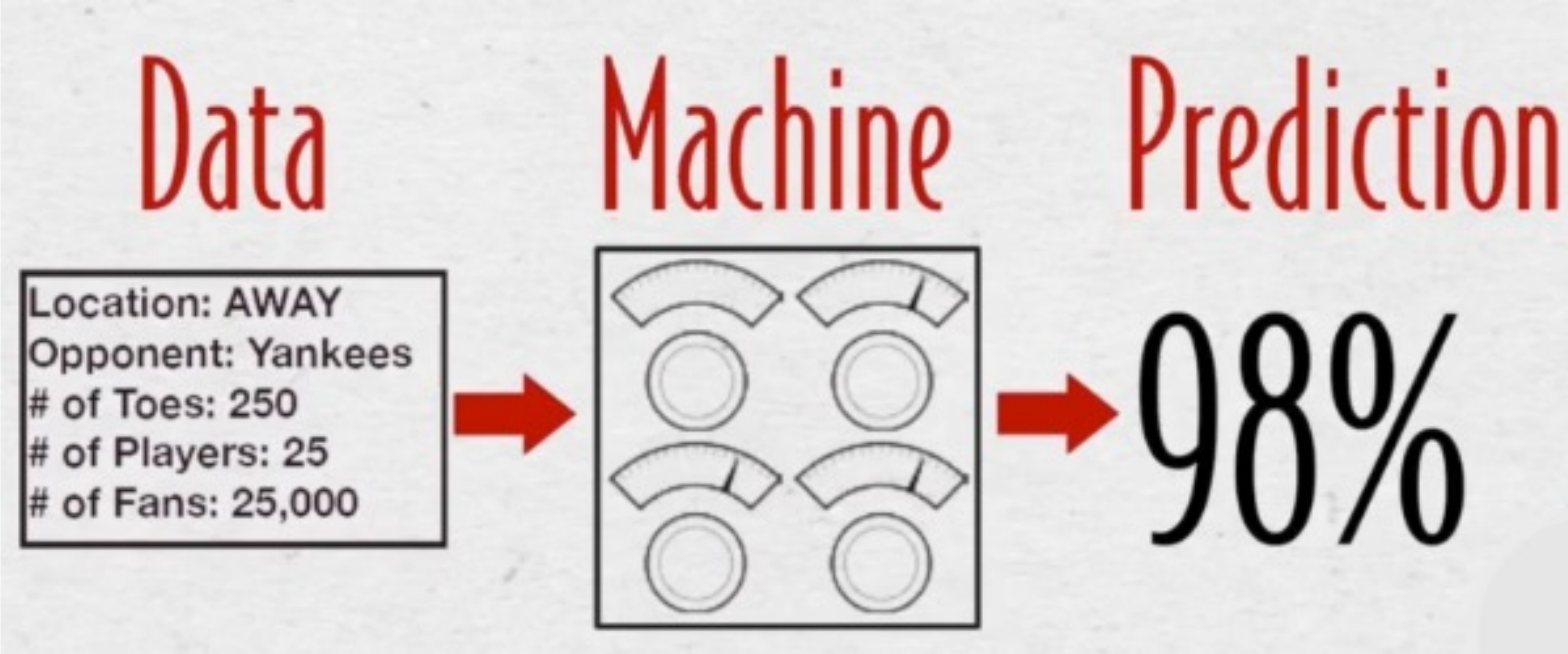
INTRODUCTION TO NEURAL PREDICTION: FORWARD PROPAGATION

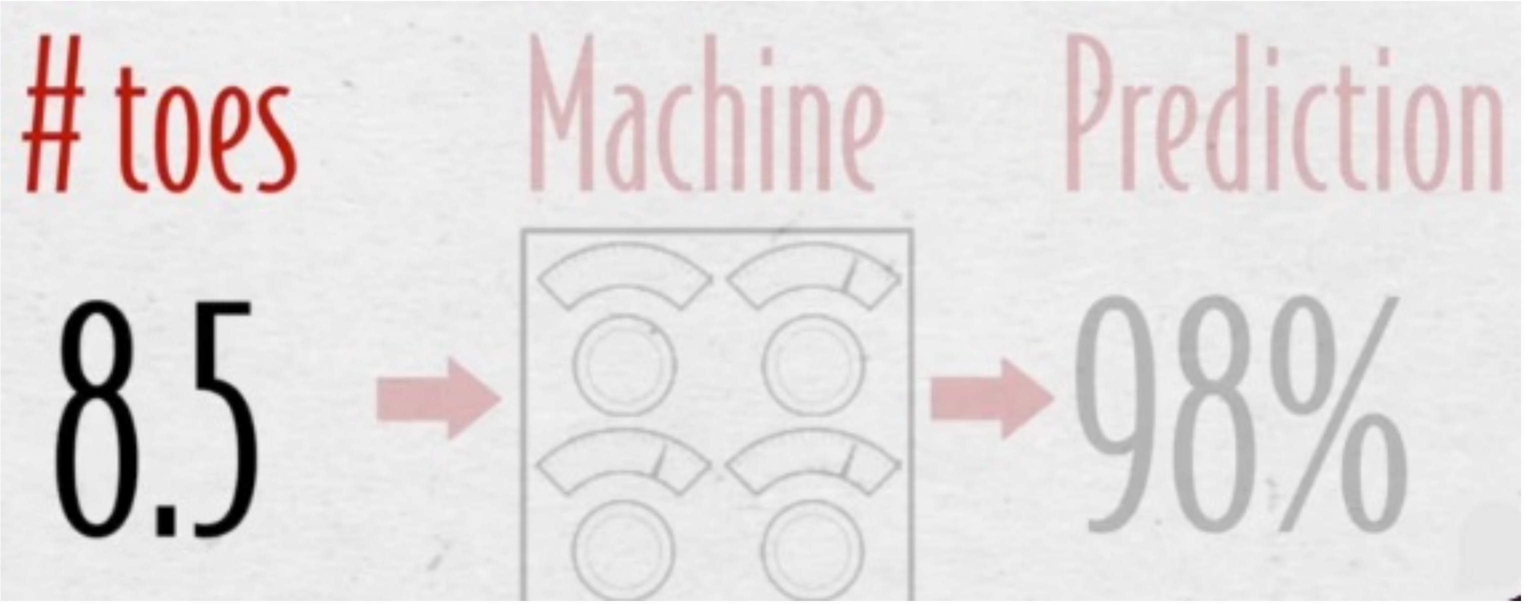
CHAPTER 3

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.

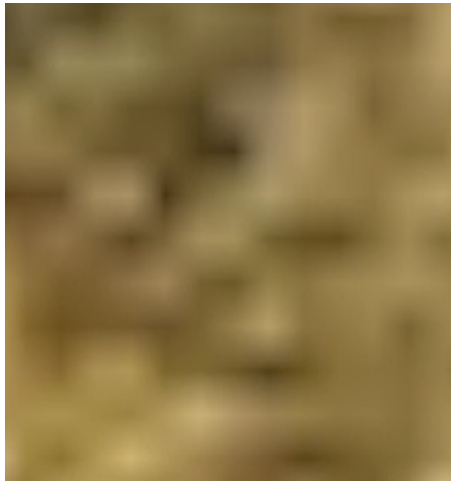
You might be wondering, “How do I choose how many datapoints to propagate at a time?”

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.)

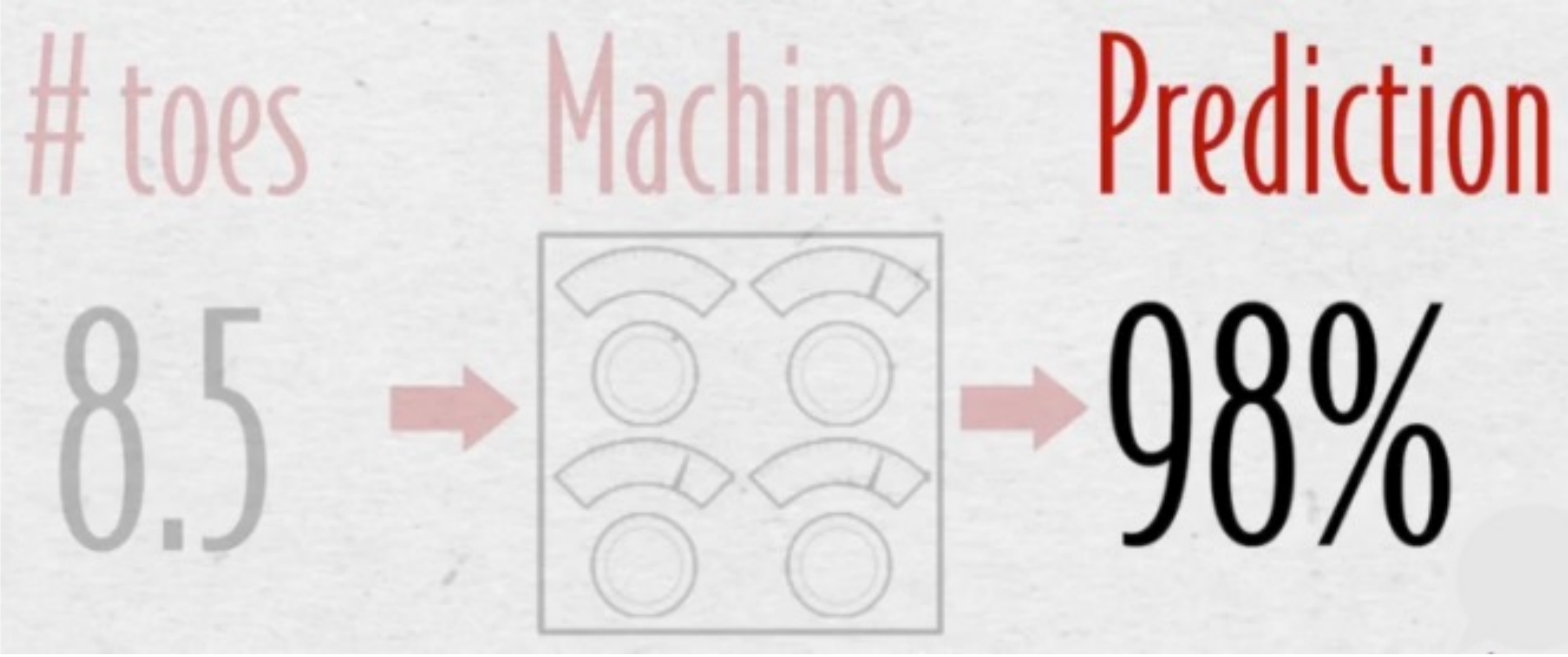


Enough Information?

Always present enough information!

How much would a human need?

