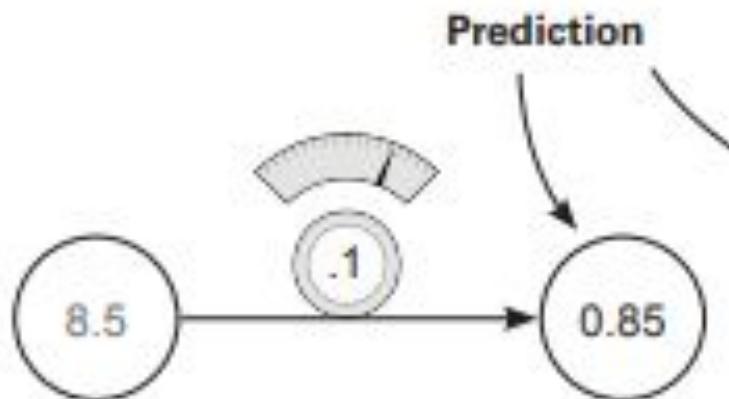
③ Multiplying input by weight



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

A simple neural network making a prediction

④ Depositing the prediction



```
number_of_toes = [8.5, 9.5, 10, 9]  
input = number_of_toes[0]  
pred = neural_network(input,weight)
```



What is a neural network?



Here is your first neural network.

To start a neural network, open a Jupyter notebook and run this code:

```
weight = 0.1

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

The network





What is a neural network?



Now, run the following:

```
number_of_toes = [8.5, 9.5, 10, 9]  
input = number_of_toes[0]  
pred = neural_network(input,weight)  
print(pred)
```



How you use the
network to predict
something





What does this neural network do?

It multiplies the input by a weight. It “scales” the input by a certain amount.

- In the previous section, you made your first prediction with a neural network.
- A neural network, in its simplest form, uses the power of multiplication.

What does this neural network do?

- If the weight is 0.01, then the network will divide the input by 100.
- As you can see, some weight values make the input bigger, and

1 An empty network

Input data enters here.

toes

.1

win?

Predictions come out here.

```
weight = 0.1  
def neural_network(input, weight):  
    prediction = input * weight  
    return prediction
```

The diagram illustrates a simple neural network structure. It consists of three nodes: an input node labeled '# toes', a weight node labeled '.1', and an output node labeled 'win?'. An arrow points from the '# toes' node to the '.1' node, and another arrow points from the '.1' node to the 'win?' node. To the left of the '# toes' node, there is an annotation 'Input data enters here.' with an arrow pointing to the node. To the right of the 'win?' node, there is an annotation 'Predictions come out here.' with an arrow pointing away from the node. The code on the right defines a function 'neural_network' that takes 'input' and 'weight' as parameters, calculates the product of input and weight, and returns the prediction.

What does this neural network do?

- Later neural networks will accept larger, more complicated input and weight values, but this same underlying premise will always ring true.

② Inserting one input datapoint

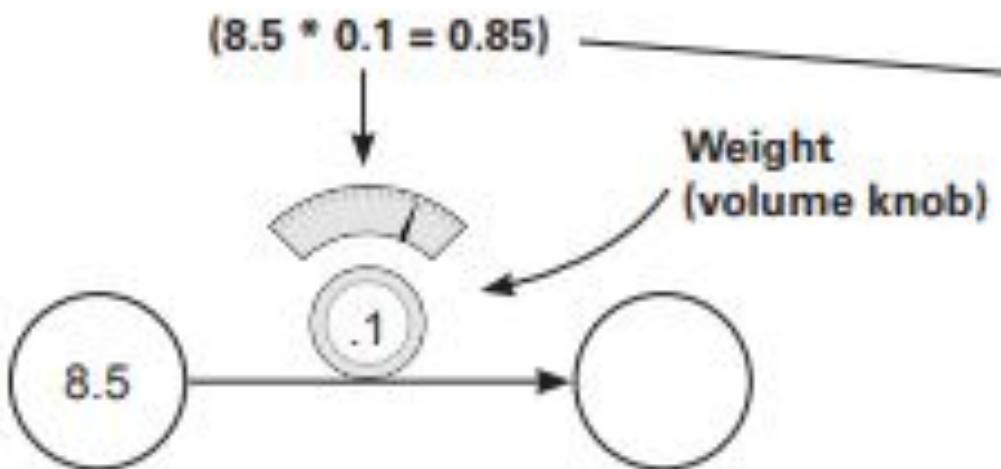
Input data
(# toes)



```
number_of_toes = [8.5, 9.5, 10, 9]  
input = number_of_toes[0]  
pred = neural_network(input,weight)
```

What does this neural network do?

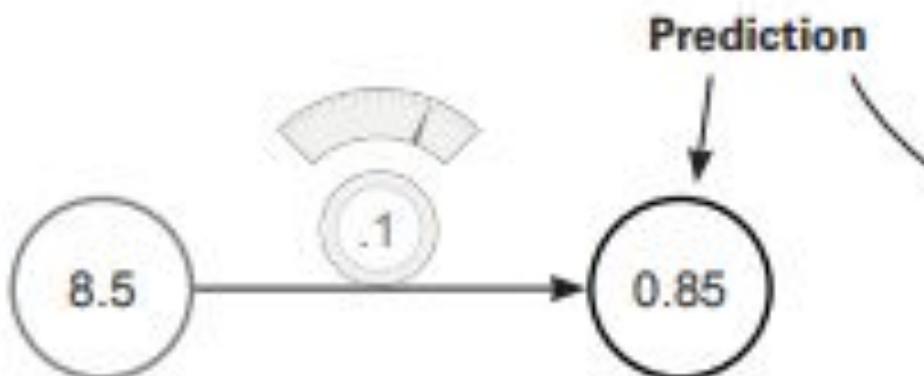
③ Multiplying input by weight



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

What does this neural network do?

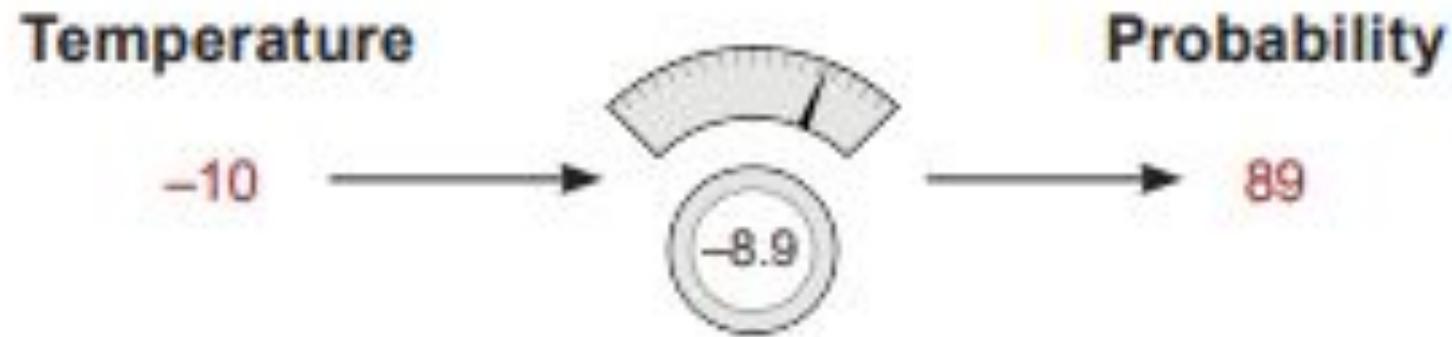
④ Depositing the prediction



```
number_of_toes = [8.5, 9.5, 10, 9]  
input = number_of_toes[0]  
pred = neural_network(input,weight)
```

What does this neural network do?

- Perhaps you want to predict the probability that people will wear coats today.
- If the temperature is -10 degrees Celsius, then a negative weight will predict a high probability that people will wear their coats.





Making a prediction with multiple inputs

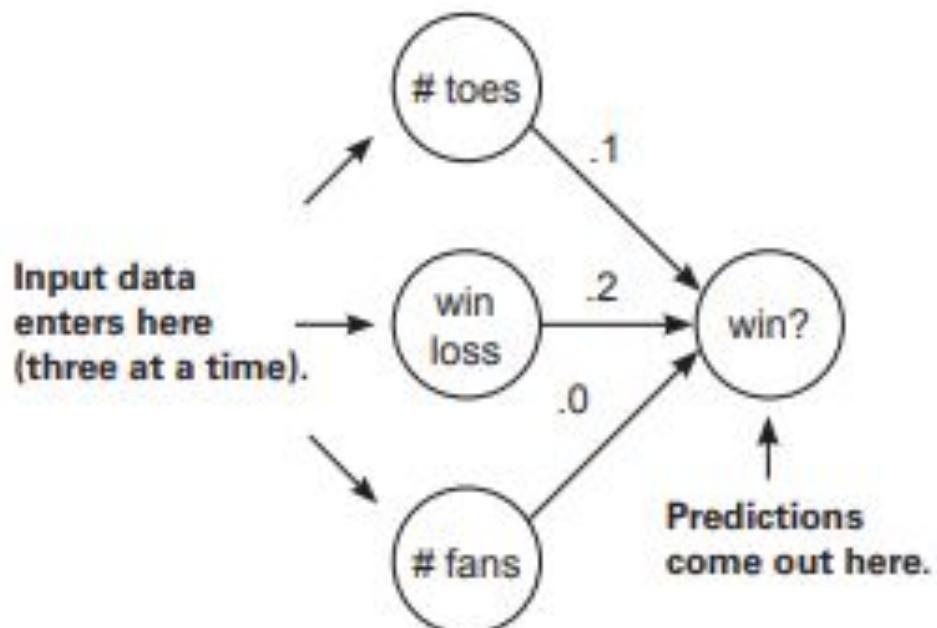
Neural networks can combine intelligence from multiple datapoints.

- The previous neural network was able to take one datapoint as input and make one prediction based on that datapoint.
- Perhaps you've been wondering, "Is the average number of toes really a good predictor, all by itself?" If so, you're onto something.

Making a prediction with multiple inputs

- Take a look at the next prediction:

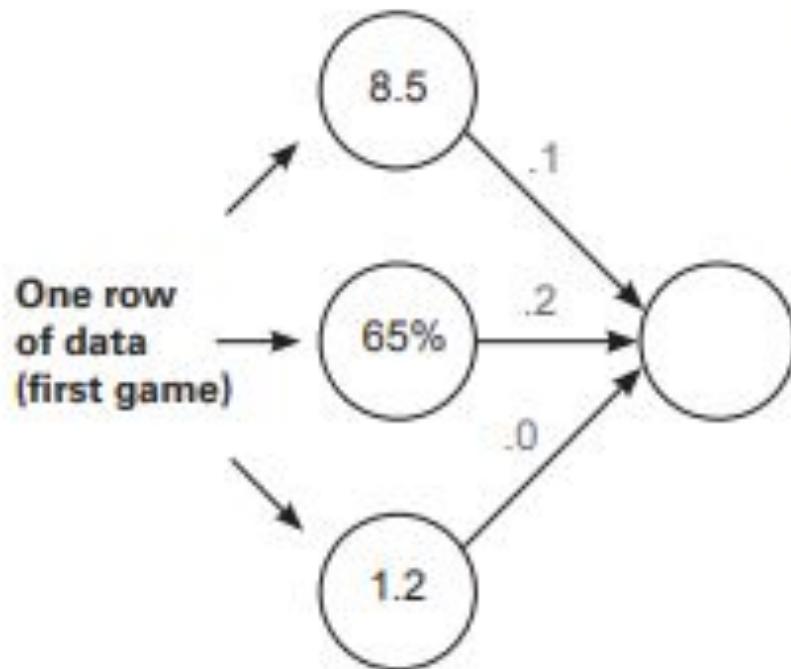
① An empty network with multiple inputs



```
weights = [0.1, 0.2, 0]  
def neural_network(input, weights):  
    pred = w_sum(input,weights)  
    return pred
```

Making a prediction with multiple inputs

② Inserting one input datapoint



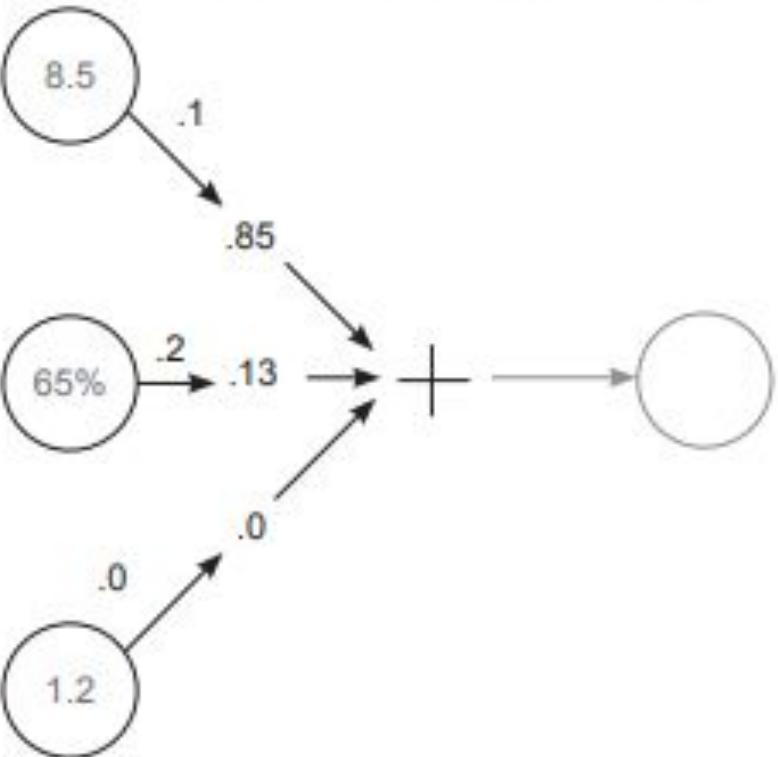
This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions).

```
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]
```

```
input = [toes[0], wlrec[0], nfans[0]]  
pred = neural_network(input, weights)
```

Input corresponds to every entry for the first game of the season.

③ Performing a weighted sum of inputs



```
def w_sum(a,b):  
    assert(len(a) == len(b))  
  
    output = 0  
  
    for i in range(len(a)):  
        output += (a[i] * b[i])  
  
    return output
```

```
def neural_network(input, weights):  
    pred = w_sum(input,weights)  
  
    return pred
```

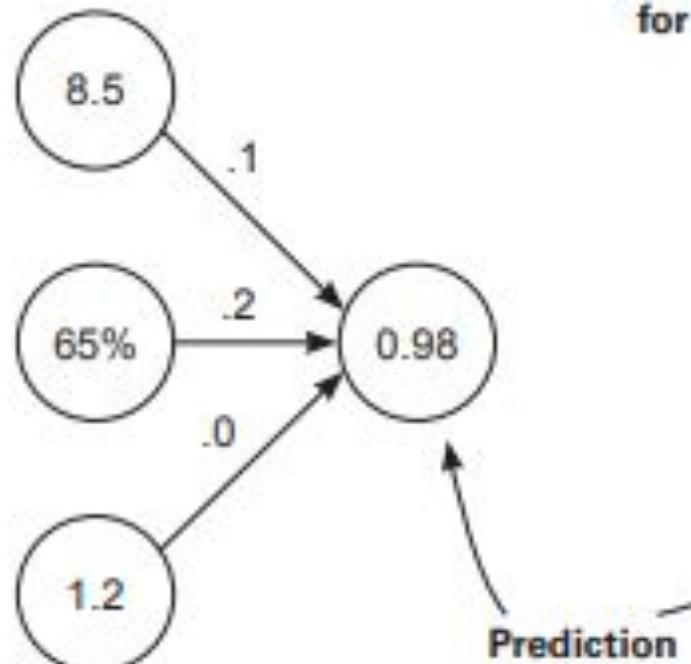
Inputs	Weights	Local predictions	
(8.50	* 0.1)	= 0.85	= toes prediction
(0.65	* 0.2)	= 0.13	= wlrec prediction
(1.20	* 0.0)	= 0.00	= fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

$$0.85 + 0.13 + 0.00 = 0.98$$

Making a prediction with multiple inputs

④ Depositing the prediction



Input corresponds to every entry
for the first game of the season.

```
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]  
  
input = [toes[0],wlrec[0],nfans[0]]  
  
pred = neural_network(input,weights)  
print(pred)
```

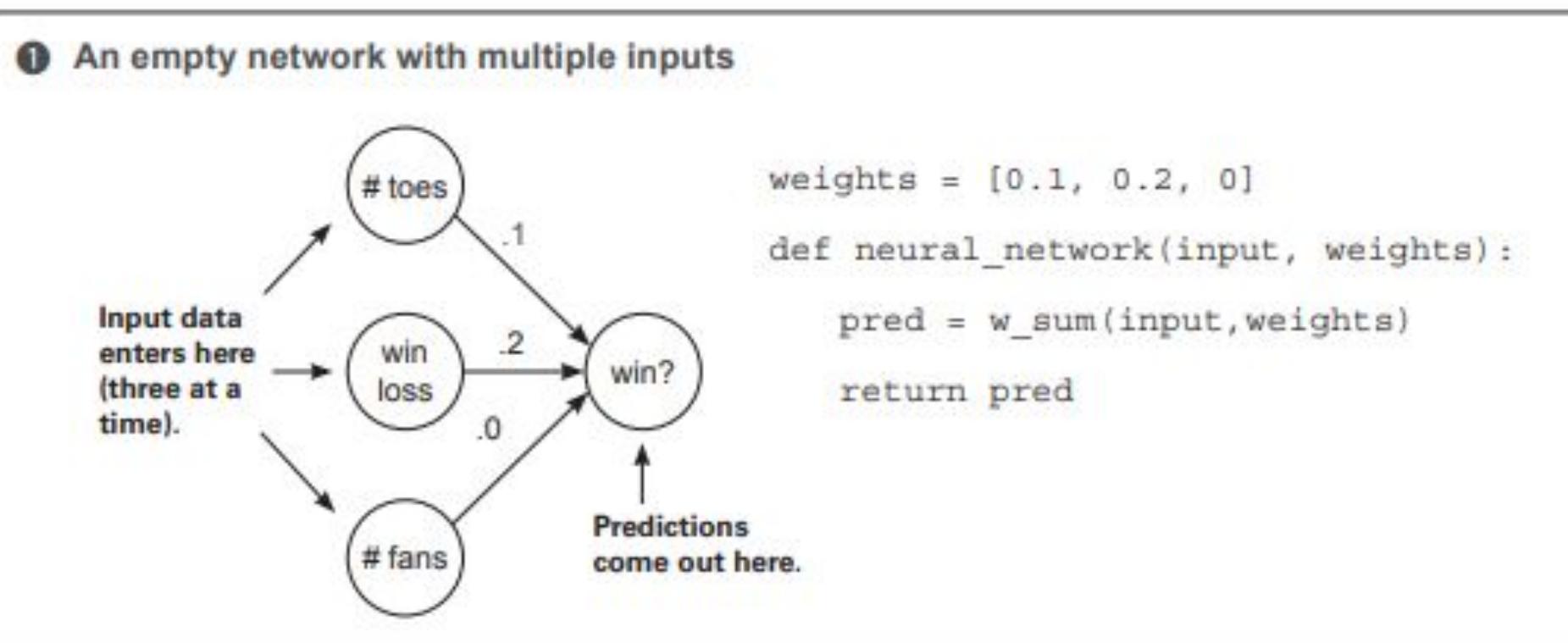
Multiple inputs: What does this neural network do?

It multiplies three inputs by three knob weights and sums them. This is a weighted sum.

- In the example, that datapoint was a baseball team's average number of toes per player.
- You learned that in order to make accurate predictions, you need to build neural networks that can combine multiple inputs at the same time

Multiple inputs: What does this neural network do?

- Fortunately, neural networks are perfectly capable of doing so.



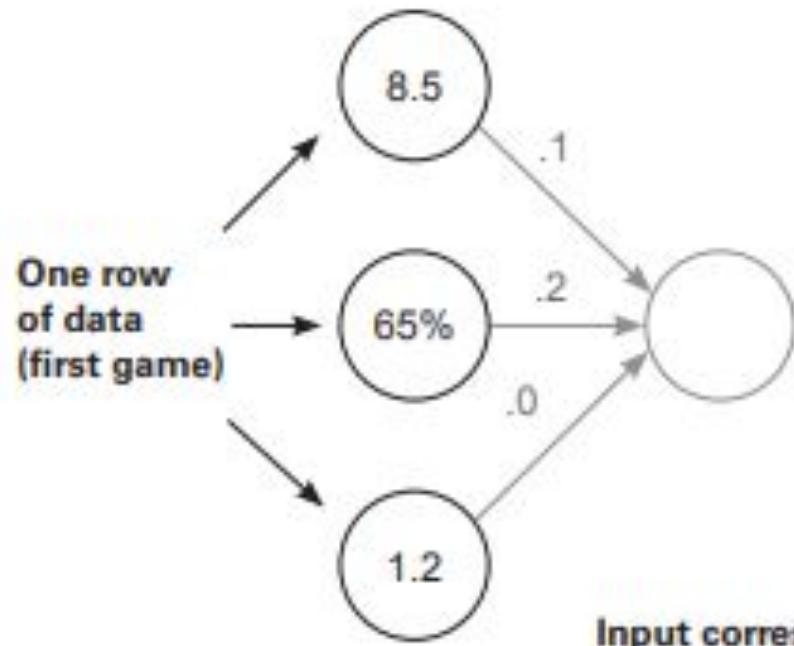


Multiple inputs: What does this neural network do?

- The new property here is that, because you have multiple inputs, you have to sum their respective predictions.
- Thus, you multiply each input by its respective weight and then sum all the local predictions together.
- This is called a weighted sum of the input, or a weighted sum for short.
- Some also refer to the weighted sum as a dot product, as you'll see

Multiple inputs: What does this neural network do?

② Inserting one input datapoint



This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current number of toes
wlrec = current games won (percent)
nfans = fan count (in millions)

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

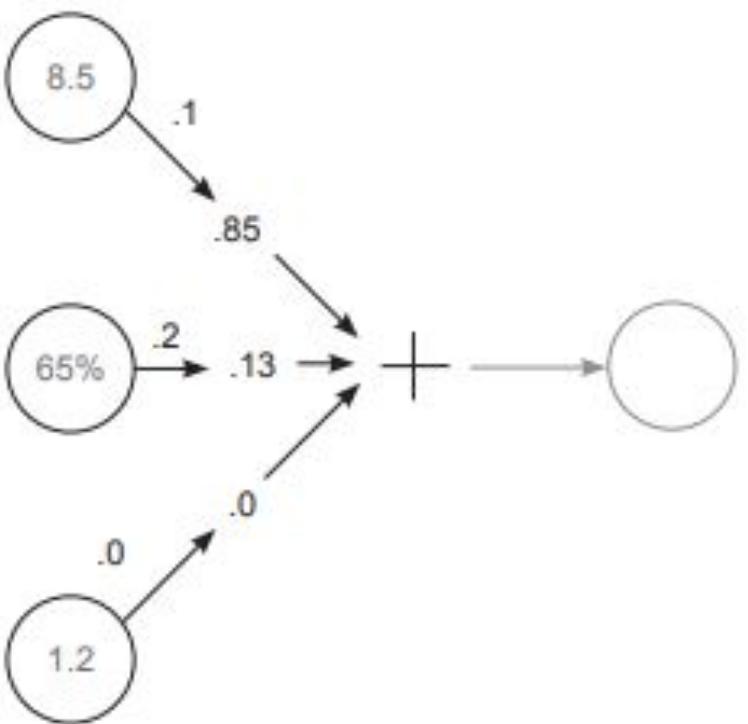
input = [toes[0], wlrec[0], nfans[0]]
pred = neural_network(input,weights)
```



Multiple inputs: What does this neural network do?

- Anytime you perform a mathematical operation between two vectors of equal length where you pair up values according to their position in the vector (again: position 0 with 0, 1 with 1, and so on), it's called an elementwise operation.
- Thus elementwise addition sums two vectors, and elementwise multiplication multiplies two vectors.

③ Performing a weighted sum of inputs



Inputs	Weights	Local predictions	
(8.50	* 0.1)	= 0.85	= toes prediction
(0.65	* 0.2)	= 0.13	= wlrec prediction
(1.20	* 0.0)	= 0.00	= fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

$$0.85 + 0.13 + 0.00 = 0.98$$

```
def w_sum(a,b):  
    assert(len(a) == len(b))  
  
    output = 0  
  
    for i in range(len(a)):  
        output += (a[i] * b[i])  
  
    return output
```

```
def neural_network(input, weights):  
    pred = w_sum(input,weights)  
  
    return pred
```

Multiple inputs: What does this neural network do?

- Loosely stated, a dot product gives you a notion of similarity between two vectors.
- Consider these examples:

```
a = [ 0, 1, 0, 1]  
b = [ 1, 0, 1, 0]  
c = [ 0, 1, 1, 0]  
d = [.5, 0, .5, 0]  
e = [ 0, 1, -1, 0]
```

w_sum(a,b) =	0
w_sum(b,c) =	1
w_sum(b,d) =	1
w_sum(c,c) =	2
w_sum(d,d) =	.5
w_sum(c,e) =	0

Multiple inputs: What does this neural network do?

- Sometimes you can equate the properties of the dot product to a logical AND. Consider a and b:

```
a = [ 0, 1, 0, 1]  
b = [ 1, 0, 1, 0]
```

- If you ask whether both a[0] AND b[0] have value, the answer is no. If you ask whether both a[1] AND b[1] have value, the answer is again no.
- Because this is always true for all four values, the final score equals 0. Each value fails the logical AND.

```
b = [ 1, 0, 1, 0]  
c = [ 0, 1, 1, 0]
```

Multiple inputs: What does this neural network do?

- b and c, however, have one column that shares value. It passes the logical AND because b[2] and c[2] have weight.
- This column (and only this column) causes the score to rise to 1.

```
c = [ 0, 1, 1, 0]
d = [.5, 0, .5, 0]
```

- Fortunately, neural networks are also able to model partial ANDing.
- In this case, c and d share the same column as b and c, but because d has only 0.5 weight there, the final score is only 0.5.
- We exploit this property when modeling probabilities in neural networks

```
d = [.5, 0, .5, 0]
e = [-1, 1, 0, 0]
```

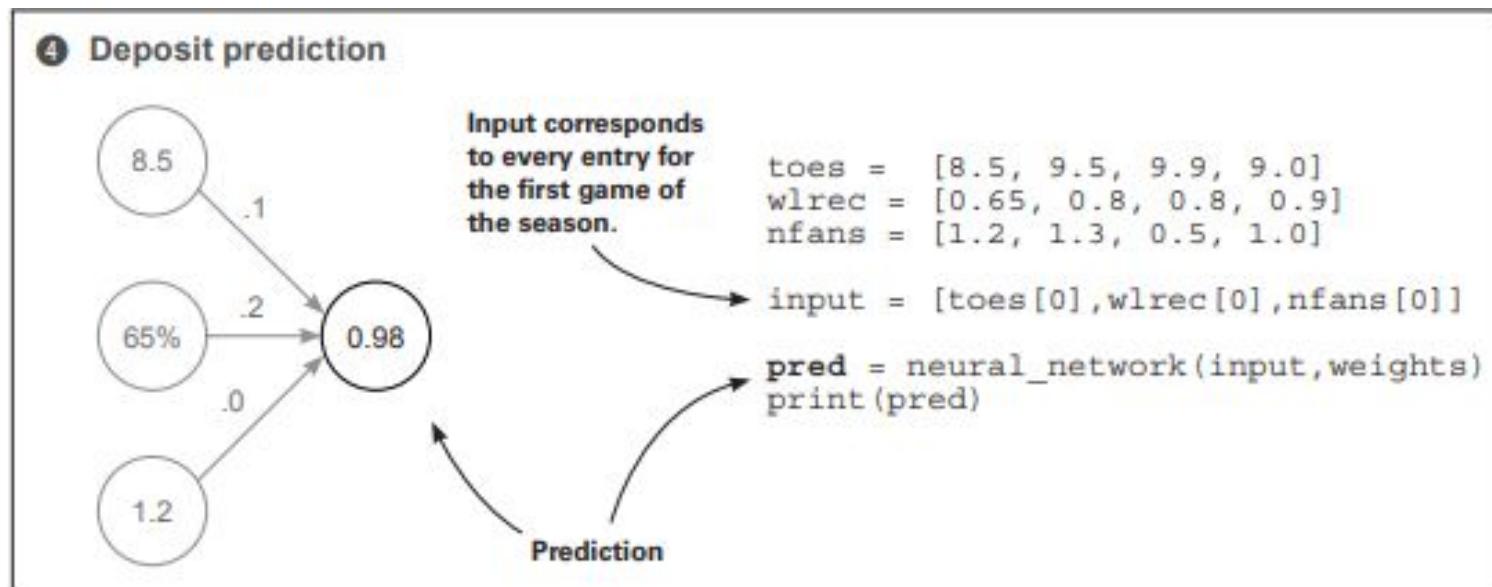
Multiple inputs: What does this neural network do?

- Amusingly, this gives us a kind of crude language for reading weights.
- Let's read a few examples, shall we? These assume you're performing `w_sum(input,weights)` and the “then” to these if statements is an abstract “then give high score”:

```
weights = [ 1, 0, 1] => if input[0] OR input[2]
weights = [ 0, 0, 1] => if input[2]
weights = [ 1, 0, -1] => if input[0] OR NOT input[2]
weights = [ -1, 0, -1] => if NOT input[0] OR NOT input[2]
weights = [ 0.5, 0, 1] => if BIG input[0] or input[2]
```

Multiple inputs: What does this neural network do?

- The most sensitive predictor is wlrec because its weight is 0.2.
- But the dominant force in the high score is the number of toes (ntoes), not because the weight is the highest, but because the input combined with the weight is by far the highest.



Multiple inputs: Complete runnable code

- The code snippets from this example come together in the following code, which creates and executes a neural network.

Previous code

```
def w_sum(a,b):  
    assert(len(a) == len(b))  
    output = 0  
    for i in range(len(a)):  
        output += (a[i] * b[i])  
    return output  
  
weights = [0.1, 0.2, 0]  
def neural_network(input, weights):  
    pred = w_sum(input,weights)  
    return pred  
  
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]  
  
input = [toes[0],wlrec[0],nfans[0]]  
pred = neural_network(input,weights)  
print(pred)
```

Input corresponds to every entry for the first game of the season.

Multiple inputs: Complete runnable code

NumPy code

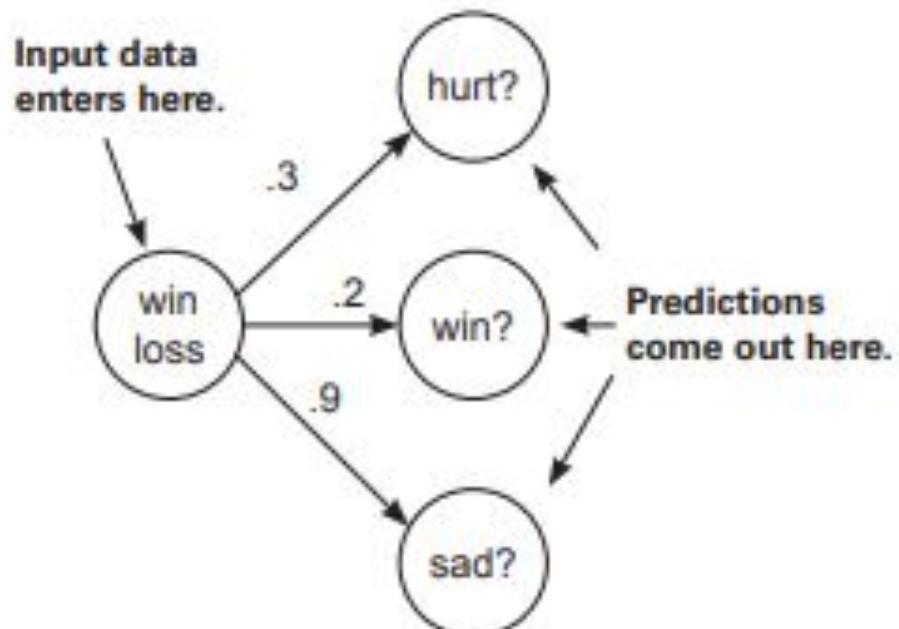
```
import numpy as np  
  
weights = np.array([0.1, 0.2, 0])  
  
def neural_network(input, weights):  
    pred = input.dot(weights)  
  
    return pred  
  
toes = np.array([8.5, 9.5, 9.9, 9.0])  
wlrec = np.array([0.65, 0.8, 0.8, 0.9])  
nfans = np.array([1.2, 1.3, 0.5, 1.0])  
  
input = np.array([toes[0], wlrec[0], nfans[0]])  
pred = neural_network(input, weights)  
print(pred)
```

Input corresponds to every entry for the first game of the season.

Making a prediction with multiple outputs

Neural networks can also make multiple predictions using only a single input.

① An empty network with multiple outputs



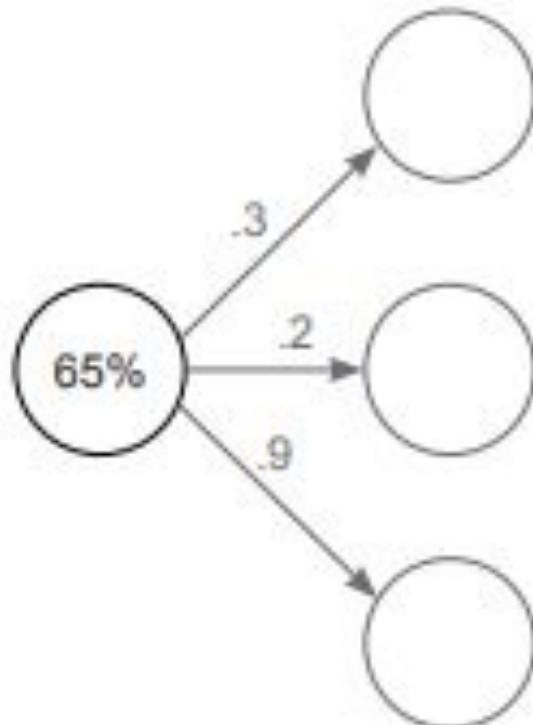
Instead of predicting just whether the team won or lost, you're also predicting whether the players are happy or sad and the percentage of team members who are hurt. You make this prediction using only the current win/loss record.

```
weights = [0.3, 0.2, 0.9]
```

```
def neural_network(input, weights):  
    pred = ele_mul(input, weights)  
    return pred
```

Making a prediction with multiple outputs

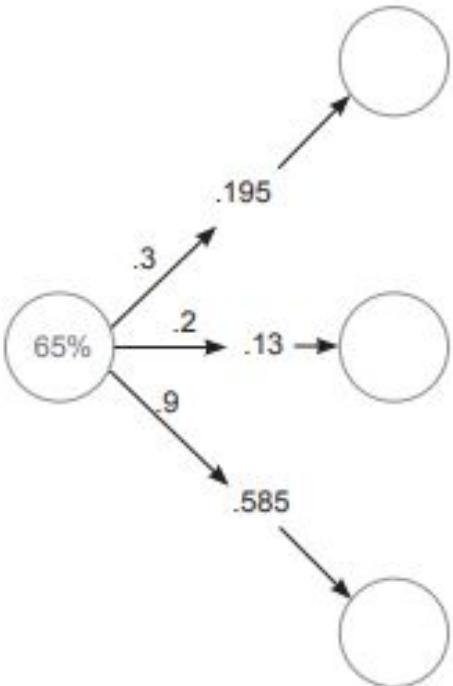
② Inserting one input datapoint



```
wlrec = [0.65, 0.8, 0.8, 0.9]  
input = wlrec[0]  
pred = neural_network(input,weights)
```

Making a prediction with multiple outputs

③ Performing elementwise multiplication



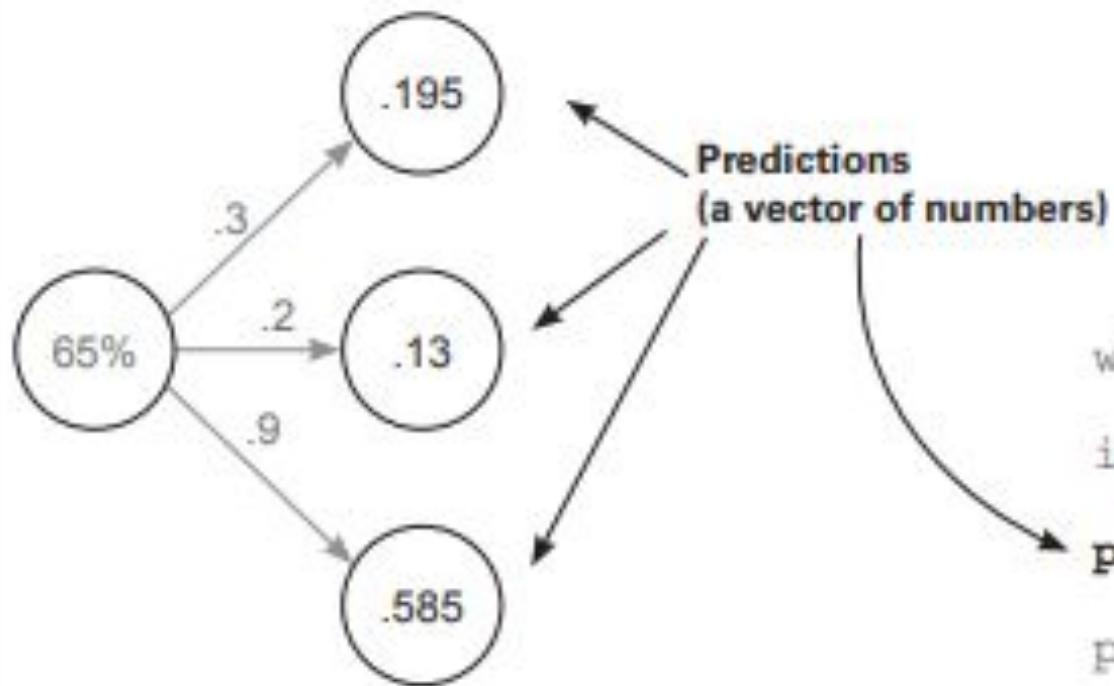
```
def ele_mul(number, vector):  
    output = [0, 0, 0]  
    assert(len(output) == len(vector))  
    for i in range(len(vector)):  
        output[i] = number * vector[i]  
    return output
```

```
def neural_network(input, weights):  
    pred = ele_mul(input, weights)  
    return pred
```

Inputs	Weights	Final predictions	
(0.65	* 0.3)	= 0.195	= hurt prediction
(0.65	* 0.2)	= 0.13	= win prediction
(0.65	* 0.9)	= 0.585	= sad prediction

Making a prediction with multiple outputs

④ Depositing predictions



```
wlrec = [0.65, 0.8, 0.8, 0.9]
```

```
input = wlrec[0]
```

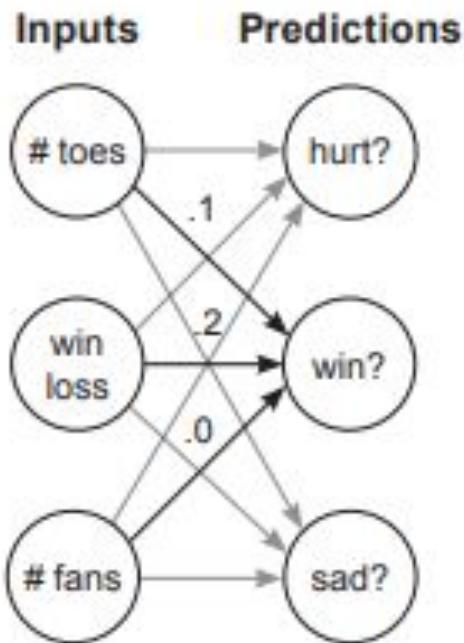
```
pred = neural_network(input, weight)
```

```
print(pred)
```

Predicting with multiple inputs and outputs

Neural networks can predict multiple outputs given multiple inputs

1 An empty network with multiple inputs and outputs



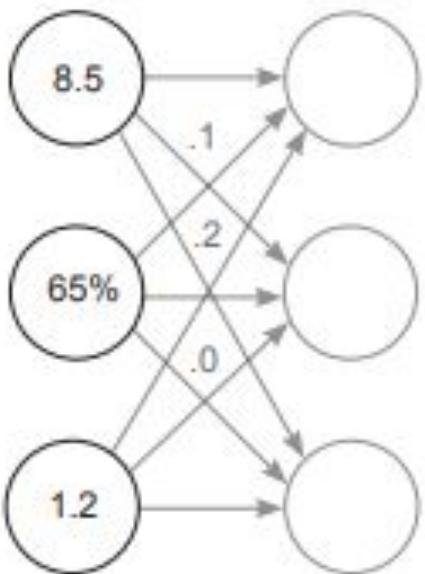
```
# toes % win # fans
weights = [ [0.1, 0.1, -0.3], # hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ] # sad?

def neural_network(input, weights):
    pred = vect_mat_mul(input, weights)
    return pred
```

Predicting with multiple inputs and outputs

② Inserting one input datapoint

Inputs Predictions



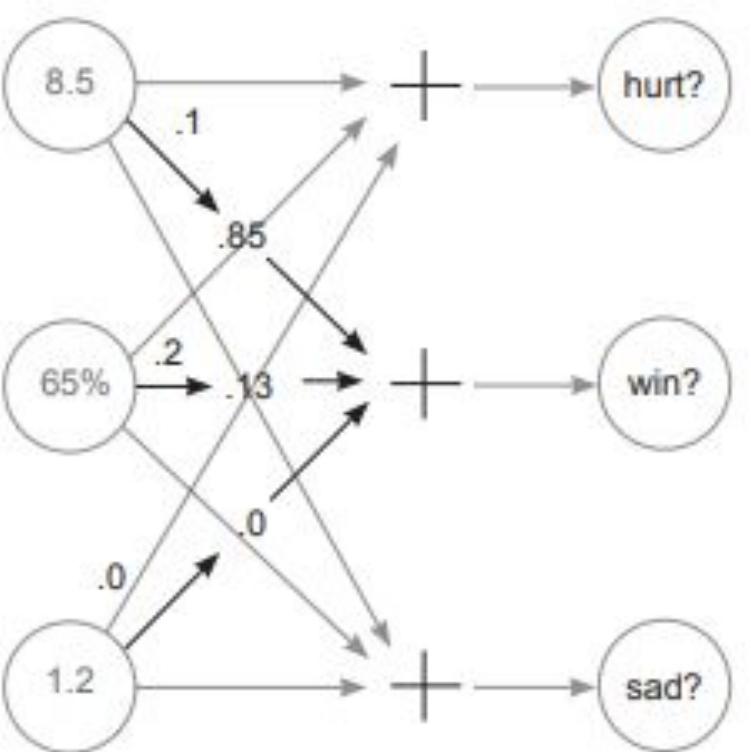
This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions)

```
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]
```

```
input = [toes[0], wlrec[0], nfans[0]]  
pred = neural_network(input, weights)
```

Input corresponds to every entry for the first game of the season.

③ For each output, performing a weighted sum of inputs



```
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output
```

```
def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]

    for i in range(len(vect)):
        output[i]=w_sum(vect,matrix[i])

    return output
```

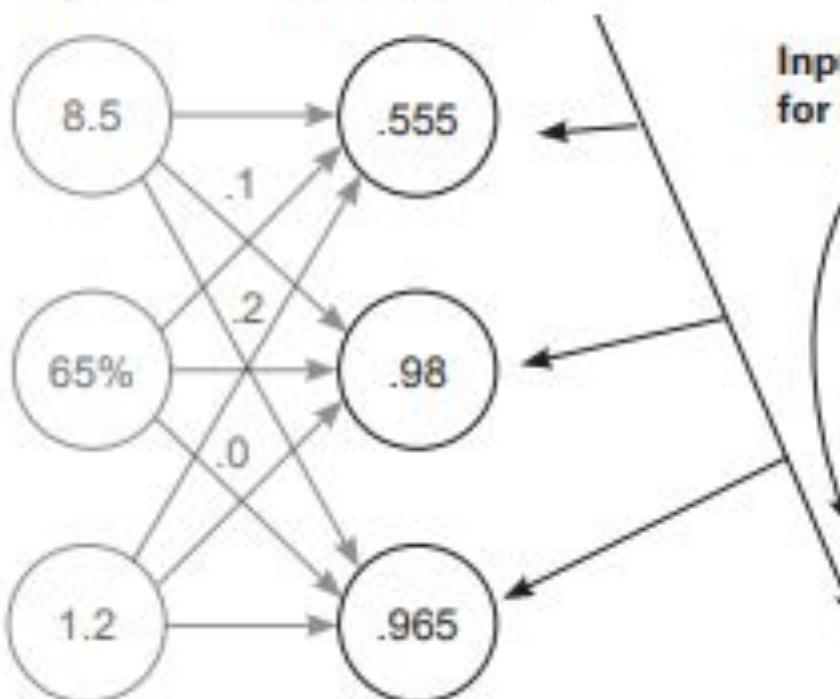
```
def neural_network(input, weights):
    pred=vect_mat_mul(input,weights)
    return pred
```

# toes	% win	# fans	
(8.5 * 0.1)	+ (0.65 * 0.1)	+ (1.2 * -0.3)	= 0.555 = hurt prediction
(8.5 * 0.1)	+ (0.65 * 0.2)	+ (1.2 * 0.0)	= 0.98 = win prediction
(8.5 * 0.0)	+ (0.65 * 1.3)	+ (1.2 * 0.1)	= 0.965 = sad prediction

Predicting with multiple inputs and outputs

④ Depositing predictions

Inputs Predictions



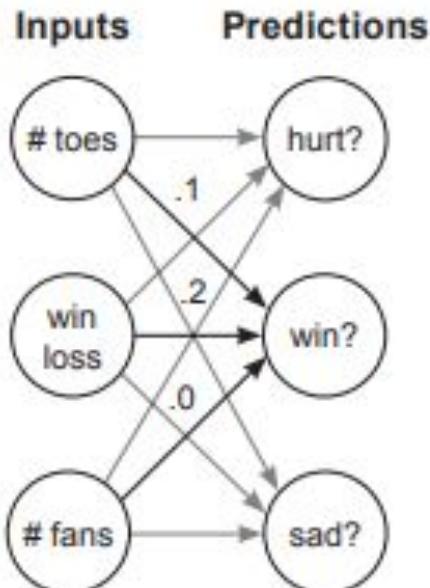
Input corresponds to every entry
for the first game of the season.

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]
input = [toes[0], wlrec[0], nfans[0]]
pred = neural_network(input, weight)
```

Multiple inputs and outputs: How does it work?

It performs three independent weighted sums of the input to make three predictions.

① An empty network with multiple inputs and outputs

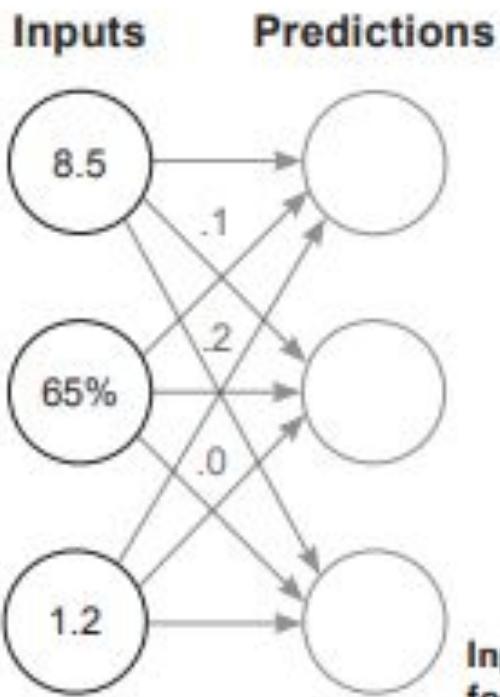


```
# toes % win # fans  
weights = [ [0.1, 0.1, -0.3], # hurt?  
           [0.1, 0.2, 0.0], # win?  
           [0.0, 1.3, 0.1] ] # sad?
```

```
def neural_network(input, weights):  
  
    pred=vect_mat_mul(input,weights)  
  
    return pred
```

Multiple inputs and outputs: How does it work?

② Inserting one input datapoint



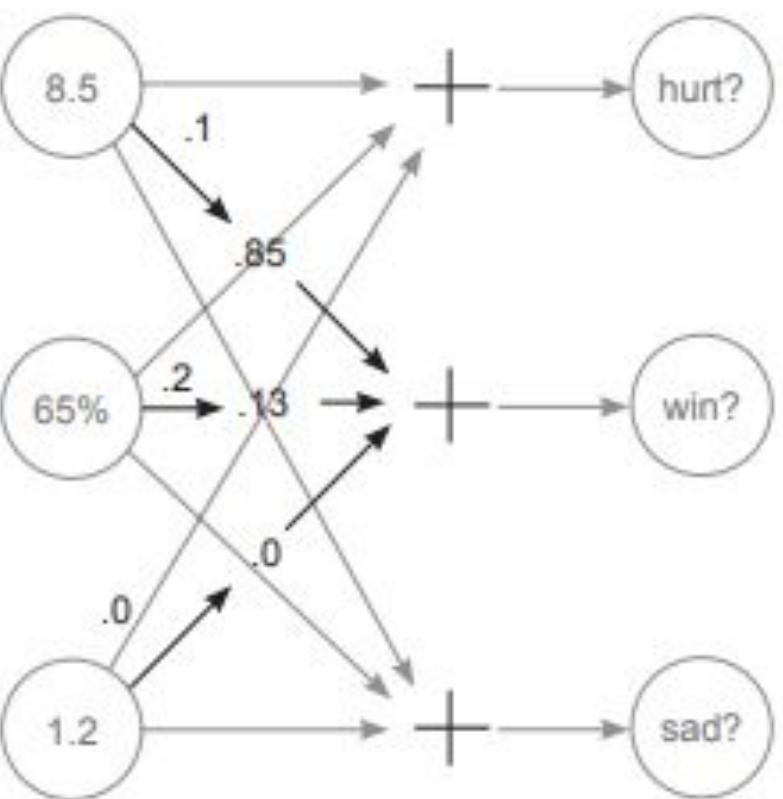
This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions)

```
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]
```

```
input = [toes[0], wlrec[0], nfans[0]]  
pred = neural_network(input, weights)
```

Input corresponds to every entry for the first game of the season.

③ For each output, performing a weighted sum of inputs



```
def w_sum(a,b):  
    assert(len(a) == len(b))  
    output = 0  
    for i in range(len(a)):  
        output += (a[i] * b[i])  
    return output
```

```
def vect_mat_mul(vect,matrix):  
    assert(len(vect) == len(matrix))  
    output = [0,0,0]  
  
    for i in range(len(vect)):  
        output[i]=w_sum(vect,matrix[i])  
  
    return output
```

```
def neural_network(input, weights):  
    pred = vect_mat_mul(input,weights)  
    return pred
```

toes

% win

fans

$$(8.5 \cdot 0.1) + (0.65 \cdot 0.1) + (1.2 \cdot -0.3) = 0.555 = \text{hurt prediction}$$

$$(8.5 \cdot 0.1) + (0.65 \cdot 0.2) + (1.2 \cdot 0.0) = 0.98 = \text{win prediction}$$

$$(8.5 \cdot 0.0) + (0.65 \cdot 1.3) + (1.2 \cdot 0.1) = 0.965 = \text{sad prediction}$$

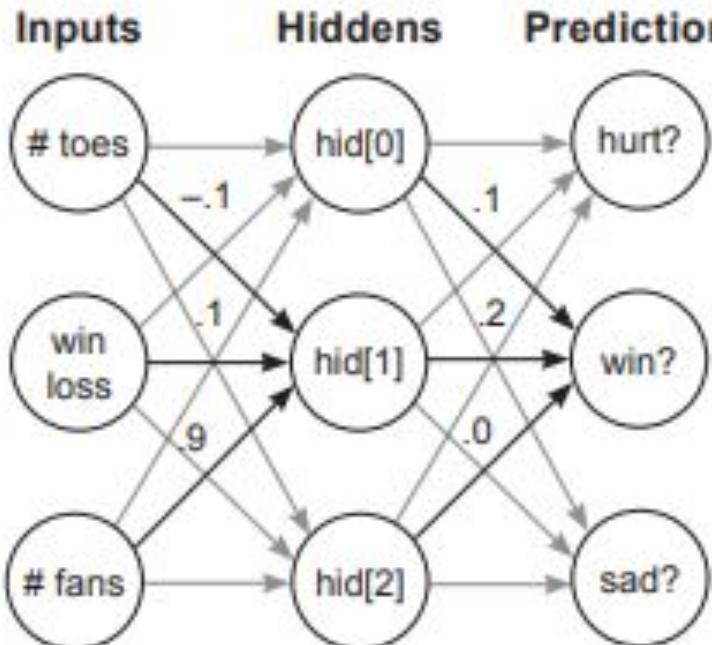
Multiple inputs and outputs: How does it work?

- I want to use this list of vectors and series of weighted sums logic to introduce two new concepts.
- See the weights variable in step 1? It's a list of vectors.
- A list of vectors is called a matrix.
- It's as simple as it sounds. Commonly used functions use matrices.
- One of these is called vector-matrix multiplication.

Predicting on predictions

Neural networks can be stacked!

① An empty network with multiple inputs and outputs



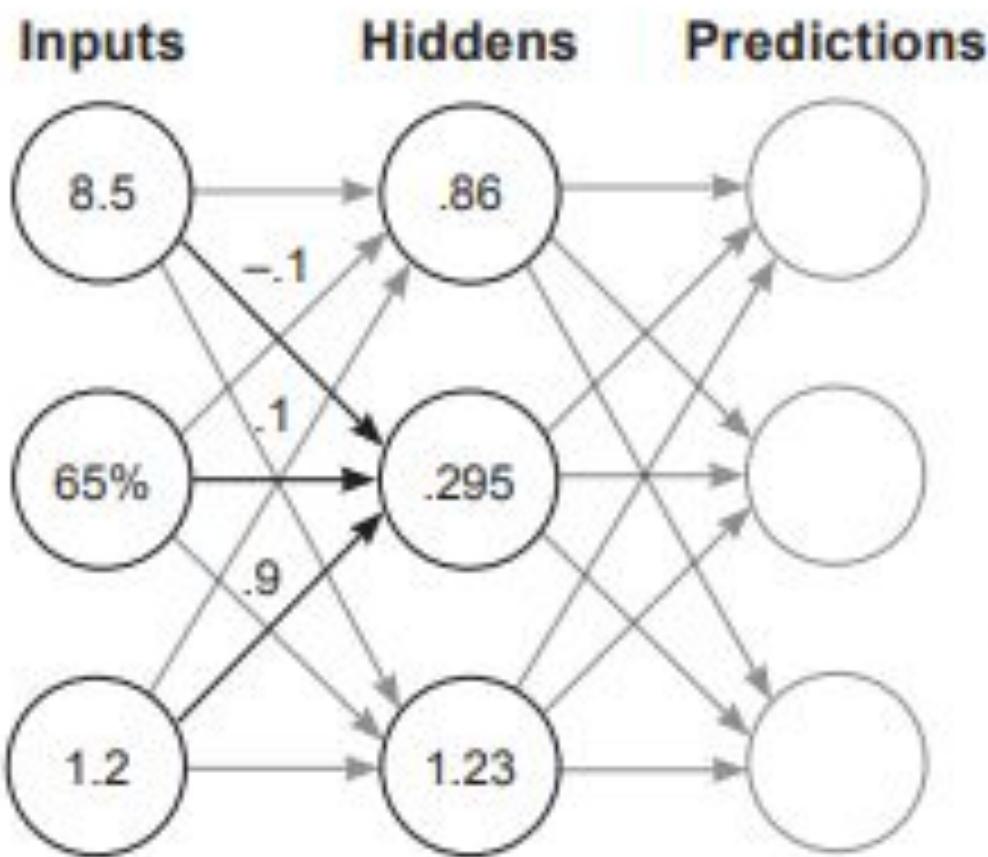
```
# toes % win # fans
ih_wgt = [ [0.1, 0.2, -0.1], # hid[0]
           [-0.1, 0.1, 0.9], # hid[1]
           [0.1, 0.4, 0.1] ] # hid[2]

# hid[0] hid[1] hid[2]
hp_wgt = [ [0.3, 1.1, -0.3], # hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ] # sad?

weights = [ih_wgt, hp_wgt]

def neural_network(input, weights):
    hid = vect_mat_mul(input, weights[0])
    pred = vect_mat_mul(hid, weights[1])
    return pred
```

② Predicting the hidden layer



**Input corresponds to every entry
for the first game of the season.**

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0], wlrec[0], nfans[0]]

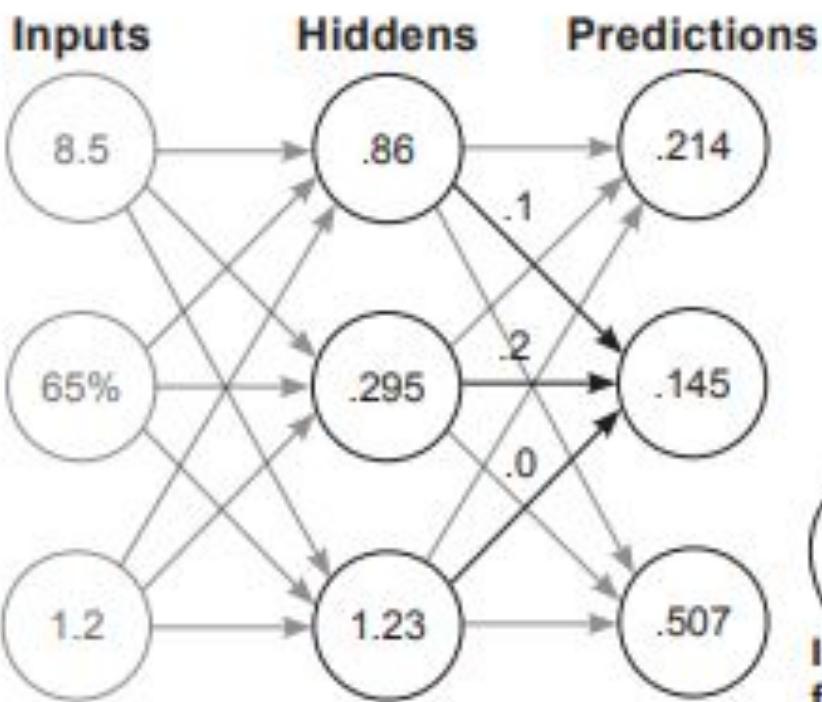
pred = neural_network(input, weights)

def neural_network(input, weights):

    hid = vect_mat_mul(input, weights[0])
    pred = vect_mat_mul(hid, weights[1])
    return pred
```

Predicting on predictions

③ Predicting the output layer (and depositing the prediction)



```
def neural_network(input, weights):  
    hid=vect_mat_mul(input,weights[0])  
    pred = vect_mat_mul(hid,weights[1])  
    return pred  
  
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]  
  
input = [toes[0],wlrec[0],nfans[0]]  
  
pred = neural_network(input,weights)  
print(pred)
```

Input corresponds to every entry
for the first game of the season.



```
import numpy as np

# toes % win # fans
ih_wgt = np.array([
    [0.1, 0.2, -0.1], # hid[0]
    [-0.1, 0.1, 0.9], # hid[1]
    [0.1, 0.4, 0.1]]).T # hid[2]

# hid[0] hid[1] hid[2]
hp_wgt = np.array([
    [0.3, 1.1, -0.3], # hurt?
    [0.1, 0.2, 0.0], # win?
    [0.0, 1.3, 0.1]]).T # sad?

weights = [ih_wgt, hp_wgt]

def neural_network(input, weights):

    hid = input.dot(weights[0])
    pred = hid.dot(weights[1])
    return pred

toes = np.array([8.5, 9.5, 9.9, 9.0])
wlrec = np.array([0.65, 0.8, 0.8, 0.9])
nfans = np.array([1.2, 1.3, 0.5, 1.0])

input = np.array([toes[0], wlrec[0], nfans[0]])

pred = neural_network(input, weights)
print(pred)
```

NumPy version



A quick primer on NumPy

NumPy does a few things for you. Let's reveal the magic.

