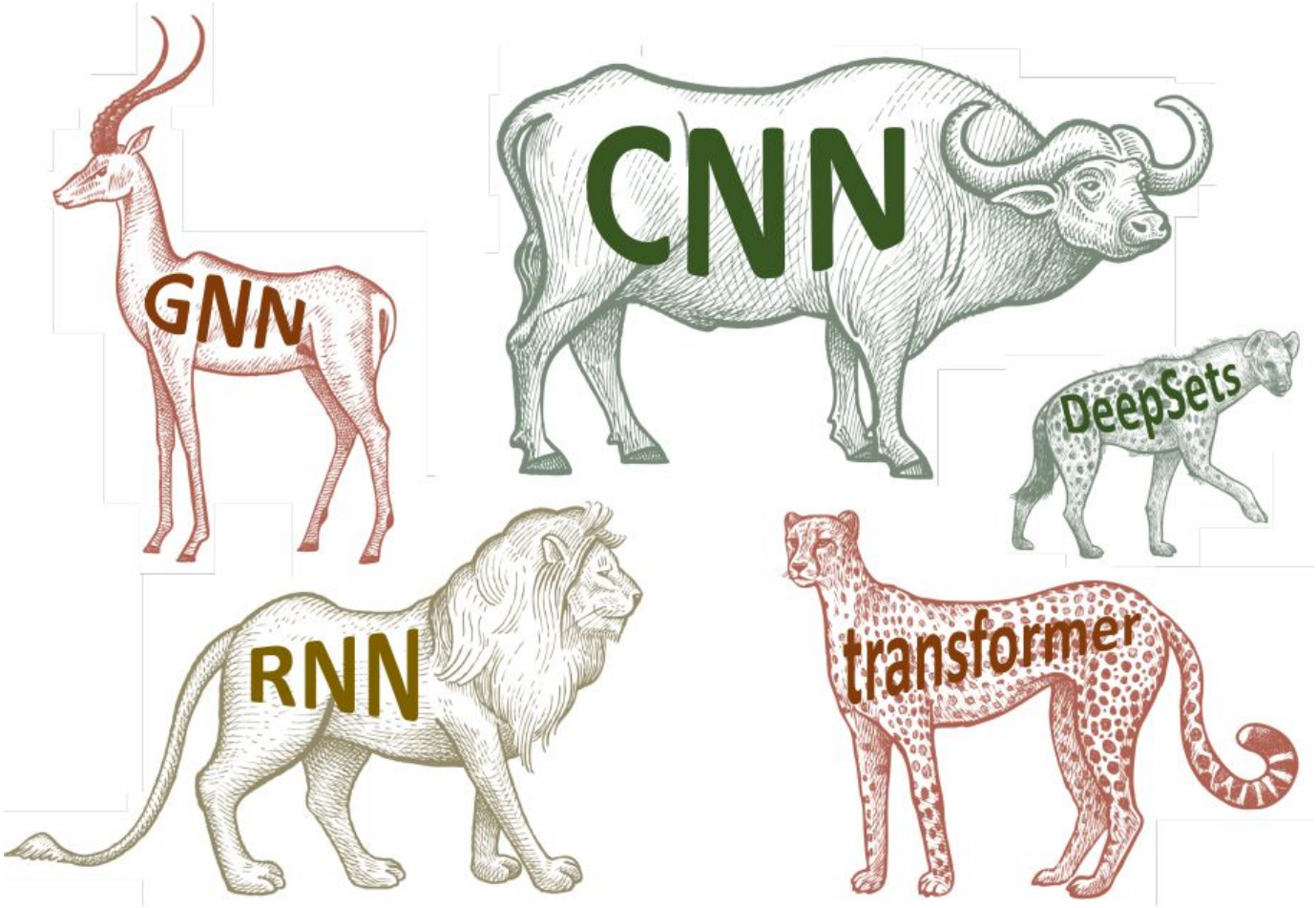
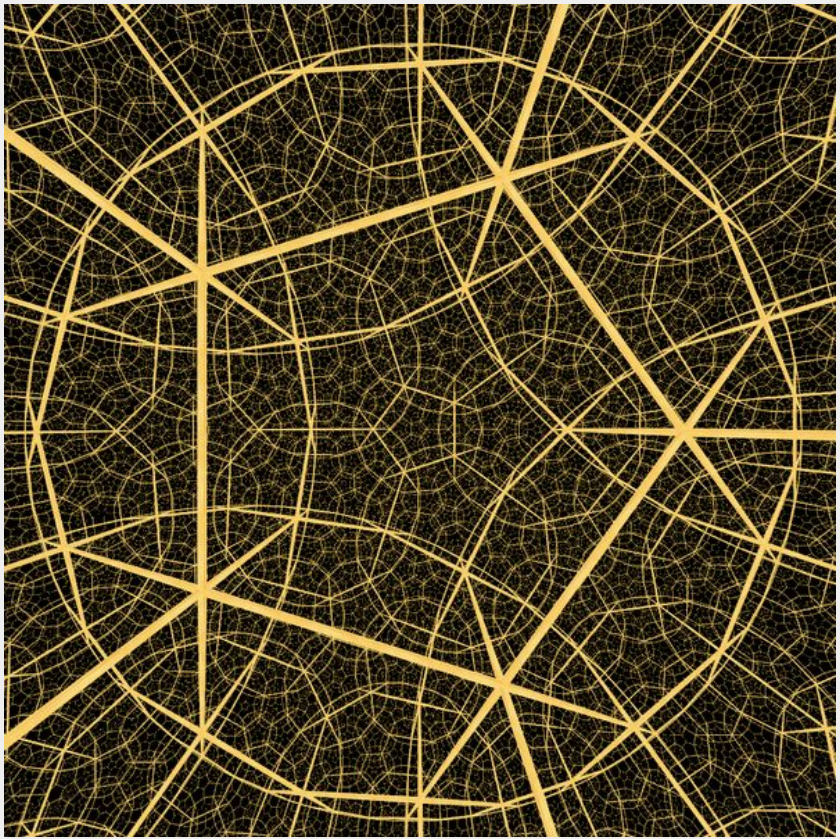
Recurrent Neural Networks