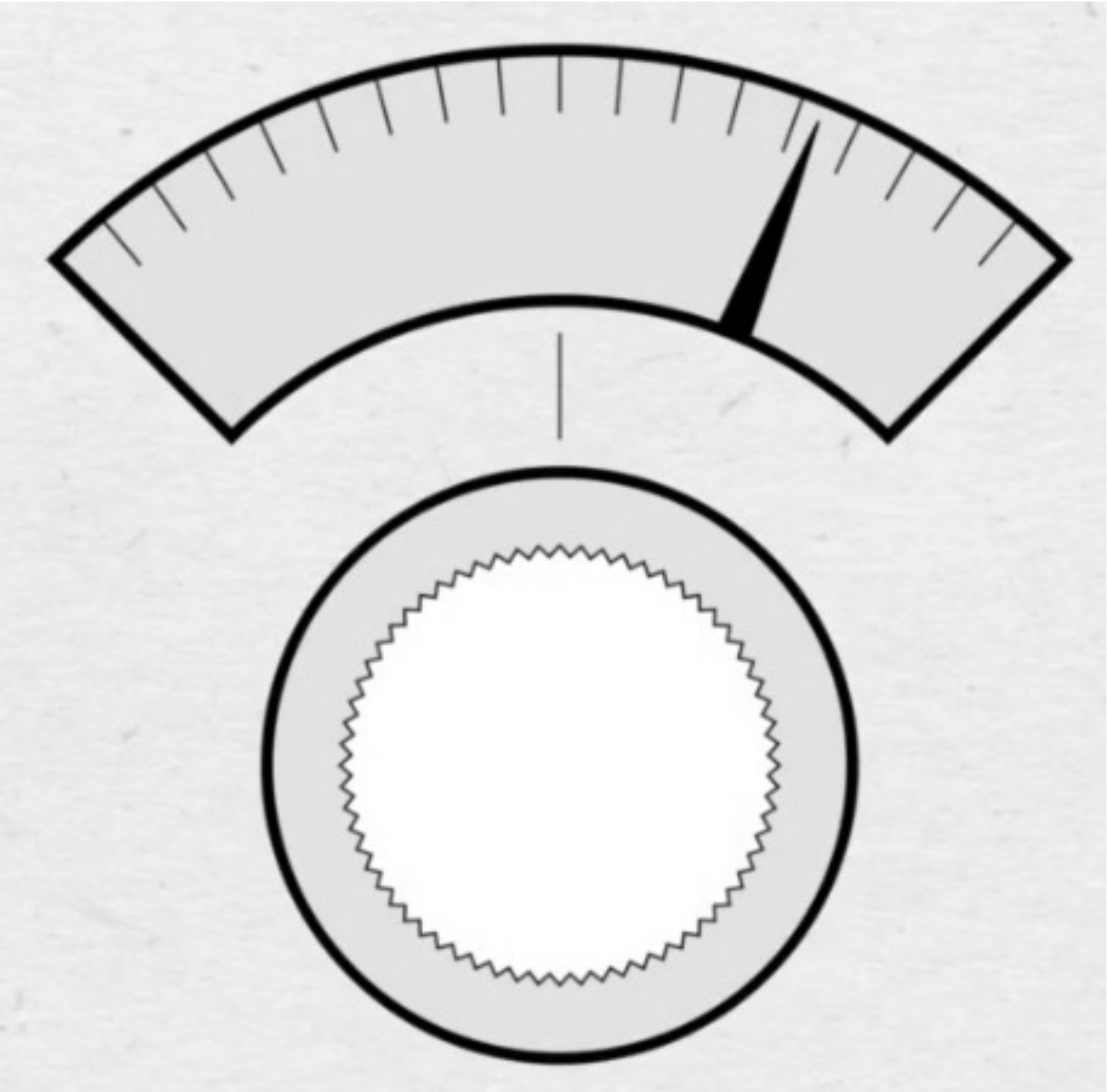
Prediciton



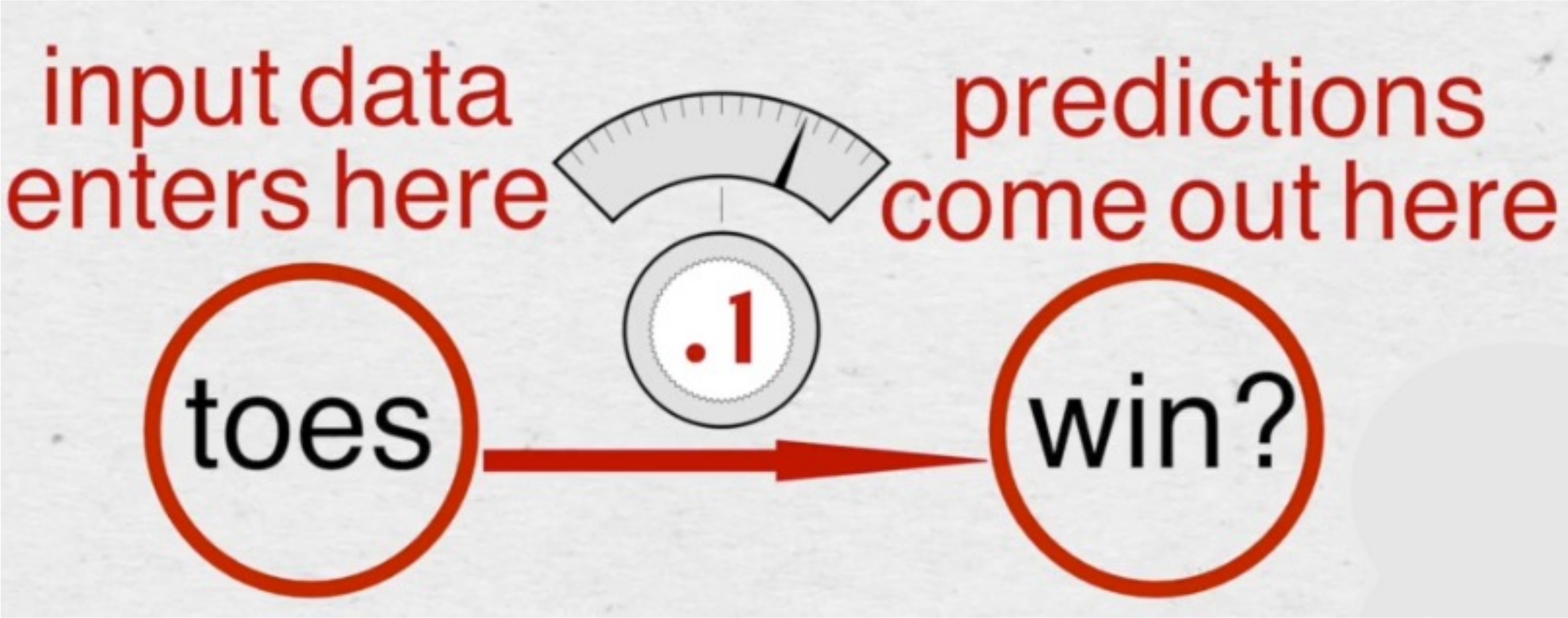
Machine

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



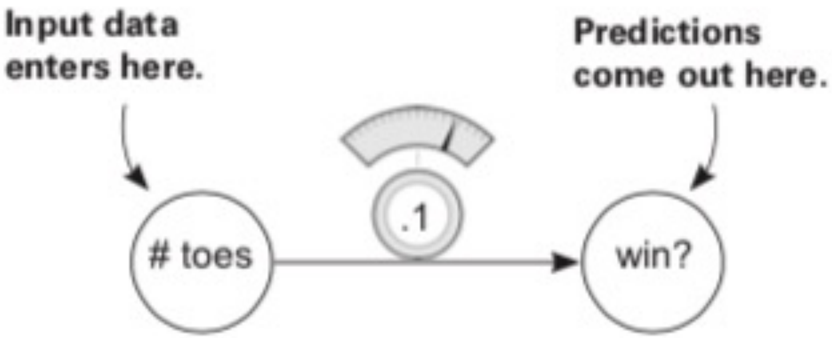
Weights



A simple neural network making a prediction

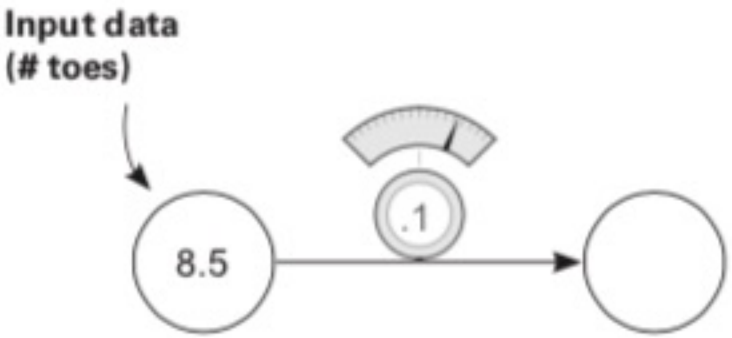
```python
# The network:

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



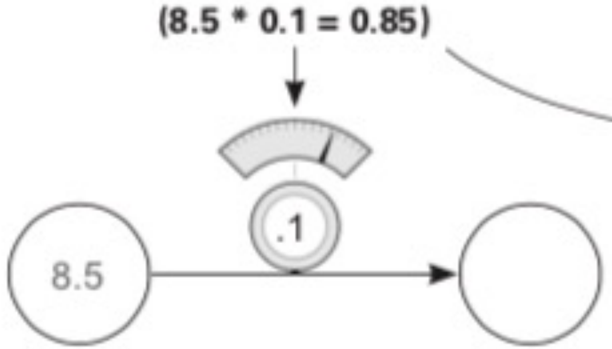

```
weight = 0.1

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

❷ Inserting one input datapoint



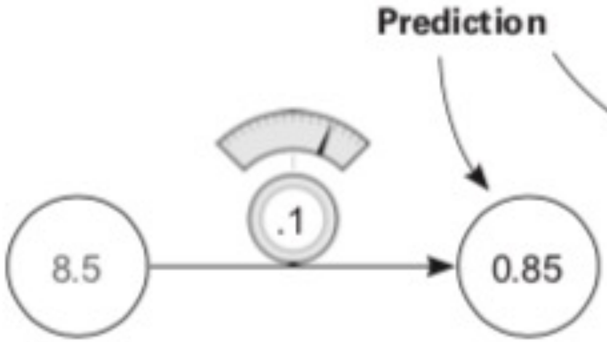

```
number_of_toes = [8.5, 9.5, 10, 9]

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)

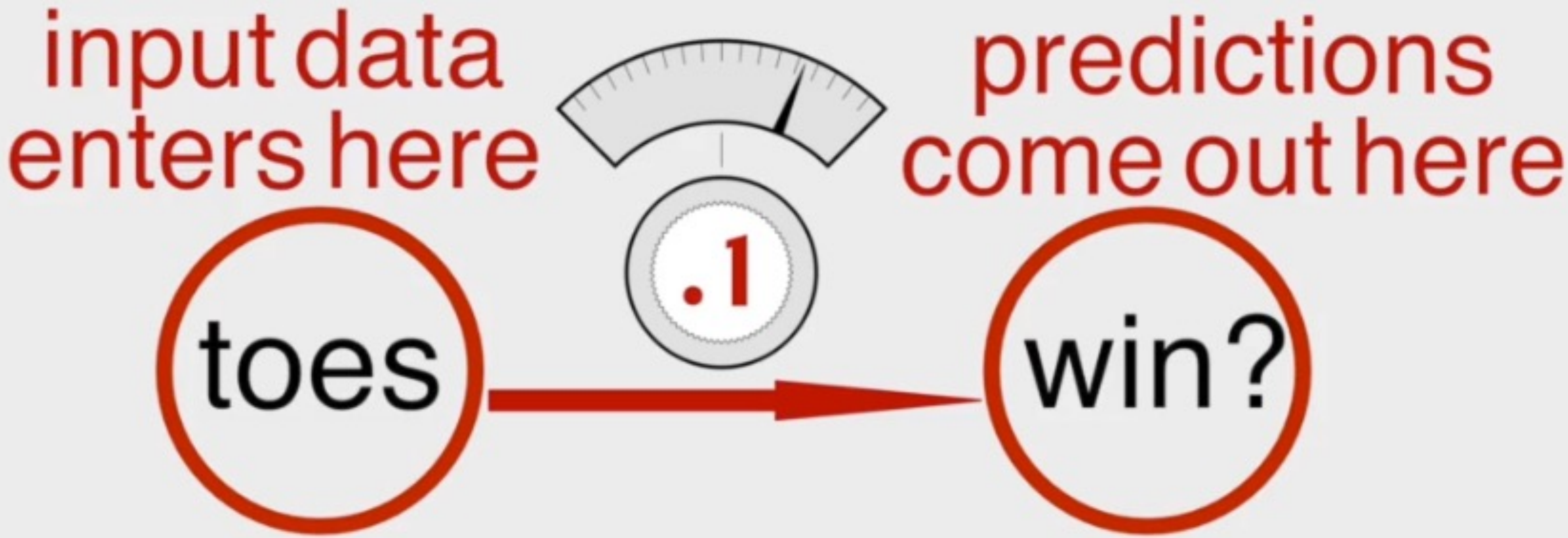
Logout

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
```