- You've also learned about different operations that occur on vectors and matrices, including dot products, elementwise multiplication and addition, and vector-matrix multiplication.
- For these operations, you've written Python functions that can operate on simple Python list objects.



A quick primer on NumPy



- You can create vectors and matrices in multiple ways in NumPy.
- Most of the common techniques for neural networks are listed in the previous code.

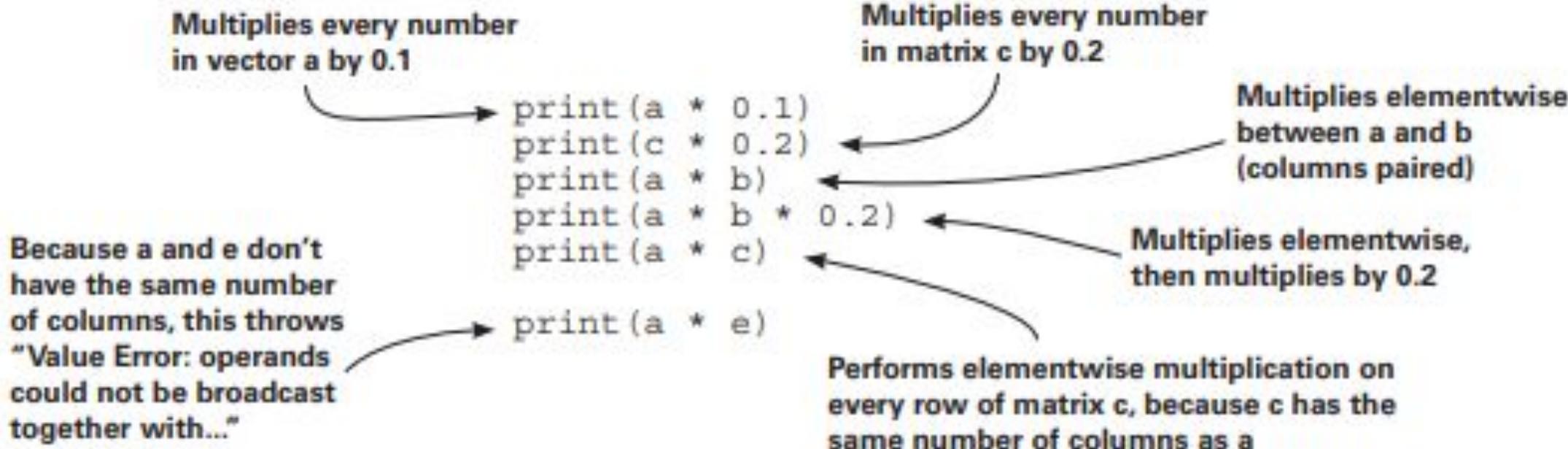
A quick primer on NumPy

```
import numpy as np  
  
a = np.array([0,1,2,3])           ← A vector  
b = np.array([4,5,6,7])           ← Another vector  
c = np.array([[0,1,2,3],  
             [4,5,6,7]])          ← A matrix  
  
d = np.zeros((2,4))              ← 2 × 4 matrix  
e = np.random.rand(2,5)           ← of zeros  
  
print(a)  
print(b)  
print(c)  
print(d)  
print(e)
```

Output

```
[0 1 2 3]  
[4 5 6 7]  
[[0 1 2 3]  
 [4 5 6 7]]  
[[ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]]  
[[ 0.22717119  0.39712632  
  0.0627734   0.08431724  
  0.53469141]  
 [ 0.09675954  0.99012254  
  0.45922775  0.3273326  
  0.28617742]]
```

A quick primer on NumPy





A quick primer on NumPy

- The general rule of thumb for anything elementwise (+, -, *, /) is that either the two variables must have the same number of columns, or one of the variables must have only one column.
 - For example, `print(a * 0.1)` multiplies a vector by a single number (a scalar). NumPy says, “Oh, I bet I’m supposed to do vector-scalar multiplication here,” and then multiples the scalar (0.1) by every value in the vector.
 - This looks exactly the same as `print(c * 0.2)`, except NumPy knows that `c` is a matrix.
- 



A quick primer on NumPy



- Again, the most confusing part is that all of these operations look the same if you don't know which variables are scalars, vectors, or matrices.
- When you “read NumPy,” you’re really doing two things: reading the operations and keeping track of the shape (number of rows and columns) of each operation.



A quick primer on NumPy



- Let's look at a few examples of matrix multiplication in NumPy, noting the input and output shapes of each matrix.

```
a = np.zeros((1,4))
b = np.zeros((4,3))

c = a.dot(b)           ← Vector of length 4
print(c.shape)         ← Matrix with
                        4 rows and
                        3 columns
```

Output

(1, 3)

```
a = np.zeros( (2, 4)) ← Matrix with 2 rows  
b = np.zeros( (4, 3)) ← and 4 columns  
  
c = a.dot(b) ← Matrix with 4 rows  
print(c.shape) ← and 3 columns Outputs (2,3)  
  
e = np.zeros( (2, 1)) ← Matrix with 2 rows and 1 column  
f = np.zeros( (1, 3)) ← Matrix with 1 row and 3 columns  
  
g = e.dot(f)  
print(g.shape) ← Outputs (2,3)  
← Throws an error; .T flips the rows and columns of a matrix.  
  
h = np.zeros( (5, 4)).T ← Matrix with 4 rows and 5 columns  
i = np.zeros( (5, 6)) ← Matrix with 6 rows and 5 columns  
  
j = h.dot(i)  
print(j.shape) ← Outputs (4,6)  
← Matrix with 5 rows and 4 columns  
  
h = np.zeros( (5, 4)) ← Matrix with 5 rows and 4 columns  
i = np.zeros( (5, 6)) ← Matrix with 5 rows and 6 columns  
j = h.dot(i)  
print(j.shape) ← Throws an error
```



Summary



To predict, neural networks perform repeated weighted sums of the input.

- You've seen an increasingly complex variety of neural networks in this lesson.
 - I hope it's clear that a relatively small number of simple rules are used repeatedly to create larger, more advanced neural networks.
 - The network's intelligence depends on the weight values you give it.
- 

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



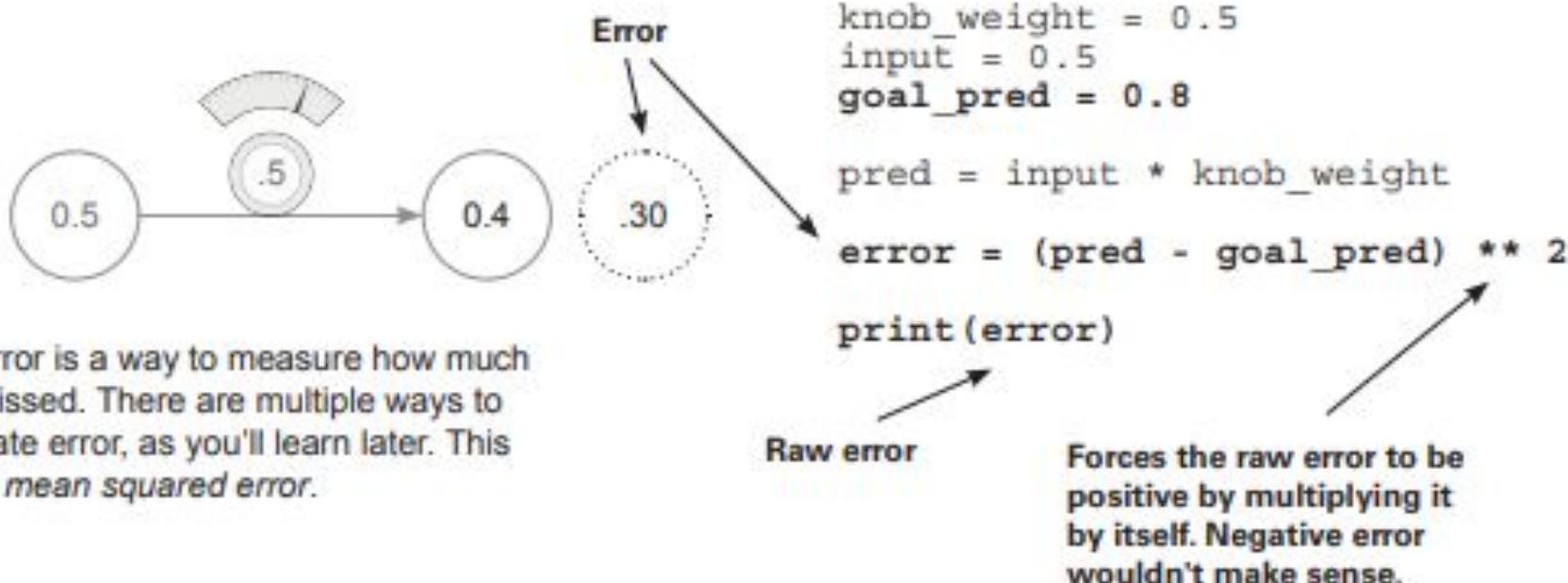
Learning tells each weight how it can change to reduce the error.

- 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?

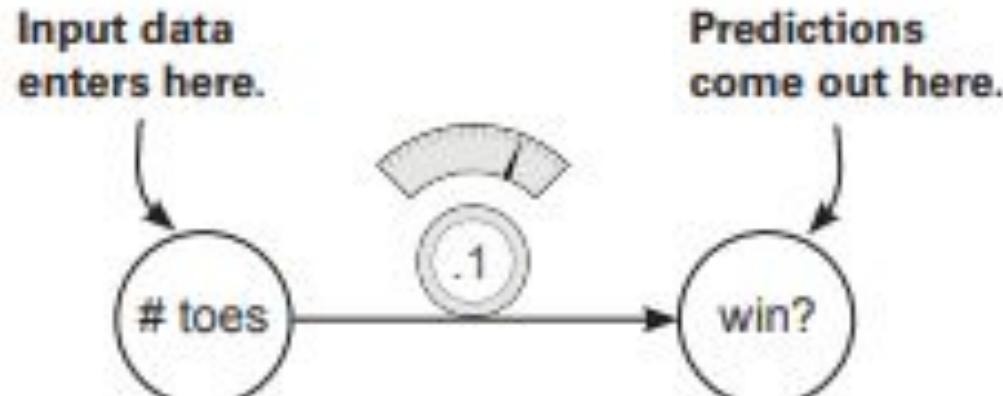


Learning using the hot and cold method.

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

What's the simplest form of neural learning?

1 An empty network



```
weight = 0.1  
lr = 0.01  
def neural_network(input, weight):  
    prediction = input * weight  
    return prediction
```

What's the simplest form of neural learning?

② 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*.

```
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)

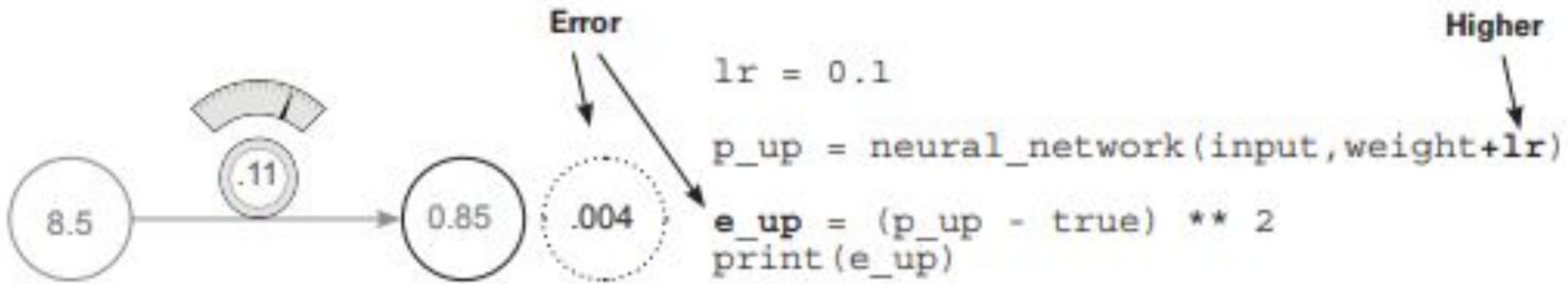
error = (pred - true) ** 2
print(error)
```

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

What's the simplest form of neural learning?

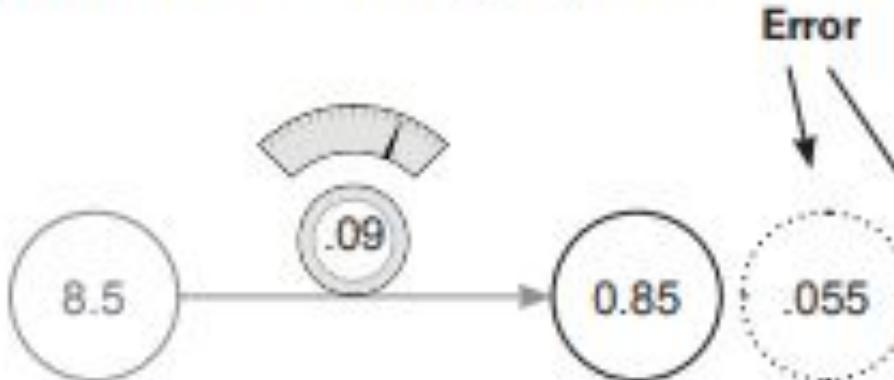
③ COMPARE: Making a prediction with a higher weight and evaluating error

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.



What's the simplest form of neural learning?

④ COMPARE: Making a prediction with a lower weight and evaluating error



lr = 0.01

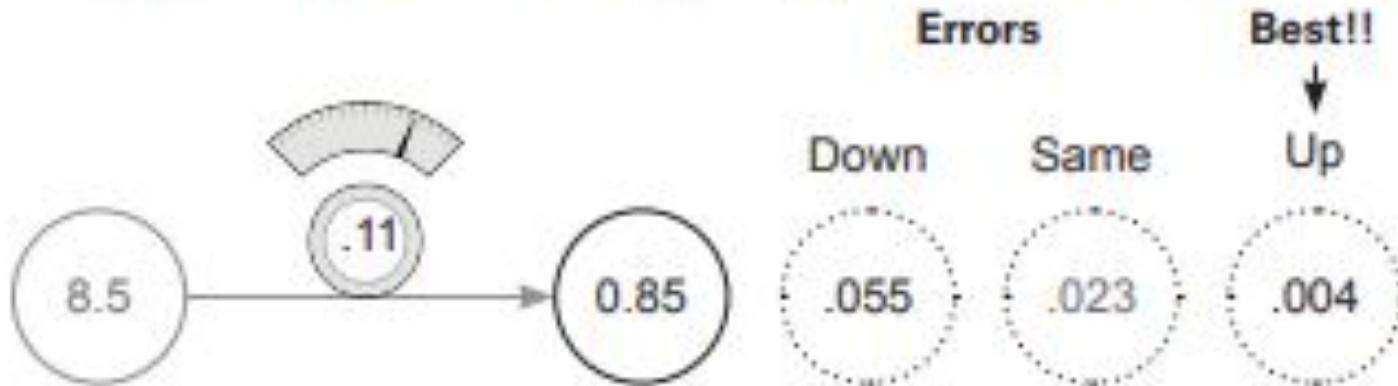
p_dn = neural_network(input, weight-lr)

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

Lower
↓

What's the simplest form of neural learning?

⑤ 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_dn):  
        weight += lr
```



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
```

```
step_amount = 0.001
```

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

How much to move
the weights each
iteration

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

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:2.5000000033e-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 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 $n * \text{step_amount}$ 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

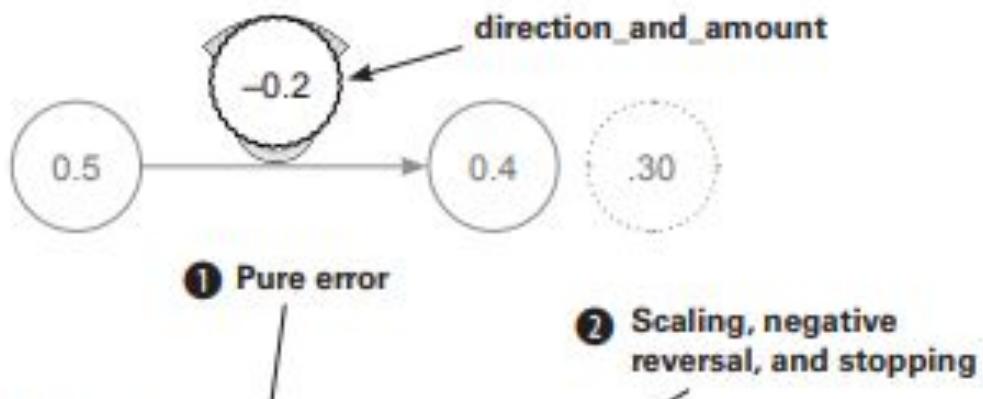
Let's measure the error and find the direction and amount!

- 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  
...
```

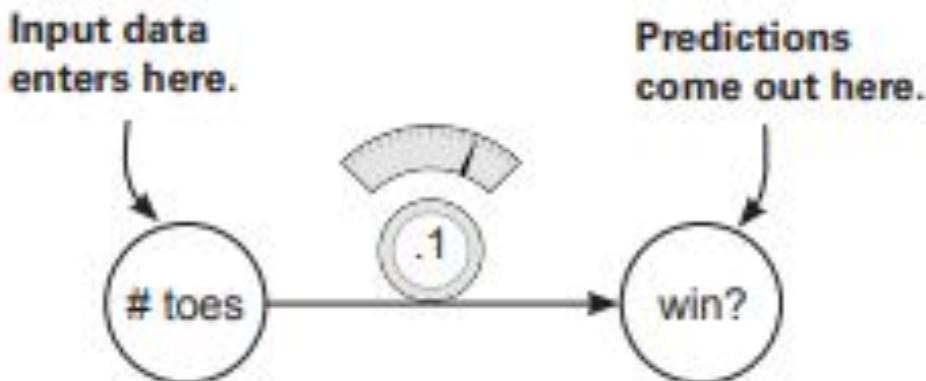
```
Error:1.7092608064e-05 Prediction:0.79586567925  
Error:9.61459203602e-06 Prediction:0.796899259437  
Error:5.40820802026e-06 Prediction:0.797674444578
```

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.

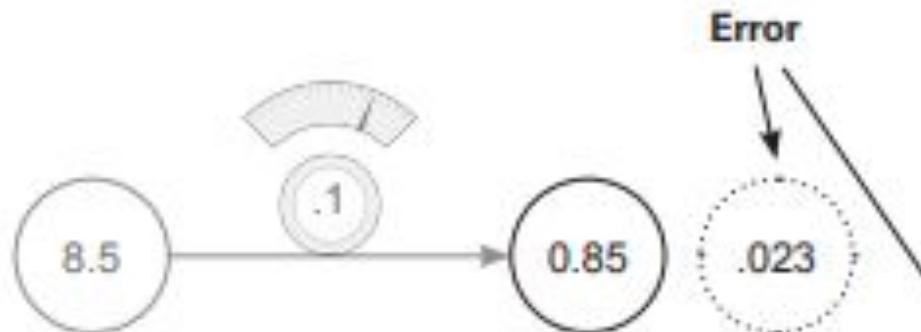
① An empty network



```
weight = 0.1
alpha = 0.01
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

One iteration of gradient descent

② 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*.

```
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)

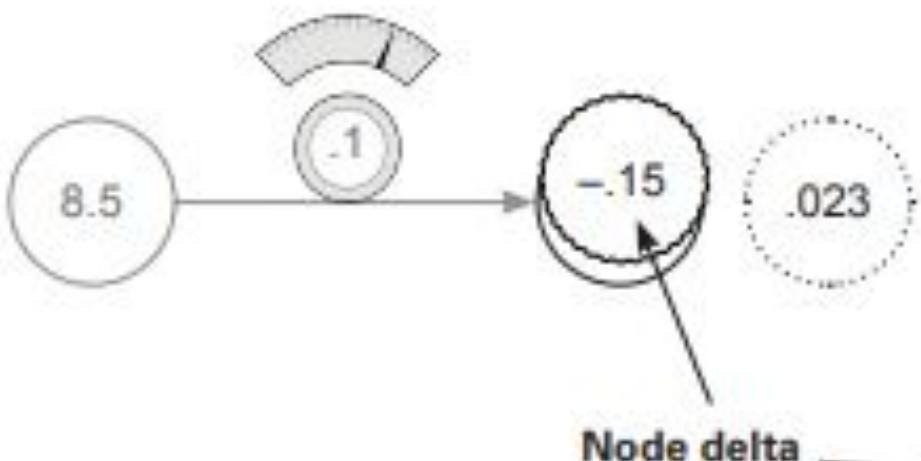
error = (pred - goal_pred) ** 2
```

Raw error

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

One iteration of gradient descent

③ 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
```



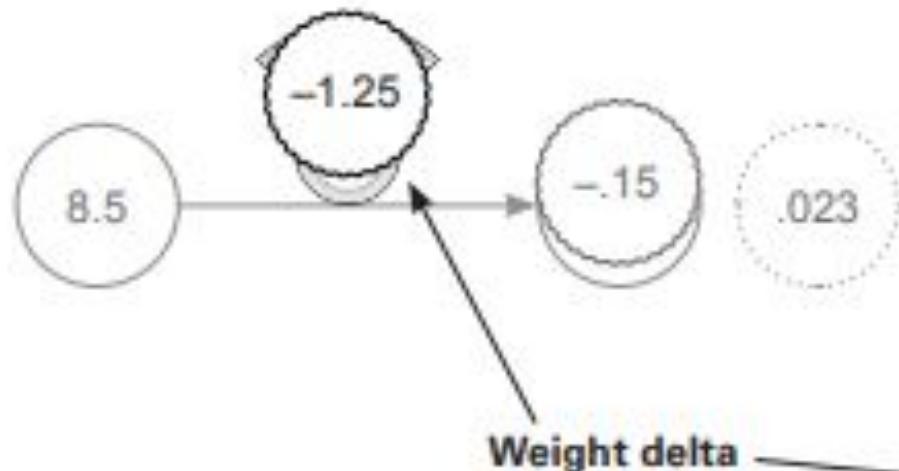
One iteration of gradient descent



- 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)

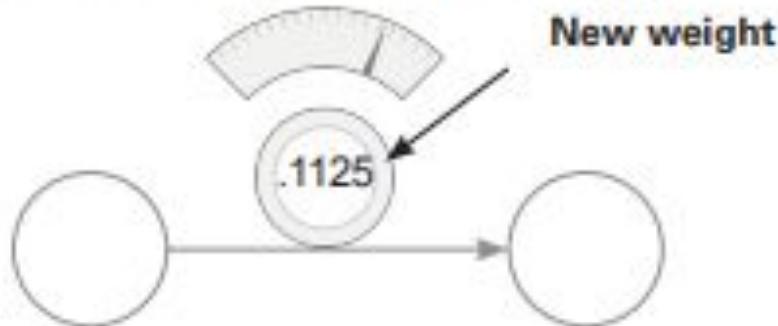
error = (pred - goal_pred) ** 2

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
```



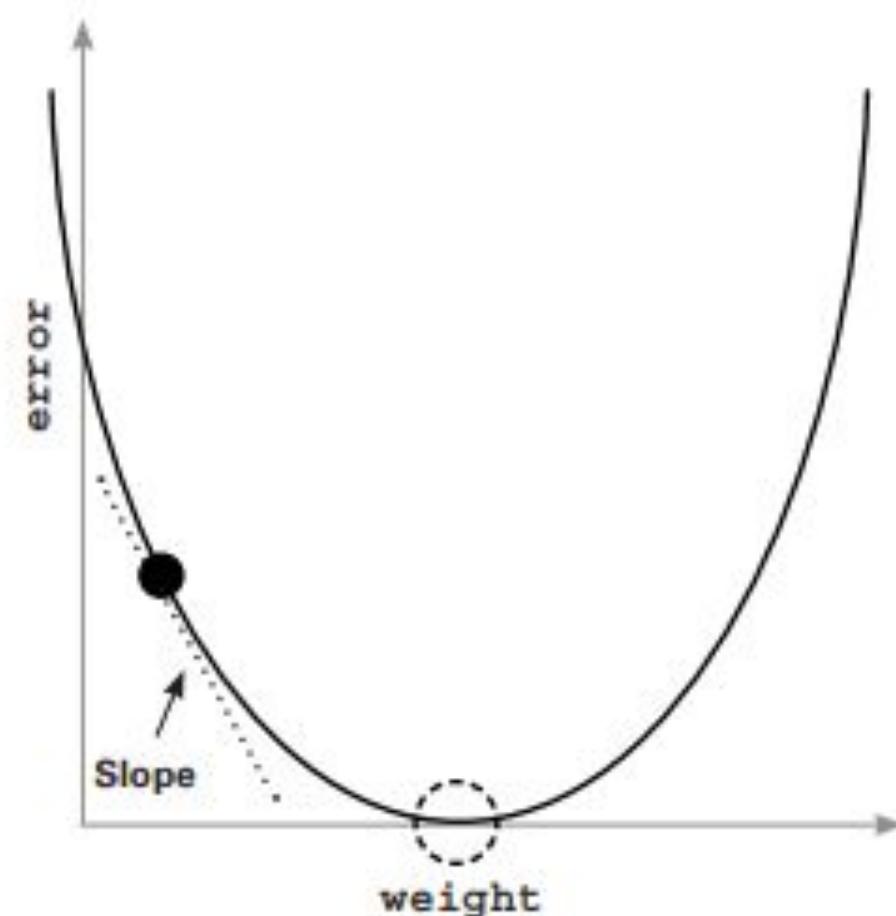
Learning is just reducing error

- 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?

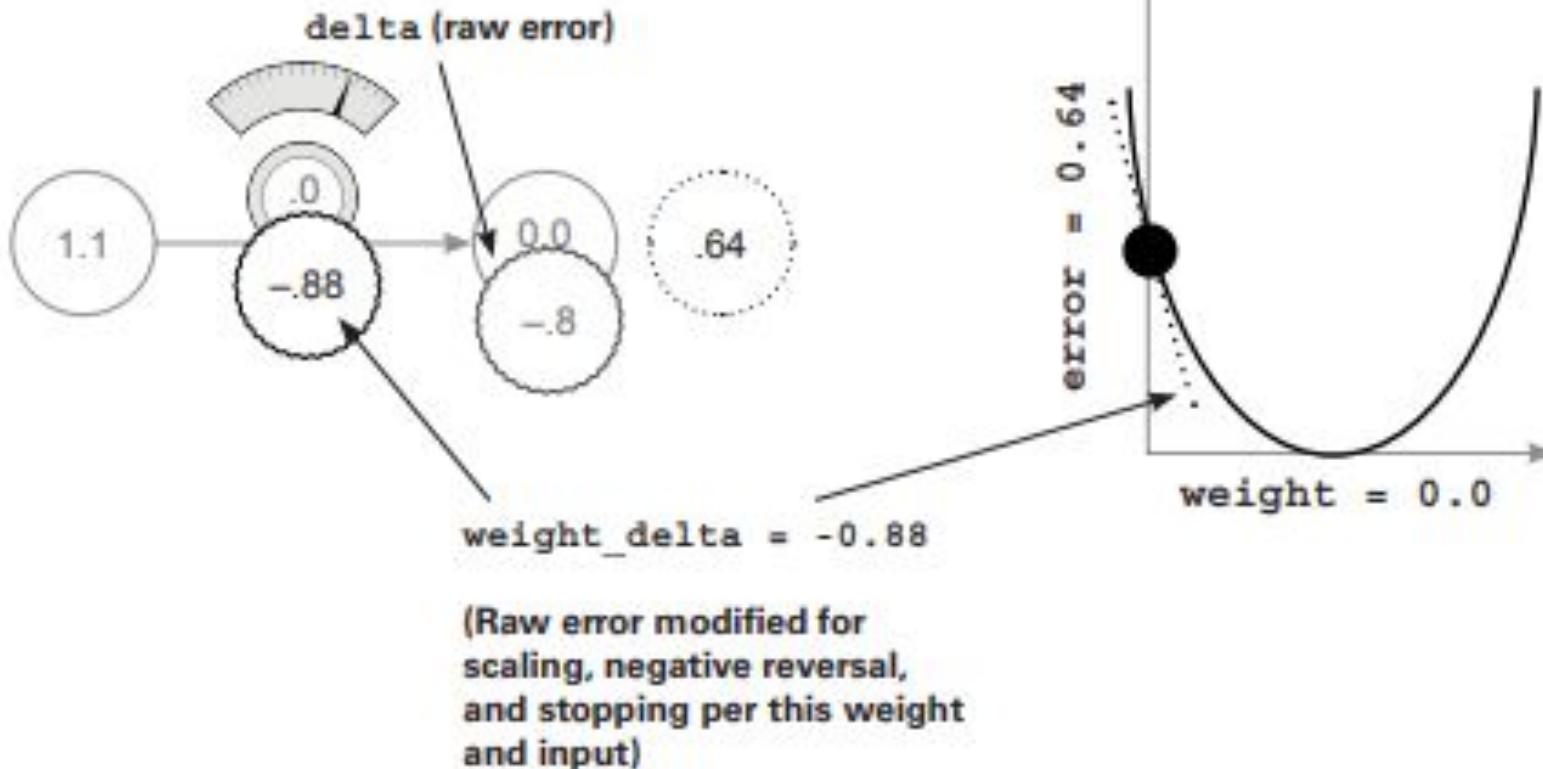
```
weight, goal_pred, input = (0.0, 0.8, 1.1)

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))
```



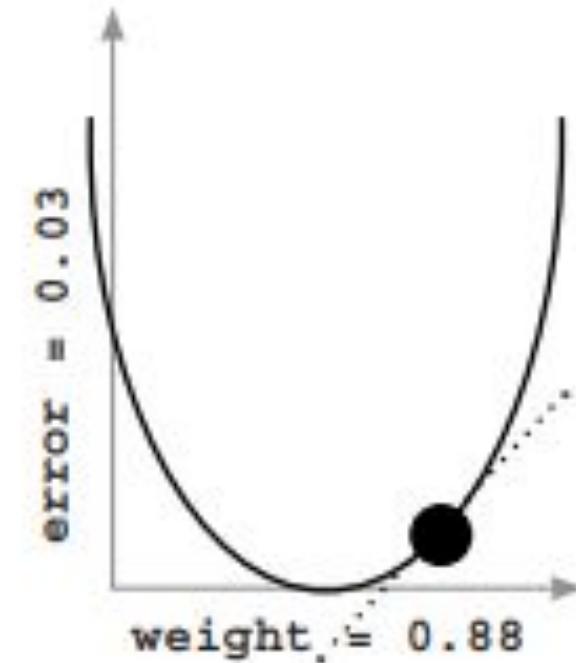
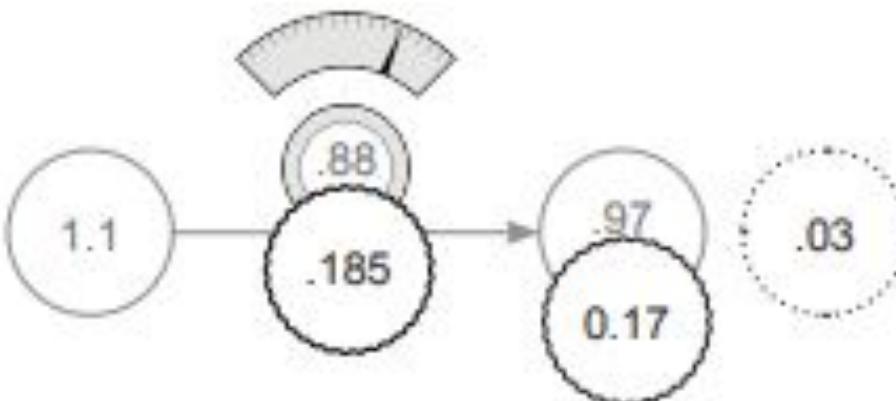
Let's watch several steps of learning

① A big weight increase



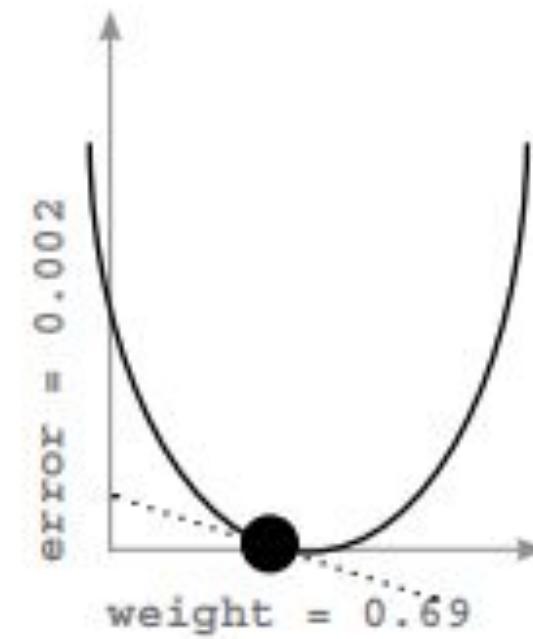
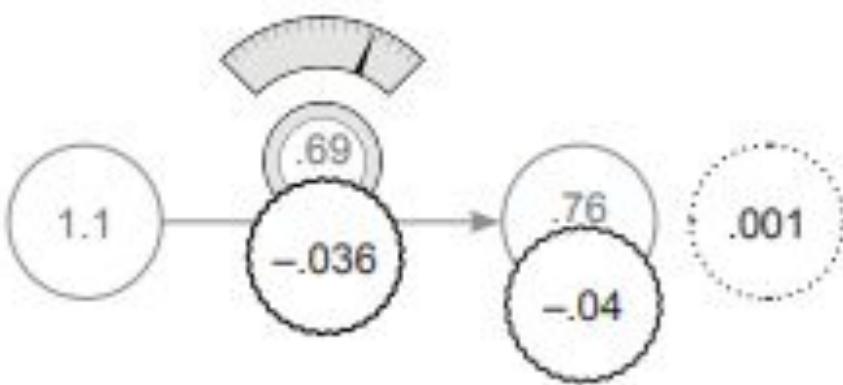
Let's watch several steps of learning

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

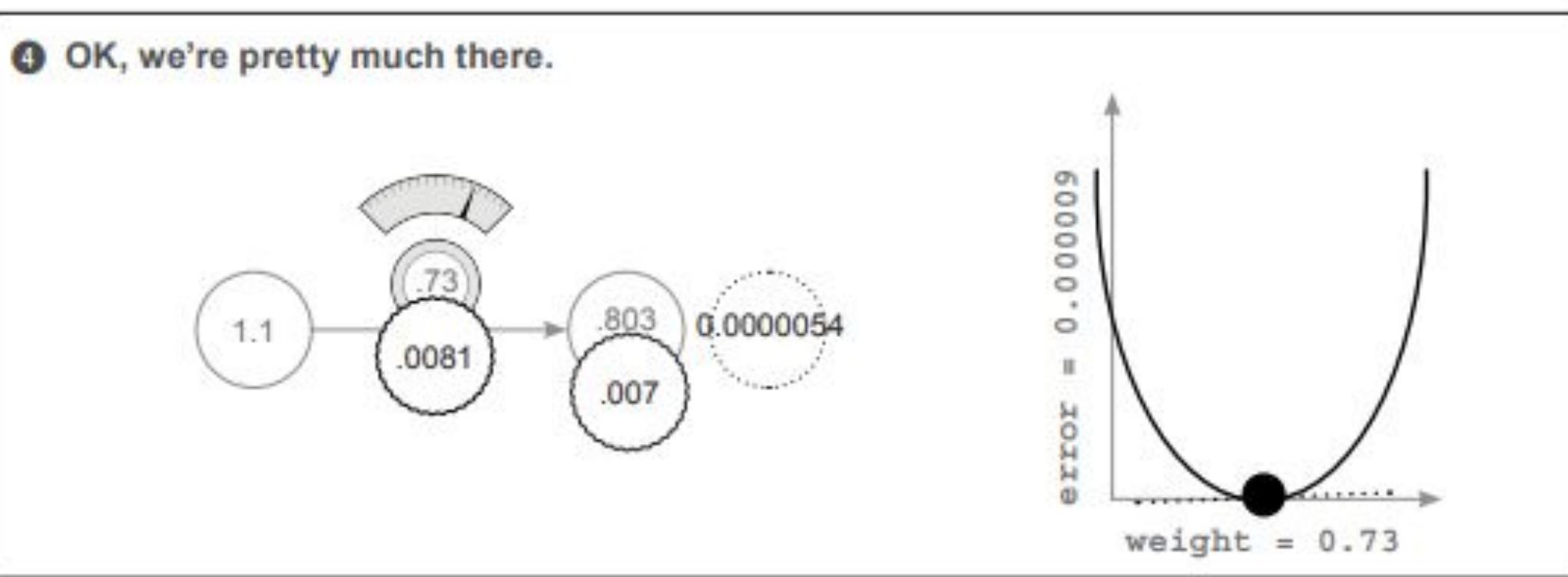


Let's watch several steps of learning

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



Let's watch several steps of learning



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:

$$\text{error} = ((\text{input} * \text{weight}) - \text{goal_pred}) ^\star 2$$



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:

red_length = 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:



```
red_length = 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:
 $\text{error} = ((\text{input} * \text{weight}) - \text{goal_pred}) ^\star 2$
- Remember, `goal_pred` and `input` are fixed, so you can rewrite this function:
 $\text{error} = ((0.5 * \text{weight}) - 0.8) ^\star 2$
- 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.



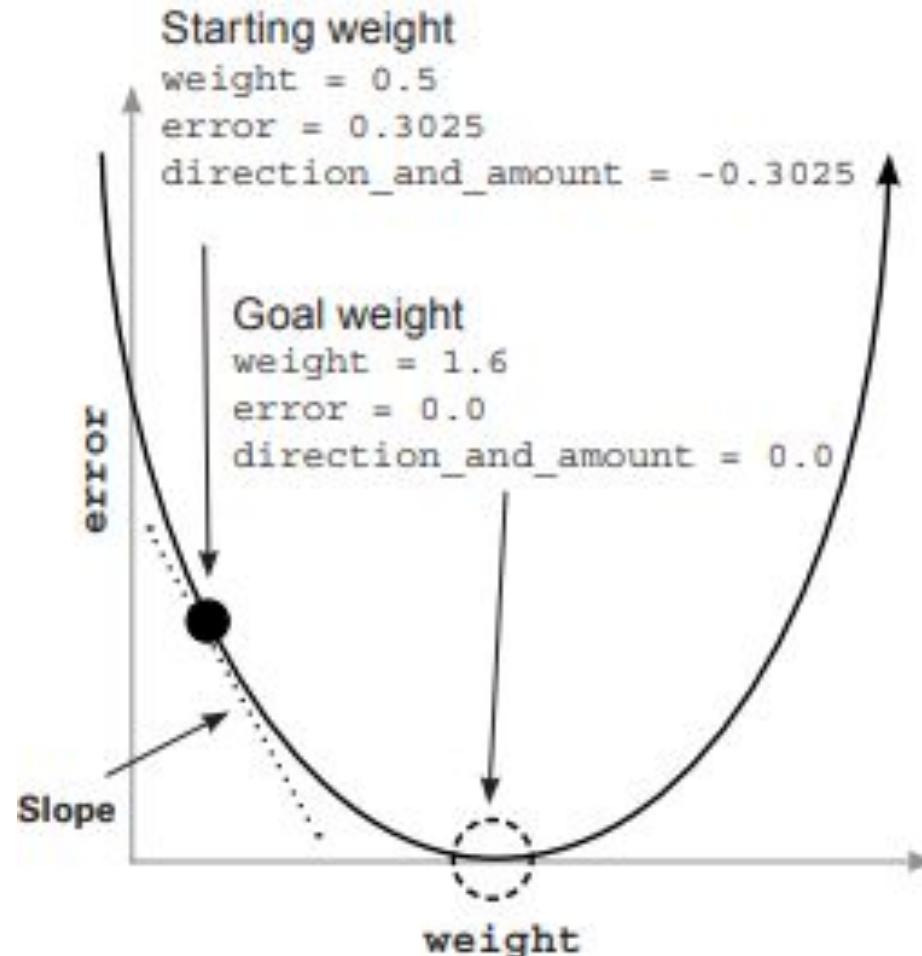
Derivatives: Take two



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

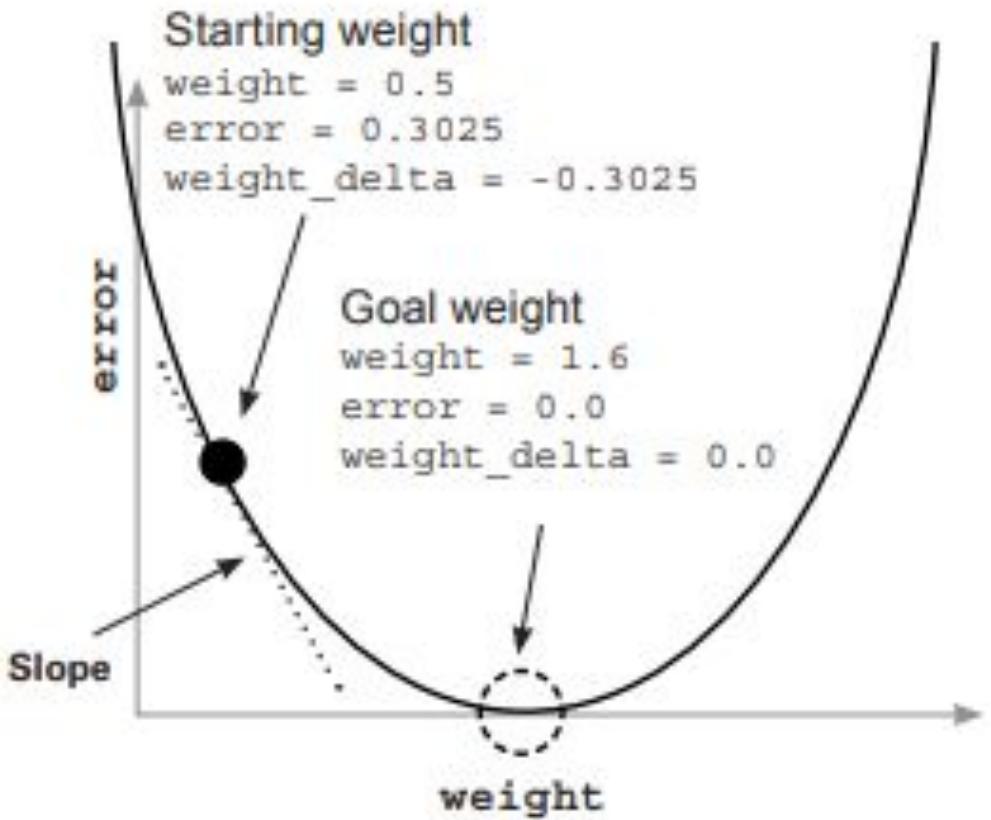


weight_delta is your derivative.

- 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

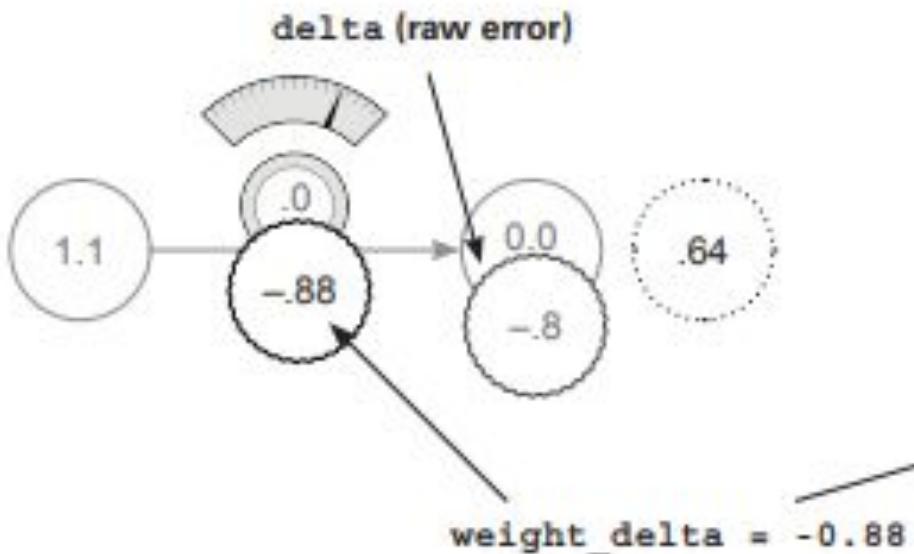
for iteration in range(4):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta

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

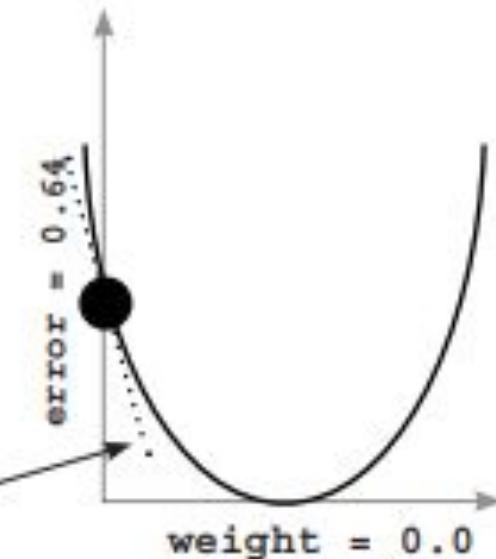
Derivative
(how fast the error changes, given changes in the weight)

Look familiar?

① A big weight increase

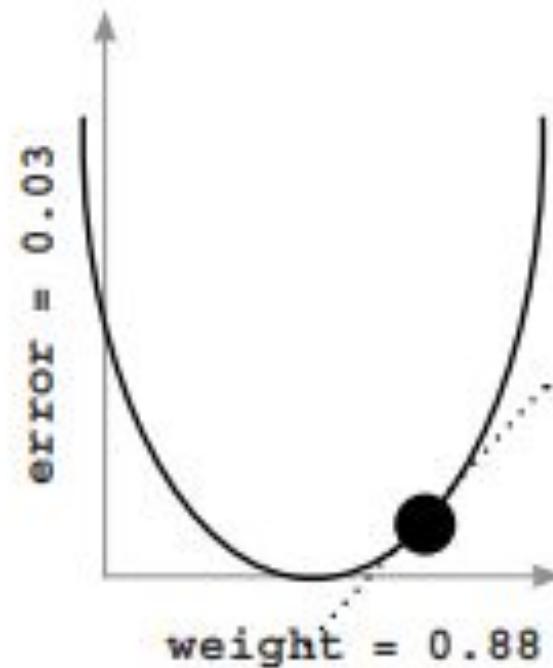
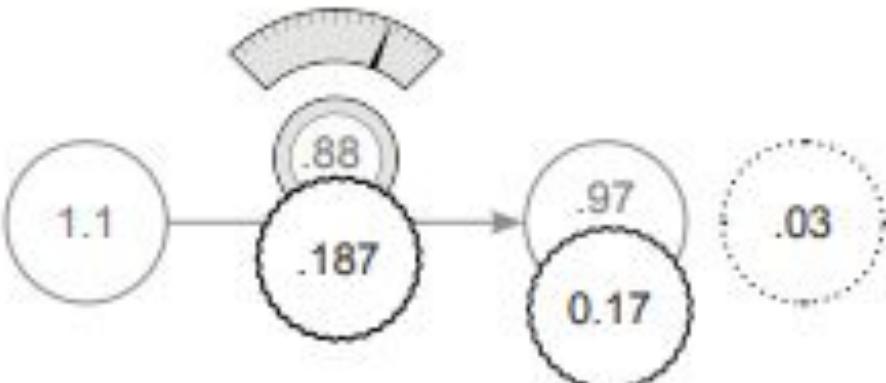


(Raw error modified for scaling, negative reversal,
and stopping per this weight
and input)



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
```

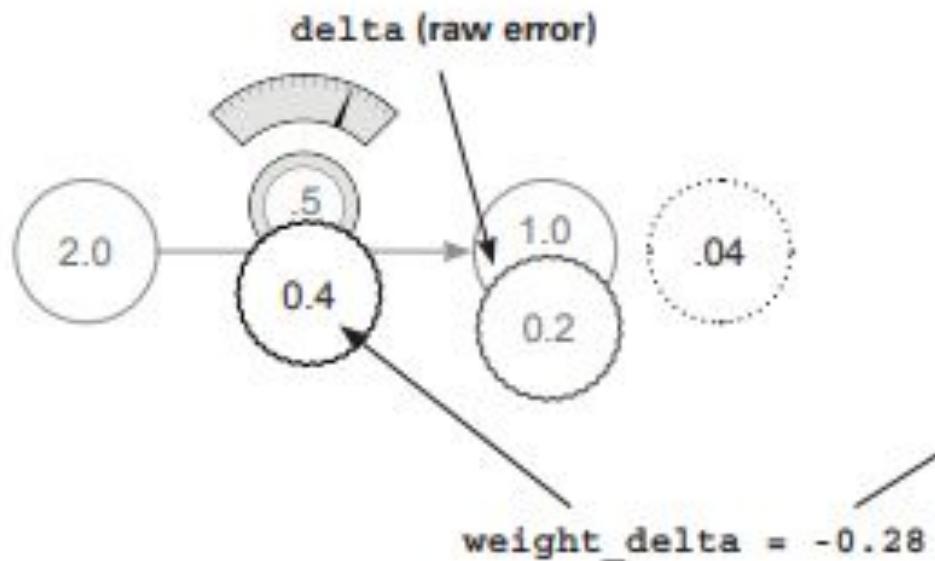
Breaking gradient descent

- 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:

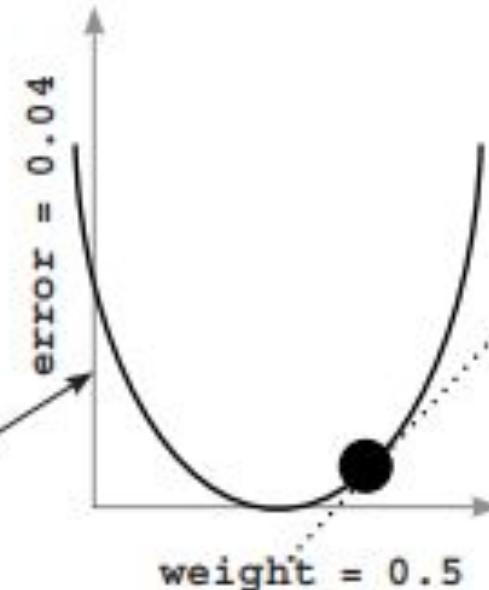
```
Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6
...
Error:6.67087267987e+14 Prediction:-25828031.8
Error:6.00378541188e+15 Prediction:77484098.6
Error:5.40340687069e+16 Prediction:-232452292.6
```