Usman Nazir

Many Architectures, Few Principles

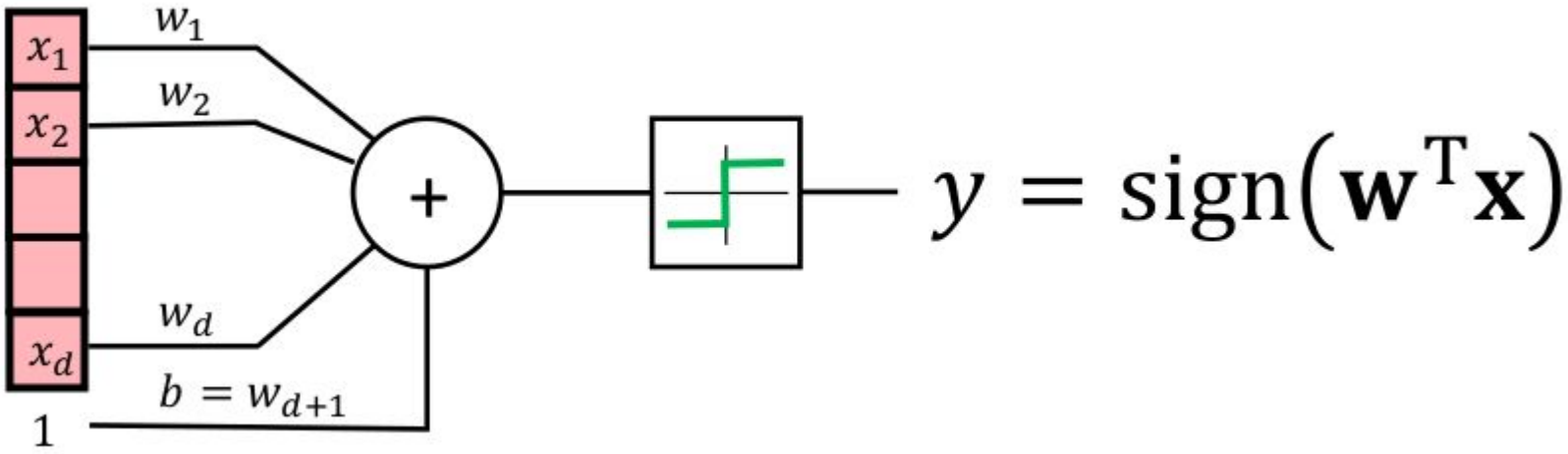


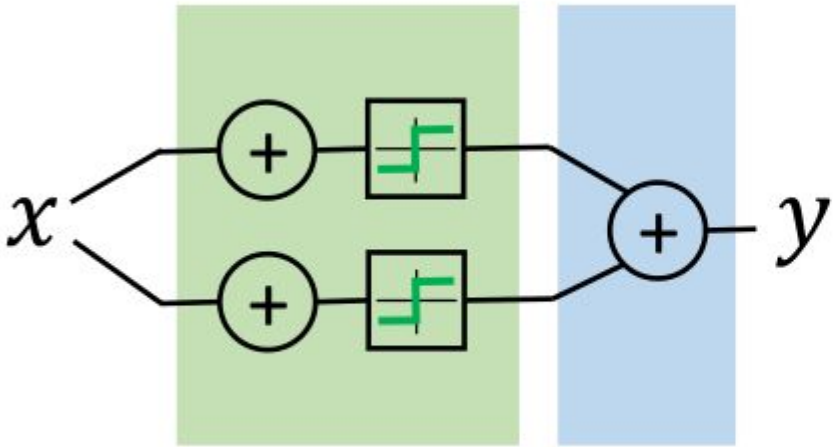
The Erlangen Programme of ML

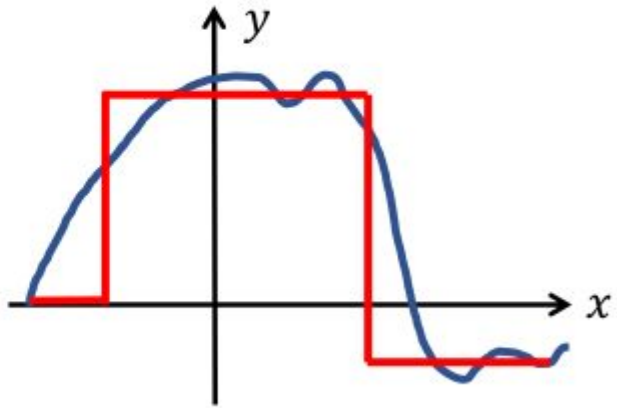


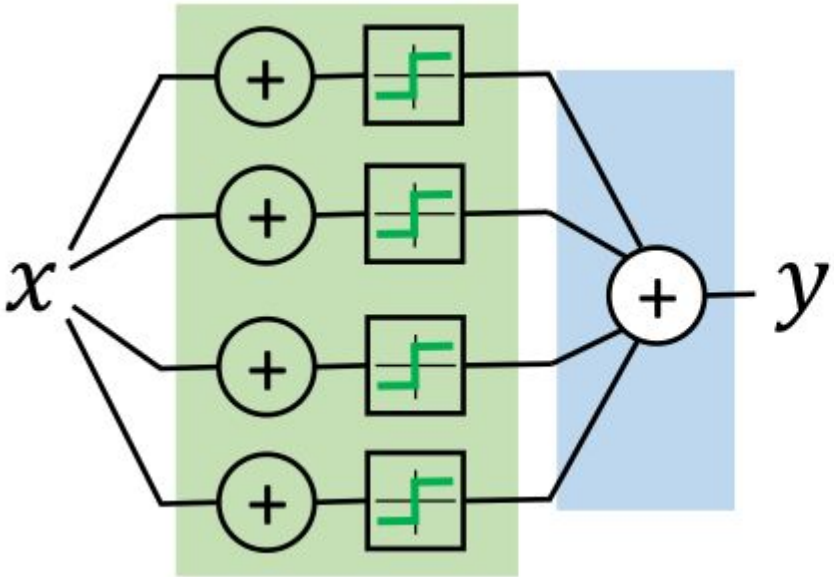

ML
black
box
{cat,dog}

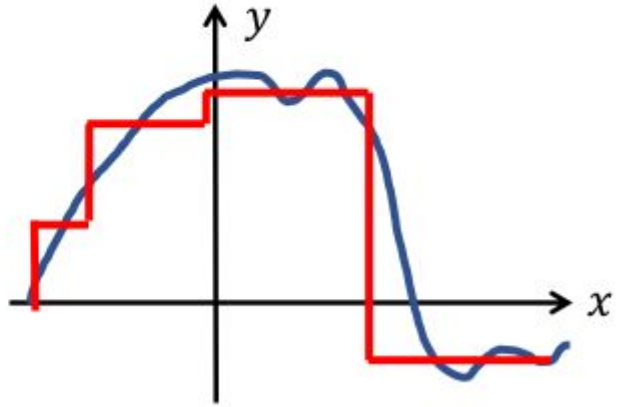
Simplest Neural Network



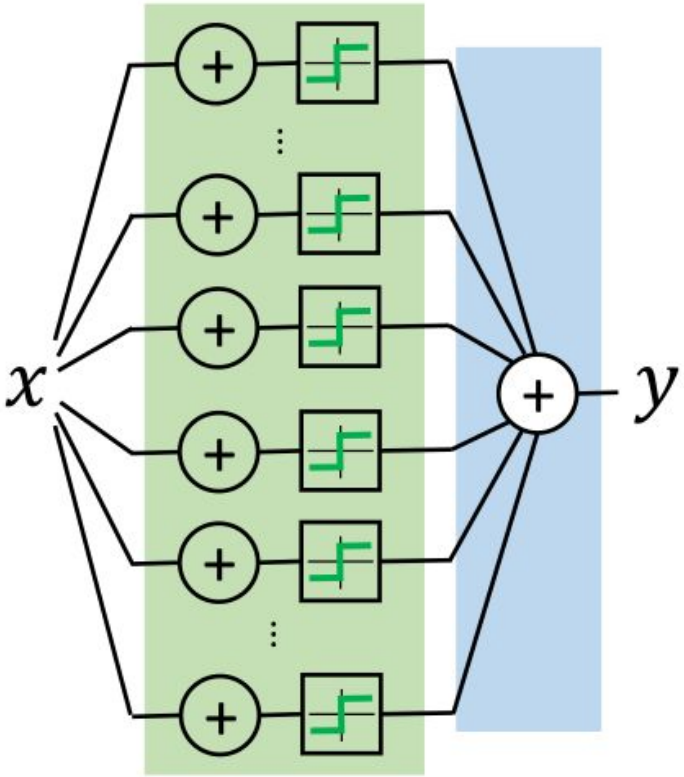

x
y

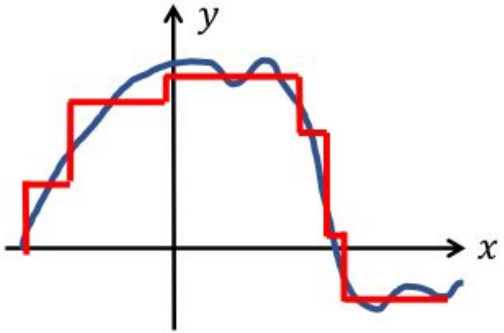

y
x


x
y


y
x

Universal Approximation

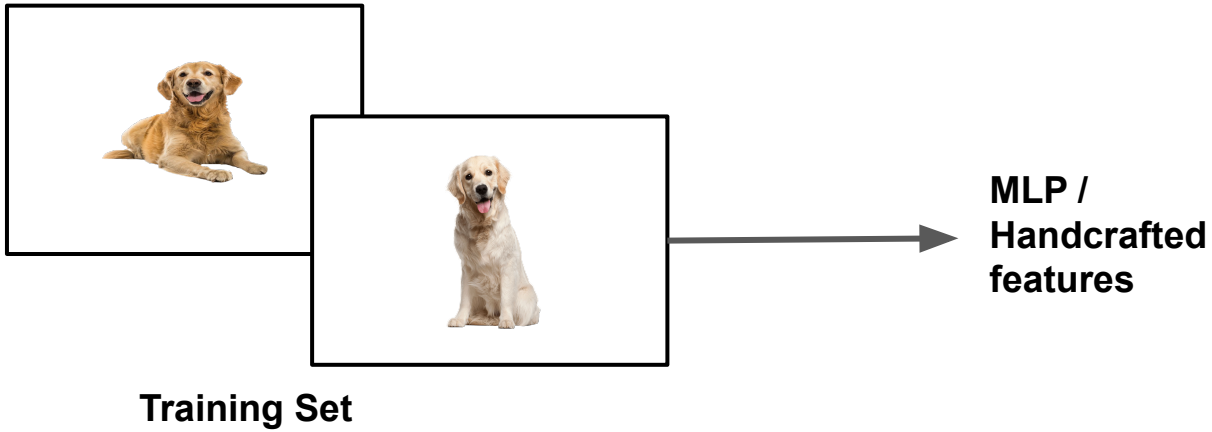




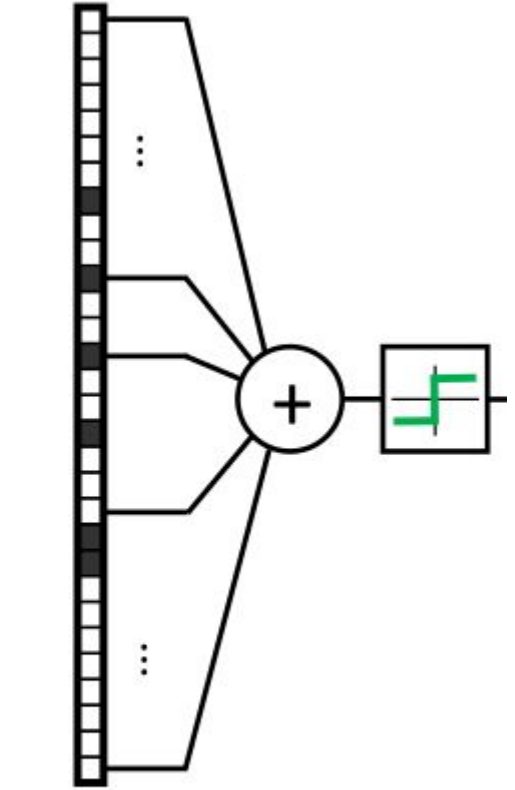
can approximate a continuous function to any desired accuracy

Cybenko 1989; Hornik 1991

Curse of Dimensionality

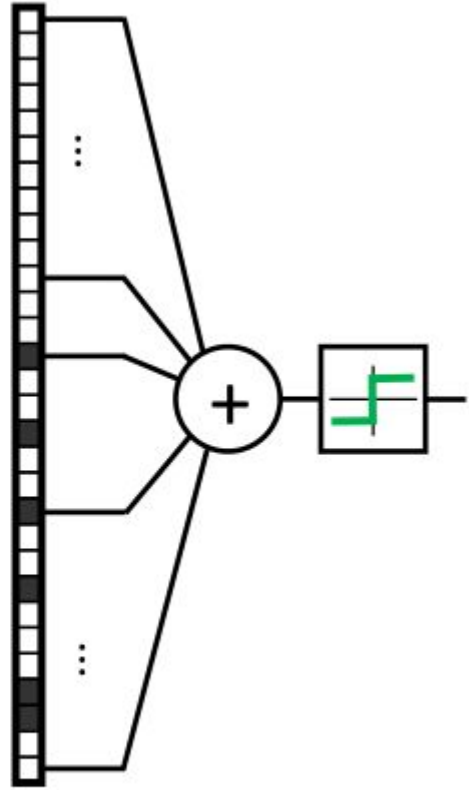


Curse of Dimensionality

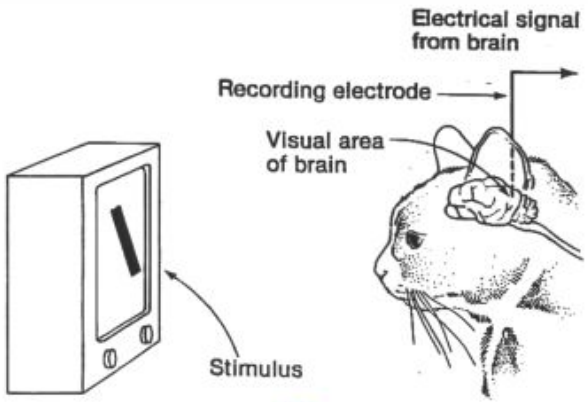


Curse of Dimensionality

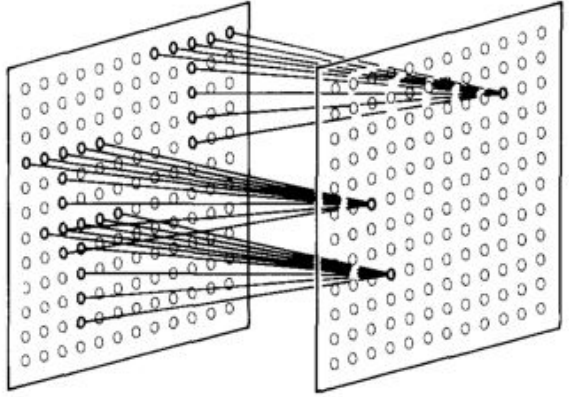
Remedy?



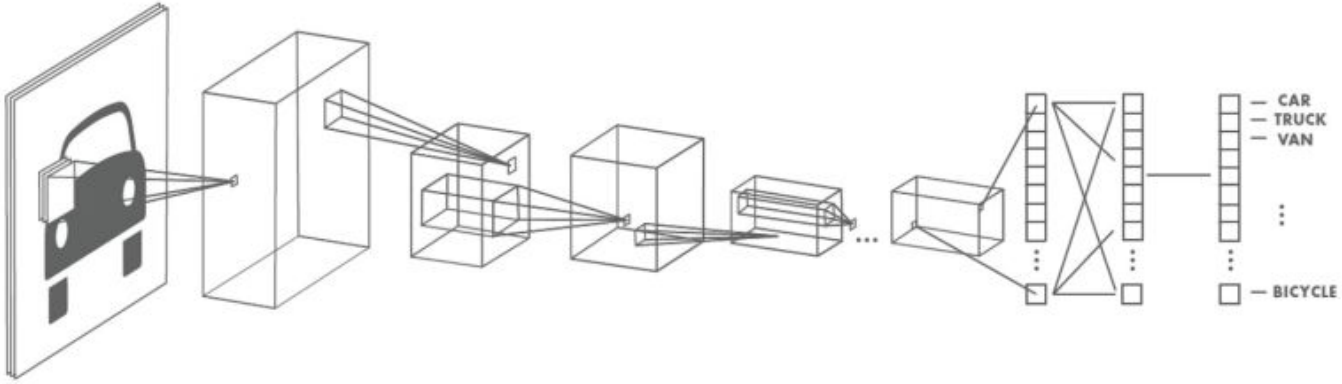
must learn shift invariance from data!



