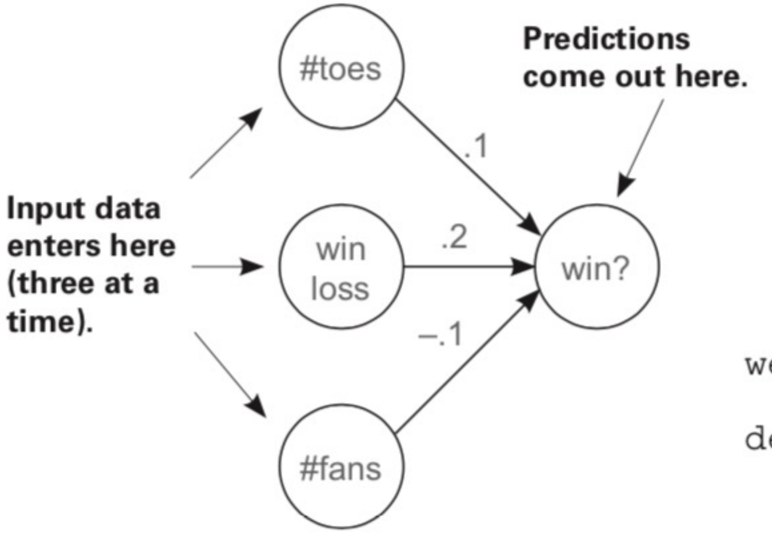
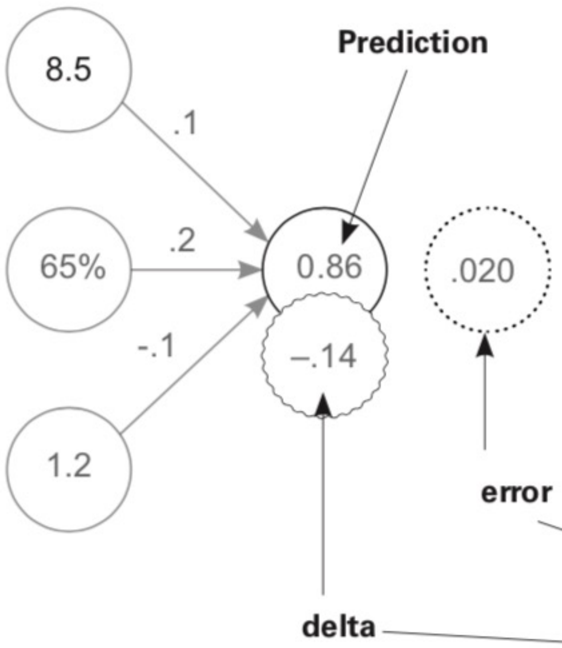
learning multiple weights at a time: generalizing gradient descent | 5

❶ An empty network with multiple 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

weights = [0.1, 0.2, -.1]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

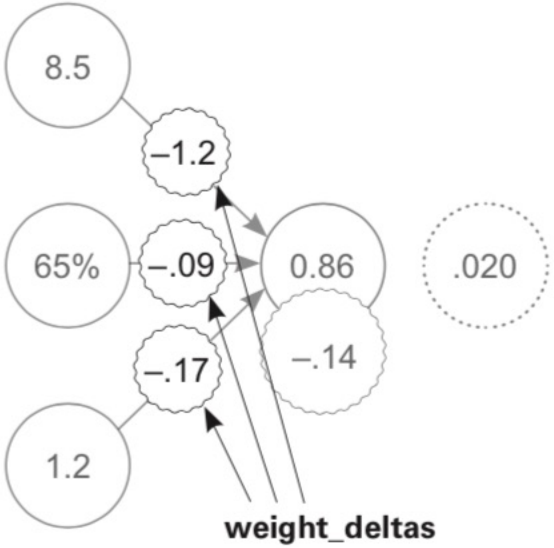
❷ PREDICT + COMPARE: Making a prediction, and calculating error and delta



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
```

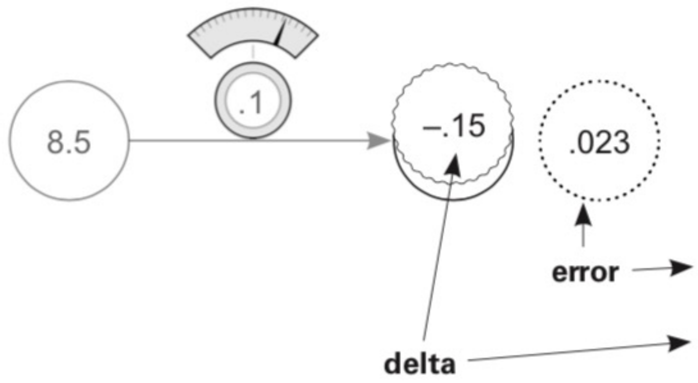
❸ LEARN: Calculating each weight_delta and putting it on each weight




```
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)
```

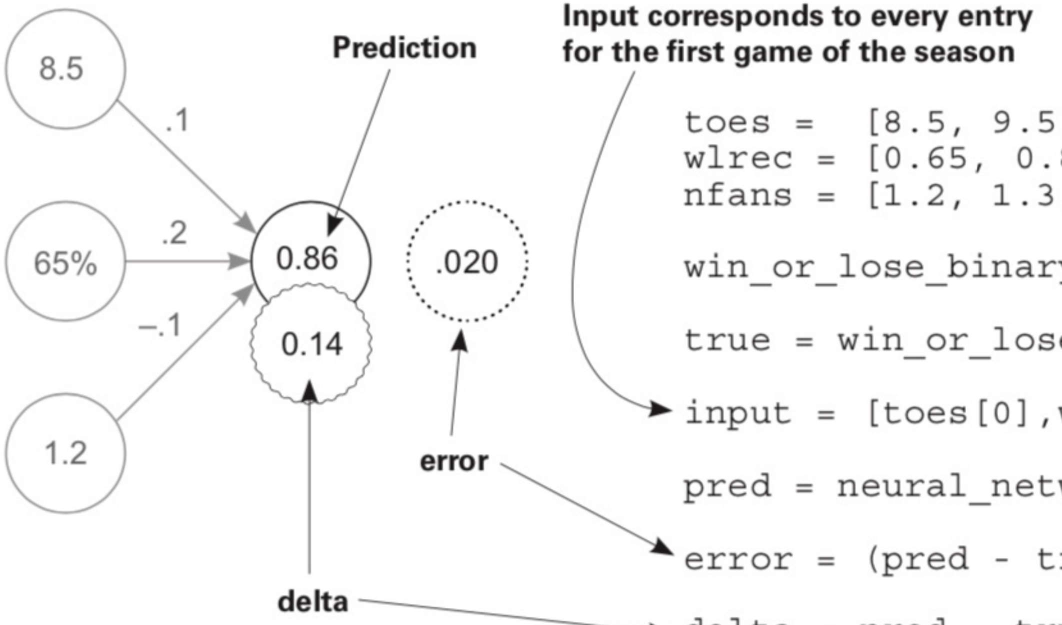
8.5 * -0.14 = -1.19 = **weight_deltas[0]**
0.65 * -0.14 = -0.091 = **weight_deltas[1]**
1.2 * -0.14 = -0.168 = **weight_deltas[2]**

❶ 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
```

❷ Multi-input: 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]

