Introduction to Neural Learning

CHAPTER 4

Predict, Compare and Learn

- How do we set weight values so the network predicts accurately?

What is Compare?

- Comparing gives a measurement of how much a prediction “missed” by.

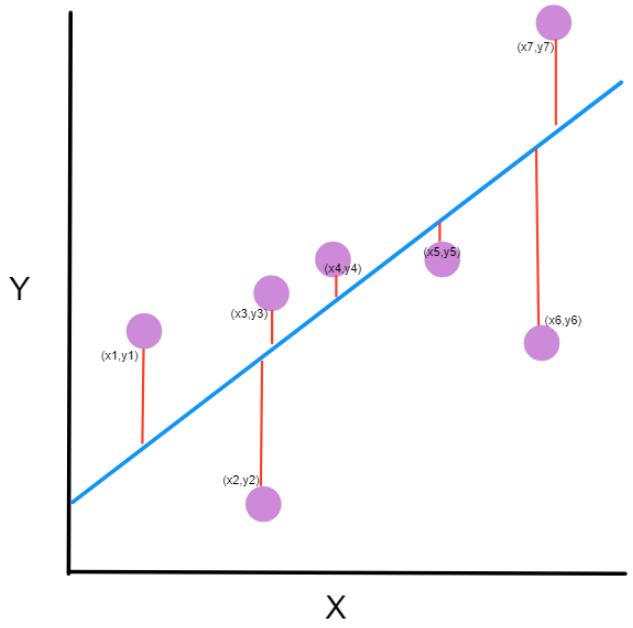
- Once you’ve made a prediction, the next step is to evaluate how well you did.
- However, to measure error is one of the most important and complicated subjects of deep learning.

Properties of Error

- Amplify bigger errors while ignoring very small ones.
- How to mathematically teach a network to do this.
- Error is always positive! - Analogy of an archer hitting a target. Whether the shot is too low by an inch or too high by an inch, the error is still just 1 inch.
- One simple way of measuring error: mean squared error.

Mean Squared Error

- Amplify bigger errors
- Reduce small errors
- 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.
- Given some prediction, it calculate an error measure that says either “a lot” or “a little.”
- It won’t tell you why you missed, what direction you missed, or what you should do to fix the error.
- It more or less says “big miss,” “little miss,” or “perfect prediction.”



$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y_i})^2$$

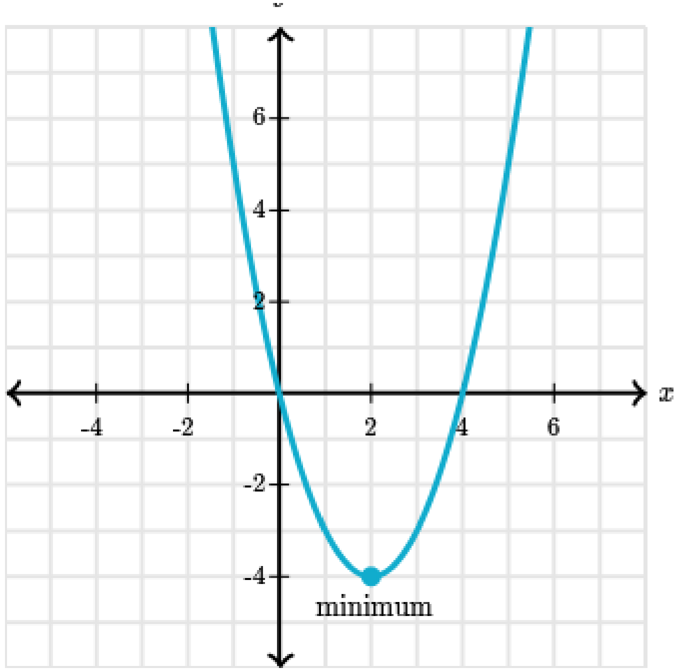
$*\ n$ is the number of data points

$*\ Y_i$ represents observed values

$*\ \hat{Y_i}$ represents predicted values

Learning

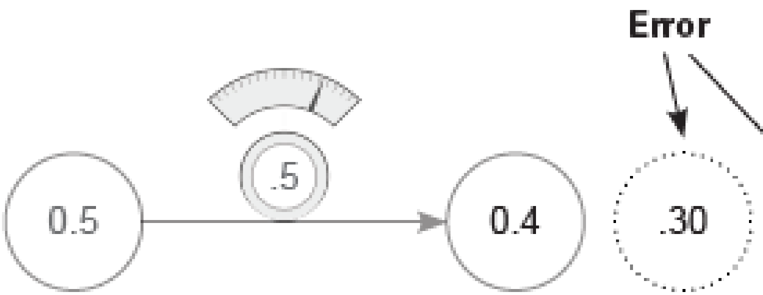
- Learning is all about error attribution, or the art of figuring out how each weight played its part in creating error.
- Algorithm: Gradient descent
- Gradient descent is a general-purpose algorithm that numerically finds minima of multivariable functions.
- It results in computing a number for each weight. That number represents how that weight should be higher or lower in order to reduce the error.
- Then
- you’ll move the weight according to that number, and you’ll be finished.



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`:

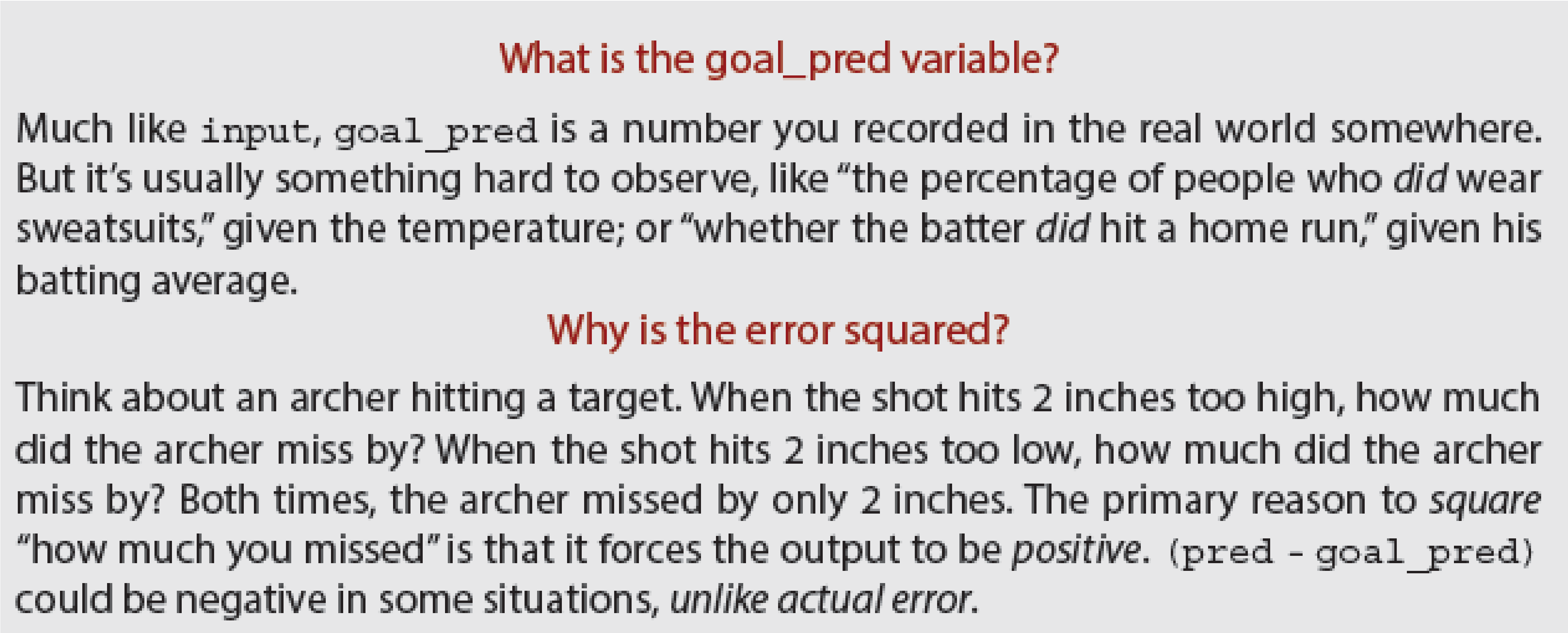


The error is a way to measure how much you missed. There are multiple ways to calculate error, as you'll learn later. This one is *mean squared error*.

```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight

error = (pred - goal_pred) ** 2

print(error)
```

