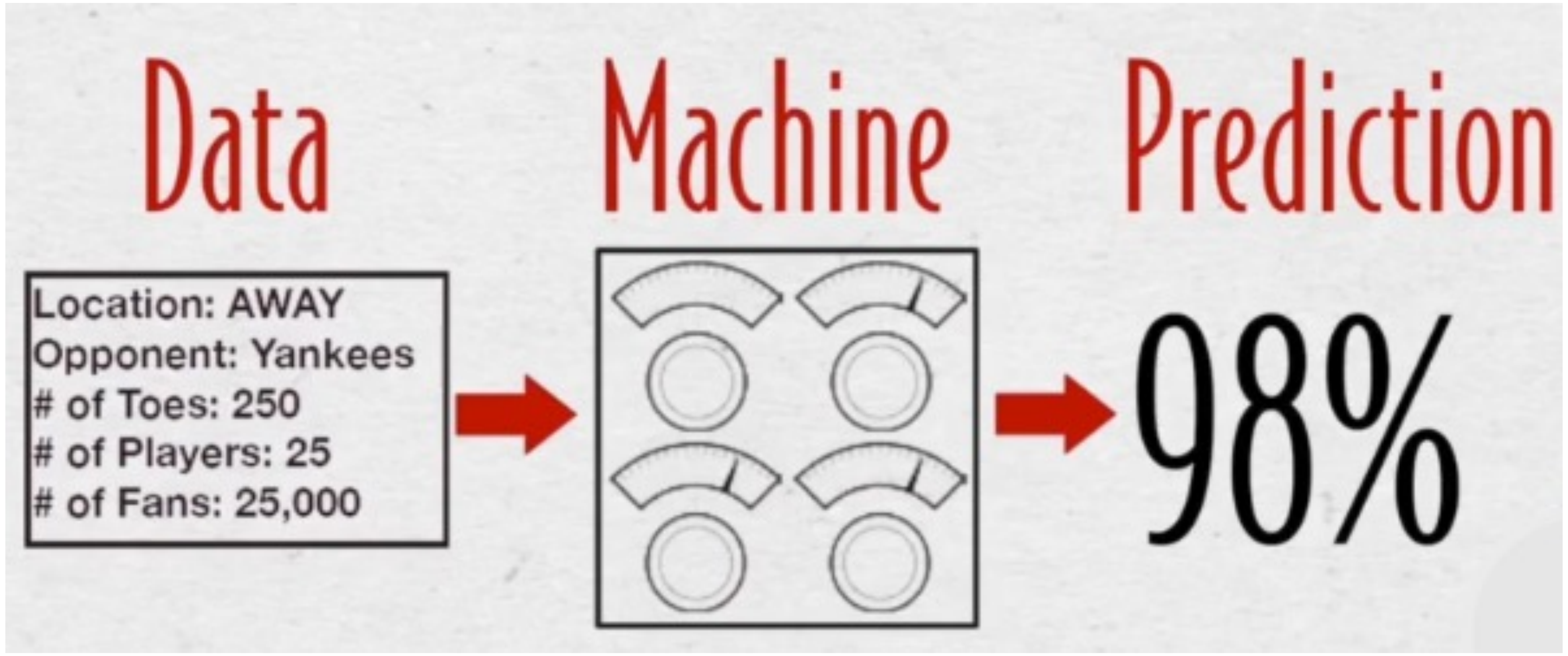
INTRODUCTION TO NEURAL PREDICTION: FORWARD PROPAGATION

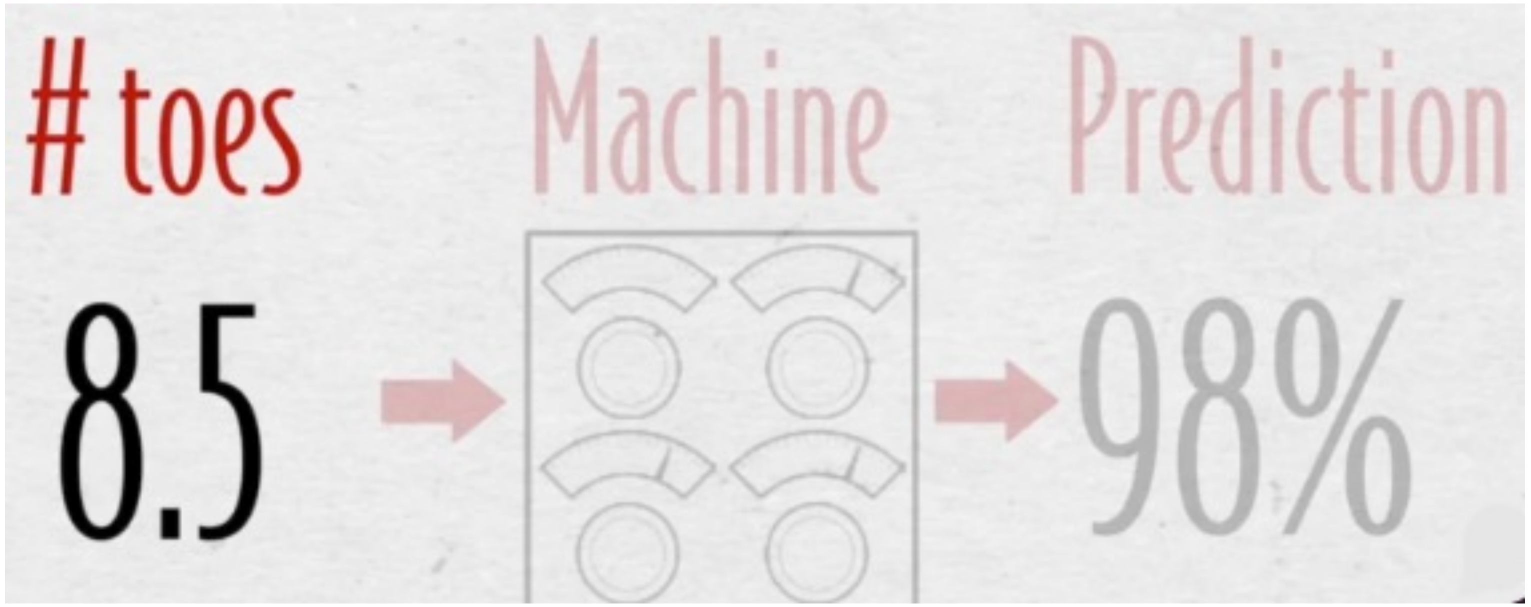
CHAPTER 3

First Neural Network



Predict -> Compare -> Learn

First Neural Network - Data



First Neural Network - Data

The number of datapoints you process at a time has a significant impact on what a network looks like.

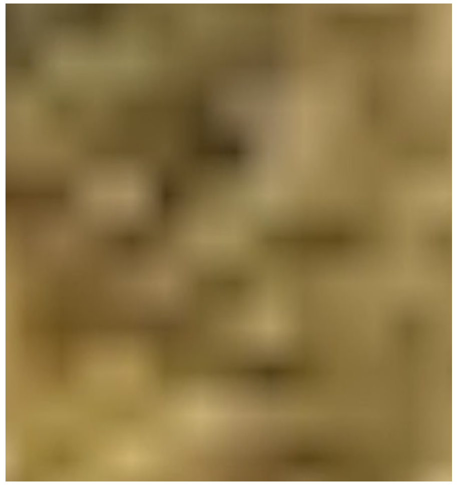
You might be wondering, “How do I choose how many datapoints to propagate at a time?”

The answer is based on whether you think the neural network can be accurate with the data you give it.

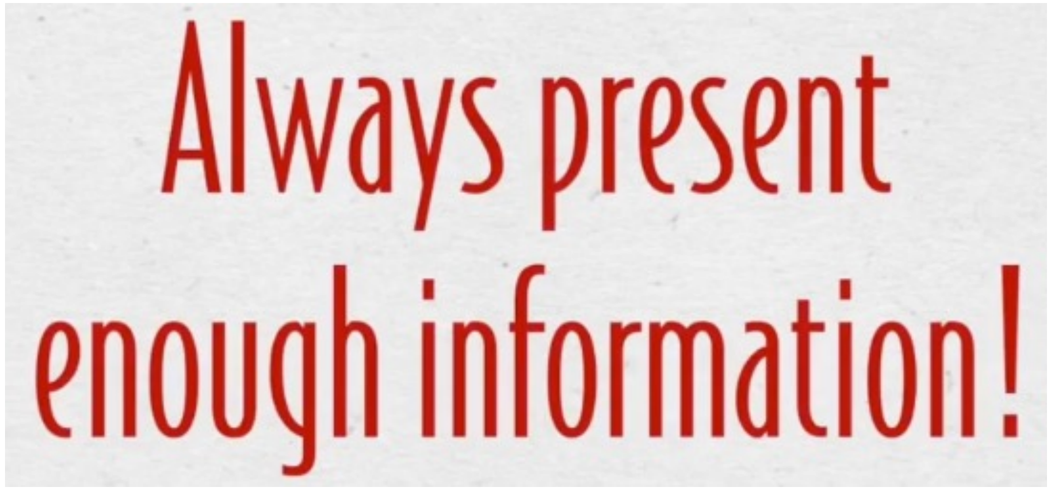
How Many Datapoints at a Time?

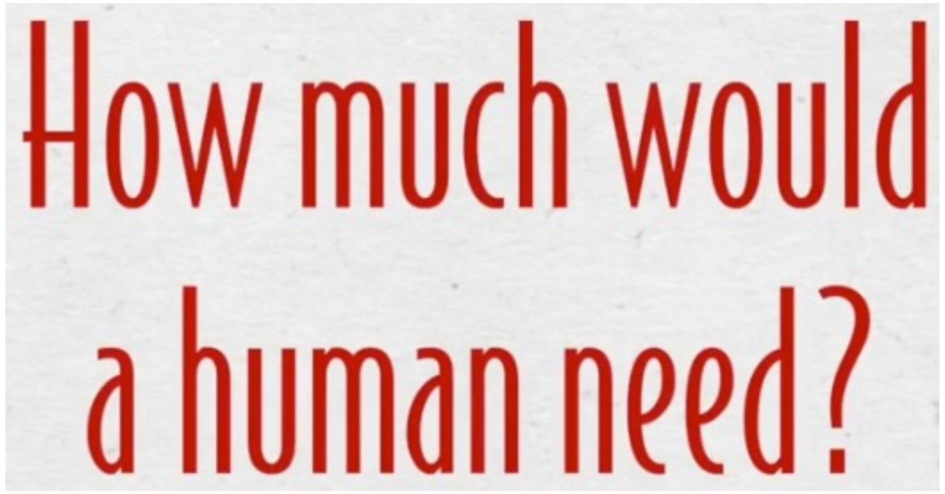
For example, if I'm trying to predict whether there's a cat in a photo, I definitely need to show my network all the pixels of an image at once.

Why? Well, if I sent you only one pixel of an image, could you classify whether the image contained a cat? Me neither! (That's a general rule of thumb, by the way: always present enough information to the network, where "enough information" is defined loosely as how much a human might need to make the same prediction.)