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
```

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

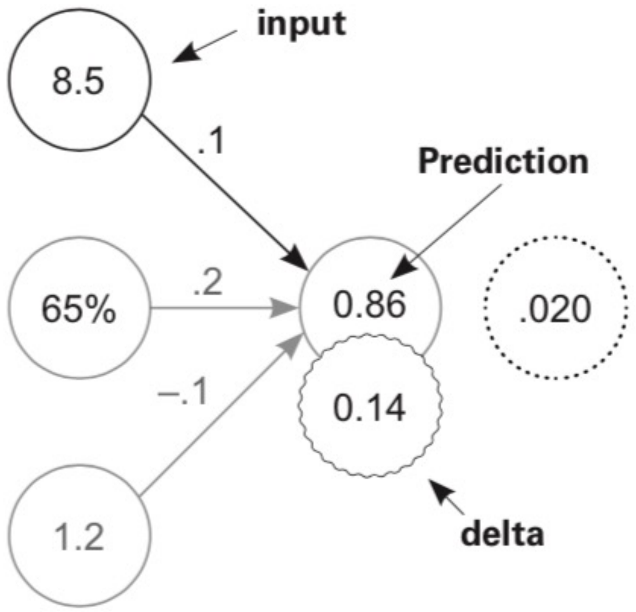
delta

A measure of how much higher or lower you want a node's value to be, to predict perfectly given the current training example.

weight_delta

A derivative-based estimate of the direction and amount you should move a weight to reduce `node_delta`, accounting for scaling, negative reversal, and stopping.

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



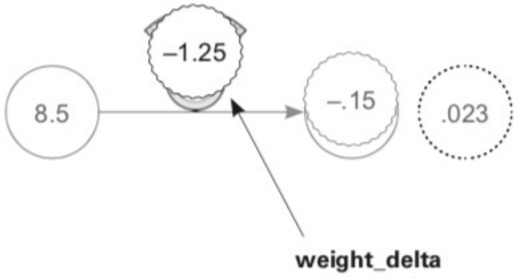
`delta`: Hey, `inputs`—yeah, you three. Next time, predict a little higher.

Single weight: Hmm: if my `input` was 0, then my weight wouldn't have mattered, and I wouldn't change a thing (*stopping*). If my `input` was negative, then I'd want to decrease my weight instead of increase it (*negative reversal*). But my `input` is positive and quite large, so I'm *guessing* that my personal prediction mattered a lot to the aggregated output. I'm going to move my weight up a lot to compensate (*scaling*).

The single weight increases its value.

What did those three properties/statements really say? They all (stopping, negative reversal, and scaling) made an observation of how the weight's role in `delta` was affected by its `input`. Thus, each `weight_delta` is a sort of input-modified version of `delta`.

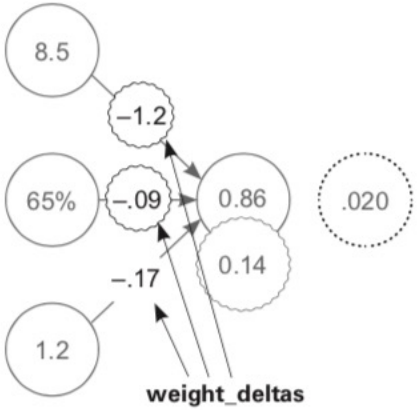
❸ Single input: Calculating weight_delta and putting it on the weight




```
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**

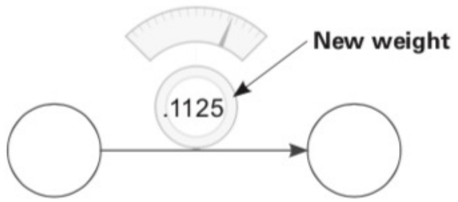
❹ Multi-input: Calculating each weight_delta and putting it on each weight




```
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]**

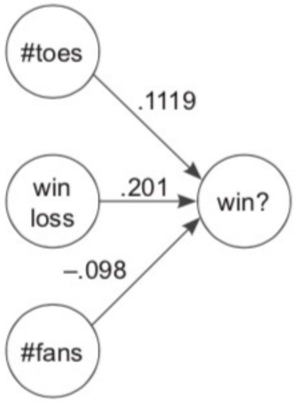
❺ Updating the weight



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
```