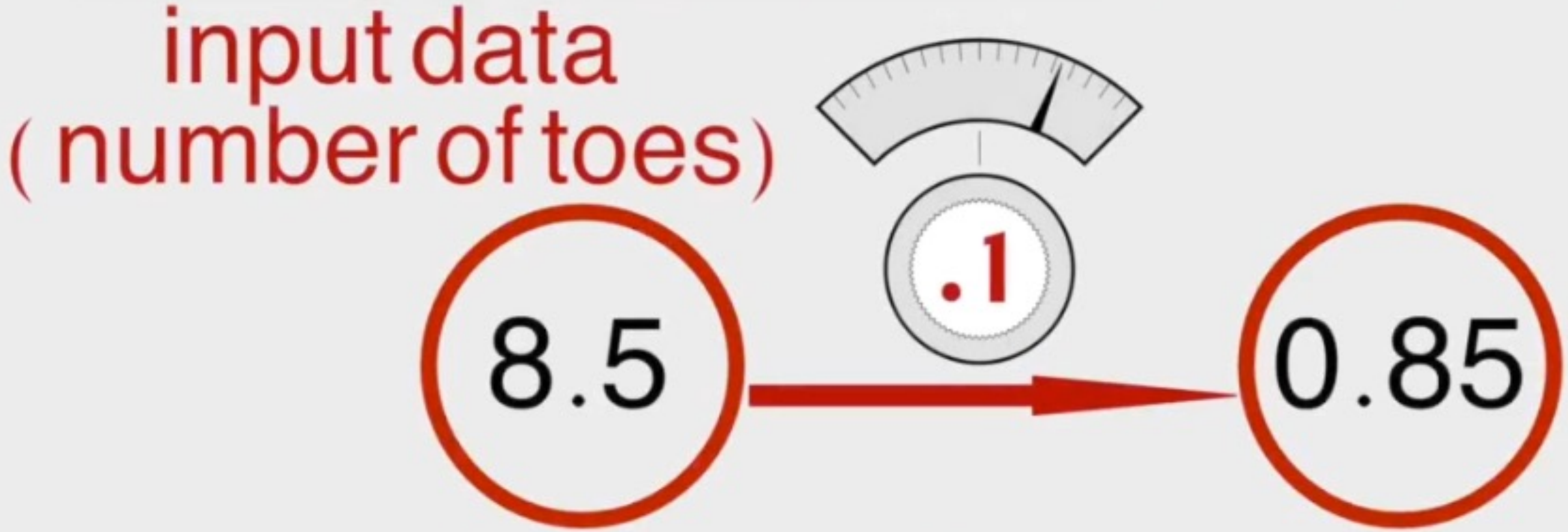


File Edit View Insert Cell Kernel Widgets Help Trusted Python 2

Run Code

In [ ]:

```python
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

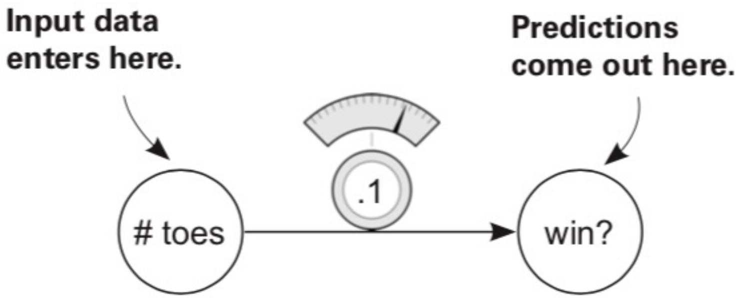
✖️ Learn

Sensitivity

What does this neural network do?

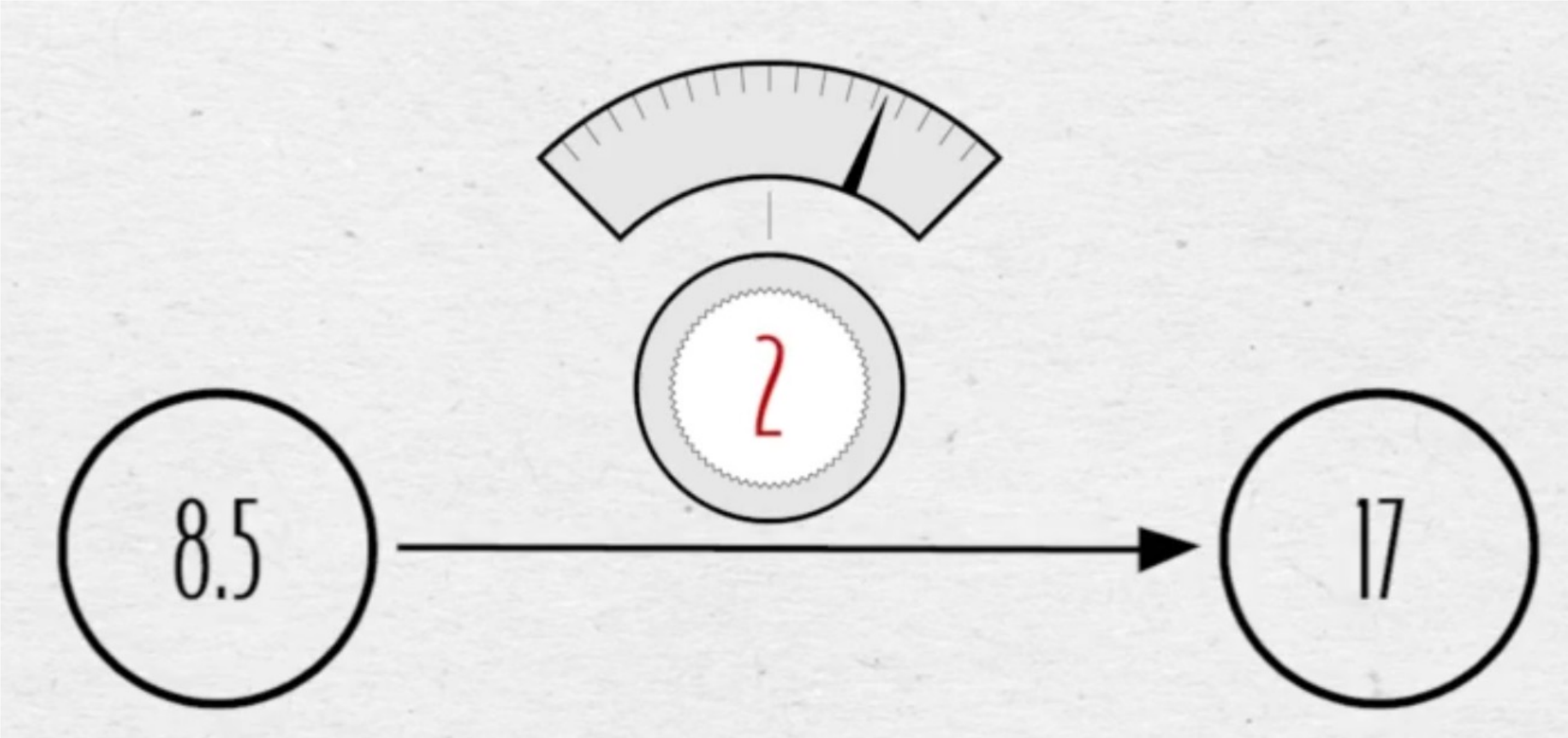
It multiplies the input by a weight. It “scales” the input by a certain amount. It accepts an input variable as information and a weight variable as knowledge and outputs a prediction.

❶ An empty network

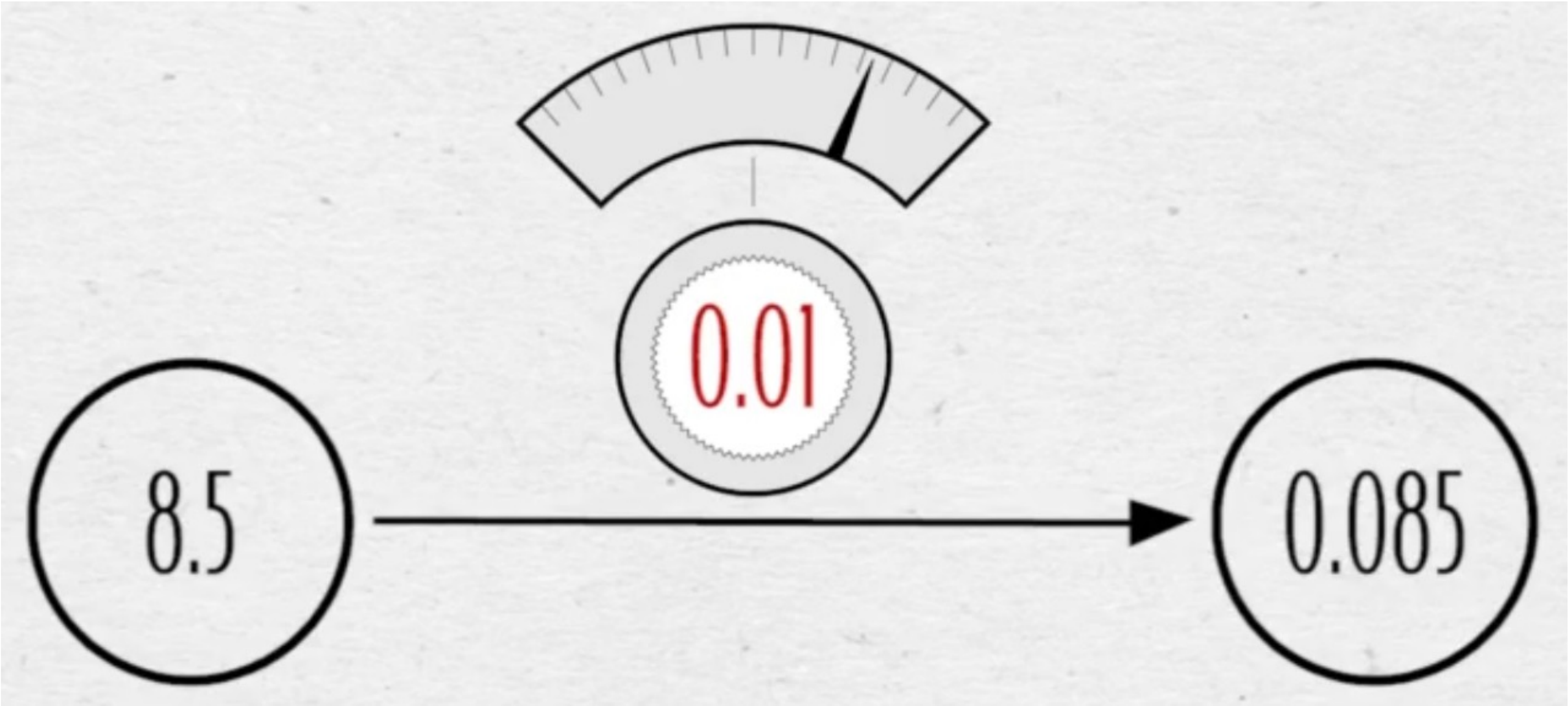



```
weight = 0.1

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)
```

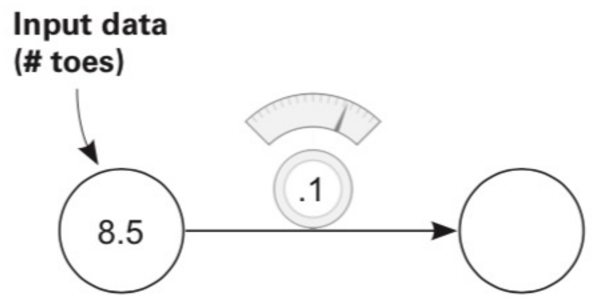
```
8.5e-05
```

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.

2 Inserting one input datapoint

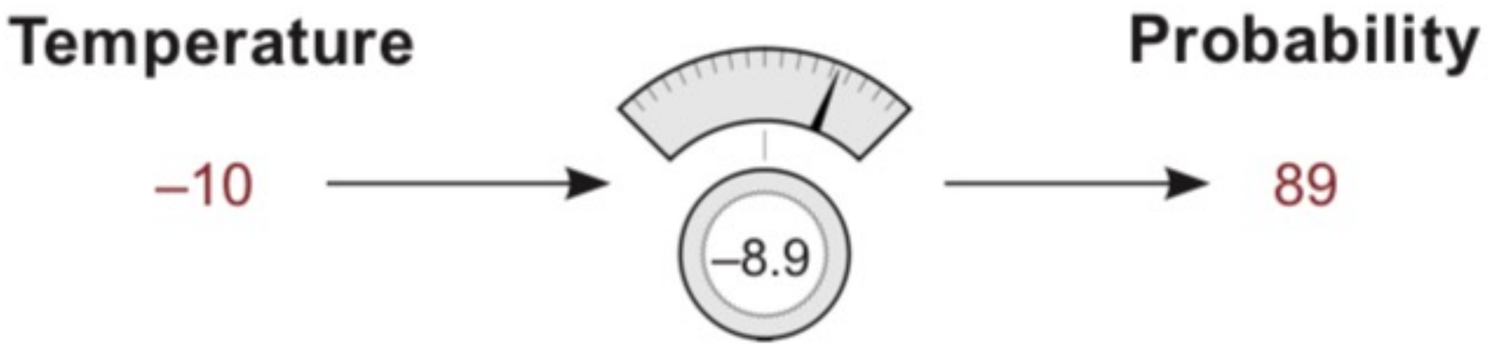



```
number_of_toes = [8.5, 9.5, 10, 9]

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

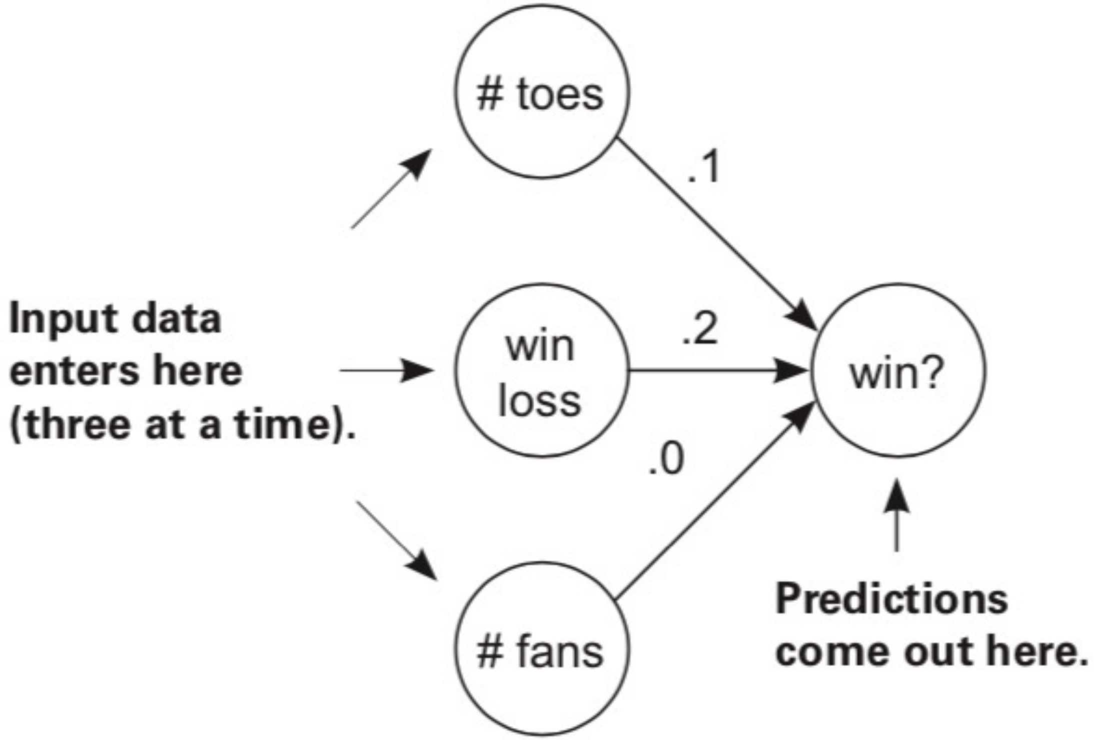
What if you could give the network more information (at one time) than just the average number of toes per player?