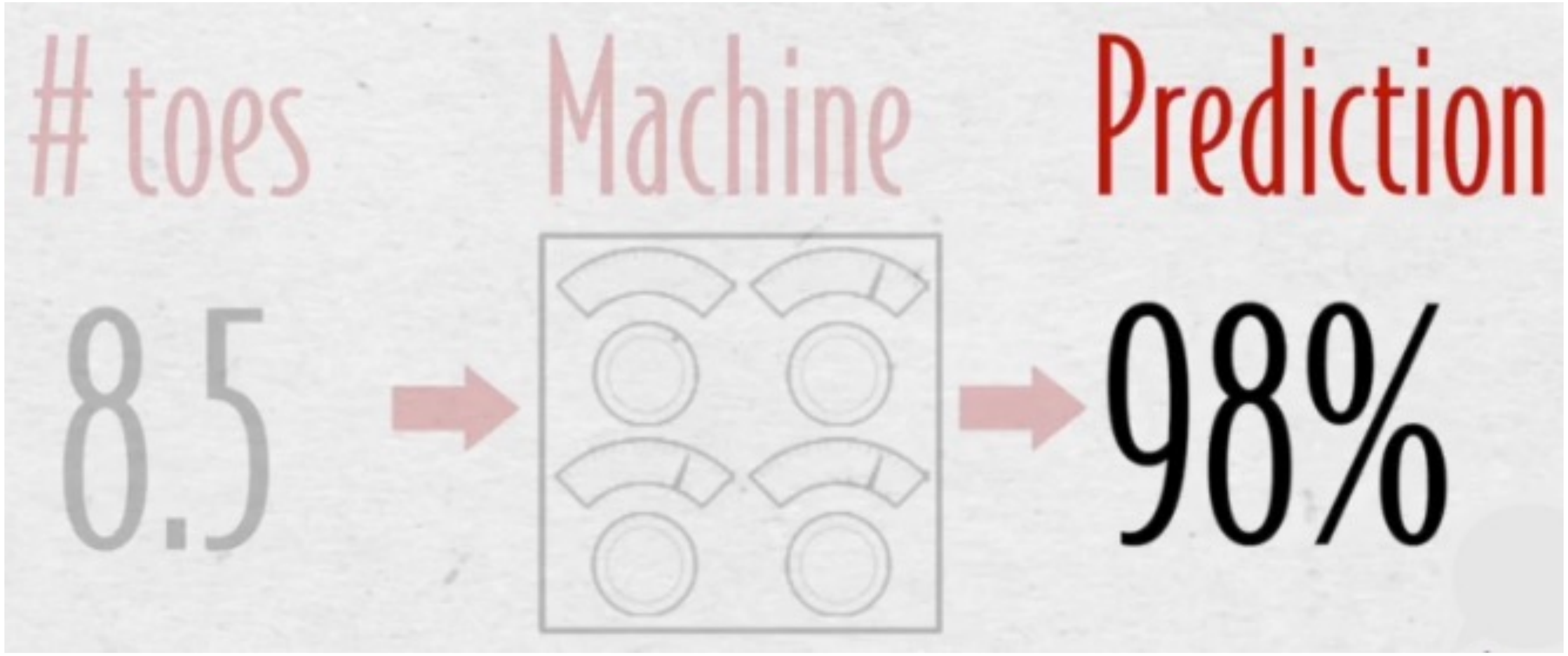


Enough Information?


Always present enough information!


How much would a human need?

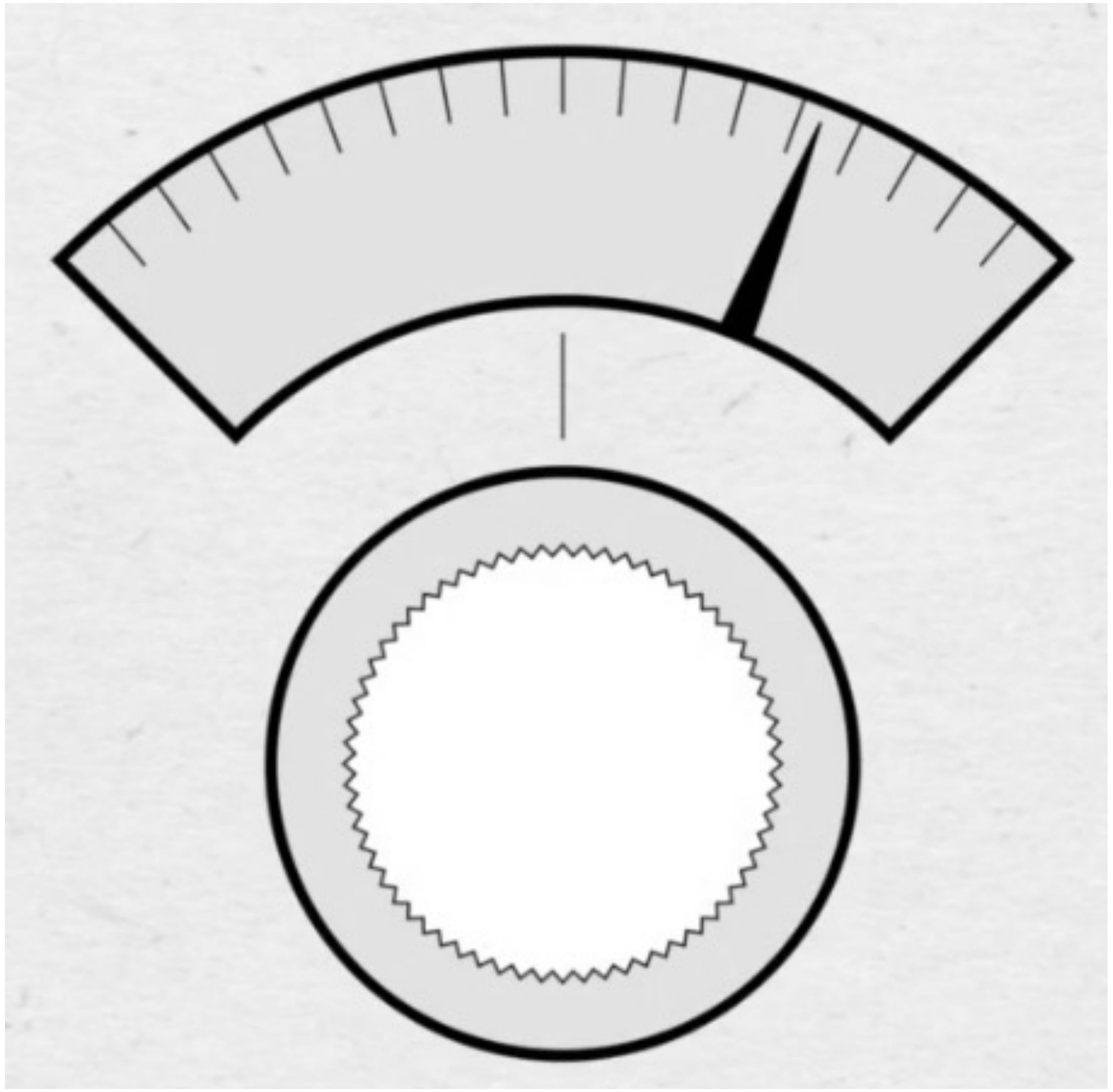
Prediciton



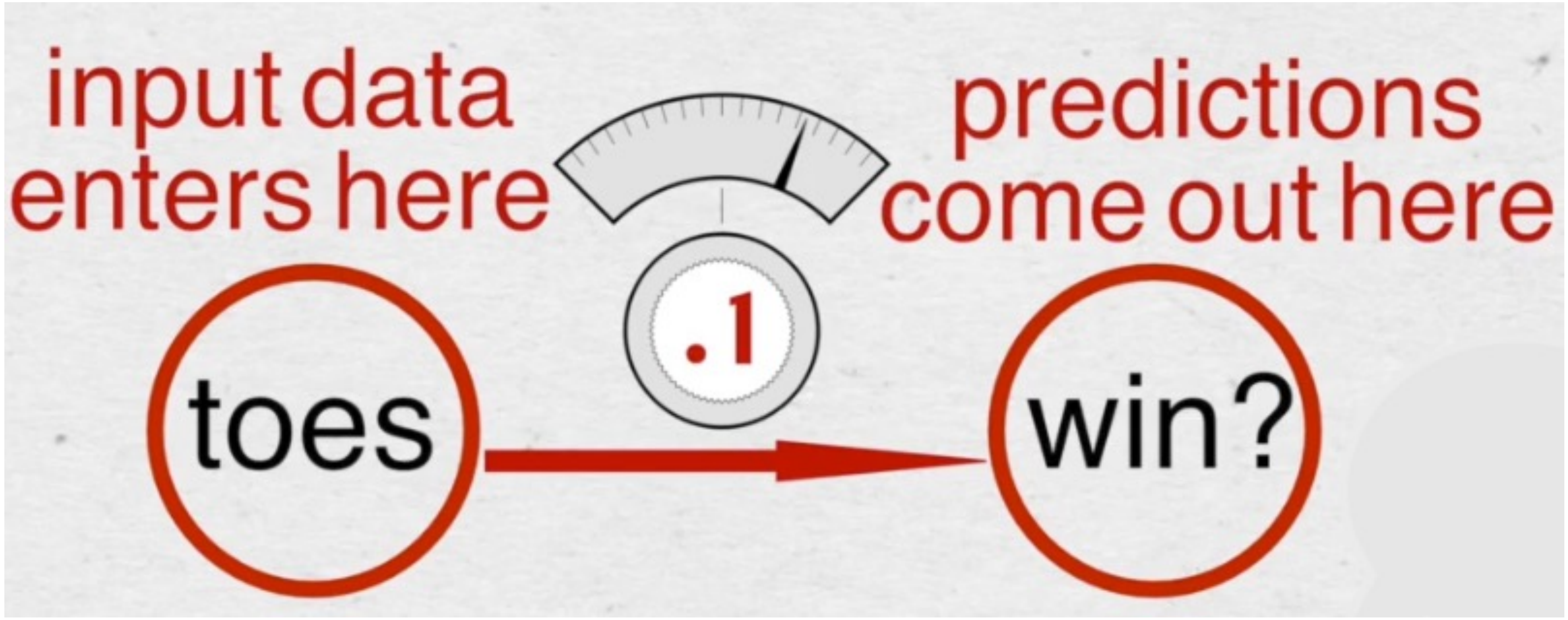
Machine

Now that you know you want to take one input datapoint and output one prediction, you can create a neural network.