Raw error

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

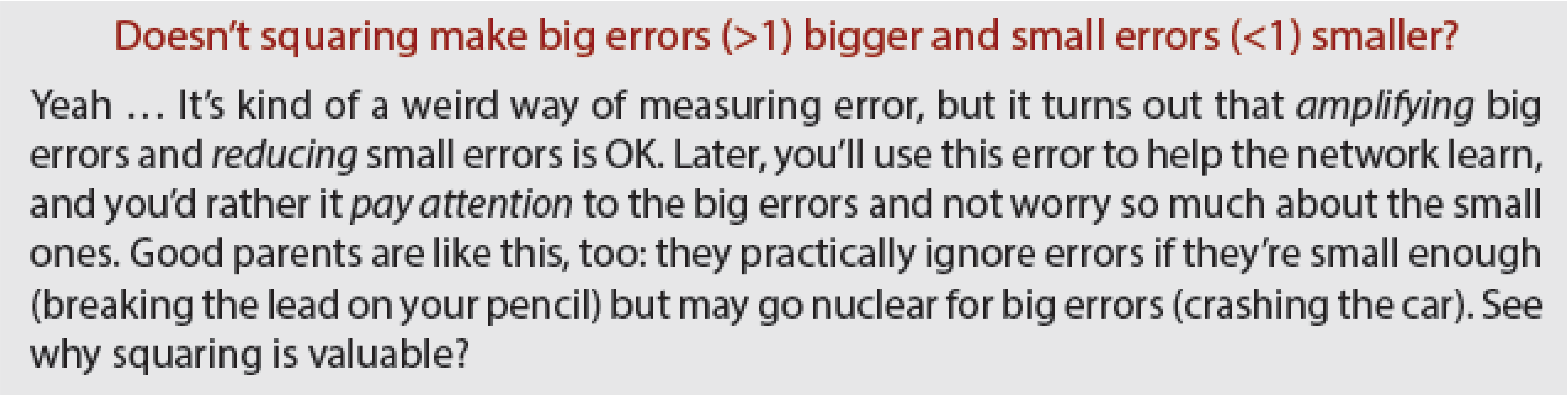


What is the goal_pred variable?

Much like `input`, `goal_pred` is a number you recorded in the real world somewhere. But it's usually something hard to observe, like "the percentage of people who *did* wear sweatsuits," given the temperature; or "whether the batter *did* hit a home run," given his batting average.

Why is the error squared?

Think about an archer hitting a target. When the shot hits 2 inches too high, how much did the archer miss by? When the shot hits 2 inches too low, how much did the archer miss by? Both times, the archer missed by only 2 inches. The primary reason to *square* "how much you missed" is that it forces the output to be *positive*. `(pred - goal_pred)` could be negative in some situations, *unlike actual error*.



Doesn’t squaring make big errors (>1) bigger and small errors (<1) smaller?

Yeah … It’s kind of a weird way of measuring error, but it turns out that *amplifying* big errors and *reducing* small errors is OK. Later, you’ll use this error to help the network learn, and you’d rather it *pay attention* to the big errors and not worry so much about the small ones. Good parents are like this, too: they practically ignore errors if they’re small enough (breaking the lead on your pencil) but may go nuclear for big errors (crashing the car). See why squaring is valuable?

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 chapter), 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.

It turns out that changing `knob_weight` to make the network correctly predict `goal_prediction` is *slightly* more complicated than changing `knob_weight` to make `error == 0`. There's something more concise about looking at the problem this way. Ultimately, both statements say the same thing, but trying to *get the error to 0* seems more straightforward.

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* (`pred - goal_pred`) so that bigger errors become *very* big and smaller errors quickly become irrelevant.

By measuring error this way, you can *prioritize* big errors over smaller ones. When you have somewhat large pure errors (say, 10), you'll tell yourself that you have *very* large error (10**2 == 100); and in contrast, when you have small pure errors (say, 0.01), you'll tell yourself that you have *very* small error (0.01**2 == 0.0001). See what I mean about prioritizing? It's just modifying what you *consider to be error* so that you amplify big ones and largely ignore small ones.

In contrast, if you took the *absolute value* instead of squaring the error, you wouldn't have this type of prioritization. The error would just be the positive version of the pure error—which would be fine, but different. More on this later.

Why do you want only *positive* 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. If the first had an error of 1,000 and the second had an error of –1,000, then the *average error* would be *zero*! You'd fool yourself into thinking you predicted perfectly, when you missed by 1,000 each time! That would be really bad. Thus, you want the error of *each prediction* to always be *positive* so they don't accidentally cancel each other out when you average them.

What's the simplest form of neural learning?

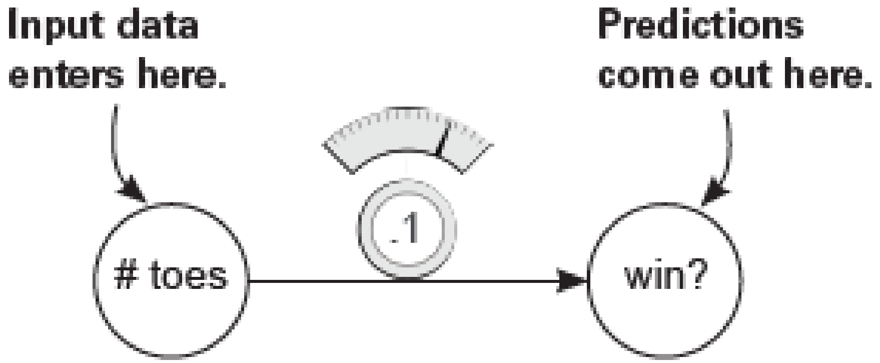
Learning using the hot and cold method.

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`. It's simple but effective. After you do this over and over again, eventually error == 0, which means the neural network is predicting with perfect accuracy.

Hot and cold learning

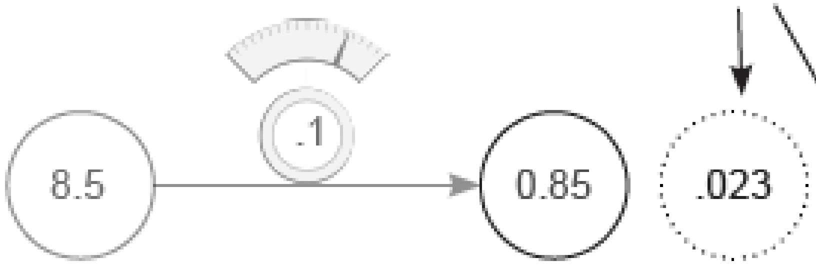
Hot and cold learning means wiggling the weights to see which direction reduces the error the most, moving the weights in that direction, and repeating until the error gets to 0.

❶ An empty network



```
weight = 0.1

lr = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

❷ PREDICT: Making a prediction and evaluating error



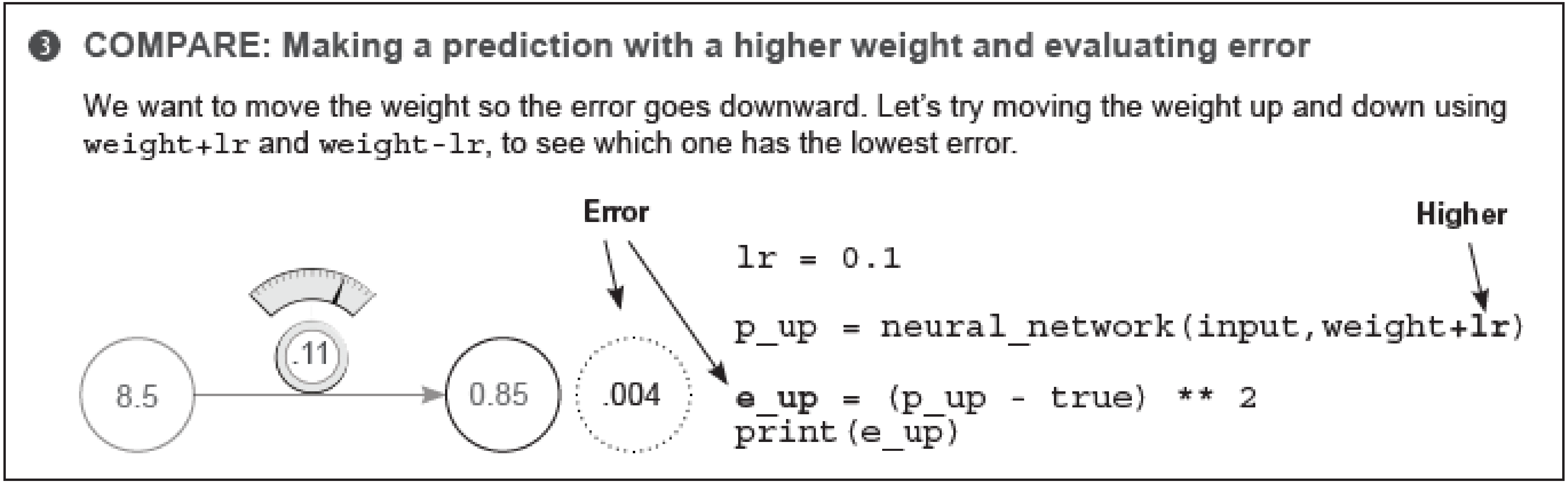
Error