Hubel, Wiesel 1962;  1981



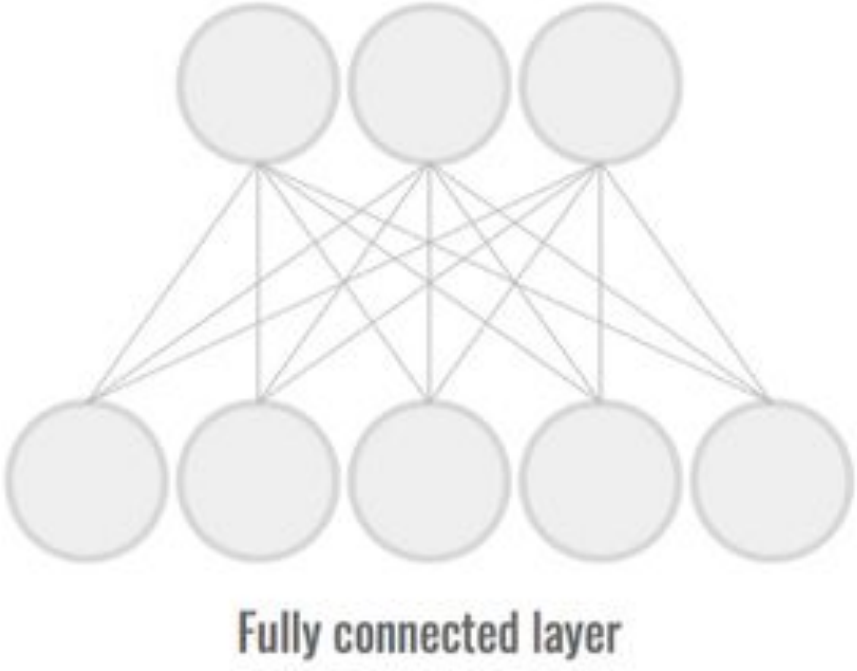
Fukushima 1980

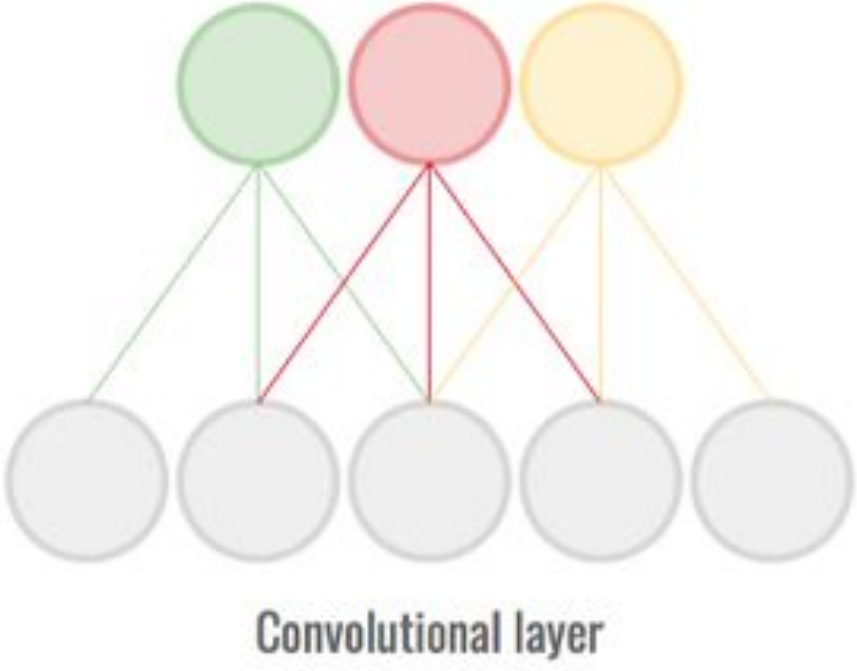


LeCun et al. 1989

Convolution

1. Local filters (local receptive field connectivity)

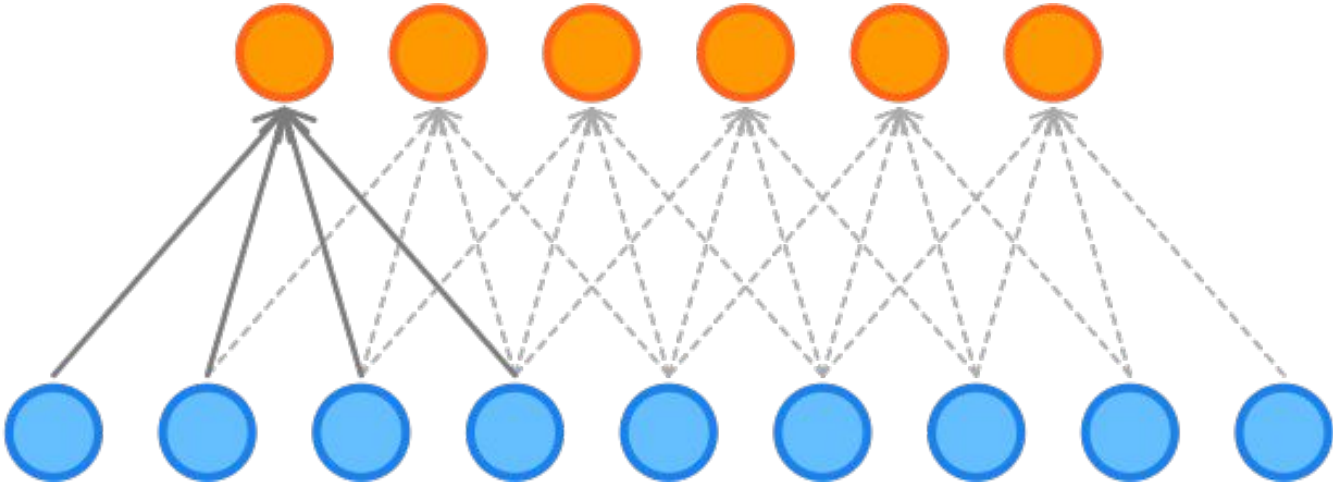




Convolution

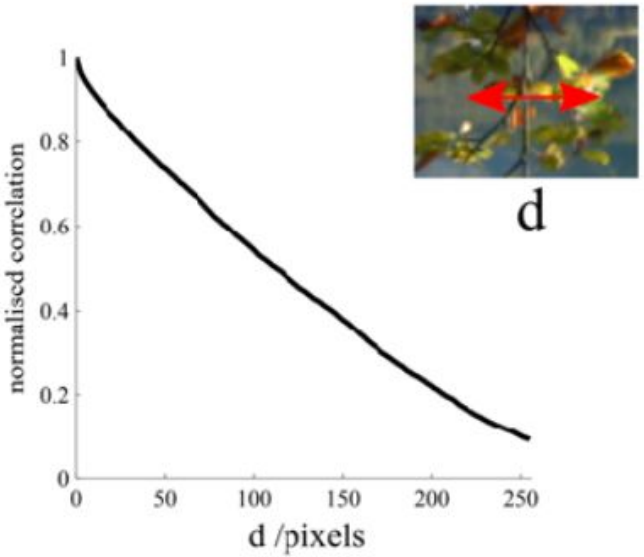
1. Local filters (local receptive field connectivity)

```
kernel_size = 4
strides = 1
padding = 'valid'
```



Convolution

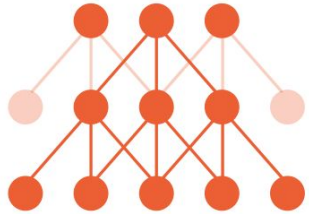
1. Local filters (local receptive field connectivity)



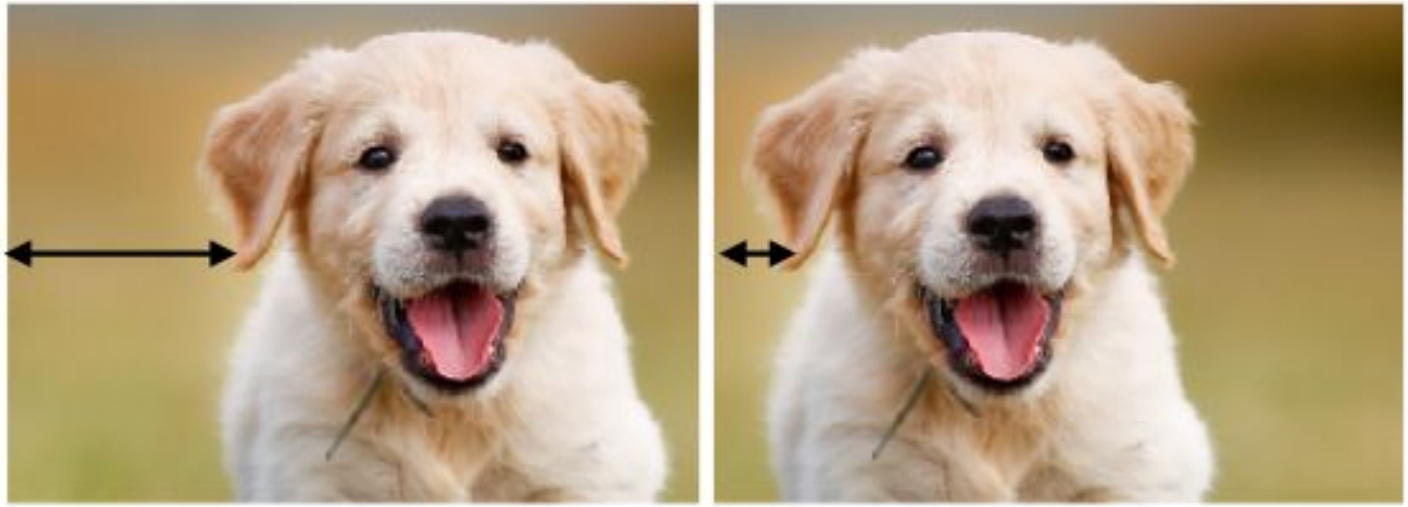
Locality of pixel statistics (Property of data)

http://www.gatsby.ucl.ac.uk/~turner/teaching/4g3/2013/stats-recep-field.pdf

Convolution



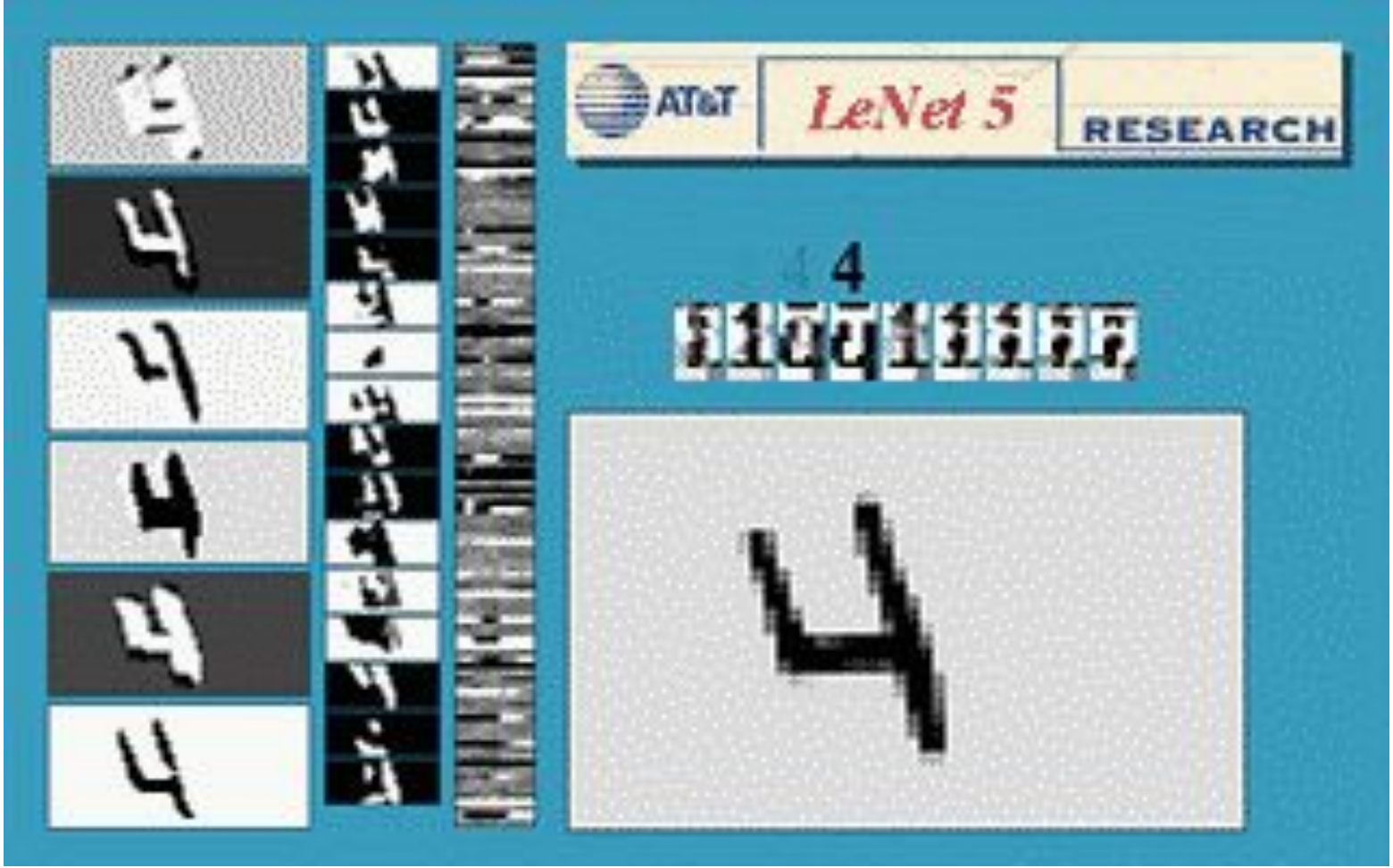
1. Local filters (local receptive field connectivity)
2. Translation weight tying (Weight sharing exploits translation symmetry)



Translational invariance of classification (Property of task)

https://www.healthypawspetinsurance.com/Images/

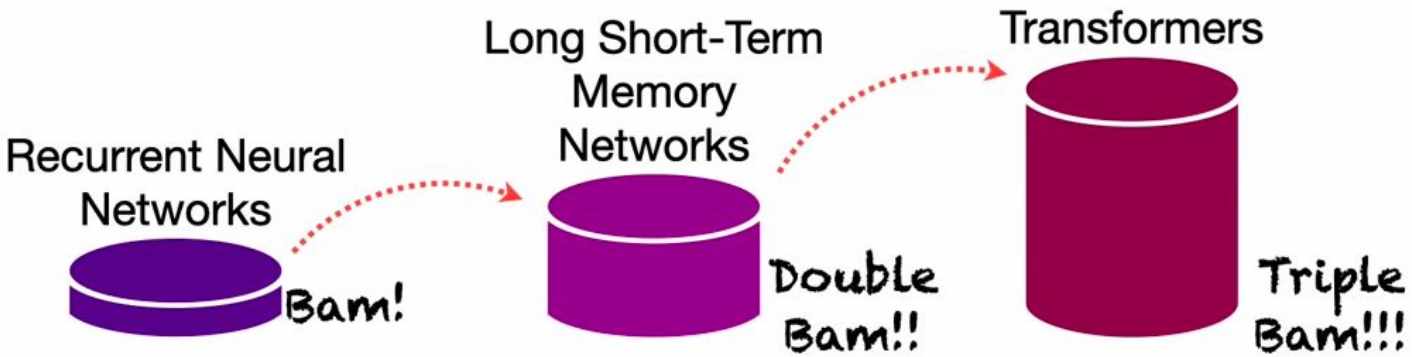
http://yann.lecun.com/exdb/lenet/translation.html