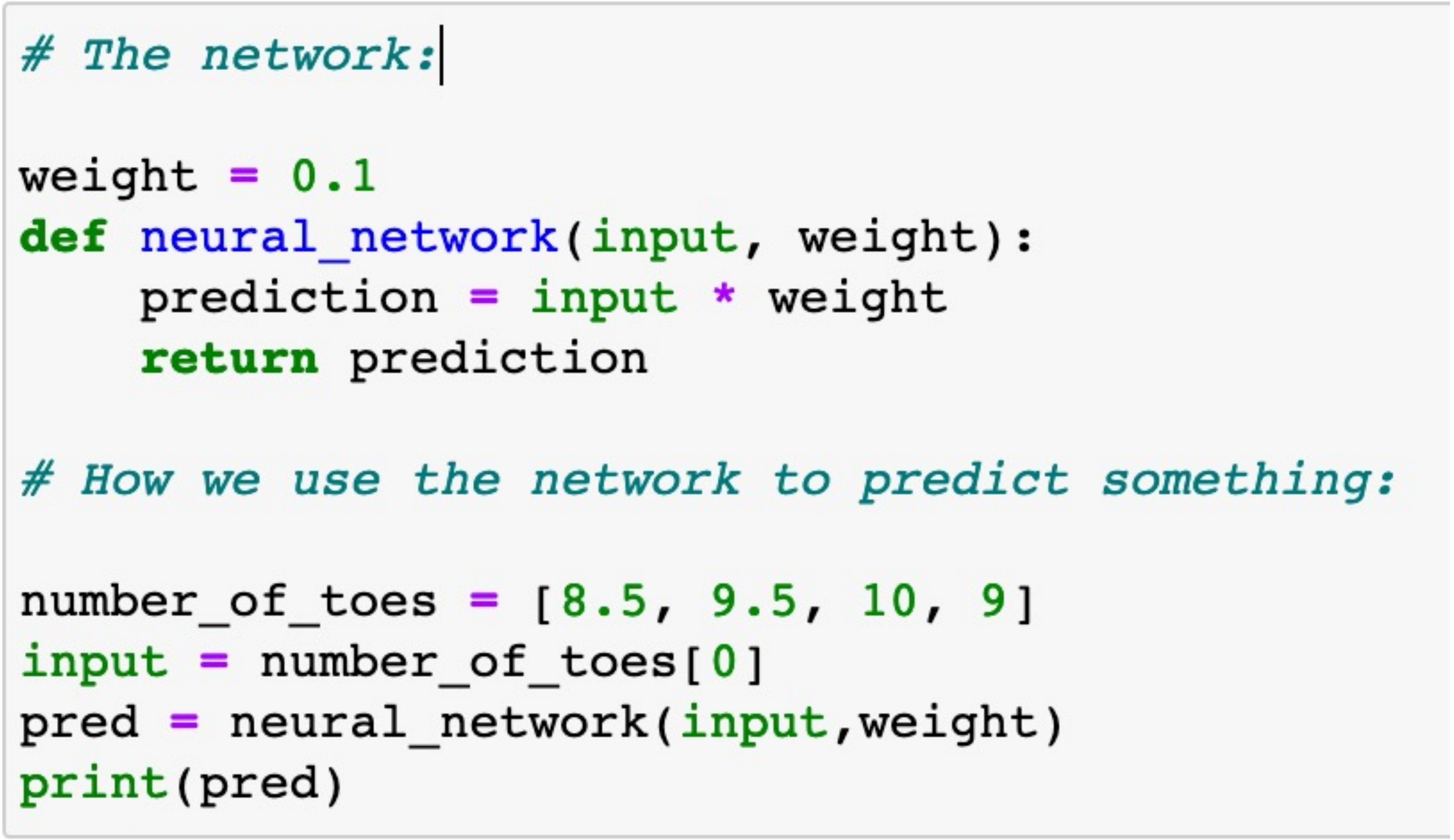
Because you have only one input datapoint and one output datapoint, you’re going to build a **network with a single knob** mapping from the input point to the output



Weights

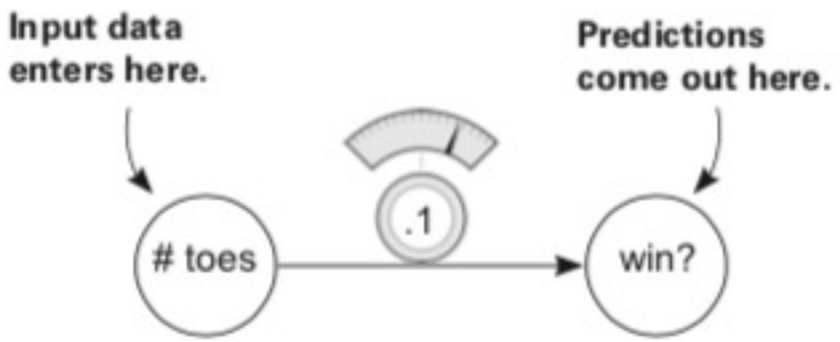


A simple neural network making a prediction



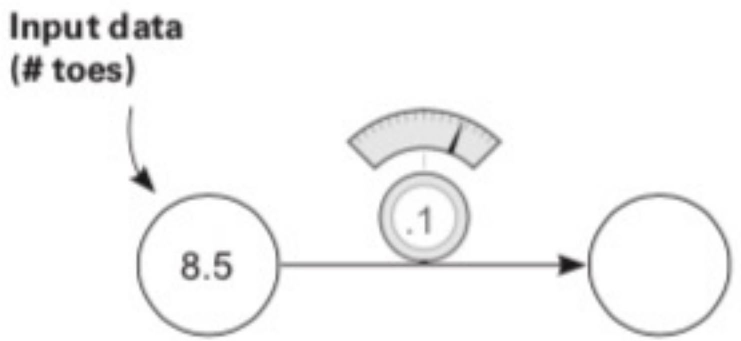
```python
# The network:

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

# How we use the network to predict something:

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

❶ **An empty network**



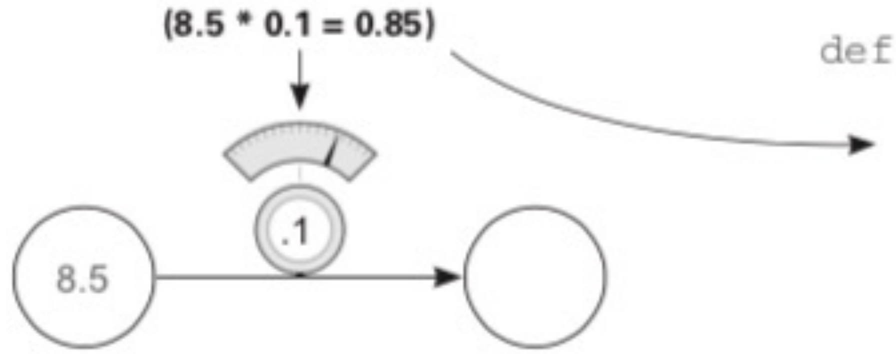
```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

❷ **Inserting one input datapoint**



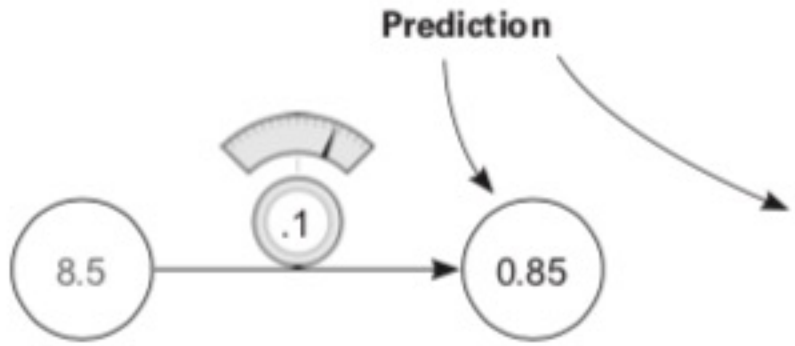
```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)

print(pred)
```

❸ Multiplying input by weight

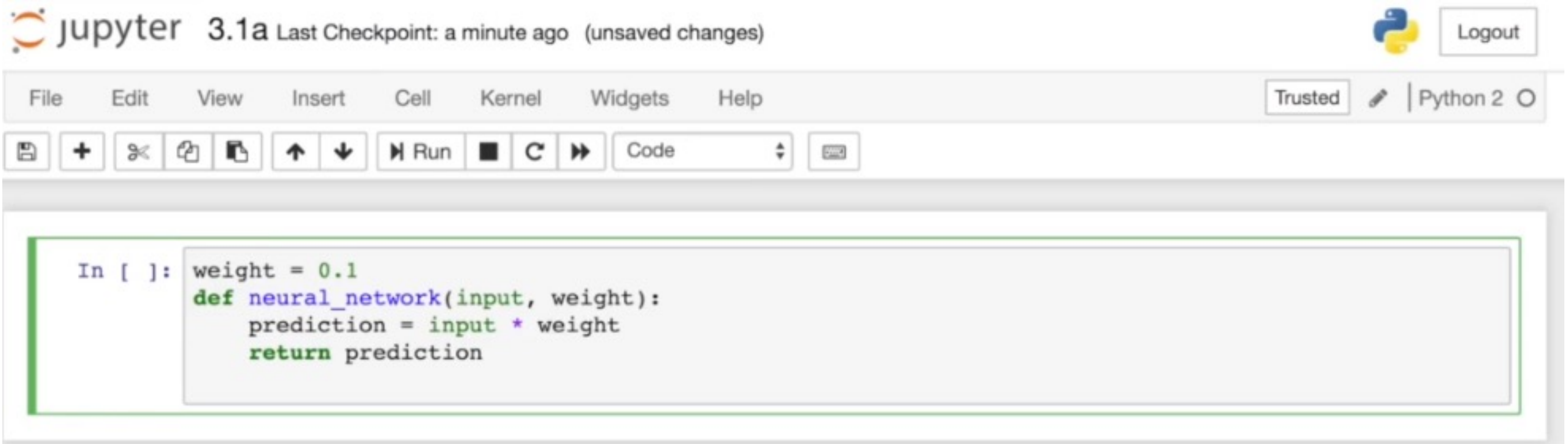


```
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

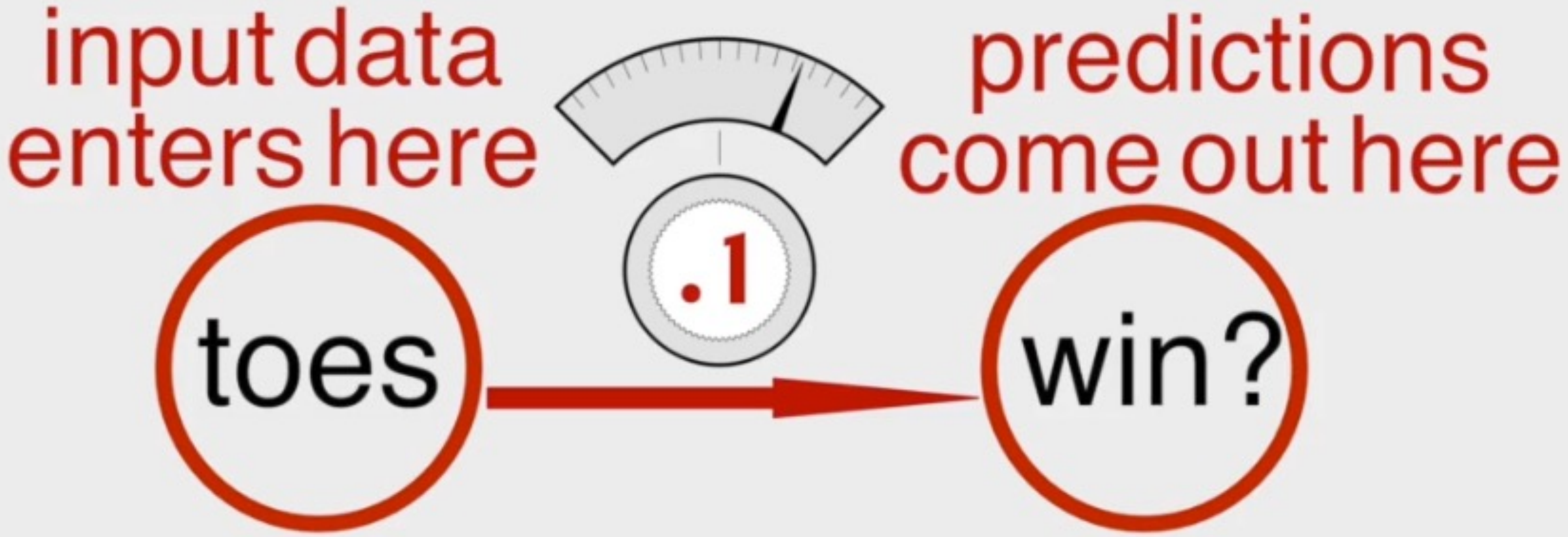
❹ Depositing the prediction

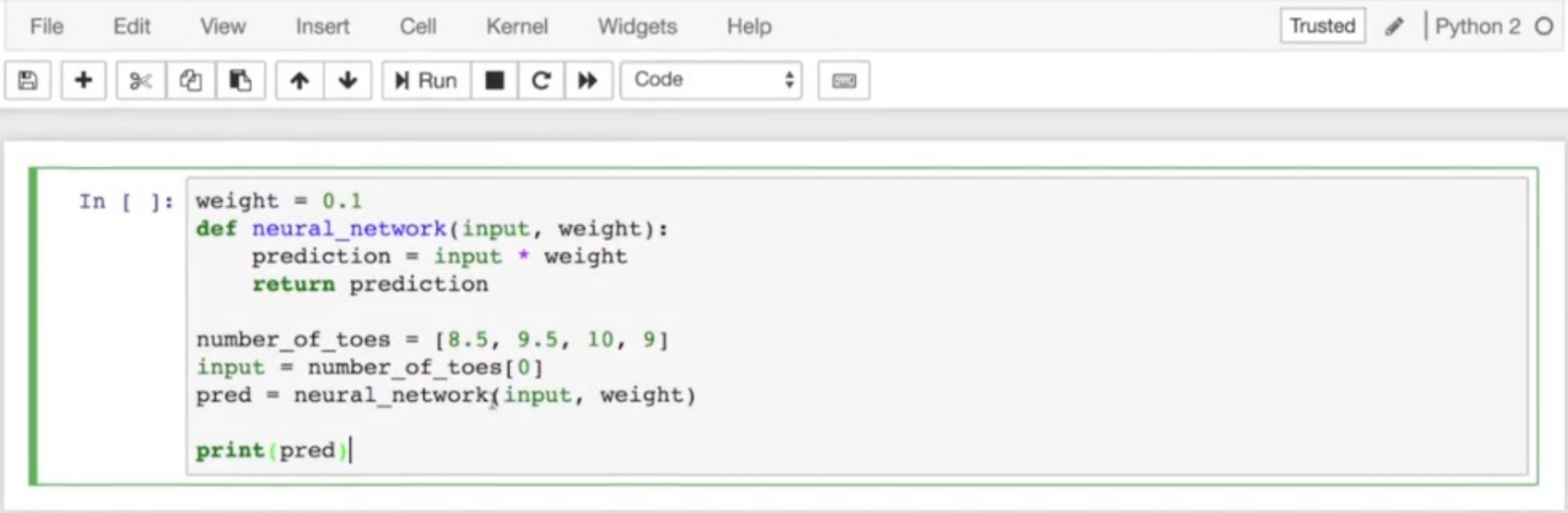