```
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)
```

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*.

Raw error

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

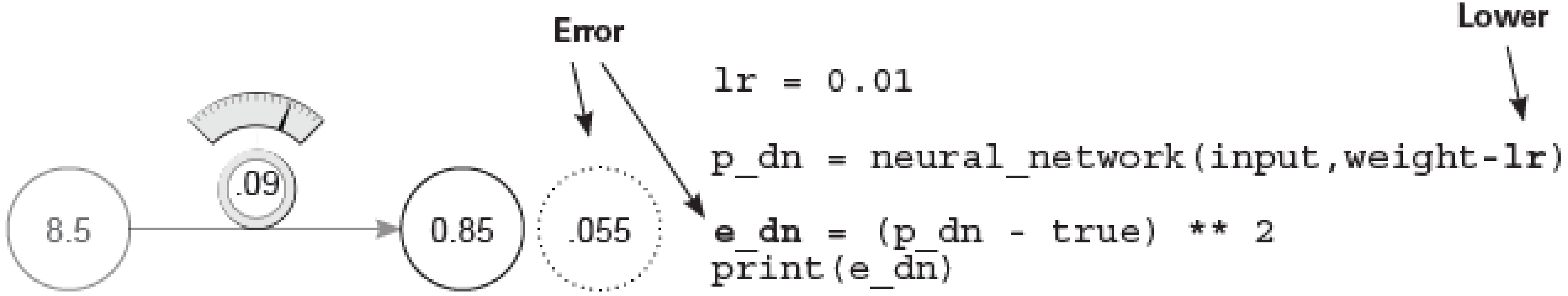


❸ COMPARE: Making a prediction with a higher weight and evaluating error

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.

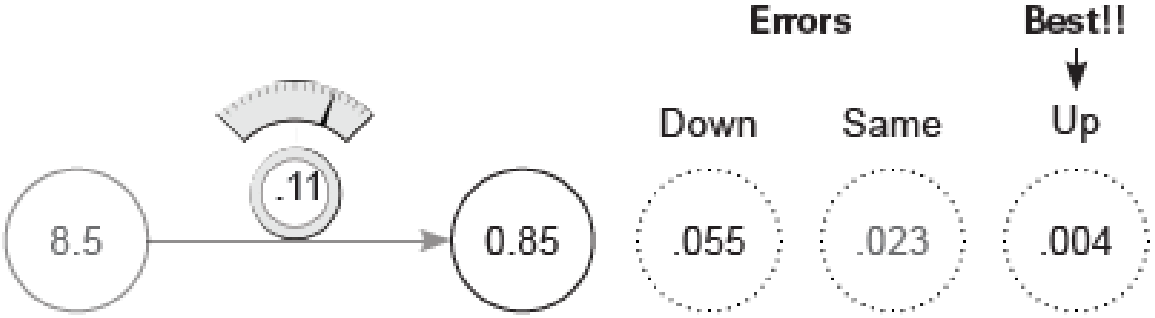
```
lr = 0.1

p_up = neural_network(input,weight+lr)

e_up = (p_up - true) ** 2
print(e_up)
```

❹ 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)
```

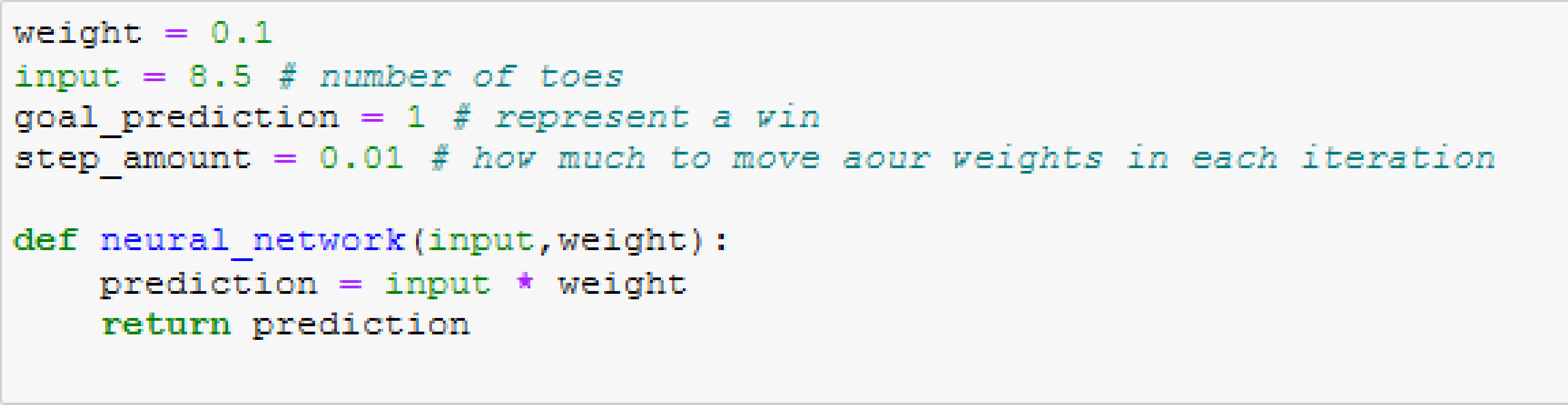
❺ COMPARE + LEARN: Comparing the errors and setting the new weight



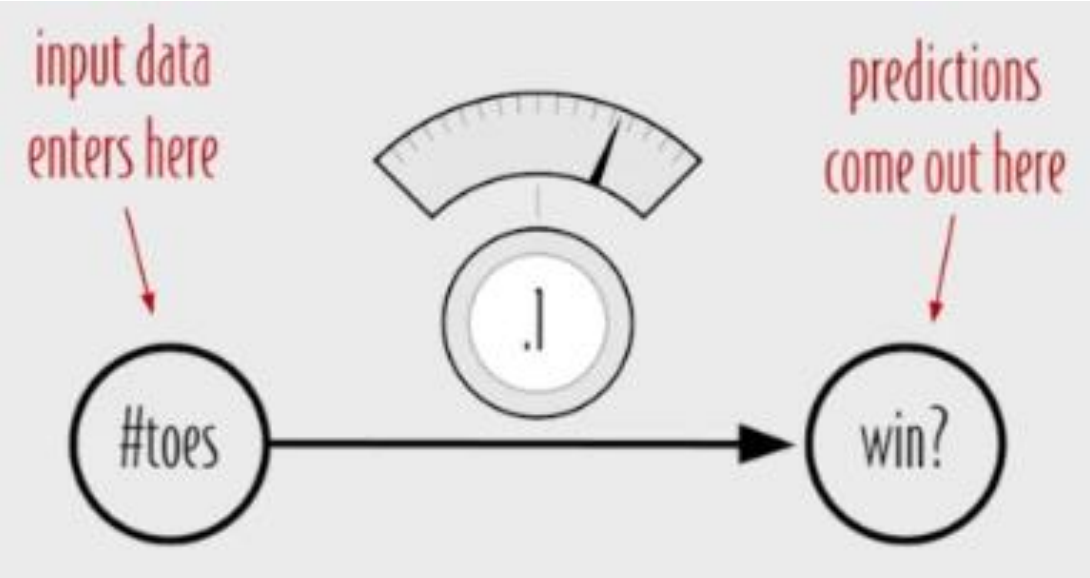
```
if(error > e_dn ||
   error > e_up):

  if(e_dn < e_up):
     weight -= lr

  if(e_up < e_up):
     weight += lr
```