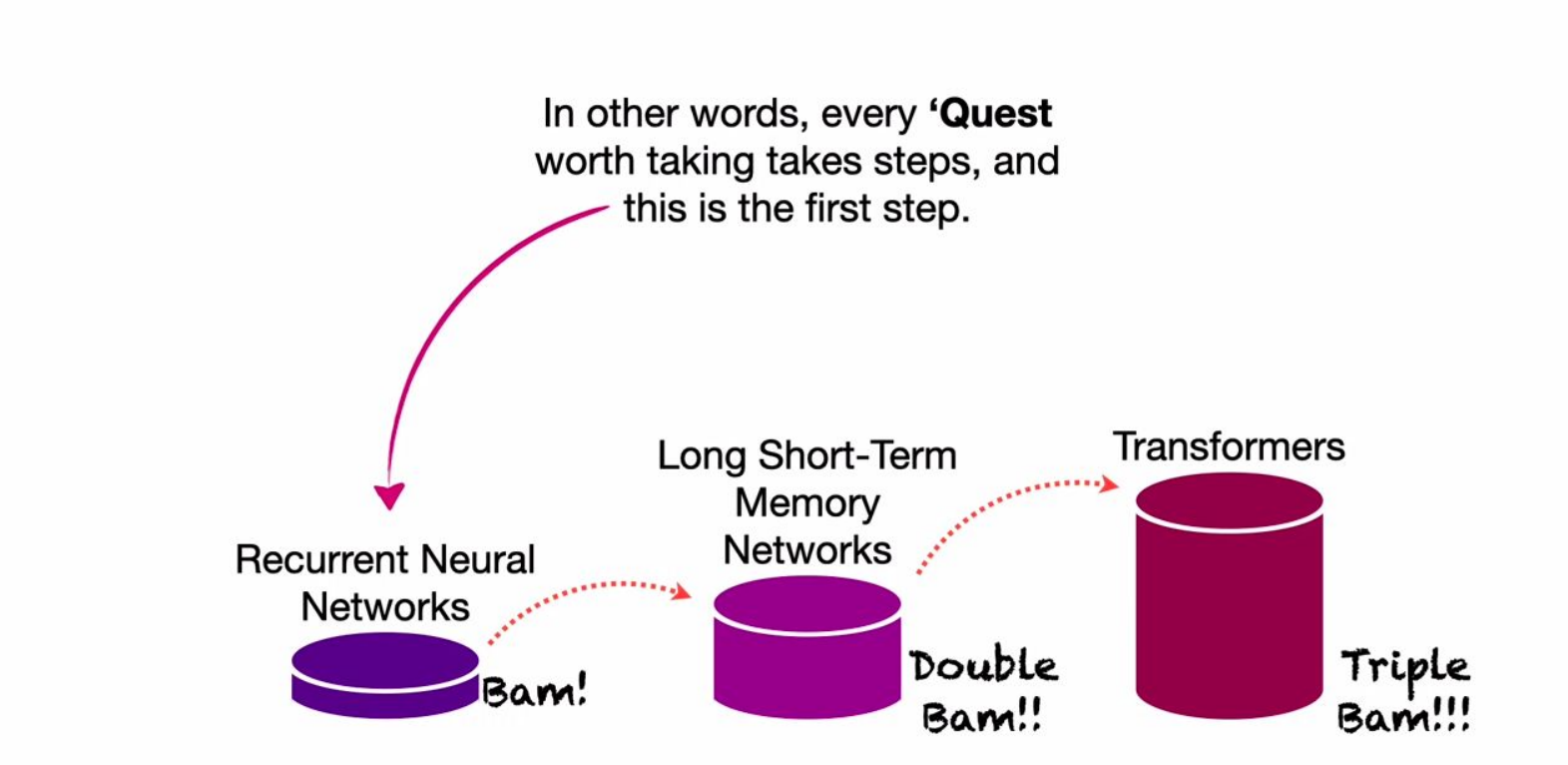
http://yann.lecun.com/exdb/lenet/translation.html

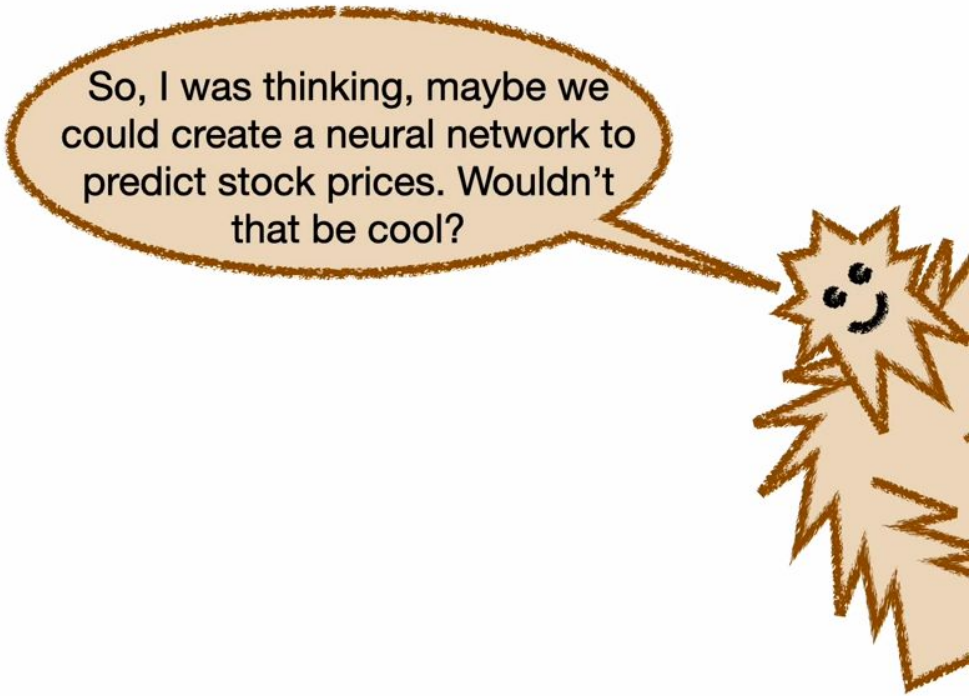
RNN —> LSTM —> Transformers

ALSO NOTE: Although ***basic***, or ***vanilla*** **Recurrent Neural Networks** are awesome, they are usually thought of as a stepping stone to understanding ***fancier*** things like **Long Short-Term Memory Networks** and **Transformers**, which we will talk about in future **StatQuests**.

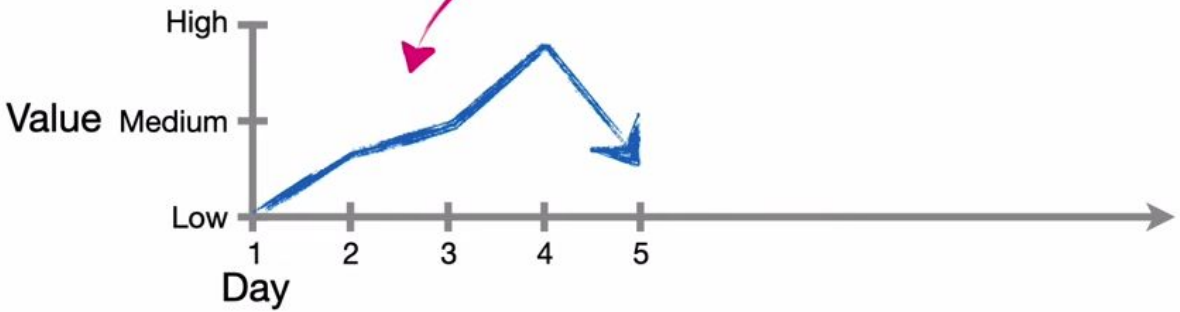


RNN


In other words, every 'Quest worth taking takes steps, and this is the first step.
Recurrent Neural Networks
Bam!
Long Short-Term Memory Networks
Double Bam!!
Transformers
Triple Bam!!!


So, I was thinking, maybe we could create a neural network to predict stock prices. Wouldn't that be cool?

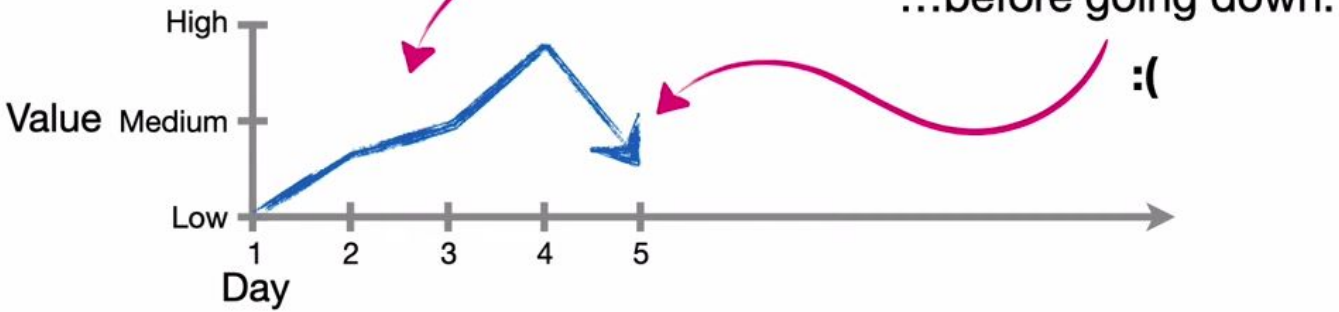
For example, the price of this stock went up for **4** days…



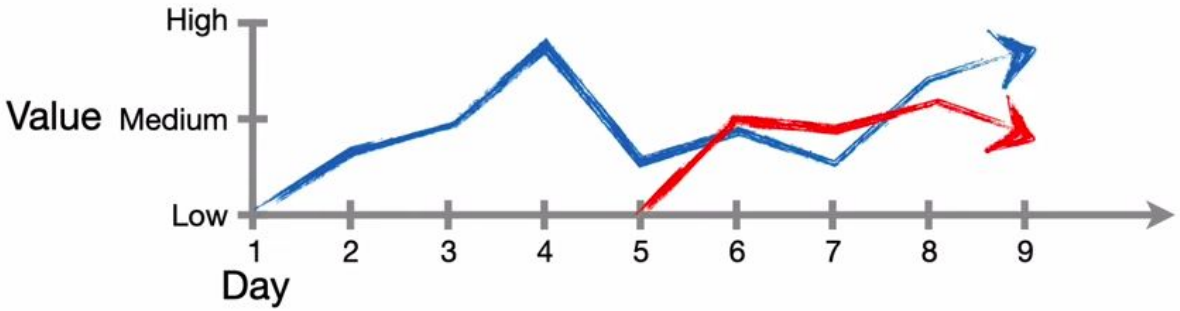
For example, the price of this stock went up for **4** days…

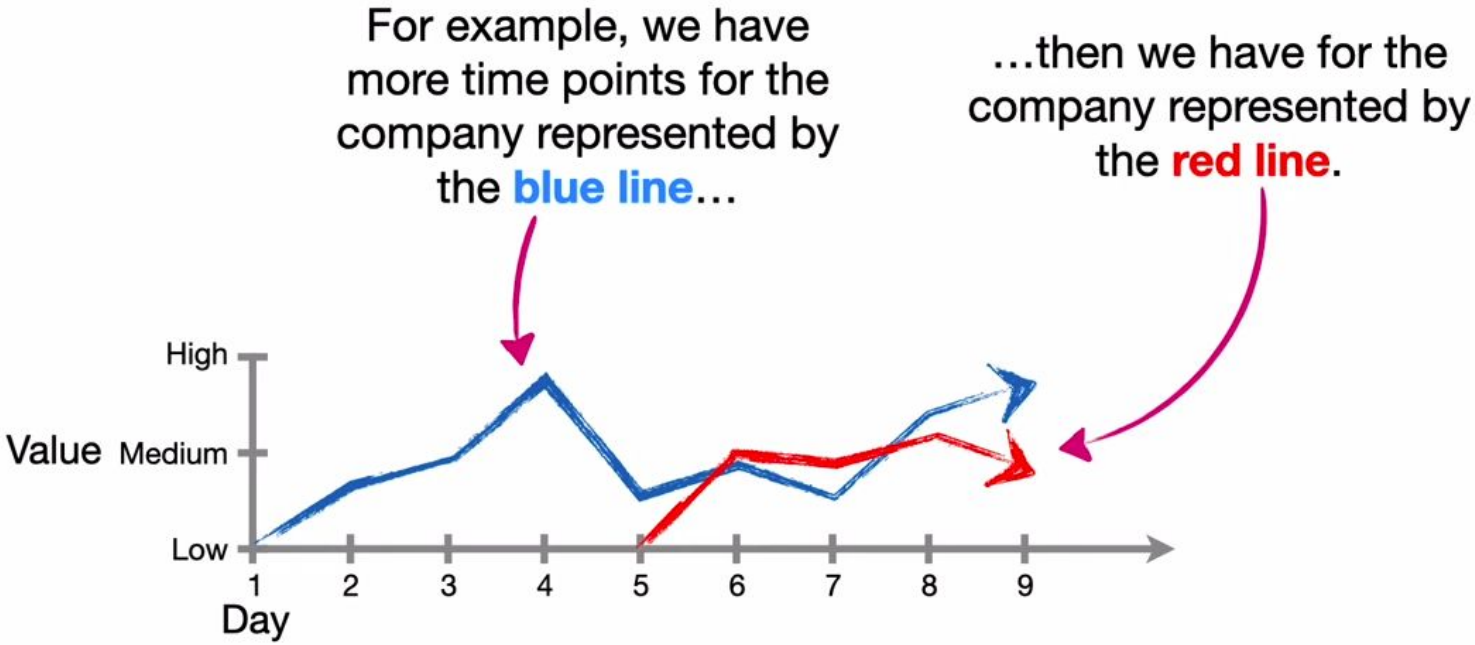
…before going down.