```
number_of_toes = [8.5, 9.5, 10, 9]
input = number_of_toes[0]
pred = neural_network(input,weight)
```

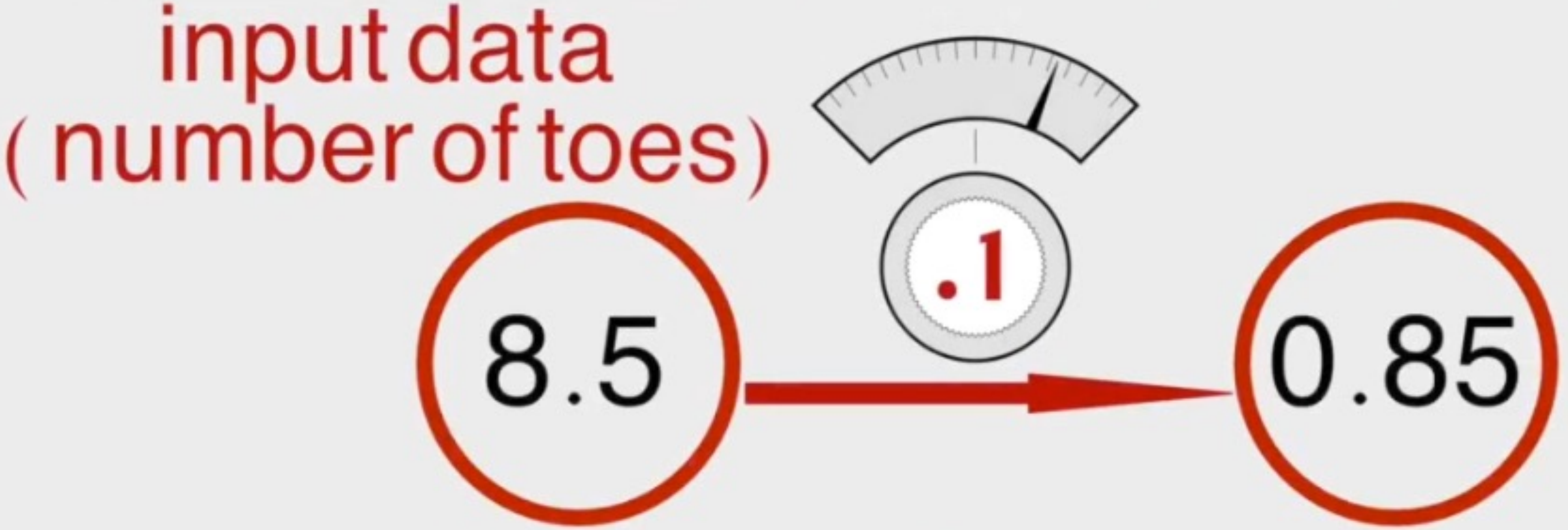

jupyter 3.1a Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code
In [ ]:

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





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

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

print(pred)
```



What is input data?

It's a number that you recorded in the real world somewhere. It's usually something that is easily knowable, like today's temperature, a baseball player's batting average, or yesterday's stock price.

What is a prediction?

A *prediction* is what the neural network tells you, *given the input data*, such as "given the temperature, it is **0%** likely that people will wear sweatsuits today" or "given a baseball player's batting average, he is **30%** likely to hit a home run" or "given yesterday's stock price, today's stock price will be **101.52**."

Is this prediction always right?

No. Sometimes a neural network will make mistakes, but it can learn from them. For example, if it predicts too high, it will adjust its weight to predict lower next time, and vice versa.

How does the network learn?

Trial and error! First, it tries to make a prediction. Then, it sees whether the prediction was too high or too low. Finally, it changes the weight (up or down) to predict more accurately the next time it sees the same input.

✅ Predict

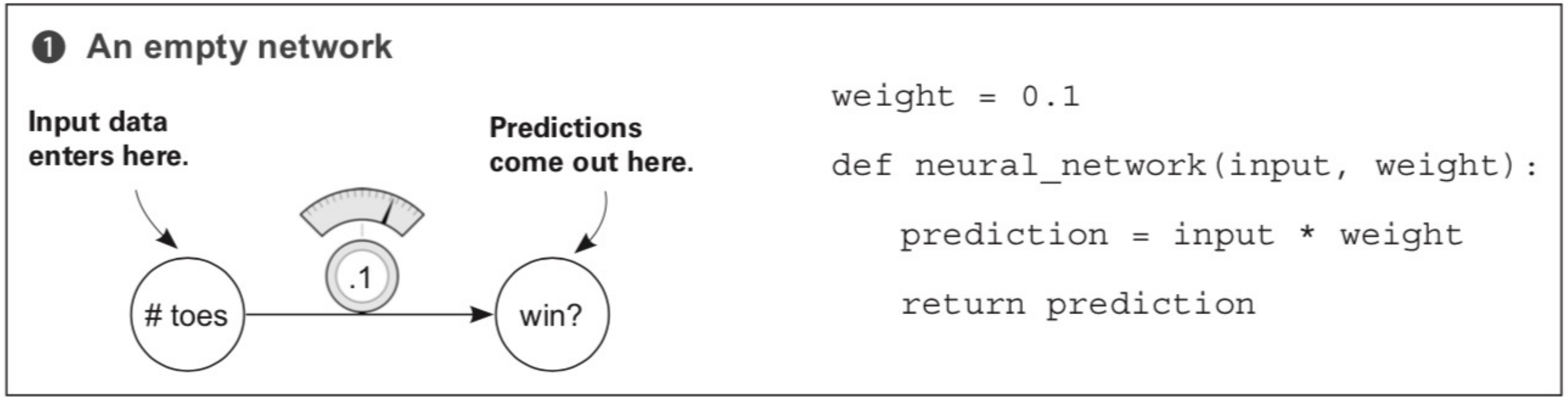
❌ Compare

❌ Learn

Sensitivity

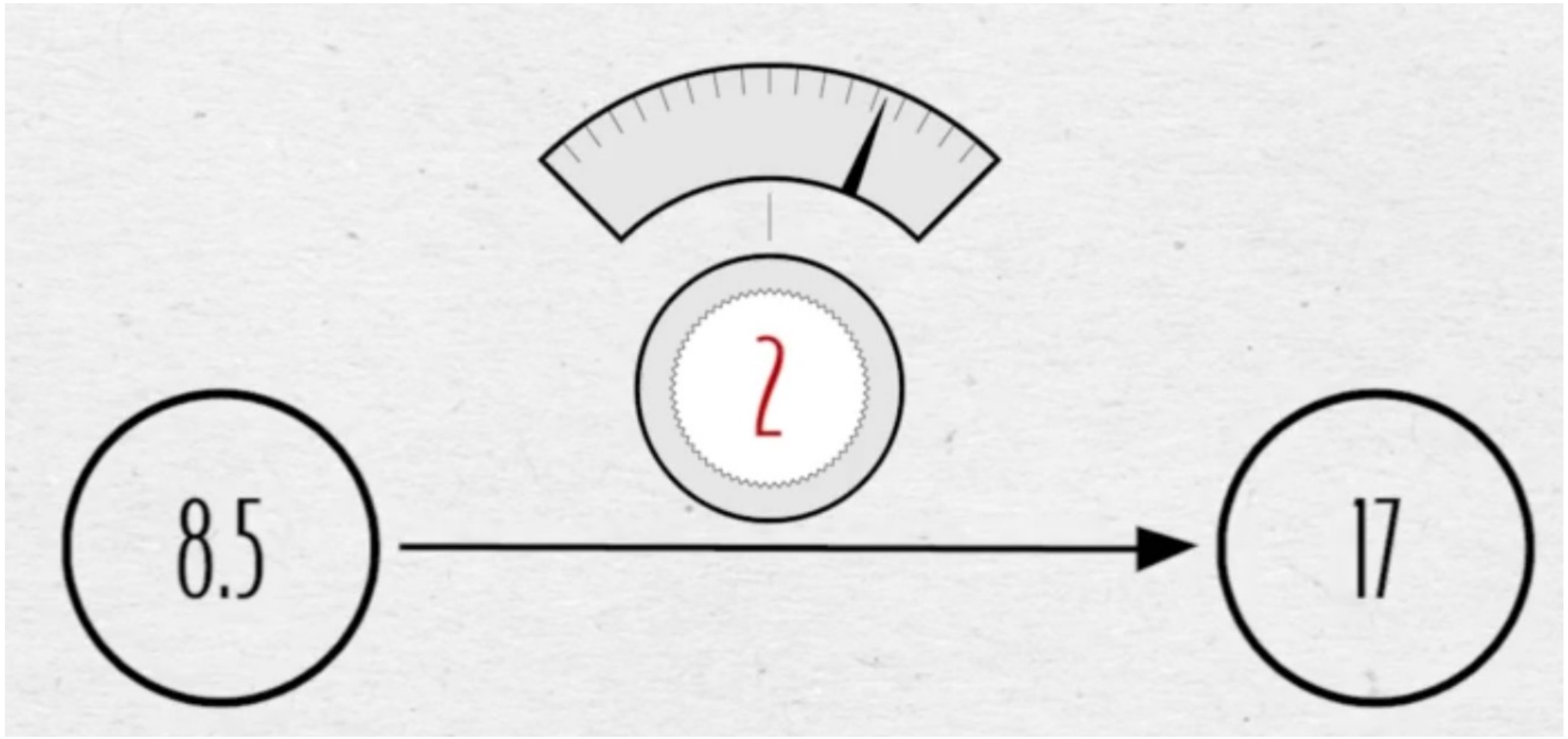
What does this neural network do?