❻ 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]
```

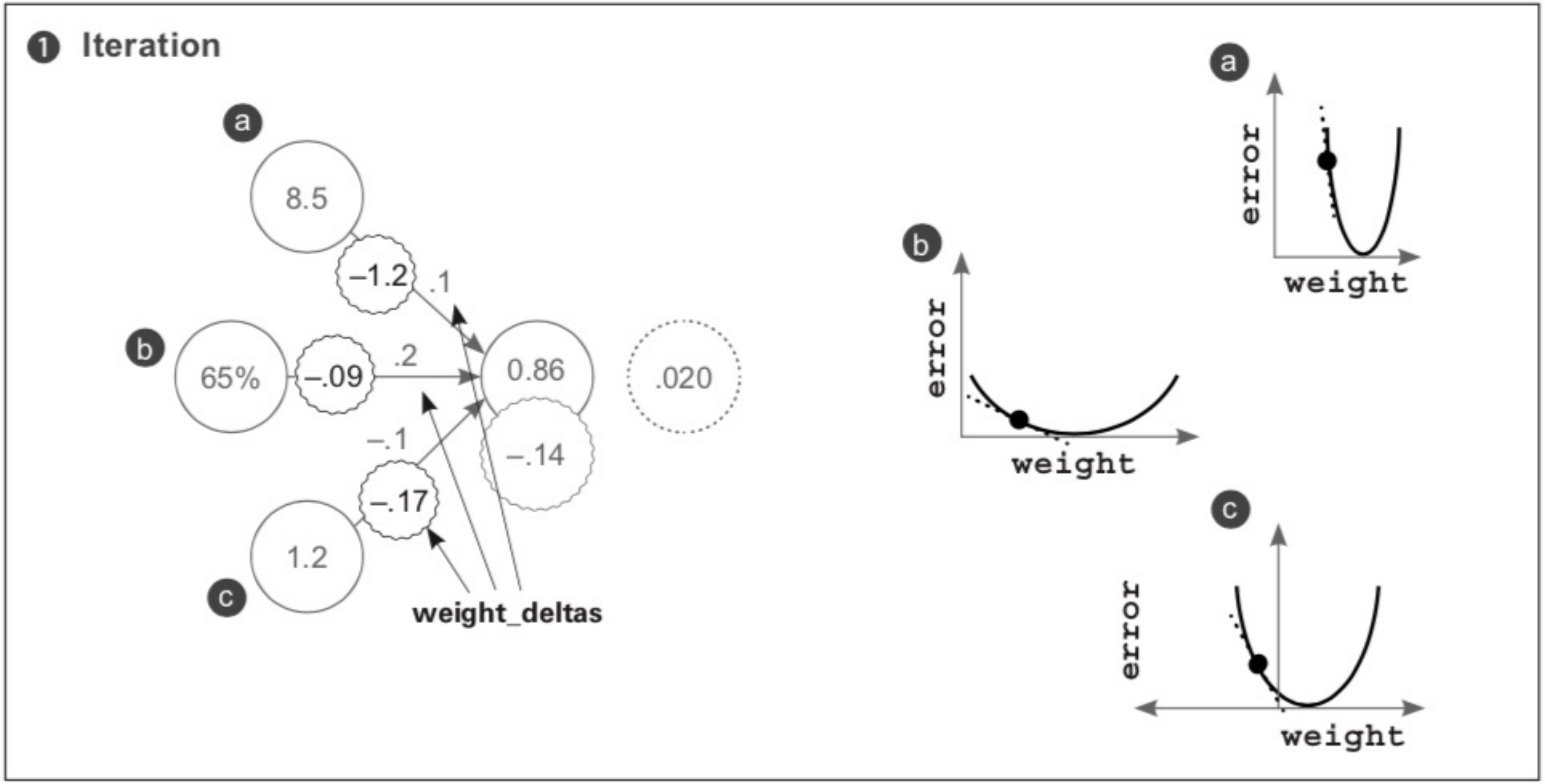
0.1 – (1.19 * 0.01) = 0.1119 = **`weights[0]`**
0.2 – (.091 * 0.01) = 0.2009 = **`weights[1]`**
–0.1 – (.168 * 0.01) = –0.098 = **`weights[2]`**

Let's watch several steps of learning

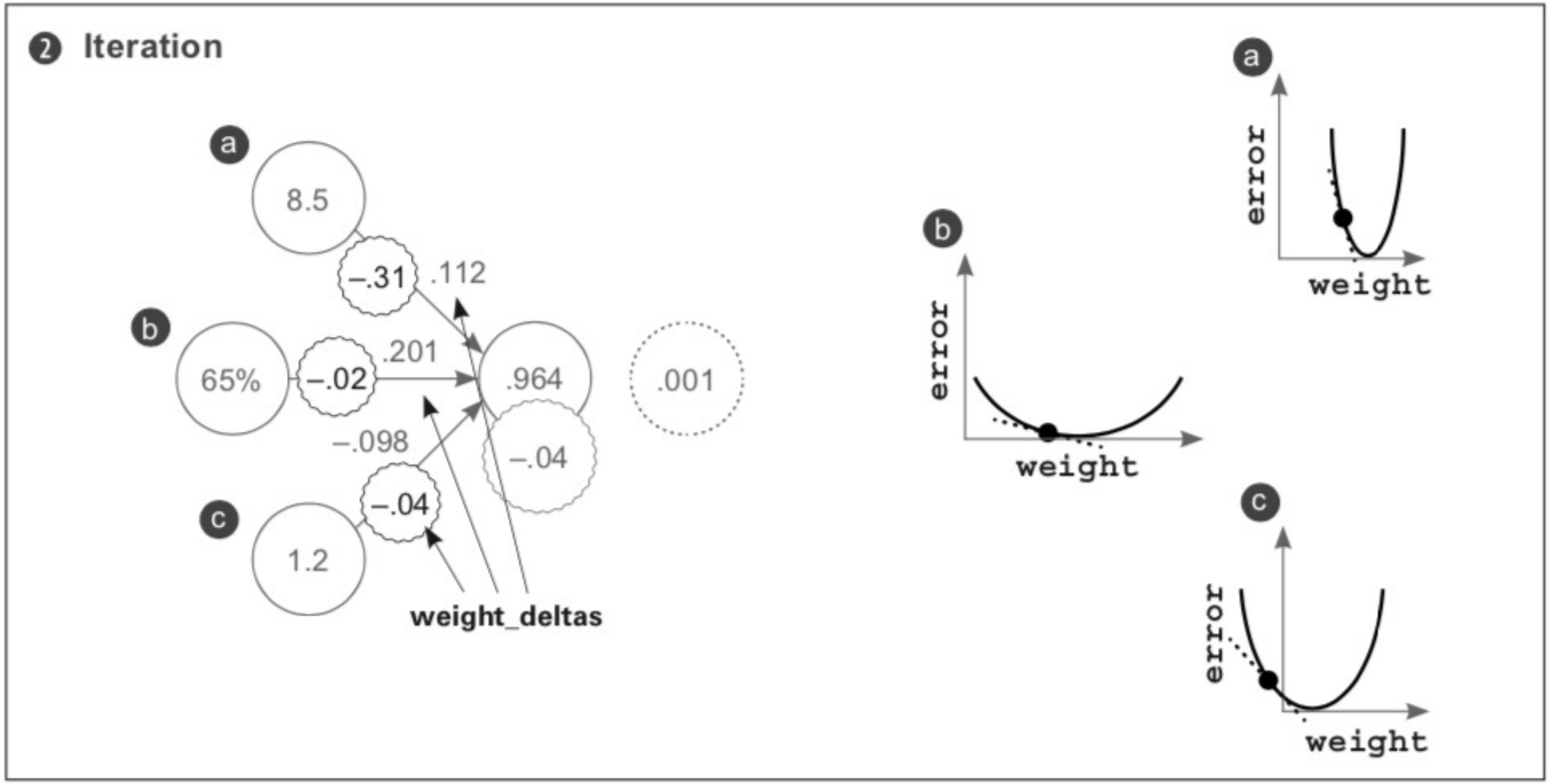
```
def neural_network(input, weights):
  out = 0
  for i in range(len(input)):
    out += (input[i] * weights[i])
  return out

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)

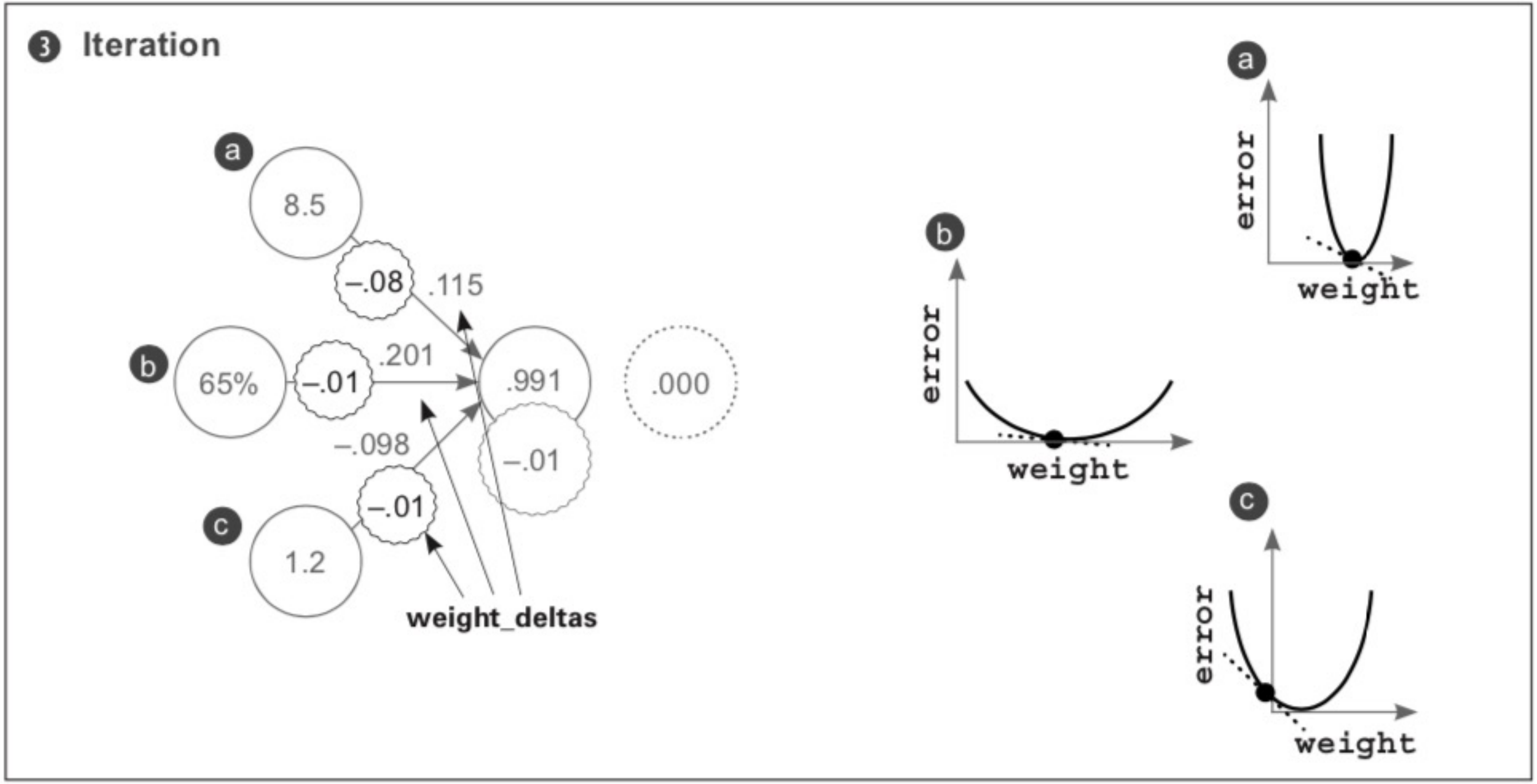
```
for iter in range(3):

  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]
```



We can make three individual error/weight curves, one for each weight. As before, the slopes of these curves (the dotted lines) are reflected by the `weight_delta` values. Notice that ⓐ is steeper than the others. Why is `weight_delta` steeper for ⓐ than the others if they share the same output `delta` and `error` measure? Because ⓐ has an `input` value that's significantly higher than the others and thus, a higher derivative.