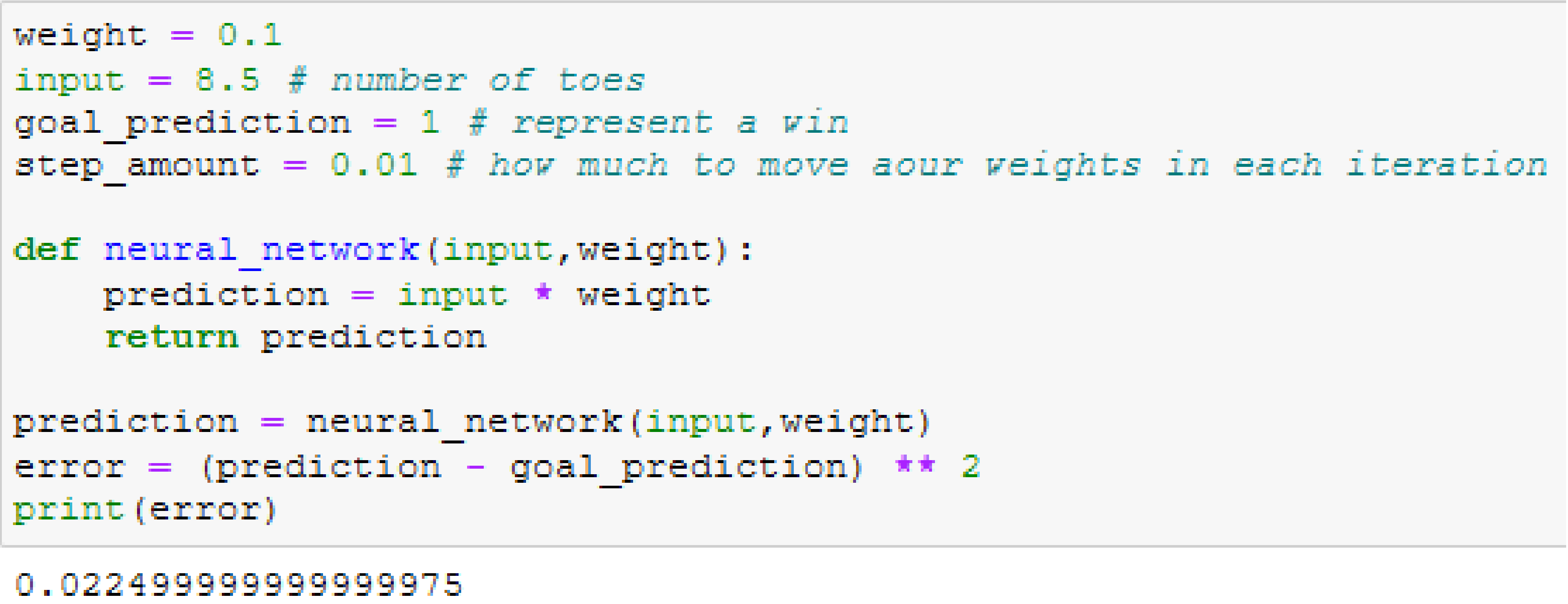
These last five steps are one iteration of hot and cold learning. Fortunately, this iteration got us pretty close to the correct answer all by itself (the new error is only 0.004). But under normal circumstances, we'd have to repeat this process many times to find the correct weights. Some people have to train their networks for weeks or months before they find a good enough weight configuration.

This reveals what learning in neural networks really is: a *search problem*. You're *searching* for the best possible configuration of weights so the network's error falls to 0 (and predicts perfectly). As with all other forms of search, you might not find exactly what you're looking for, and even if you do, it may take some time. Next, we'll use hot and cold learning for a slightly more difficult prediction so you can see this searching in action!



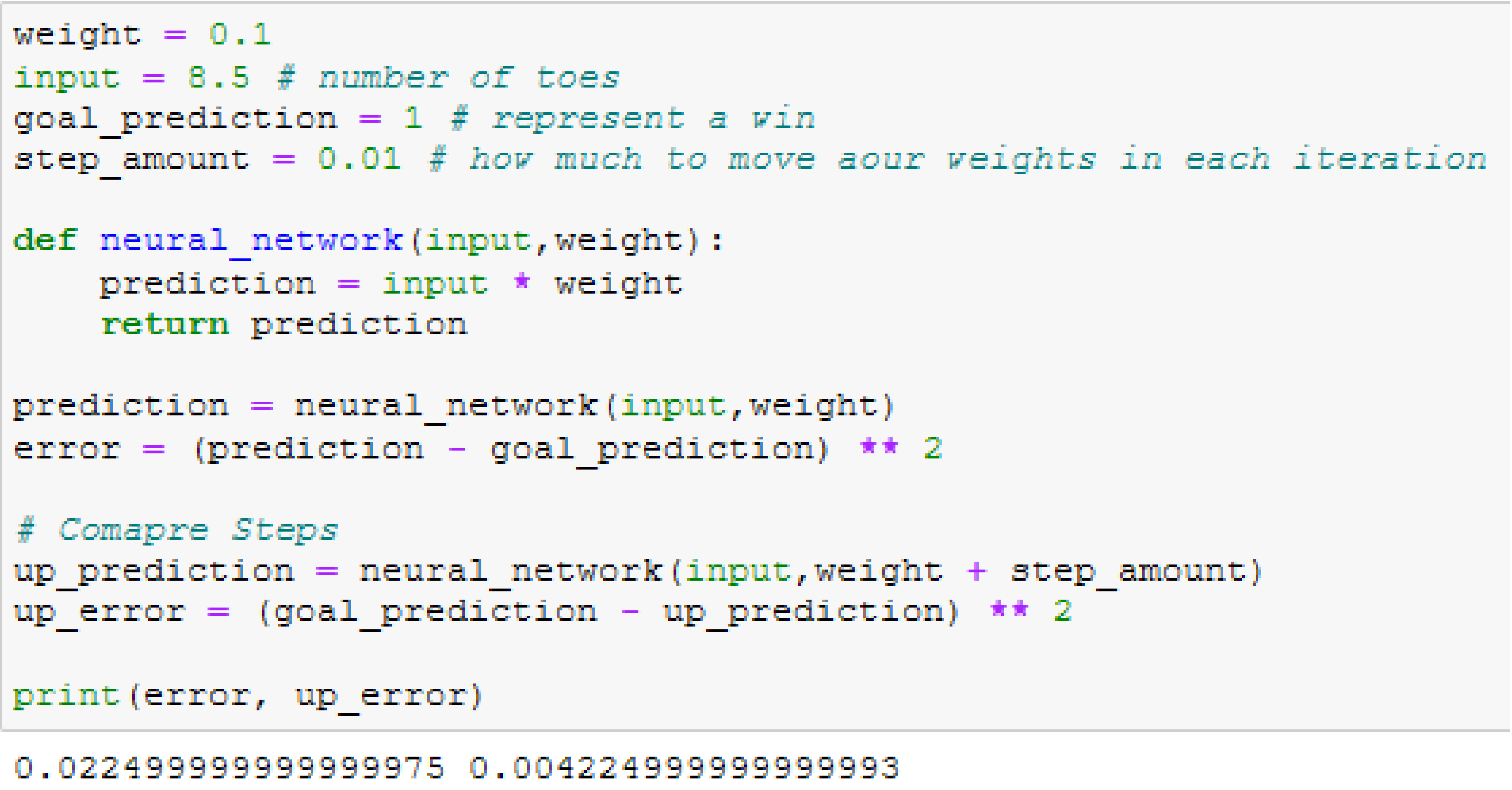
```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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





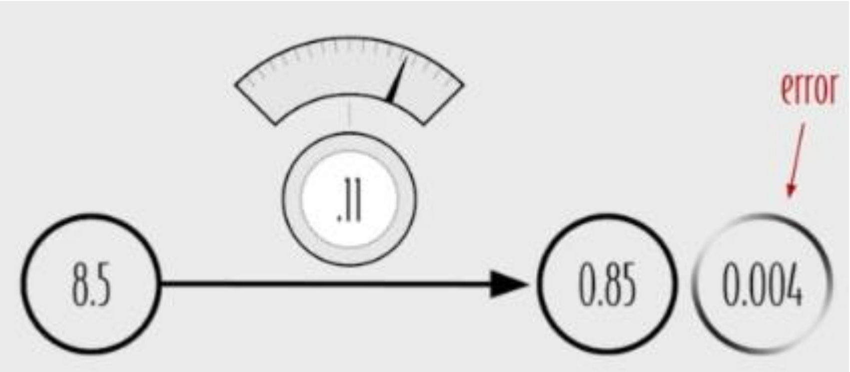
```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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