:(

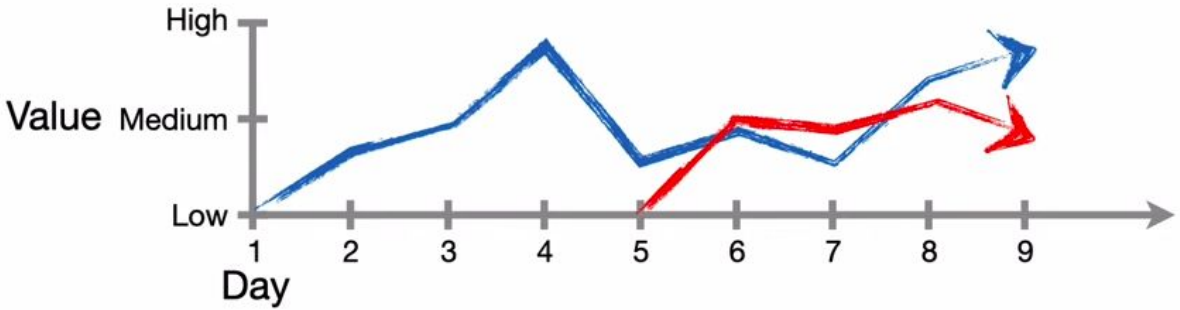


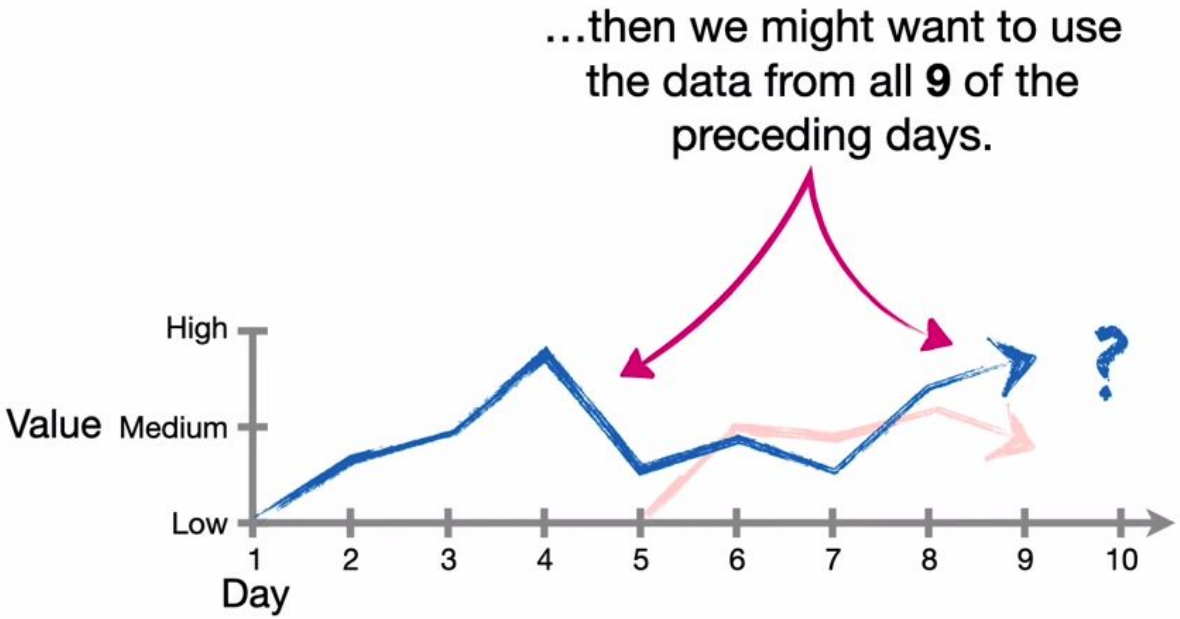
Also, the longer a company has been traded on the stock market, the more data we'll have for it.



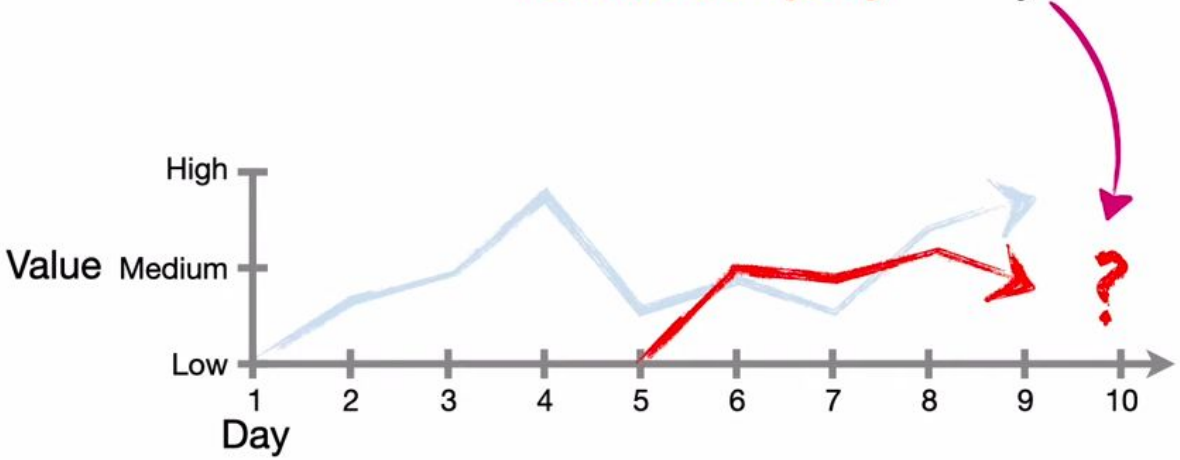

For example, we have more time points for the company represented by the blue line…
…then we have for the company represented by the red line.
High
Value
Medium
Low
1
2
3
4
5
6
7
8
9
Day

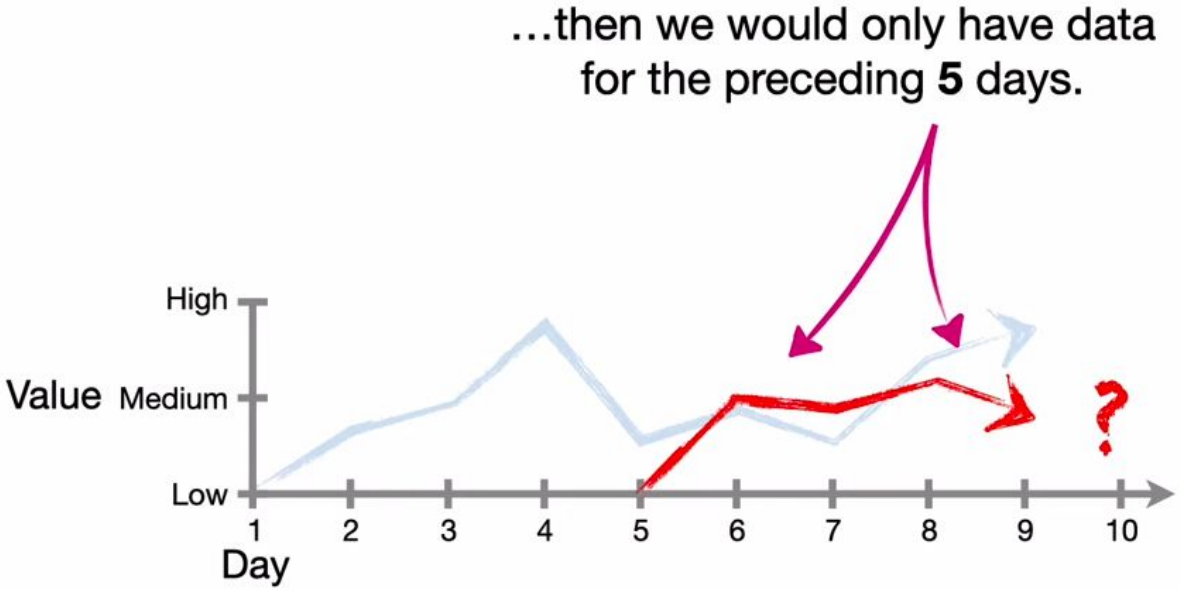
…then we need a neural network that works with different amounts of sequential data.



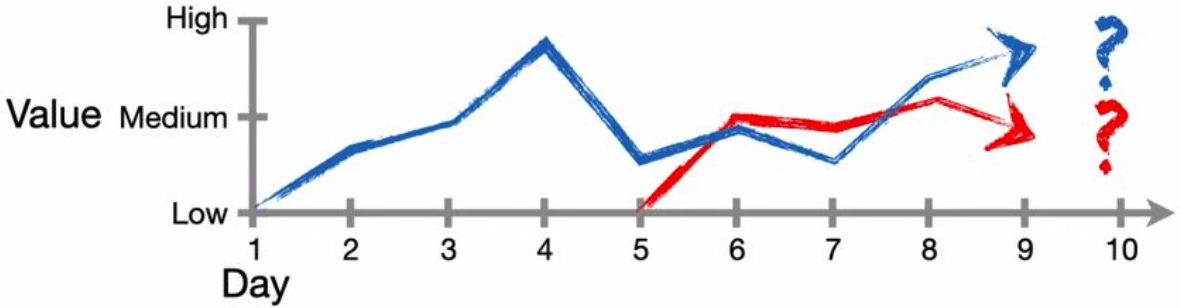

…then we might want to use the data from all 9 of the preceding days.
High
Value
Medium
Low
1
2
3
4
5
6
7
8
9
10
Day
?

In contrast, if we wanted to predict the stock price for the **red line company** on day **10**…

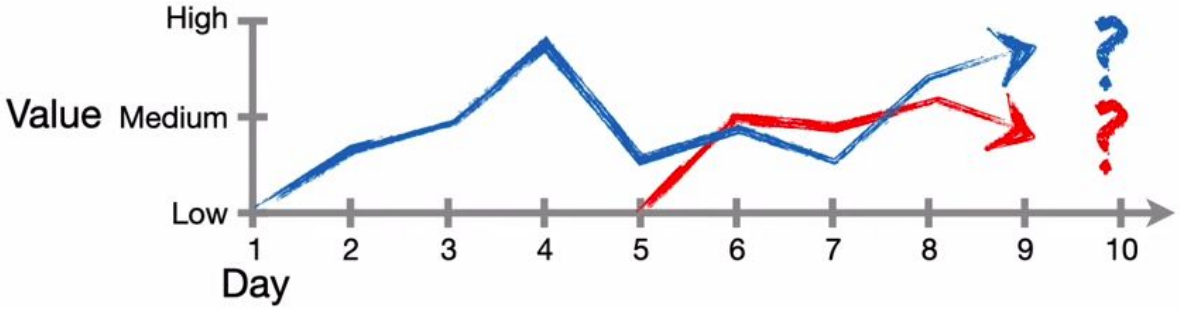



…then we would only have data
for the preceding **5** days.
High
Value
Medium
Low
1
2
3
4
5
6
7
8
9
10
Day
?