2 Iteration
a
8.5
–.31
.112
b
65%
–.02
.201
.964
.001
–.098
–.04
c
1.2
–.04
weight_deltas
a
error
weight
b
error
weight
c
error
weight



Here are a few additional takeaways. Most of the learning (weight changing) was performed on the weight with the largest input a, because the input changes the slope significantly. This isn't necessarily advantageous in all settings. A subfield called *normalization* helps encourage learning across all weights despite dataset characteristics such as this. This significant difference in slope forced me to set `alpha` lower than I wanted (0.01 instead of 0.1). Try setting `alpha` to 0.1: do you see how a causes it to diverge?

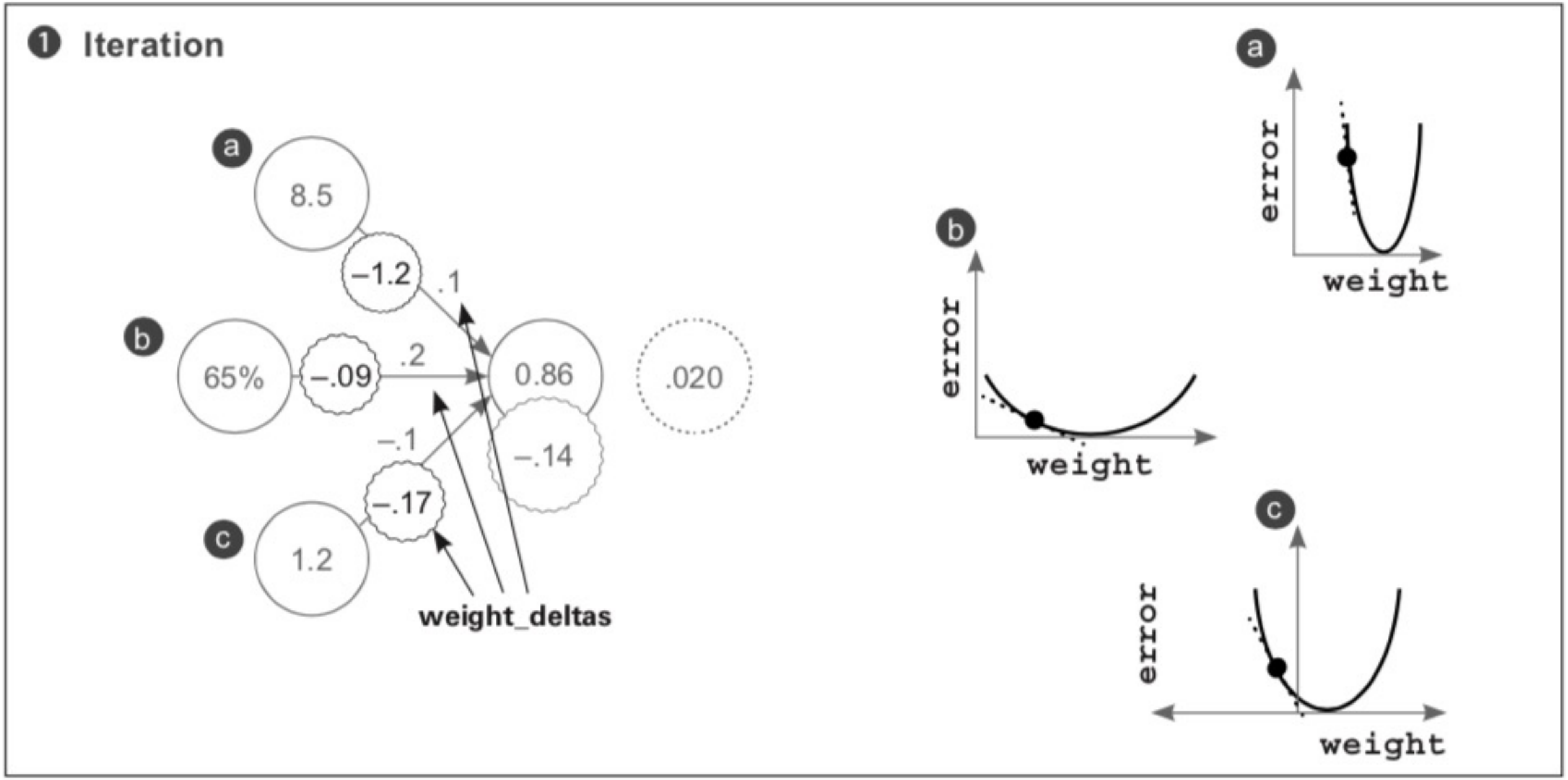
Freezing one weight: What does it do?

This experiment is a bit advanced in terms of theory, but I think it's a great exercise to understand how the weights affect each other. You're going to train again, except weight ⓐ won't ever be adjusted. You'll try to learn the training example using only weights ⓑ and ⓒ (`weights[1]` and `weights[2]`).

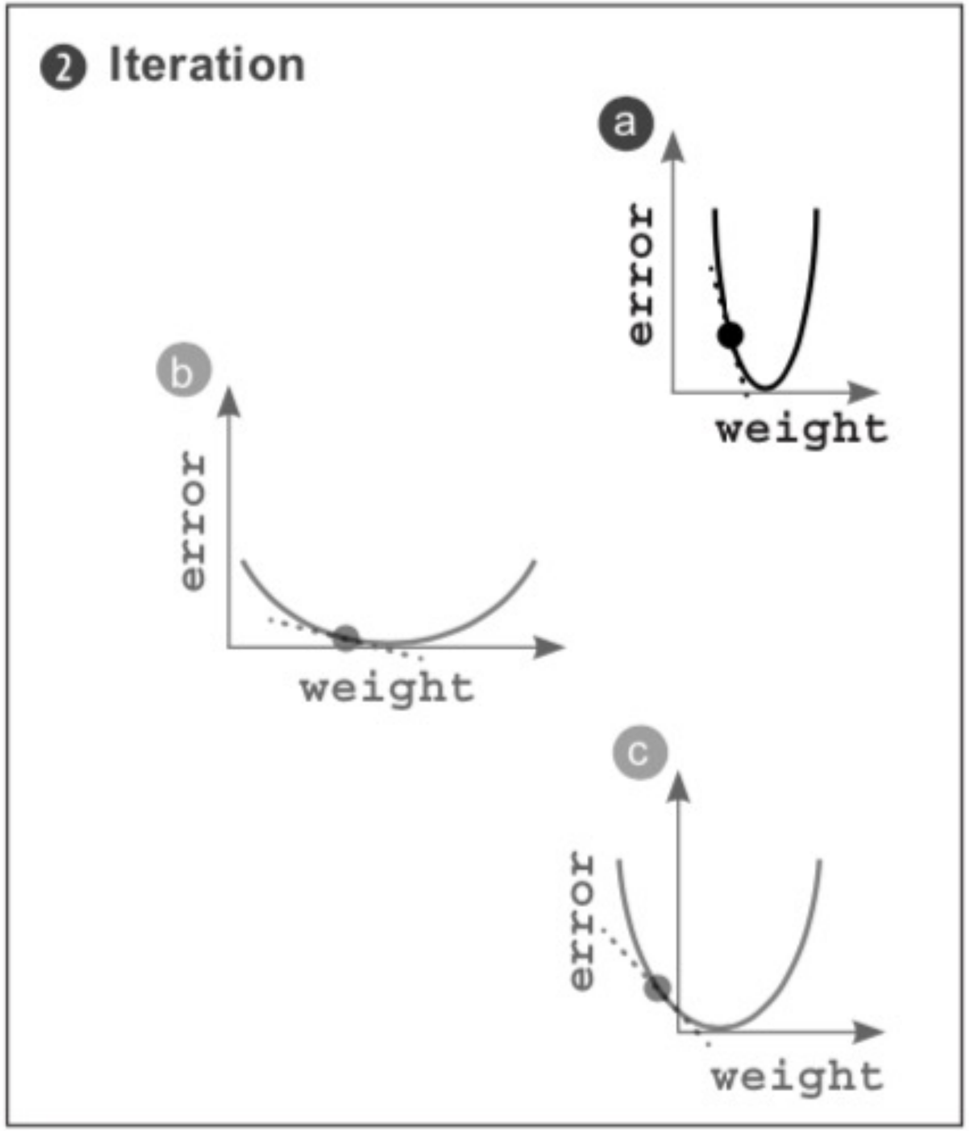
```
def neural_network(input, weights):
  out = 0
  for i in range(len(input)):
    out += (input[i] * weights[i])
  return out

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]
```