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

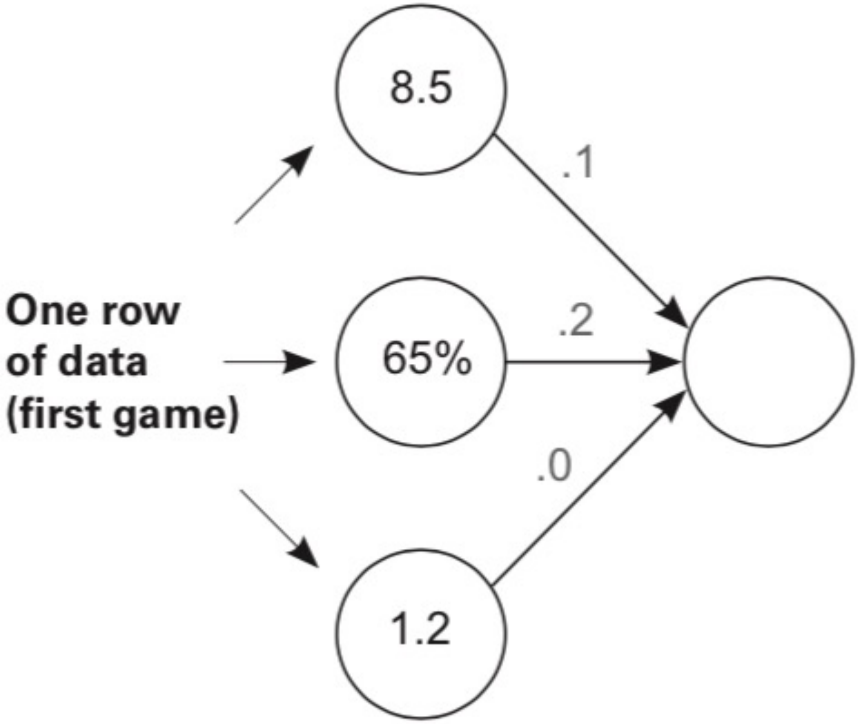
❶ **An empty network with multiple inputs**




```
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

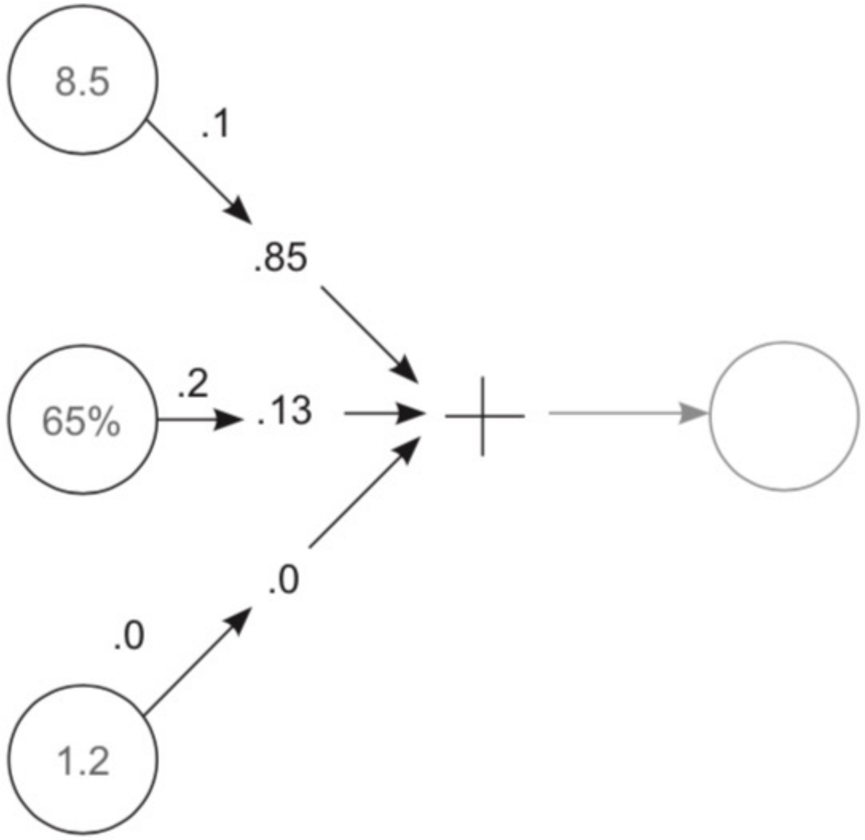


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions).

```
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

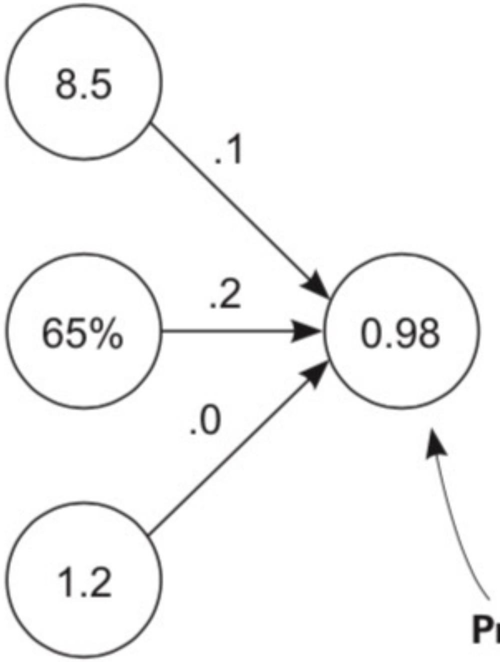
you multiply each input by its respective weight and then sum all the local predictions together.

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

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)
```

Prediction

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.

```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)
```

```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)
```

```
0.9800000000000001
```

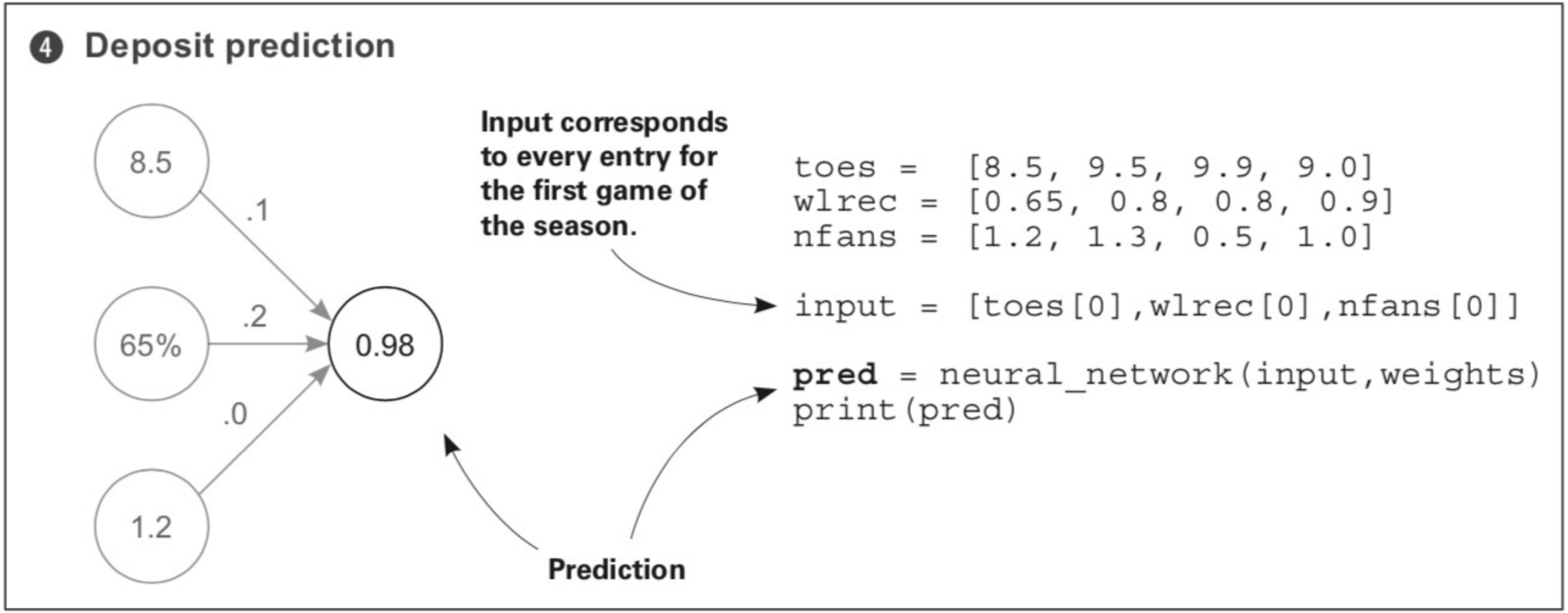
NumPy Code

```
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 corresponds to every entry
# for the first game of the season.