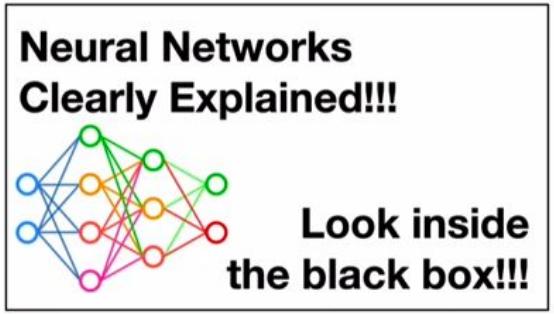
So we need the neural network
to be flexible in terms of how
much sequential data we use
to make a prediction.



This is a big difference compared
to the other neural networks
we've looked at in this series.

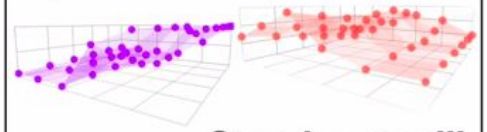


For example, in **Neural Networks Clearly Explained**, we examined a neural network that made predictions using **1** input value, no more and no less.

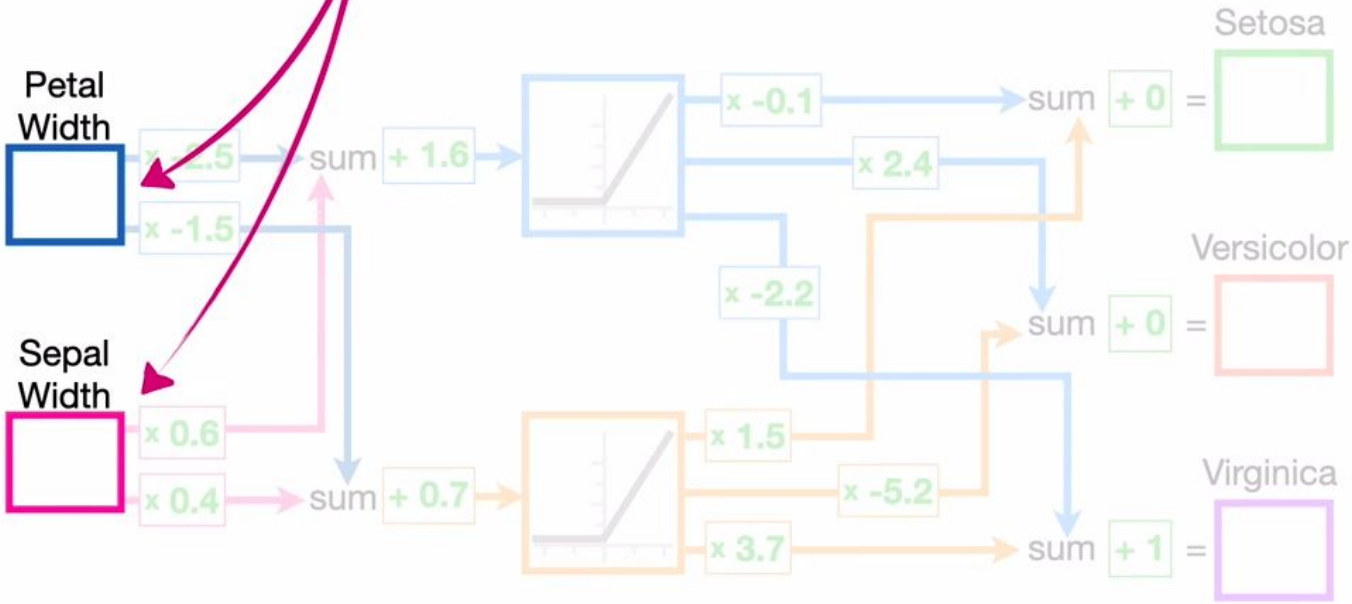


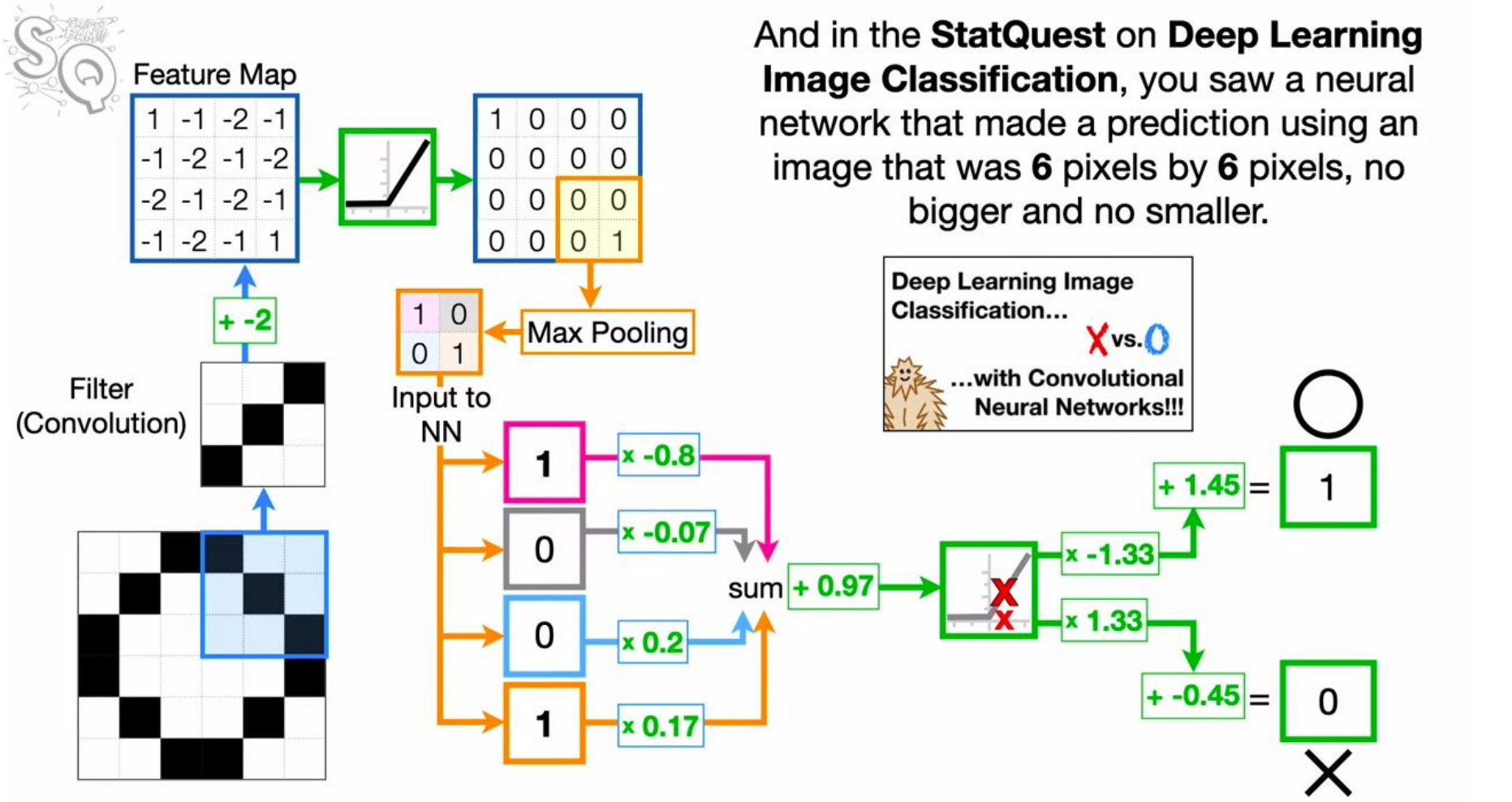
And if you saw the **StatQuest** on **Neural Networks with Multiple Inputs and Outputs**, you saw this neural network that made predictions using **2** input values, no more and no less.

Neural Networks: multiple inputs and outputs...

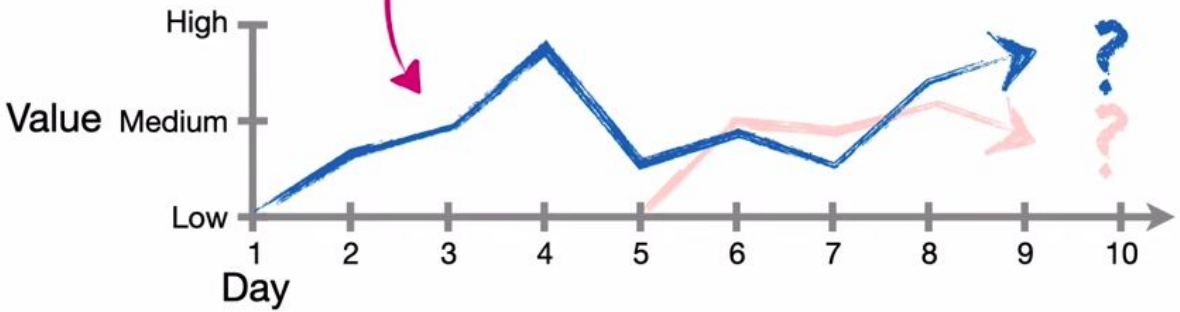


...Step-by-step!!!

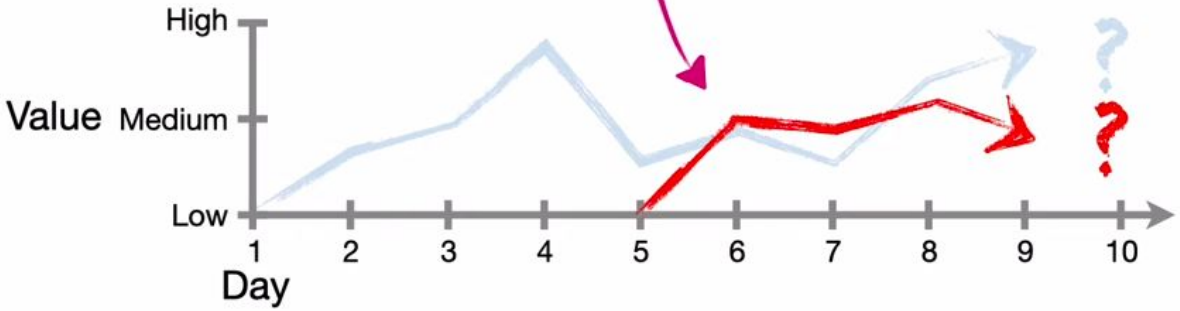



Feature Map
1 -1 -2 -1
-1 -2 -1 -2
-2 -1 -2 -1
-1 -2 -1 1
+ -2
Filter (Convolution)
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 1
Max Pooling
1 0
0 1
Input to NN
1
0
0
1
x -0.8
x -0.07
x 0.2
x 0.17
sum
+ 0.97
x -1.33
x 1.33
+ 1.45 = 1
+ -0.45 = 0
And in the StatQuest on Deep Learning Image Classification, you saw a neural network that made a prediction using an image that was 6 pixels by 6 pixels, no bigger and no smaller.
Deep Learning Image Classification...
X vs. O
...with Convolutional Neural Networks!!!

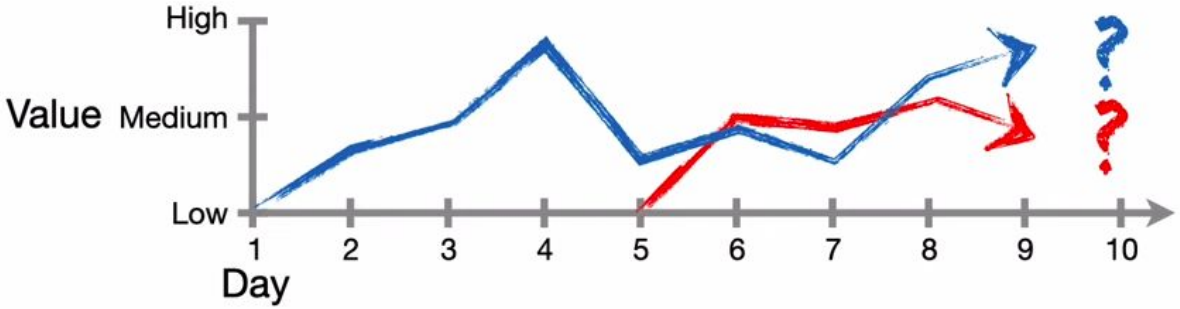
However, now we need a neural network that can make a prediction using the **9** values we have for the **blue company**…



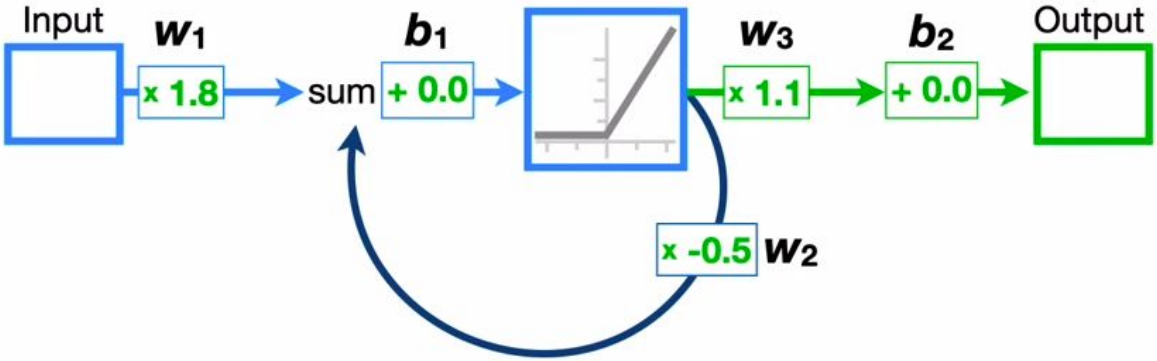
…and make a prediction using the **5** values we have for the **red company**.