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

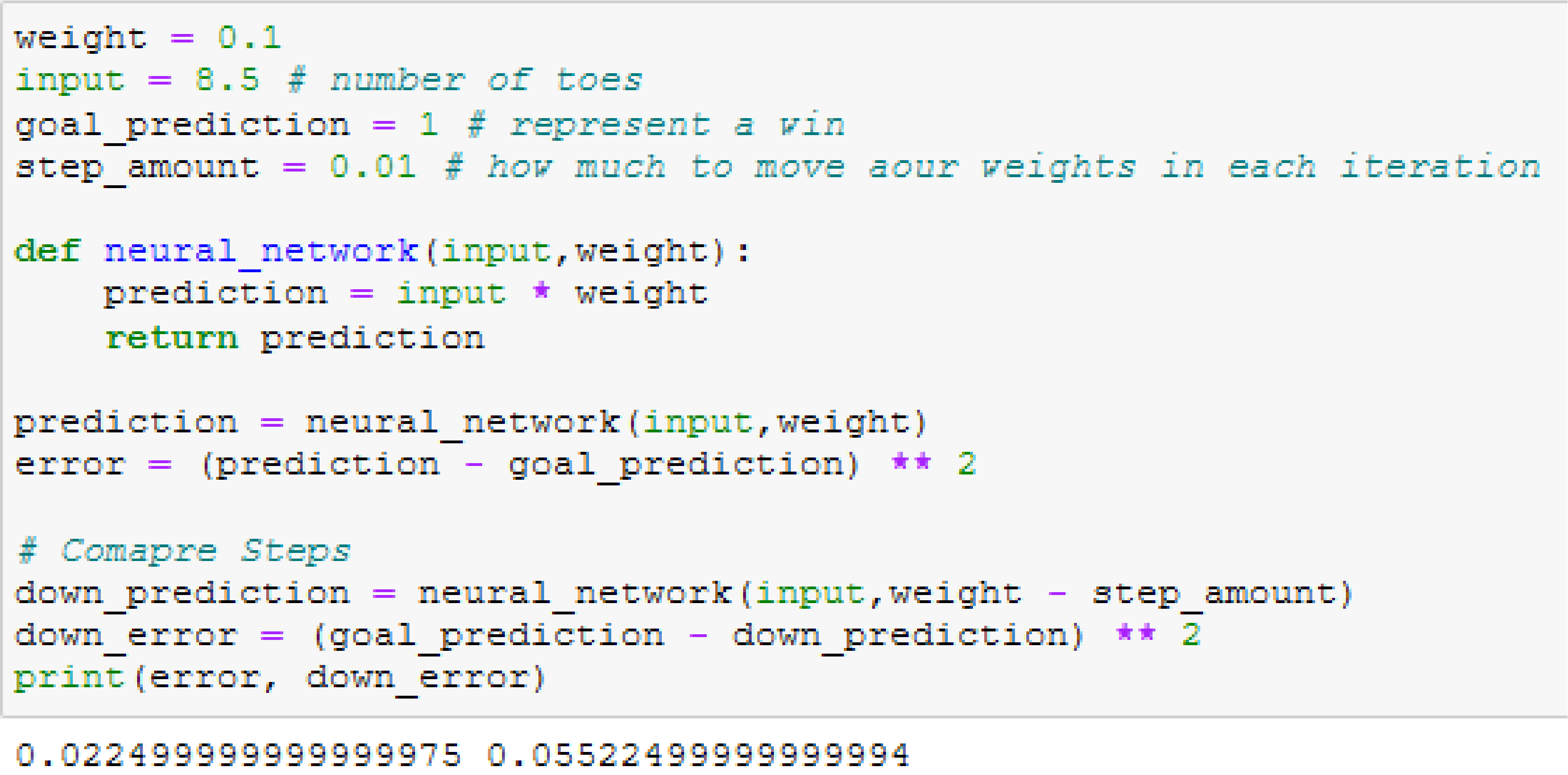
```
0.022499999999999975
```



```python
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

print(error, up_error)
```

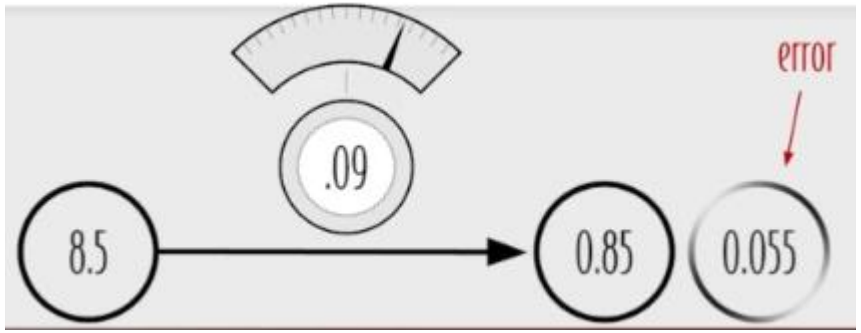
```
0.022499999999999975 0.004224999999999993
```

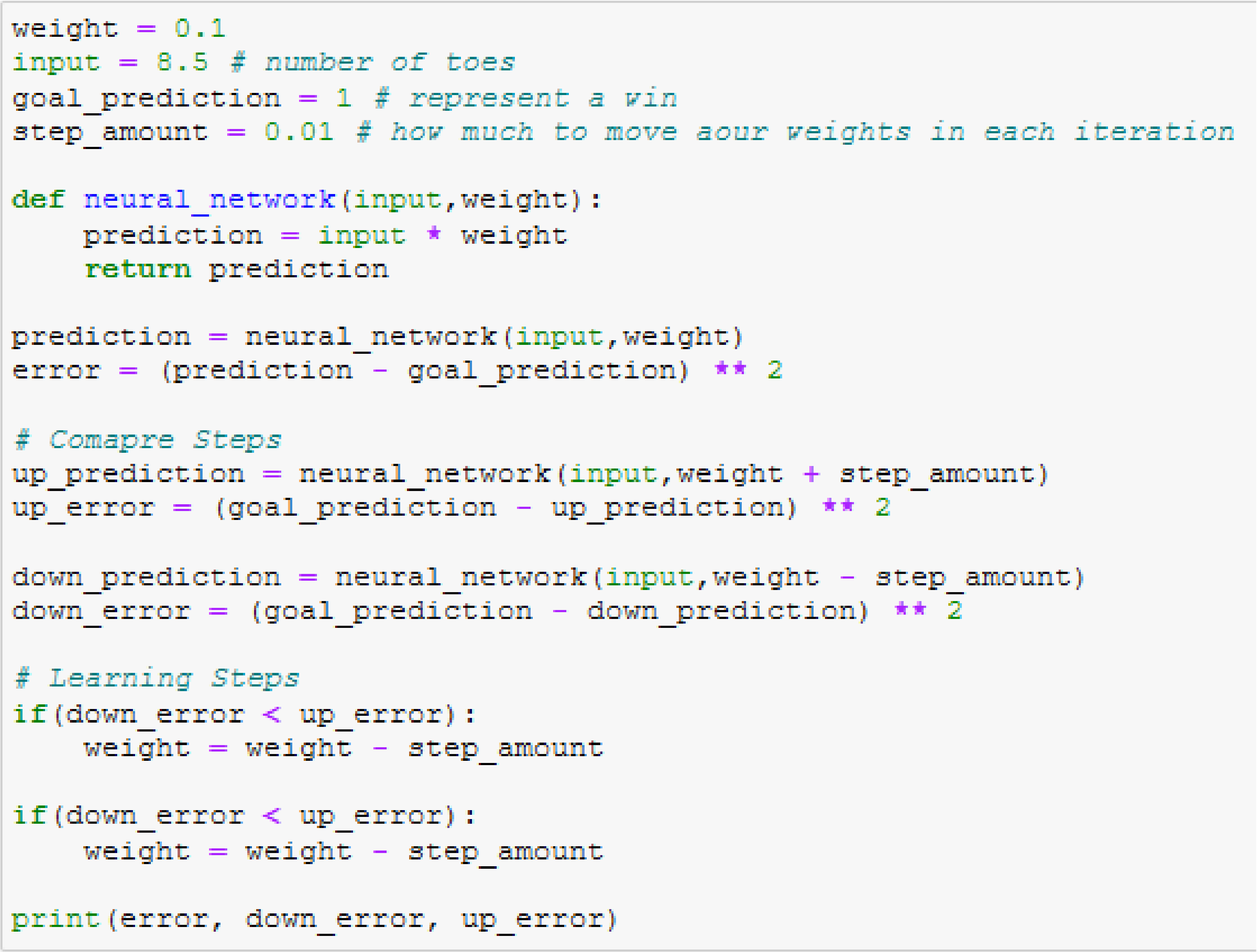




```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2
print(error, down_error)
```