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

print(pred)
```

```
0.9800000000000001
```

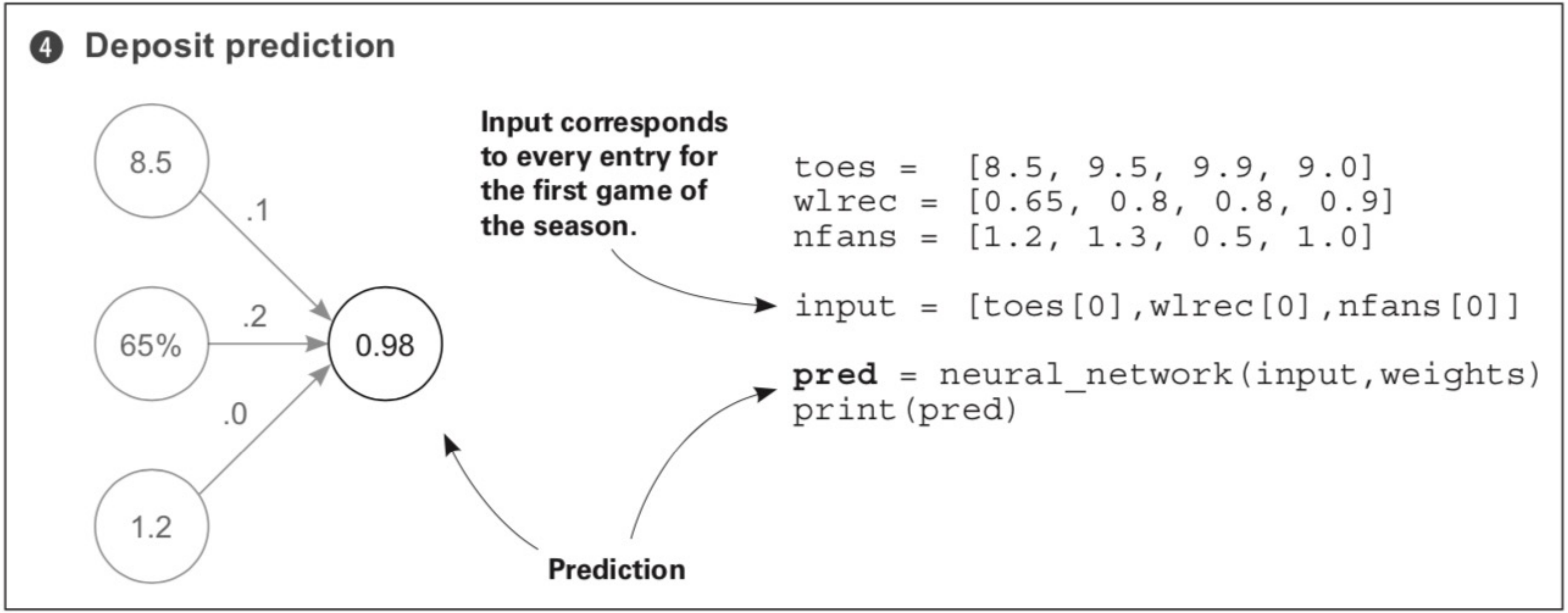
Inputs, Weights and Predections

The network gives a high score of the inputs based on *how similar they are to the weights*.



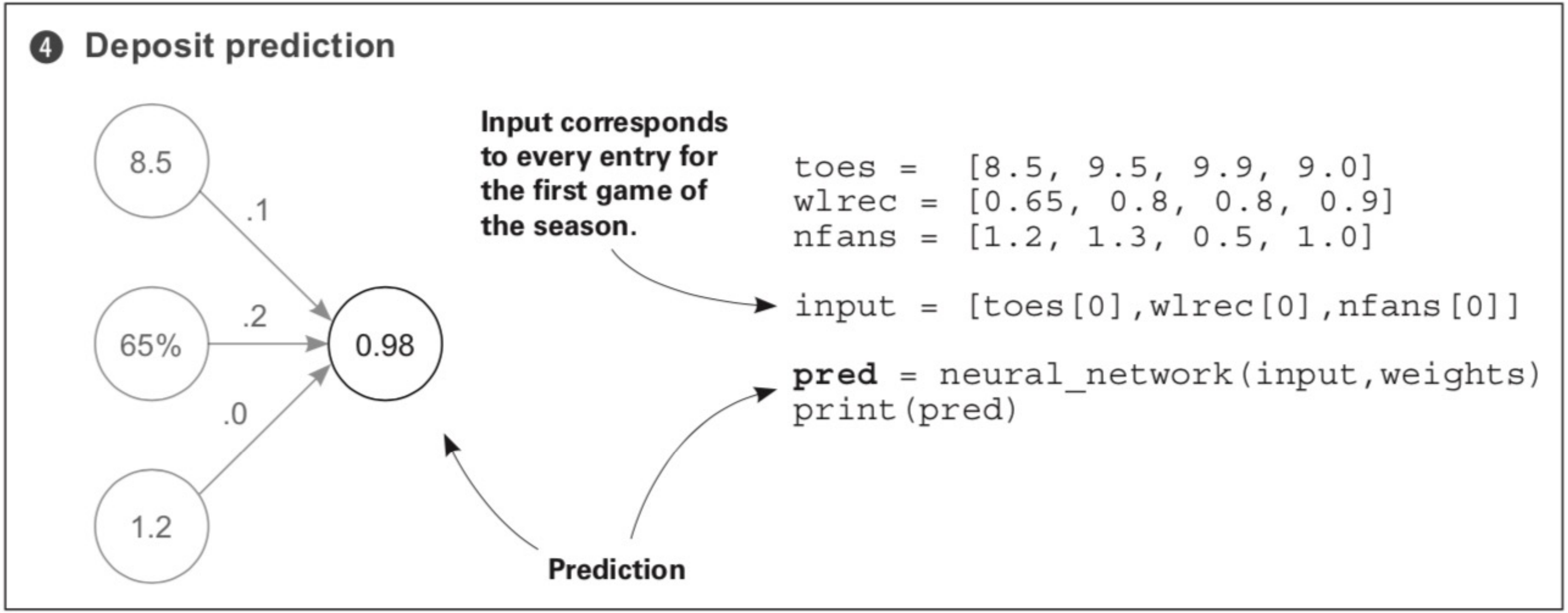
Inputs, Weights and Predections

1. In the example nfans is completely ignored in the prediction because the weight associated with it is 0.



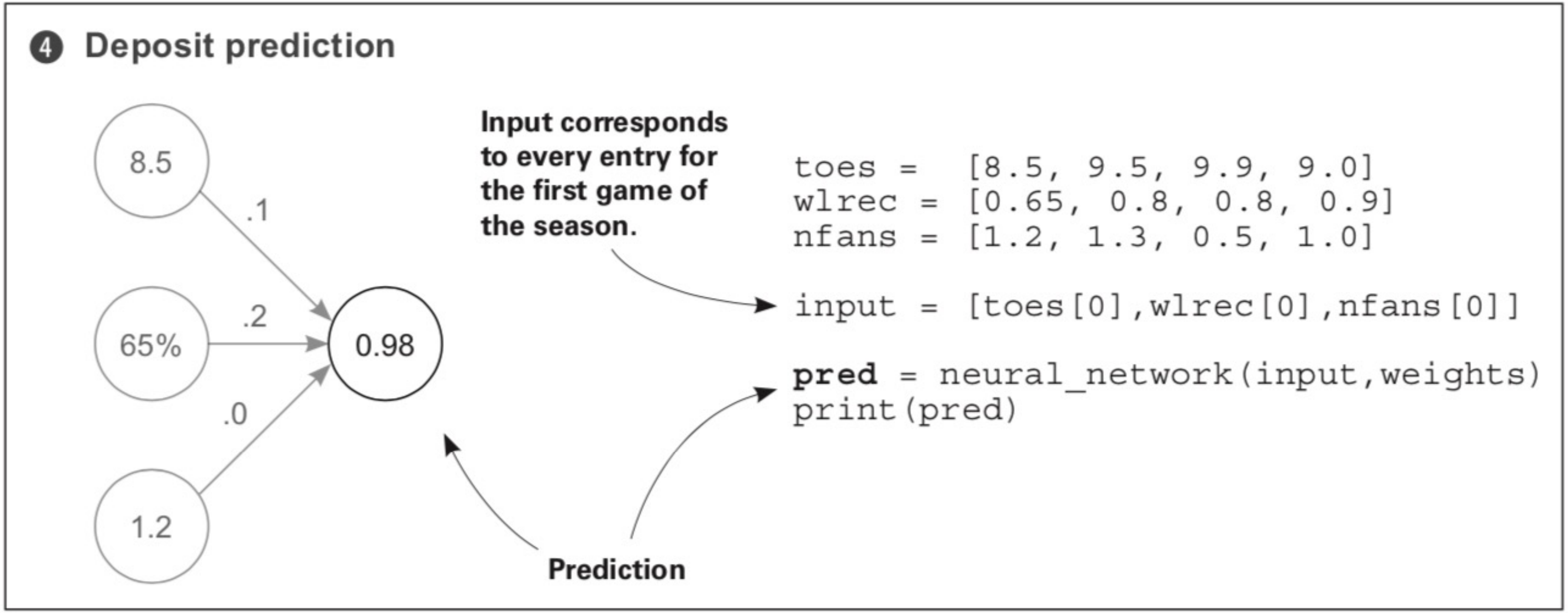
Inputs, Weights and Predections

1. In the example nfans is completely ignored in the prediction because the weight associated with it is 0.



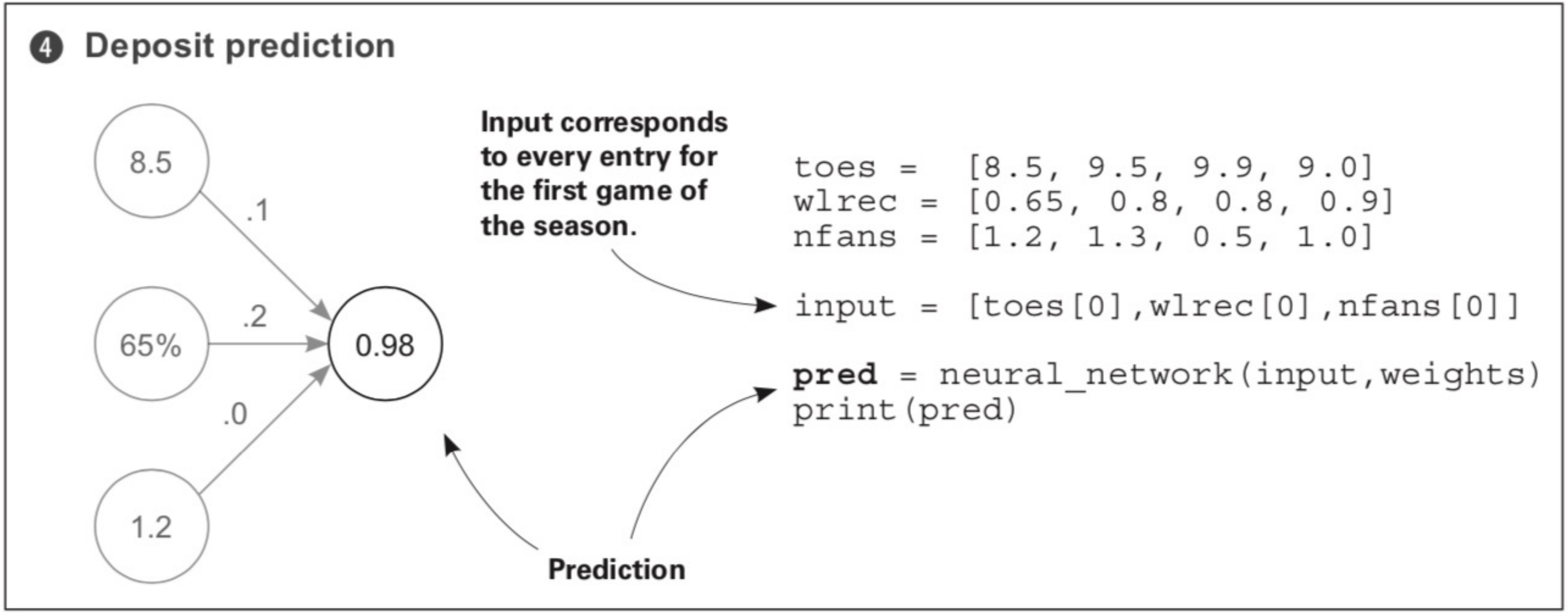
Inputs, Weights and Predections

2. The most sensitive predictor is wlrec because its weight is 0.2.



Inputs, Weights and Predections

3. 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.



Inputs, Weights and Predections

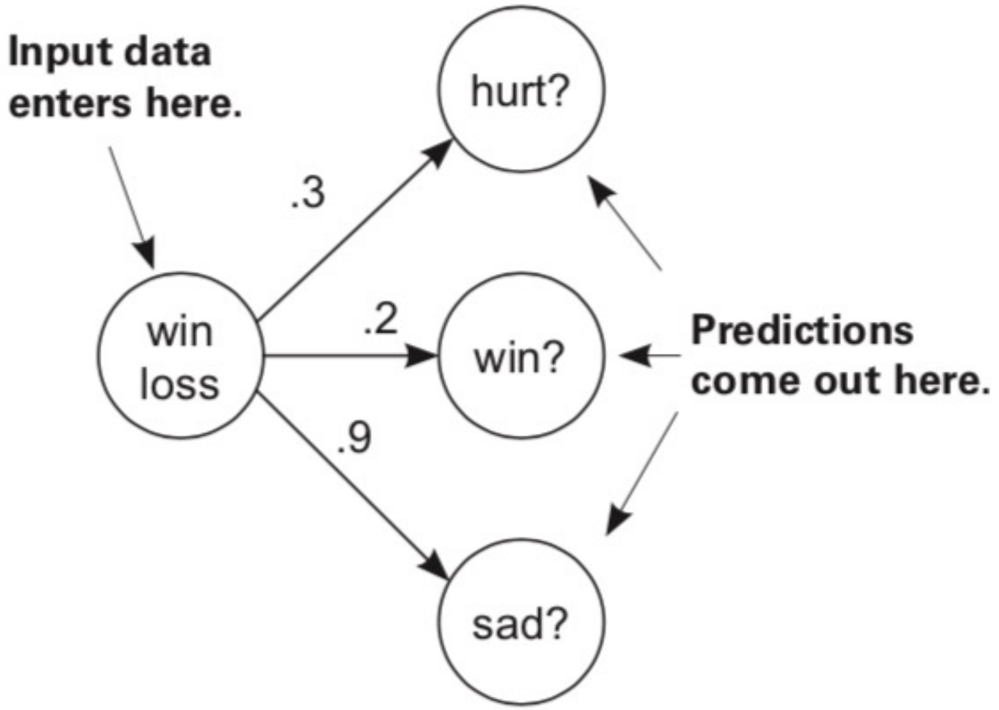
1. We can’t shuffle weights: they have specific positions they need to be in.

2. Both the value of the weight and the value of the input determine the overall impact on the final score.

3. Finally, a negative weight will cause some inputs to reduce the final prediction (and vice versa).

Making a prediction with multiple outputs

❶ **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
```

Predection – Un Connected

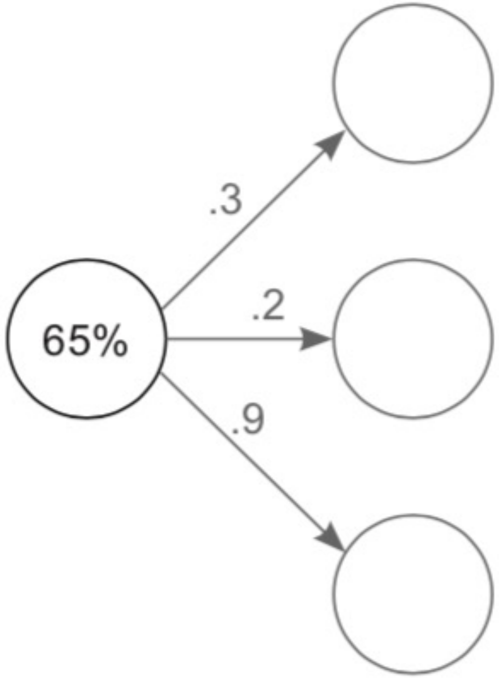
The most important comment in this setting is to notice that the three predictions are completely separate.

Unlike neural networks with multiple inputs and a single output, where the prediction is undeniably connected.

This network truly behaves as three independent components, each receiving the same input data.