Visualizing the overcorrections

① A big weight increase

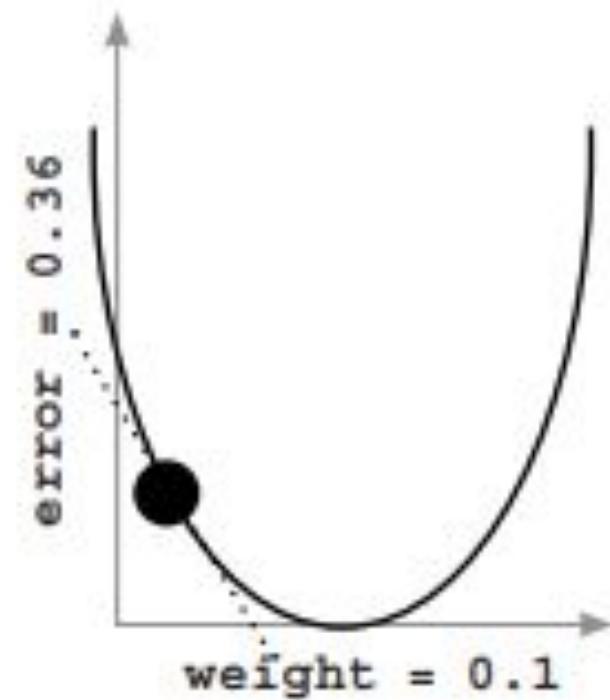
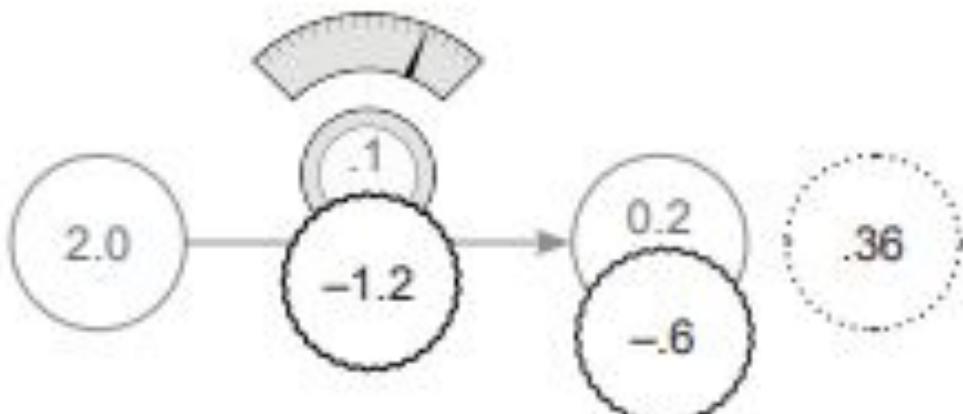


(Raw error modified for scaling,
negative reversal, and stopping
per this weight and input)



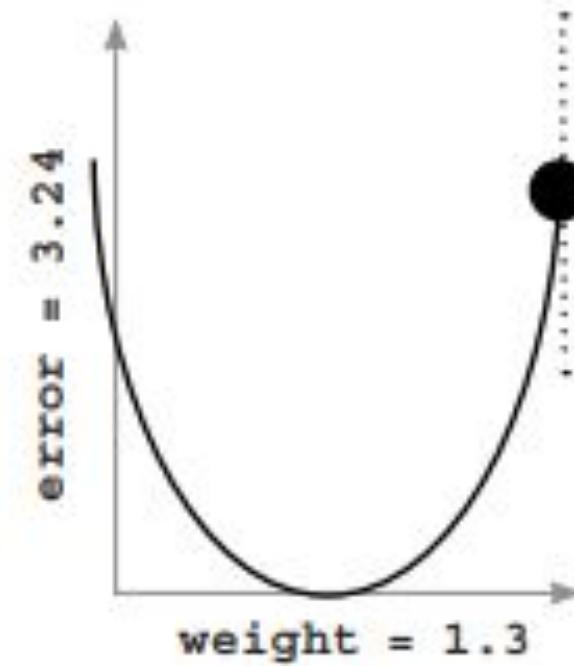
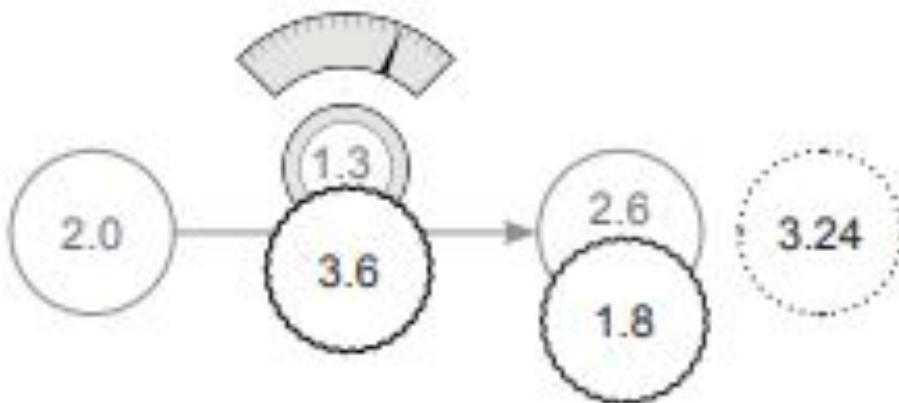
Visualizing the overcorrections

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



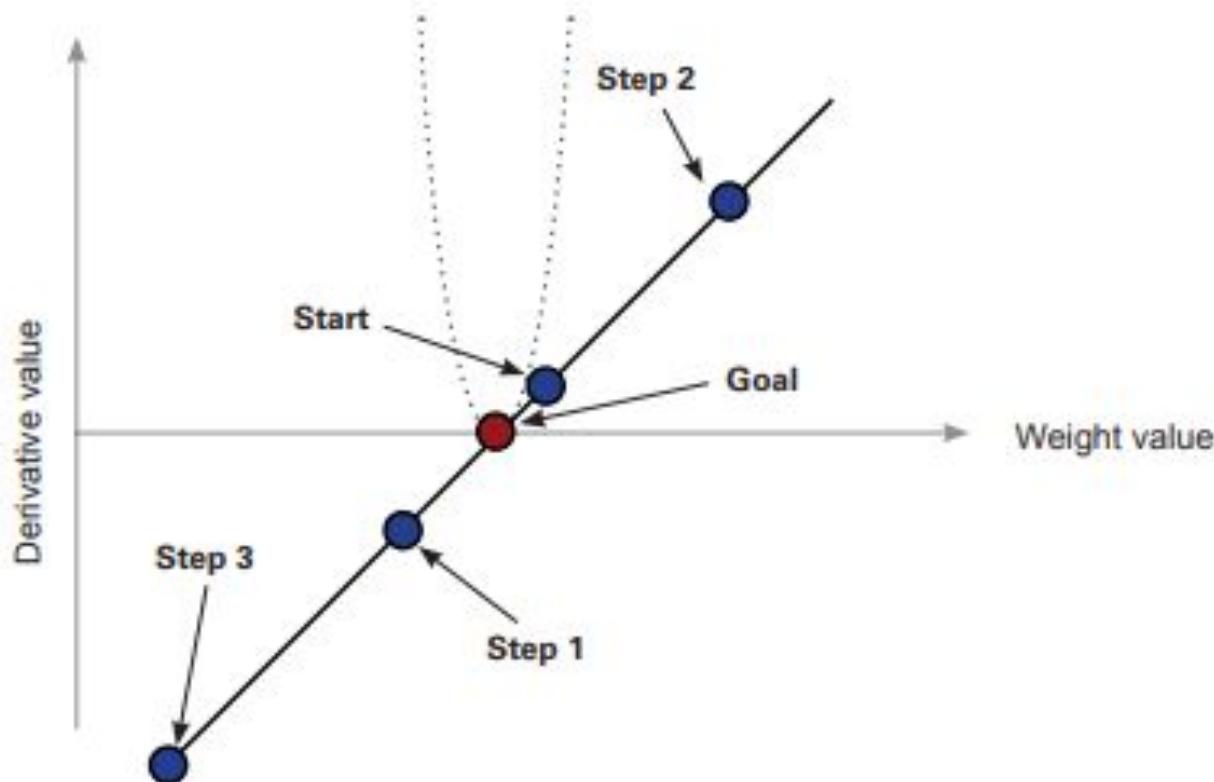
Visualizing the overcorrections

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:

weight = weight - (input * (pred - goal_pred))



Divergence



- If you have a big input, the prediction is very sensitive to changes in the weight (**because pred = input * 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:

weight = weight - 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:

weight = weight - (alpha * 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 ←

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))
```

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

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.

Lesson 5 learning multiple weights at a time: generalizing gradient descent





Generalizing gradient descent



In this lesson

- Gradient descent learning with multiple inputs
- Freezing one weight: what does it do?
- Gradient descent learning with multiple outputs
- Gradient descent learning with multiple inputs and outputs
- Visualizing weight values
- Visualizing dot products

Gradient descent learning with multiple inputs

Gradient descent also works with multiple inputs.

- In the preceding lesson, you learned how to use gradient descent to update a weight.
- In this lesson, we'll more or less reveal how the same techniques can be used to update a network that contains multiple weights.
- Let's start by jumping in the deep end, shall we?

Gradient descent learning with multiple inputs

- The following diagram shows how a network with multiple inputs can learn.

1 An empty network with multiple inputs

Input data enters here (three at a time).

#toes

win loss

#fans

Predictions come out here.

.1

.2

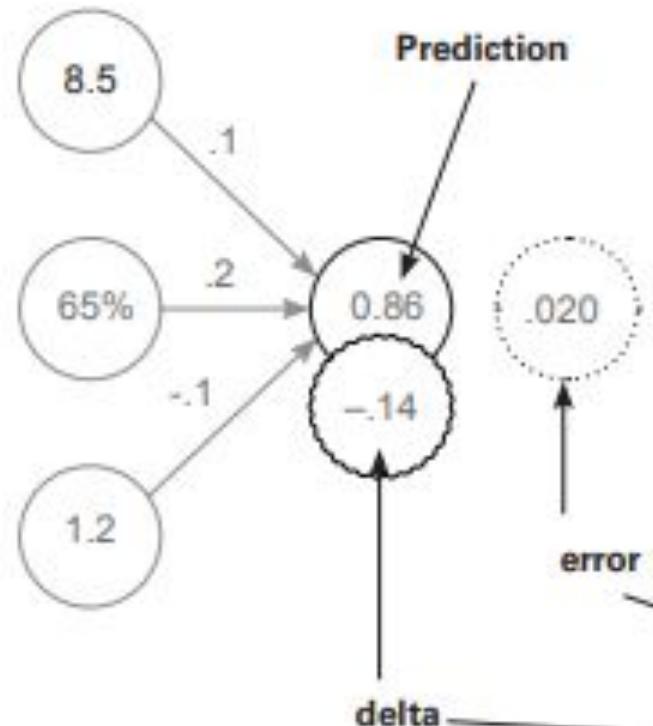
-.1

win?

```
def w_sum(a,b):  
    assert(len(a) == len(b))  
    output = 0  
    for i in range(len(a)):  
        output += (a[i] * b[i])  
    return output  
  
weights = [0.1, 0.2, -.1]  
  
def neural_network(input, weights):  
    pred = w_sum(input,weights)  
    return pred
```

Gradient descent learning with multiple inputs

② PREDICT + COMPARE: Making a prediction, and calculating error and delta



Input corresponds to every entry
for the first game of the season.

```
toes = [8.5 , 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2 , 1.3, 0.5, 1.0]

win_or_lose_binary = [1, 1, 0, 1]

true = win_or_lose_binary[0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)

error = (pred - true) ** 2

delta = pred - true
```

Gradient descent learning with multiple inputs

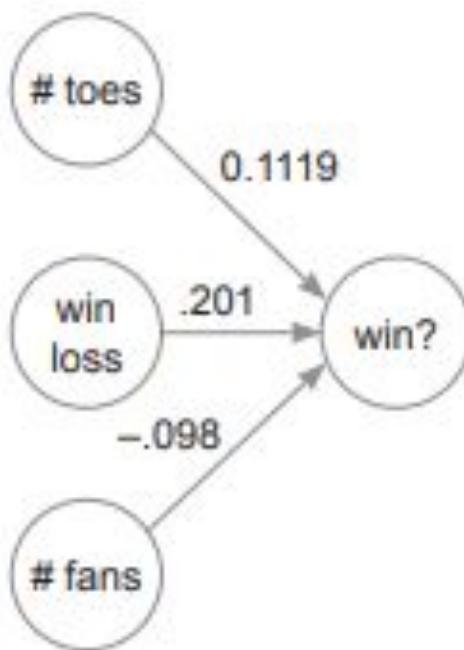
③ LEARN: Calculating each weight_delta and putting it on each weight



```
def ele_mul(number,vector):  
    output = [0,0,0]  
    assert(len(output) == len(vector))  
    for i in range(len(vector)):  
        output[i] = number * vector[i]  
    return output  
  
input = [toes[0],wlrec[0],nfans[0]]  
pred = neural_network(input,weight)  
error = (pred - true) ** 2  
delta = pred - true  
weight_deltas = ele_mul(delta,input)  
  
8.5 * -0.14 = -1.19 = weight_deltas[0]  
0.65 * -0.14 = -0.091 = weight_deltas[1]  
1.2 * -0.14 = -0.168 = weight_deltas[2]
```

Gradient descent learning with multiple inputs

④ LEARN: Updating the weights



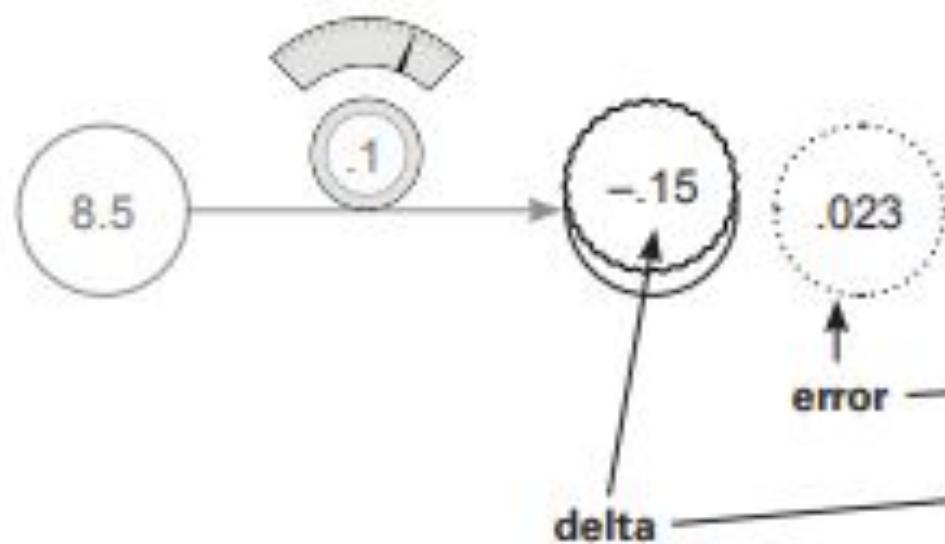
```
input = [toes[0],wlrec[0],nfans[0]]  
  
pred = neural_network(input,weight)  
error = (pred - true) ** 2  
delta = pred - true  
  
weight_deltas = ele_mul(delta,input)  
  
alpha = 0.01  
  
for i in range(len(weights)):  
    weights[i] -= alpha * weight_deltas[i]  
print("Weights:" + str(weights))  
print("Weight Deltas:" + str(weight_deltas))
```

$$0.1 - (-1.19 \cdot 0.01) = 0.1119 = \text{weights}[0]$$
$$0.2 - (-0.91 \cdot 0.01) = 0.2009 = \text{weights}[1]$$
$$-0.1 - (-1.68 \cdot 0.01) = -0.098 = \text{weights}[2]$$

Gradient descent with multiple inputs explained

- First, let's take a look at them side by side.

① Single input: Making a prediction and calculating error and delta



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

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

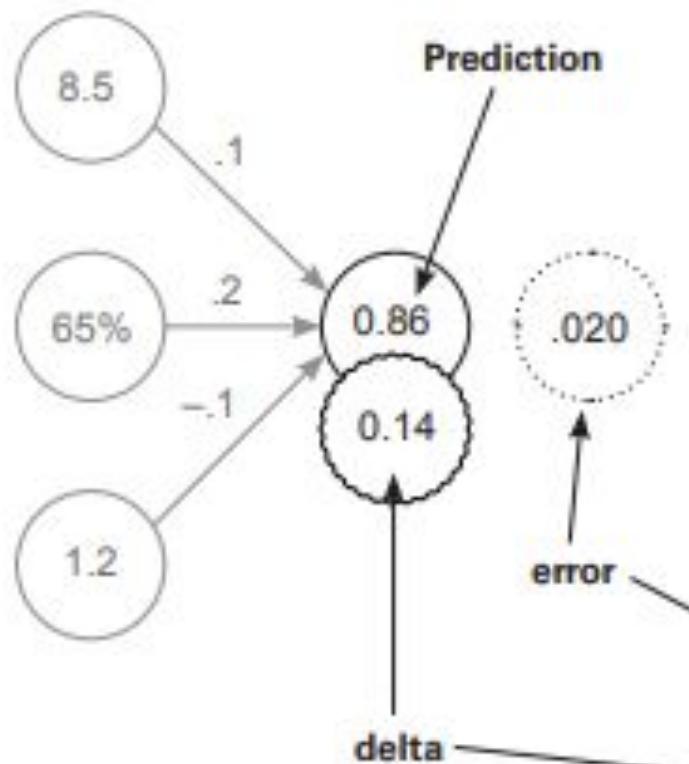
pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true
```

Gradient descent with multiple inputs explained

② Multi-input: Making a prediction and calculating error and delta



Input corresponds to every entry
for the first game of the season

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

win_or_lose_binary = [1, 1, 0, 1]

true = win_or_lose_binary[0]

input = [toes[0], wlrec[0], nfans[0]]

pred = neural_network(input, weights)

error = (pred - true) ** 2

delta = pred - true
```

Gradient descent with multiple inputs explained

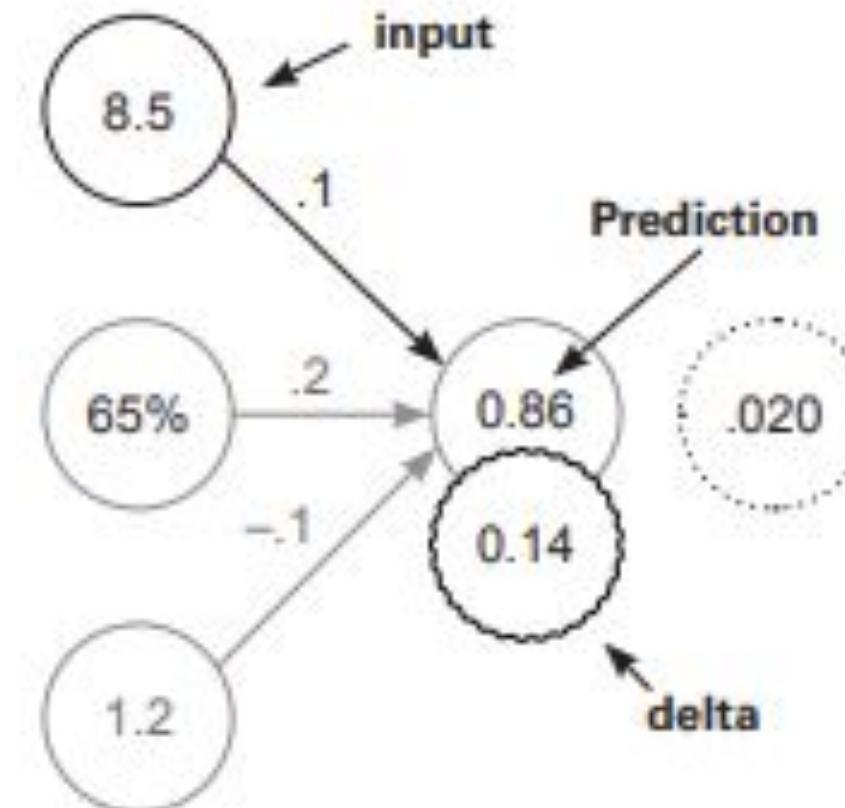
How do you turn a single delta (on the node) into three weight_delta values?

- Remember the definition and purpose of delta versus weight_delta.
- Delta is a measure of how much you want a node's value to be different.
- In this case, you compute it by a direct subtraction between the node's value and what you wanted the node's value to be ($\text{pred} - \text{true}$).
- Positive delta indicates the node's value was too high, and negative that it was too low.

Gradient descent with multiple inputs explained

Consider this from the perspective of a single weight, highlighted at right:

- **delta:** Hey, inputs—yeah, you three. Next time, predict a little higher.
- **Single weight:** Hmm: if my input was 0, then my weight wouldn't have mattered, and I wouldn't change a thing (stopping).



Gradient descent with multiple inputs explained

- Notice that the multi-weight version multiplies delta (0.14) by every input to create the various weight_deltas. It's a simple process.

③ Single input: Calculating weight_delta and putting it on the weight

```
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

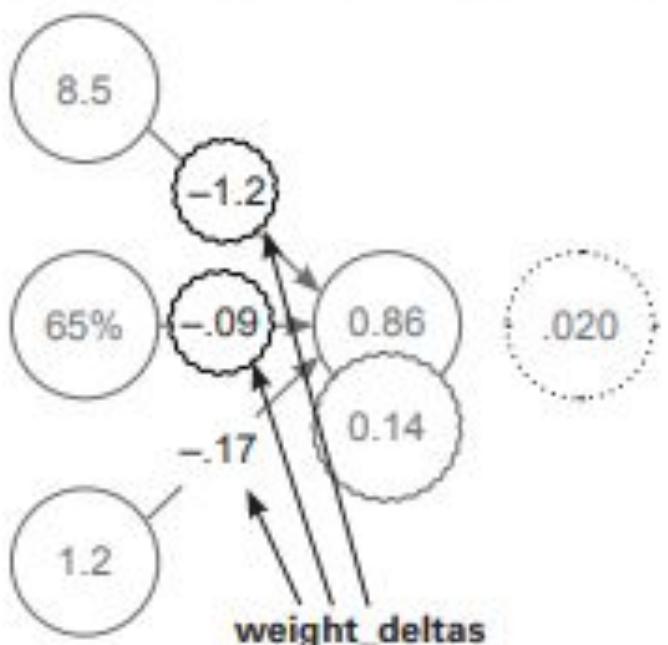
delta = pred - true

weight_delta = input * delta

8.5 * -0.15 = -1.25 => weight_delta
```

$8.5 * -0.15 = -1.25 \Rightarrow \text{weight_delta}$

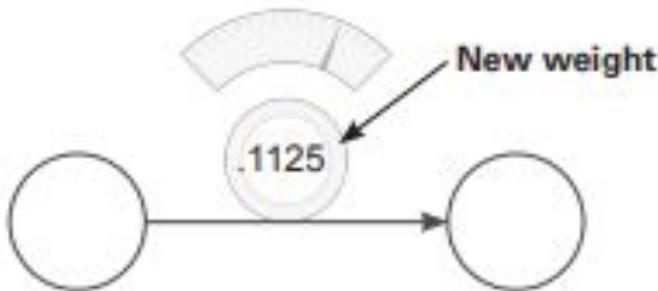
④ Multi-input: Calculating each weight_delta and putting it on each weight



```
def ele_mul(number,vector):  
    output = [0,0,0]  
    assert(len(output) == len(vector))  
    for i in range(len(vector)):  
        output[i] = number * vector[i]  
    return output  
  
input = [toes[0],wlrec[0],nfans[0]]  
pred = neural_network(input,weights)  
error = (pred - true) ** 2  
delta = pred - true  
weight_deltas = ele_mul(delta,input)  
  
8.5 * 0.14 = -1.2 => weight_deltas[0]  
0.65 * 0.14 = -.09 => weight_deltas[1]  
1.2 * 0.14 = -.17 => weight_deltas[2]
```

Gradient descent with multiple inputs explained

⑤ Updating the weight



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

```
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

delta = pred - true

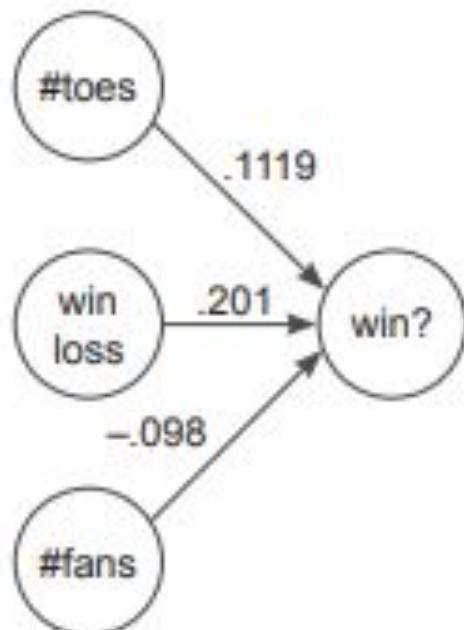
weight_delta = input * delta

alpha = 0.01 ← Fixed before training

weight -= weight_delta * alpha
```

Gradient descent with multiple inputs explained

⑥ Updating the weights



```
input = [toes[0],wlrec[0],nfans[0]]  
pred = neural_network(input,weights)  
error = (pred - true) ** 2  
delta = pred - true  
weight_deltas = ele_mul(delta,input)  
alpha = 0.01  
  
for i in range(len(weights)):  
    weights[i] -= alpha * weight_deltas[i]
```

$$0.1 - (1.19 \cdot 0.01) = 0.1119 = \text{weights}[0]$$
$$0.2 - (0.91 \cdot 0.01) = 0.2009 = \text{weights}[1]$$
$$-0.1 - (1.68 \cdot 0.01) = -0.098 = \text{weights}[2]$$

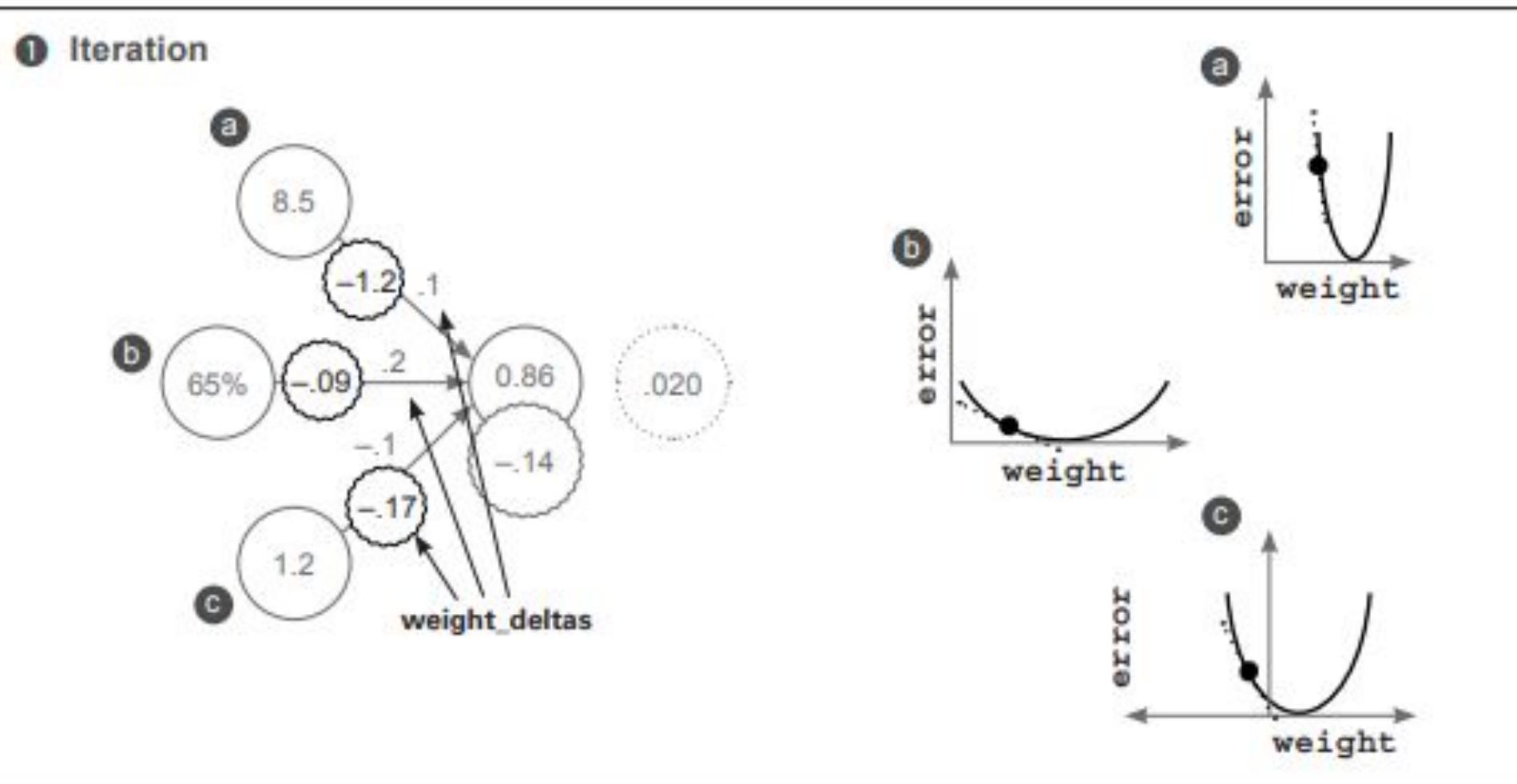
Let's watch several steps of learning

```
def neural_network(input, weights):  
    out = 0  
    for i in range(len(input)):  
        out += (input[i] * weights[i])  
    return out  
  
def ele_mul(scalar, vector):  
    out = [0,0,0]  
    for i in range(len(out)):  
        out[i] = vector[i] * scalar  
    return out  
  
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]  
  
win_or_lose_binary = [1, 1, 0, 1]  
true = win_or_lose_binary[0]  
  
alpha = 0.01  
weights = [0.1, 0.2, -.1]  
input = [toes[0],wlrec[0],nfans[0]]
```

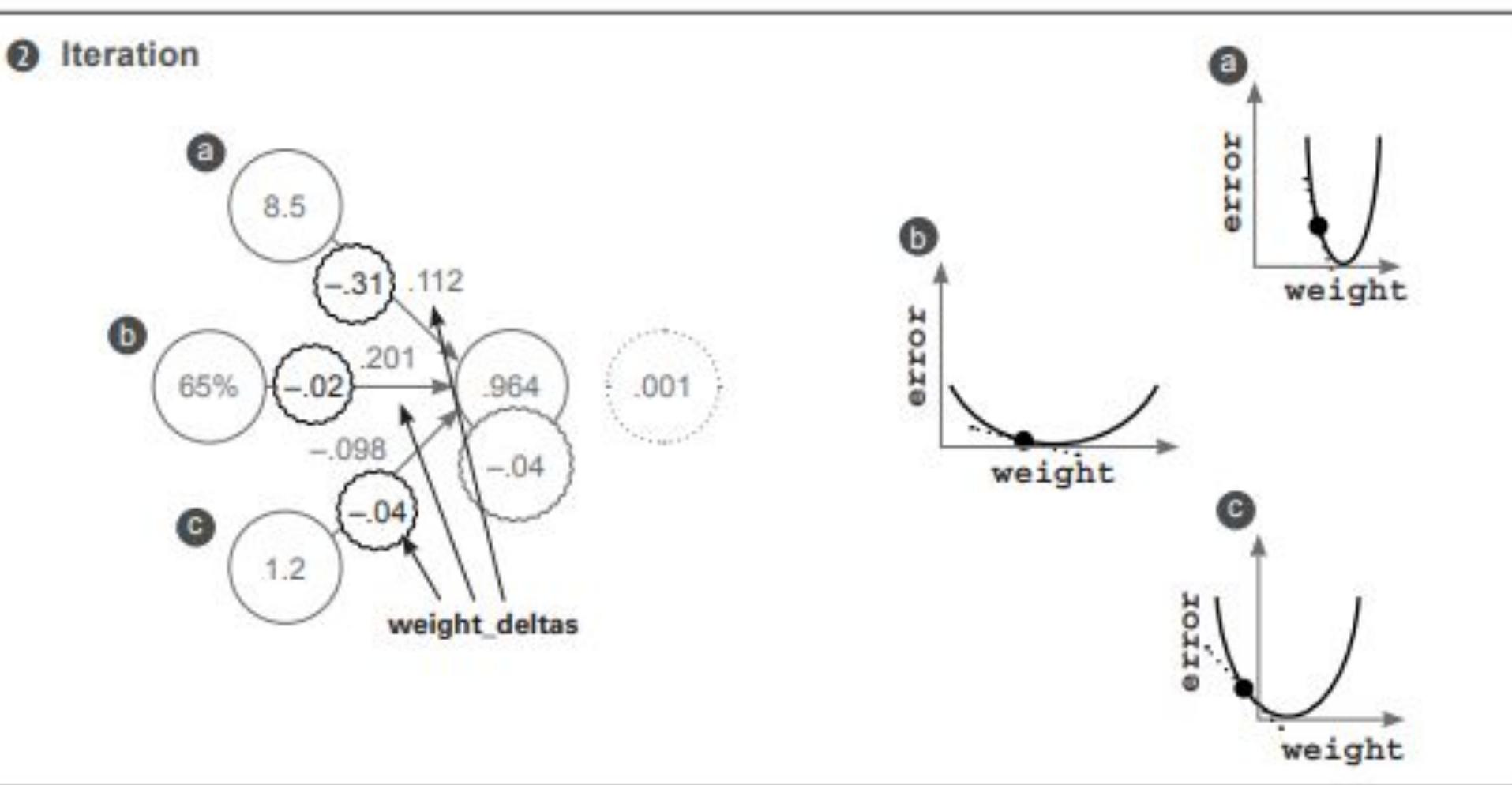
(continued)

```
for iter in range(3):  
  
    pred = neural_network(input,weights)  
  
    error = (pred - true) ** 2  
    delta = pred - true  
  
    weight_deltas=ele_mul(delta,input)  
  
    print("Iteration:" + str(iter+1))  
    print("Pred:" + str(pred))  
    print("Error:" + str(error))  
    print("Delta:" + str(delta))  
    print("Weights:" + str(weights))  
    print("Weight_Deltas:")  
    print(str(weight_deltas))  
    print()  
  
    for i in range(len(weights)):  
        weights[i] -= alpha*weight_deltas[i]
```

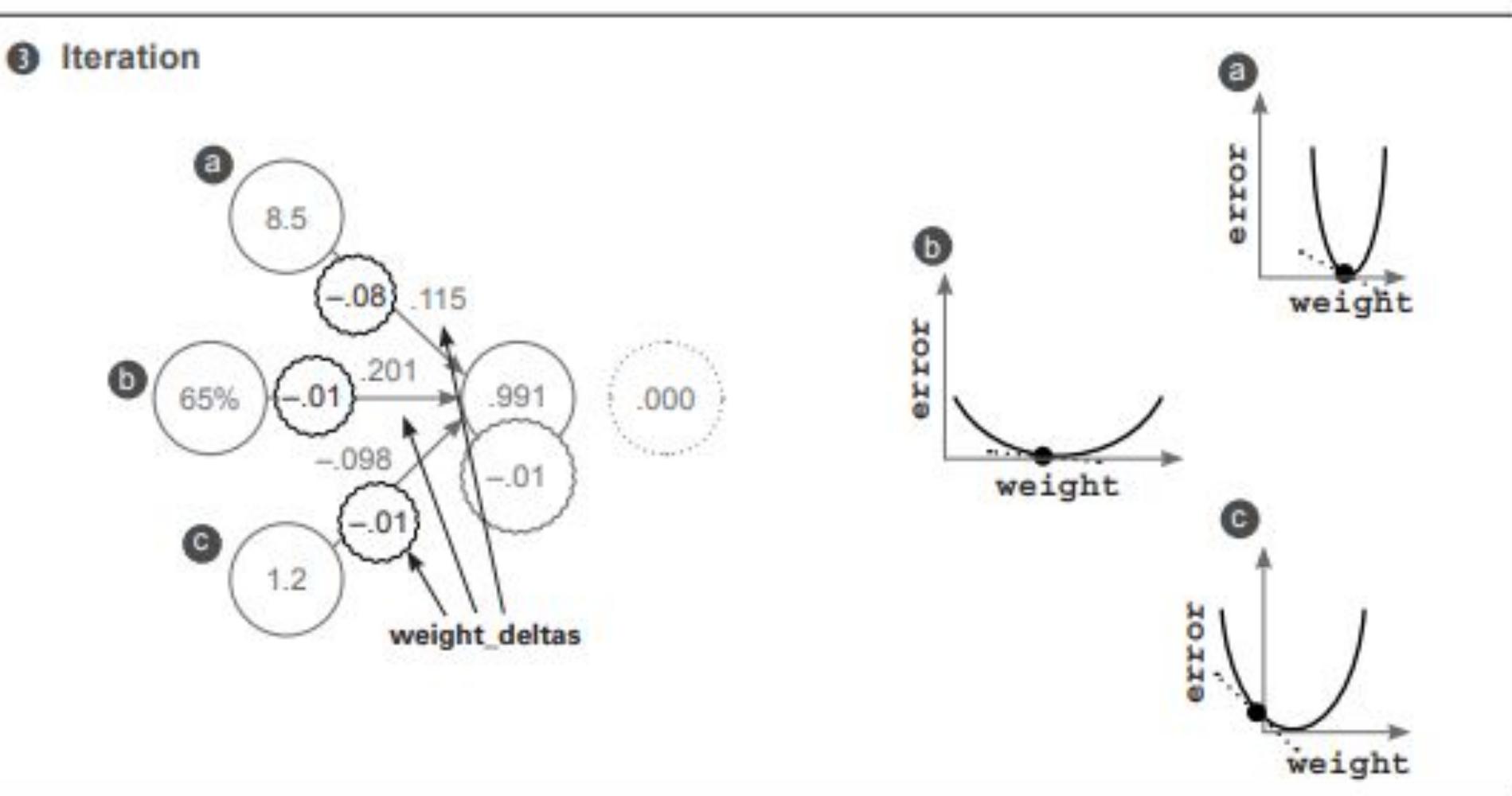
Let's watch several steps of learning



Let's watch several steps of learning



Let's watch several steps of learning



Freezing one weight: What does it do?

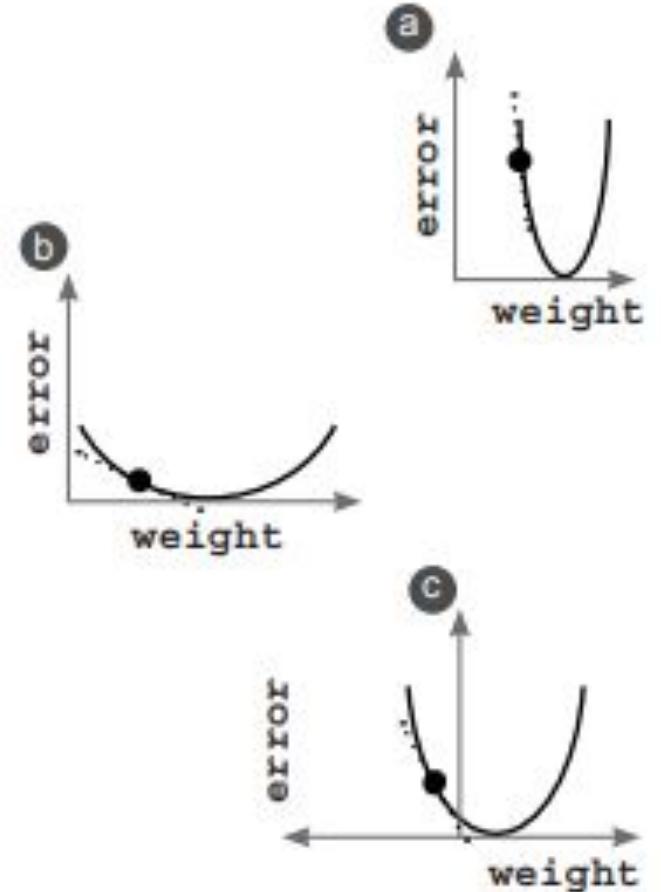
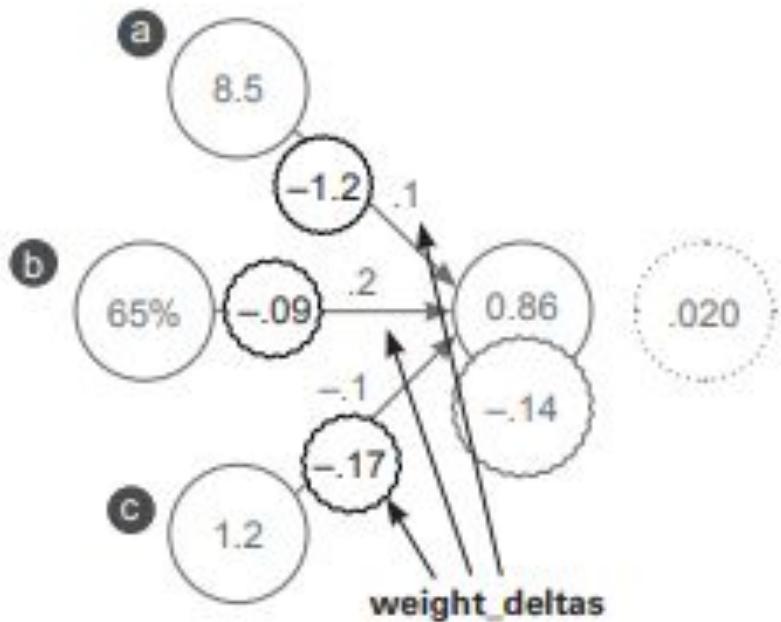
```
def neural_network(input, weights):  
    out = 0  
    for i in range(len(input)):  
        out += (input[i] * weights[i])  
    return out  
  
def ele_mul(scalar, vector):  
    out = [0,0,0]  
    for i in range(len(out)):  
        out[i] = vector[i] * scalar  
    return out  
  
toes = [8.5, 9.5, 9.9, 9.0]  
wlrec = [0.65, 0.8, 0.8, 0.9]  
nfans = [1.2, 1.3, 0.5, 1.0]  
  
win_or_lose_binary = [1, 1, 0, 1]  
true = win_or_lose_binary[0]  
  
alpha = 0.3  
weights = [0.1, 0.2, -.1]  
input = [toes[0],wlrec[0],nfans[0]]
```

(continued)

```
for iter in range(3):  
  
    pred = neural_network(input,weights)  
  
    error = (pred - true) ** 2  
    delta = pred - true  
  
    weight_deltas=ele_mul(delta,input)  
    weight_deltas[0] = 0  
  
    print("Iteration:" + str(iter+1))  
    print("Pred:" + str(pred))  
    print("Error:" + str(error))  
    print("Delta:" + str(delta))  
    print("Weights:" + str(weights))  
    print("Weight_Deltas:")  
    print(str(weight_deltas))  
    print()  
  
    for i in range(len(weights)):  
        weights[i]-=alpha*weight_deltas[i]
```

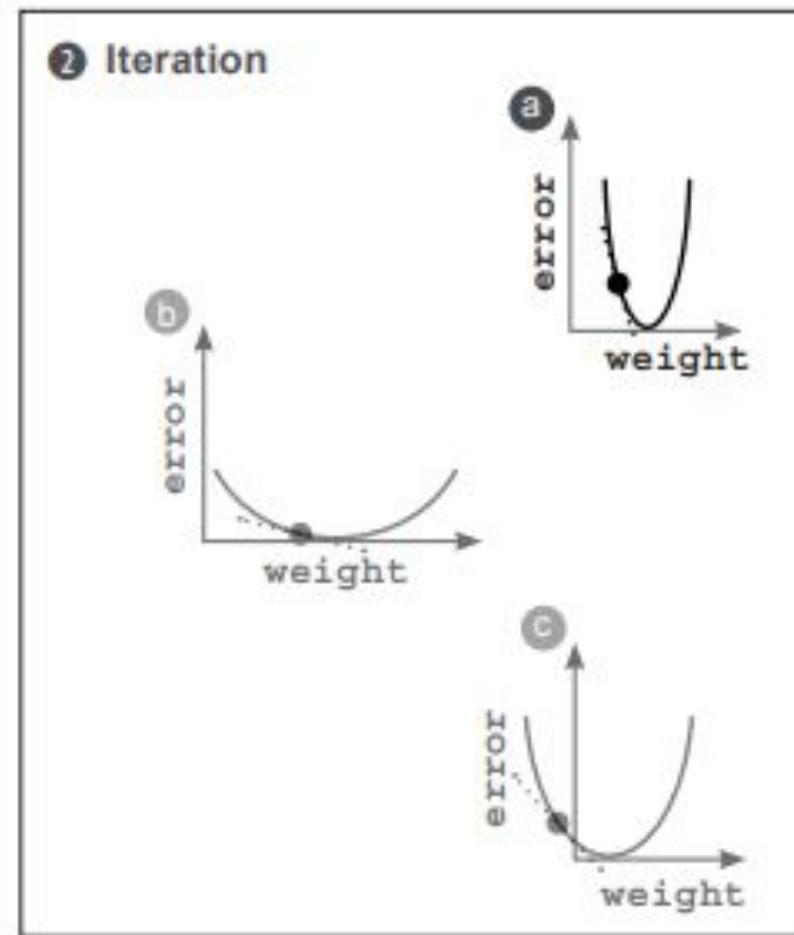
Freezing one weight: What does it do?

① Iteration



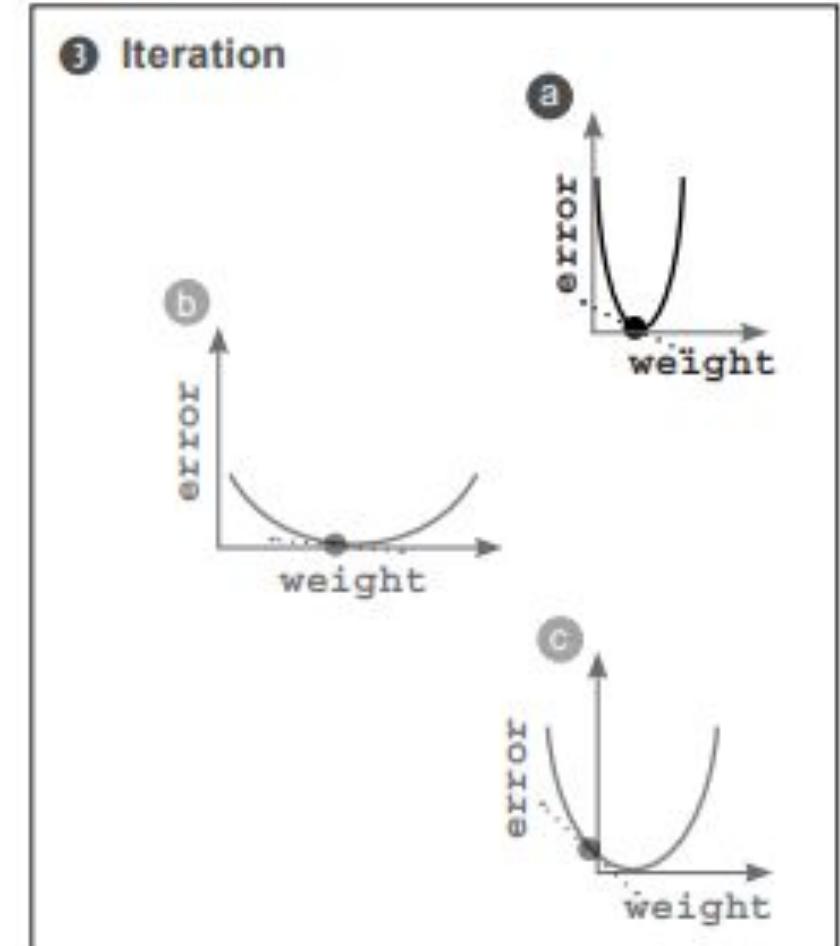
Freezing one weight: What does it do?

- This is an extremely important lesson.
- First, if you converged (reached error = 0) with b and c weights and then tried to train a , a wouldn't move. Why? error = 0, which means `weight_delta` is 0



Freezing one weight: What does it do?

- Instead of the black dot moving, the curve seems to move to the left.
- What does this mean? The black dot can move horizontally only if the weight is updated.
- Because the weight for a is frozen for this experiment, the dot must stay fixed. But error clearly goes to 0.



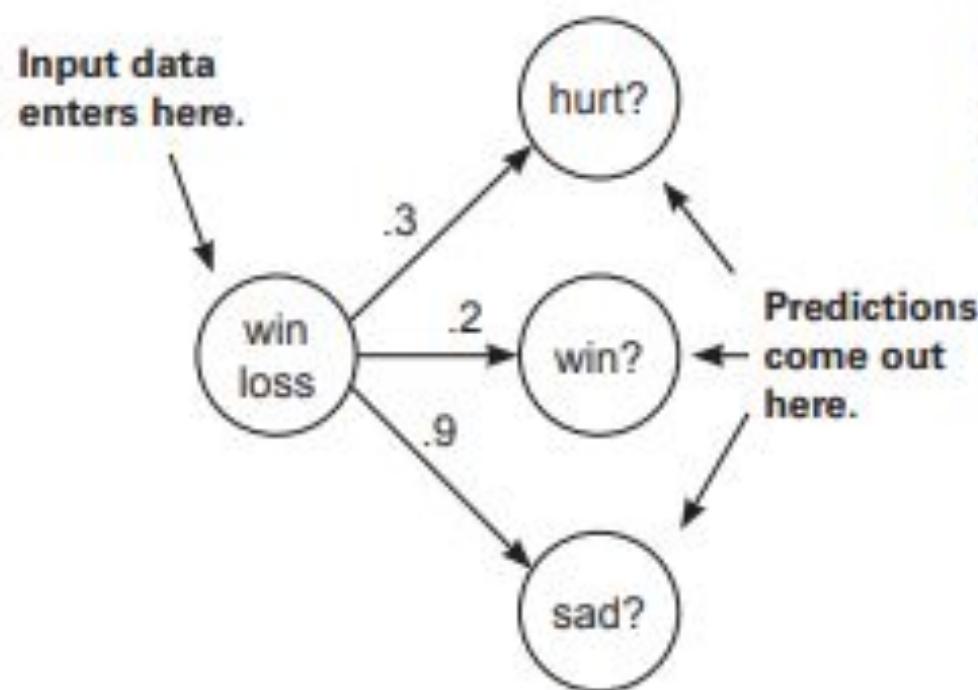
Gradient descent learning with multiple outputs

Neural networks can also make multiple predictions using only a single input.

- Perhaps this will seem a bit obvious.
- You calculate each delta the same way and then multiply them all by the same, single input.
- This becomes each weight's `weight_delta`.

Gradient descent learning with multiple outputs

① An empty network with multiple outputs

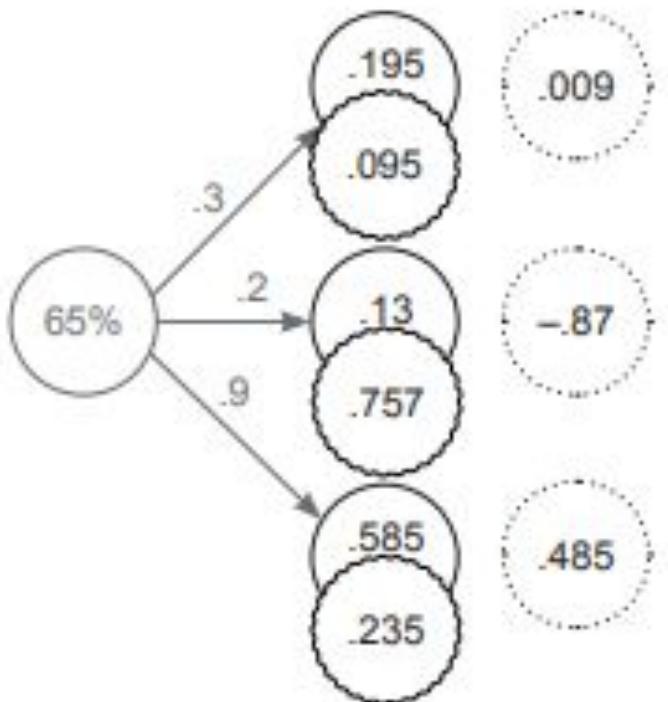


Instead of predicting just whether the team won or lost, now you're also predicting whether they're happy or sad and the percentage of the team members who are hurt. You're making this prediction using only the current win/loss record.

```
weights = [0.3, 0.2, 0.9]  
def neural_network(input, weights):  
    pred = ele_mul(input, weights)  
    return pred
```

Gradient descent learning with multiple outputs

② PREDICT: Making a prediction and calculating error and delta



```
wlrec = [0.65, 1.0, 1.0, 0.9]
hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [ 1, 1, 0, 1]
sad   = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]
```

Gradient descent learning with multiple outputs

③ COMPARE: Calculating each weight_delta and putting it on each weight



As before, weight_deltas are computed by multiplying the input node value with the output node delta for each weight. In this case, the weight_deltas share the same input node and have unique output nodes (deltas). Note also that you can reuse the ele_mul function.