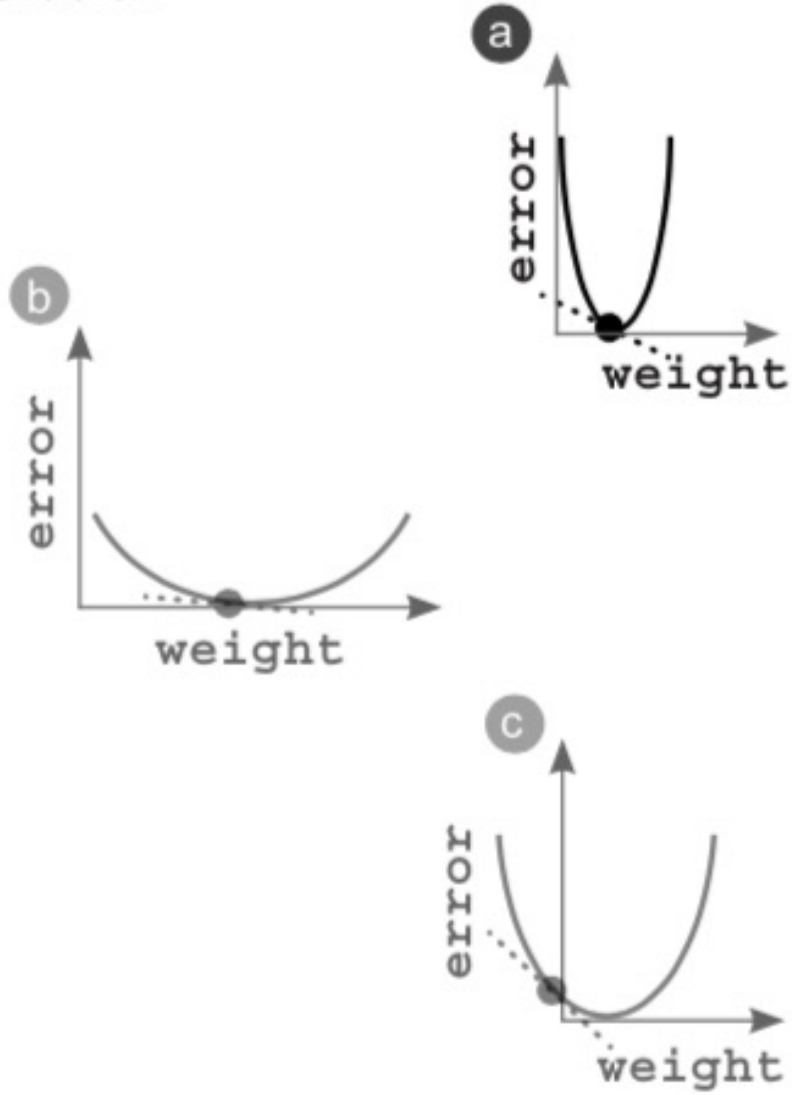


❶ Iteration
a
8.5
–1.2
.1
b
65%
–.09
.2
0.86
.020
–.1
–.14
–.17
c
1.2
weight_deltas
a
error
weight
b
error
weight
c
error
weight



2 Iteration
a
error
weight
b
error
weight
c
error
weight

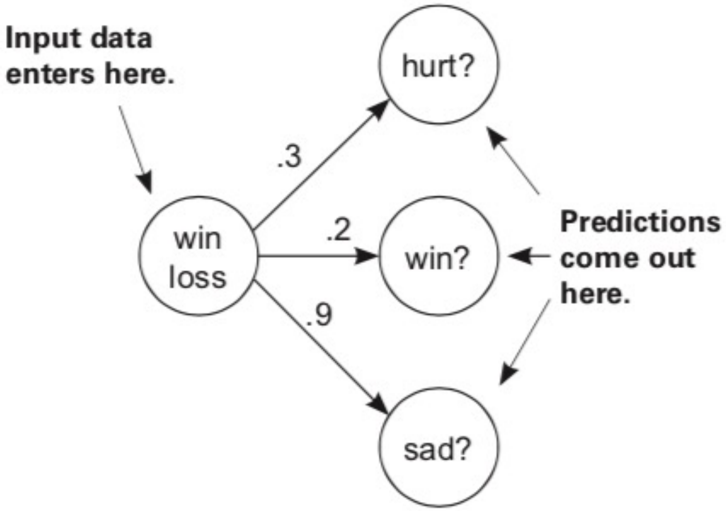
❸ Iteration



Gradient descent learning with multiple outputs

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

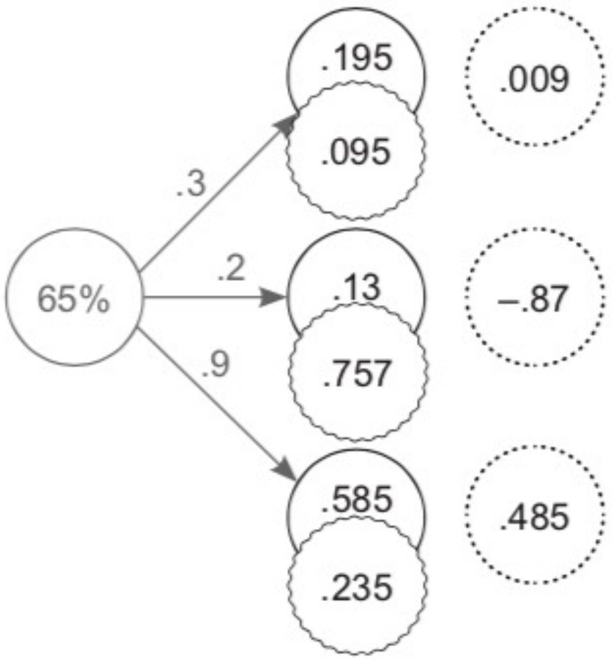
❶ An empty network with multiple outputs



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.