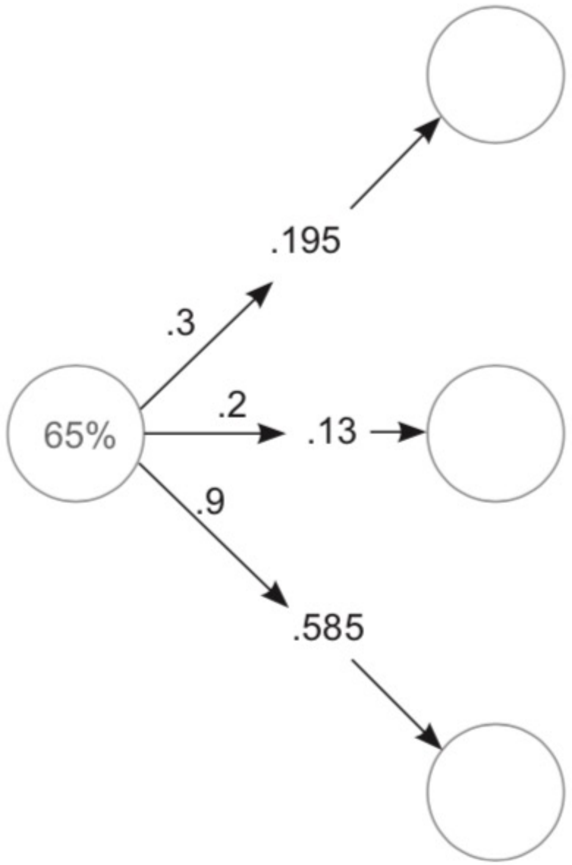
One Input Data Point

❷ Inserting one input datapoint




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

input = wlrec[0]

pred = neural_network(input,weights)
```

❸ 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
```

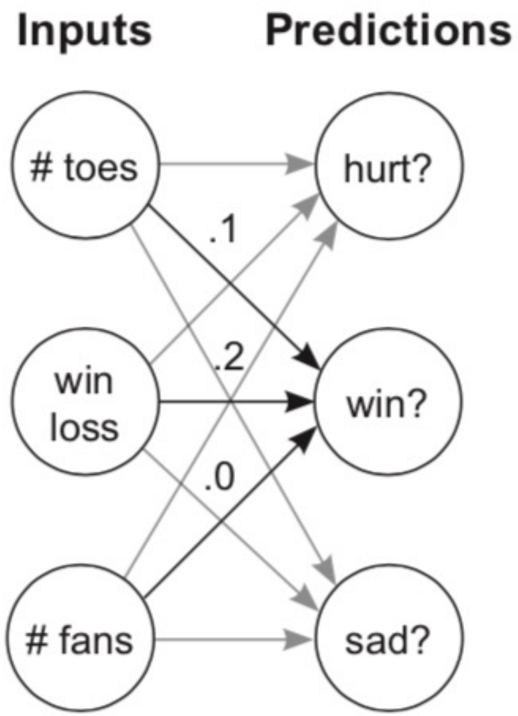
Inputs		Weights		Final predictions		
(0.65	*	0.3)	=	0.195	=	hurt prediction
(0.65	*	0.2)	=	0.13	=	win prediction
(0.65	*	0.9)	=	0.585	=	sad prediction

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

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

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

print(pred)
```

```
[0.195, 0.13, 0.5850000000000001]
```

Predicting with multiple inputs and outputs

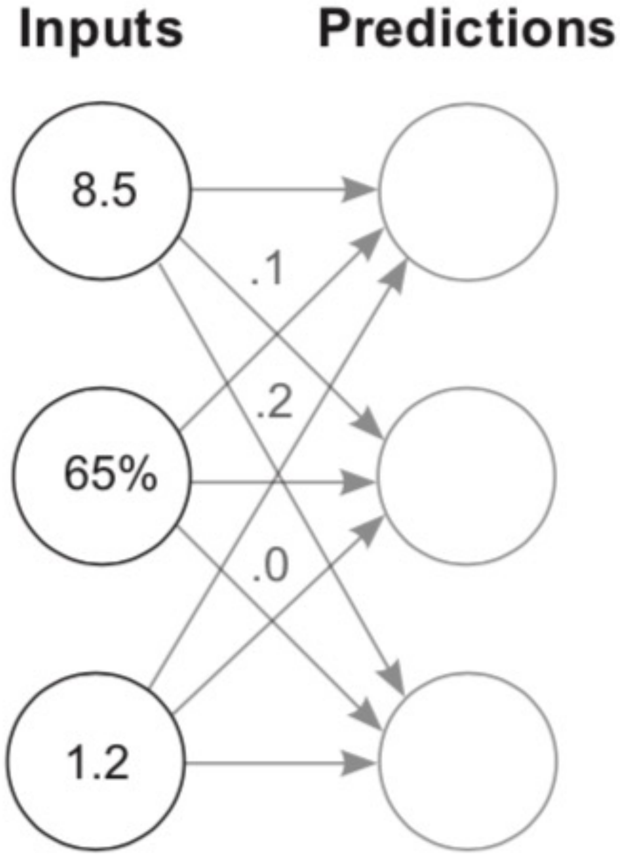
1 An empty network with multiple inputs and outputs




```
                 # 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
```

```
                #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?
```

❷ Inserting one input datapoint

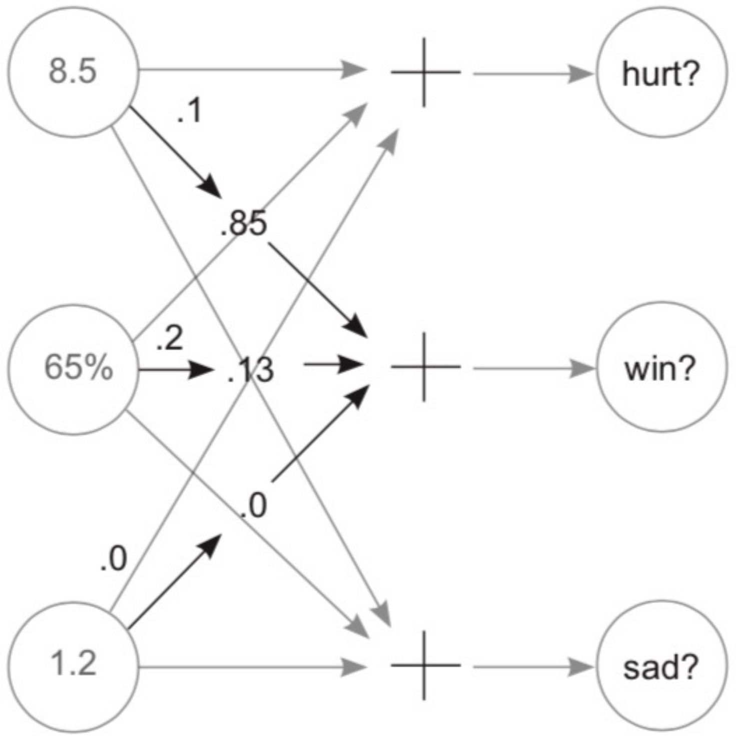


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
fans = fan count (in millions)

```
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)
```

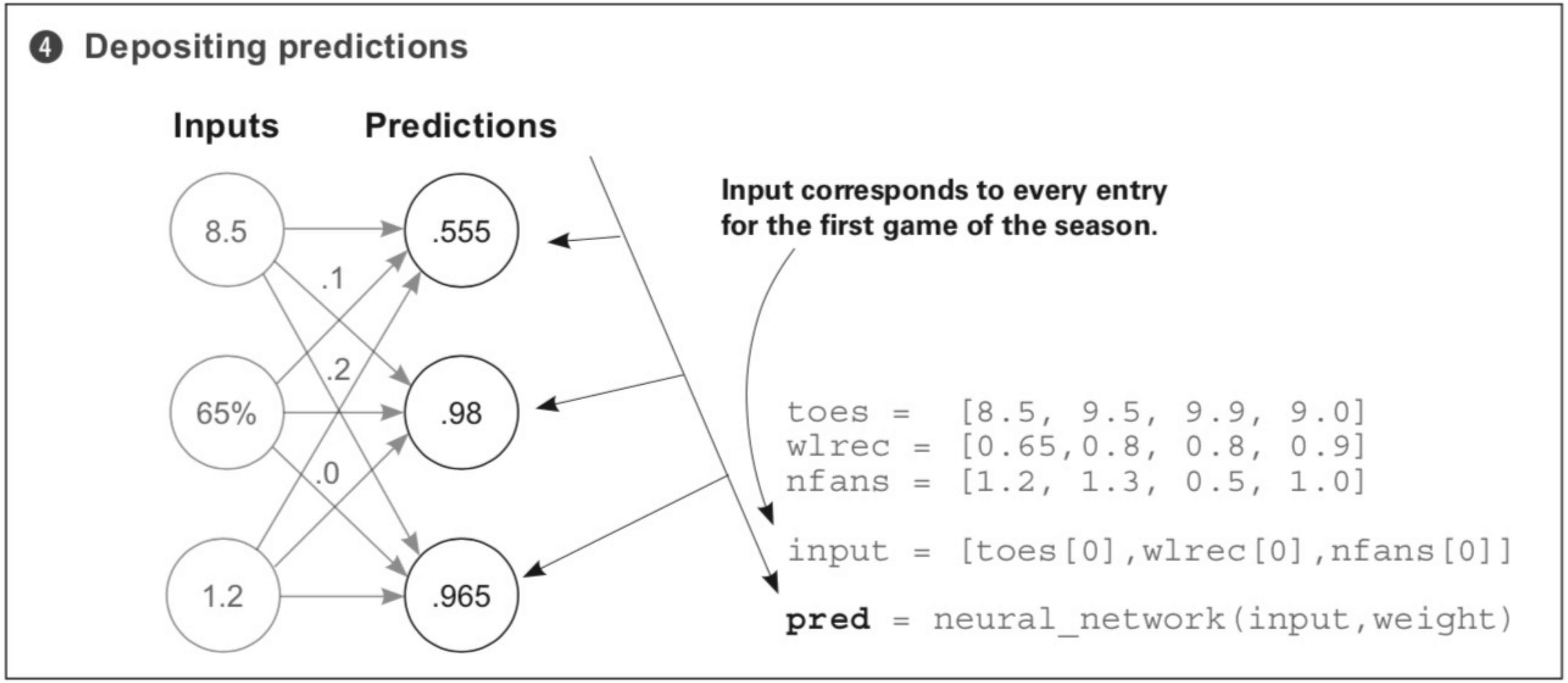
Input corresponds to every entry for the first game of the season.

❸ For each output, performing a weighted sum of inputs




```python
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
```

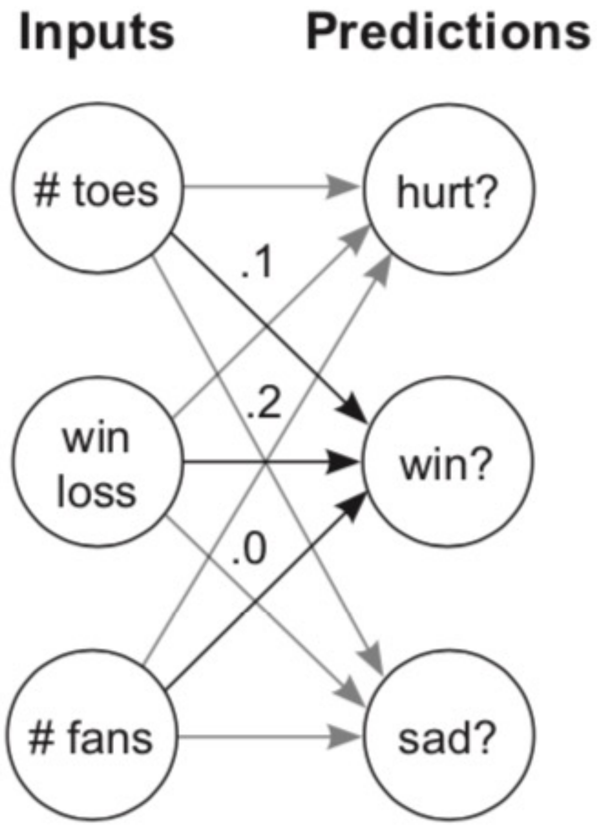
# toes		% win		# fans				
(8.5 * 0.1)	+	(0.65 * 0.1)	+	(1.2 * –0.3)	=	0.555	=	hurt prediction
(8.5 * 0.1)	+	(0.65 * 0.2)	+	(1.2 * 0.0)	=	0.98	=	win prediction
(8.5 * 0.0)	+	(0.65 * 1.3)	+	(1.2 * 0.1)	=	0.965	=	sad prediction



❹ Depositing predictions
Inputs
Predictions
8.5
65%
1.2
.555
.98
.965
.1
.2
.0
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)

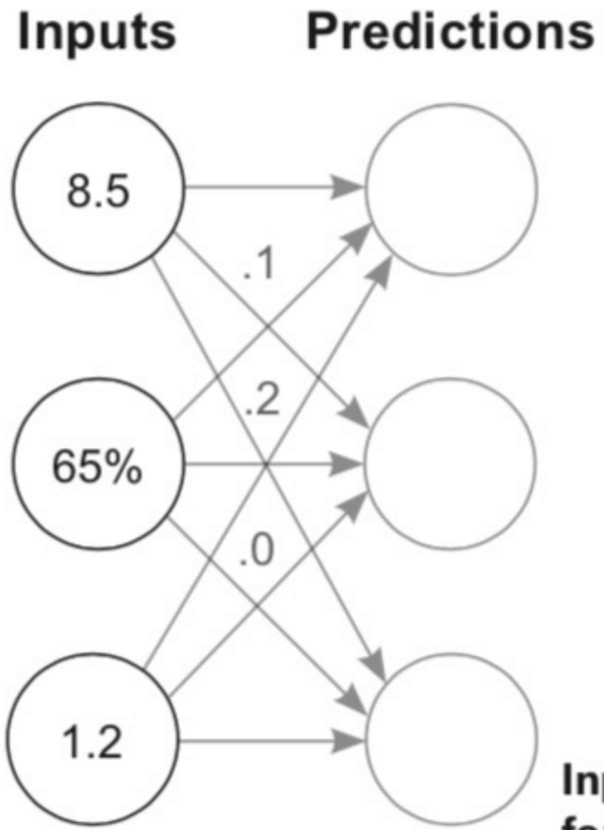
Multiple inputs and outputs: How does it work?

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 neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

❷ Inserting one input datapoint

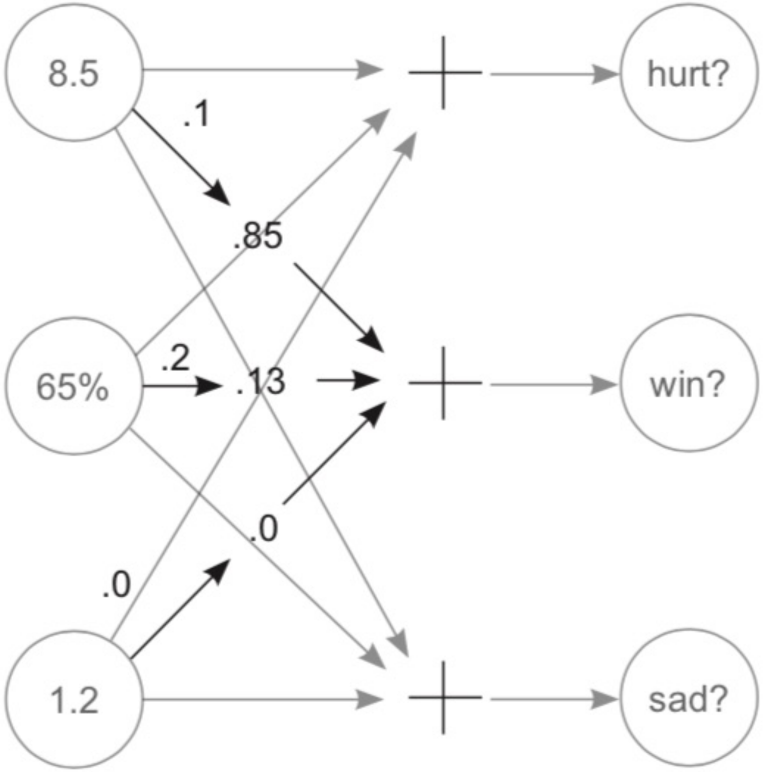


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
fans = fan count (in millions)

```
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.