```
def scalar_ele_mul(number,vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output

wlrec = [0.65, 1.0, 1.0, 0.9]
hurt = [0.1, 0.0, 0.0, 0.1]
win = [1, 1, 0, 1]
sad = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

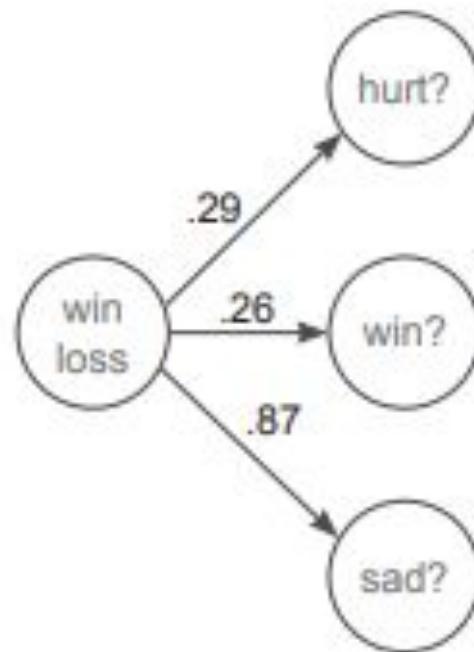
for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]

weight_deltas = scalar_ele_mul(input,weights)
```

Gradient descent learning with multiple outputs

④ LEARN: Updating the weights



```
input = wlrec[0]
true = [hurt[0], win[0], sad[0]]
pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):
    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]

weight_deltas = scalar_ele_mul(input,weights)
alpha = 0.1

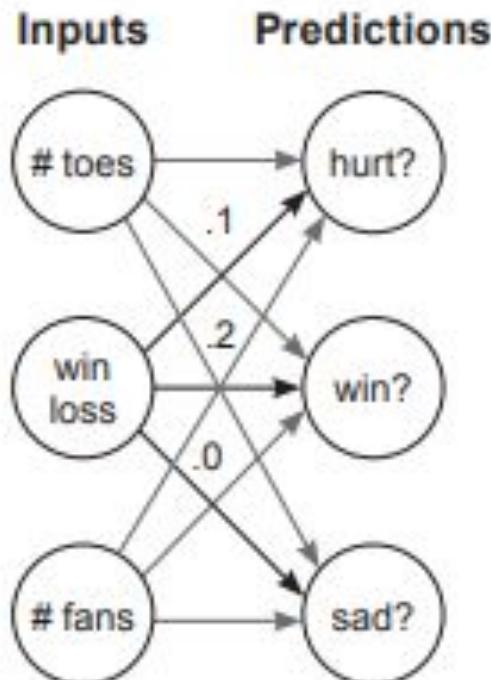
for i in range(len(weights)):
    weights[i] -= (weight_deltas[i] * alpha)

print("Weights:" + str(weights))
print("Weight Deltas:" + str(weight_deltas))
```

Gradient descent with multiple inputs and outputs

Gradient descent generalizes to arbitrarily large networks.

① An empty network with multiple inputs and outputs



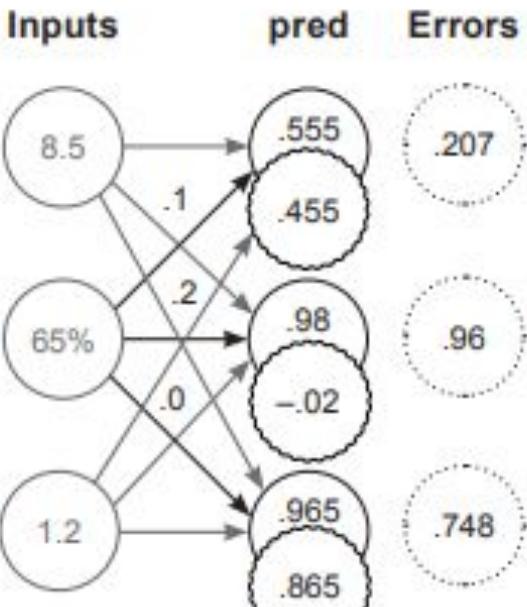
```
# toes %win # fans
weights = [ [0.1, 0.1, -0.3], # hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ] # sad?

def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]
    for i in range(len(vect)):
        output[i] = w_sum(vect,matrix[i])
    return output

def neural_network(input, weights):
    pred = vect_mat_mul(input,weights)
    return pred
```

Gradient descent with multiple inputs and outputs

② PREDICT: Making a prediction and calculating error and delta



```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

hurt = [0.1, 0.0, 0.0, 0.1]
win = [1, 1, 0, 1]
sad = [0.1, 0.0, 0.1, 0.2]

alpha = 0.01

input = [toes[0], wlrec[0], nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input, weights)

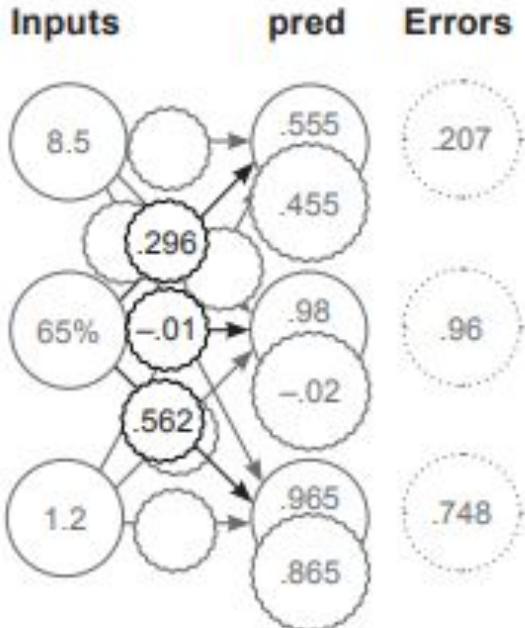
error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]
```

Gradient descent with multiple inputs and outputs

- ③ COMPARE: Calculating each weight_delta and putting it on each weight



```
def outer_prod(vec_a, vec_b):
    out = zeros_matrix(len(a),len(b))
    for i in range(len(a)):
        for j in range(len(b)):
            out[i][j] = vec_a[i]*vec_b[j]
    return out

input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

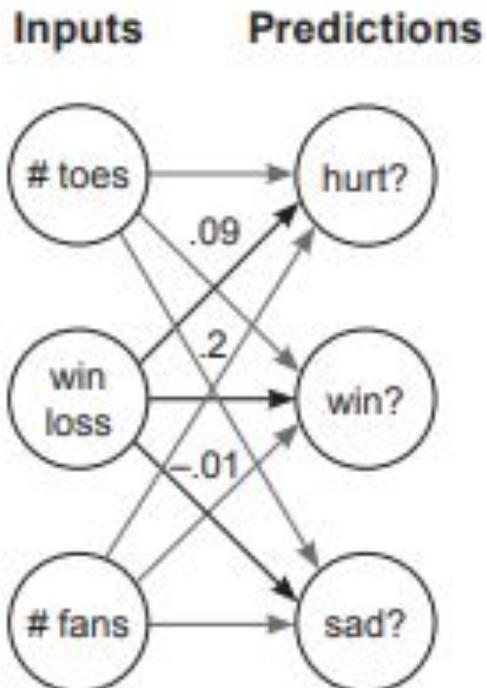
for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]

    weight_deltas = outer_prod(input,delta)
```

Gradient descent with multiple inputs and outputs

④ LEARN: Updating the weights



```
input = [toes[0],wlrec[0],nfans[0]]  
true = [hurt[0], win[0], sad[0]]  
  
pred = neural_network(input,weights)  
  
error = [0, 0, 0]  
delta = [0, 0, 0]  
  
for i in range(len(true)):  
  
    error[i] = (pred[i] - true[i]) ** 2  
    delta[i] = pred[i] - true[i]  
  
weight_deltas = outer_prod(input,delta)  
  
for i in range(len(weights)):  
    for j in range(len(weights[0])):  
        weights[i][j] -= alpha * \  
            weight_deltas[i][j]
```



What do these weights learn?



- Congratulations! This is the part of the course where we move on to the first real-world dataset.
 - As luck would have it, it's one with historical significance.
 - It's called the Modified National Institute of Standards and Technology (MNIST) dataset, and it consists of digits that high school students and employees of the US Census Bureau handwrote some years ago.
- 

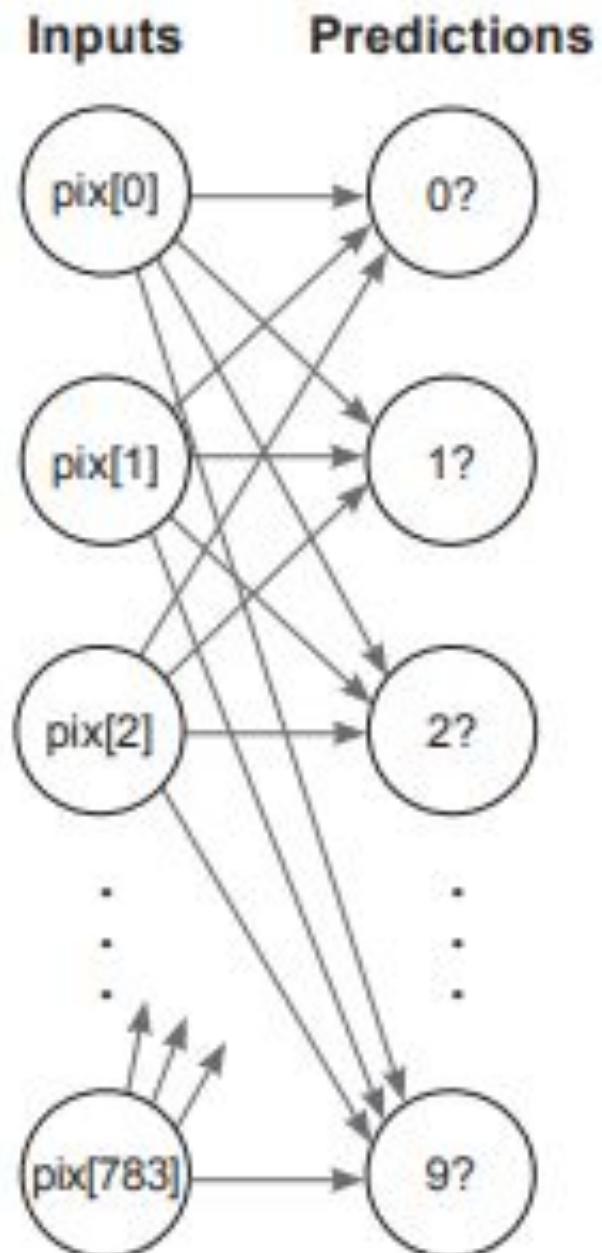


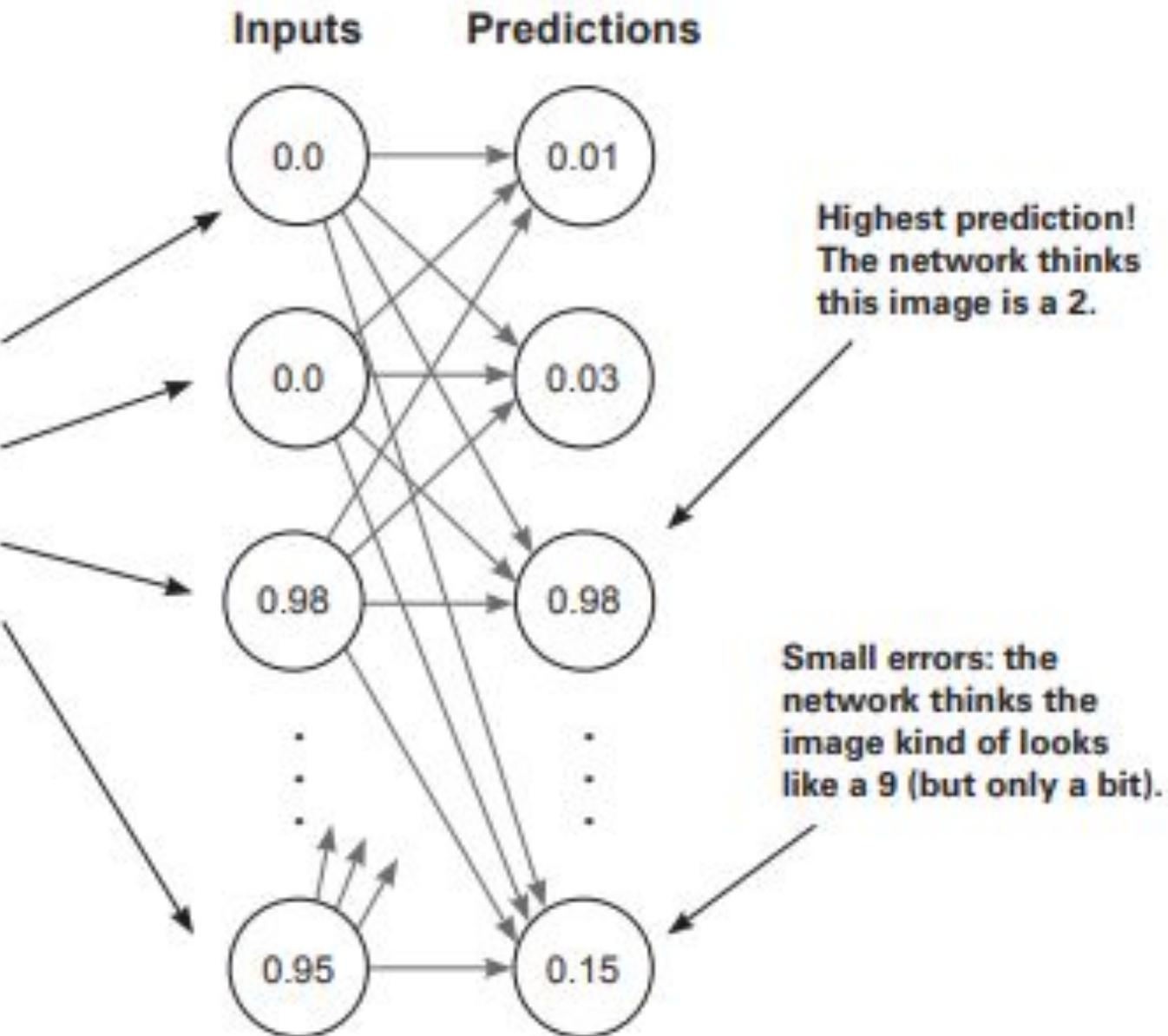
What do these weights learn?

- Each image is only 784 pixels (28×28). Given that you have 784 pixels as input and 10 possible labels as output, you can imagine the shape of the neural network: each training example contains 784 values (one for each pixel), so the neural network must have 784 input values

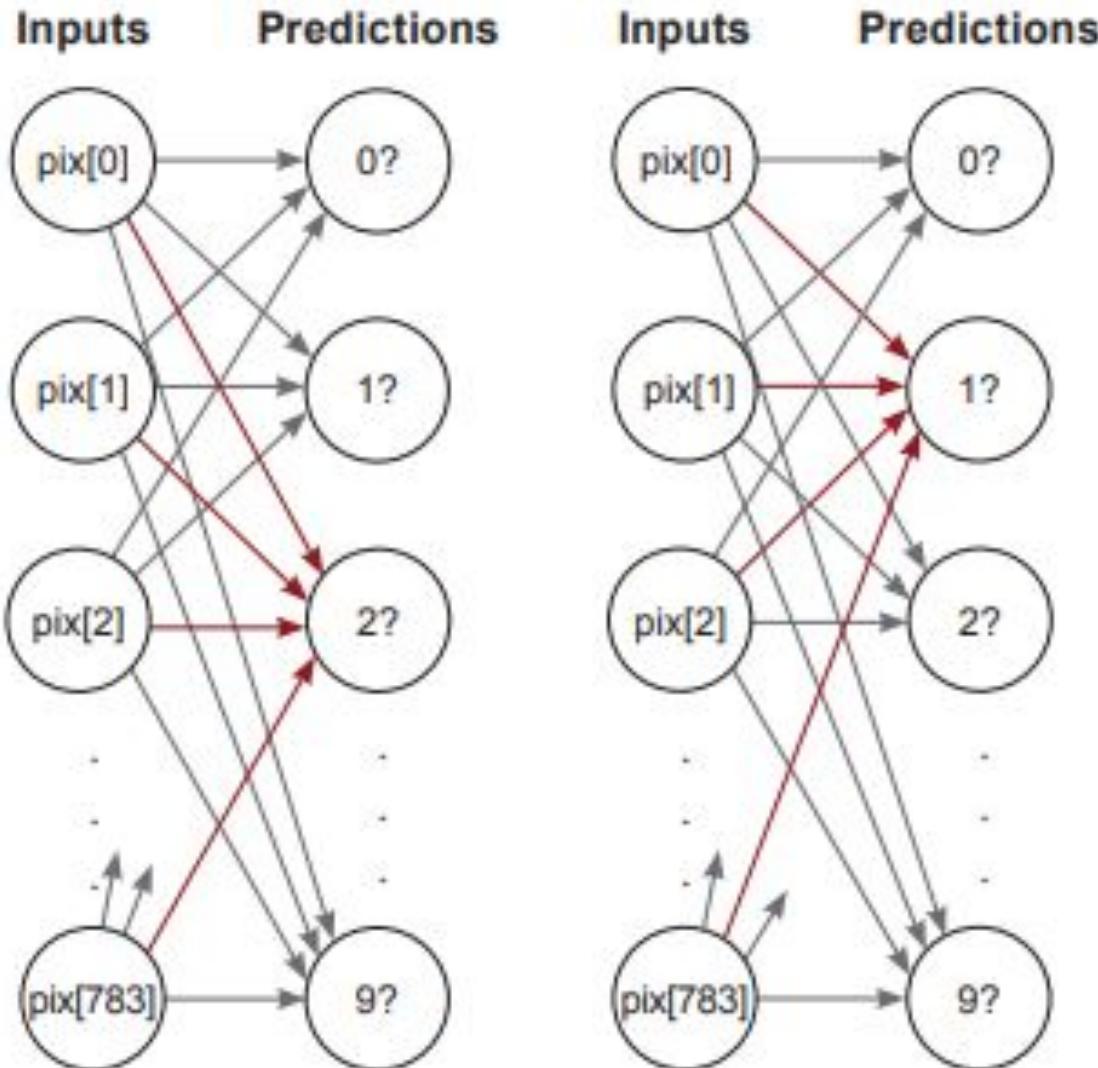


- This diagram represents the new MNIST classification neural network.
- It most closely resembles the network you trained with multiple inputs and outputs earlier.
- The only difference is the number of inputs and outputs, which has increased substantially.

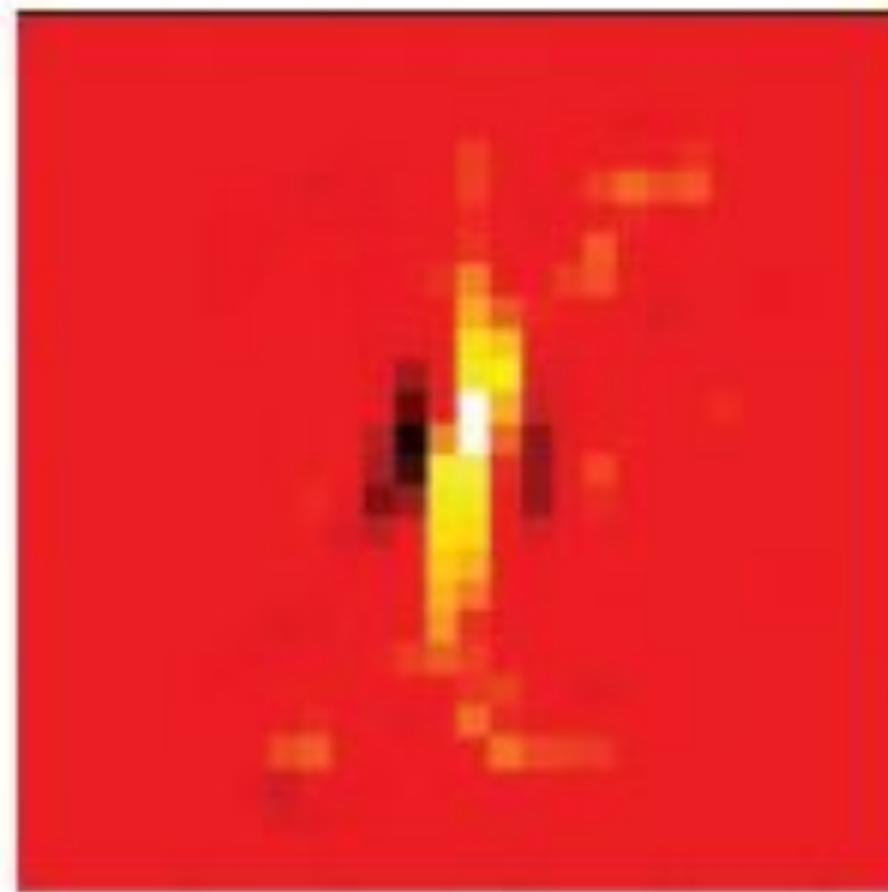
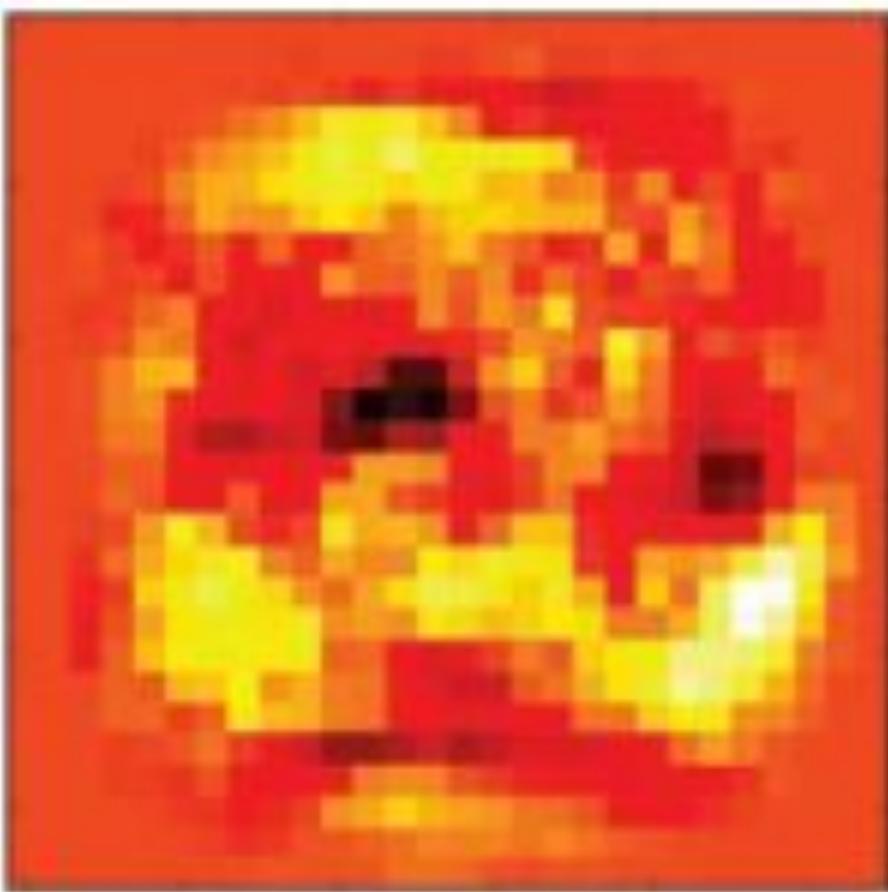




Visualizing weight values



Visualizing weight values



Visualizing dot products (weighted sums)

- Recall how dot products work.
- They take two vectors, multiply them together (elementwise), and then sum over the output.
- Consider this example:

```
a = [ 0, 1, 0, 1]  
b = [ 1, 0, 1, 0]
```

```
[ 0, 0, 0, 0] -> 0
```

Score

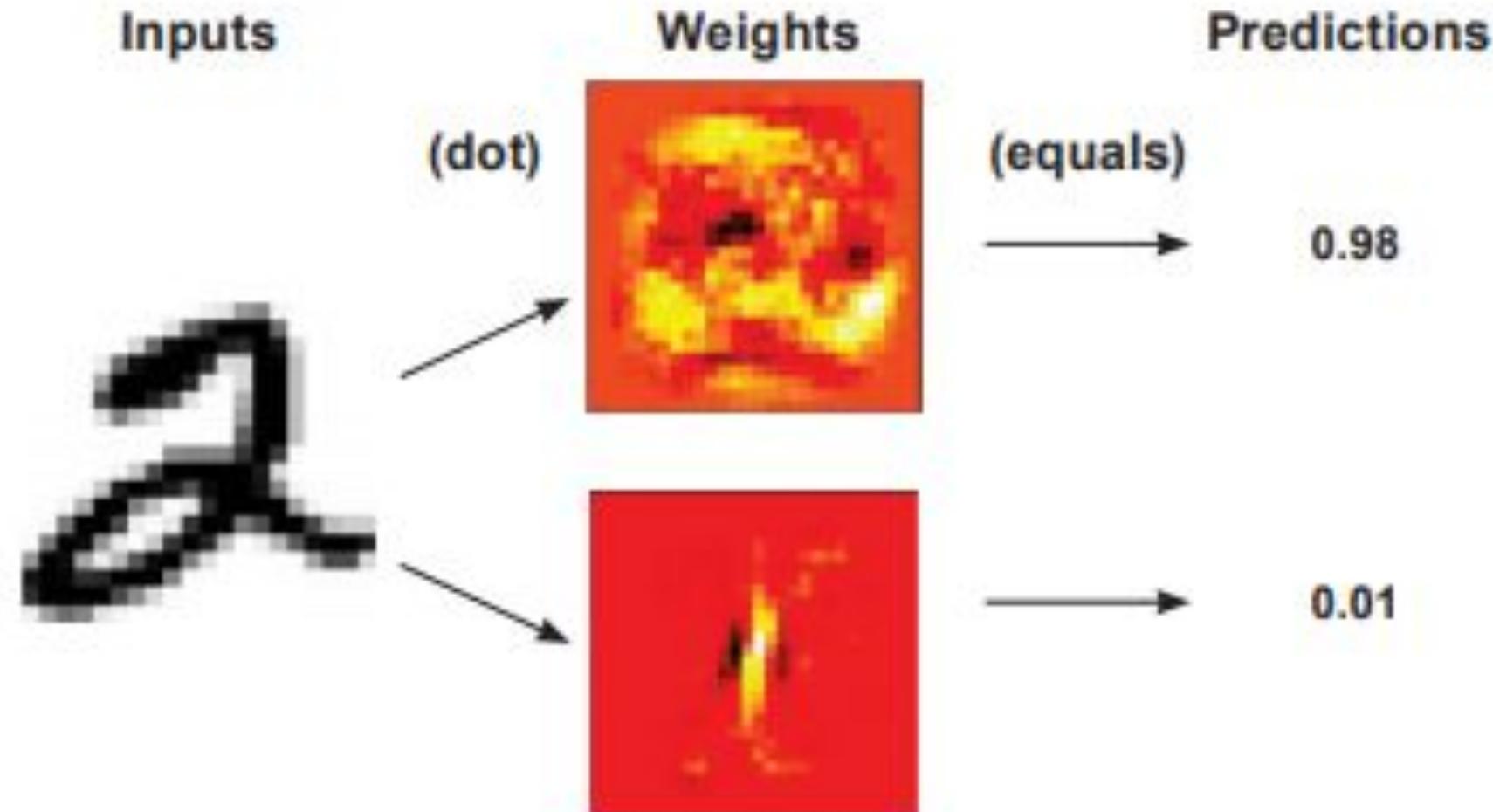
Visualizing dot products (weighted sums)

- First you multiply each element in a and b by each other, in this case creating a vector of 0s.
- The sum of this vector is also 0.
- Why? Because the vectors have nothing in common.

$c = [0, 1, 1, 0]$
 $d = [.5, 0, .5, 0]$

$b = [1, 0, 1, 0]$
 $c = [0, 1, 1, 0]$

Visualizing dot products (weighted sums)





Summary

Gradient descent is a general learning algorithm.

- Perhaps the most important subtext of this lesson is that
 - gradient descent is a very flexible learning algorithm.
 - If you combine weights in a way that allows you to calculate an error function and a delta, gradient descent can show you how to move the weights to reduce the error.
 - We'll spend the rest of this course exploring different types of weight combinations and error functions for which gradient descent is useful.
- 

Lesson 6 building your first deep neural network: introduction to backpropagation





Introduction to backpropagation

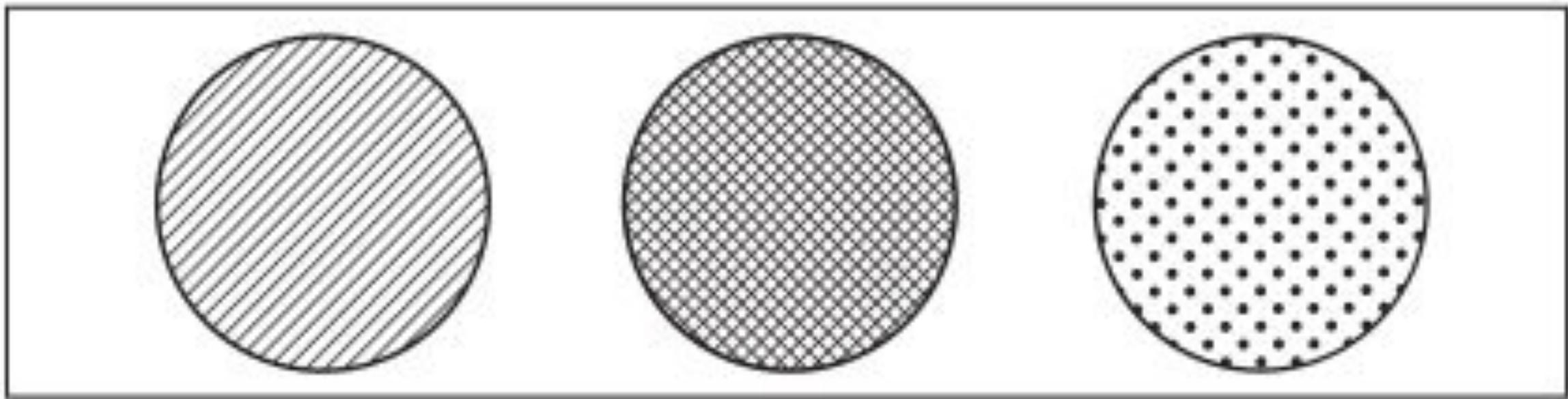


In this lesson

- The streetlight problem
 - Matrices and the matrix relationship
 - Full, batch, and stochastic gradient descent
 - Neural networks learn correlation
 - Overfitting
 - Creating your own correlation
 - Backpropagation: long-distance error attribution
- 

The streetlight problem

This toy problem considers how a network learns entire datasets





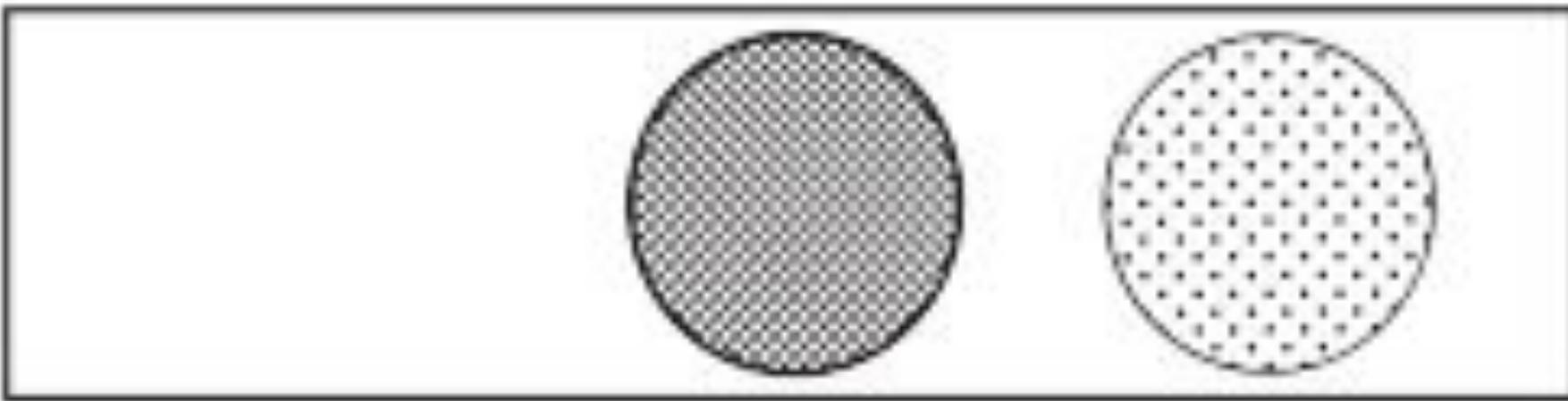
The streetlight problem



STOP



The streetlight problem

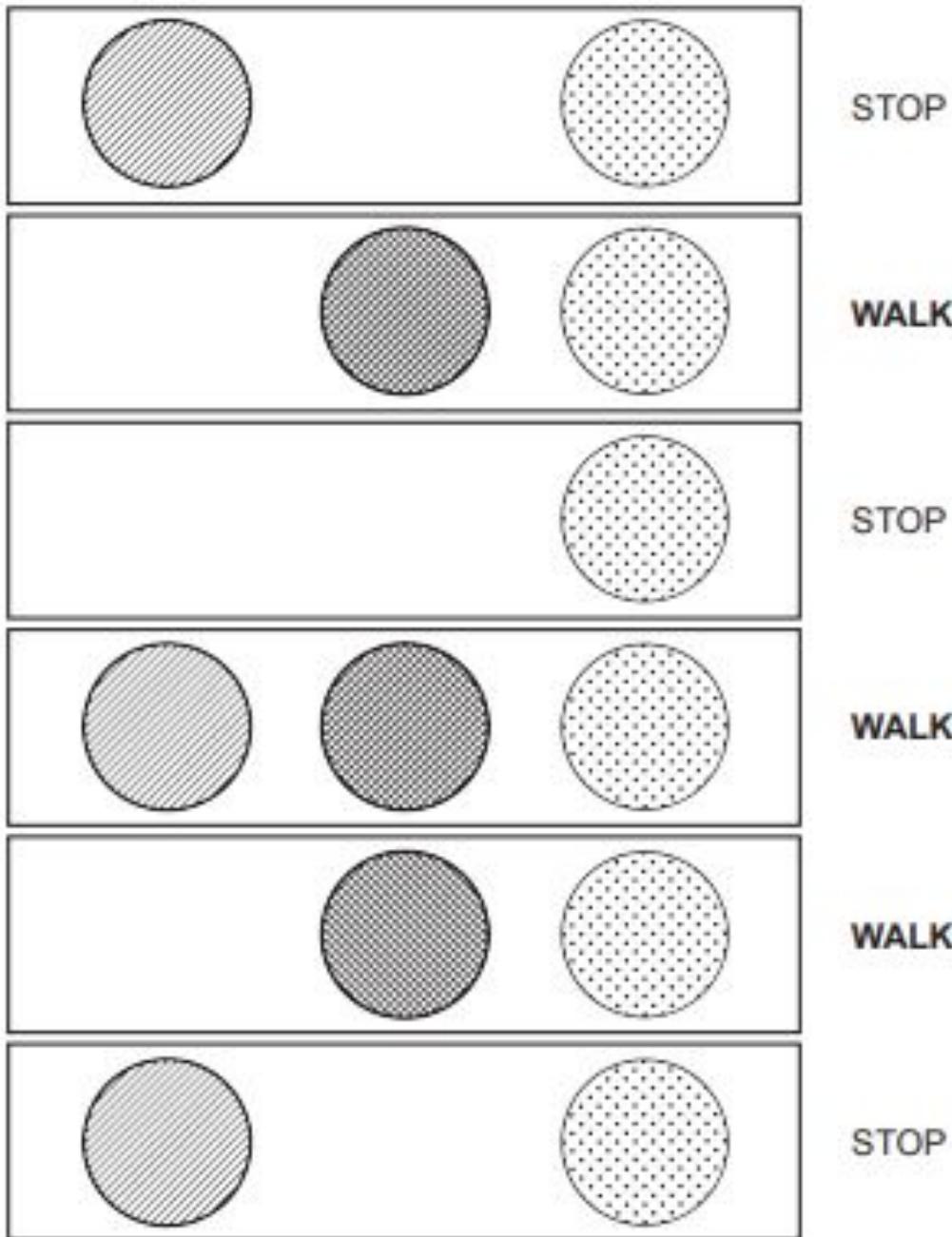


WALK

The streetlight problem



STOP



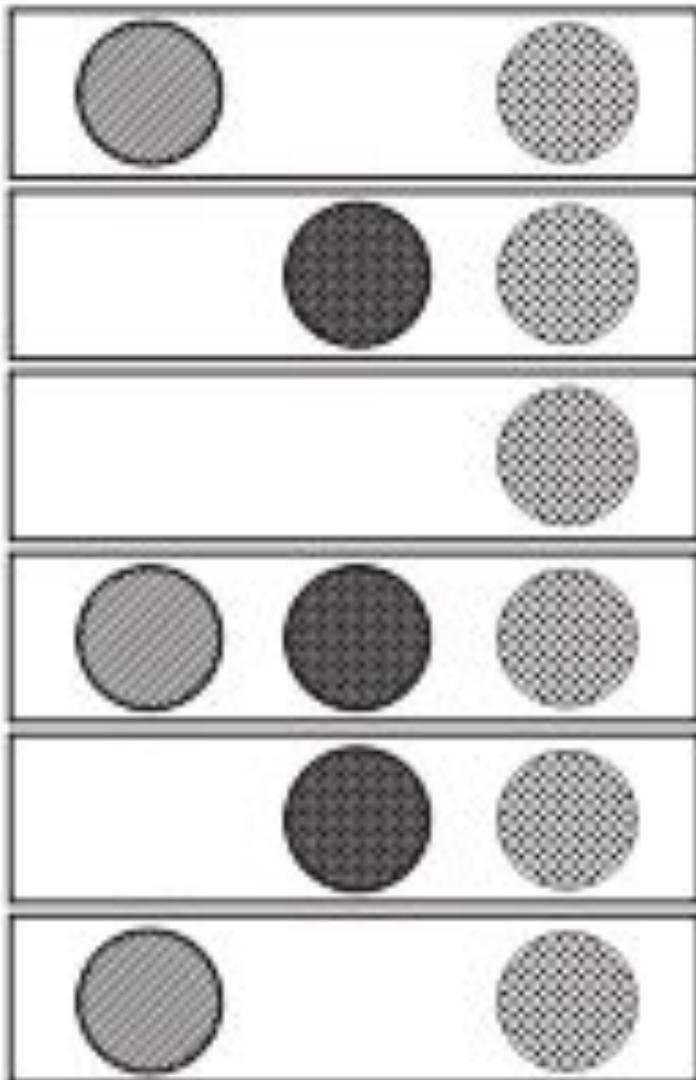


Preparing the data

Neural networks don't read streetlights.

- In the previous lesson, you learned about supervised algorithms.
 - You learned that they can take one dataset and turn it into another.
 - More important, they can take a dataset of what you know and turn it into a dataset of what you want to know.
 - How do you train a supervised neural network? You present it with two datasets and ask it to learn how to transform one into the other.
- 

What you know



What you want to know

STOP

WALK

STOP

WALK

WALK

STOP

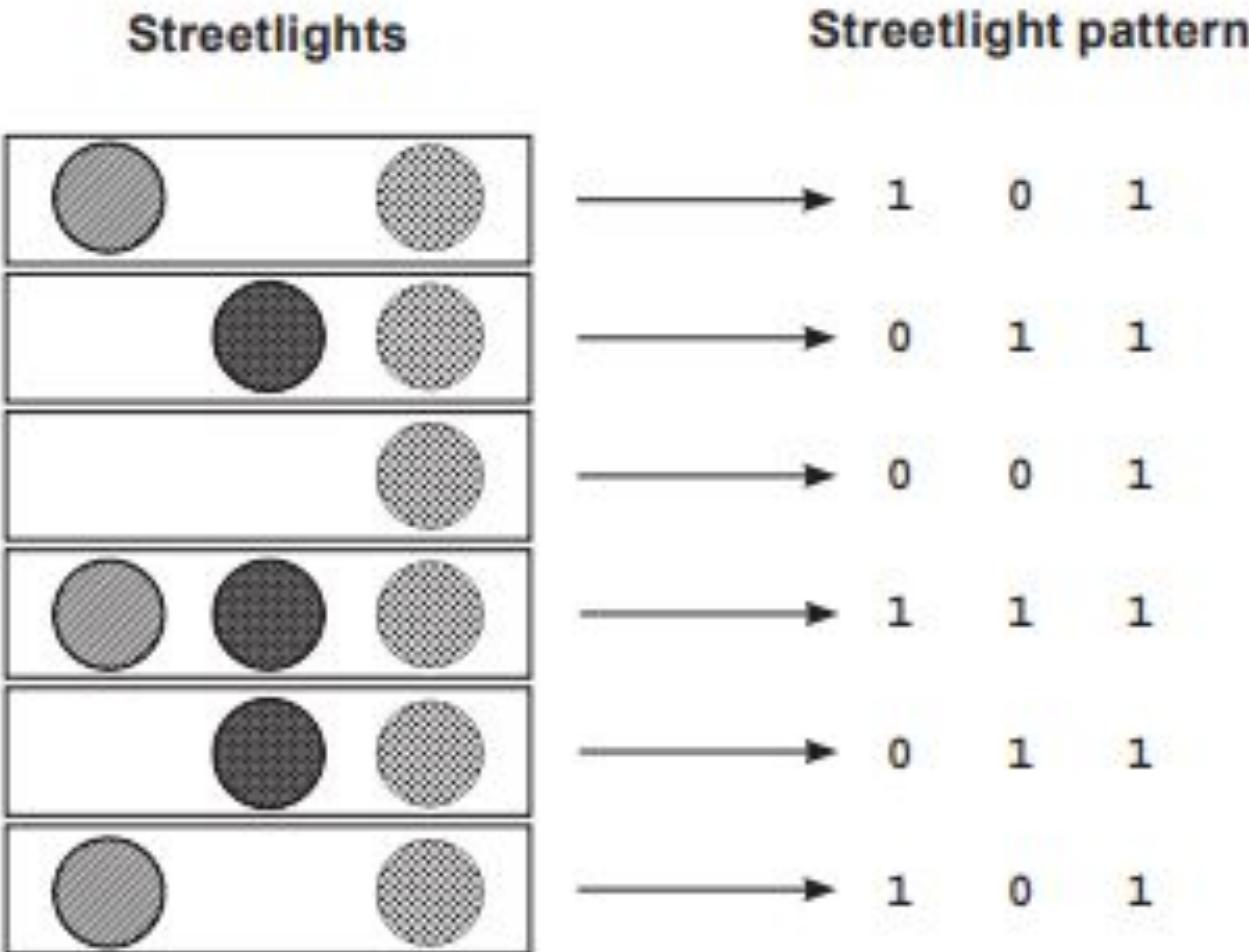


Matrices and the matrix relationship

Translate the streetlight into math.

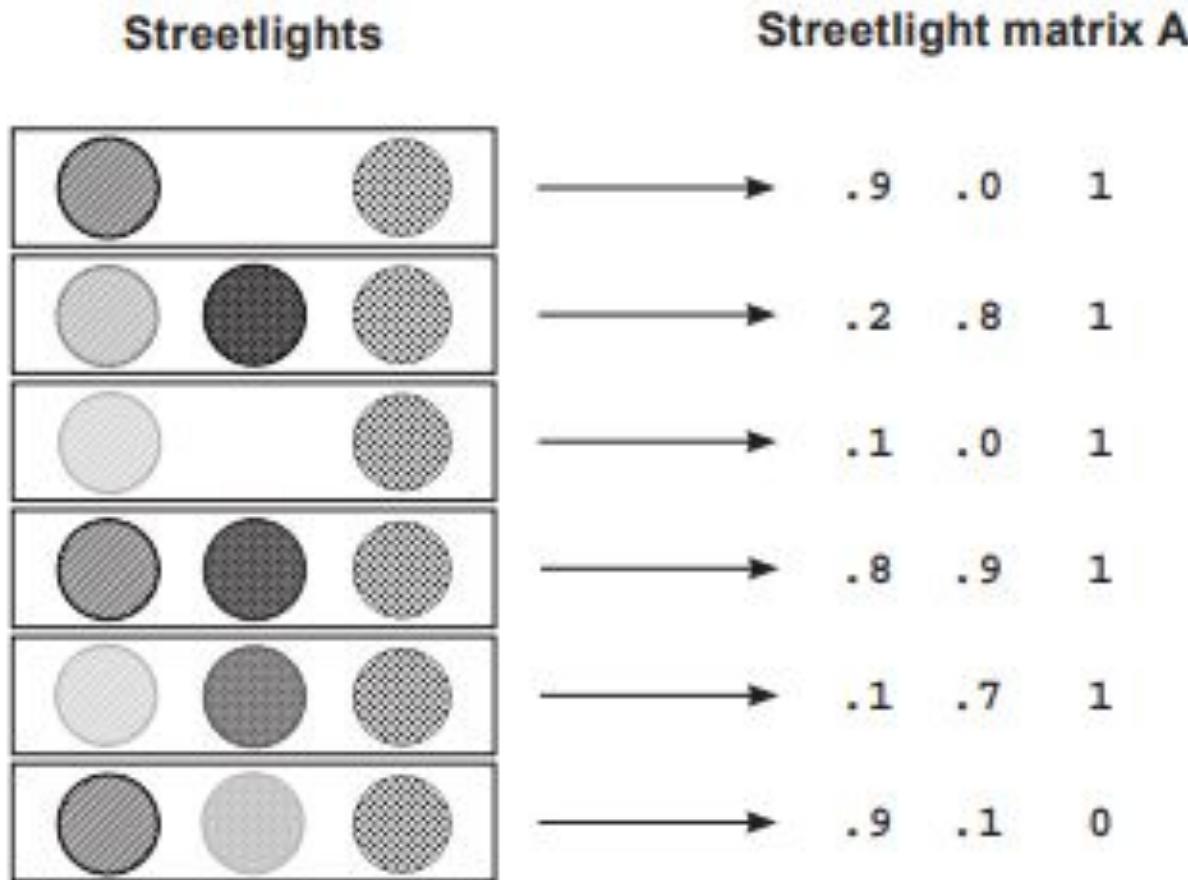
- Math doesn't understand streetlights.
- As mentioned in the previous section, you want to teach a neural network to translate a streetlight pattern into the correct stop/walk pattern.
- The operative word here is pattern.
- What you really want to do is mimic the pattern of the streetlight in the form of numbers.

Matrices and the matrix relationship

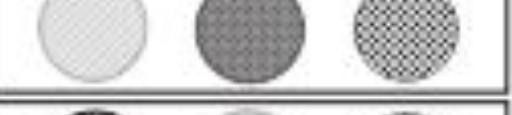


Matrices and the matrix relationship

Good data matrices perfectly mimic the outside world.



Matrices and the matrix relationship

Streetlights	Streetlight matrix B
	9 0 10
	2 8 10
	1 0 10
	8 9 10
	1 7 10
	9 1 0



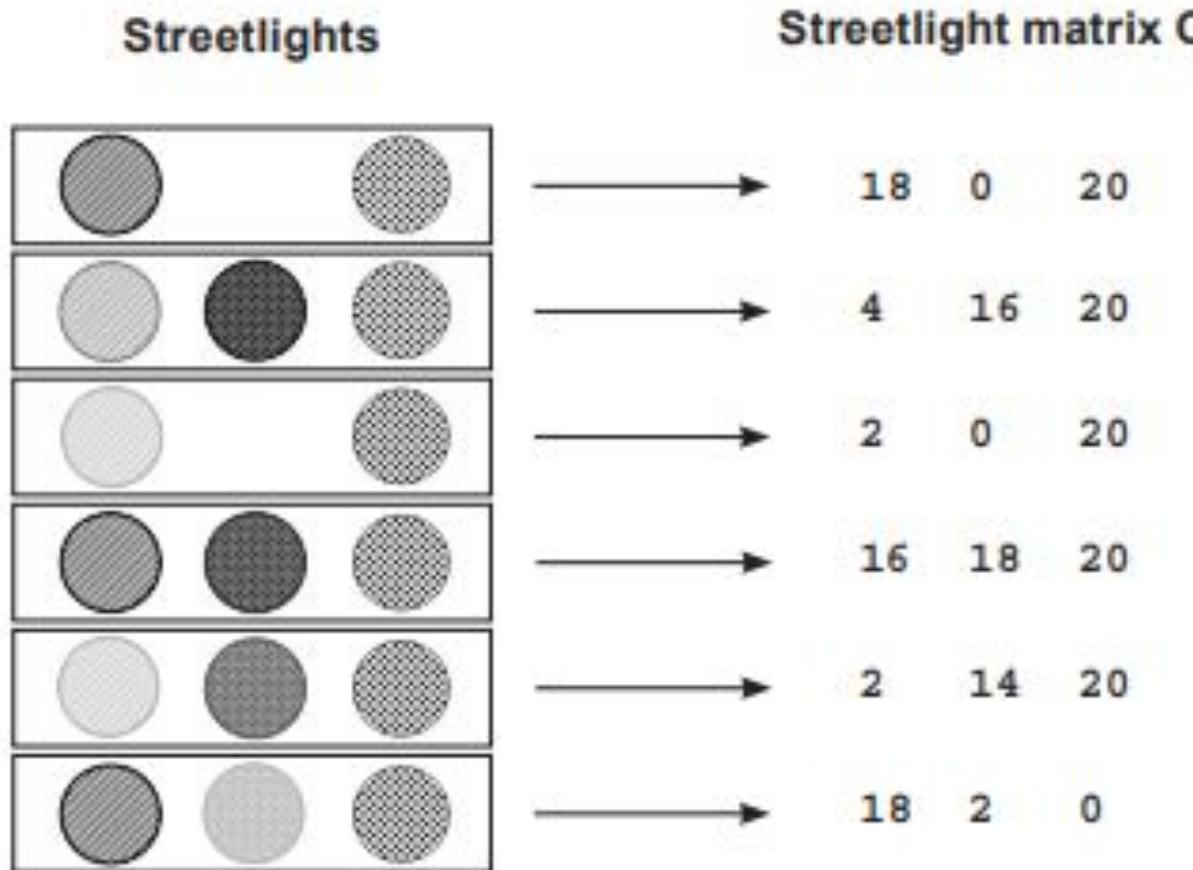
Matrices and the matrix relationship



- Matrix (B) is valid.
- It adequately captures the relationships between various training examples (rows) and lights (columns).
- Note that $\text{Matrix A} * 10 == \text{Matrix B}$ ($A * 10 == B$).
- This means these matrices are scalar multiples of each other.

Matrices and the matrix relationship

Matrices A and B both contain the same underlying pattern





Matrices and the matrix relationship



- It's important to recognize that the underlying pattern isn't the same as the matrix.
- It's a property of the matrix. In fact, it's a property of all three of these matrices (A, B, and C).
- The pattern is what each of these matrices is expressing.
- The pattern also existed in the streetlights.

Matrices and the matrix relationship

STOP → 0

WALK → 1

STOP → 0

WALK → 1

WALK → 1

STOP → 0



Creating a matrix or two in Python

Import the matrices into Python.

- You've converted the streetlight pattern into a matrix (one with just 1s and 0s).
- Now let's create that matrix (and, more important, its underlying pattern) in Python so the neural network can read it.
- Python's NumPy library (introduced in lesson 3) was built just for handling matrices.



Creating a matrix or two in Python



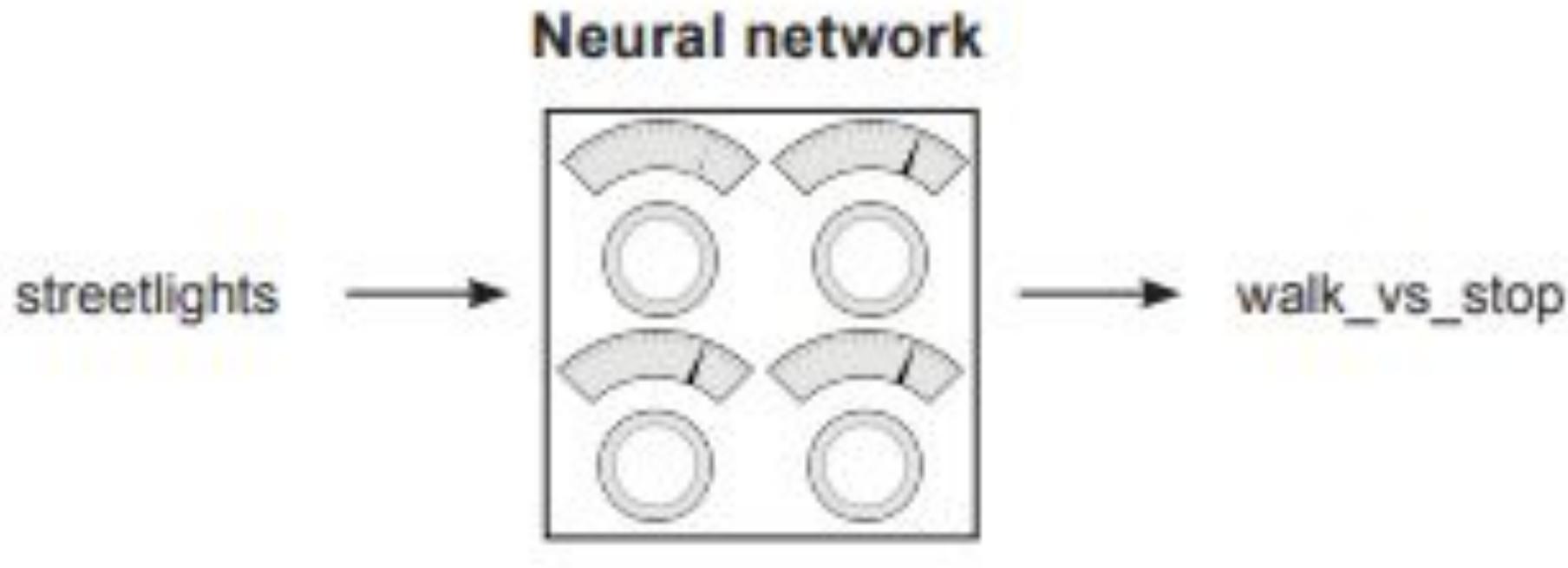
- Let's see it in action:

```
import numpy as np
streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )
```

Creating a matrix or two in Python

```
walk_vs_stop = np.array( [ [ 0 ],  
                           [ 1 ],  
                           [ 0 ],  
                           [ 1 ],  
                           [ 1 ],  
                           [ 0 ] ] )
```

Creating a matrix or two in Python



Building a neural network

```
import numpy as np
weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [ [ 1,  0,  1 ],
                          [ 0,  1,  1 ],
                          [ 0,  0,  1 ],
                          [ 1,  1,  1 ],
                          [ 0,  1,  1 ],
                          [ 1,  0,  1 ] ] )

walk_vs_stop = np.array( [ 0,  1,  0,  1,  1,  0 ] )

input = streetlights[0] ← [1,0,1]
goal_prediction = walk_vs_stop[0] ← Equals 0 (stop)

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

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

Building a neural network

Elementwise
multiplication

```
import numpy as np
```

```
a = np.array([0,1,2,1])  
b = np.array([2,2,2,3])
```

```
print(a*b)  
print(a+b)  
print(a * 0.5)  
print(a + 0.5)
```

Elementwise
addition

Vector-scalar
multiplication

Vector-scalar
addition

NumPy makes these operations easy. When you put a `+` between two vectors, it does what you expect: it adds the two vectors together. Other than these nice NumPy operators and the new dataset, the neural network shown here is the same as the ones built previously.



Learning the whole dataset

```
import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1,  0,  1 ],
                         [ 0,  1,  1 ],
                         [ 0,  0,  1 ],
                         [ 1,  1,  1 ],
                         [ 0,  1,  1 ],
                         [ 1,  0,  1 ] ] )

walk_vs_stop = np.array( [ 0,  1,  0,  1,  1,  0 ] )

input = streetlights[0]           ← [1,0,1]
goal_prediction = walk_vs_stop[0] ← Equals 0 (stop)

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]

        prediction = input.dot(weights)

        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error

        delta = prediction - goal_prediction
        weights = weights - (alpha * (input * delta))
        print("Prediction:" + str(prediction))
        print("Error:" + str(error_for_all_lights) + "\n")

Error:2.6561231104
Error:0.962870177672
...
Error:0.000614343567483
Error:0.000533736773285
```



Full, batch, and stochastic gradient descent

Stochastic gradient descent updates weights one example at a time.

- As it turns out, this idea of learning one example at a time is a variant on gradient descent called stochastic gradient descent, and it's one of the handful of methods that can be used to learn an entire dataset.
- How does stochastic gradient descent work?
- As you saw in the previous example, it performs a prediction and weight update for each training example separately.

Full, batch, and stochastic gradient descent

(Full) gradient descent updates weights one dataset at a time.

- As introduced in lesson 4, another method for learning an entire dataset is gradient descent (or average/full gradient descent).
- Instead of updating the weights once for each training example, the network calculates the average weight_delta over the entire dataset, changing the weights only each time it computes a full average.

Full, batch, and stochastic gradient descent

Batch gradient descent updates weights after n examples.

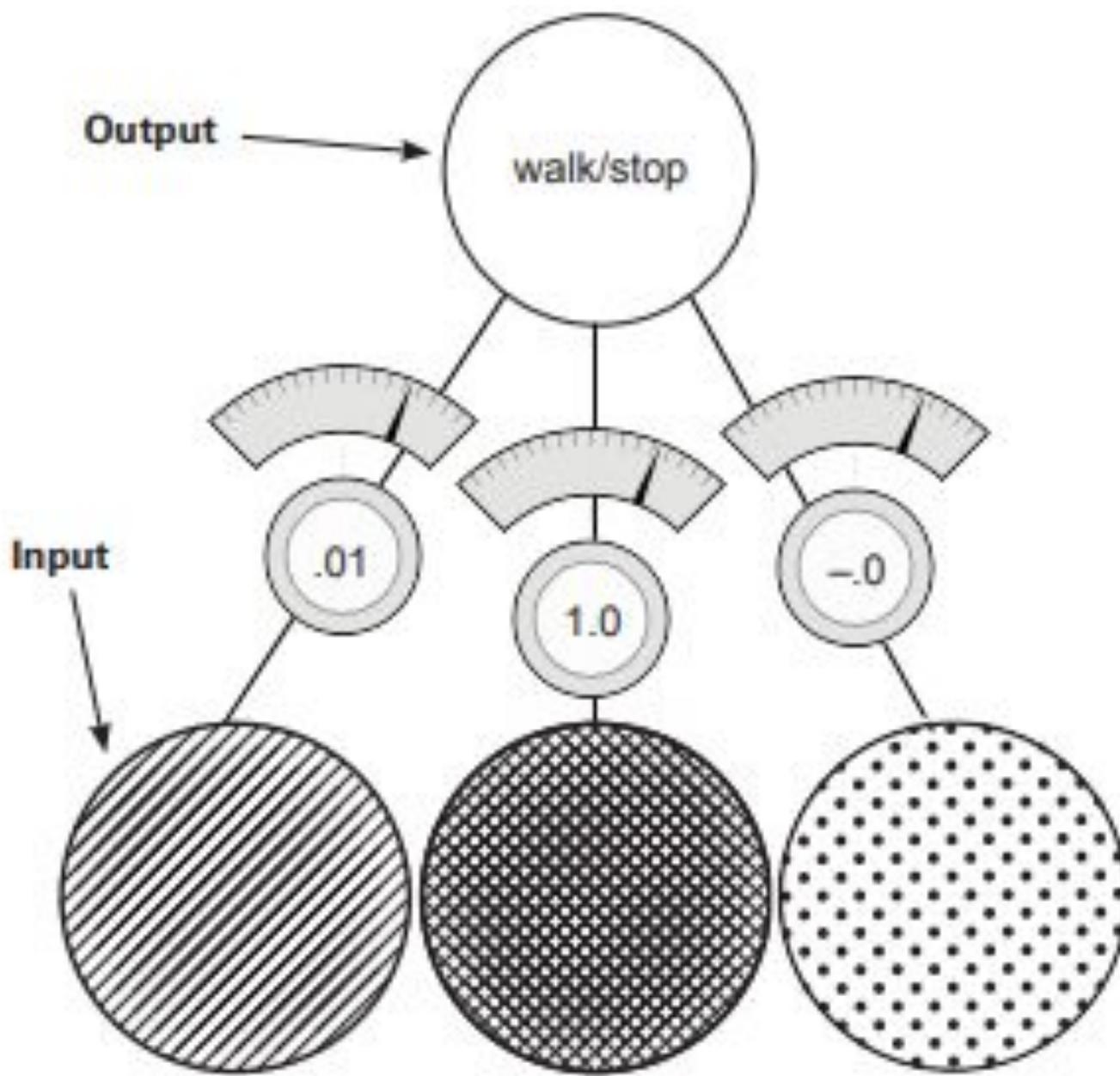
- This will be covered in more detail later, but there's also a third configuration that sort of splits the difference between stochastic gradient descent and full gradient descent.
- Instead of updating the weights after just one example or after the entire dataset of examples, you choose a batch size (typically between 8 and 256) of examples, after which the weights are updated.