It multiplies the input by a weight. It “scales” the input by a certain amount. It accepts an input variable as information and a weight variable as knowledge and outputs a prediction.

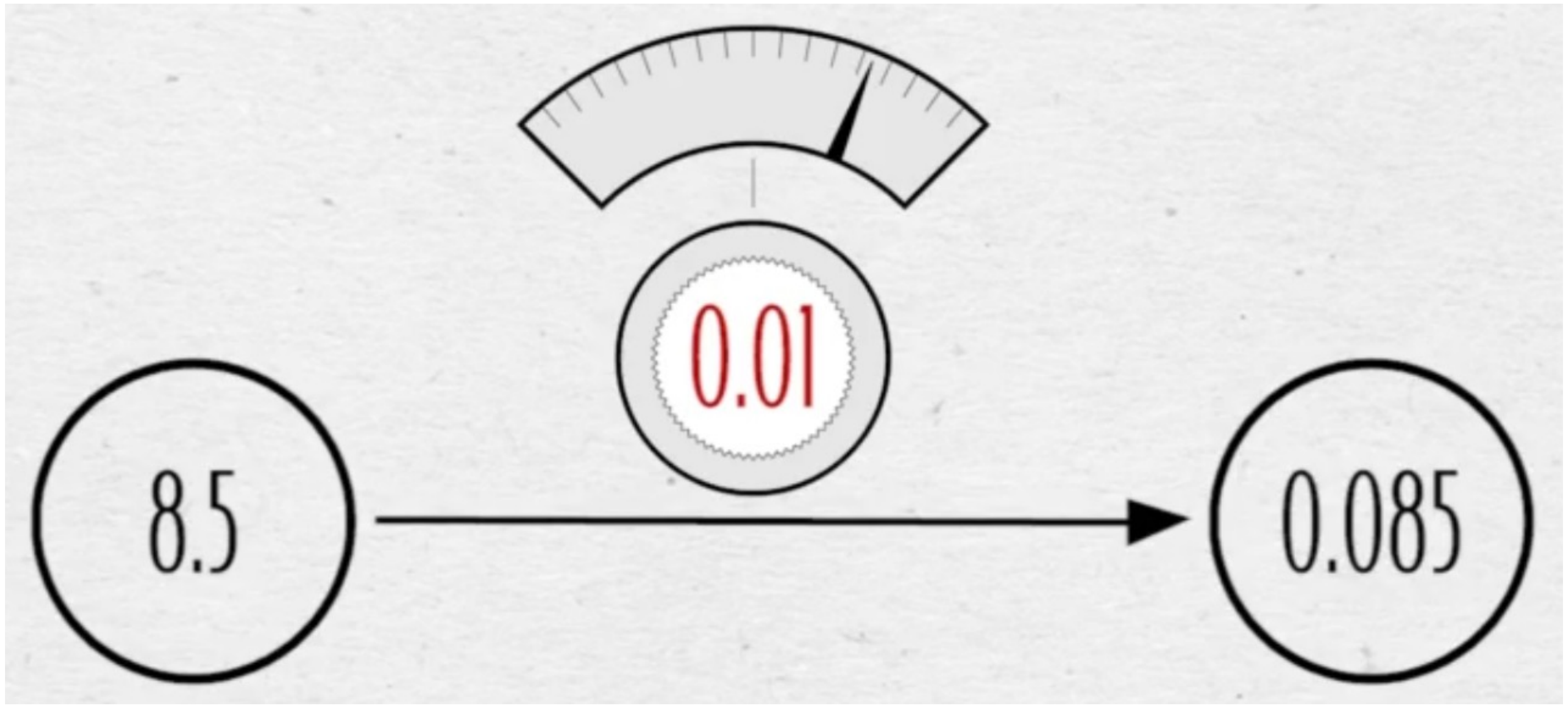


```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

Sensitivity



Sensitivity



Sensitivity

It uses the *knowledge* in the weights to interpret the *information* in the input data.

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

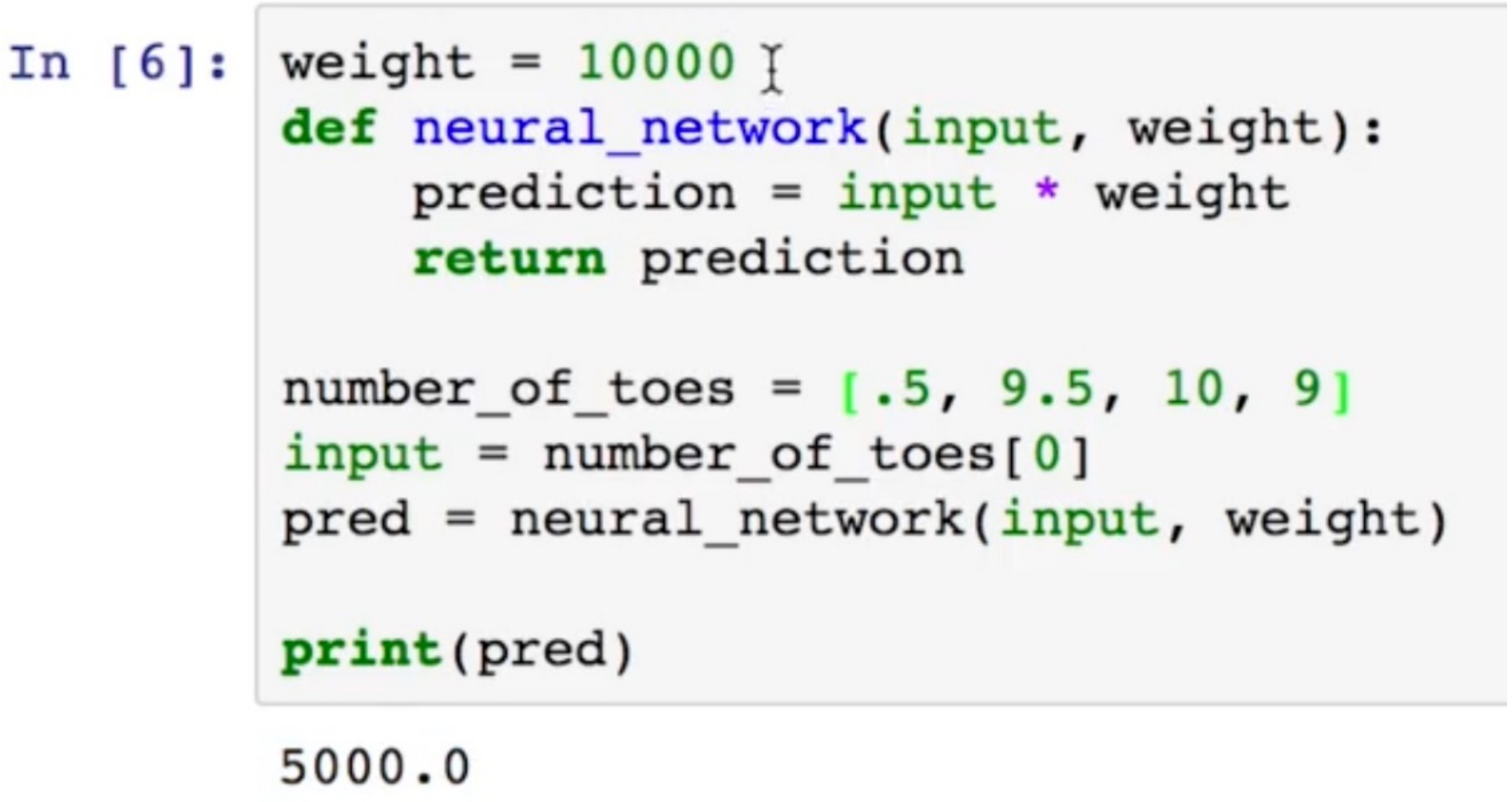
Sensitivity

Another way to think about a neural network’s weight value is as a measure of *sensitivity* between the input of the network and its prediction.

If the weight is very high, then even the tiniest input can create a really large prediction!

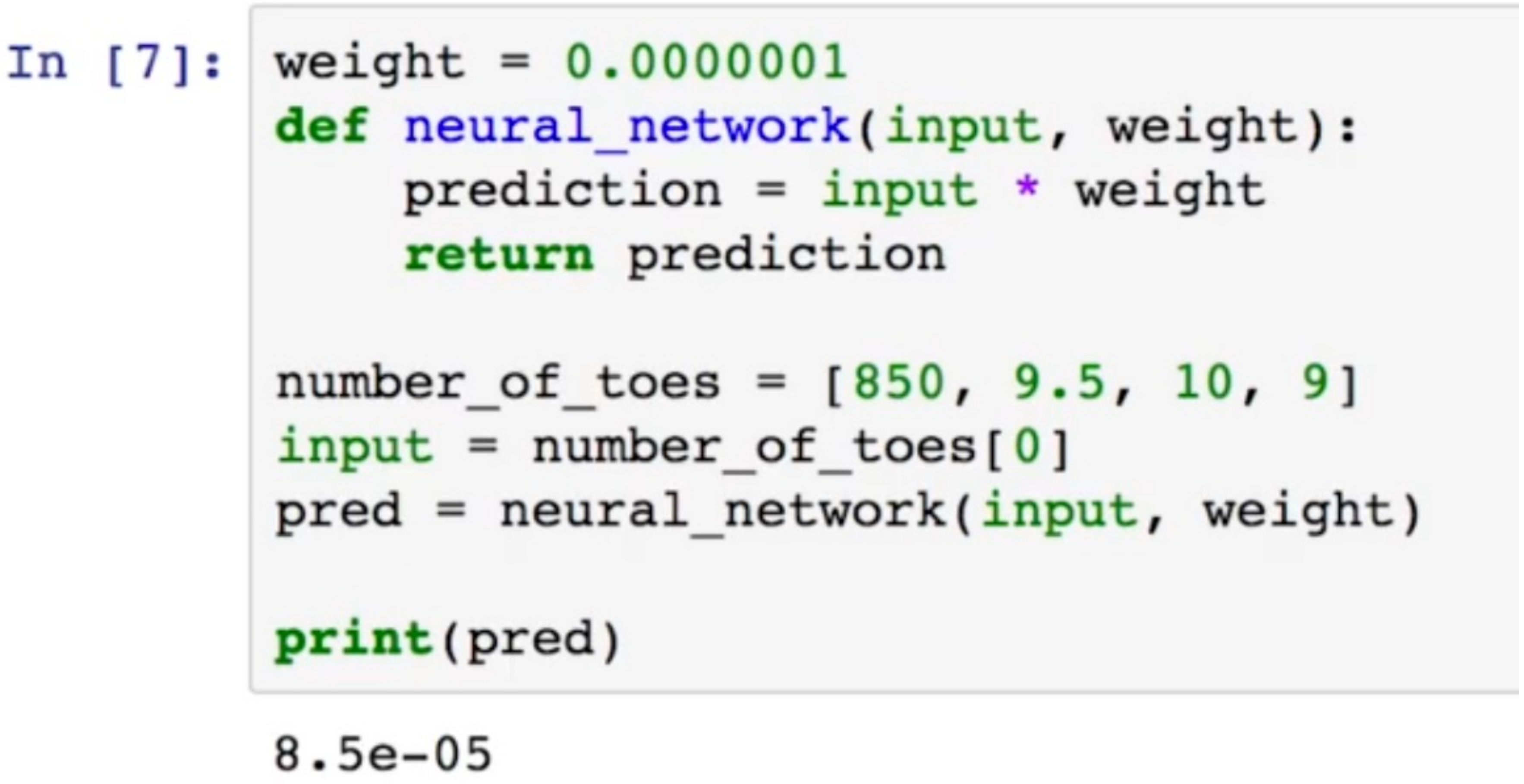
If the weight is very small, then even large inputs will make small predictions.

Sensitivity