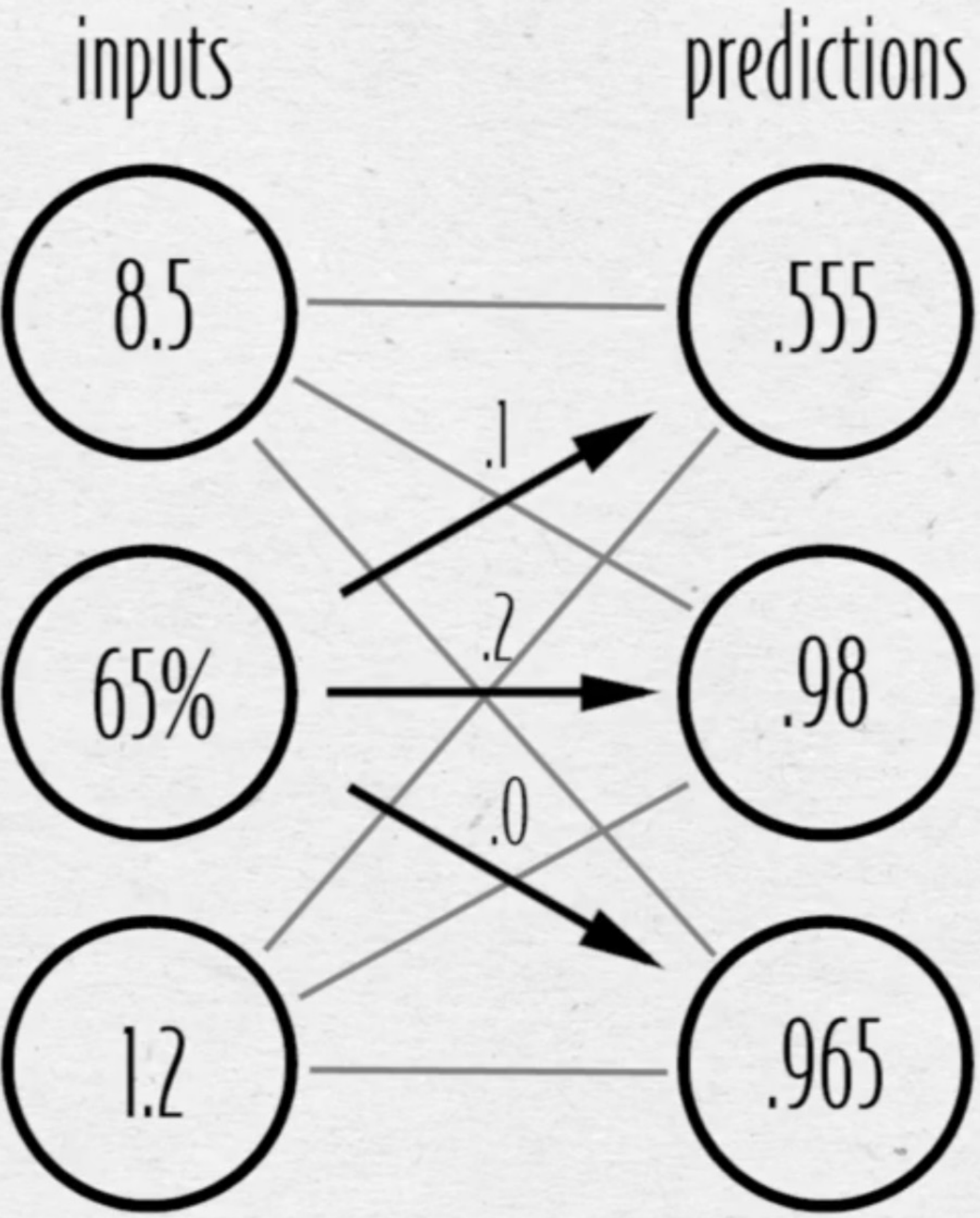
❸ For each output, performing a weighted sum of inputs




```
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

# 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




```
In [ ]: weights = [ [0.1, 0.1, -0.3],
                    [0.1, 0.2, 0.0],
                    [0.0, 1.3, 0.1] ]

        def neural_network(input, weights):
          pred = vect_mat_mul(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]]

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

```python
weights = [ [0.1, 0.1, -0.3],
            [0.1, 0.2, 0.0],
            [0.0, 1.3, 0.1] ]

def neural_network(input, weights):
  pred = vect_mat_mul(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]]

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

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

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

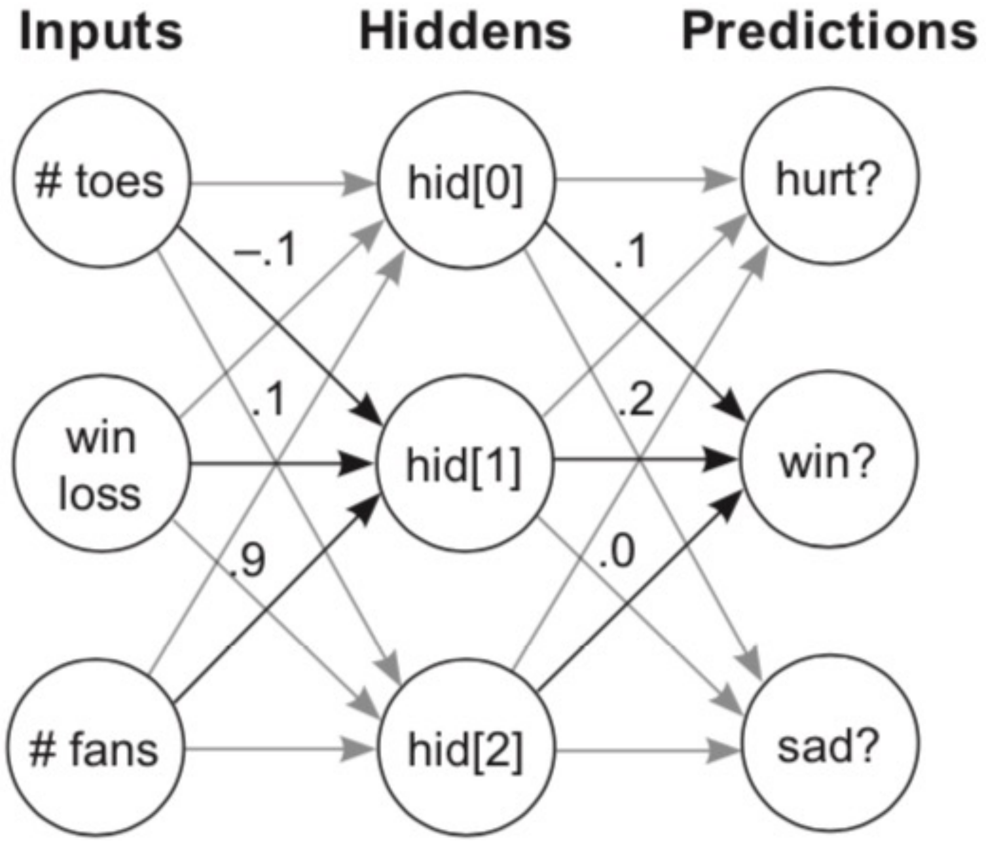
```
[0.555, 0.9800000000000001, 0.9650000000000001]
```

Predicting on predictions

Neural networks can be stacked!

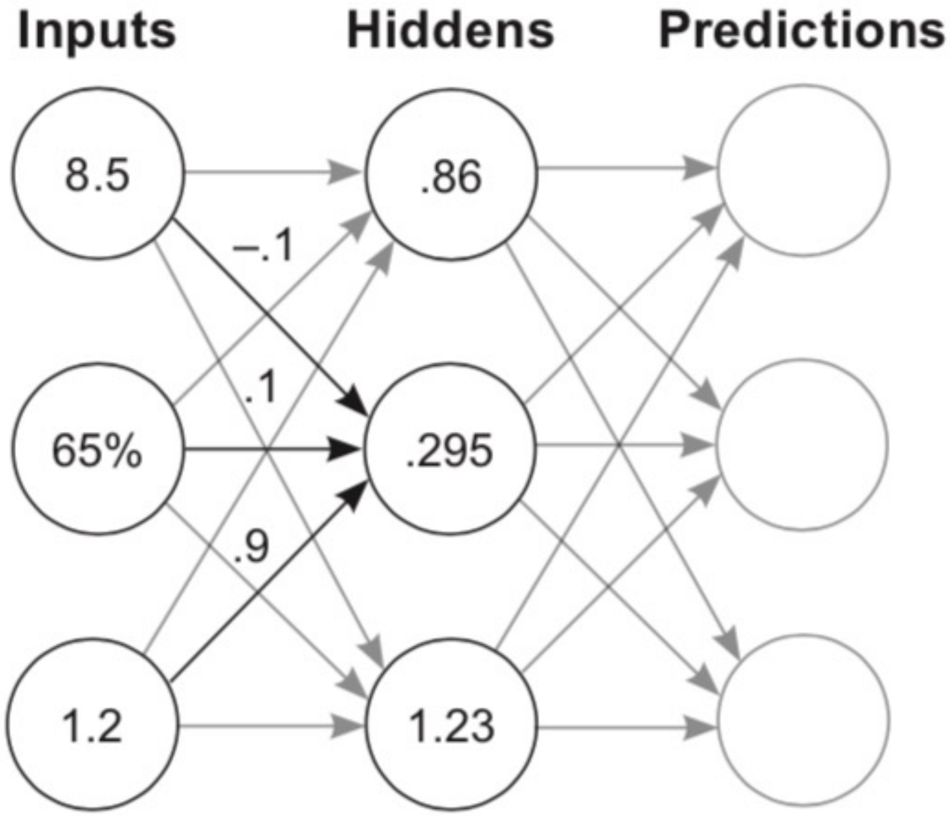
You can also take the output of one network and feed it as input to another network. This results in two consecutive vector-matrix multiplications.

❶ 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
```