Neural networks learn correlation

What did the last neural network learn?

- You just got done training a single-layer neural network to take a streetlight pattern and identify whether it was safe to cross the street.
 - Let's take on the neural network's perspective for a moment.
 - The neural network doesn't know that it was processing streetlight data.
 - All it was trying to do was identify which input (of the three possible) correlated with the output.
- 





Up and down pressure



It comes from the data.

- Each node is individually trying to correctly predict the output given the input.
- For the most part, each node ignores all the other nodes when attempting to do so.
- The only cross communication occurs in that all three weights must share the same error measure.
- The weight update is nothing more than taking this shared error measure and multiplying it by each respective input.

Up and down pressure

Training data				Weight pressure				
1	0	1	→	0	-	0	-	0
0	1	1	→	1	0	+	+	1
0	0	1	→	0	0	0	-	0
1	1	1	→	1	+	+	+	1
0	1	1	→	1	0	+	+	1
1	0	1	→	0	-	0	-	0

Up and down pressure

Training data

1	0	1	0
0	1	1	1
0	0	1	0
1	1	1	1
0	1	1	1
1	0	1	0

Weight pressure

-	0	-	0
0	+	+	1
0	0	-	0
+	+	+	1
0	+	+	1
-	0	-	0



Up and down pressure

- The mathematician in you may be cringing a little.
- Upward pressure and downward pressure are hardly precise mathematical expressions, and they have plenty of edge cases where this logic doesn't hold (which we'll address in a second).



Edge case: Overfitting

Sometimes correlation happens accidentally.

- Consider again the first example in the training data. What if the far-left weight was 0.5 and the far-right weight was -0.5 ? Their prediction would equal 0.
 - The network would predict perfectly.
 - But it hasn't remotely learned how to safely predict streetlights (those weights would fail in the real world).
 - This phenomenon is known as overfitting.



Edge case: Overfitting



- If you train on only two streetlights and the network finds just these edge-case configurations, it could fail to tell you whether it's safe to cross the street when it sees a streetlight that wasn't in the training data.



Edge case: Conflicting pressure

Sometimes correlation fights itself.

- Consider the far-right column in the following Weight Pressure table.
- What do you see? This column seems to have an equal number of upward and downward pressure moments.
- But the network correctly pushes this (far-right) weight down to 0, which means the downward pressure moments must be larger than the upward ones. How does this work?

Training data

1	0	1	0
0	1	1	1
0	0	1	0
1	1	1	1
0	1	1	1
1	0	1	0

Weight pressure

-	0	-	0
0	+	+	1
0	0	-	0
+	+	+	1
0	+	+	1
-	0	-	0



Edge case: Conflicting pressure

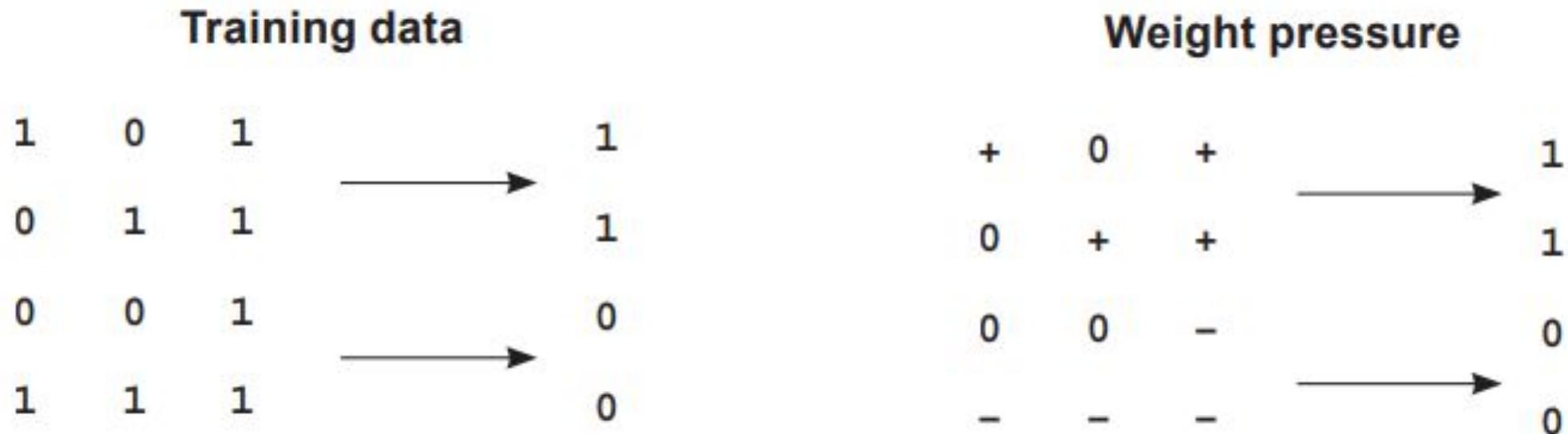
It doesn't always work out like this.

- In some ways, you kind of got lucky.
 - If the middle node hadn't been so perfectly correlated, the network might have struggled to silence the far-right weight.
 - Later you'll learn about regularization, which forces weights with conflicting pressure to move toward 0.
 - As a preview, regularization is advantageous because if a weight has equal pressure upward and downward, it isn't good for anything
- 



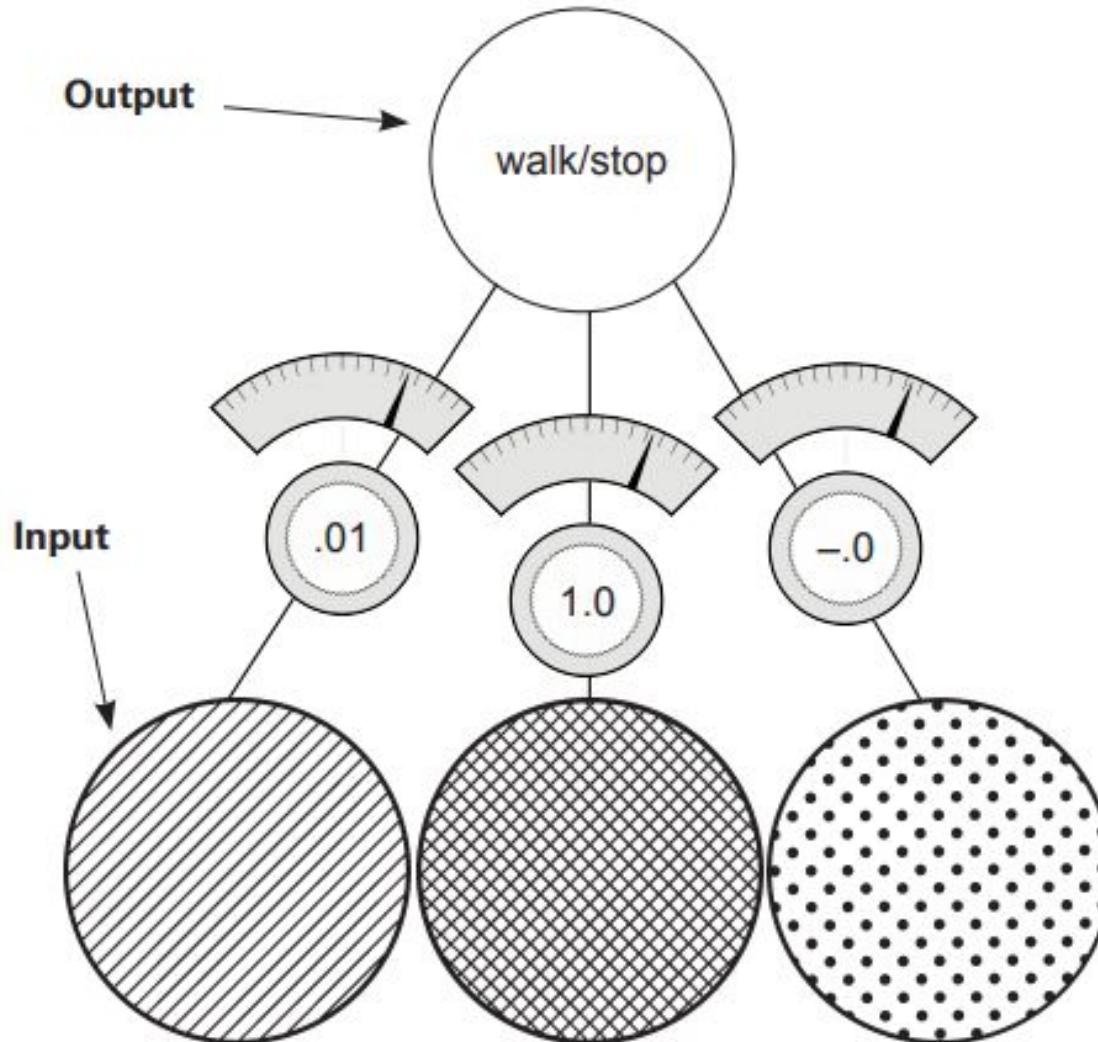
Edge case: Conflicting pressure

- If networks look for correlation between an input column of data and the output column, what would the neural network do with the following dataset?

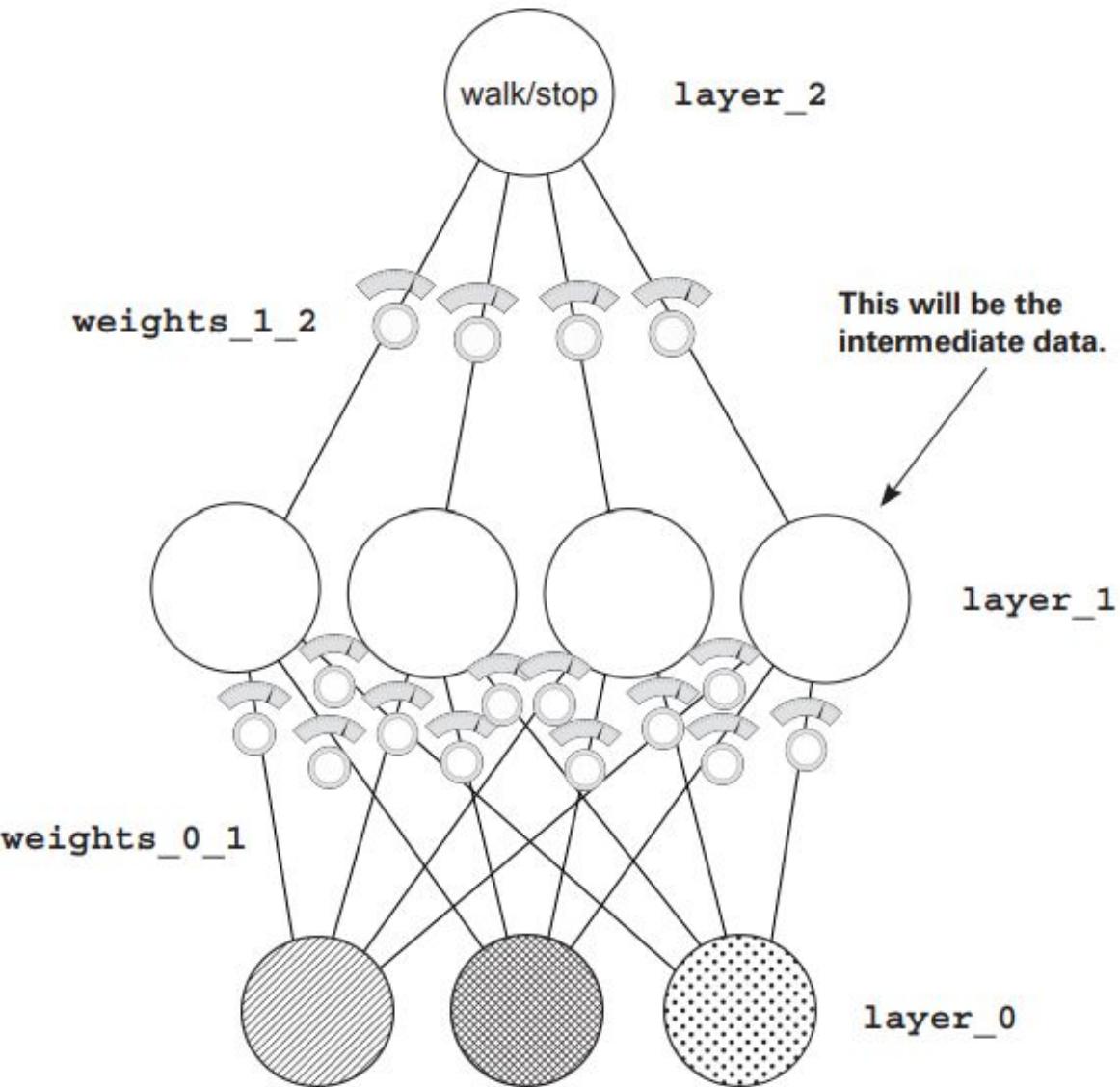


Learning indirect correlation

If your data doesn't have correlation, create intermediate data that

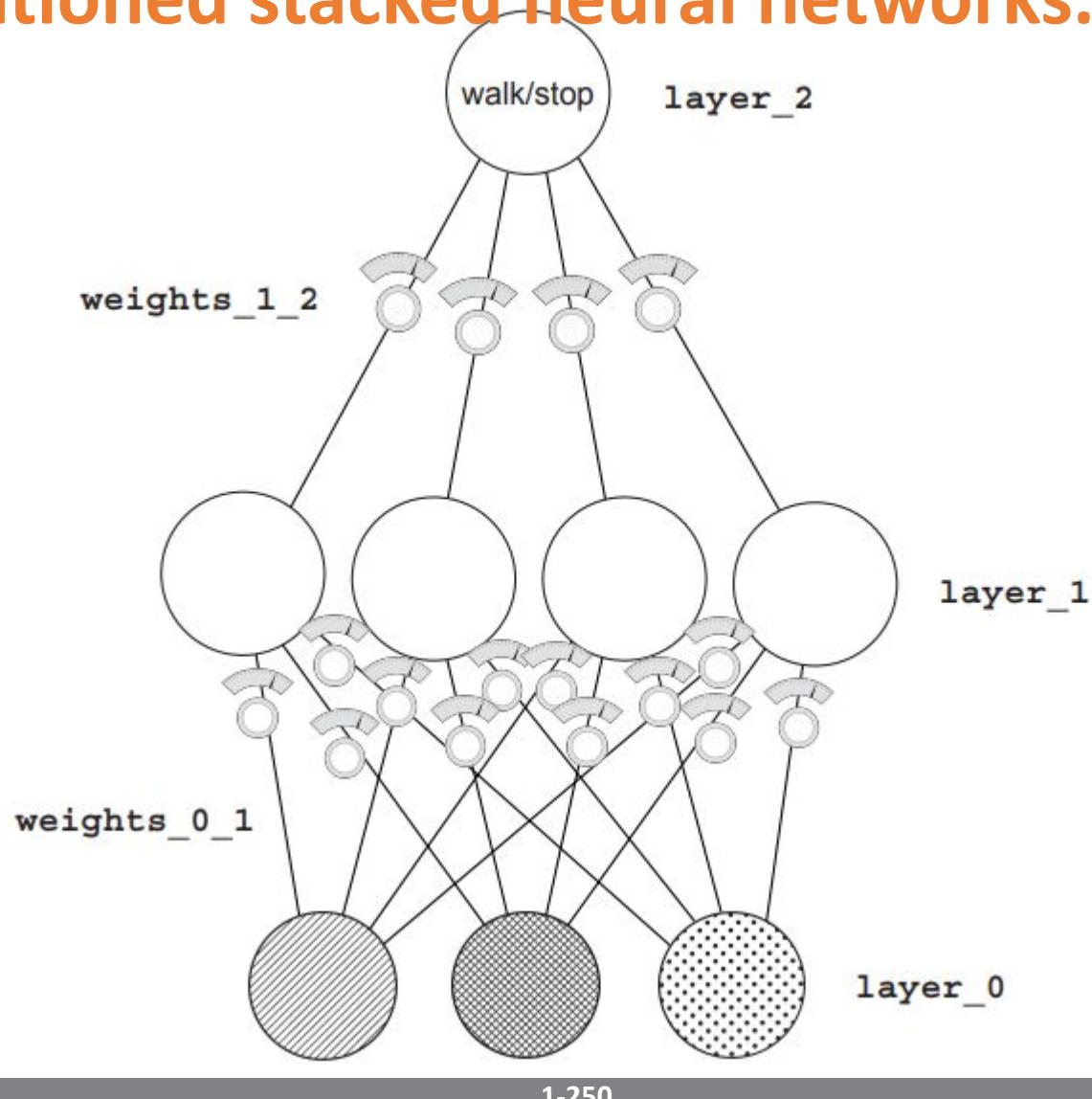


Creating correlation



Stacking neural networks: A review

briefly mentioned stacked neural networks. Let's review.



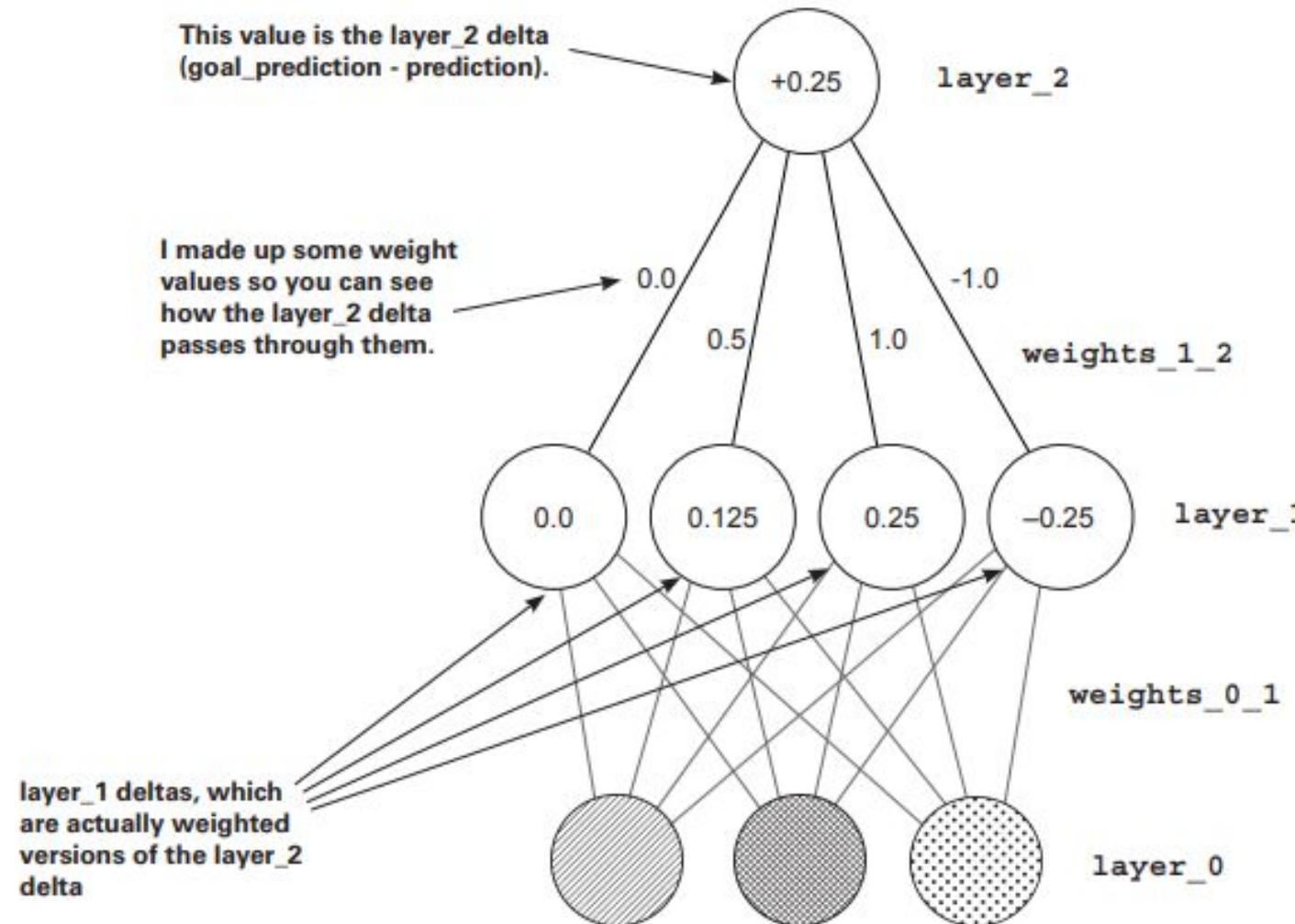


Stacking neural networks: A review

- As you start to think about how this neural network learns, you already know a great deal.
 - If you ignore the lower weights and consider their output to be the training set, the top half of the neural network (layer_1 to layer_2) is just like the networks trained in the preceding lesson.
 - You can use all the same learning logic to help them learn.
 - The part that you don't yet understand is how to update the weights between layer_0 and layer_1.
- 

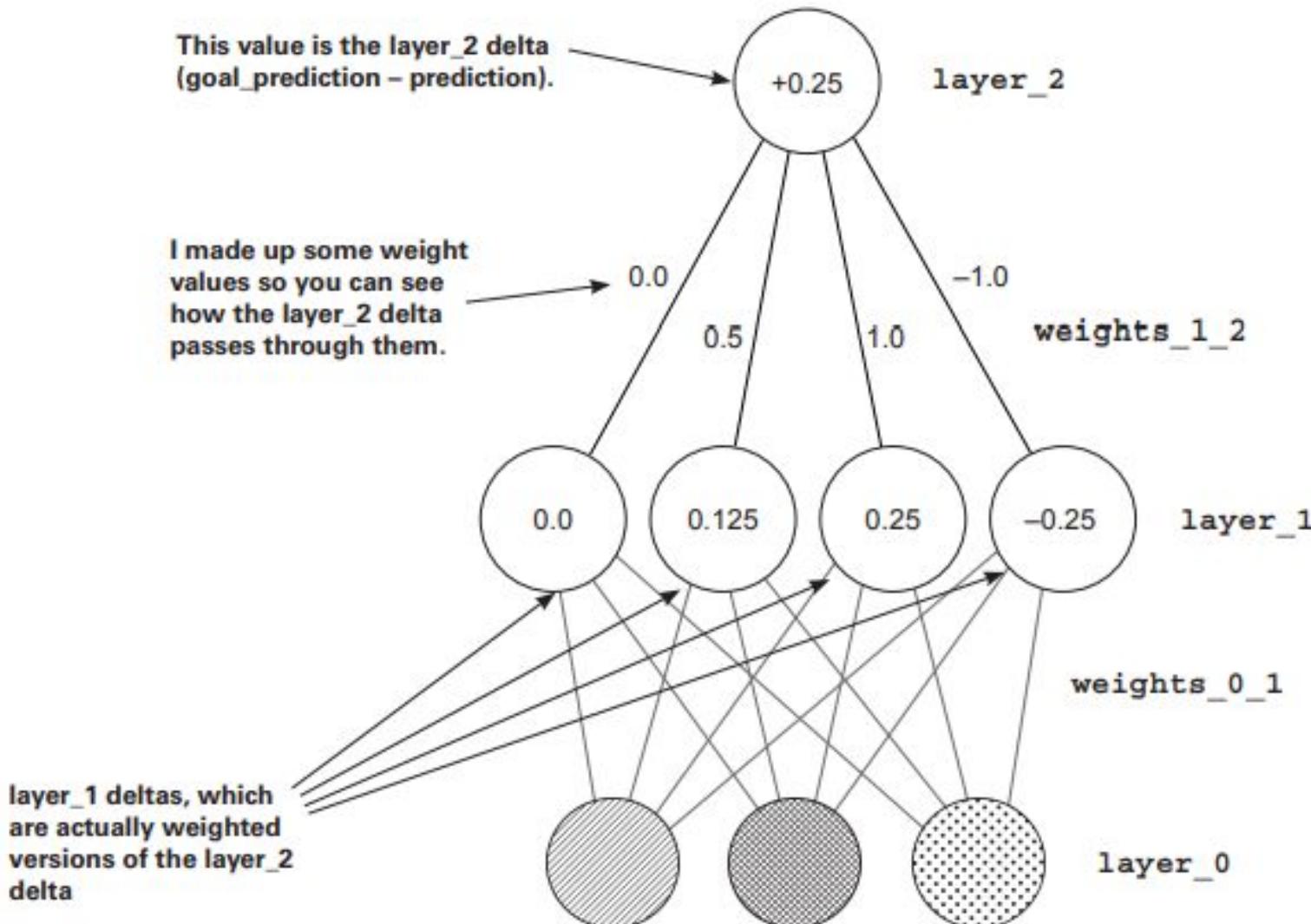
Backpropagation: Long-distance error attribution

The weighted average error



Backpropagation: Why does this work?

The weighted average delta



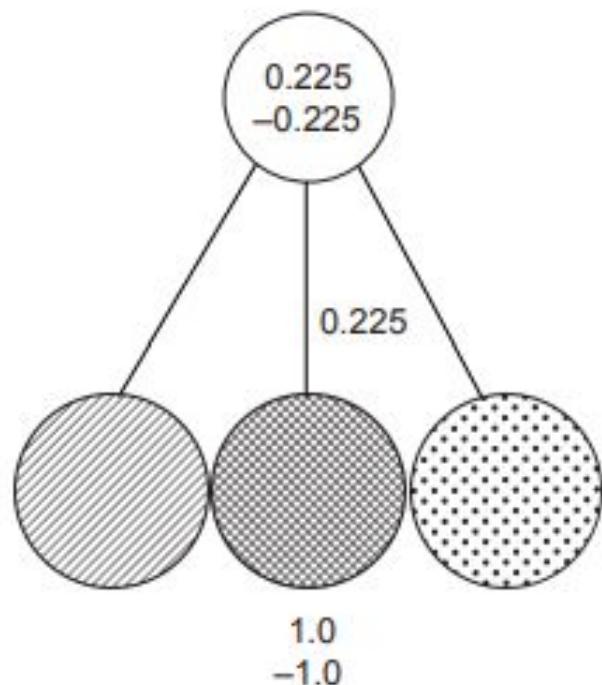
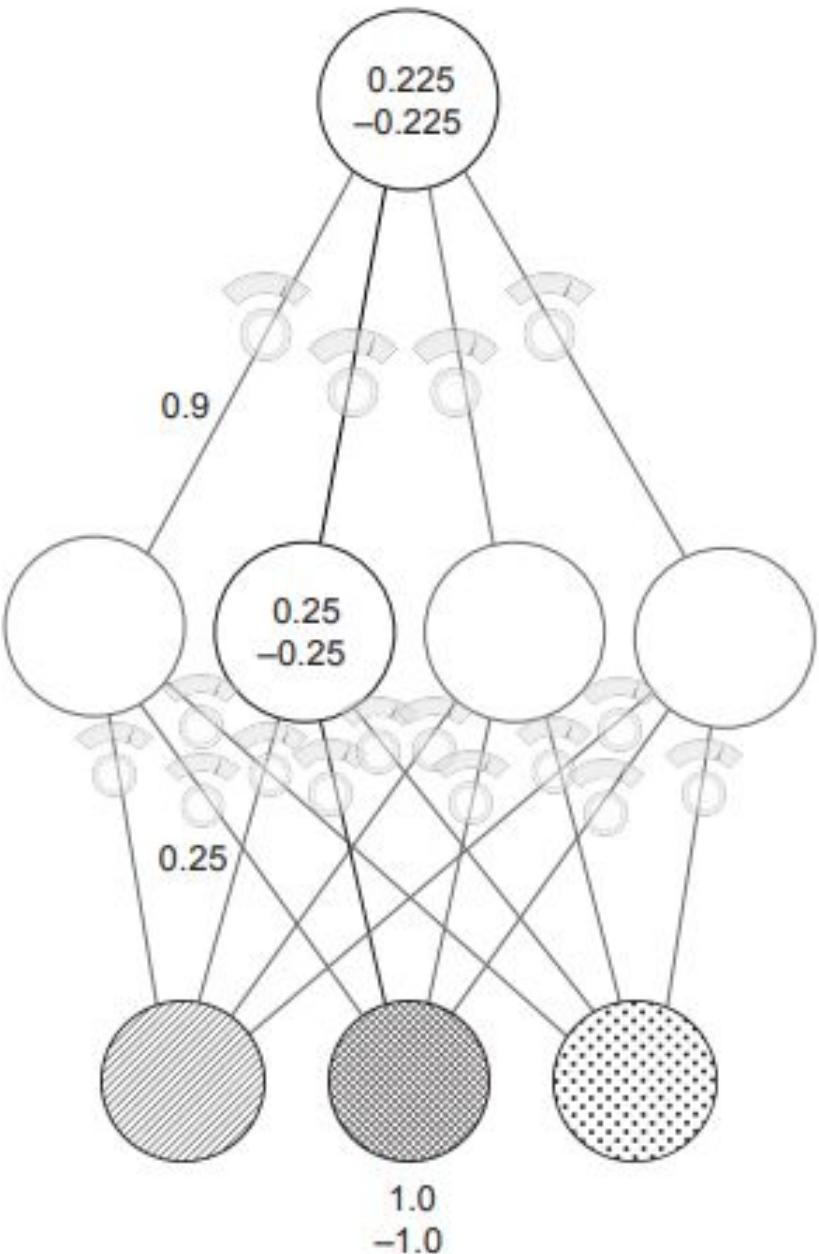


Linear vs. nonlinear

- I'm going to show you a phenomenon. As it turns out, you need one more piece to make this neural network train.
- Let's take it from two perspectives. The first will show why the neural network can't train without it.
- In other words, first I'll show you why the neural network is currently broken.
- Then, once you add this piece, I'll show you what it does to fix this problem. For now, check out this simple algebra:

$$\begin{aligned} 1 * 10 * 2 &= 100 \\ 5 * 20 &= 100 \end{aligned}$$

$$\begin{aligned} 1 * 0.25 * 0.9 &= 0.225 \\ 1 * 0.225 &= 0.225 \end{aligned}$$



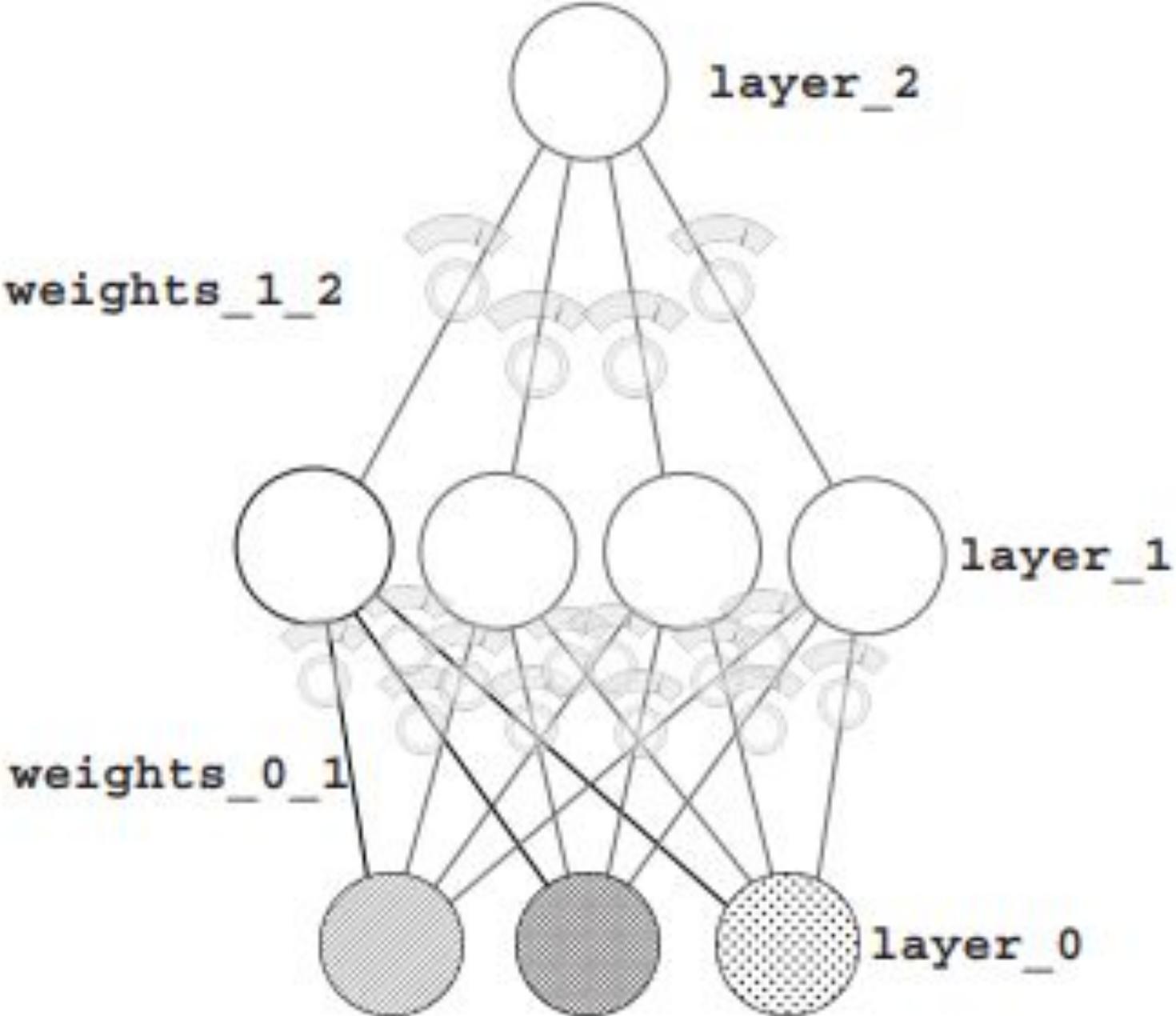
These two graphs show two training examples each, one where the input is 1.0 and another where the input is -1.0. The bottom line: *for any three-layer network you create, there's a two-layer network that has identical behavior.* Stacking two neural nets (as you know them at the moment) doesn't give you any more power. Two consecutive weighted sums is just a more expensive version of one weighted sum.



Why the neural network still doesn't work

If you trained the three-layer network as it is now, it wouldn't converge.

- Let's talk about the middle layer (layer_1) before it's fixed. Right now, each node (out of the four) has a weight coming to it from each of the inputs.
- Let's think about this from a correlation standpoint.
- Each node in the middle layer subscribes to a certain amount of correlation with each input node.
- If the weight from an input to the middle layer is 1.0, then it subscribes to exactly 100% of that node's movement.





The secret to sometimes correlation

Turn off the node when the value would be below 0.

- This might seem too simple to work, but consider this: if a node's value dropped below 0, normally the node would still have the same correlation to the input as always.
- It would just happen to be negative in value.
- But if you turn off the node (setting it to 0) when it would be negative, then it has zero correlation to any inputs whenever it's negative.



A quick break



That last part probably felt a little abstract, and that's totally OK.

- Neural networks look for correlation between input and output, and you no longer have to worry about how that happens.
 - You just know it does. Now we're building on that idea.
 - Let yourself relax and trust the things you've already learned.
- 

Your first deep neural network

Here's how to make the prediction.

The following code initializes the weights and makes a forward propagation. New code is **bold**.

```
import numpy as np
np.random.seed(1)

def relu(x):
    return (x > 0) * x

alpha = 0.2
hidden_size = 4

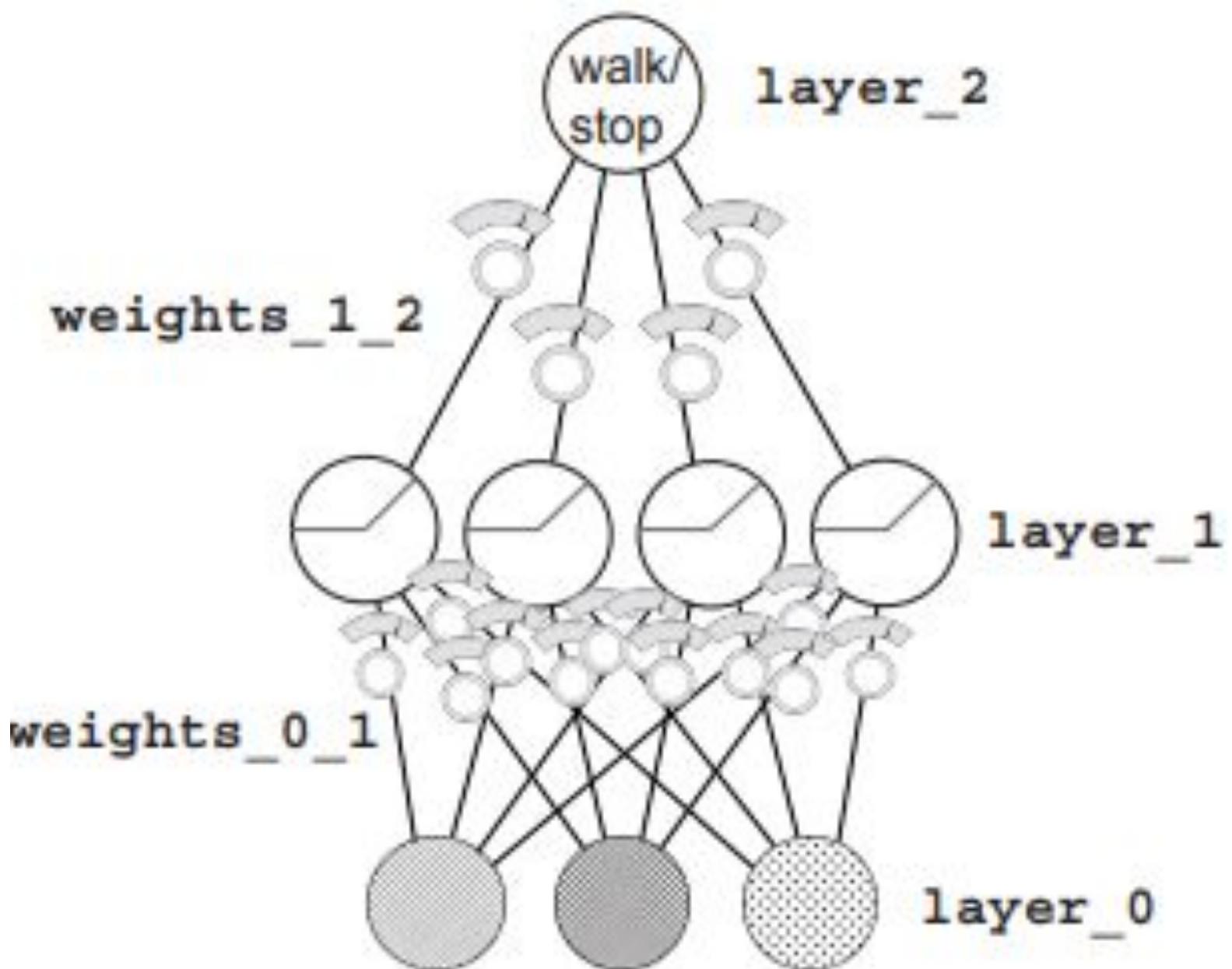
streetlights = np.array( [[ 1,  0,  1 ],
                         [ 0,  1,  1 ],
                         [ 0,  0,  1 ],
                         [ 1,  1,  1 ] ] )

walk_vs_stop = np.array([[ 1,  1,  0,  0]]).T

weights_0_1 = 2*np.random.random((3,hidden_size)) - 1
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1

layer_0 = streetlights[0]
layer_1 = relu(np.dot(layer_0,weights_0_1))
layer_2 = np.dot(layer_1,weights_1_2)

This function sets all negative numbers to 0.
Two sets of weights now to connect the three layers (randomly initialized)
The output of layer_1 is sent through relu, where negative values become 0. This is the input for the next layer, layer_2.
```



Backpropagation in code

You can learn the amount that each weight contributes to the final

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x

def relu2deriv(output):
    return output>0

alpha = 0.2
hidden_size = 4

weights_0_1 = 2*np.random.random((3,hidden_size)) - 1
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1

for iteration in range(60):
    layer_2_error = 0
    for i in range(len(streetlights)):
        layer_0 = streetlights[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)

        layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2)

        layer_2_delta = (walk_vs_stop[i:i+1] - layer_2)
        layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)

        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    if(iteration % 10 == 9):
        print("Error:" + str(layer_2_error))
```

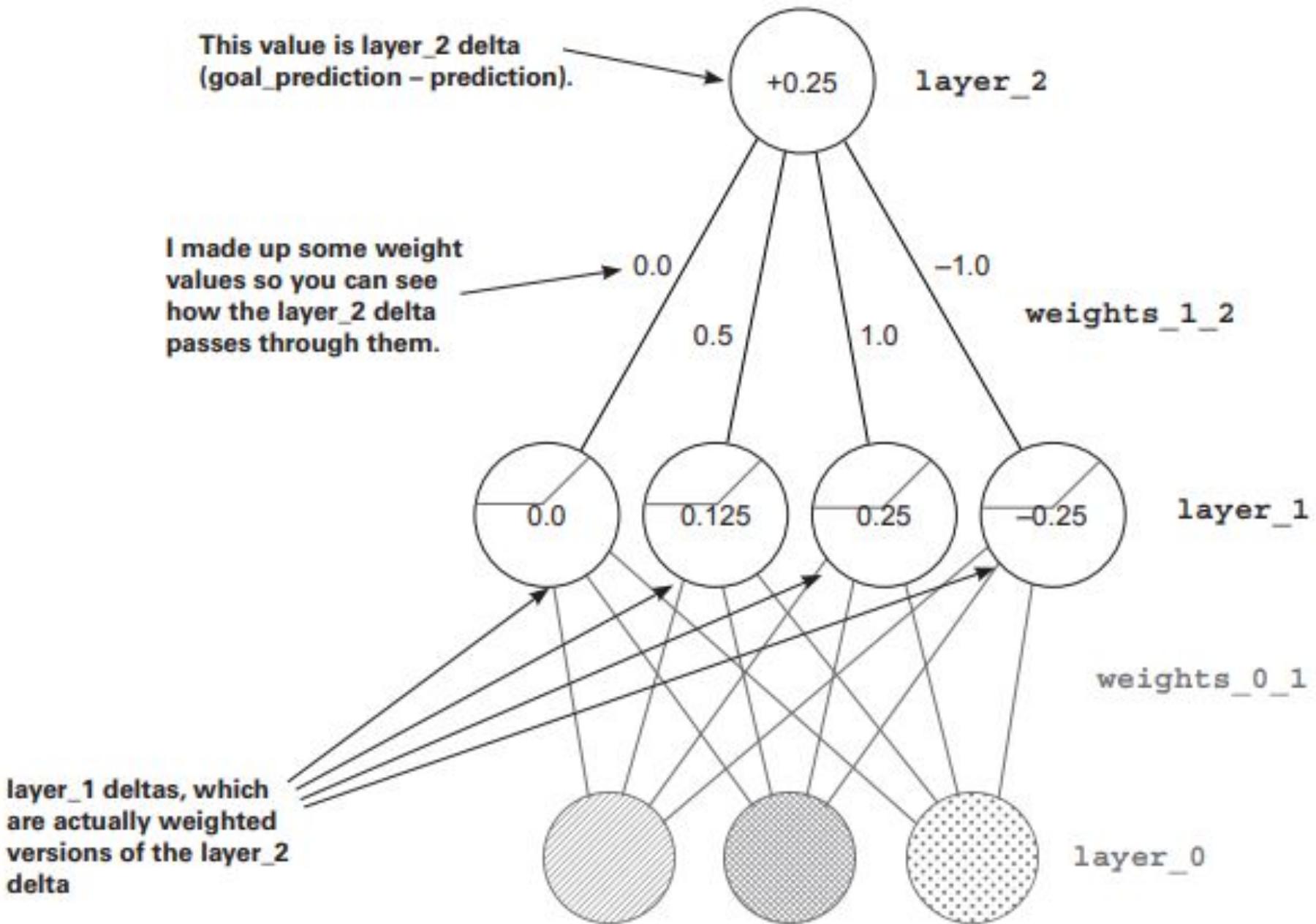
Returns x if x > 0;
returns 0 otherwise

Returns 1 for input > 0;
returns 0 otherwise

This line computes the
delta at layer_1 given
the delta at layer_2
by taking the layer_2_
delta and multiplying
it by its connecting
weights_1_2.



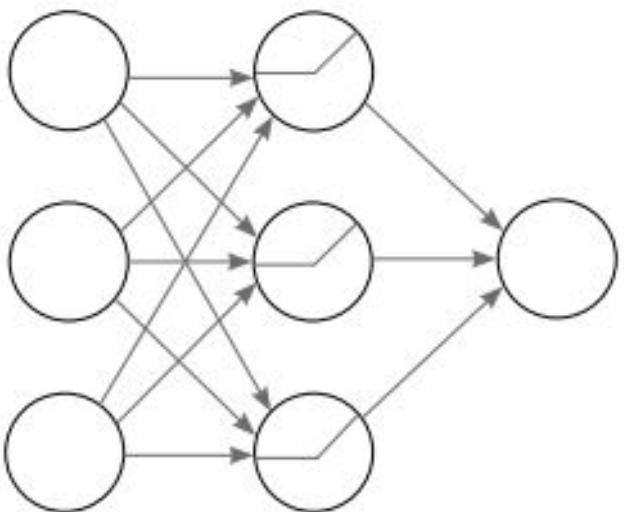
This value is layer_2 delta
(goal_prediction - prediction).



One iteration of backpropagation

① Initializing the network's weights and data

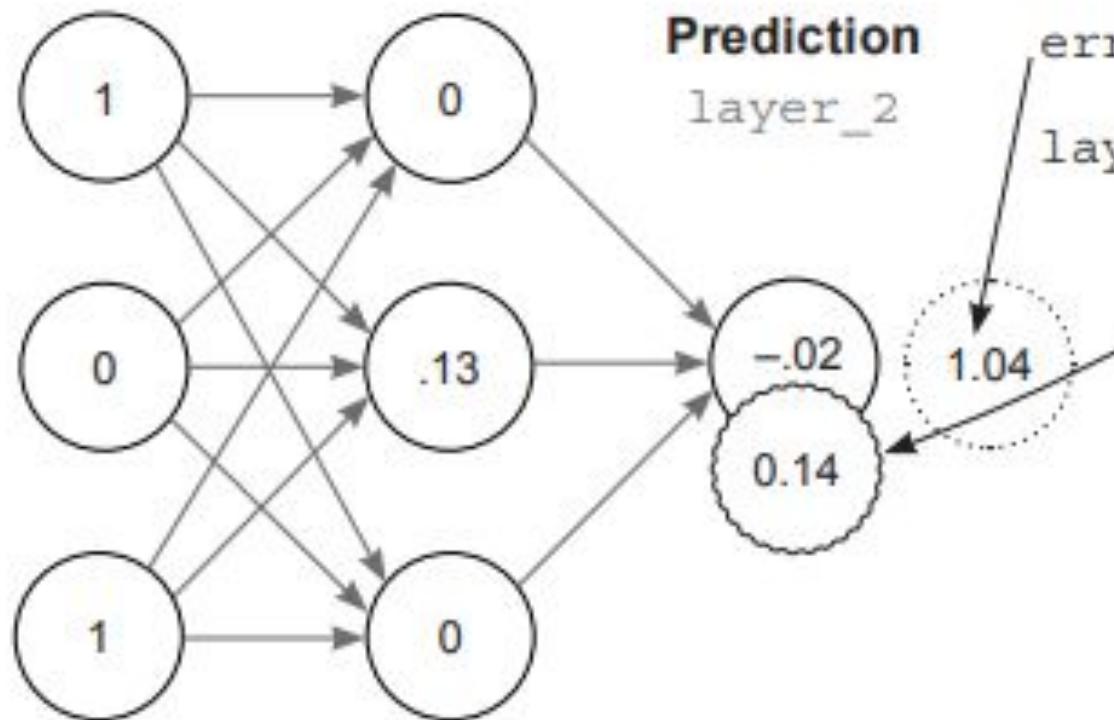
Inputs Hiddens Prediction



```
import numpy as np  
np.random.seed(1)  
  
def relu(x):  
    return (x > 0) * x  
  
def relu2deriv(output):  
    return output>0  
  
lights = np.array([[ 1,  0,  1 ],  
                  [ 0,  1,  1 ],  
                  [ 0,  0,  1 ],  
                  [ 1,  1,  1 ] ] )  
  
walk_stop = np.array([[ 1,  1,  0,  0]]).T  
  
alpha = 0.2  
hidden_size = 3  
  
weights_0_1 = 2*np.random.random(\n            (3,hidden_size)) - 1  
weights_1_2 = 2*np.random.random(\n            (hidden_size,1)) - 1
```

② PREDICT + COMPARE: Making a prediction, and calculating the output error and delta

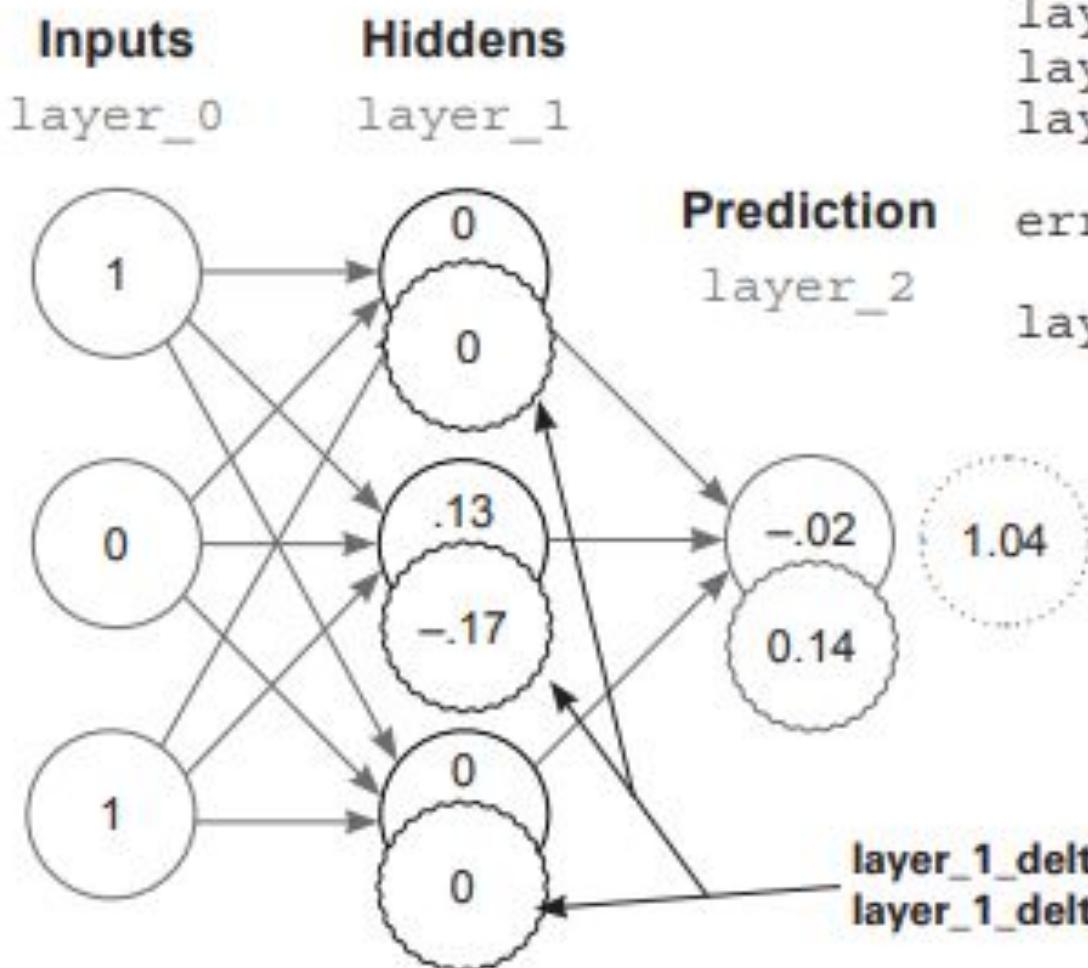
Inputs **Hiddens**
layer_0 layer_1



```
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)
```

```
error = (layer_2-walk_stop[0:1])**2
layer_2_delta=(layer_2-walk_stop[0:1])
```

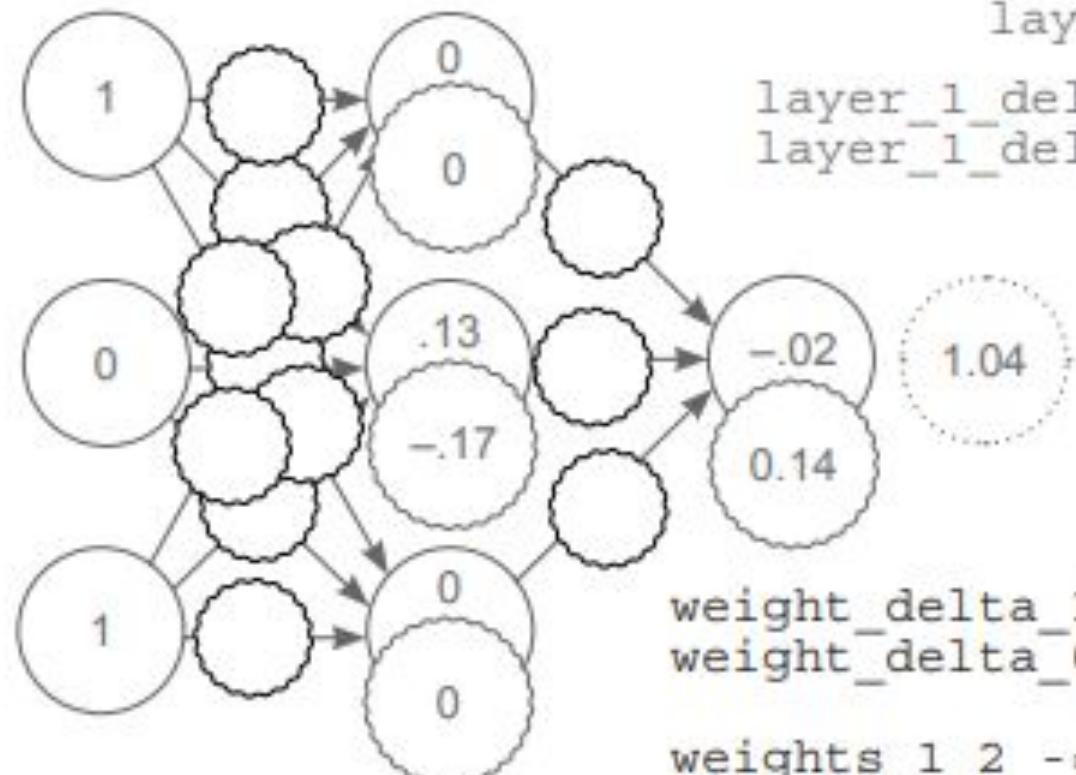
③ LEARN: Backpropagating from layer_2 to layer_1



④ LEARN: Generating weight_deltas, and updating weights

Inputs **Hiddens** **Prediction**

layer_0 layer_1 layer_2



```
layer_0 = lights[0:1]
layer_1 = np.dot(layer_0,weights_0_1)
layer_1 = relu(layer_1)
layer_2 = np.dot(layer_1,weights_1_2)
error = (layer_2-walk_stop[0:1])**2
layer_2_delta=(layer_2-walk_stop[0:1])
```

```
layer_1_delta=layer_2_delta.dot(weights_1_2.T)
layer_1_delta *= relu2deriv(layer_1)
```

```
weight_delta_1_2 = layer_1.T.dot(layer_2_delta)
weight_delta_0_1 = layer_0.T.dot(layer_1_delta)
```

```
weights_1_2 -= alpha * weight_delta_1_2
weights_0_1 -= alpha * weight_delta_0_1
```

Putting it all together

Here's the self-sufficient program you should be able to run (runtime output follows).

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x > 0) * x

def relu2deriv(output):
    return output>0

streetlights = np.array( [[ 1,  0,  1 ],
                         [ 0,  1,  1 ],
                         [ 0,  0,  1 ],
                         [ 1,  1,  1 ] ] )

walk_vs_stop = np.array([[ 1,  1,  0,  0]]).T

alpha = 0.2
hidden_size = 4

weights_0_1 = 2*np.random.random((3,hidden_size)) - 1
weights_1_2 = 2*np.random.random((hidden_size,1)) - 1
```

Returns **x if $x > 0$;
returns 0 otherwise**

Returns **1 for input > 0 ;
returns 0 otherwise**

```
for iteration in range(60):
    layer_2_error = 0
    for i in range(len(streetlights)):
        layer_0 = streetlights[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)

        layer_2_error += np.sum((layer_2 - walk_vs_stop[i:i+1]) ** 2)

    layer_2_delta = (layer_2 - walk_vs_stop[i:i+1])
    layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)

    weights_1_2 -= alpha * layer_1.T.dot(layer_2_delta)
    weights_0_1 -= alpha * layer_0.T.dot(layer_1_delta)

    if(iteration % 10 == 9):
        print("Error:" + str(layer_2_error))
```

```
Error:0.634231159844
Error:0.358384076763
Error:0.0830183113303
Error:0.0064670549571
Error:0.000329266900075
Error:1.50556226651e-05
```



Why do deep networks matter?