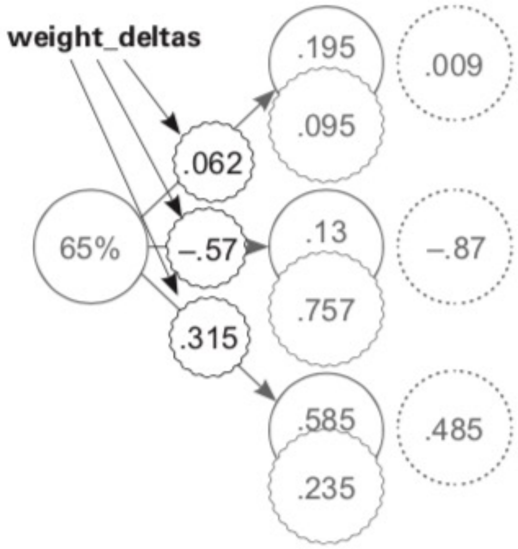
```
weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):

    pred = ele_mul(input,weights)

    return pred
```

❷ 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]
```

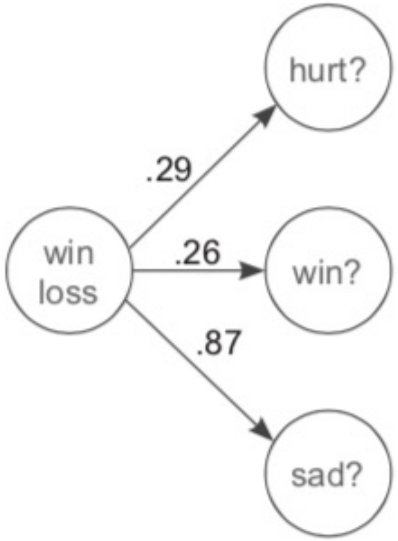
❸ COMPARE: Calculating each weight_delta and putting it on each weight



As before, `weight_delta`s are computed by multiplying the input node value with the output node `delta` for each weight. In this case, the `weight_delta`s share the same input node and have unique output nodes (`delta`s). 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)
```

❹ LEARN: Updating the weights

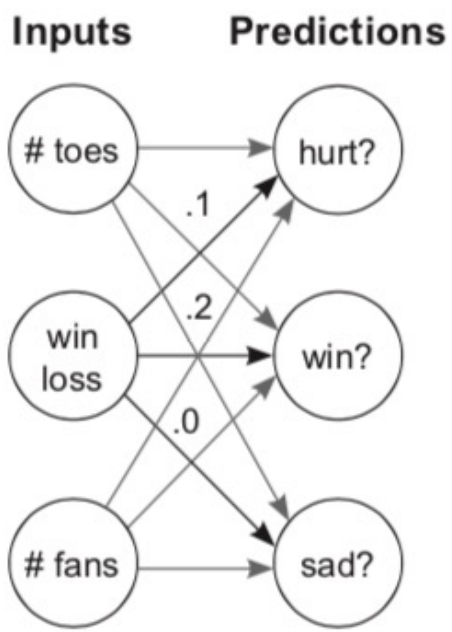



```
    input = wlrec[0]
    true = [hurt[0], win[0], sad[0]]
    pred = neural_network(input,weights)

    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

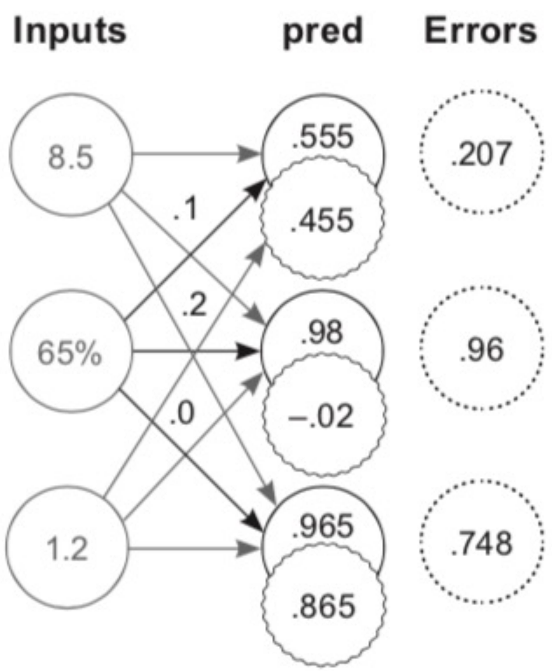
Gradient descent generalizes to arbitrarily large networks.

❶ **An empty network with multiple inputs and outputs**



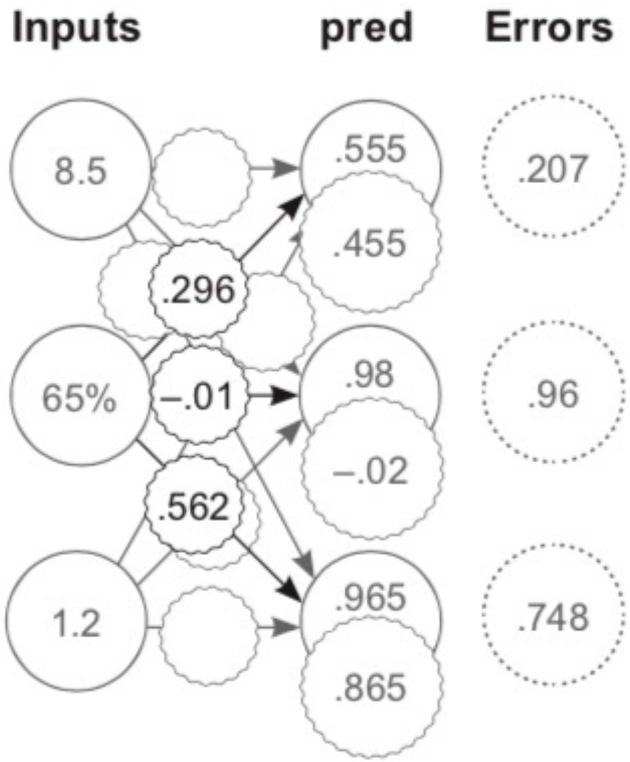

```
                    # toes %win # fans
weights = [ [0.1, 0.1, -0.3],# hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ]# sad?

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
```

❷ 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 = pred[i] - true[i]
```

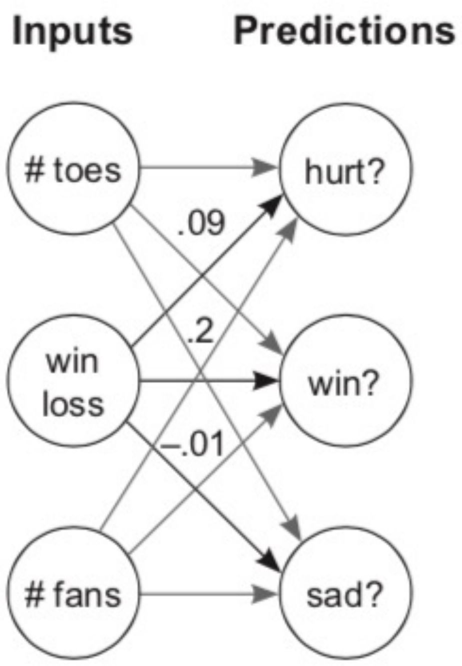
❸ COMPARE: Calculating each weight_delta and putting it on each weight



(`weight_delta`s are shown for only one input, to save space.)

```
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 = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)
```

❹ 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 = 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?

Each weight tries to reduce the error, but what do they learn in aggregate?