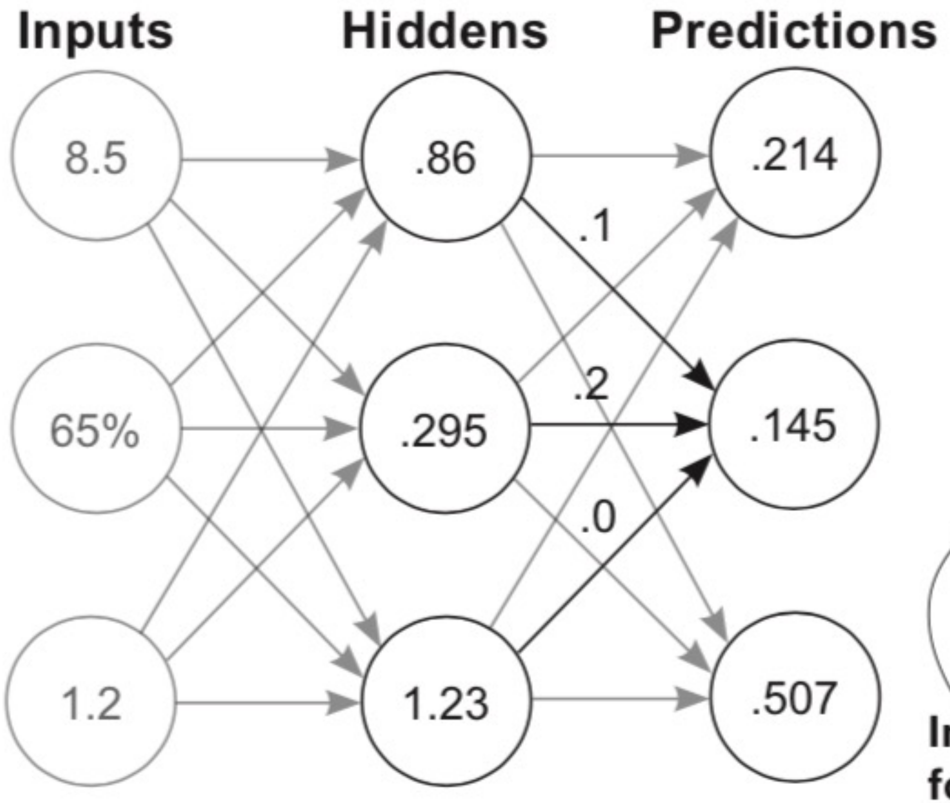
❷ Predicting the hidden layer



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
```

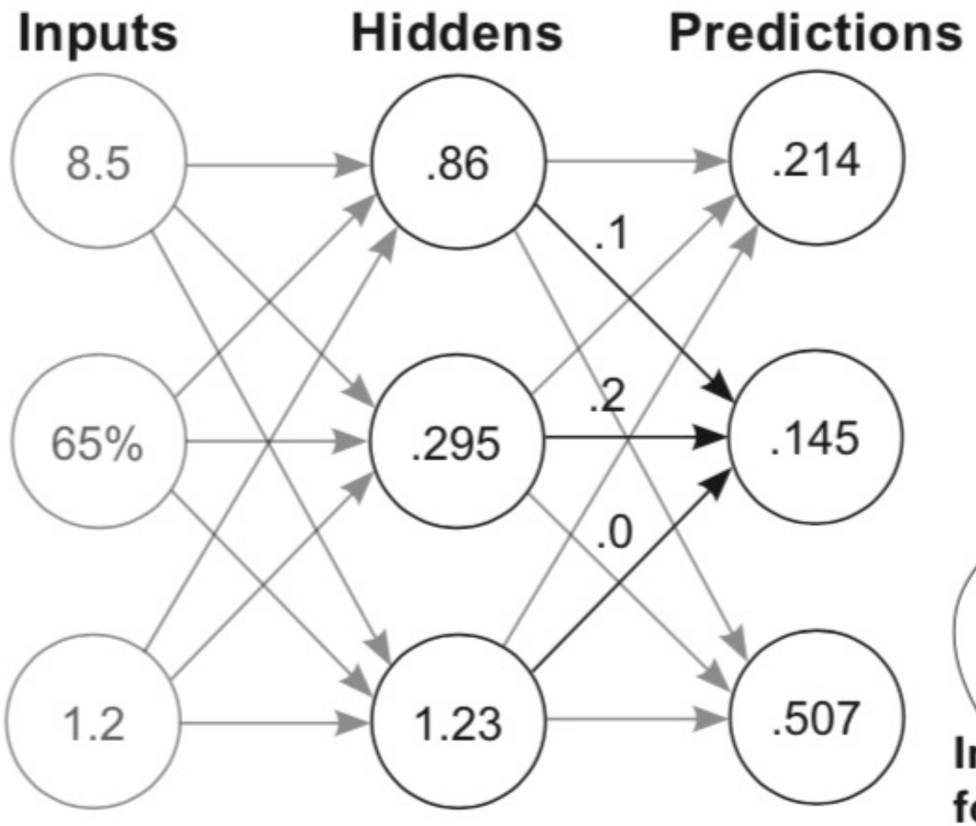
❸ Predicting the output layer (and depositing the prediction)




```
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.

❸ Predicting the output layer (and depositing the prediction)



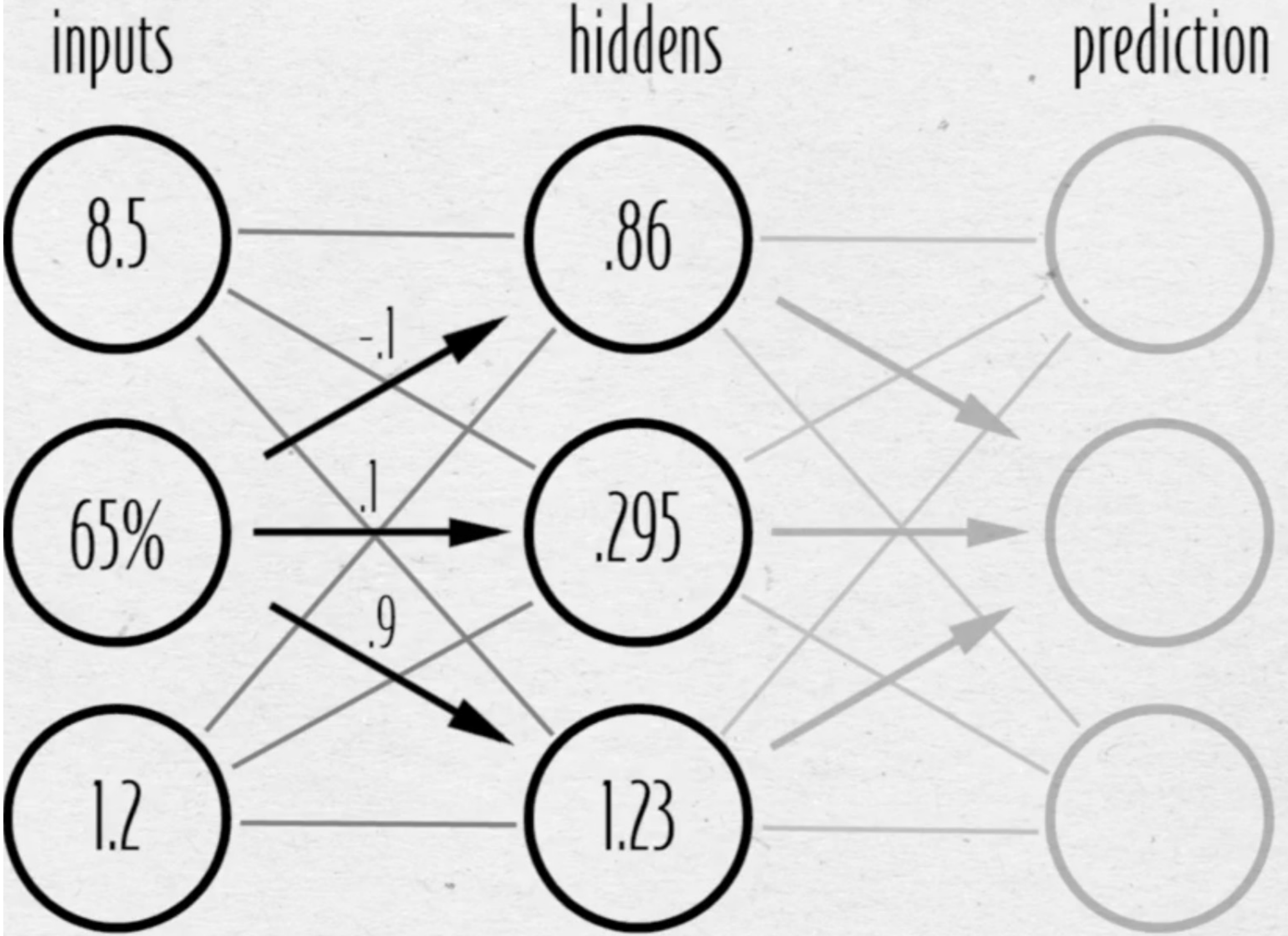

```
def neural_network(input, weights):

    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.

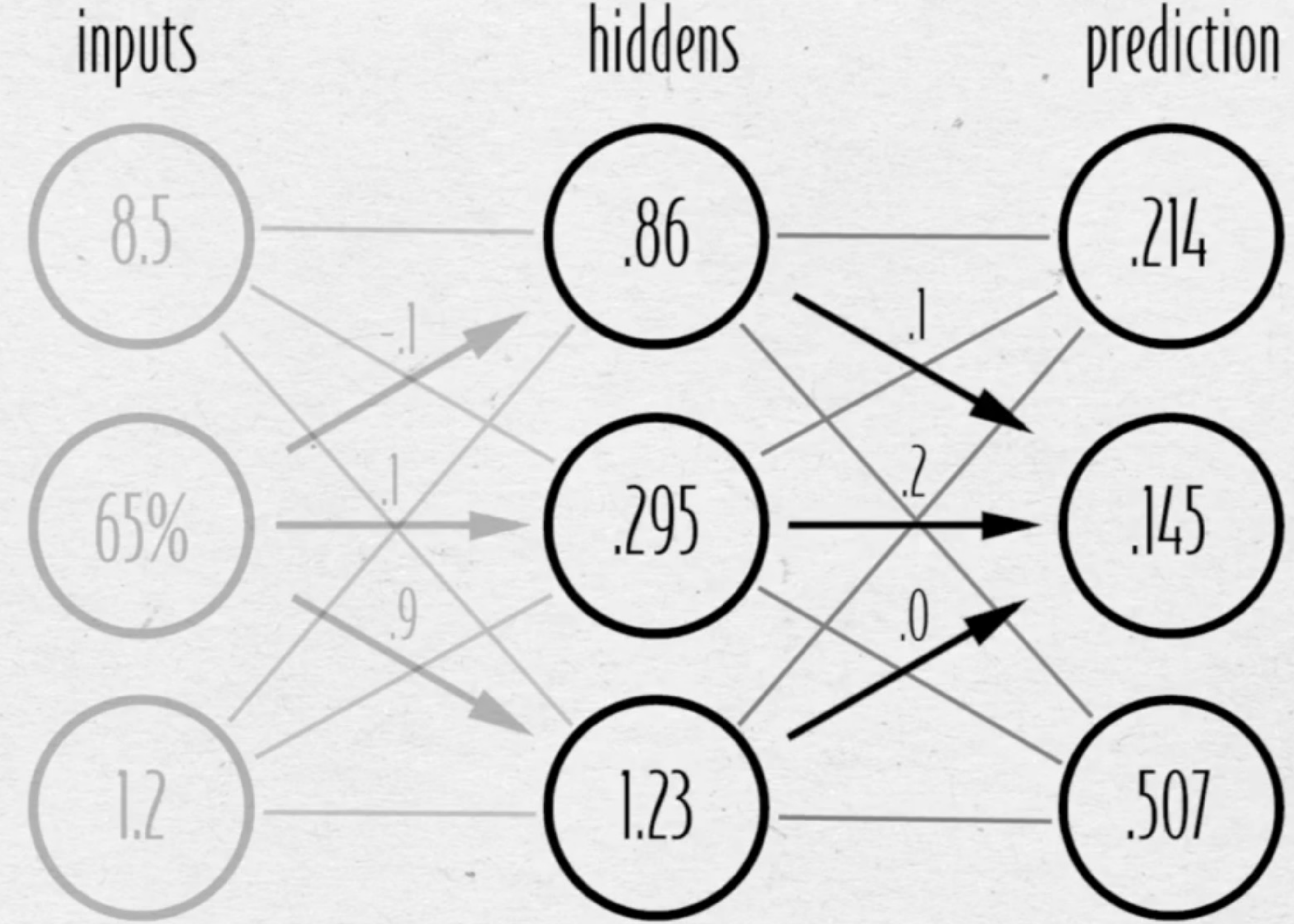
```python
              #toes %win #fans
ih_wgt = [ [0.1, 0.2, -0.1], #hid[0]
           [-0.1,0.1, 0.9], #hid[1]
           [0.1, 0.4, 0.1] ] #hid[2]

          #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

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]
```

```
            #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

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

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

print(pred)
```

```
[0.21350000000000002, 0.14500000000000002, 0.5065]
```



inputs
hiddens
prediction
8.5
65%
1.2
.86
.295
1.23
-.1
.1
.9



inputs
hiddens
prediction
8.5
65%
1.2
-.1
.1
.9
.86
.295
1.23
.1
.2
.0
.214
.145
.507

A quick primer on NumPy

```python
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], # a matrix
              [4,5,6,7]])

d = np.zeros((2,4)) # (2x4 matrix of zeros)
e = np.random.rand(2,5) # random 2x5
# matrix with all numbers between 0 and 1

print(a)
print(b)
print(c)
print(d)
print(e)
```

```
[0 1 2 3]
[4 5 6 7]
[[0 1 2 3]
 [4 5 6 7]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0.21974111 0.10348748 0.7637431  0.08270333 0.91284573]
 [0.40148012 0.06615097 0.79984481 0.90307965 0.3134763 ]]
```

```python
print(a * 0.1) # multiplies every number in vector "a" by 0.1

print(c * 0.2) # multiplies every number in matrix "c" by 0.2

print(a * b) # multiplies elementwise between a and b (columns paired up)

print(a * b * 0.2) # elementwise multiplication then multiplied by 0.2

print(a * c) # since c has the same number of columns as a, this performs
# elementwise multiplication on every row of the matrix "c"

print(a * e) # since a and e don't have the same number of columns, this
# throws a "Value Error: operands could not be broadcast together with.."
```

```
a = np.zeros((1,4)) # vector of length 4
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print(c.shape)
```

```
(1, 3)
```

```python
a = np.zeros((2,4)) # matrix with 2 rows and 4 columns
b = np.zeros((4,3)) # matrix with 4 rows & 3 columns

c = a.dot(b)
print(c.shape) # outputs (2,3)

e = np.zeros((2,1)) # matrix with 2 rows and 1 columns
f = np.zeros((1,3)) # matrix with 1 row & 3 columns

g = e.dot(f)
print(g.shape) # outputs (2,3)

h = np.zeros((5,4)).T # matrix with 4 rows and 5 columns
i = np.zeros((5,6)) # matrix with 6 rows & 5 columns

j = h.dot(i)
print(j.shape) # outputs (4,6)

h = np.zeros((5,4)) # matrix with 5 rows and 4 columns
i = np.zeros((5,6)) # matrix with 5 rows & 6 columns
j = h.dot(i)
print(j.shape) # throws an error
```

```python
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)
```

```
[0.2135 0.145  0.5065]
```

Summary

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