What's the point of creating “intermediate datasets” that have correlation? .

- Consider the cat picture shown here. Consider further that I had a dataset of images with cats and without cats (and I labeled them as such).
- If I wanted to train a neural network to take the pixel values and predict whether there's a cat in the picture, the two-layer network might have a problem



Why do deep networks matter?

- Just as in the last streetlight dataset, no individual pixel correlates with whether there's a cat in the picture.
- Only different configurations of pixels correlate with whether there's a cat.



Lesson 7 how to picture neural networks: in your head and on paper





How to picture neural networks:



In this lesson

- Correlation summarization
- Simplified visualization
- Seeing the network predict
- Visualizing using letters instead of pictures
- Linking variables
- The importance of visualization tools



It's time to simplify

**It's impractical to think about everything all the time.
Mental tools can help.**

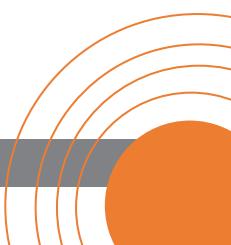
- In this lesson, this construction of efficient concepts in your mind is exactly what I want to talk about.
- Even though it's not an architecture or experiment, it's perhaps the most important value I can give you.
- In this case, I want to show how I summarize all the little lessons in an efficient way in my mind so that I can do things like build new architectures, debug experiments, and use an architecture on new problems and new datasets



It's time to simplify



Let's start by reviewing the concepts you've learned so far.

- This course began with small lessons and then built layers of abstraction on top of them.
 - We began by talking about the ideas behind machine learning in general.
 - Then we progressed to how individual linear nodes (or neurons) learned, followed by horizontal groups of neurons (layers) and then vertical groups (stacks of layers).
- 



Correlation summarization

This is the key to sanely moving forward to more advanced neural networks.

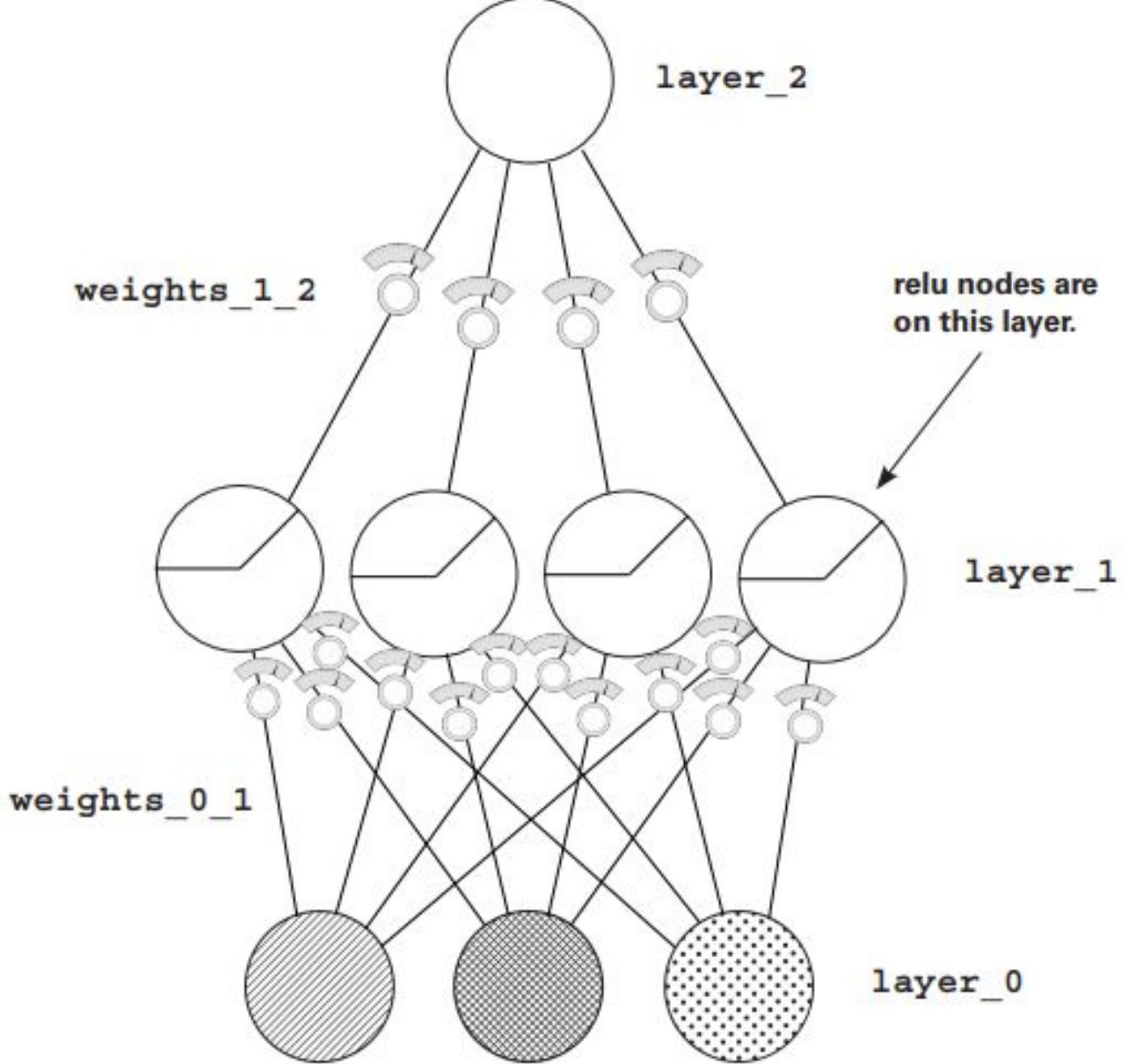
Correlation summarization

- Neural networks seek to find direct and indirect correlation between an input layer and an output layer, which are determined by the input and output datasets, respectively.
- At the 10,000-foot level, this is what all neural networks do.
- Given that a neural network is really just a series of matrices connected by layers, let's zoom in slightly and consider what any particular weight matrix is doing.

The previously overcomplicated visualization

While simplifying the mental picture, let's simplify the visualization as well.

- At this point, I expect the visualization of neural networks in your head is something like the picture shown here (because that's the one we used).
- The input dataset is in layer_0, connected by a weight matrix (a bunch of lines) to layer_1, and so on.
- This was a useful tool to learn the basics of how collections of weights and layers come together to learn a function.





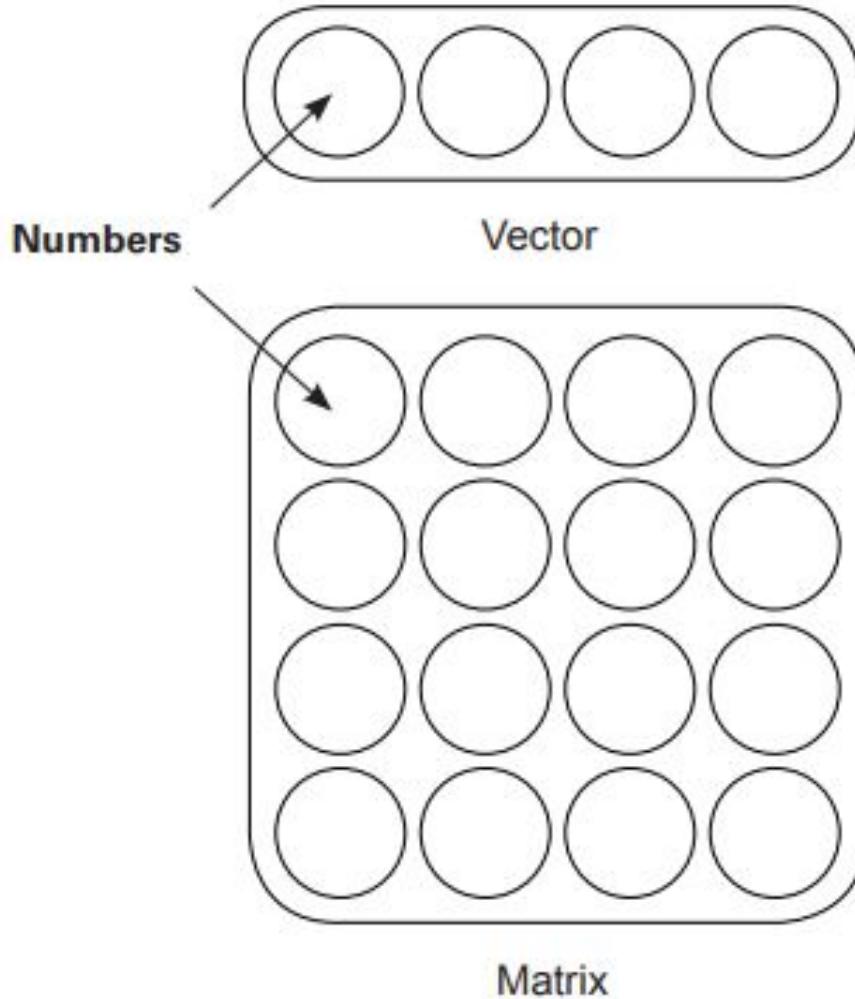
The simplified visualization

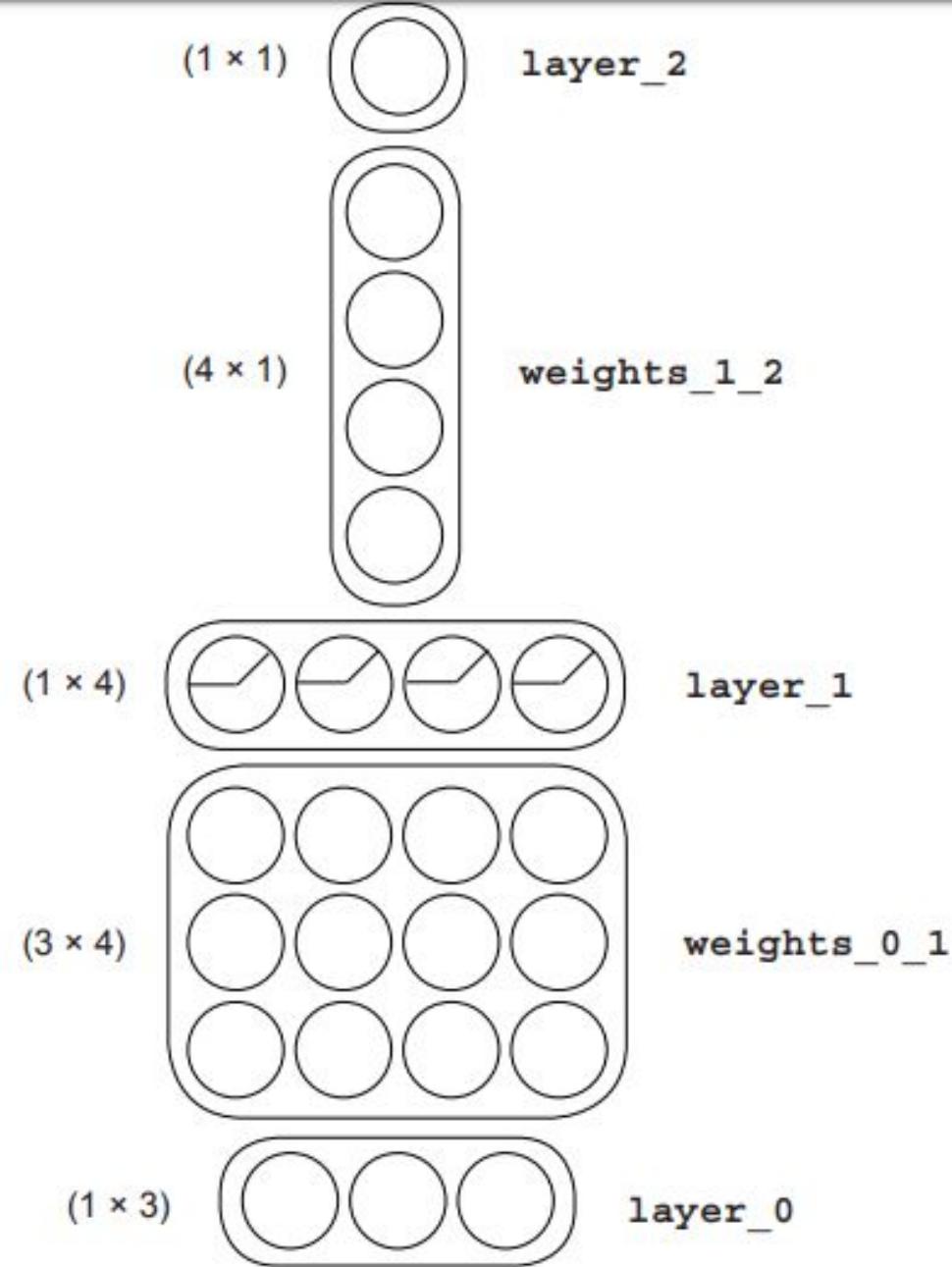


Neural networks are like LEGO bricks, and each brick is a vector or matrix.

- Moving forward, we'll build new neural network architectures in the same way people build new structures with LEGO pieces.
- The great thing about the correlation summarization is that all the bits and pieces that lead to it (backpropagation, gradient descent, alpha, dropout, mini-batching, and so on) don't depend on a particular configuration of the LEGOs.

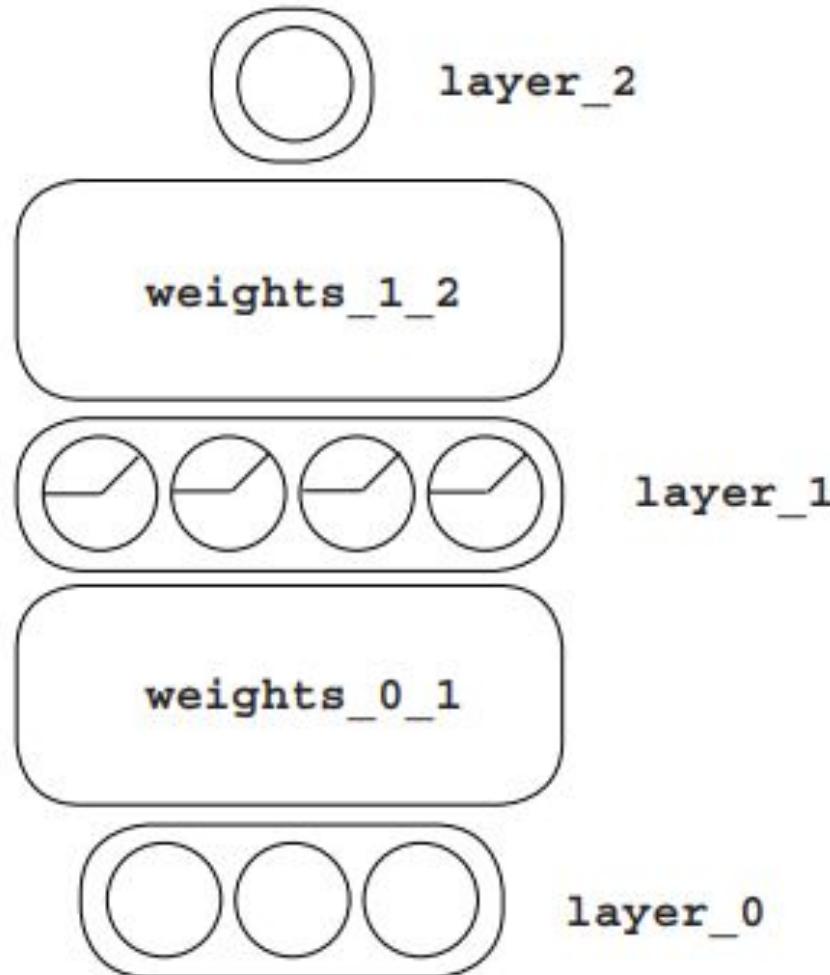
The simplified visualization



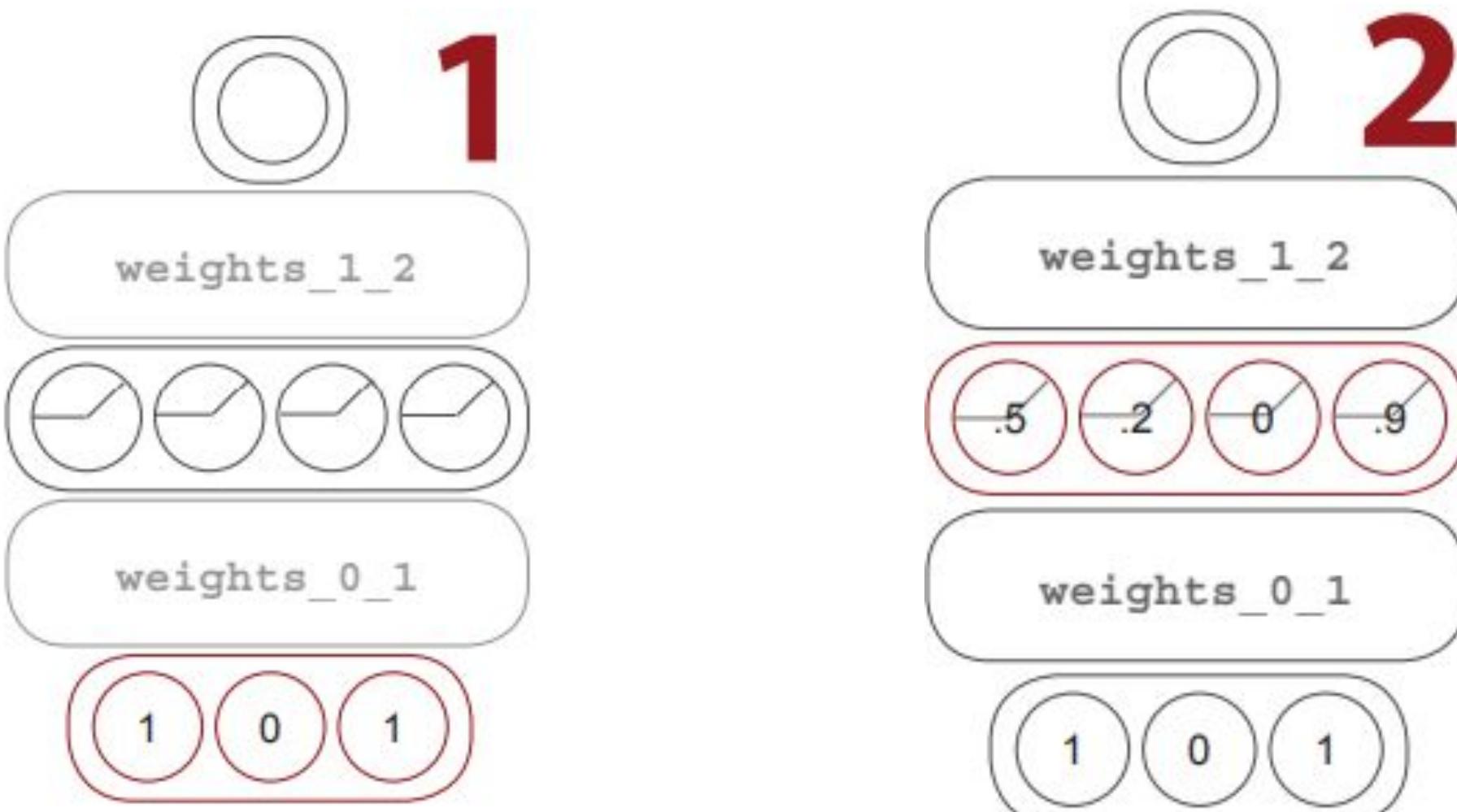


Simplifying even further

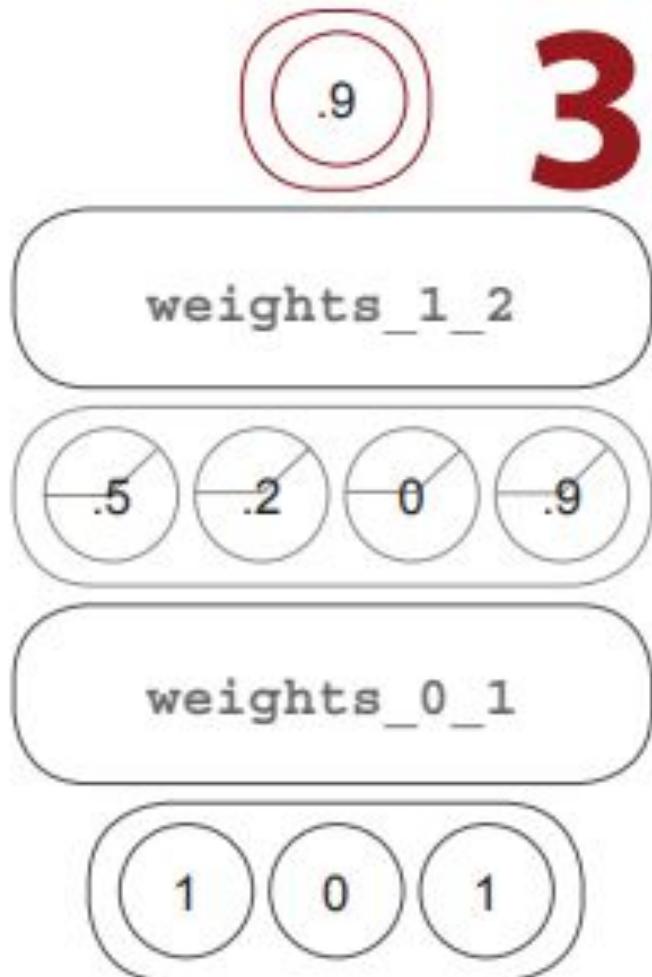
The dimensionality of the matrices is determined by the layers.



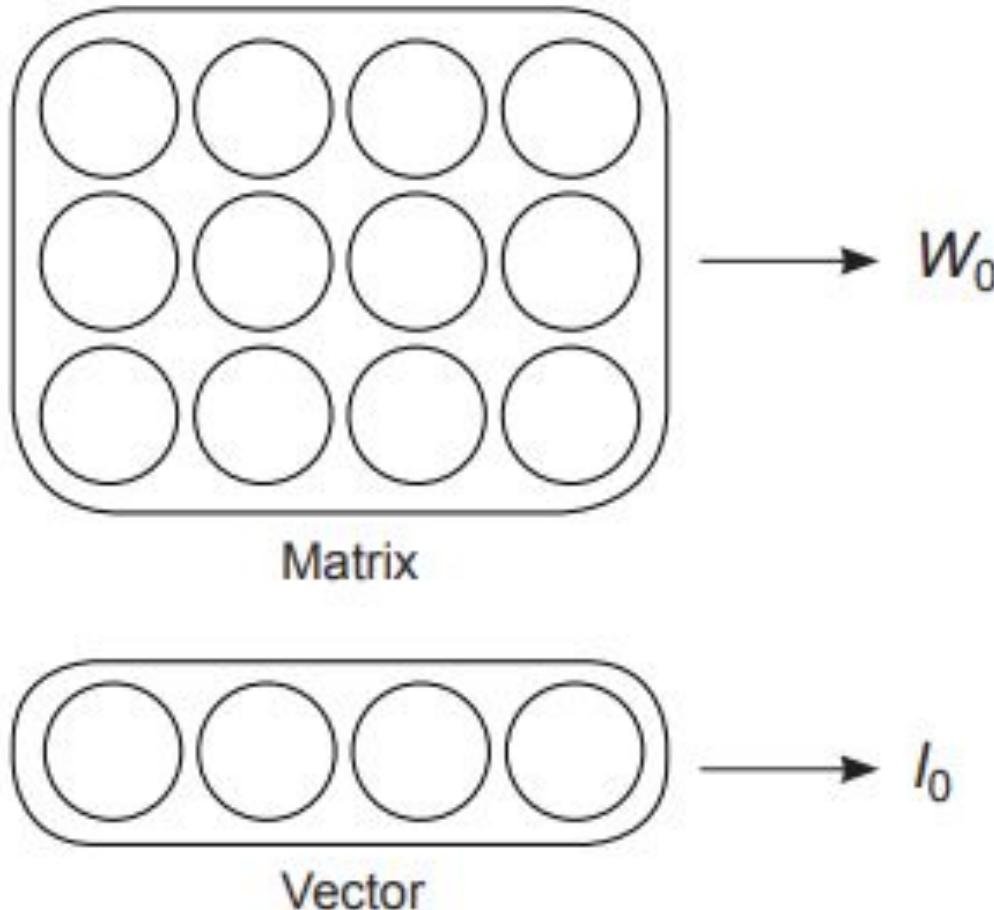
Let's see this network predict



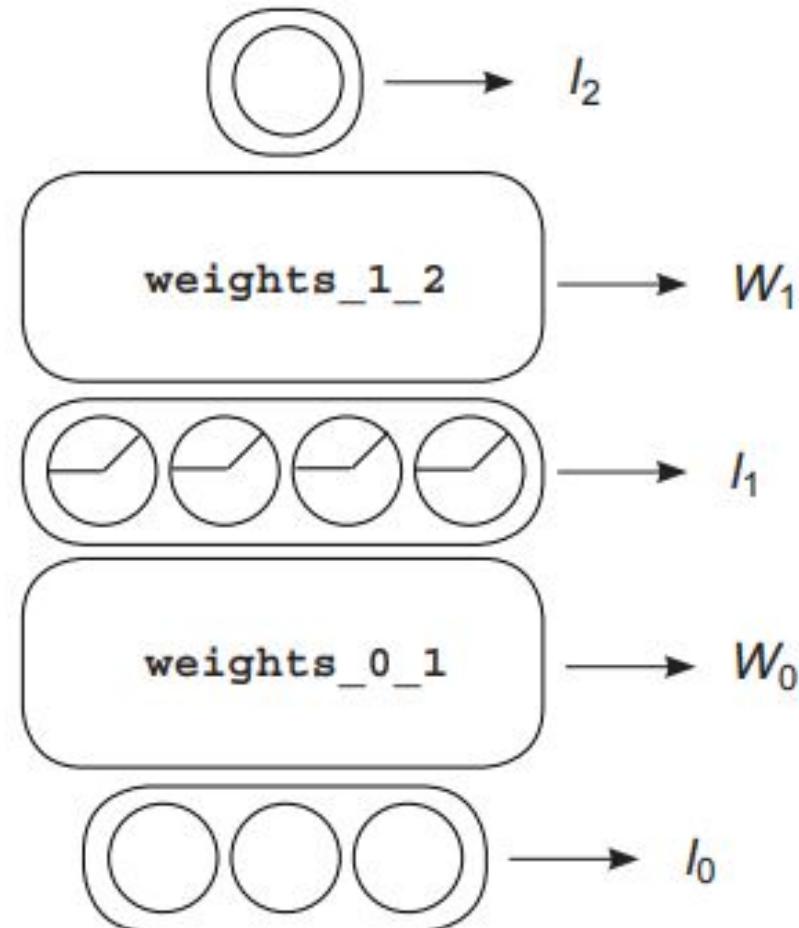
Let's see this network predict



Visualizing using letters instead of pictures



Visualizing using letters instead of pictures





Linking the variables



The letters can be combined to indicate functions and

Vector-matrix multiplication is simple. To visualize that two letters are being multiplied by each other, put them next to each other. For example:

Algebra

$$I_0 W_0$$

$$I_1 W_1$$

Translation

“Take the layer 0 vector and perform vector-matrix multiplication with the weight matrix 0.”

“Take the layer 1 vector and perform vector-matrix multiplication with the weight matrix 1.”

You can even throw in arbitrary functions like `relu` using notation that looks almost exactly like the Python code. This is crazy-intuitive stuff.

$$l_1 = \text{relu}(l_0 W_0)$$

“To create the layer 1 vector, take the layer 0 vector and perform vector-matrix multiplication with the weight matrix 0; then perform the `relu` function on the output (setting all negative numbers to 0).”

$$l_2 = l_1 W_1$$

“To create the layer 2 vector, take the layer 1 vector and perform vector-matrix multiplication with the weight matrix 1.”

If you notice, the layer 2 algebra contains layer 1 as an input variable. This means you can represent the *entire neural network* in one expression by chaining them together.

$$l_2 = \text{relu}(l_0 W_0) W_1$$

Thus, all the logic in the forward propagation step can be contained in this one formula. Note: baked into this formula is the assumption that the vectors and matrices have the right dimensions.



Everything side by side

Let's see the visualization, algebra formula, and Python code in one place

- I don't think much dialogue is necessary on this page.
- Take a minute and look at each piece of forward propagation through these four different ways of seeing it.
- It's my hope that you'll truly grok forward propagation and understand the architecture by seeing it from different perspectives, all in one place

```
layer_2 = relu(layer_0.dot(weights_0_1)).dot(weights_1_2)  

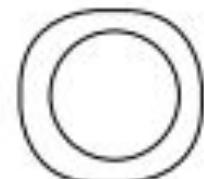
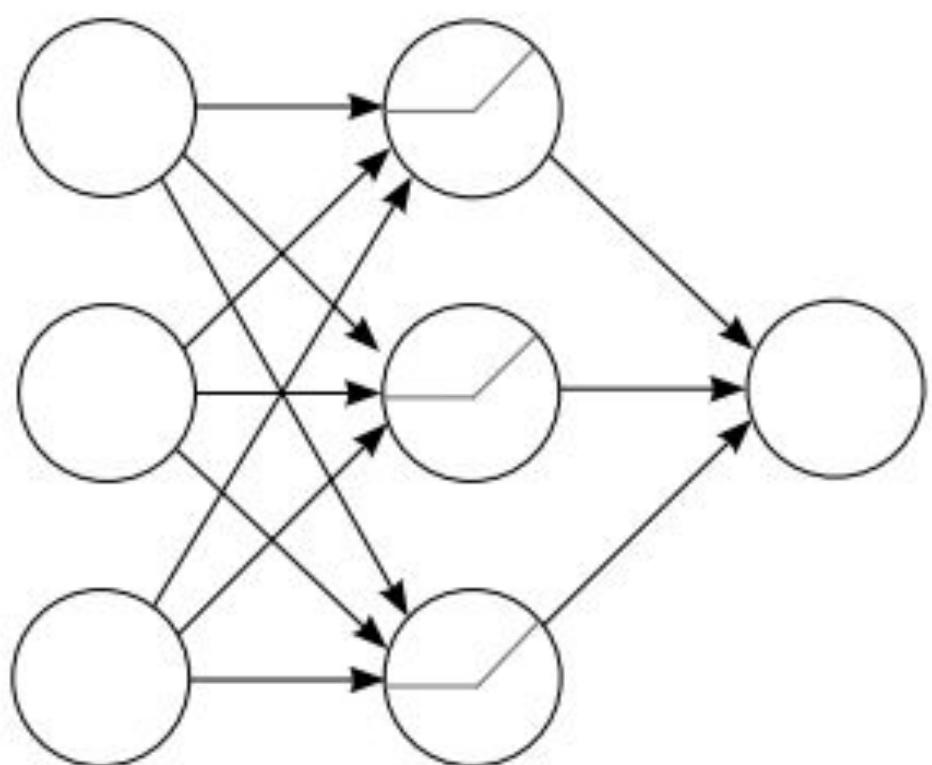
$$l_2 = \text{relu}(l_0 W_0) W_1$$

```

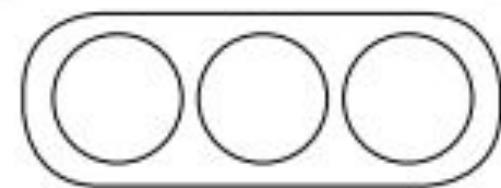
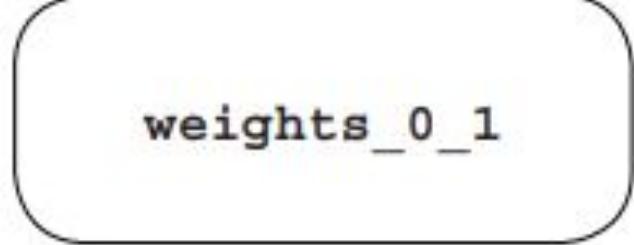
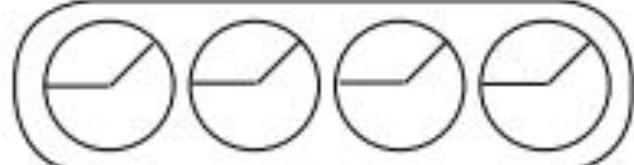
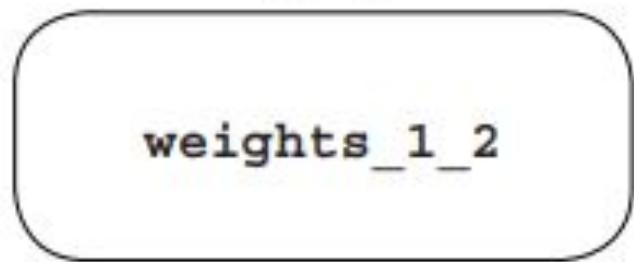
Inputs

Hiddens

Prediction



layer_2



layer_0



The importance of visualization tools

We're going to be studying new architectures.

- Different datasets and domains have different characteristics.
 - For example, image data has different kinds of signal and noise than text data.
 - Even though neural networks can be used in many situations, different architectures will be better suited to different problems because of their ability to locate certain types of correlations.
- 

Lesson 8 learning signal and ignoring noise: introduction to regularization and batching



Introduction to regularization and batching

In this lesson

- Overfitting
- Dropout
- Batch gradient descent



Three-layer network on MNIST

Let's return to the MNIST dataset and attempt to classify it with the new network.

- In this lesson, we're going to study the basics of regularization, which is key to combatting overfitting in neural networks.
- To do this, we'll start with the most powerful neural network (three-layer network with relu hidden layer) on the most challenging task (MNIST digit classification).

```

import sys, numpy as np
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

images, labels = (x_train[0:1000].reshape(1000,28*28) \
                   / 255, y_train[0:1000])
one_hot_labels = np.zeros((len(labels),10))

for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test),28*28) / 255
test_labels = np.zeros((len(y_test),10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1

np.random.seed(1)
relu = lambda x: (x>=0) * x
relu2deriv = lambda x: x>=0
alpha, iterations, hidden_size, pixels_per_image, num_labels = \
    (0.005, 350, 40, 784, 10)
weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

```

Returns **x if $x > 0$**
returns **0 otherwise**

Returns **1 for input > 0**
returns **0 otherwise**

```
for j in range(iterations):
    error, correct_cnt = (0.0, 0)

    for i in range(len(images)):
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)
        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == \
                            np.argmax(labels[i:i+1]))
        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta = layer_2_delta.dot(weights_1_2.T) \
                        * relu2deriv(layer_1)
        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    sys.stdout.write("\r" + \
                    " I:" + str(j) + \
                    " Error:" + str(error/float(len(images)))[0:5] + \
                    " Correct:" + str(correct_cnt/float(len(images))))
```

....
I:349 Error:0.108 Correct:1.0



Well, that was easy

The neural network perfectly learned to predict all 1,000 images.

- In some ways, this is a real victory. The neural network was able to take a dataset of 1,000 images and learn to correlate each input image with the correct label.
 - How did it do this? It iterated through each image, made a prediction, and then updated each weight ever so slightly so the prediction was better next time.
 - Doing this long enough on all the images eventually reached a state where the network could correctly predict on all the images.
- 

Well, that was easy

```
if(j % 10 == 0 or j == iterations-1):
    error, correct_cnt = (0.0, 0)

for i in range(len(test_images)):

    layer_0 = test_images[i:i+1]
    layer_1 = relu(np.dot(layer_0,weights_0_1))
    layer_2 = np.dot(layer_1,weights_1_2)

    error += np.sum((test_labels[i:i+1] - layer_2) ** 2)
    correct_cnt += int(np.argmax(layer_2) == \
                        np.argmax(test_labels[i:i+1]))
sys.stdout.write(" Test-Err:" + str(error/float(len(test_images)))[0:5] +\
                 " Test-Acc:" + str(correct_cnt/float(len(test_images))))
print()
```

Error:0.653 Correct:0.7073



Well, that was easy



- The network did horribly! It predicted with an accuracy of only 70.7%.
- Why does it do so terribly on these new testing images when it learned to predict with 100% accuracy on the training data? How strange. This 70.7% number is called the test accuracy.
- It's the accuracy of the neural network on data the network was not trained on..



Memorization vs. generalization

Memorizing 1,000 images is easier than generalizing to all images.

- Let's consider again how a neural network learns. It adjusts each weight in each matrix so the network is better able to take specific inputs and make a specific prediction.
- Perhaps a better question might be, "If we train it on 1,000 images, which it learns to predict perfectly, why does it work on other images at all?"

I:0 Train-Err:0.722 Train-Acc:0.537 Test-Err:0.601 Test-Acc:0.6488
I:10 Train-Err:0.312 Train-Acc:0.901 Test-Err:0.420 Test-Acc:0.8114
I:20 Train-Err:0.260 Train-Acc:0.93 Test-Err:0.414 Test-Acc:0.8111
I:30 Train-Err:0.232 Train-Acc:0.946 Test-Err:0.417 Test-Acc:0.8066
I:40 Train-Err:0.215 Train-Acc:0.956 Test-Err:0.426 Test-Acc:0.8019
I:50 Train-Err:0.204 Train-Acc:0.966 Test-Err:0.437 Test-Acc:0.7982
I:60 Train-Err:0.194 Train-Acc:0.967 Test-Err:0.448 Test-Acc:0.7921
I:70 Train-Err:0.186 Train-Acc:0.975 Test-Err:0.458 Test-Acc:0.7864
I:80 Train-Err:0.179 Train-Acc:0.979 Test-Err:0.466 Test-Acc:0.7817
I:90 Train-Err:0.172 Train-Acc:0.981 Test-Err:0.474 Test-Acc:0.7758
I:100 Train-Err:0.166 Train-Acc:0.984 Test-Err:0.482 Test-Acc:0.7706
I:110 Train-Err:0.161 Train-Acc:0.984 Test-Err:0.489 Test-Acc:0.7686
I:120 Train-Err:0.157 Train-Acc:0.986 Test-Err:0.496 Test-Acc:0.766
I:130 Train-Err:0.153 Train-Acc:0.99 Test-Err:0.502 Test-Acc:0.7622
I:140 Train-Err:0.149 Train-Acc:0.991 Test-Err:0.508 Test-Acc:0.758

....
I:210 Train-Err:0.127 Train-Acc:0.998 Test-Err:0.544 Test-Acc:0.7446
I:220 Train-Err:0.125 Train-Acc:0.998 Test-Err:0.552 Test-Acc:0.7416
I:230 Train-Err:0.123 Train-Acc:0.998 Test-Err:0.560 Test-Acc:0.7372
I:240 Train-Err:0.121 Train-Acc:0.998 Test-Err:0.569 Test-Acc:0.7344
I:250 Train-Err:0.120 Train-Acc:0.999 Test-Err:0.577 Test-Acc:0.7316
I:260 Train-Err:0.118 Train-Acc:0.999 Test-Err:0.585 Test-Acc:0.729
I:270 Train-Err:0.117 Train-Acc:0.999 Test-Err:0.593 Test-Acc:0.7259
I:280 Train-Err:0.115 Train-Acc:0.999 Test-Err:0.600 Test-Acc:0.723
I:290 Train-Err:0.114 Train-Acc:0.999 Test-Err:0.607 Test-Acc:0.7196
I:300 Train-Err:0.113 Train-Acc:0.999 Test-Err:0.614 Test-Acc:0.7183
I:310 Train-Err:0.112 Train-Acc:0.999 Test-Err:0.622 Test-Acc:0.7165
I:320 Train-Err:0.111 Train-Acc:0.999 Test-Err:0.629 Test-Acc:0.7133
I:330 Train-Err:0.110 Train-Acc:0.999 Test-Err:0.637 Test-Acc:0.7125
I:340 Train-Err:0.109 Train-Acc:1.0 Test-Err:0.645 Test-Acc:0.71
I:349 Train-Err:0.108 Train-Acc:1.0 Test-Err:0.653 Test-Acc:0.7073



Overfitting in neural networks

Neural networks can get worse if you train them too much!

- For some reason, the test accuracy went up for the first 20 iterations and then slowly decreased as the network trained more and more (during which time the training accuracy was still improving).
- This is common in neural networks.
- Let me explain the phenomenon via an analogy



Where overfitting comes from

What causes neural networks to overfit?

- Let's alter this scenario a bit. Picture the fresh clay again (unmolded).
 - What if you pushed only a single fork into it? Assuming the clay was very thick, it wouldn't have as much detail as the previous mold (which was imprinted many times).
 - Thus, it would be only a very general shape of a fork.
 - This shape might be compatible with both the three- and fourpronged varieties of fork, because it's still a fuzzy imprint.
- 

Where overfitting comes from





The simplest regularization: Early stopping

Stop training the network when it starts getting worse.

- In the fork-mold example, it takes many forks imprinted many times to create the perfect outline of a three-pronged fork.
- The first few imprints generally capture only the shallow outline of a fork.
- The same can be said for neural networks. As a result, early stopping is the cheapest form of regularization, and if you're in a pinch, it can be quite effective.



Industry standard regularization: Dropout

The method: Randomly turn off neurons (set them to 0) during training.

- This regularization technique is as simple as it sounds.
- During training, you randomly set neurons in the network to 0 (and usually the deltas on the same nodes during backpropagation, but you technically don't have to).
- This causes the neural network to train exclusively using random subsections of the neural network.



Why dropout works: Ensembling works

Dropout is a form of training a bunch of networks and averaging them.

- Something to keep in mind: neural networks always start out randomly. Why does this matter? Well, because neural networks learn by trial and error, this ultimately means every neural network learns a little differently.
- It may learn equally effectively, but no two neural networks are ever exactly the same (unless they start out exactly the same for some random or intentional reason).

Dropout in code

Here's how to use dropout in the real world.

```
i = 0
layer_0 = images[i:i+1]
dropout_mask = np.random.randint(2, size=layer_1.shape)

layer_1 *= dropout_mask * 2
layer_2 = np.dot(layer_1, weights_1_2)

error += np.sum((labels[i:i+1] - layer_2) ** 2)

correct_cnt += int(np.argmax(layer_2) == \
                    np.argmax(labels[i:i+1]))

layer_2_delta = (labels[i:i+1] - layer_2)
layer_1_delta = layer_2_delta.dot(weights_1_2.T) \
    * relu2deriv(layer_1)

layer_1_delta *= dropout_mask

weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)
```



Dropout in code



- To implement dropout on a layer (in this case, `layer_1`), multiply the `layer_1` values by a random matrix of 1s and 0s.
- This has the effect of randomly turning off nodes in `layer_1` by setting them to equal 0.
- Note that `dropout_mask` uses what's called a 50% Bernoulli distribution such that 50% of the time, each value in `dropout_mask` is 1, and $(1 - 50\% = 50\%)$ of the time, it's 0.

```
import numpy, sys
np.random.seed(1)
def relu(x):
    return (x >= 0) * x
Returns x if x > 0;
returns 0 otherwise

def relu2deriv(output):
    return output >= 0
Returns 1
for input > 0

alpha, iterations, hidden_size = (0.005, 300, 100)
pixels_per_image, num_labels = (784, 10)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0,0)
    for i in range(len(images)):
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2, size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == \
                           np.argmax(labels[i:i+1]))
    layer_2_delta = (labels[i:i+1] - layer_2)
    layer_1_delta = layer_2_delta.dot(weights_1_2.T) * relu2deriv(layer_1)
    layer_1_delta *= dropout_mask
```

Dropout in code

```
weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

if(j%10 == 0):
    test_error = 0.0
    test_correct_cnt = 0

    for i in range(len(test_images)):
        layer_0 = test_images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1, weights_1_2)

        test_error += np.sum((test_labels[i:i+1] - layer_2) ** 2)
        test_correct_cnt += int(np.argmax(layer_2) == \
                               np.argmax(test_labels[i:i+1]))


    sys.stdout.write("\n" + \
                    "I:" + str(j) + \
                    " Test-Err:" + str(test_error/ float(len(test_images)))[0:5] + \
                    " Test-Acc:" + str(test_correct_cnt/ float(len(test_images)))+ \
                    " Train-Err:" + str(error/ float(len(images)))[0:5] + \
                    " Train-Acc:" + str(correct_cnt/ float(len(images))))
```



Dropout evaluated on MNIST

I:0 Test-Err:0.641 Test-Acc:0.6333 Train-Err:0.891 Train-Acc:0.413
I:10 Test-Err:0.458 Test-Acc:0.787 Train-Err:0.472 Train-Acc:0.764
I:20 Test-Err:0.415 Test-Acc:0.8133 Train-Err:0.430 Train-Acc:0.809
I:30 Test-Err:0.421 Test-Acc:0.8114 Train-Err:0.415 Train-Acc:0.811
I:40 Test-Err:0.419 Test-Acc:0.8112 Train-Err:0.413 Train-Acc:0.827
I:50 Test-Err:0.409 Test-Acc:0.8133 Train-Err:0.392 Train-Acc:0.836
I:60 Test-Err:0.412 Test-Acc:0.8236 Train-Err:0.402 Train-Acc:0.836
I:70 Test-Err:0.412 Test-Acc:0.8033 Train-Err:0.383 Train-Acc:0.857
I:80 Test-Err:0.410 Test-Acc:0.8054 Train-Err:0.386 Train-Acc:0.854
I:90 Test-Err:0.411 Test-Acc:0.8144 Train-Err:0.376 Train-Acc:0.868
I:100 Test-Err:0.411 Test-Acc:0.7903 Train-Err:0.369 Train-Acc:0.864
I:110 Test-Err:0.411 Test-Acc:0.8003 Train-Err:0.371 Train-Acc:0.868
I:120 Test-Err:0.402 Test-Acc:0.8046 Train-Err:0.353 Train-Acc:0.857
I:130 Test-Err:0.408 Test-Acc:0.8091 Train-Err:0.352 Train-Acc:0.867
I:140 Test-Err:0.405 Test-Acc:0.8083 Train-Err:0.355 Train-Acc:0.885
I:150 Test-Err:0.404 Test-Acc:0.8107 Train-Err:0.342 Train-Acc:0.883
I:160 Test-Err:0.399 Test-Acc:0.8146 Train-Err:0.361 Train-Acc:0.876
I:170 Test-Err:0.404 Test-Acc:0.8074 Train-Err:0.344 Train-Acc:0.889
I:180 Test-Err:0.399 Test-Acc:0.807 Train-Err:0.333 Train-Acc:0.892
I:190 Test-Err:0.407 Test-Acc:0.8066 Train-Err:0.335 Train-Acc:0.898
I:200 Test-Err:0.405 Test-Acc:0.8036 Train-Err:0.347 Train-Acc:0.893
I:210 Test-Err:0.405 Test-Acc:0.8034 Train-Err:0.336 Train-Acc:0.894
I:220 Test-Err:0.402 Test-Acc:0.8067 Train-Err:0.325 Train-Acc:0.896
I:230 Test-Err:0.404 Test-Acc:0.8091 Train-Err:0.321 Train-Acc:0.894
I:240 Test-Err:0.415 Test-Acc:0.8091 Train-Err:0.332 Train-Acc:0.898
I:250 Test-Err:0.395 Test-Acc:0.8182 Train-Err:0.320 Train-Acc:0.899
I:260 Test-Err:0.390 Test-Acc:0.8204 Train-Err:0.321 Train-Acc:0.899
I:270 Test-Err:0.382 Test-Acc:0.8194 Train-Err:0.312 Train-Acc:0.906
I:280 Test-Err:0.396 Test-Acc:0.8208 Train-Err:0.317 Train-Acc:0.9
I:290 Test-Err:0.399 Test-Acc:0.8181 Train-Err:0.301 Train-Acc:0.908





Dropout evaluated on MNIST

- Not only does the network instead peak at a score of 82.36%, it also doesn't overfit nearly as badly, finishing training with a testing accuracy of 81.81%.
 - Notice that the dropout also slows down Training-Acc, which previously went straight to 100% and stayed there.
 - This should point to what dropout really is: it's noise. It makes it more difficult for the network to train on the training data.
- 



Batch gradient descent



Here's a method for increasing the speed of training and the rate of convergence.

- In the context of this lesson, I'd like to briefly apply a concept introduced several lessons ago: mini-batched stochastic gradient descent.
- I won't go into too much detail, because it's something that's largely taken for granted in neural network training.

Batch gradient descent

```
I:0 Test-Err:0.815 Test-Acc:0.3832 Train-Err:1.284 Train-Acc:0.165
I:10 Test-Err:0.568 Test-Acc:0.7173 Train-Err:0.591 Train-Acc:0.672
I:20 Test-Err:0.510 Test-Acc:0.7571 Train-Err:0.532 Train-Acc:0.729
I:30 Test-Err:0.485 Test-Acc:0.7793 Train-Err:0.498 Train-Acc:0.754
I:40 Test-Err:0.468 Test-Acc:0.7877 Train-Err:0.489 Train-Acc:0.749
I:50 Test-Err:0.458 Test-Acc:0.793 Train-Err:0.468 Train-Acc:0.775
I:60 Test-Err:0.452 Test-Acc:0.7995 Train-Err:0.452 Train-Acc:0.799
I:70 Test-Err:0.446 Test-Acc:0.803 Train-Err:0.453 Train-Acc:0.792
I:80 Test-Err:0.451 Test-Acc:0.7968 Train-Err:0.457 Train-Acc:0.786
I:90 Test-Err:0.447 Test-Acc:0.795 Train-Err:0.454 Train-Acc:0.799
I:100 Test-Err:0.448 Test-Acc:0.793 Train-Err:0.447 Train-Acc:0.796
I:110 Test-Err:0.441 Test-Acc:0.7943 Train-Err:0.426 Train-Acc:0.816
I:120 Test-Err:0.442 Test-Acc:0.7966 Train-Err:0.431 Train-Acc:0.813
I:130 Test-Err:0.441 Test-Acc:0.7906 Train-Err:0.434 Train-Acc:0.816
I:140 Test-Err:0.447 Test-Acc:0.7874 Train-Err:0.437 Train-Acc:0.822
I:150 Test-Err:0.443 Test-Acc:0.7899 Train-Err:0.414 Train-Acc:0.823
I:160 Test-Err:0.438 Test-Acc:0.797 Train-Err:0.427 Train-Acc:0.811
I:170 Test-Err:0.440 Test-Acc:0.7884 Train-Err:0.418 Train-Acc:0.828
I:180 Test-Err:0.436 Test-Acc:0.7935 Train-Err:0.407 Train-Acc:0.834
I:190 Test-Err:0.434 Test-Acc:0.7935 Train-Err:0.410 Train-Acc:0.831
I:200 Test-Err:0.435 Test-Acc:0.7972 Train-Err:0.416 Train-Acc:0.829
I:210 Test-Err:0.434 Test-Acc:0.7923 Train-Err:0.409 Train-Acc:0.83
I:220 Test-Err:0.433 Test-Acc:0.8032 Train-Err:0.396 Train-Acc:0.832
I:230 Test-Err:0.431 Test-Acc:0.8036 Train-Err:0.393 Train-Acc:0.853
I:240 Test-Err:0.430 Test-Acc:0.8047 Train-Err:0.397 Train-Acc:0.844
I:250 Test-Err:0.429 Test-Acc:0.8028 Train-Err:0.386 Train-Acc:0.843
I:260 Test-Err:0.431 Test-Acc:0.8038 Train-Err:0.394 Train-Acc:0.843
I:270 Test-Err:0.428 Test-Acc:0.8014 Train-Err:0.384 Train-Acc:0.845
I:280 Test-Err:0.430 Test-Acc:0.8067 Train-Err:0.401 Train-Acc:0.846
I:290 Test-Err:0.428 Test-Acc:0.7975 Train-Err:0.383 Train-Acc:0.851
```

Batch gradient descent

```
import numpy as np
np.random.seed(1)

def relu(x):
    return (x >= 0) * x ← Returns x  
if x > 0

def relu2deriv(output):
    return output >= 0 ← Returns 1  
for input > 0

batch_size = 100
alpha, iterations = (0.001, 300)
pixels_per_image, num_labels, hidden_size = (784, 10, 100)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0, 0)
    for i in range(int(len(images) / batch_size)):
        batch_start, batch_end = ((i * batch_size), ((i+1)*batch_size))

        layer_0 = images[batch_start:batch_end]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2,size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)
```

```
error += np.sum((labels[batch_start:batch_end] - layer_2) ** 2)
for k in range(batch_size):
    correct_cnt += int(np.argmax(layer_2[k:k+1]) == \
        np.argmax(labels[batch_start+k:batch_start+k+1]))

    layer_2_delta = (labels[batch_start:batch_end]-layer_2) \
                    /batch_size
    layer_1_delta = layer_2_delta.dot(weights_1_2.T)* \
                    relu2deriv(layer_1)
    layer_1_delta *= dropout_mask

    weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
    weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

if(j%10 == 0):
    test_error = 0.0
    test_correct_cnt = 0

    for i in range(len(test_images)):
        layer_0 = test_images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1, weights_1_2)
```



Summary



- This lesson addressed two of the most widely used methods for increasing the accuracy and training speed of almost any neural architecture.
- In the following lessons, we'll pivot from sets of tools that are universally applicable to nearly all neural networks, to special purpose architectures that are advantageous for modeling specific types of phenomena in data.