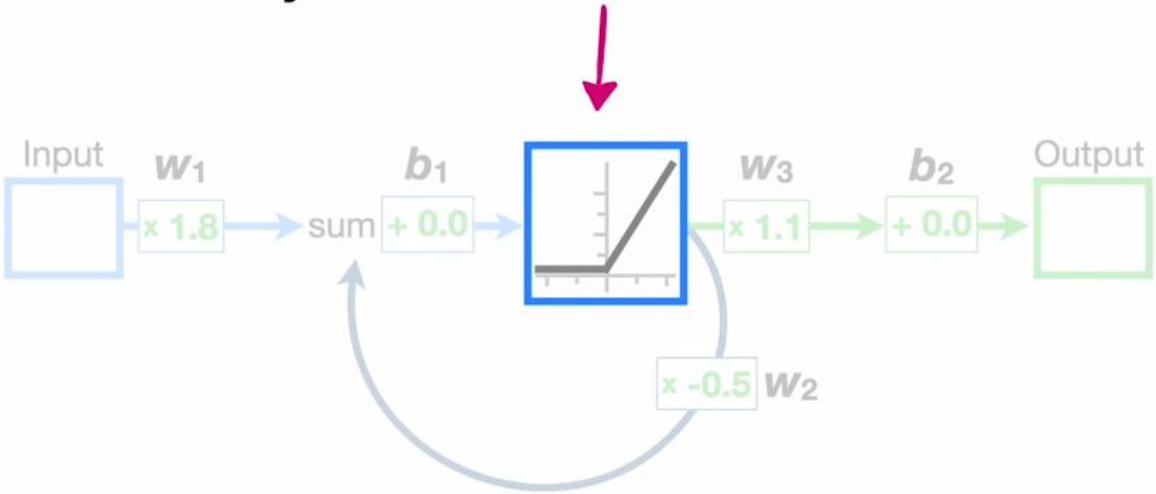
The good news is that one way to deal with the problem of having different amounts of input values is to use a **Recurrent Neural Network (RNN)**.



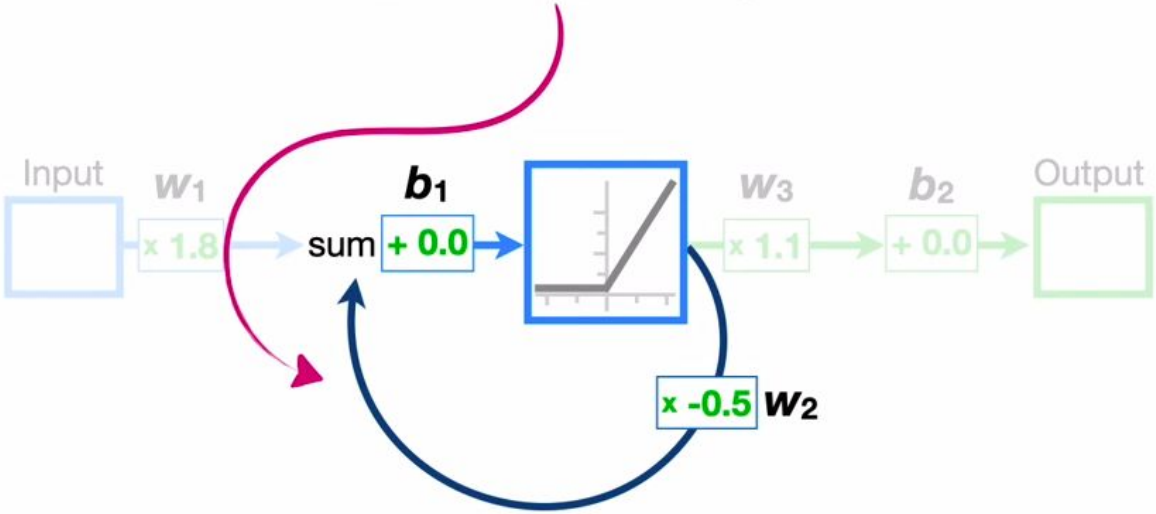
Just like the other neural networks that we've seen before, **Recurrent Neural Networks** have **weights**, **biases**,



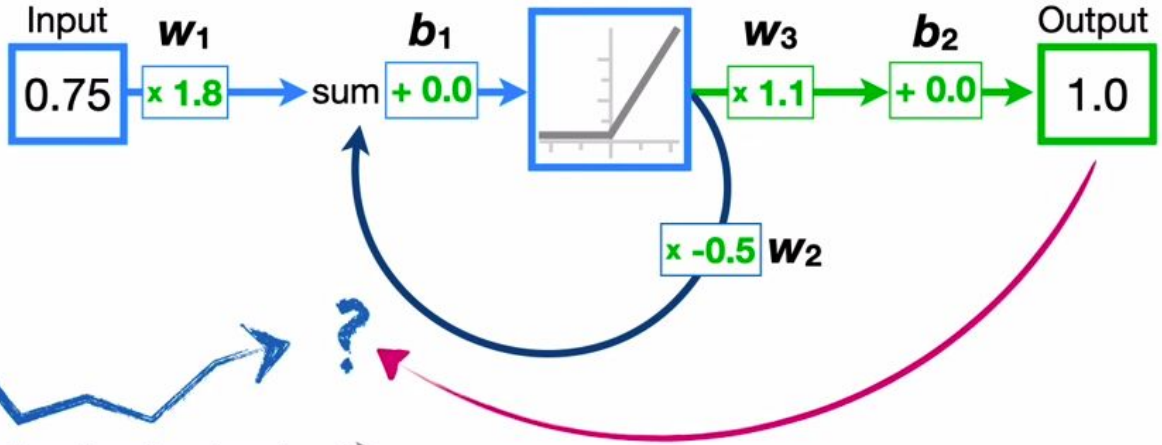
Just like the other neural networks that we've seen before, **Recurrent Neural Networks** have **weights**, **biases**, **layers** and **activation functions**.

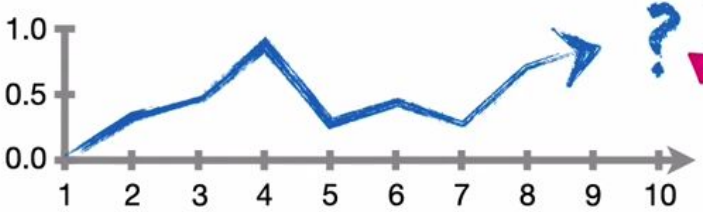


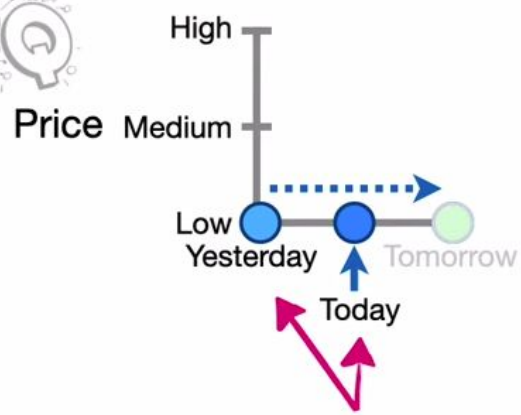
The big difference is that
Recurrent Neural Networks also
have **feedback loops.**



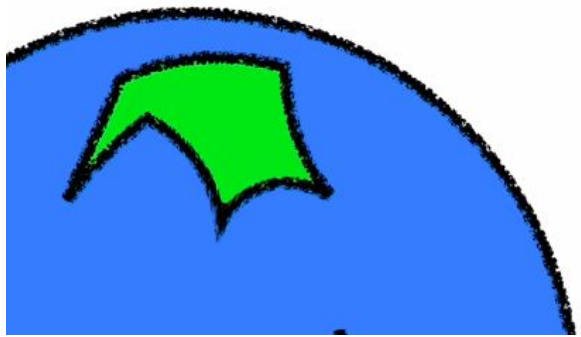
…the **feedback loop** makes it possible to use *sequential* input values, like stock market prices collected over time, to make predictions.

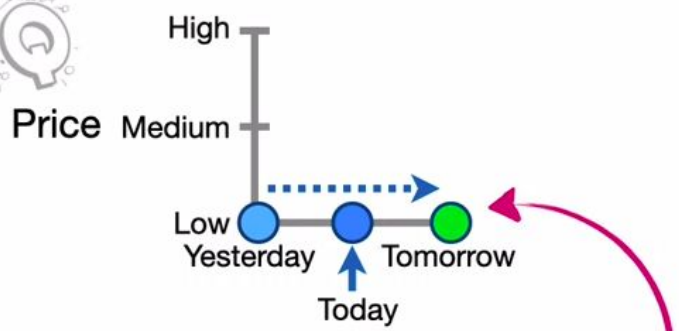




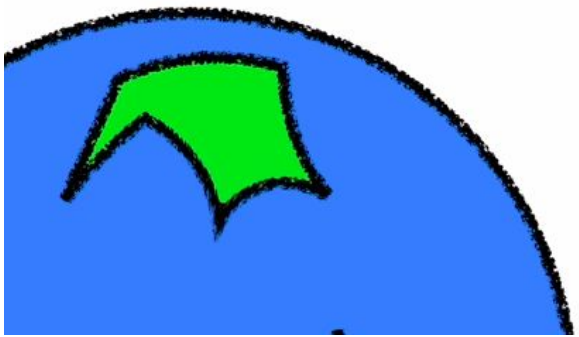


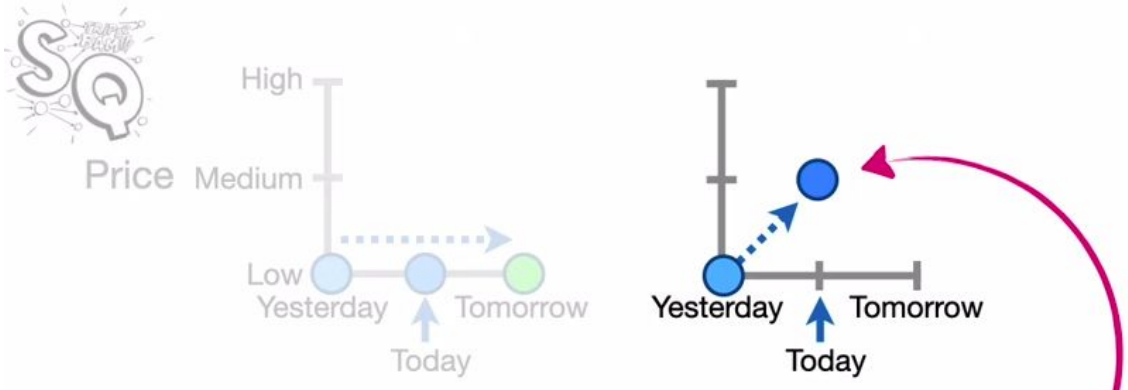
In other words, if **yesterday** and **today's** stock price is low…



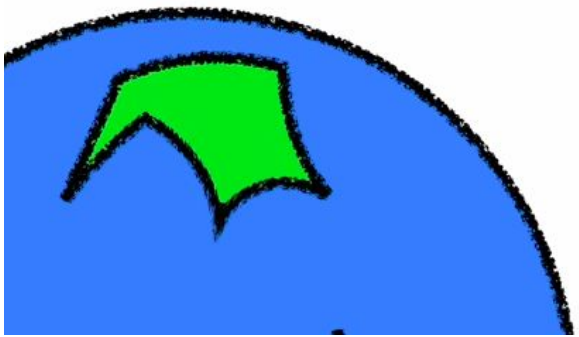


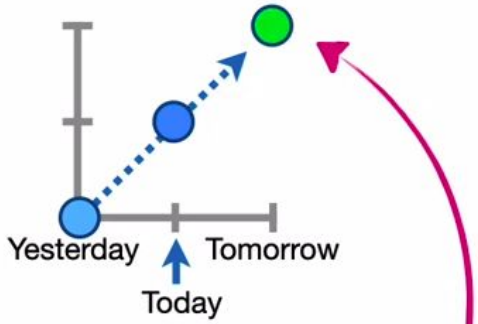
…then **tomorrow's** price should also be low.