```
In [6]: weight = 10000
        def neural_network(input, weight):
            prediction = input * weight
            return prediction

        number_of_toes = [.5, 9.5, 10, 9]
        input = number_of_toes[0]
        pred = neural_network(input, weight)

        print(pred)

        5000.0
```

Sensitivity



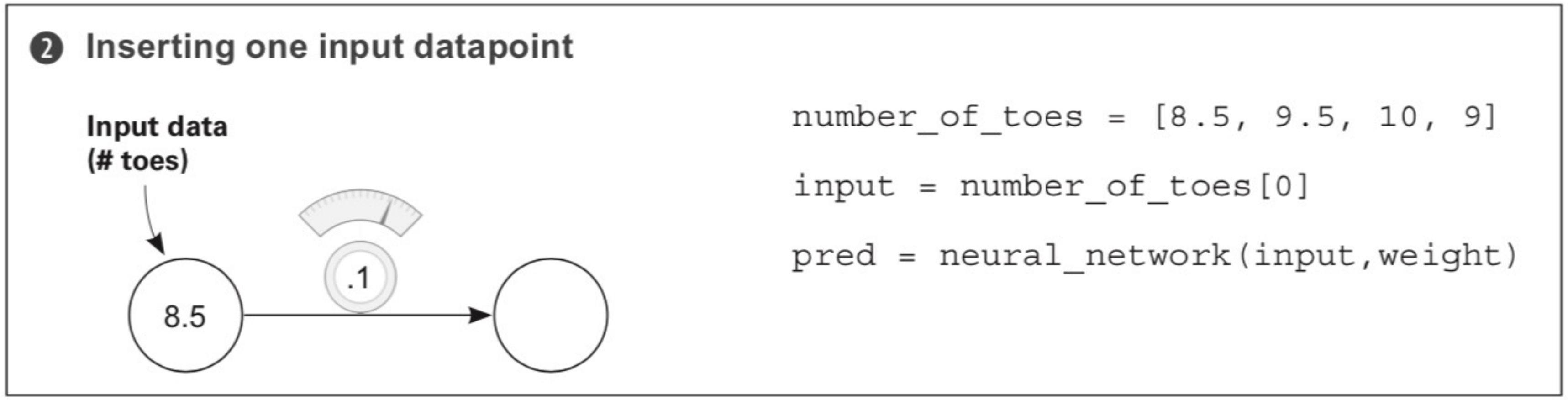
```
In [7]: weight = 0.0000001
        def neural_network(input, weight):
            prediction = input * weight
            return prediction

        number_of_toes = [850, 9.5, 10, 9]
        input = number_of_toes[0]
        pred = neural_network(input, weight)

        print(pred)
```

```
8.5e-05
```

Short-term Memory

A neural network knows only what you feed it as input. It forgets everything else.

Later, you'll learn how to give a neural network a "short-term memory" by feeding in multiple inputs at once.

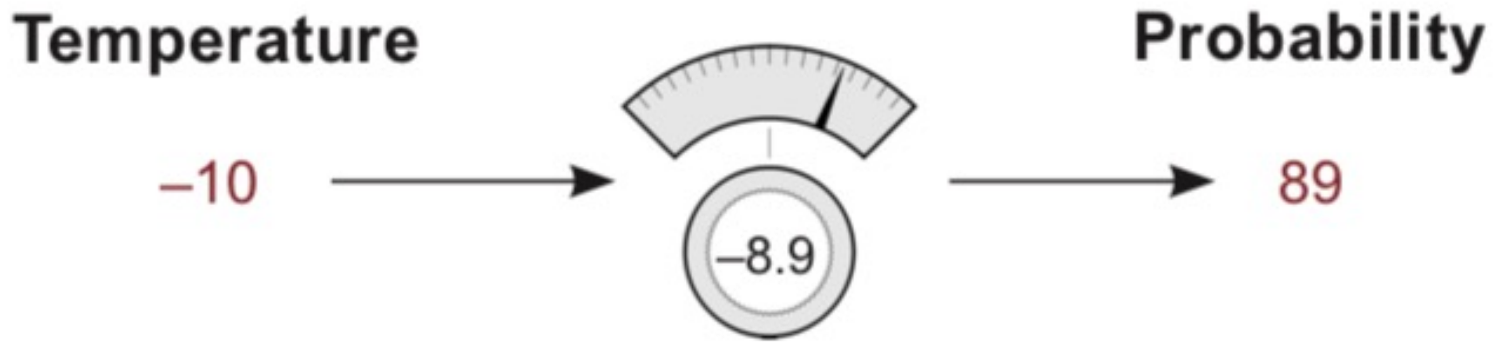


```
number_of_toes = [8.5, 9.5, 10, 9]

input = number_of_toes[0]

pred = neural_network(input,weight)
```

Negative Inputs or Weights

If we 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

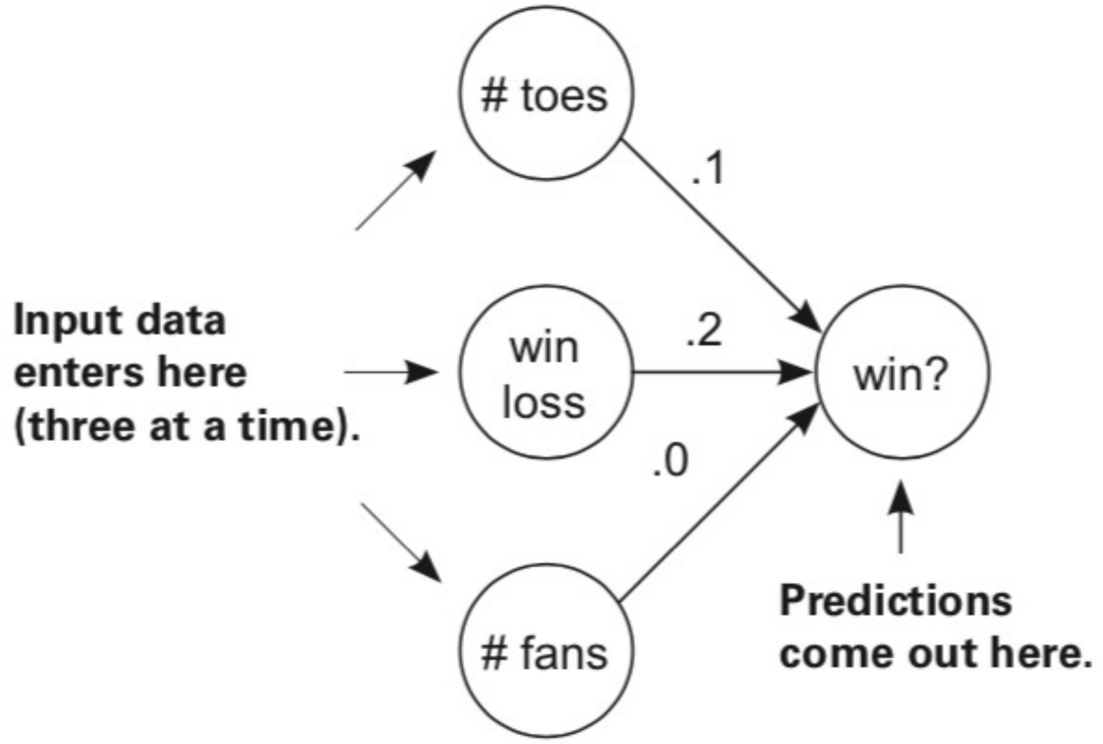
What if you could give the network more information (at one time) than just the average number of toes per player?

In that case, the network should, in theory, be able to make more-accurate predictions.

A network can accept multiple input datapoints at a time.

Multiple Inputs

❶ **An empty network with multiple inputs**

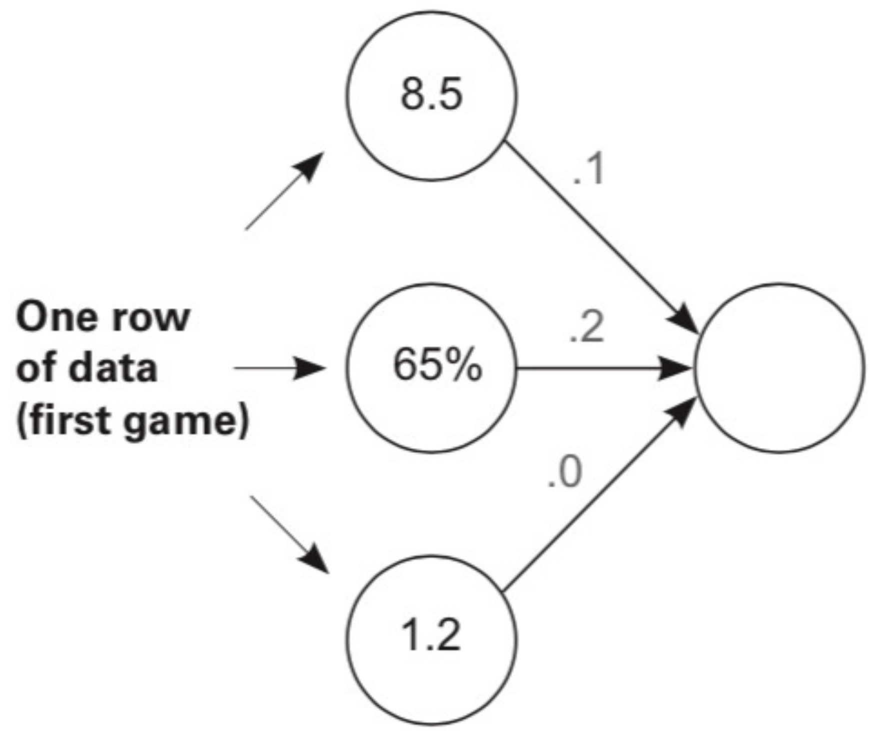


# toes
win loss
# fans
.1
.2
.0
win?
Input data enters here (three at a time).
Predictions come out here.