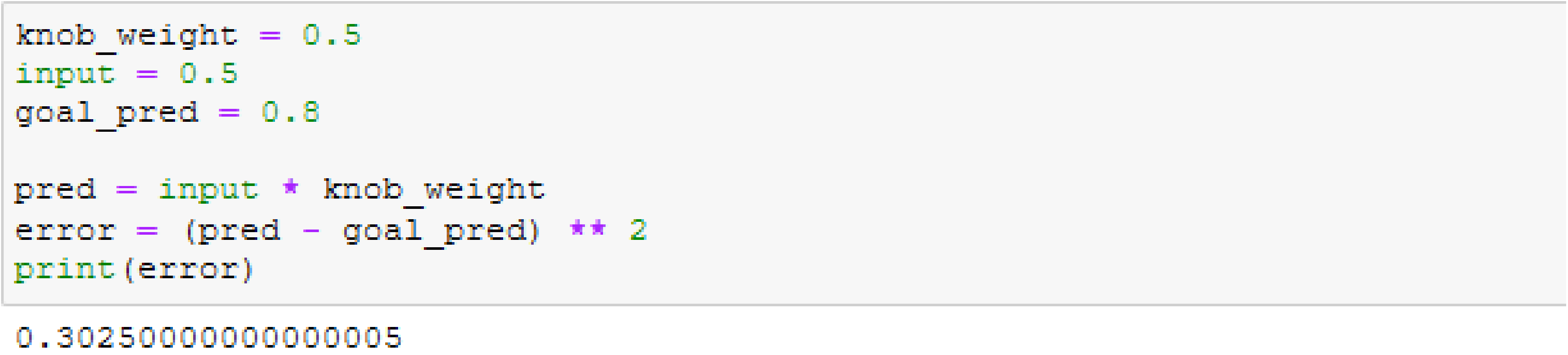
```
0.022499999999999975 0.05522499999999994
```





```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

# Learning Steps
if(down_error < up_error):
    weight = weight - step_amount

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

print(error, down_error, up_error)
```

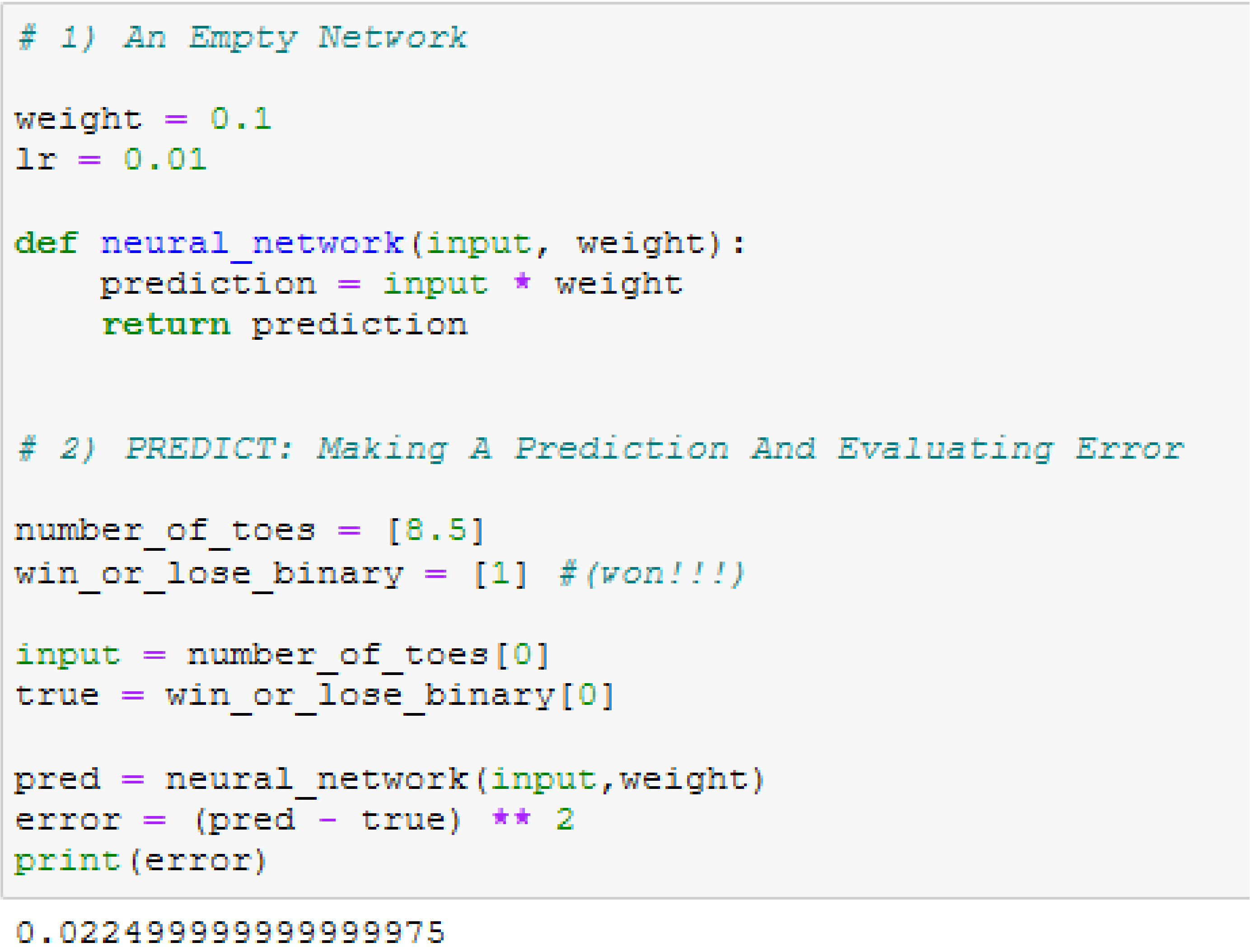
Compare: Does our network make good predictions?



```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight
error = (pred - goal_pred) ** 2
print(error)
```