Congratulations! This is the part of the book where we move on to the first real-world dataset. As luck would have it, it's one with historical significance.

colah's blog

Visualizing MNIST: An Exploration of Dimensionality Reduction

Posted on October 9, 2014

MNIST, data visualization, machine learning, word embeddings, neural networks, deep learning

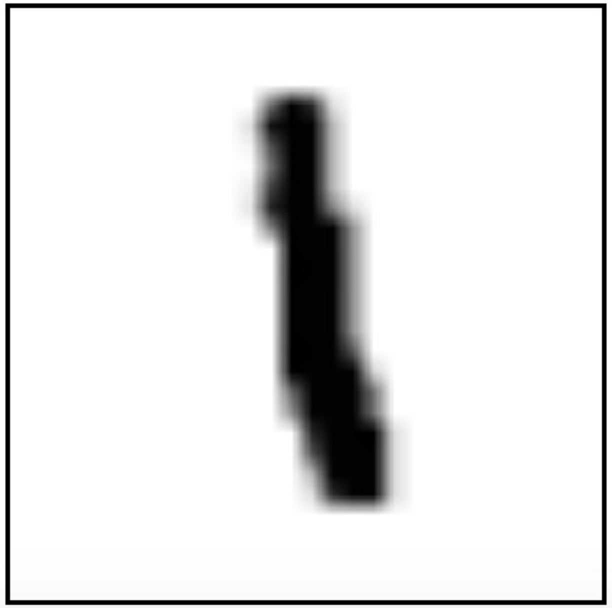
At some fundamental level, no one understands machine learning.

It isn't a matter of things being too complicated. Almost everything we do is fundamentally very simple. Unfortunately, an innate human handicap interferes with us understanding these simple things.

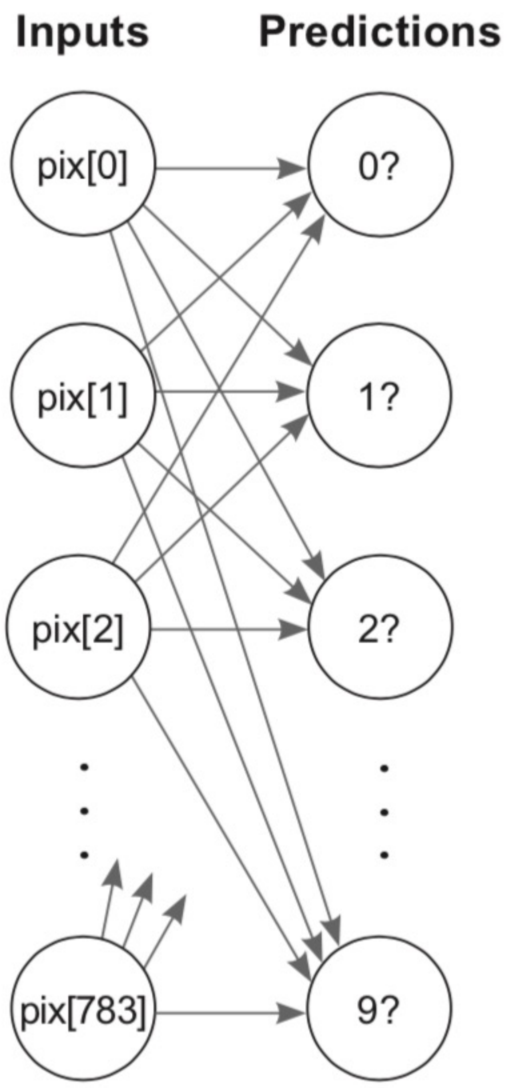
Humans evolved to reason fluidly about two and three dimensions. With some effort, we may think in four dimensions. Machine learning often demands we work with thousands of dimensions

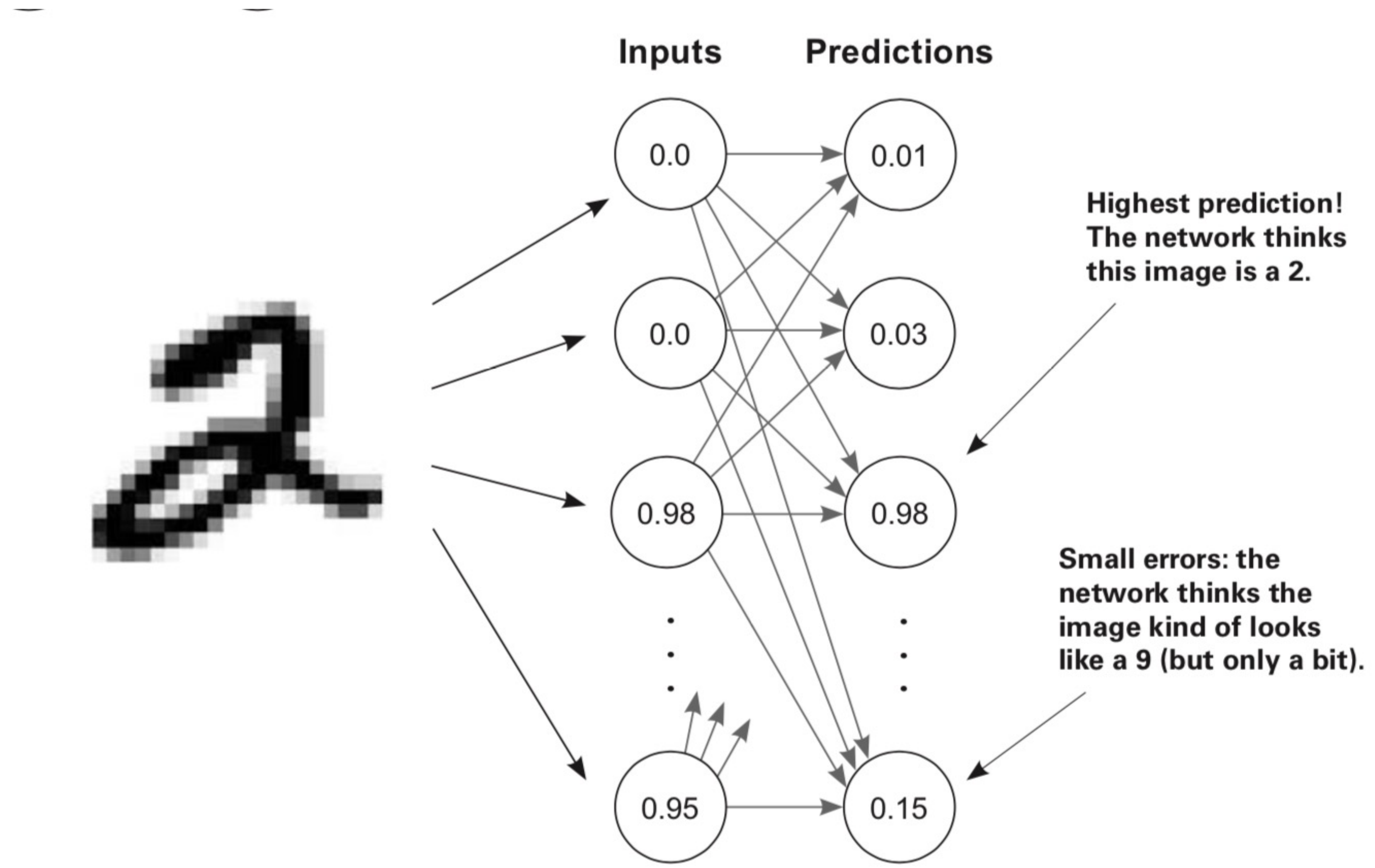
MNIST is a simple computer vision dataset. It consists of 28x28 pixel images of handwritten digits, such as:

Every MNIST data point, every image, can be thought of as an array of numbers describing how dark each pixel is. For example, we might think of as something like:

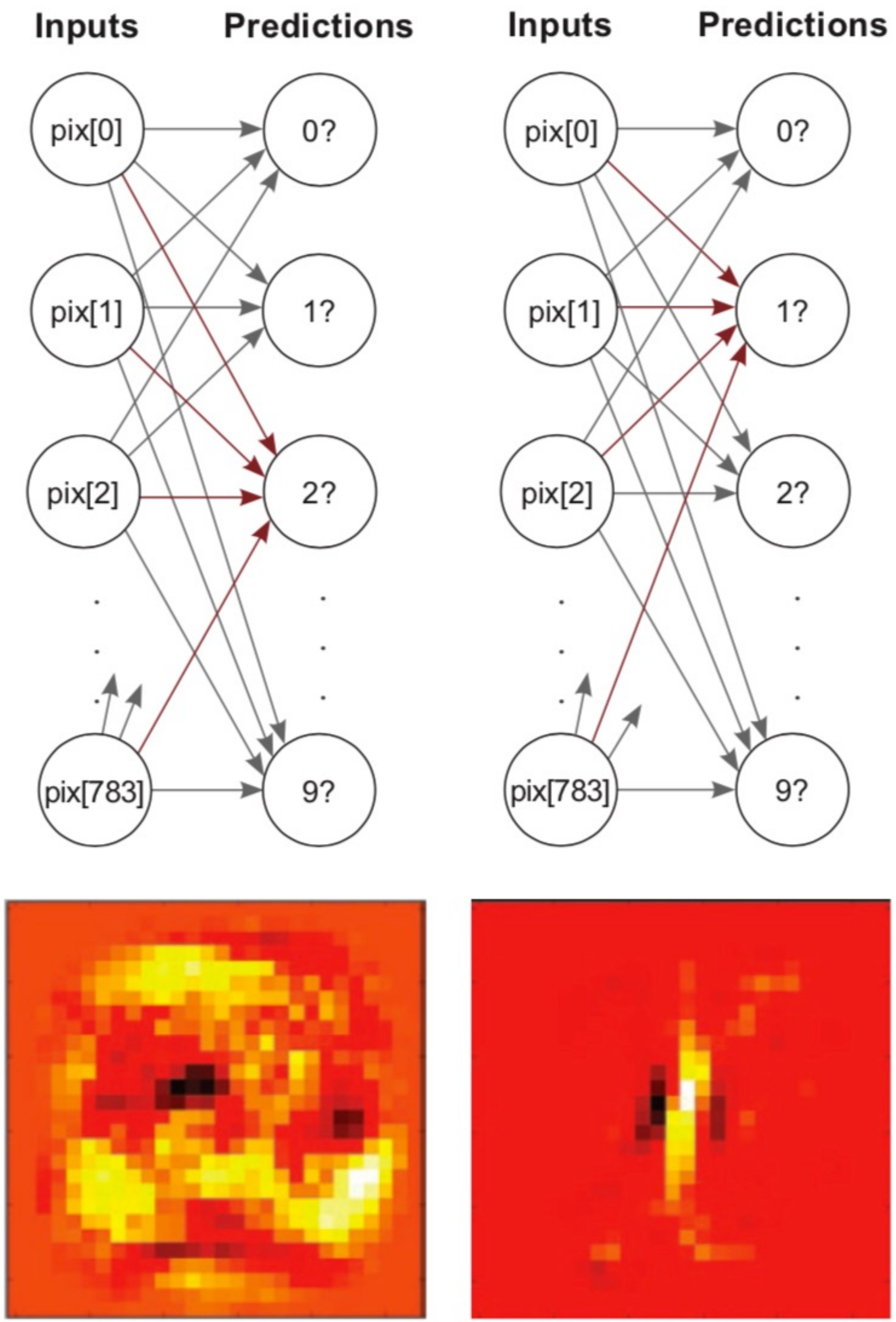


$$\approx \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .6 & .8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .5 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .9 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .3 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$


Inputs
Predictions
pix[0]
pix[1]
pix[2]
pix[783]
0?
1?
2?
9?


Inputs
Predictions
0.0
0.0
0.98
0.95
0.01
0.03
0.98
0.15
Highest prediction! The network thinks this image is a 2.
Small errors: the network thinks the image kind of looks like a 9 (but only a bit).

Visualizing weight values



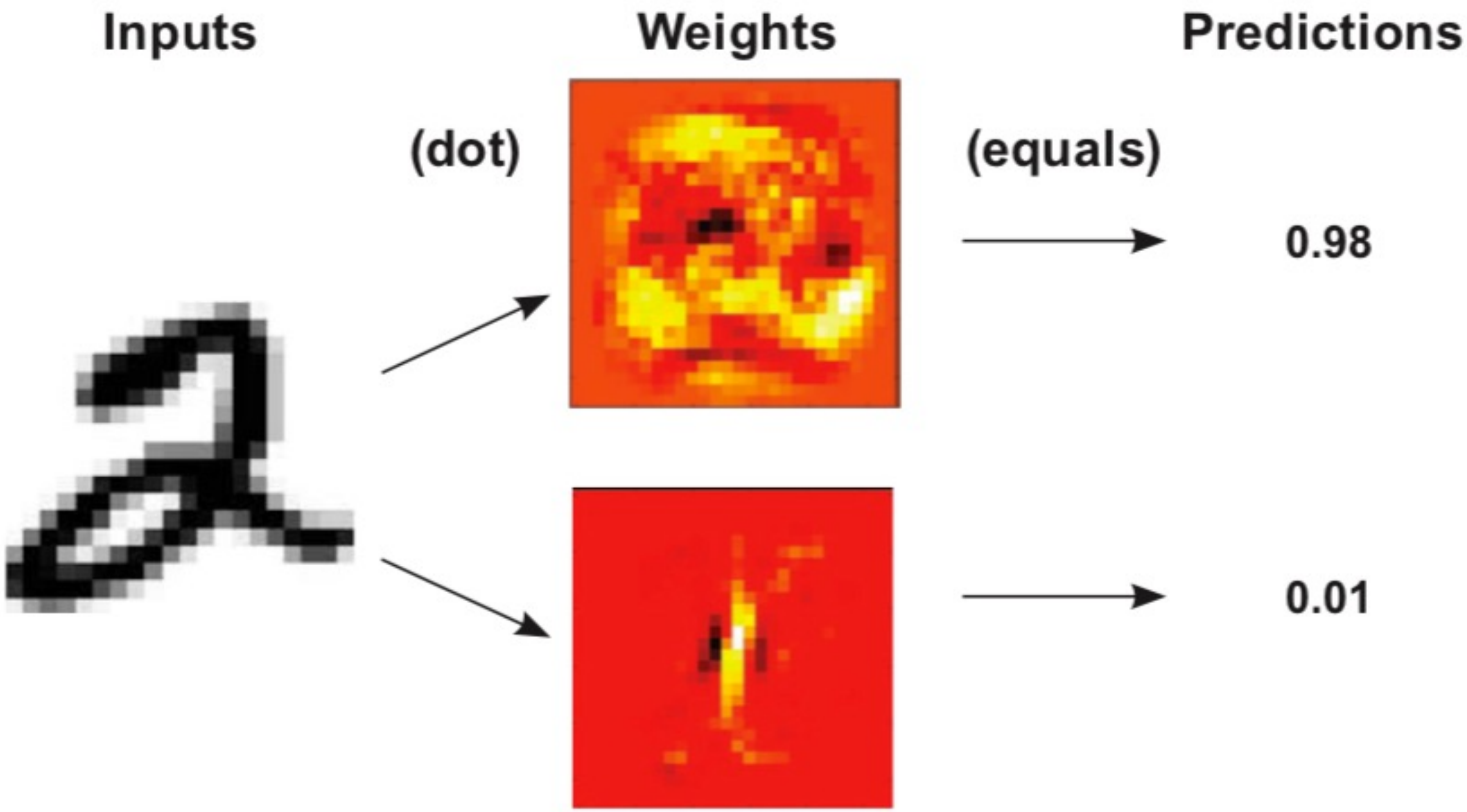
Visualizing dot products (weighted sums)

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

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


Inputs
Weights
Predictions
(dot)
(equals)
0.98
0.01