```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

Multiple Inputs & Vectors

❷ Inserting one input datapoint

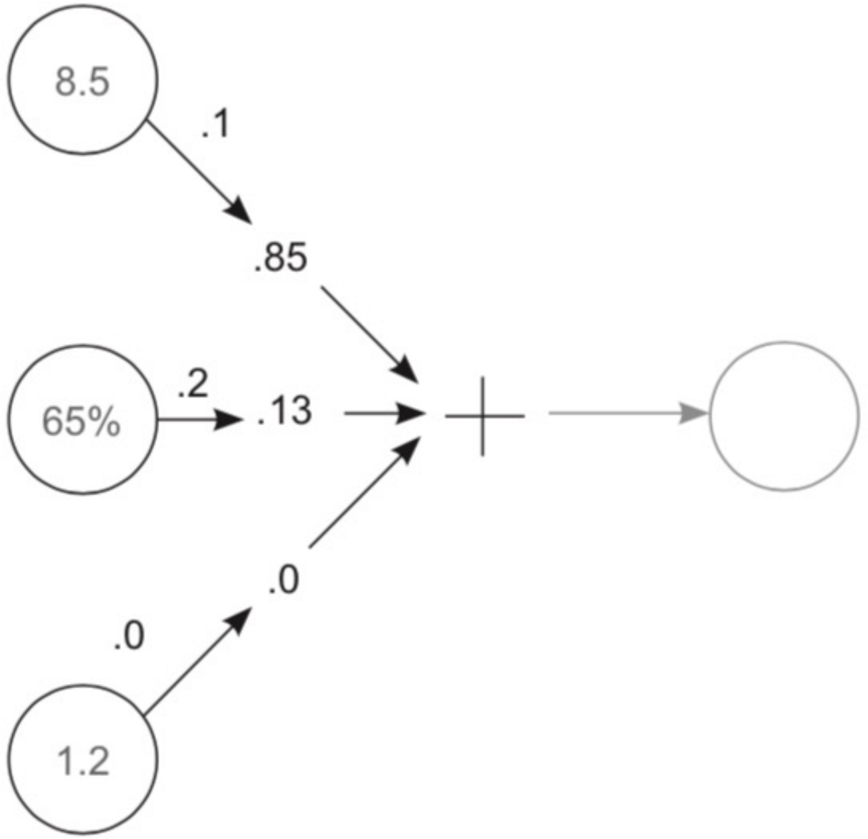


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions).

```
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.

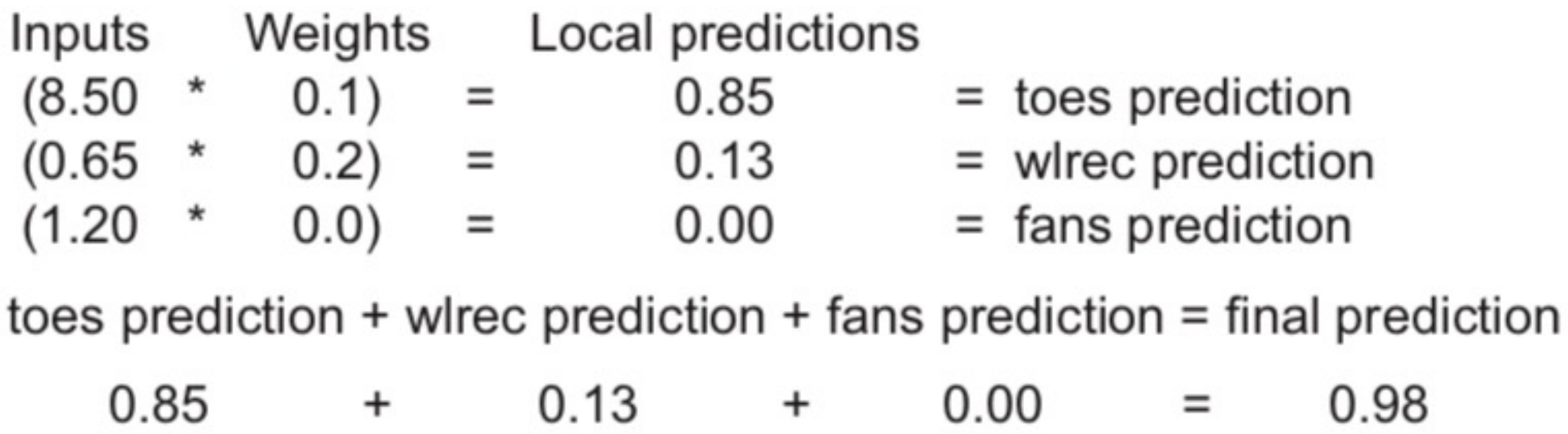
❸ Performing a weighted sum of inputs



```
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
```

Dot Products



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

Element wise Operations

Element wise Addation – Sum two vectors

Element wise Multiplication – Multiply two vectors

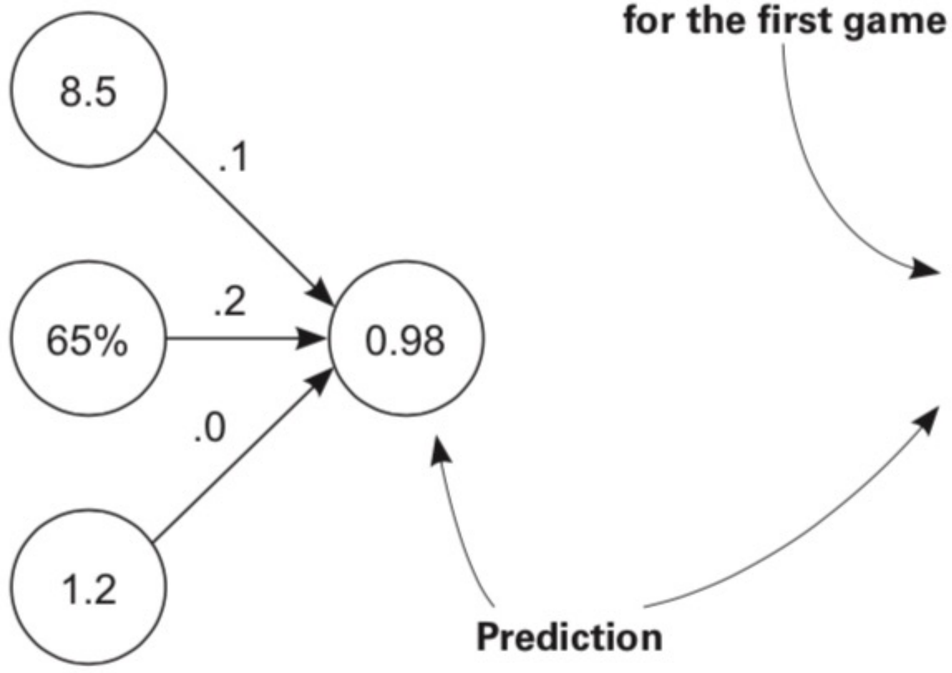
Dot Products

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.

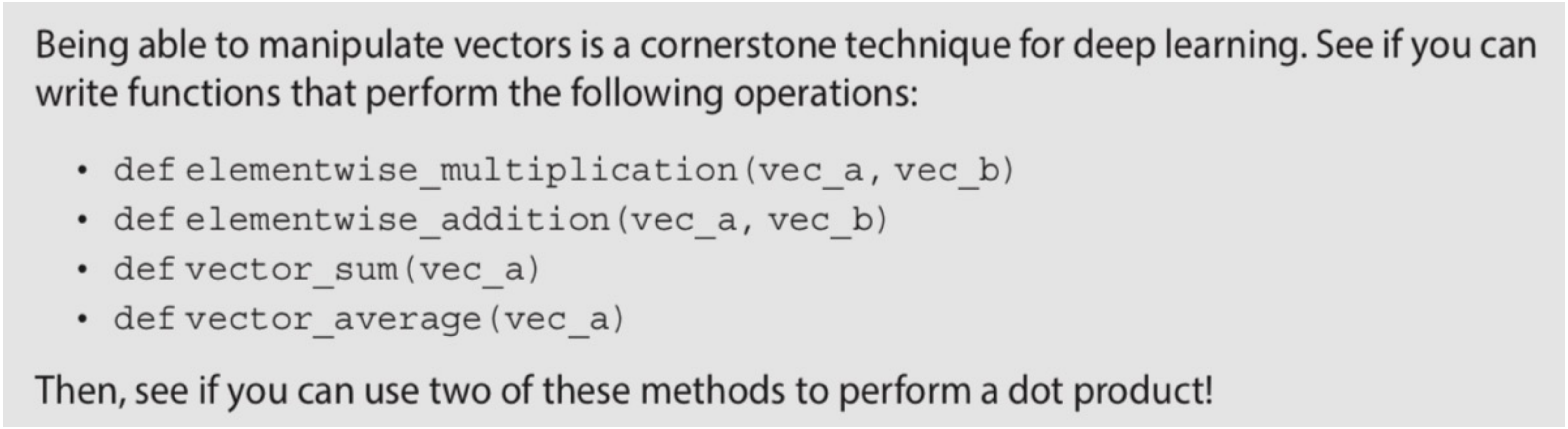
Some also refer to the weighted sum as a *dot product*,

❹ Depositing the prediction


8.5
65%
1.2
.1
.2
.0
0.98
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)
```

Exercise 1



Being able to manipulate vectors is a cornerstone technique for deep learning. See if you can write functions that perform the following operations:

- `def elementwise_multiplication(vec_a, vec_b)`
- `def elementwise_addition(vec_a, vec_b)`
- `def vector_sum(vec_a)`
- `def vector_average(vec_a)`

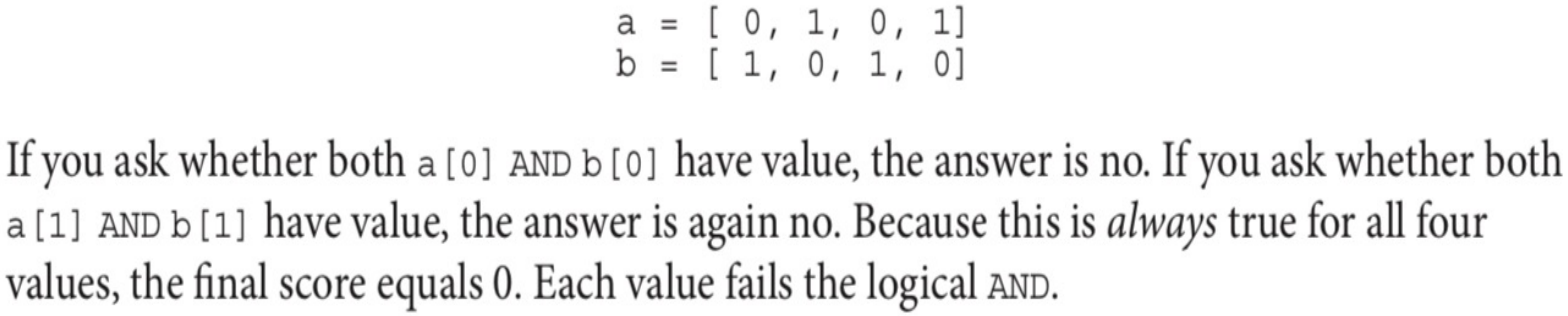
Then, see if you can use two of these methods to perform a dot product!