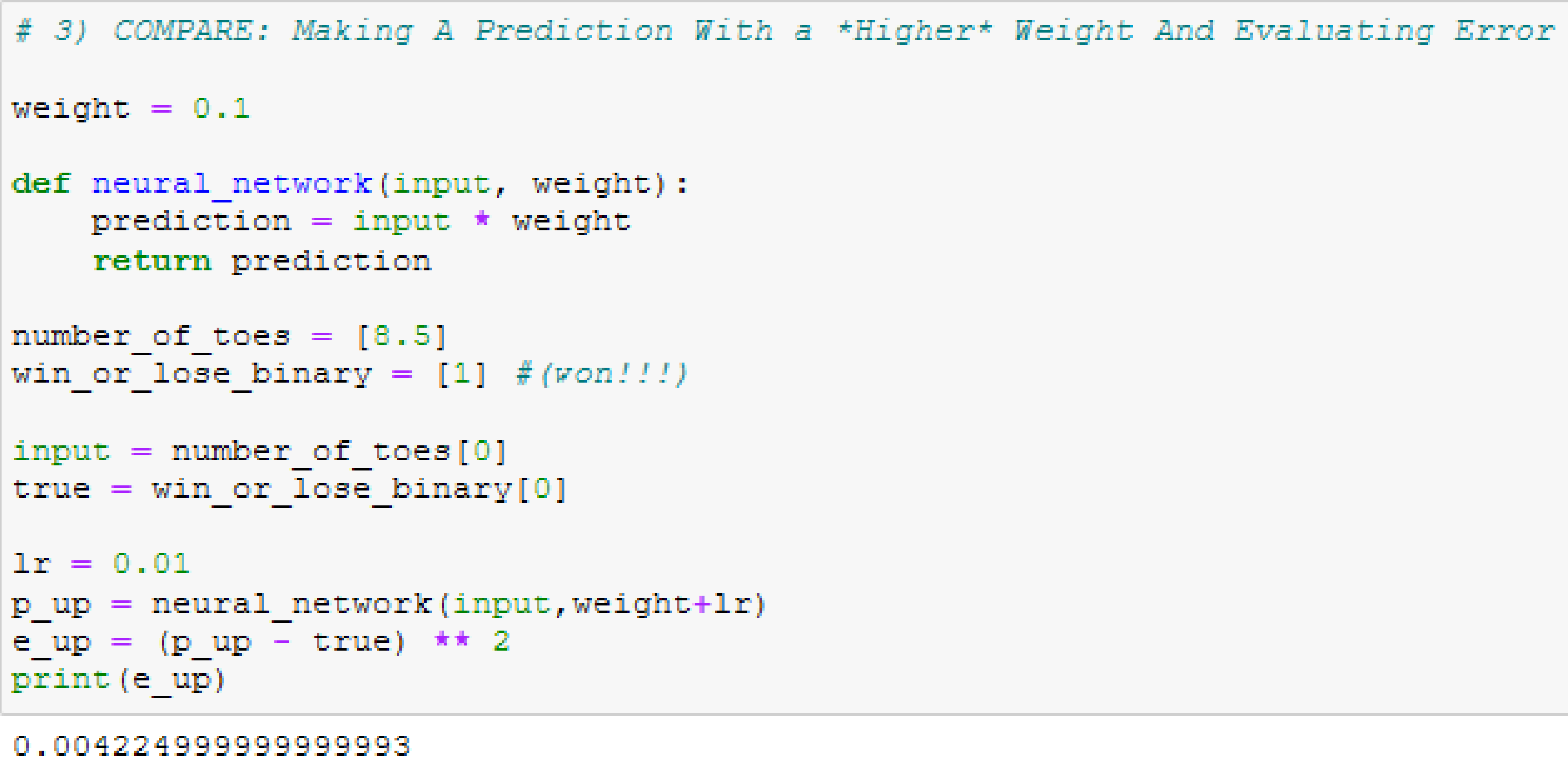
```
0.30250000000000005
```

Learning using the Hot and Cold Method



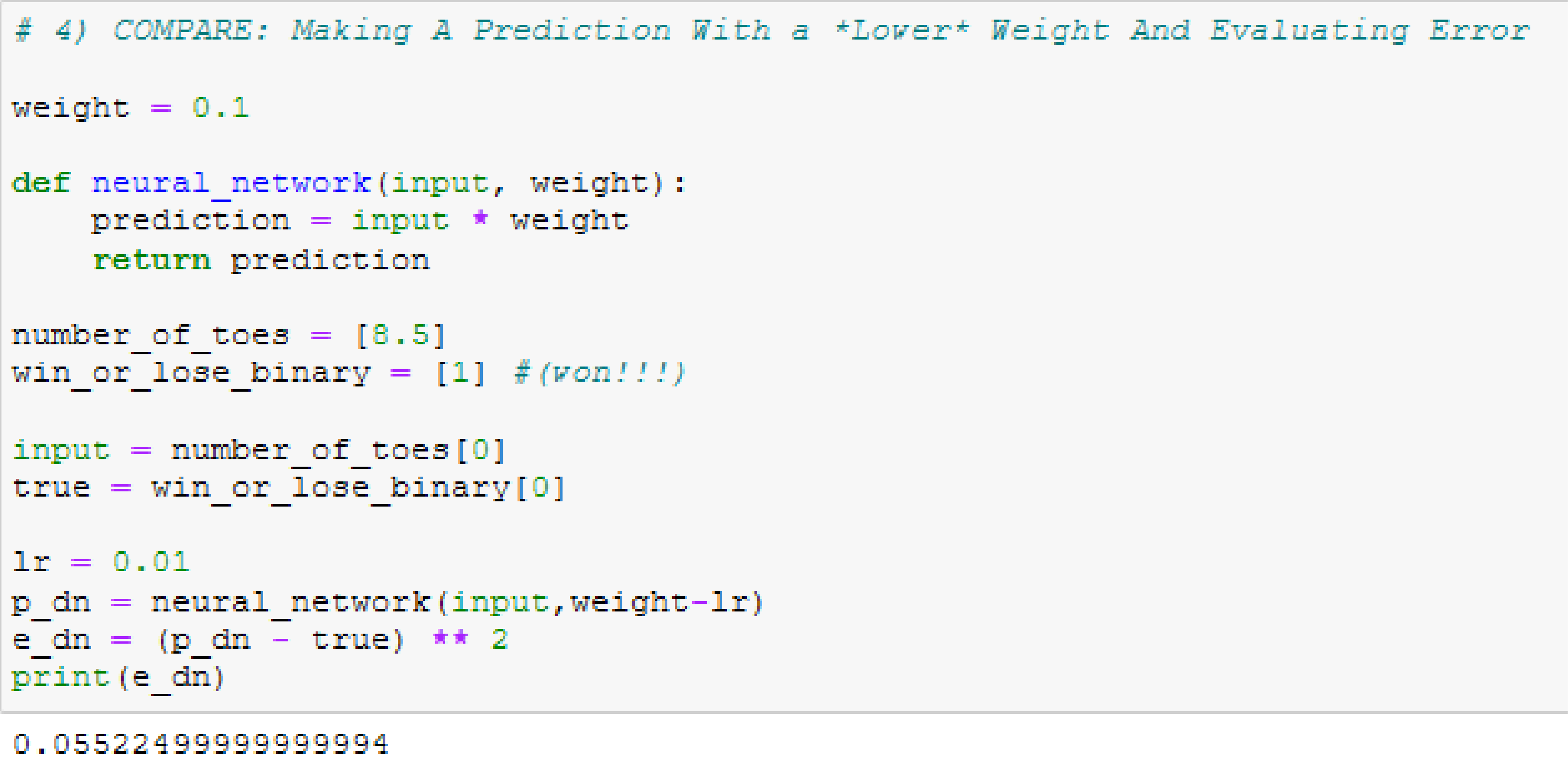
```
# 1) An Empty Network

weight = 0.1
lr = 0.01

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


# 2) PREDICT: Making A Prediction And Evaluating 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)
```

```
0.022499999999999975
```



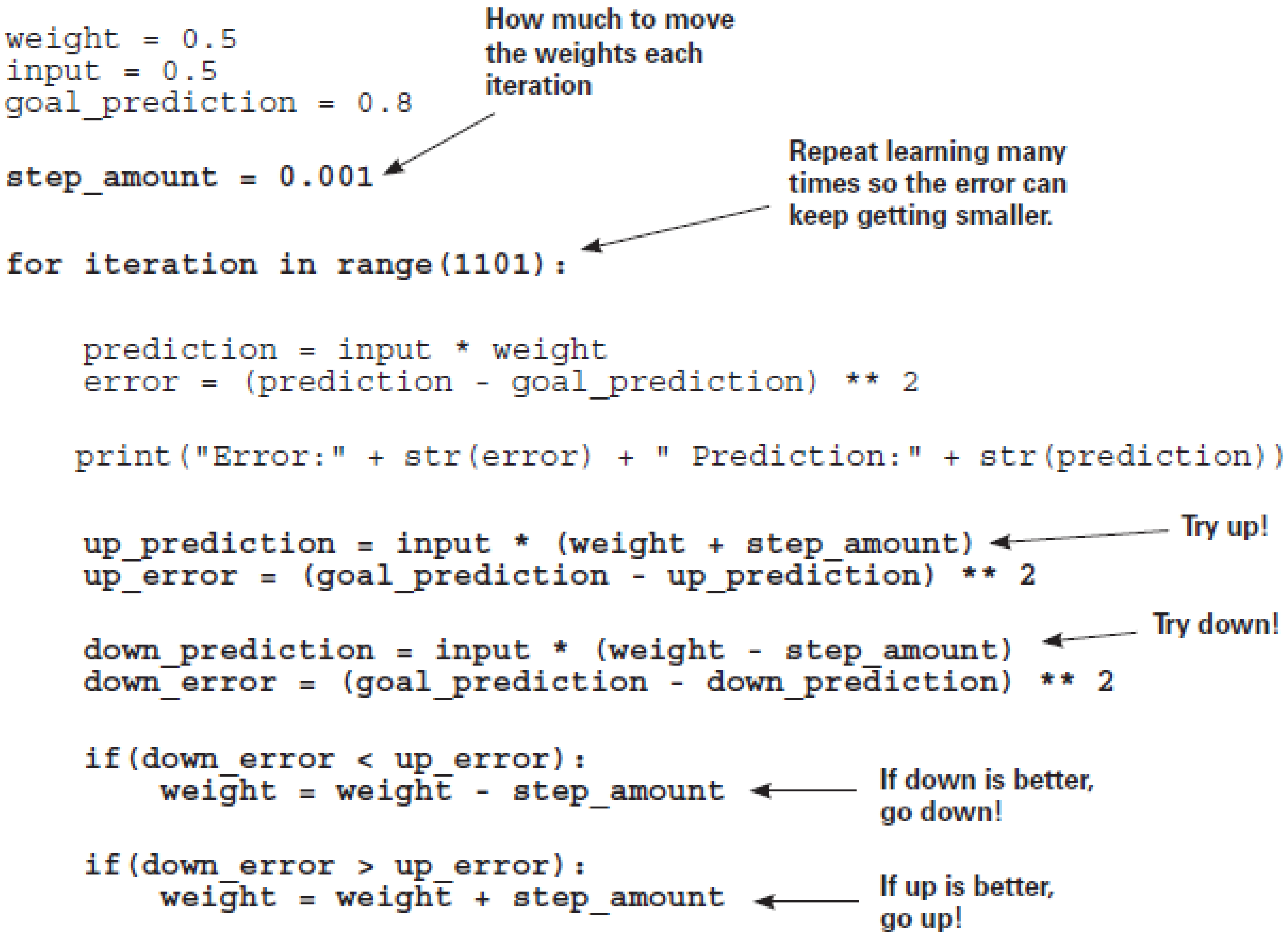
```
# 3) COMPARE: Making A Prediction With a *Higher* Weight And Evaluating Error

weight = 0.1

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

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

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

lr = 0.01
p_up = neural_network(input,weight+lr)
e_up = (p_up - true) ** 2
print(e_up)
```

```
0.004224999999999993
```



```python
# 4) COMPARE: Making A Prediction With a *Lower* Weight And Evaluating Error

weight = 0.1

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

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

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

lr = 0.01
p_dn = neural_network(input,weight-lr)
e_dn = (p_dn - true) ** 2
print(e_dn)
```

```
0.05522499999999994
```