Lesson 9 Modeling probabilities and nonlinearities: activation functions





Activation functions



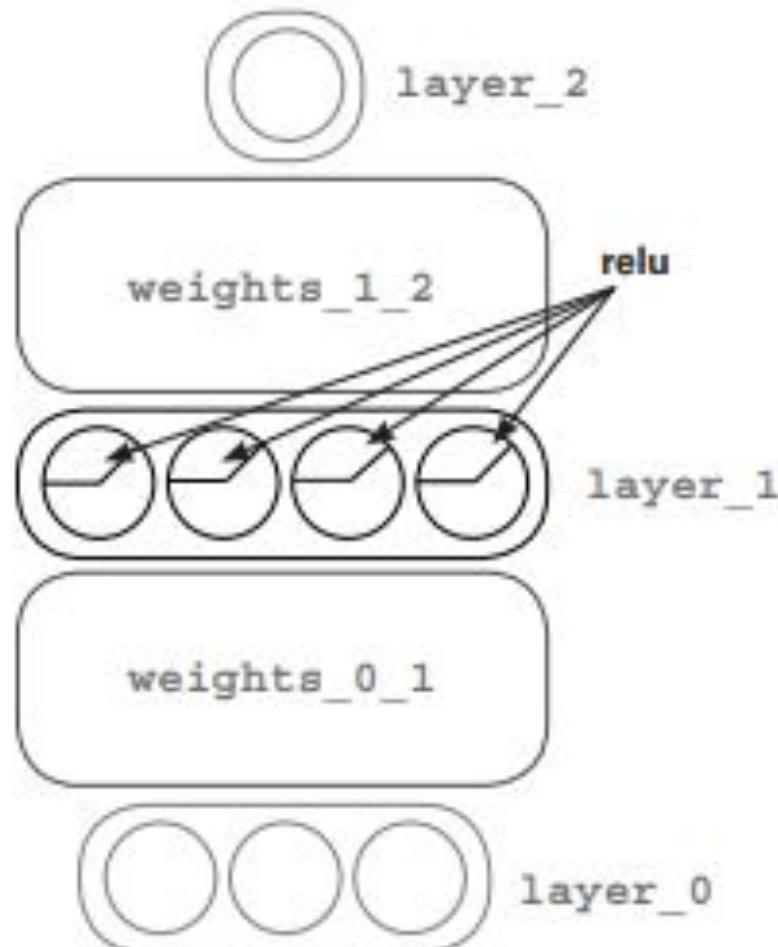
In this lesson

What is an activation function?

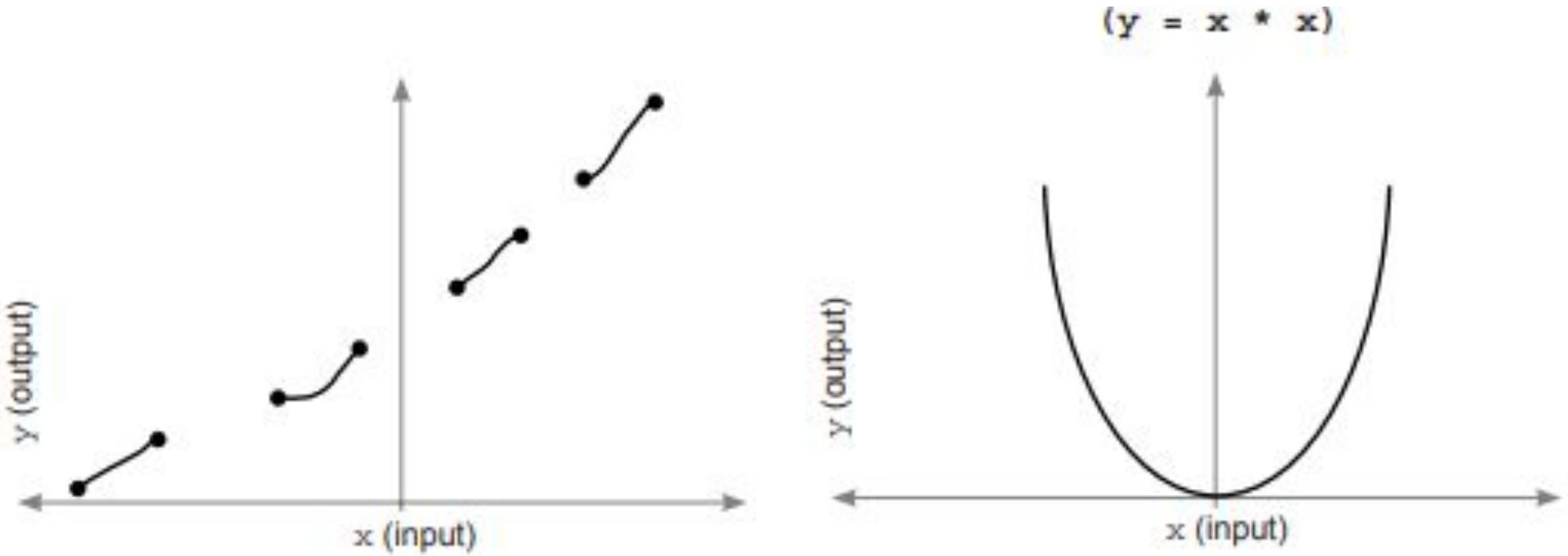
- Standard hidden activation functions
 - Sigmoid
 - Tanh
- Standard output activation functions
 - Softmax
- Activation function installation instructions

What is an activation function?

It's a function applied to the neurons in a layer during prediction.



Constraint 1: The function must be continuous and infinite in domain.

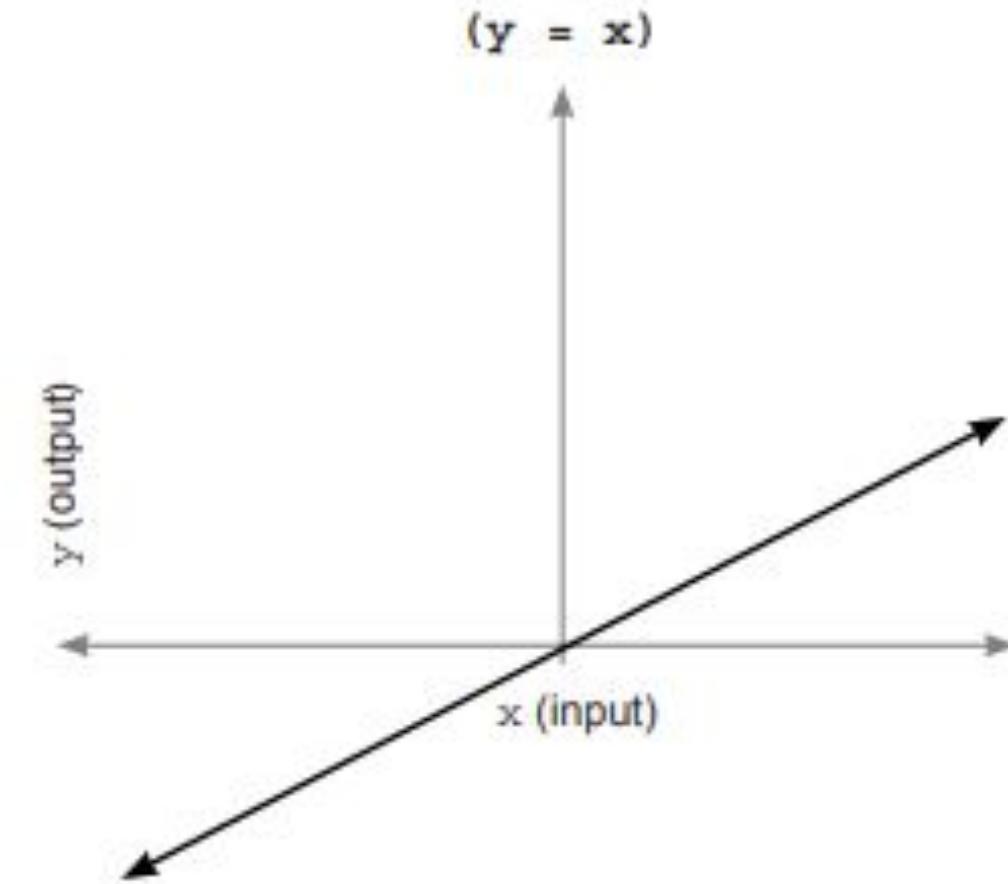
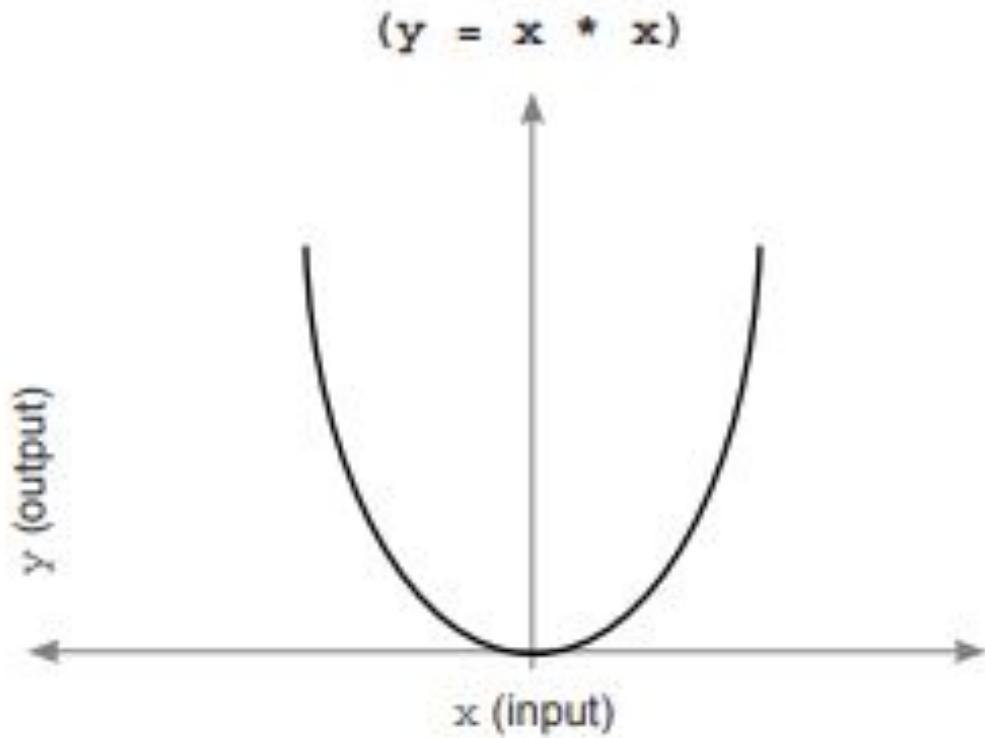




Constraint 2: Good activation functions are monotonic, never changing direction.

- The second constraint is that the function is 1:1. It must never change direction.
 - In other words, it must either be always increasing or always decreasing.
 - As an example, look at the following two functions.
 - These shapes answer the question, “Given x as input, what value of y does the function describe?” The function on the left ($y = x * x$) isn’t an ideal activation function because it isn’t either always increasing or always decreasing.
- 

Constraint 2: Good activation functions are monotonic, never changing direction.

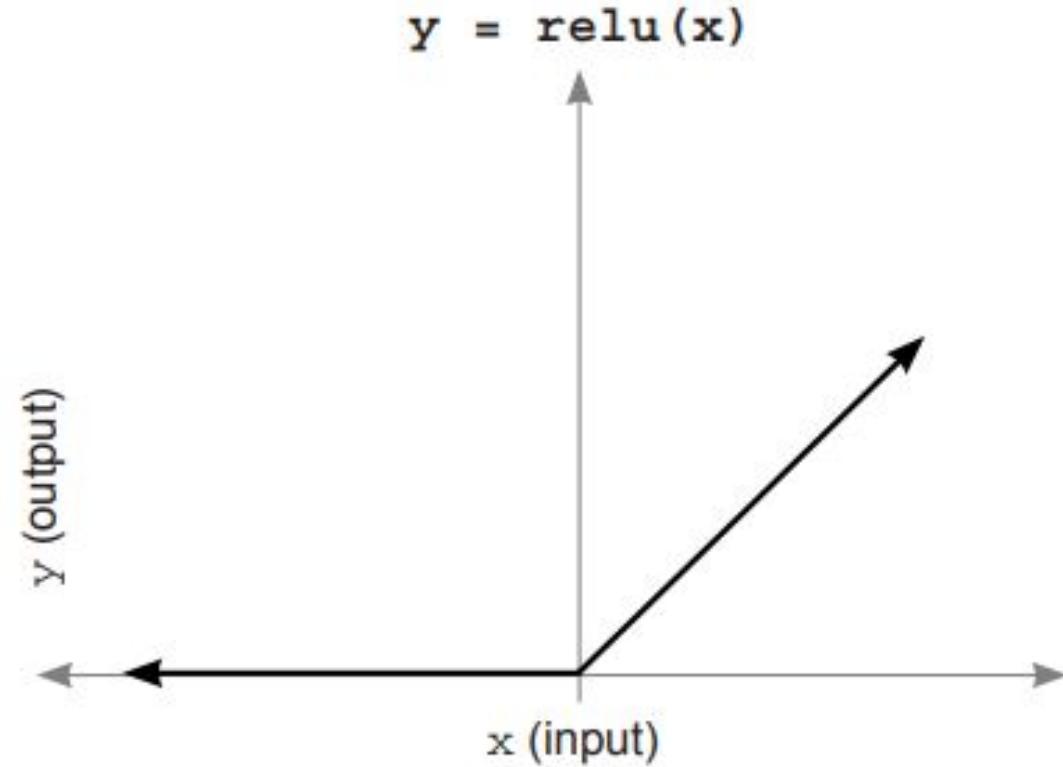
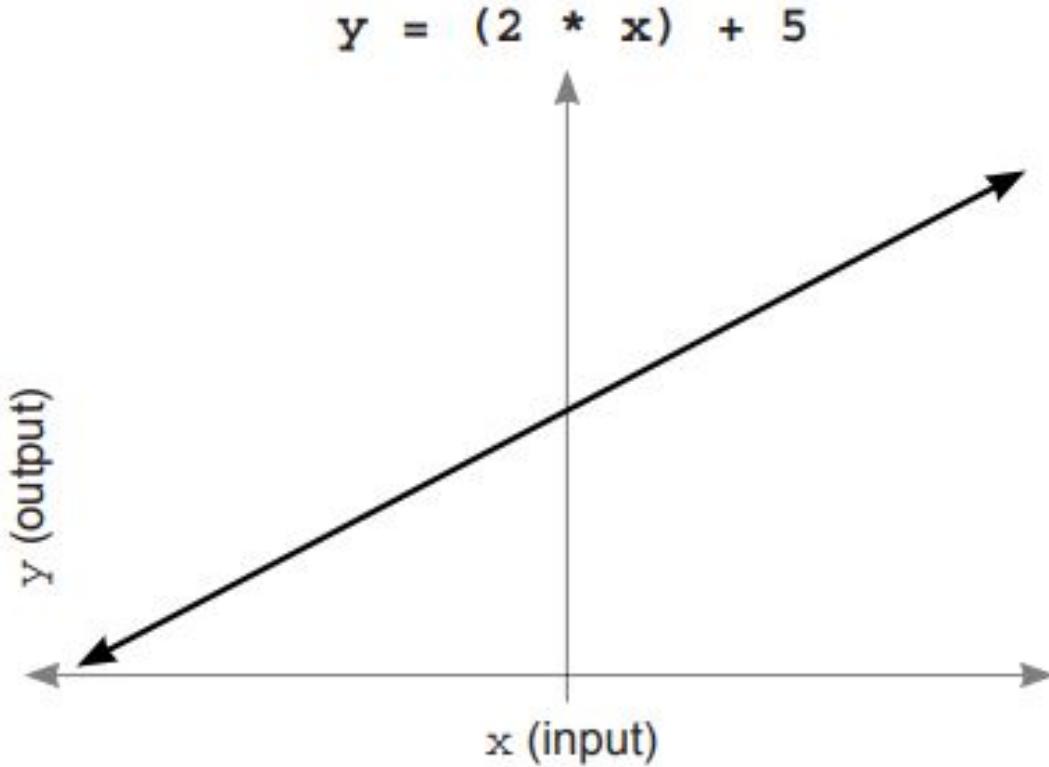




Constraint 3: Good activation functions are nonlinear (they squiggle or turn).

- The third constraint requires a bit of recollection back to lesson 6.
- Remember sometimes correlation? In order to create it, you had to allow the neurons to selectively correlate to input neurons such that a very negative signal from one input into a neuron could reduce how much it correlated to any input (by forcing the neuron to drop to 0, in the case of relu).

Constraint 3: Good activation functions are nonlinear (they squiggle or turn).





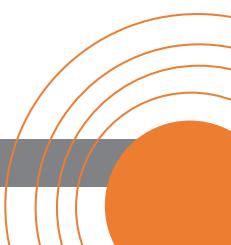
Constraint 4: Good activation functions (and their derivatives) should be efficiently computable.

- This one is pretty simple.
- You'll be calling this function a lot (sometimes billions of times), so you don't want it to be too slow to compute.
- Many recent activation functions have become popular because they're so easy to compute at the expense of their expressiveness (relu is a great example of this)



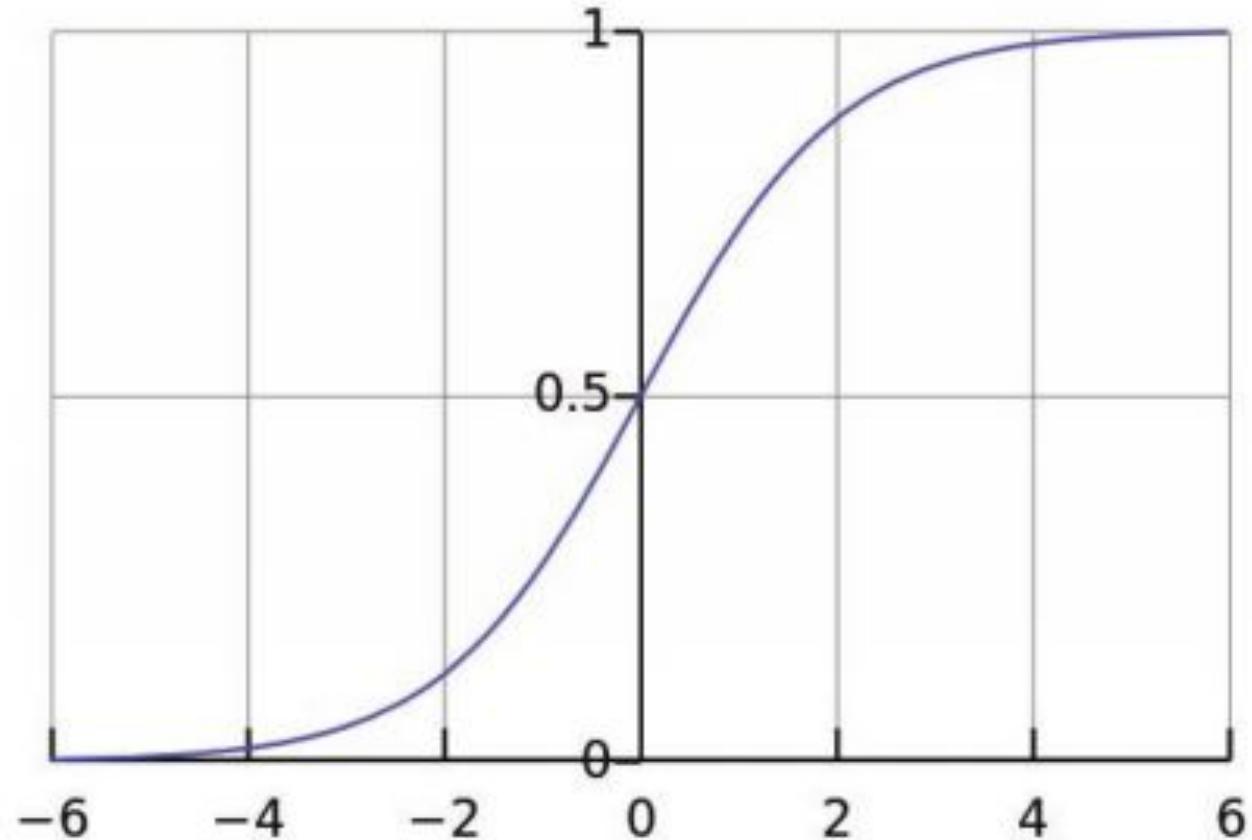
Standard hidden-layer activation functions

Of the infinite possible functions, which ones are most commonly used?

- Even with these constraints, it should be clear that an infinite (possibly transfinite?) number of functions could be used as activation functions.
 - The last few years have seen a lot of progress in state-of-the-art activations.
 - But there's still a relatively small list of activations that account for the vast majority of activation needs, and improvements on them have been minute in most cases.
- 

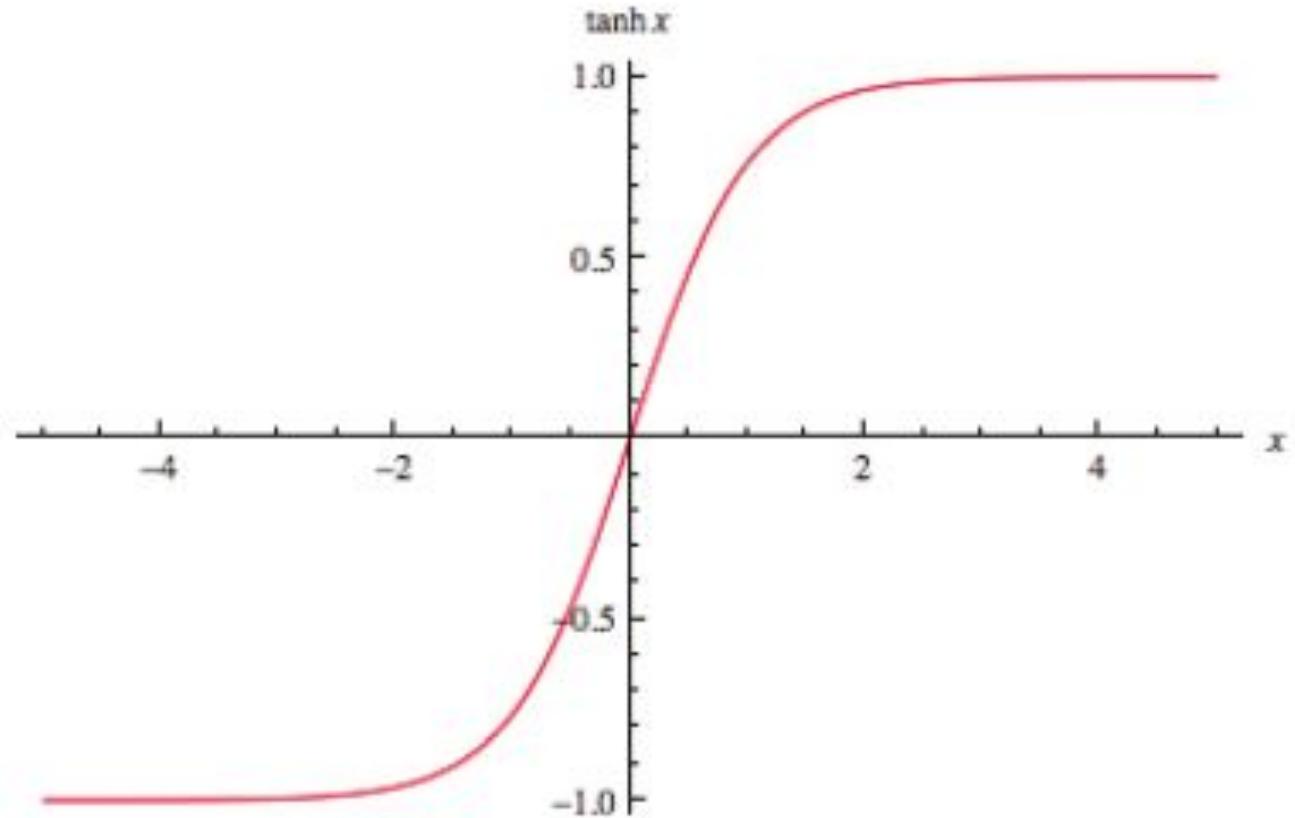
Standard hidden-layer activation functions

sigmoid is the
bread-and-butter
activation.



Standard hidden-layer activation functions

tanh is better than
sigmoid for hidden
layers.





Standard output layer activation functions



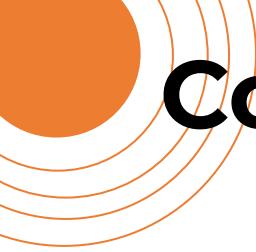
Choosing the best one depends on what you're trying to predict.

- It turns out that what's best for hidden-layer activation functions can be quite different from what's best for output-layer activation functions, especially when it comes to classification.
- Broadly speaking, there are three major types of output layer



Configuration 1: Predicting raw data values (no activation function)

- This is perhaps the most straightforward but least common type of output layer.
- In some cases, people want to train a neural network to transform one matrix of numbers into another matrix of numbers, where the range of the output (difference between lowest and highest values) is something other than a probability.



Configuration 2: Predicting unrelated yes/no probabilities (sigmoid)

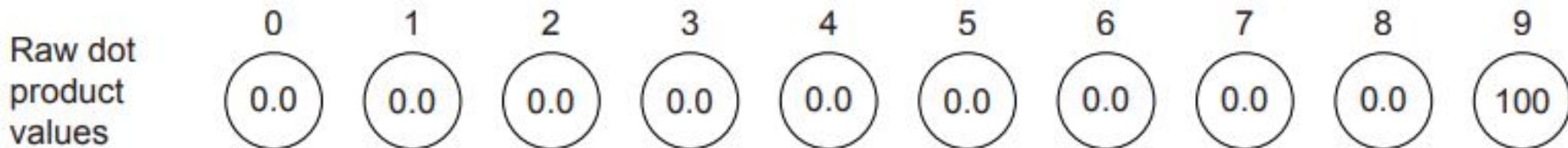
- You'll often want to make multiple binary probabilities in one neural network.
 - We did this in the “Gradient descent with multiple inputs and outputs” section of lesson 5, predicting whether the team would win, whether there would be injuries, and the morale of the team (happy or sad) based on the input data.
 - As an aside, when a neural network has hidden layers, predicting multiple things at once can be beneficial.
- 



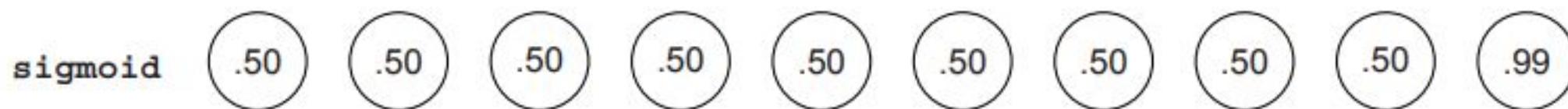
Configuration 3: Predicting which-one probabilities (softmax)

- By far the most common use case in neural networks is predicting a single label out of many.
 - For example, in the MNIST digit classifier, you want to predict which number is in the image.
 - You know ahead of time that the image can't be more than one number.
 - You can train this network with a sigmoid activation function and declare that the highest output probability is the most likely.
- 

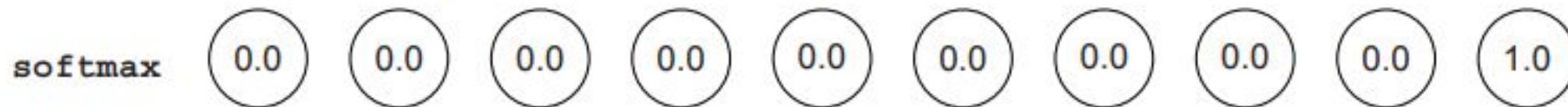
Why do we like this phenomenon? Consider how weight updates are performed. Let's say the MNIST digit classifier should predict that the image is a 9. Also say that the raw weighted sums going into the final layer (before applying an activation function) are the following values:



The network's raw input to the last layer predicts a 0 for every node but 9, where it predicts 100. You might call this perfect. Let's see what happens when these numbers are run through a sigmoid activation function:

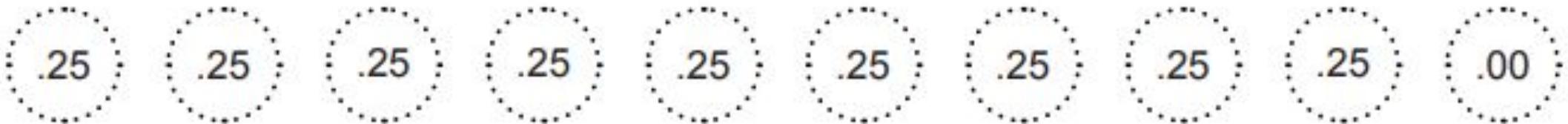


Strangely, the network seems less sure now: 9 is still the highest, but the network seems to think there's a 50% chance that it could be any of the other numbers. Weird! softmax, on the other hand, interprets the input very differently:



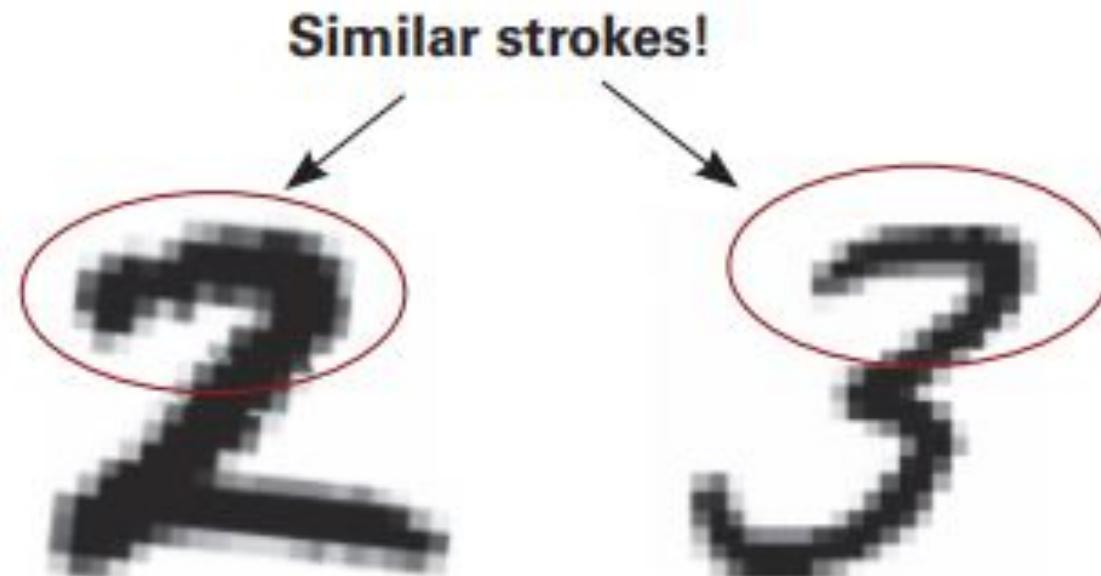
- This looks great. Not only is 9 the highest, but the network doesn't even suspect it's any of the other possible MNIST digits.
- This might seem like a theoretical flaw of sigmoid, but it can have serious consequences when you backpropagate.
- Consider how the mean squared error is calculated on the sigmoid output. In theory, the network is predicting nearly perfectly, right? Surely it won't backprop much error. Not so for sigmoid:

sigmoid
MSE

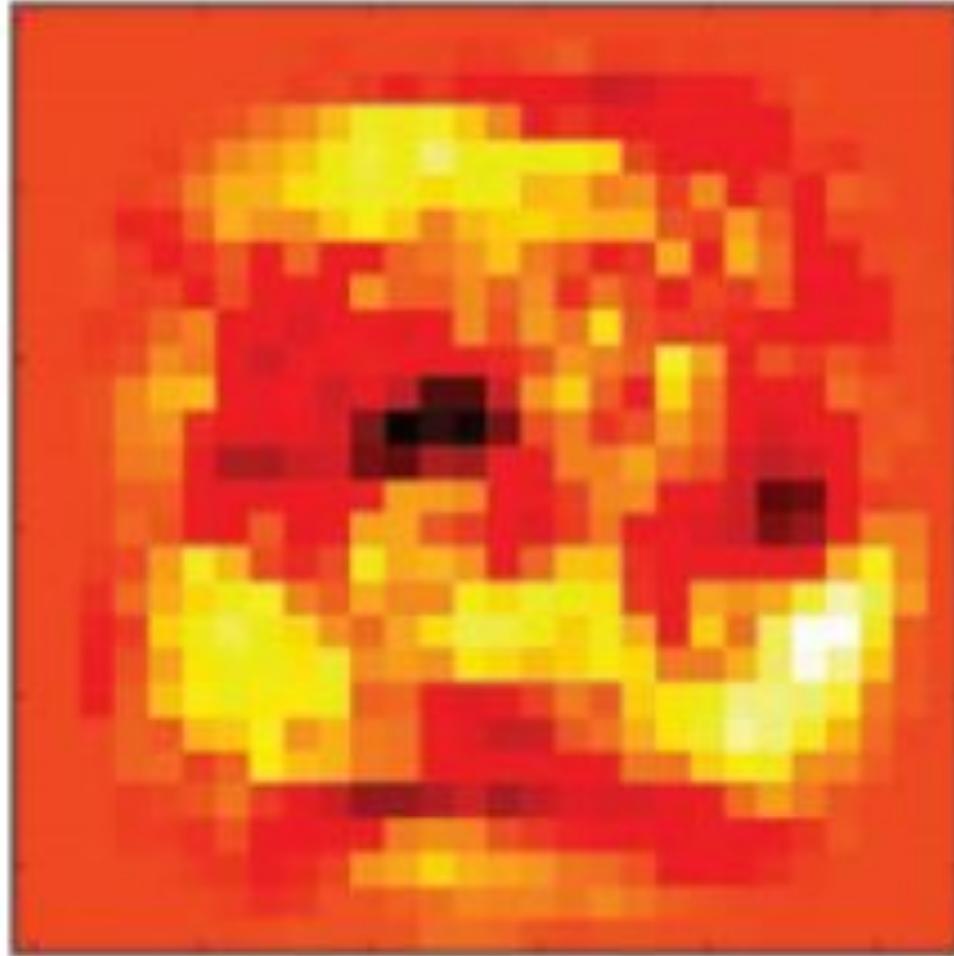


The core issue: Inputs have similarity

Different numbers share characteristics. It's good to let the network believe that.



The core issue: Inputs have similarity

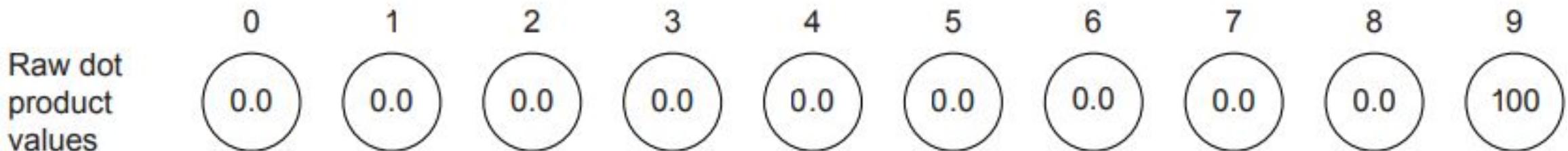




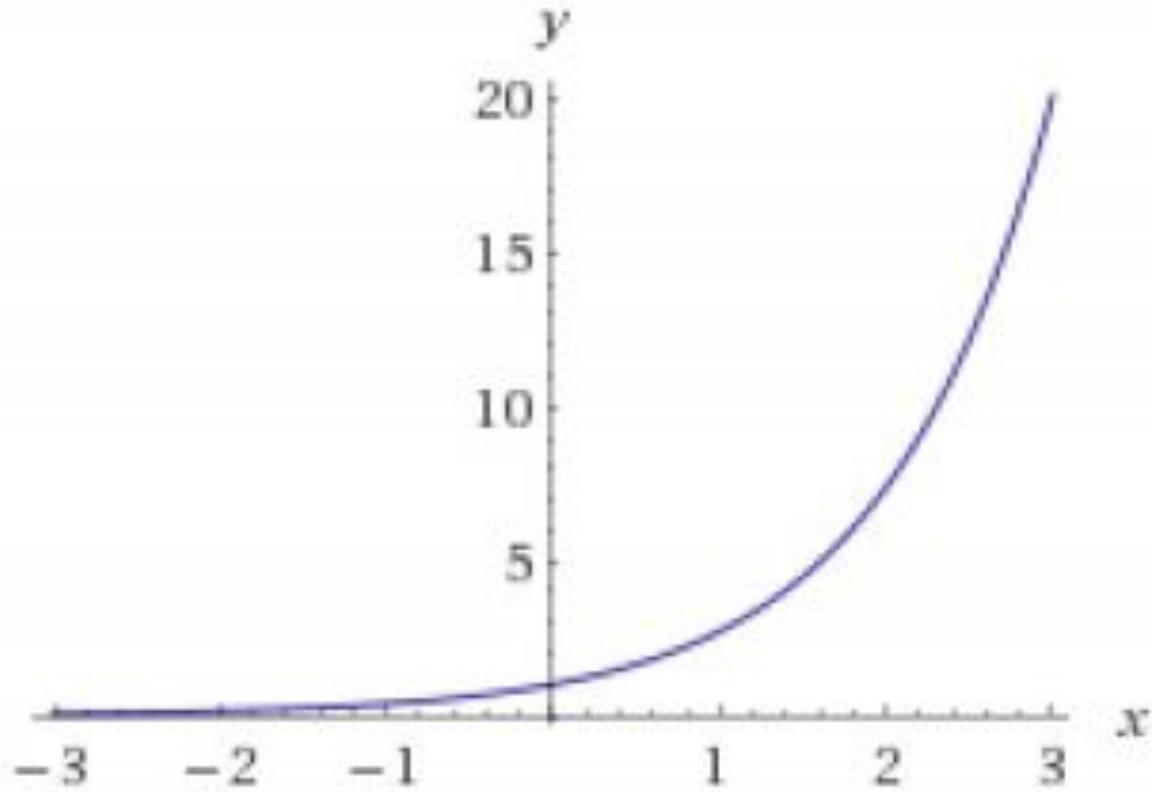
Softmax computation

SoftMax raises each input value exponentially and then divides by the layer's sum.

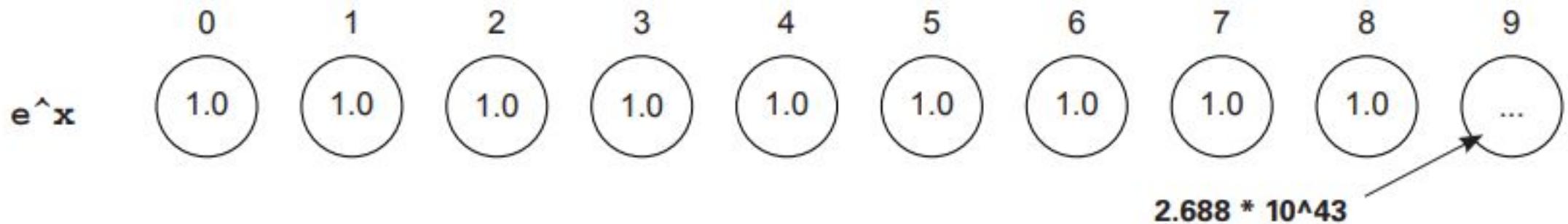
- Let's see a softmax computation on the neural network's hypothetical output values from earlier.
- I'll show them here again so you can see the input to softmax:



Softmax computation



Softmax computation



In short, all the 0s turn to 1s (because 1 is the y intercept of e^x), and the 100 turns into a massive number (2 followed by 43 zeros). If there were any negative numbers, they turned into something between 0 and 1. The next step is to sum all the nodes in the layer and divide each value in the layer by that sum. This effectively makes every number 0 except the value for label 9.





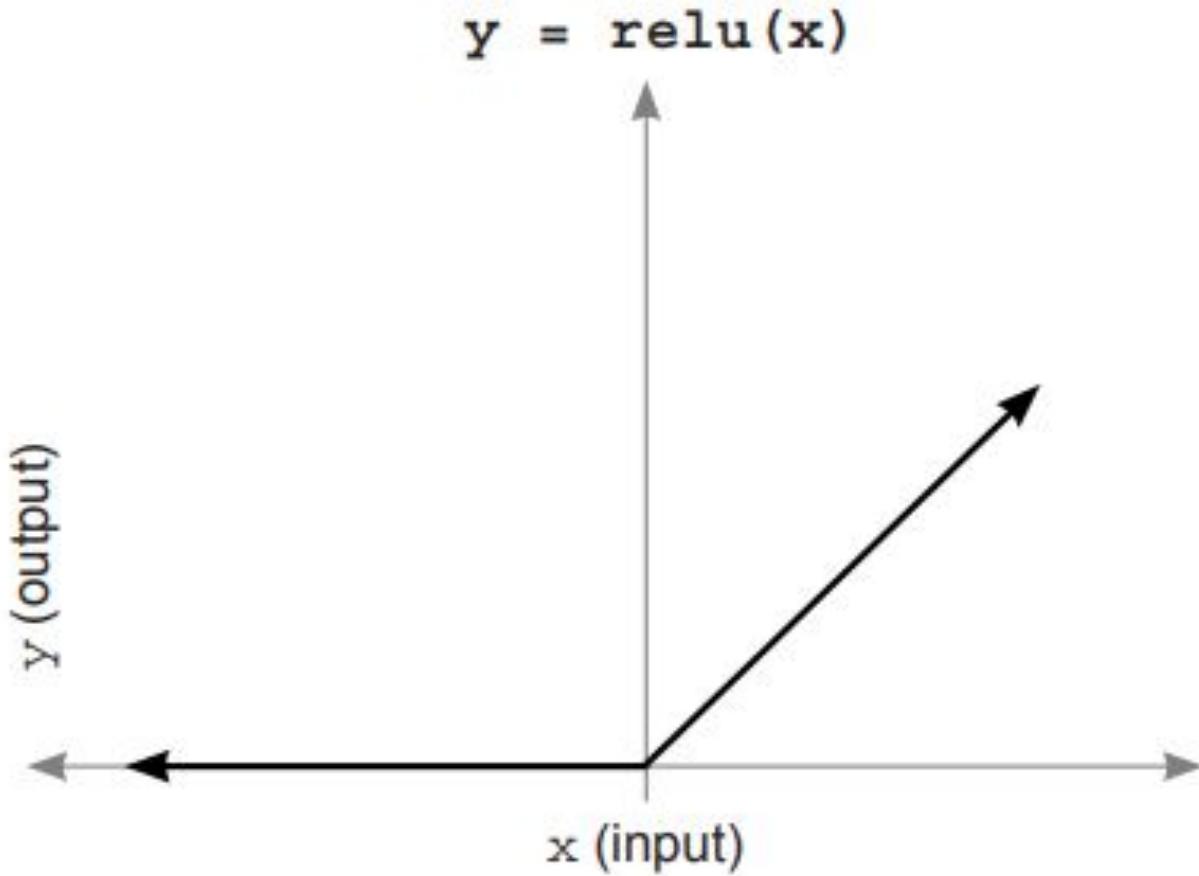
Activation installation instructions

How do you add your favorite activation function to any layer?

- you've already seen an example of how to use a nonlinearity in your first deep neural network: you added a relu activation function to the hidden layer.
- Adding this to forward propagation was relatively straightforward. You took what layer_1 would have been (without an activation) and applied the relu function to each

```
\ layer_0 = images[i:i+1]
layer_1 = relu(np.dot(layer_0,weights_0_1))
layer_2 = np.dot(layer_1,weights_1_2)
```

Activation installation instructions



Activation installation instructions

```
error += np.sum((labels[i:i+1] - layer_2) ** 2)

correct_cnt += int(np.argmax(layer_2) == \
                    np.argmax(labels[i:i+1]))

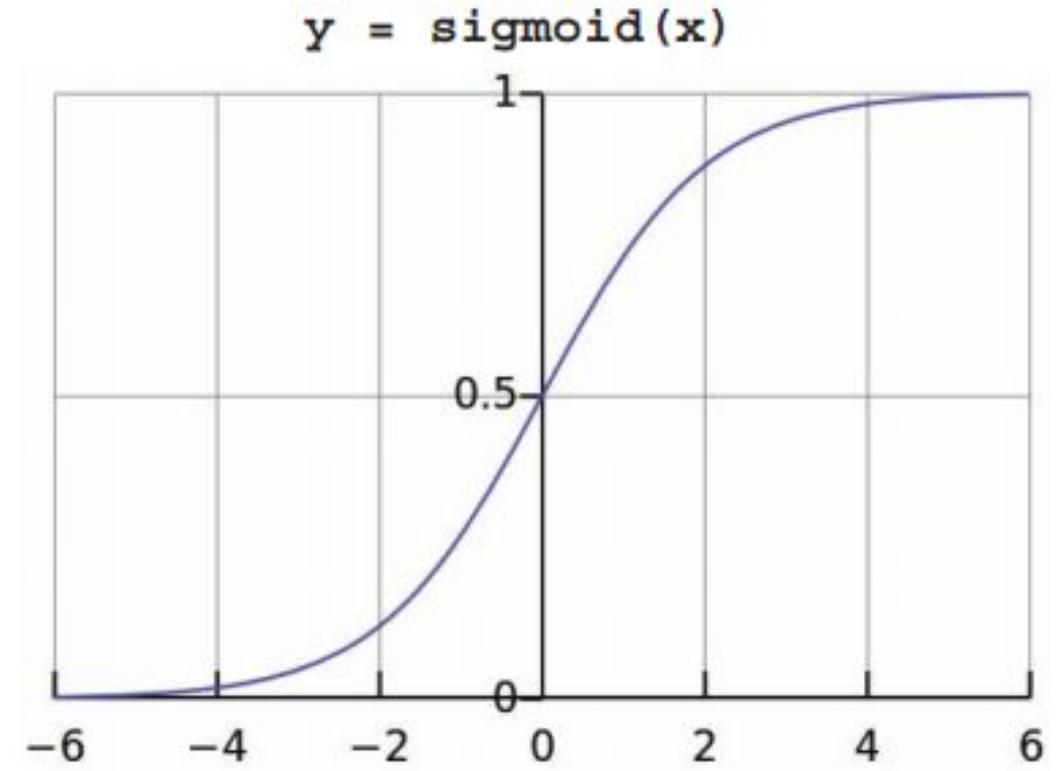
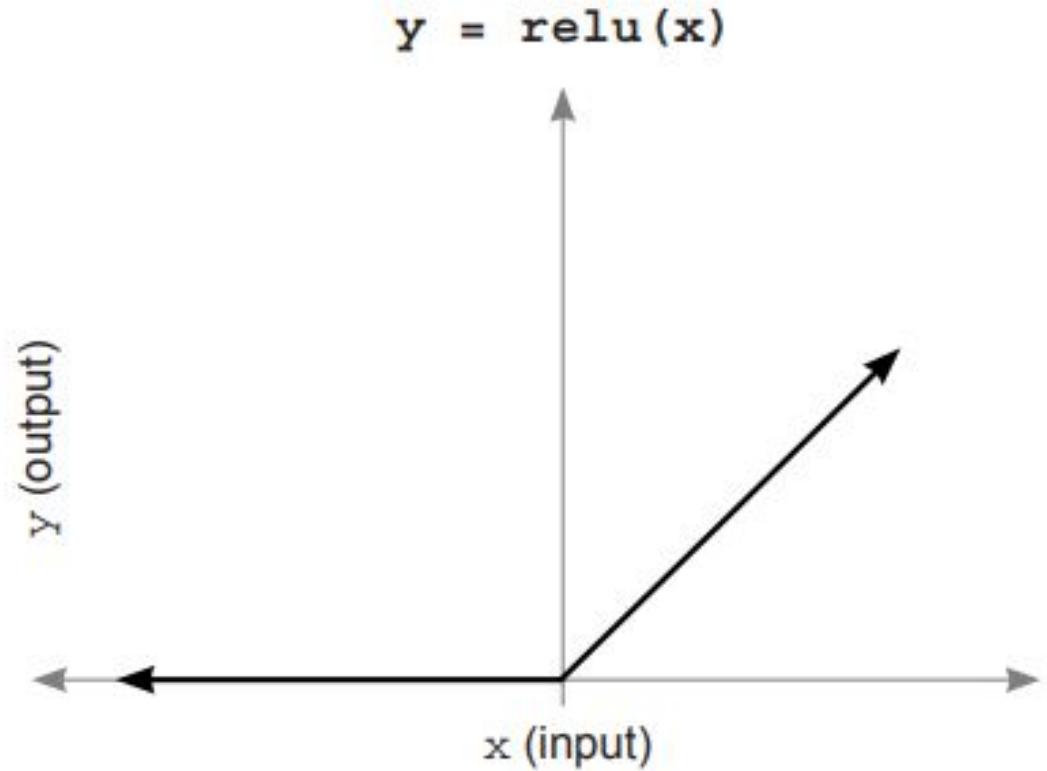
layer_2_delta = (labels[i:i+1] - layer_2)
layer_1_delta = layer_2_delta.dot(weights_1_2.T) \
                    * relu2deriv(layer_1)

weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

def relu(x):
    return (x >= 0) * x                                Returns x if x > 0;
                                                       returns 0 otherwise

def relu2deriv(output):
    return output >= 0                                  Returns 1 for input > 0;
                                                       returns 0 otherwise
```

Activation installation instructions



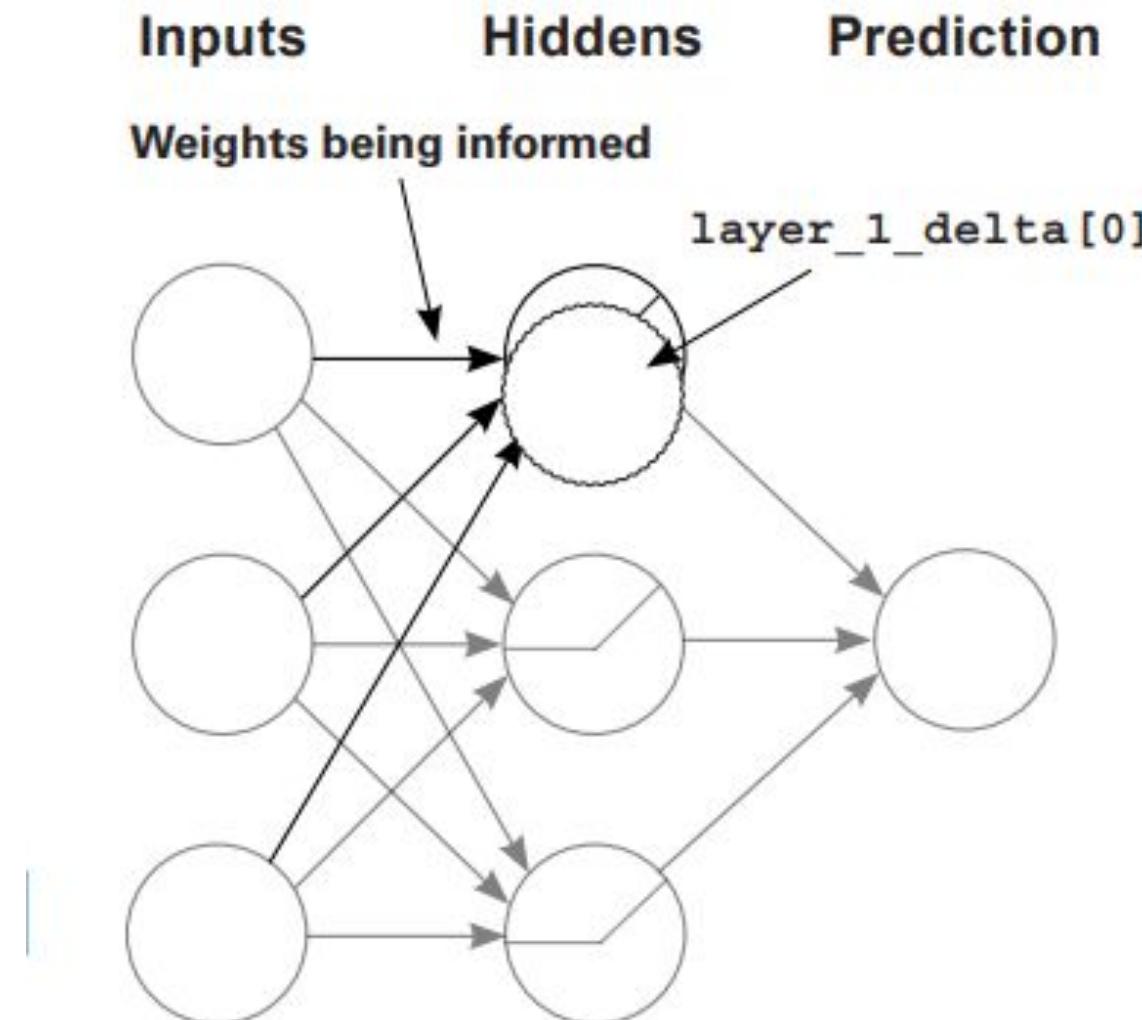


Activation installation instructions

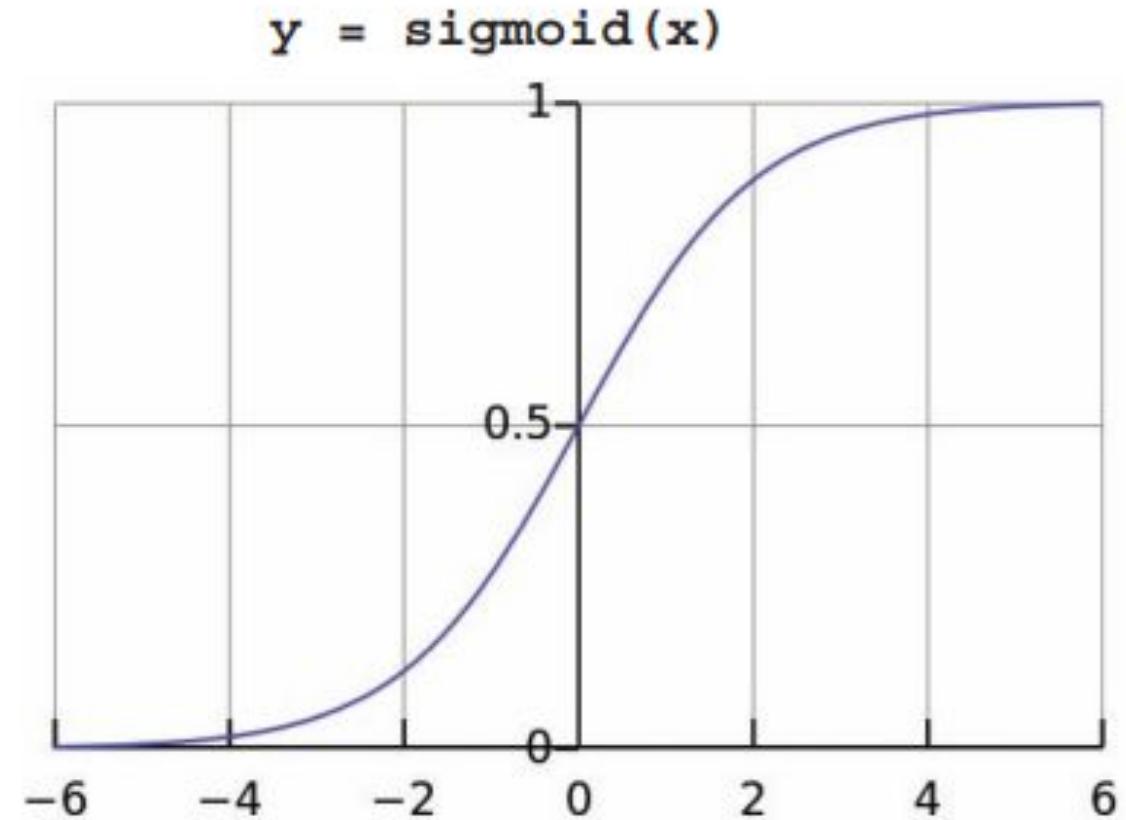
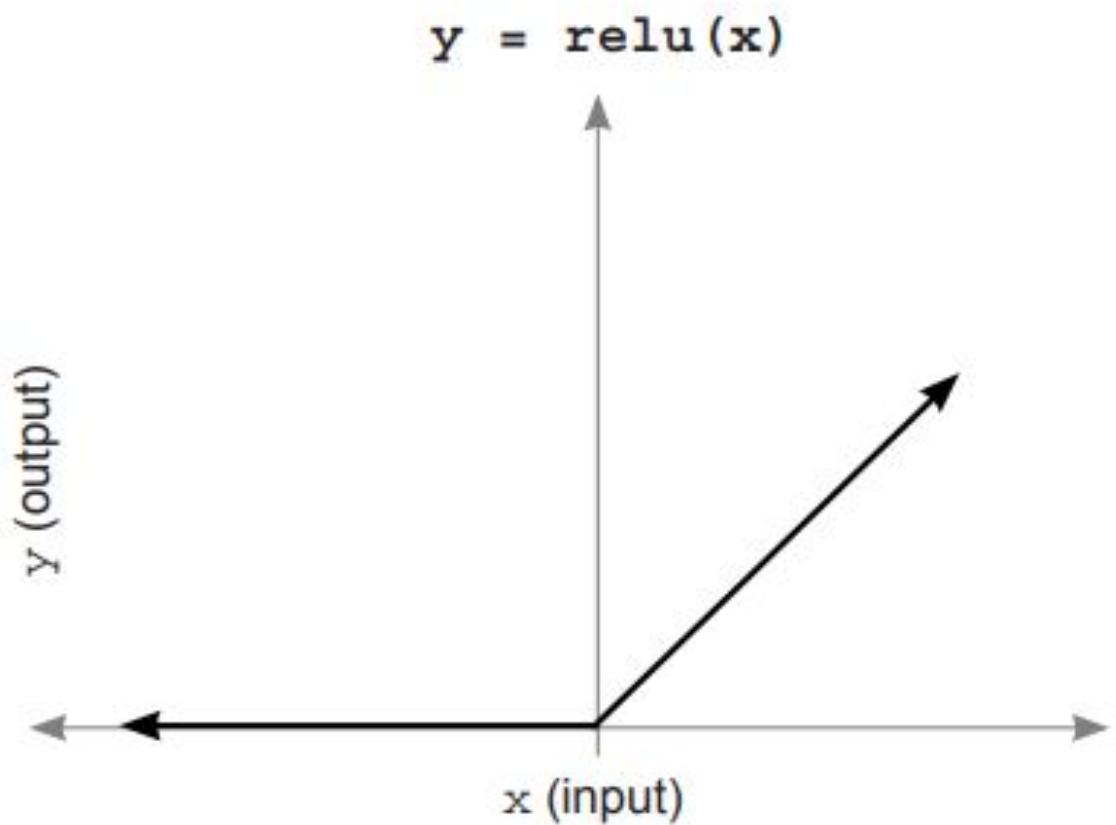
- The important thing in these figures is that the slope is an indicator of how much a tiny change to the input affects the output.
- You want to modify the incoming delta (from the following layer) to take into account whether a weight update before this node would have any effect.
- Remember, the end goal is to adjust weights to reduce error.