Weighted Sum

```
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
```

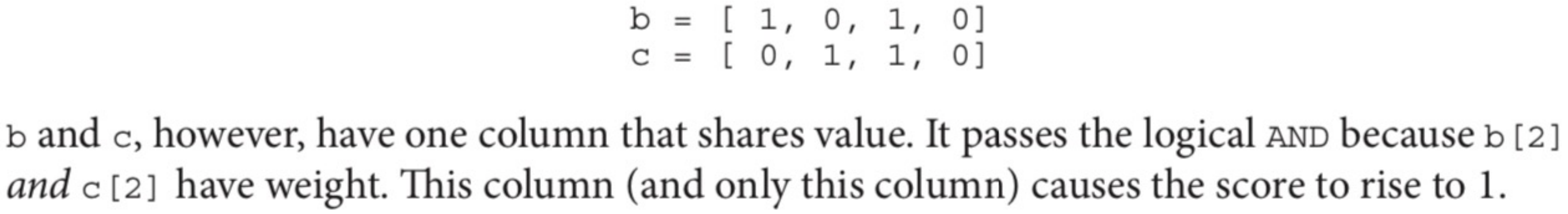
Property 1 – Dot product to a Logical AND



```
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.

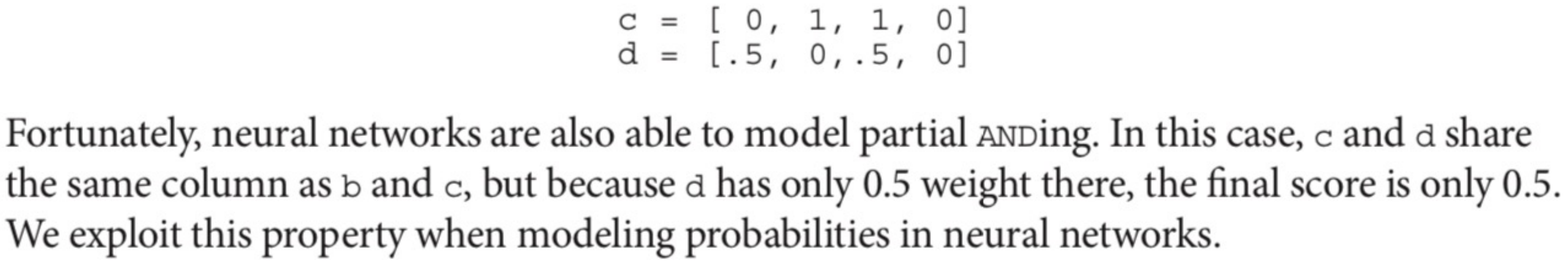
Property 1 – Dot product to a Logical AND



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

`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.

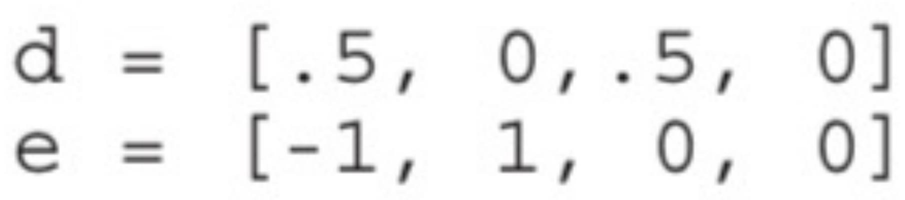
Property 2 – Dot product to a Partial Logical AND



```
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.

Property 3 – Dot product to a logical NOT operator



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

Negative weights tend to imply a logical `NOT` operator, given that any positive weight paired with a negative weight will cause the score to go down.

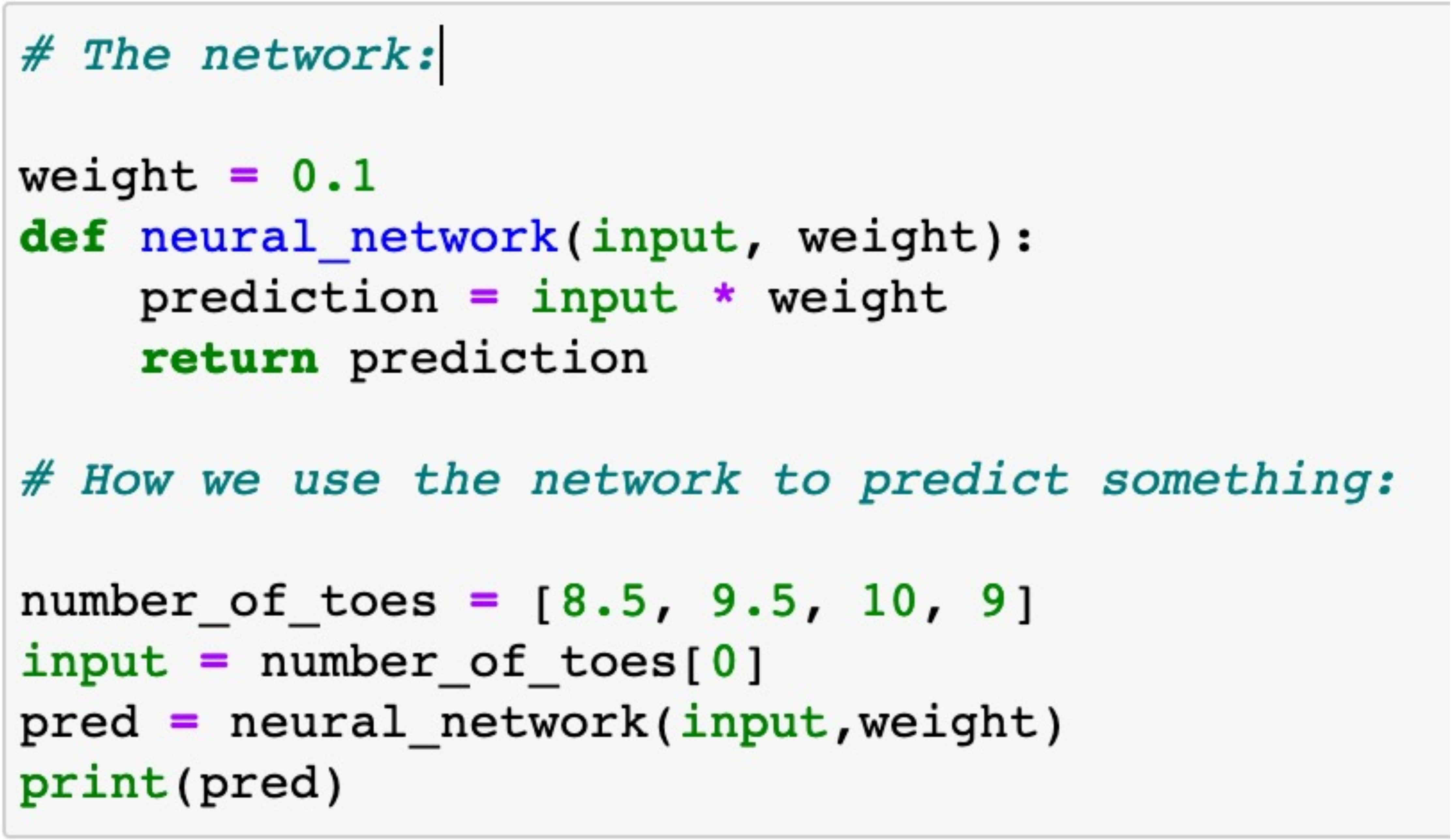
Property 5 – Dot product to a Double Negative

Furthermore, if both vectors have negative weights (such as w_sum(e,e)), then the neural network will perform a *double negative* and add weight instead.

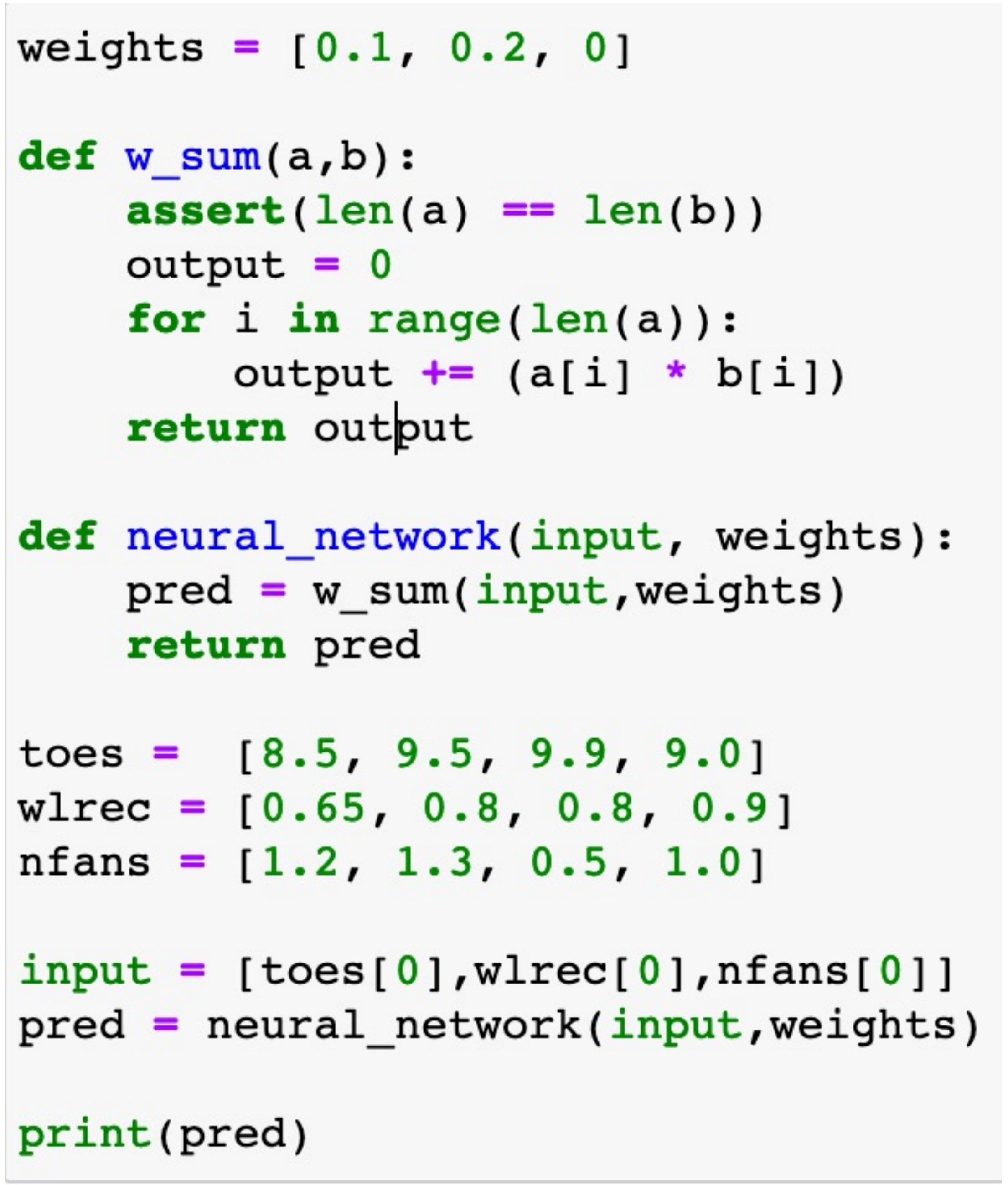
Additionally, some might say it’s an OR after the AND, because if any of the rows show weight, the score is affected.

Thus, for w_sum(a,b), if (a[0] AND b[0]) OR (a[1] AND b[1]), and so on, then w_sum(a,b) returns a positive score.

Furthermore, if one value is negative, then that column gets a NOT.



```
# The network:

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

# How we use the network to predict something:

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



```python
weights = [0.1, 0.2, 0]

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

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)
```