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

step_amount = 0.001

for iteration in range(1101):

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

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

    up_prediction = input * (weight + step_amount)
    up_error = (goal_prediction - up_prediction) ** 2

    down_prediction = input * (weight - step_amount)
    down_error = (goal_prediction - down_prediction) ** 2

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

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

How much to move the weights each iteration

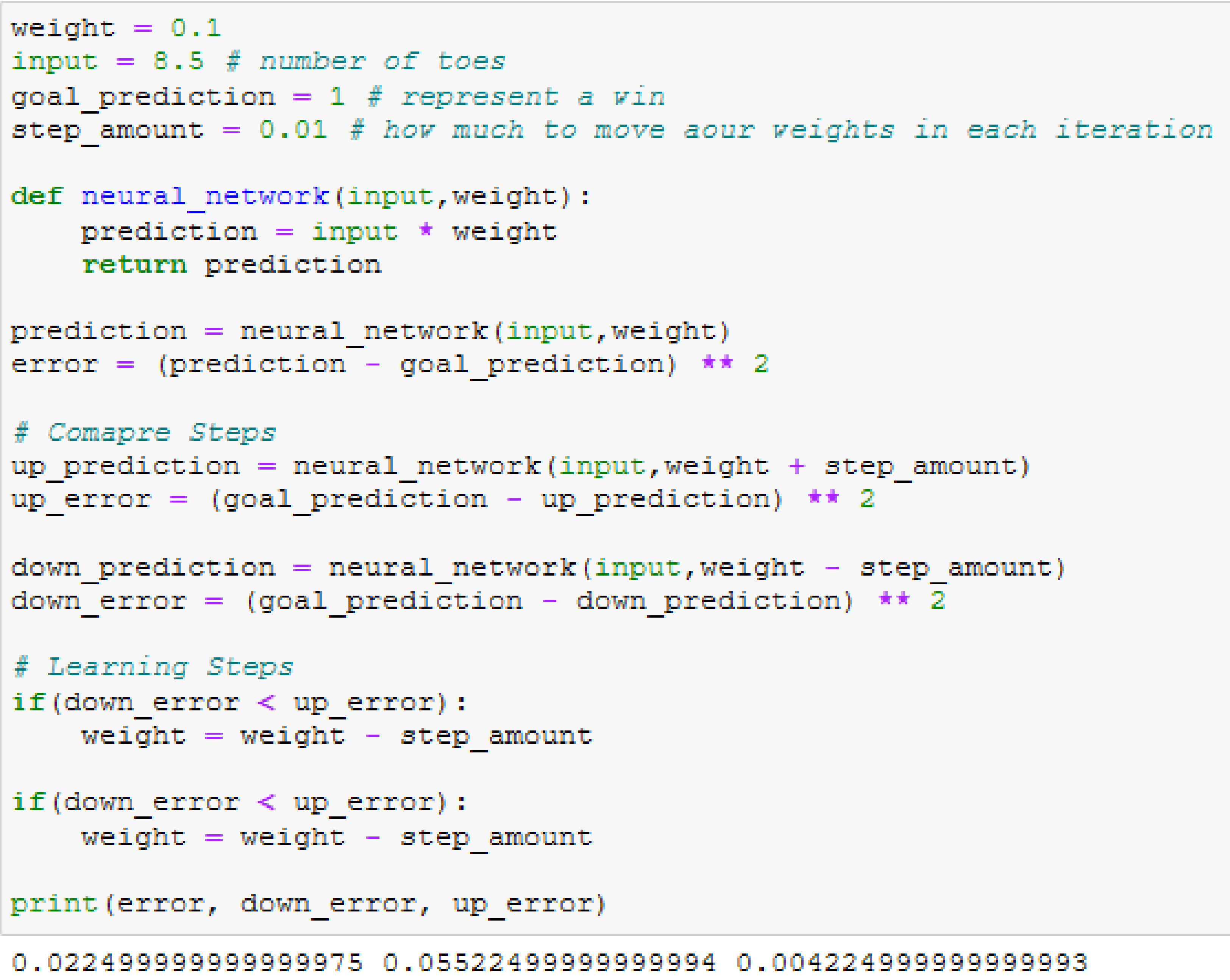
Repeat learning many times so the error can keep getting smaller.

Try up!

Try down!

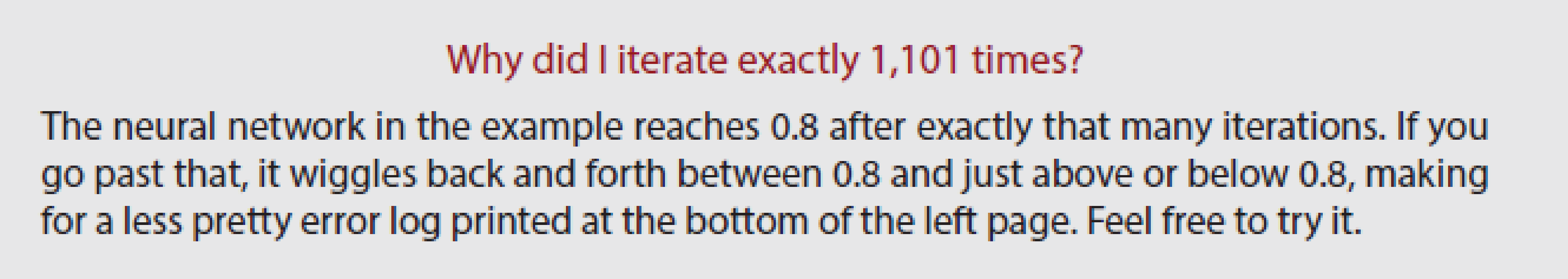
If down is better, go down!

If up is better, go up!



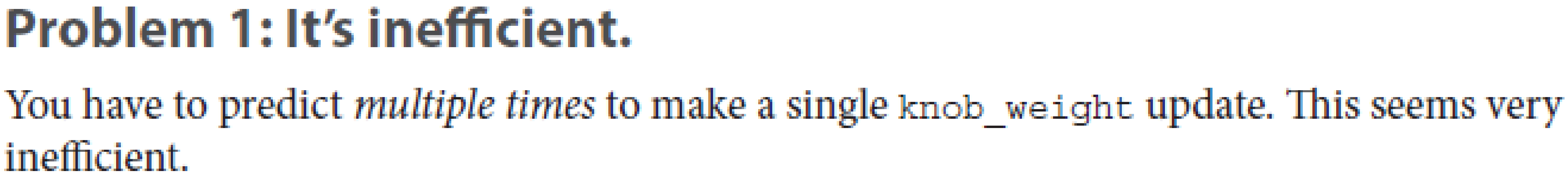
```
weight = 0.1
input = 8.5 # number of toes
goal_prediction = 1 # represent a win
step_amount = 0.01 # how much to move aour weights in each iteration

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

prediction = neural_network(input,weight)
error = (prediction - goal_prediction) ** 2

# Comapre Steps
up_prediction = neural_network(input,weight + step_amount)
up_error = (goal_prediction - up_prediction) ** 2

down_prediction = neural_network(input,weight - step_amount)
down_error = (goal_prediction - down_prediction) ** 2

# Learning Steps
if(down_error < up_error):
    weight = weight - step_amount

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

print(error, down_error, up_error)
```

```
0.022499999999999975 0.05522499999999994 0.004224999999999993
```



Why did I iterate exactly 1,101 times?

The neural network in the example reaches 0.8 after exactly that many iterations. If you go past that, it wiggles back and forth between 0.8 and just above or below 0.8, making for a less pretty error log printed at the bottom of the left page. Feel free to try it.



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*step_amount` away, the network will eventually overshoot by some number less than `step_amount`. When it does, it will then start alternating back and forth between each side of `goal_prediction`. Set `step_amount` to 0.2 to see this in action. 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...
```

The real problem is that even though you know the correct *direction* to move `weight`, you don't know the correct *amount*. Instead, you pick a fixed one at random (`step_amount`). Furthermore, this amount has *nothing* to do with `error`. Whether `error` is big or tiny, `step_amount` is the same. So, hot and cold learning is kind of a bummer. It's inefficient because you predict three times for each `weight` update, and `step_ amount` is arbitrary, which can prevent you from learning the correct `weight` value.