Multiplying delta by the slope

To compute layer_delta, multiply the backpropagated delta by the layer's slope.



Multiplying delta by the slope





Converting output to slope (derivative)

Most great activations can convert their output to their slope.
(Efficiency win!).

- Now that you know that adding an activation to a layer changes how to compute delta for that layer, let's discuss how the industry does this efficiently.
- The new operation necessary is the computation of the derivative of whatever nonlinearity was used.

Converting output to slope (derivative)

Function	Forward prop	Backprop delta
relu	<code>ones_and_zeros = (input > 0) output = input*ones_and_zeros</code>	<code>mask = output > 0 deriv = output * mask</code>
sigmoid	<code>output = 1/(1 + np.exp(-input))</code>	<code>deriv = output*(1-output)</code>
tanh	<code>output = np.tanh(input)</code>	<code>deriv = 1 - (output**2)</code>
softmax	<code>temp = np.exp(input) output /= np.sum(temp)</code>	<code>temp = (output - true) output = temp/len(true)</code>



Upgrading the MNIST network

Let's upgrade the MNIST network to reflect what you've learned.

- Theoretically, the tanh function should make for a better hidden-layer activation, and softmax should make for a better output-layer activation function.
- When we test them, they do in fact reach a higher score. But things aren't always as simple as they seem.
- I had to make a couple of adjustments in order to tune the network properly with these new activations.

Upgrading the MNIST network

```
import numpy as np, sys
np.random.seed(1)

from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

images, labels = (x_train[0:1000].reshape(1000,28*28) \
                   / 255, y_train[0:1000])
one_hot_labels = np.zeros((len(labels),10))
for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test),28*28) / 255
test_labels = np.zeros((len(y_test),10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1

def tanh(x):
    return np.tanh(x)
def tanh2deriv(output):
    return 1 - (output ** 2)
def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis=1, keepdims=True)
```

```
alpha, iterations, hidden_size = (2, 300, 100)
pixels_per_image, num_labels = (784, 10)
batch_size = 100

weights_0_1 = 0.02*np.random.random((pixels_per_image,hidden_size))-0.01
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    correct_cnt = 0
    for i in range(int(len(images) / batch_size)):
        batch_start, batch_end=((i * batch_size),((i+1)*batch_size))
        layer_0 = images[batch_start:batch_end]
        layer_1 = tanh(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2,size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = softmax(np.dot(layer_1,weights_1_2))

        for k in range(batch_size):
            correct_cnt += int(np.argmax(layer_2[k:k+1]) == \
                               np.argmax(labels[batch_start+k:batch_start+k+1]))
    layer_2_delta = (labels[batch_start:batch_end]-layer_2) \
                    / (batch_size * layer_2.shape[0])
    layer_1_delta = layer_2_delta.dot(weights_1_2.T) \
                    * tanh2deriv(layer_1)
    layer_1_delta *= dropout_mask
```

```

weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)
test_correct_cnt = 0

for i in range(len(test_images)):

    layer_0 = test_images[i:i+1]
    layer_1 = tanh(np.dot(layer_0, weights_0_1))
    layer_2 = np.dot(layer_1, weights_1_2)
    test_correct_cnt += int(np.argmax(layer_2) == \
                           np.argmax(test_labels[i:i+1]))

if(j % 10 == 0):
    sys.stdout.write("\n" + "I:" + str(j) + \
                     " Test-Acc:" + str(test_correct_cnt/float(len(test_images))) + \
                     " Train-Acc:" + str(correct_cnt/float(len(images))))

```

I:0	Test-Acc:0.394	Train-Acc:0.156	I:150	Test-Acc:0.8555	Train-Acc:0.914
I:10	Test-Acc:0.6867	Train-Acc:0.723	I:160	Test-Acc:0.8577	Train-Acc:0.925
I:20	Test-Acc:0.7025	Train-Acc:0.732	I:170	Test-Acc:0.8596	Train-Acc:0.918
I:30	Test-Acc:0.734	Train-Acc:0.763	I:180	Test-Acc:0.8619	Train-Acc:0.933
I:40	Test-Acc:0.7663	Train-Acc:0.794	I:190	Test-Acc:0.863	Train-Acc:0.933
I:50	Test-Acc:0.7913	Train-Acc:0.819	I:200	Test-Acc:0.8642	Train-Acc:0.926
I:60	Test-Acc:0.8102	Train-Acc:0.849	I:210	Test-Acc:0.8653	Train-Acc:0.931
I:70	Test-Acc:0.8228	Train-Acc:0.864	I:220	Test-Acc:0.8668	Train-Acc:0.93
I:80	Test-Acc:0.831	Train-Acc:0.867	I:230	Test-Acc:0.8672	Train-Acc:0.937
I:90	Test-Acc:0.8364	Train-Acc:0.885	I:240	Test-Acc:0.8681	Train-Acc:0.938
I:100	Test-Acc:0.8407	Train-Acc:0.88	I:250	Test-Acc:0.8687	Train-Acc:0.937
I:110	Test-Acc:0.845	Train-Acc:0.891	I:260	Test-Acc:0.8684	Train-Acc:0.945
I:120	Test-Acc:0.8481	Train-Acc:0.90	I:270	Test-Acc:0.8703	Train-Acc:0.951
I:130	Test-Acc:0.8505	Train-Acc:0.90	I:280	Test-Acc:0.8699	Train-Acc:0.949
I:140	Test-Acc:0.8526	Train-Acc:0.90	I:290	Test-Acc:0.8701	Train-Acc:0.94

Lesson 10 neural learning about edges and corners: intro to convolutional neural networks





Intro to convolutional neural networks

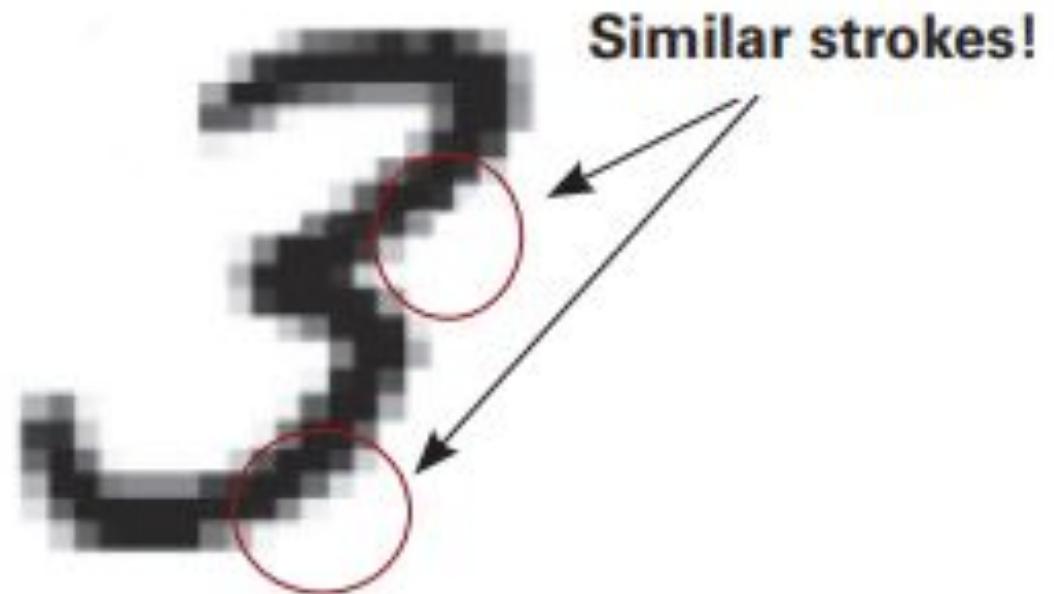


In this lesson

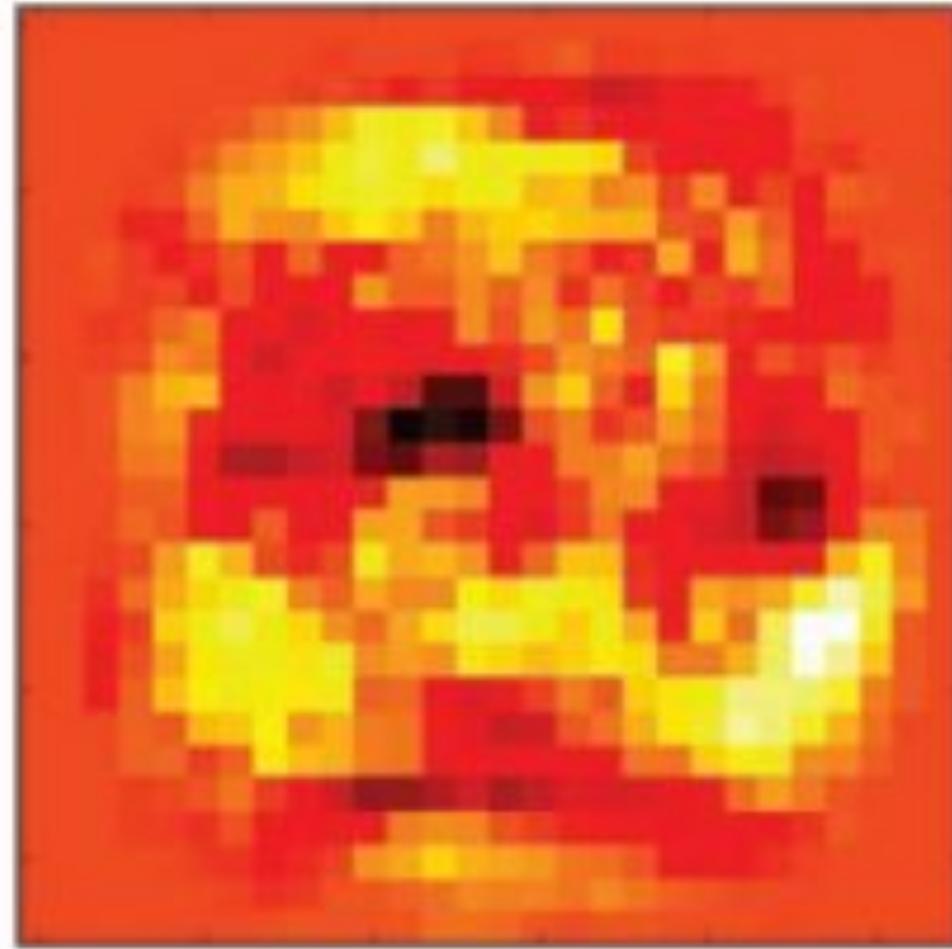
- Reusing weights in multiple places
- The convolutional layer

Reusing weights in multiple places

If you need to detect the same feature in multiple places, use the same weights!.



Reusing weights in multiple places



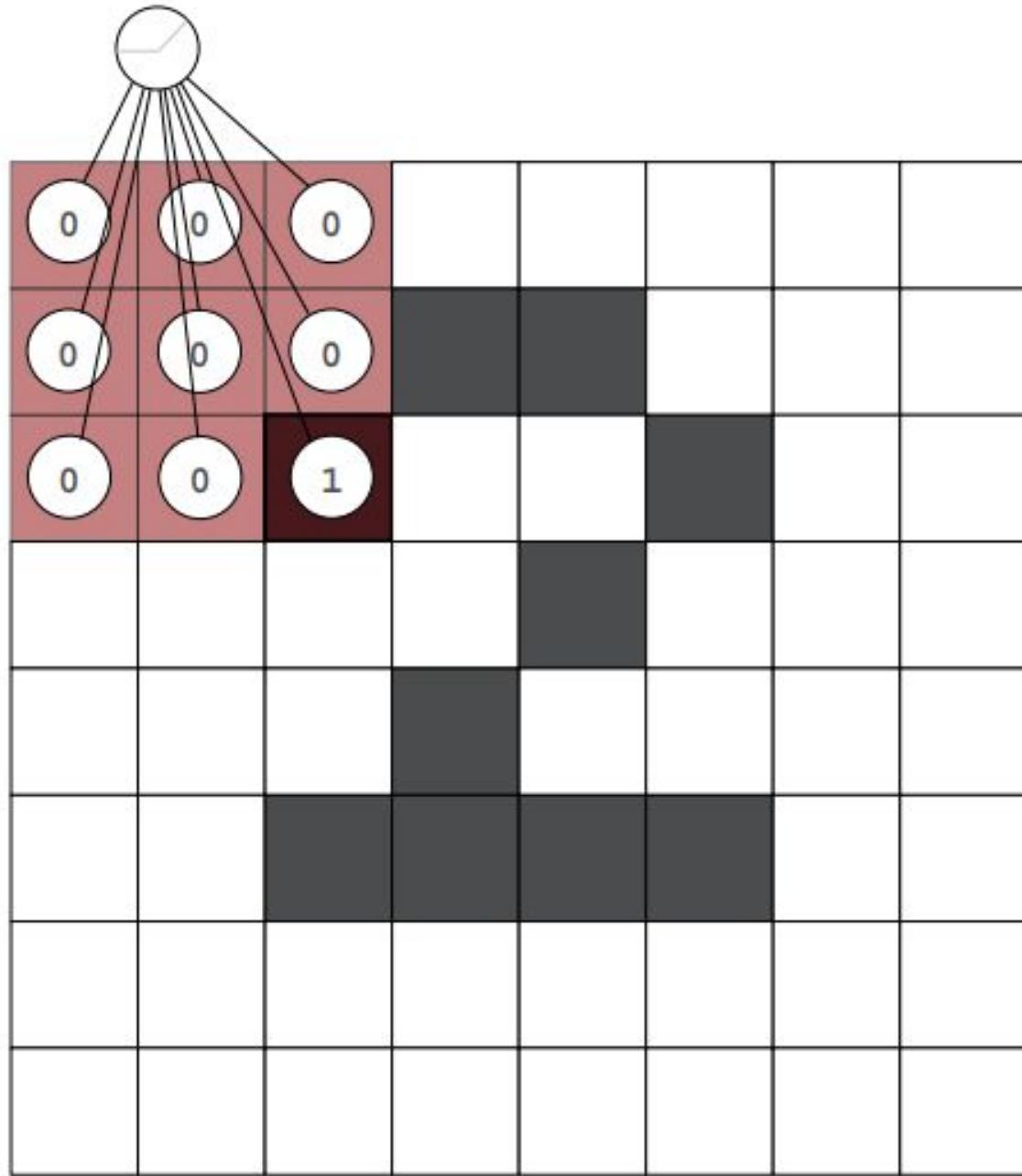


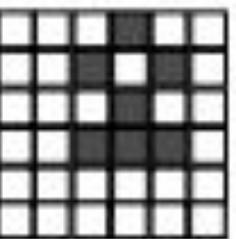
The convolutional layer



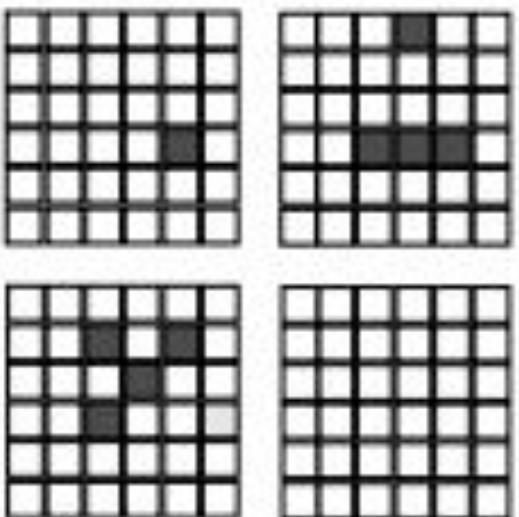
Lots of very small linear layers are reused in every position, instead of a single big one.

- The core idea behind a convolutional layer is that instead of having a large, dense linear layer with a connection from every input to every output, you instead have lots of very small linear layers, usually with fewer than 25 inputs and a single output, which you use in every input position.





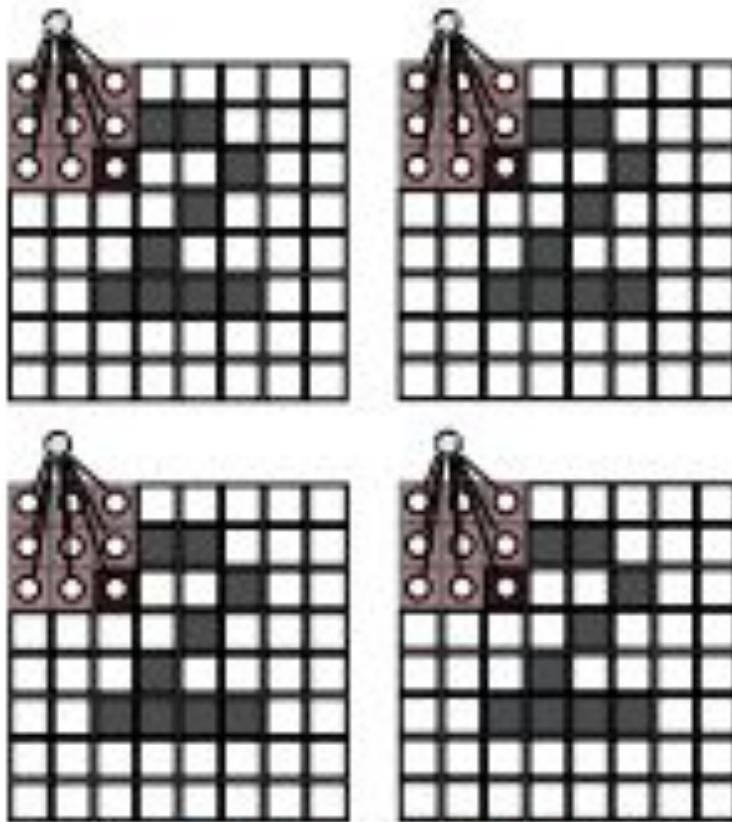
The max value of each kernel's output forms a meaningful representation and is passed to the next layer.



Outputs from each of the four kernels in each position



each position



Four convolutional kernels predicting over the same 2



A simple implementation in NumPy

Just think mini-linear layers, and you already know what
you
need to know.

- Let's start with forward propagation. This method shows how to select a subregion in a batch of images in NumPy.
 - Note that it selects the same subregion for the entire
- . . .

```
def get_image_section(layer, row_from, row_to, col_from, col_to):  
    sub_section = layer[:,row_from:row_to,col_from:col_to]  
    return subsection.reshape(-1,1, row_to-row_from, col_to-col_from)
```

Now, let's see how this method is used. Because it selects a subsection of a batch of input images, you need to call it multiple times (on every location within the image). Such a `for` loop looks something like this:

```
layer_0 = images[batch_start:batch_end]
layer_0 = layer_0.reshape(layer_0.shape[0], 28, 28)
layer_0.shape

sects = list()
for row_start in range(layer_0.shape[1]-kernel_rows):
    for col_start in range(layer_0.shape[2] - kernel_cols):
        sect = get_image_section(layer_0,
                                  row_start,
                                  row_start+kernel_rows,
                                  col_start,
                                  col_start+kernel_cols)
        sects.append(sect)

expanded_input = np.concatenate(sects, axis=1)
es = expanded_input.shape
flattened_input = expanded_input.reshape(es[0]*es[1], -1)
```



A simple implementation in NumPy

- If you instead forward propagate using a linear layer with n output neurons, it will generate the outputs that are the same as predicting n linear layers (kernels) in every input position of the image.
- You do it this way because it makes the code both simpler and faster:

```
kernels = np.random.random( (kernel_rows*kernel_cols, num_kernels) )  
        ...  
kernel_output = flattened_input.dot(kernels)
```



- The following listing shows the entire NumPy implementation:

```
import numpy as np, sys
np.random.seed(1)

from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

images, labels = (x_train[0:1000].reshape(1000,28*28) / 255,
                   y_train[0:1000])

one_hot_labels = np.zeros((len(labels),10))
for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test),28*28) / 255
test_labels = np.zeros((len(y_test),10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1

def tanh(x):
    return np.tanh(x)

def tanh2deriv(output):
    return 1 - (output ** 2)
```

```
def softmax(x):
    temp = np.exp(x)
    return temp / np.sum(temp, axis=1, keepdims=True)

alpha, iterations = (2, 300)
pixels_per_image, num_labels = (784, 10)
batch_size = 128

input_rows = 28
input_cols = 28

kernel_rows = 3
kernel_cols = 3
num_kernels = 16

hidden_size = ((input_rows - kernel_rows) *
               (input_cols - kernel_cols)) * num_kernels

kernels = 0.02*np.random.random((kernel_rows*kernel_cols,
                                 num_kernels))-0.01

weights_1_2 = 0.2*np.random.random((hidden_size,
                                    num_labels)) - 0.1

def get_image_section(layer, row_from, row_to, col_from, col_to):
    section = layer[:,row_from:row_to,col_from:col_to]
    return section.reshape(-1,1,row_to-row_from, col_to-col_from)
```

```
for j in range(iterations):
    correct_cnt = 0
    for i in range(int(len(images) / batch_size)):
        batch_start, batch_end=((i * batch_size),((i+1)*batch_size))
        layer_0 = images[batch_start:batch_end]
        layer_0 = layer_0.reshape(layer_0.shape[0],28,28)
        layer_0.shape

        sects = list()
        for row_start in range(layer_0.shape[1]-kernel_rows):
            for col_start in range(layer_0.shape[2] - kernel_cols):
                sect = get_image_section(layer_0,
                                         row_start,
                                         row_start+kernel_rows,
                                         col_start,
                                         col_start+kernel_cols)
                sects.append(sect)

        expanded_input = np.concatenate(sects, axis=1)
        es = expanded_input.shape
        flattened_input = expanded_input.reshape(es[0]*es[1],-1)

        kernel_output = flattened_input.dot(kernels)
        layer_1 = tanh(kernel_output.reshape(es[0],-1))
        dropout_mask = np.random.randint(2,size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = softmax(np.dot(layer_1,weights_1_2))
```

```
for k in range(batch_size):
    labelset = labels[batch_start+k:batch_start+k+1]
    _inc = int(np.argmax(layer_2[k:k+1]) ==
               np.argmax(labelset))
    correct_cnt += _inc

    layer_2_delta = (labels[batch_start:batch_end]-layer_2) \
                    / (batch_size * layer_2.shape[0])
    layer_1_delta = layer_2_delta.dot(weights_1_2.T) * \
                    tanh2deriv(layer_1)
    layer_1_delta *= dropout_mask
    weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
    l1d_reshape = layer_1_delta.reshape(kernel_output.shape)
    k_update = flattened_input.T.dot(l1d_reshape)
    kernels -= alpha * k_update

test_correct_cnt = 0

for i in range(len(test_images)):

    layer_0 = test_images[i:i+1]
    layer_0 = layer_0.reshape(layer_0.shape[0], 28, 28)
    layer_0.shape

    sects = list()
    for row_start in range(layer_0.shape[1]-kernel_rows):
        for col_start in range(layer_0.shape[2] - kernel_cols):
            sect = get_image_section(layer_0,
                                      row_start,
                                      row_start+kernel_rows,
```

```

        col_start,
        col_start+kernel_cols)
sects.append(sect)

expanded_input = np.concatenate(sects, axis=1)
es = expanded_input.shape
flattened_input = expanded_input.reshape(es[0]*es[1], -1)

kernel_output = flattened_input.dot(kernels)
layer_1 = tanh(kernel_output.reshape(es[0], -1))
layer_2 = np.dot(layer_1, weights_1_2)

test_correct_cnt += int(np.argmax(layer_2) ==
                        np.argmax(test_labels[i:i+1]))

if(j % 1 == 0):
    sys.stdout.write("\n" + \
        "I:" + str(j) + \
        " Test-Acc:" + str(test_correct_cnt/float(len(test_images)))+\
        " Train-Acc:" + str(correct_cnt/float(len(images))))
```

I:0 Test-Acc:0.0288 Train-Acc:0.055
 I:1 Test-Acc:0.0273 Train-Acc:0.037
 I:2 Test-Acc:0.028 Train-Acc:0.037
 I:3 Test-Acc:0.0292 Train-Acc:0.04
 I:4 Test-Acc:0.0339 Train-Acc:0.046
 I:5 Test-Acc:0.0478 Train-Acc:0.068
 I:6 Test-Acc:0.076 Train-Acc:0.083
 I:7 Test-Acc:0.1316 Train-Acc:0.096
 I:8 Test-Acc:0.2137 Train-Acc:0.127

....

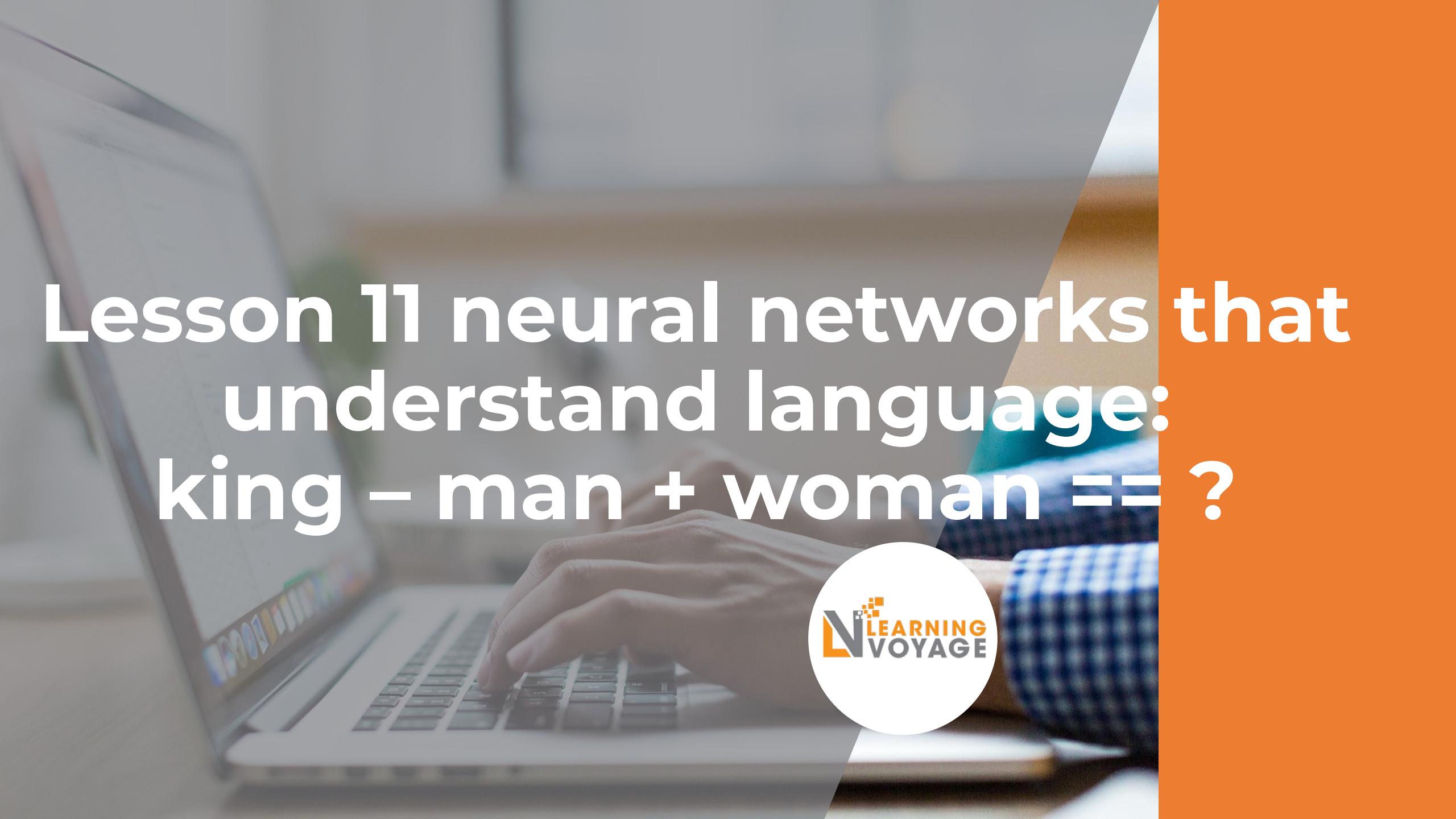
I:297 Test-Acc:0.8774 Train-Acc:0.816
 I:298 Test-Acc:0.8774 Train-Acc:0.804
 I:299 Test-Acc:0.8774 Train-Acc:0.814



Summary

Reusing weights is one of the most important innovations in deep learning

- Convolutional neural networks are a more general development than you might realize.
 - The notion of reusing weights to increase accuracy is hugely important and has an intuitive basis. Consider what you need to understand in order to detect that a cat is in an image.
 - You first need to understand colors, then lines and edges, corners and small shapes, and eventually the combination of such lower-level features that correspond to a cat.
- 



Lesson 11 neural networks that understand language:
king - man + woman == ?





king - man + woman == ?



In this lesson

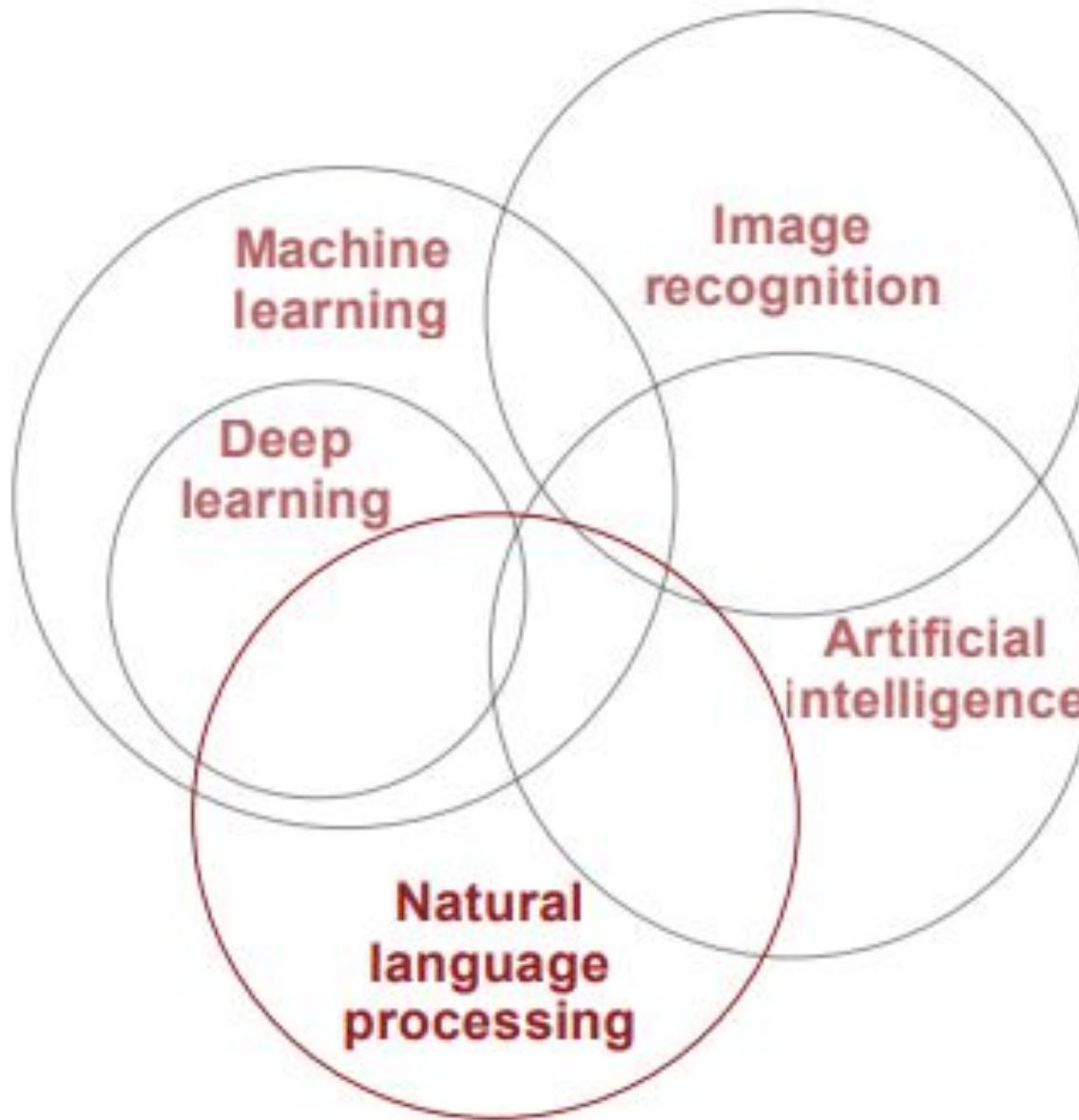
- Natural language processing (NLP)
 - Supervised NLP
 - Capturing word correlation in input data
 - Intro to an embedding layer
 - Neural architecture
 - Comparing word embeddings
 - Filling in the blank
 - Meaning is derived from loss
 - Word analogies
- 



What does it mean to understand language?

What kinds of predictions do people make about language?

- Up until now, we've been using neural networks to model image data.
- But neural networks can be used to understand a much wider variety of datasets.
- Exploring new datasets also teaches us a lot about neural networks in general, because different datasets often justify different styles of neural network training according to the challenges hidden in the data.





Natural language processing (NLP)

NLP is divided into a collection of tasks or challenges.

- Using the *characters* of a document to predict *where words start and end*.
 - Using the *words* of a document to predict *where sentences start and end*.
 - Using the *words in a sentence* to predict *the part of speech for each word*.
 - Using *words in a sentence* to predict *where phrases start and end*.
 - Using *words in a sentence* to predict *where named entity (person, place, thing) references start and end*.
 - Using *sentences in a document* to predict *which pronouns refer to the same person / place / thing*.
 - Using *words in a sentence* to predict the *sentiment* of a sentence.
- 



Natural language processing (NLP)

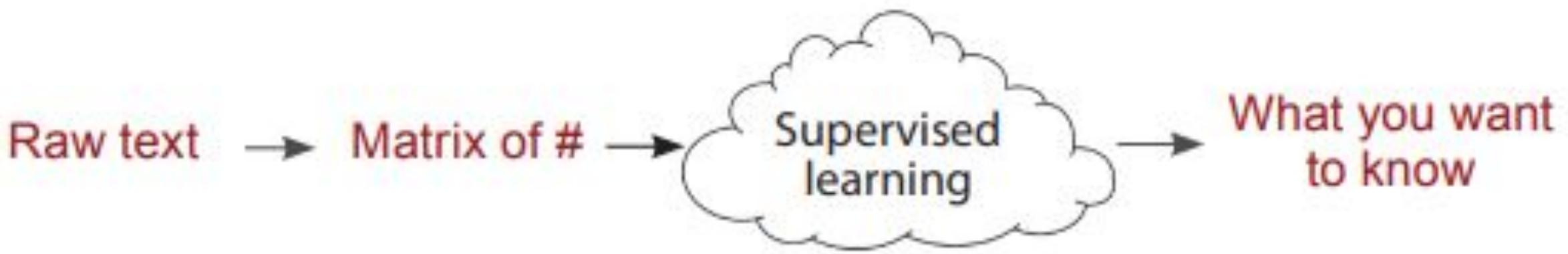
- Generally speaking, NLP tasks seek to do one of three things: label a region of text (such as part-of-speech tagging, sentiment classification, or named-entity recognition);
 - link two or more regions of text (such as coreference, which tries to answer whether two mentions of a real-world thing are in fact referencing the same real-world thing, where the real-world thing is generally a person, place, or some other named entity);
 - or try to fill in missing information (missing words) based on context.
- 

Supervised NLP

Words go in, and predictions come out.



Supervised NLP





Supervised NLP



- How should we convert text to numbers? Answering that question requires some thought regarding the problem. Remember, neural networks look for correlation between their input and output layers.
- Thus, we want to convert text into numbers in such a way that the correlation between input and output is most obvious to the network.
- This will make for faster training and better generalization.



IMDB movie reviews dataset



You can predict whether people post positive or negative reviews.

The IMDB movie reviews dataset is a collection of review -> rating pairs that often look like the following (this is an imitation, not pulled from IMDB):

“ This movie was terrible! The plot was dry, the acting unconvincing, and I spilled popcorn on my shirt.” ”

Rating: 1 (stars)



IMDB movie reviews dataset



- The entire dataset consists of around 50,000 of these pairs, where the input reviews are usually a few sentences and the output ratings are between 1 and 5 stars.
 - People consider it a sentiment dataset because the stars are indicative of the overall sentiment of the movie review.
 - But it should be obvious that this sentiment dataset might be very different from other sentiment datasets, such as product reviews or hospital patient reviews.
- 



Capturing word correlation in input data

Bag of words: Given a review's vocabulary, predict the sentiment.

- If you observe correlation between the vocabulary of an IMDB review and its rating, then you can proceed to the next step: creating an input matrix that represents the vocabulary of a movie review.
- What's commonly done in this case is to create a matrix where each row (vector) corresponds to each movie review, and each column represents whether a review contains a particular word in the vocabulary

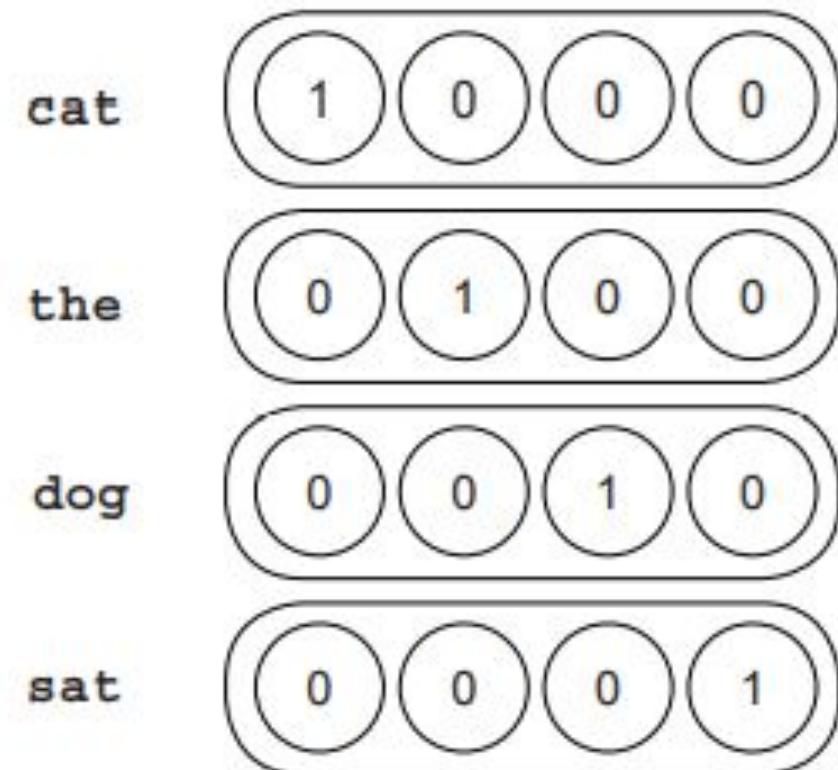
Capturing word correlation in input data

```
import numpy as np

onehots = {}
onehots['cat'] = np.array([1,0,0,0])
onehots['the'] = np.array([0,1,0,0])
onehots['dog'] = np.array([0,0,1,0])
onehots['sat'] = np.array([0,0,0,1])

sentence = ['the', 'cat', 'sat']
x = word2hot[sentence[0]] + \
    word2hot[sentence[1]] + \
    word2hot[sentence[2]]

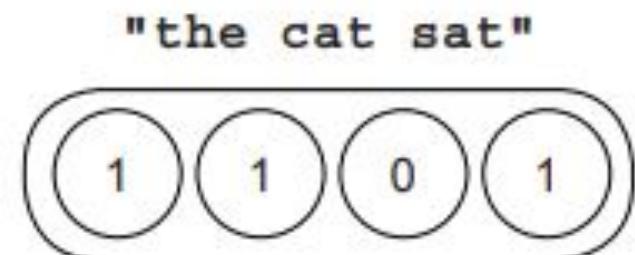
print("Sent Encoding:" + str(x))
```



Capturing word correlation in input data

- As you can see, we create a vector for each term in the vocabulary, and this allows you to use simple vector addition to create a vector representing a subset of the total vocabulary (such as a subset corresponding to the words in a sentence).

Output: Sent Encoding: [1 1 0 1]





Predicting movie reviews

With the encoding strategy and the previous network, you can predict sentiment..

- Using the strategy we just identified, you can build a vector for each word in the sentiment dataset and use the previous two-layer network to predict sentiment.
 - I'll show you the code, but I strongly recommend attempting this from memory.
 - Open a new Jupyter notebook, load in the dataset, build your one-hot vectors, and then build a neural network to predict the rating of each movie review (positive or negative).
- 

```
import sys

f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()

f = open('labels.txt')
raw_labels = f.readlines()
f.close()

tokens = list(map(lambda x:set(x.split(" ")), raw_reviews))

vocab = set()
for sent in tokens:
    for word in sent:
        if(len(word)>0):
            vocab.add(word)
vocab = list(vocab)

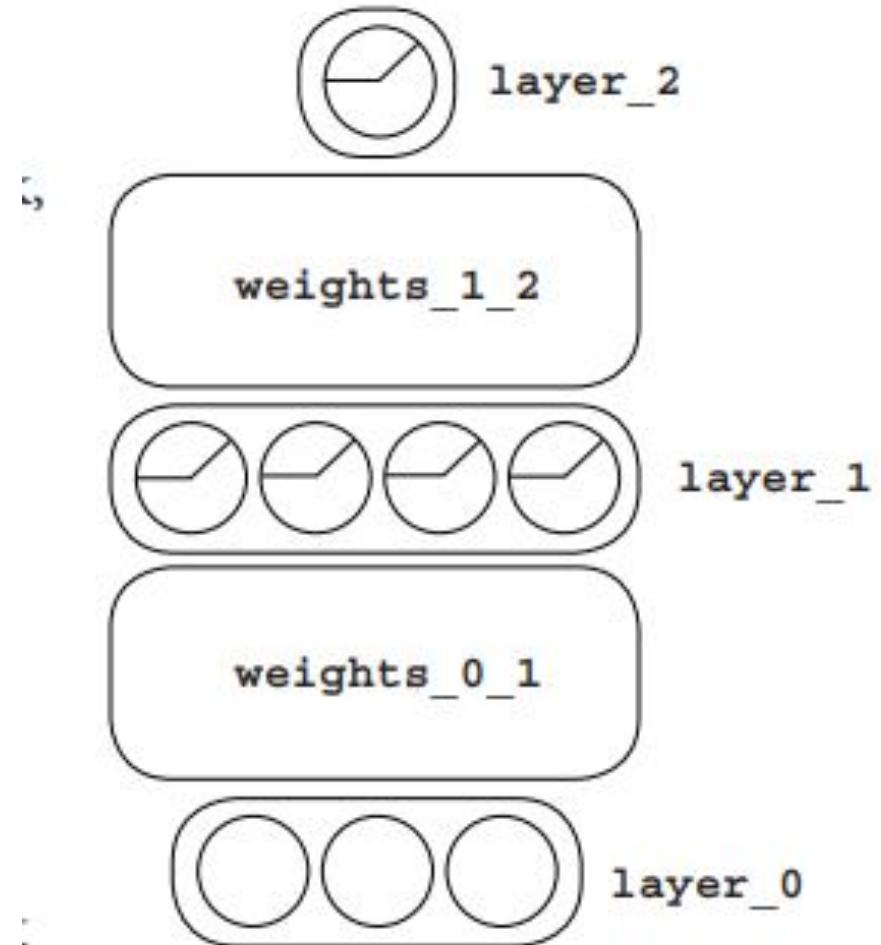
word2index = {}
for i,word in enumerate(vocab):
    word2index[word]=i

input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
        try:
            sent_indices.append(word2index[word])
        except:
            ""
    input_dataset.append(list(set(sent_indices)))

target_dataset = list()
for label in raw_labels:
    if label == 'positive\n':
        target_dataset.append(1)
    else:
        target_dataset.append(0)
```

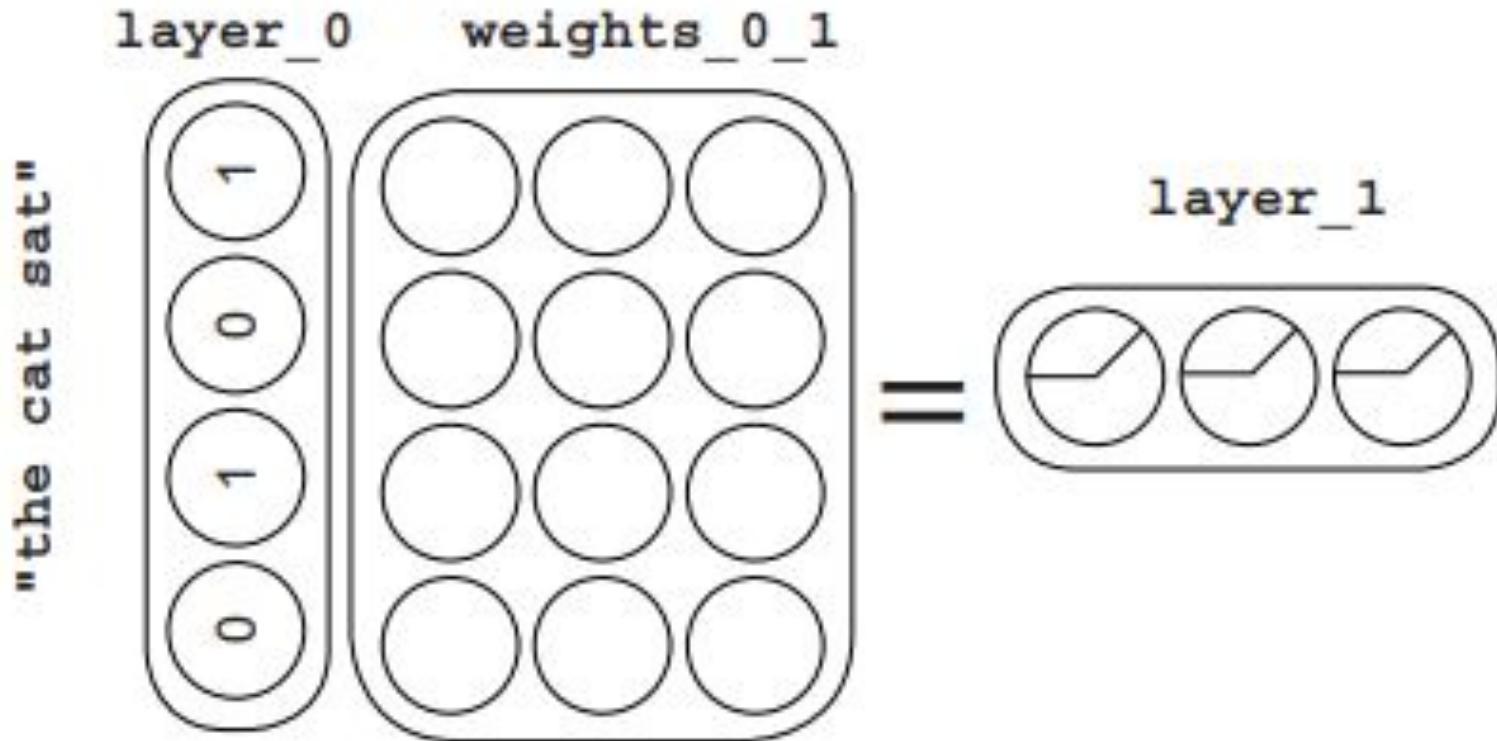
Intro to an embedding layer

Here's one more trick to make the network faster.

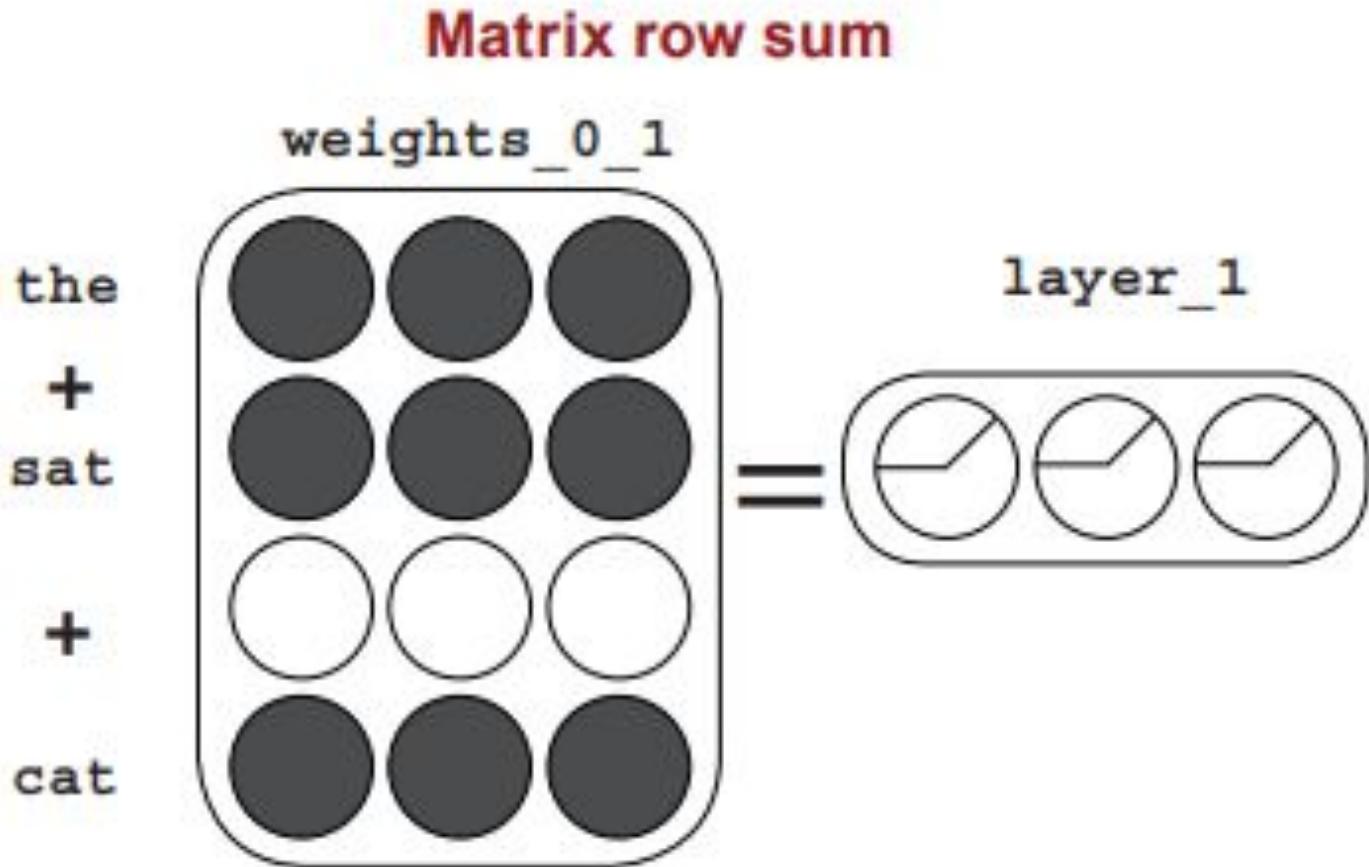


Intro to an embedding layer

One-hot vector-matrix multiplication



Intro to an embedding layer



Intro to an embedding layer

```
import numpy as np
np.random.seed(1)

def sigmoid(x):
    return 1/(1 + np.exp(-x))

alpha, iterations = (0.01, 2)
hidden_size = 100

weights_0_1 = 0.2*np.random.random((len(vocab),hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,1)) - 0.1

correct, total = (0,0)
for iter in range(iterations):

    for i in range(len(input_dataset)-1000): ← Trains on the first
        x,y = (input_dataset[i],target_dataset[i]) ← 24,000 reviews
        layer_1 = sigmoid(np.sum(weights_0_1[x],axis=0)) ← embed + sigmoid
        layer_2 = sigmoid(np.dot(layer_1,weights_1_2)) ← linear + softmax

    ← Compares the prediction with the truth
    layer_2_delta = layer_2 - y ← Backpropagation
    layer_1_delta = layer_2_delta.dot(weights_1_2.T) ← Backpropagation

    weights_0_1[x] -= layer_1_delta * alpha
    weights_1_2 -= np.outer(layer_1,layer_2_delta) * alpha

    if(np.abs(layer_2_delta) < 0.5):
        correct += 1
```



Intro to an embedding layer



```
if(np.abs(layer_2_delta) < 0.5):
    correct += 1
total += 1
if(i % 10 == 9):
    progress = str(i/float(len(input_dataset)))
    sys.stdout.write('\rIter:' + str(iter) +
                     ' Progress:' + progress[2:4] +
                     '.' + progress[4:6] +
                     '% Training Accuracy:' +
                     str(correct/float(total)) + '%')
print()
correct, total = (0, 0)
for i in range(len(input_dataset)-1000, len(input_dataset)):

    x = input_dataset[i]
    y = target_dataset[i]

    layer_1 = sigmoid(np.sum(weights_0_1[x], axis=0))
    layer_2 = sigmoid(np.dot(layer_1, weights_1_2))

    if(np.abs(layer_2 - y) < 0.5):
        correct += 1
    total += 1
print("Test Accuracy:" + str(correct / float(total)))
```



Interpreting the output



What did the neural network learn along the way?

- Here's the output of the movie reviews neural network.
- From one perspective, this is the same correlation summarization we've already discussed:

```
Iter:0 Progress:95.99% Training Accuracy:0.832%
Iter:1 Progress:95.99% Training Accuracy:0.8663333333333333%
Test Accuracy:0.849
```

Interpreting the output

Pos/Neg label

(Neural network)

Review vocab

Interpreting the output