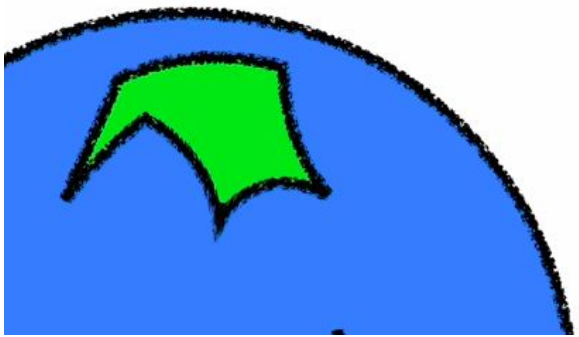


In contrast, if **yesterday's** price was low and **today's** price is medium…

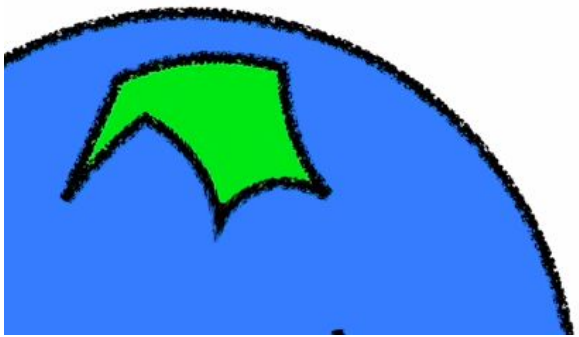


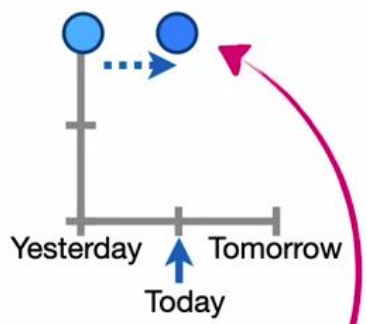


…then **tomorrow's** price should be even higher.

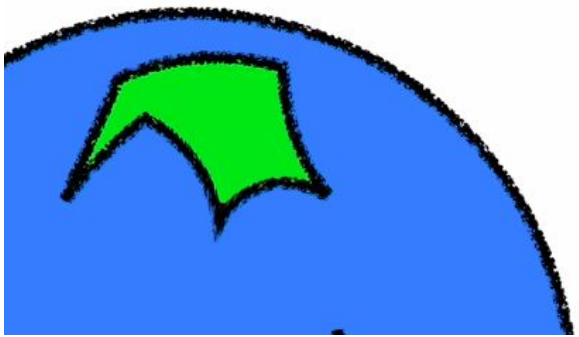


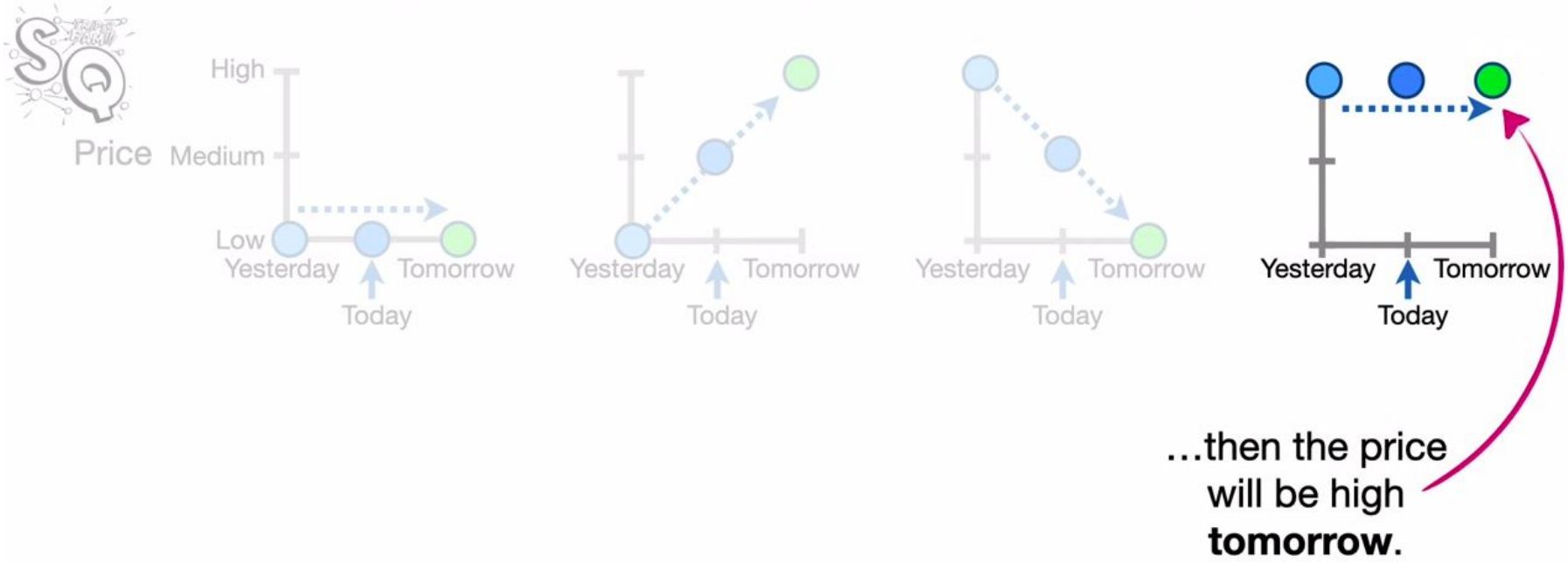

Price
High
Medium
Low
Yesterday
Today
Tomorrow
Yesterday
Today
Tomorrow
Yesterday
Today
Tomorrow
…then **tomorrow's** price will be even lower.

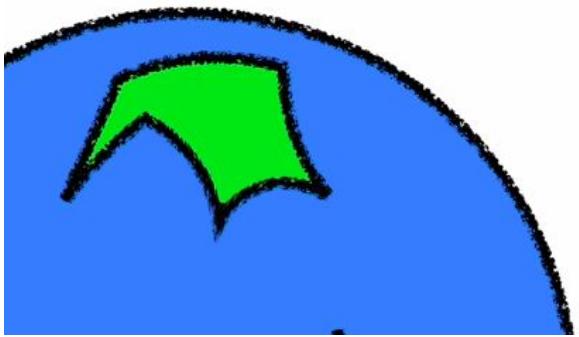


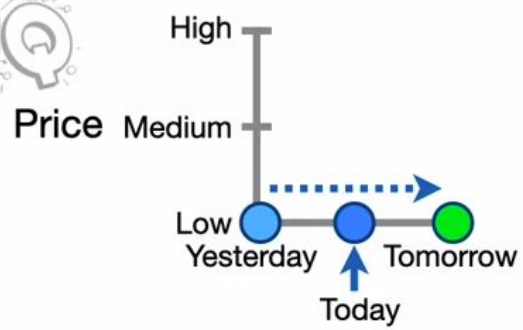


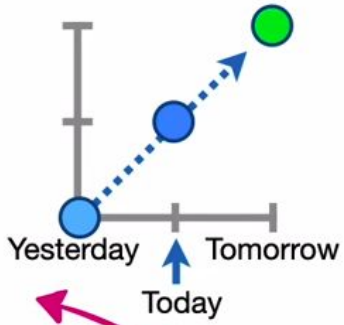
Lastly, if the price stays high for two days in a row…

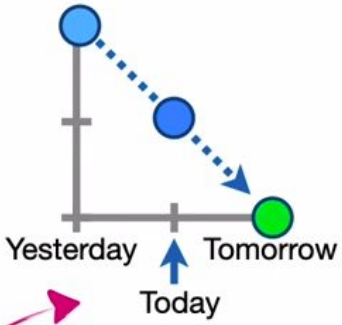


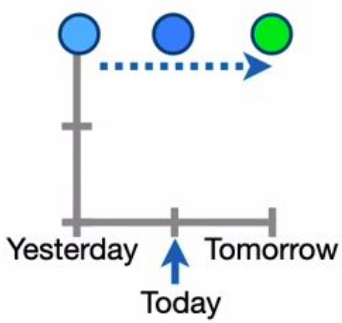

Price
High
Medium
Low
Yesterday
Tomorrow
Today
Yesterday
Tomorrow
Today
Yesterday
Tomorrow
Today
Yesterday
Tomorrow
Today
…then the price will be high **tomorrow**.



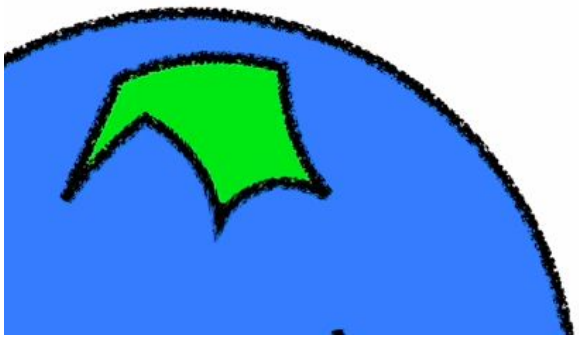


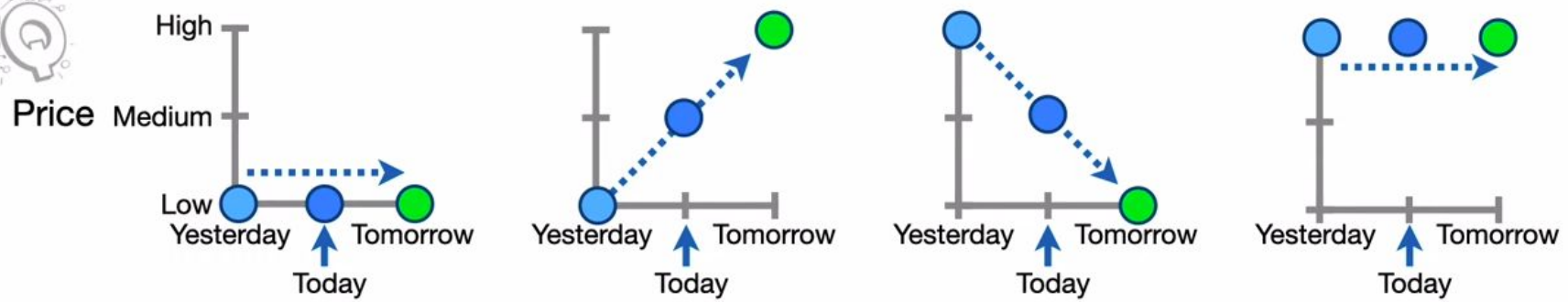




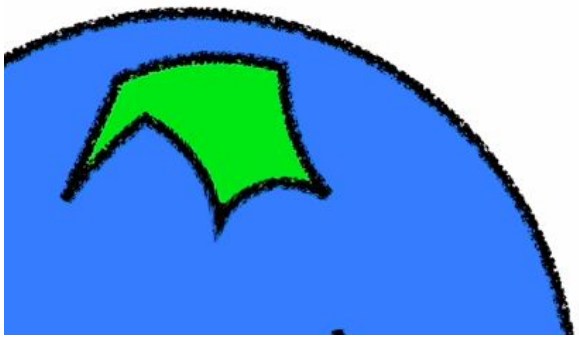


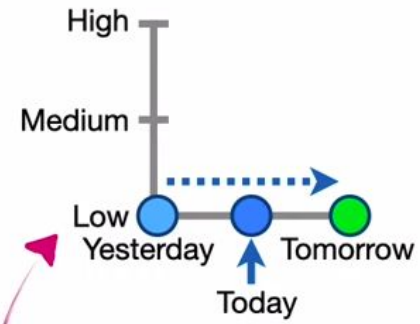
Now that we see the general trends in stock prices in **StatLand**…

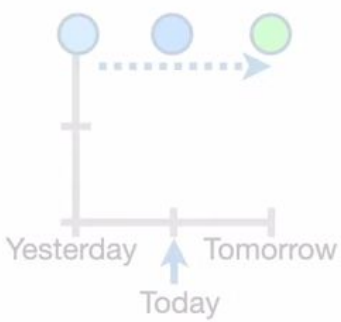




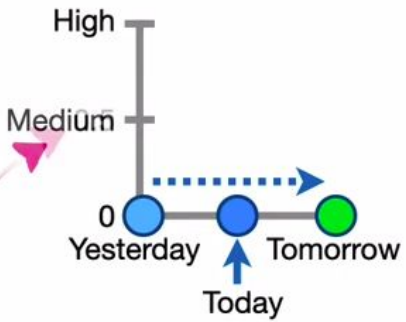
…we can talk about how to run **yesterday** and **today's** data through a **Recurrent Neural Network** to predict **tomorrow's** price.





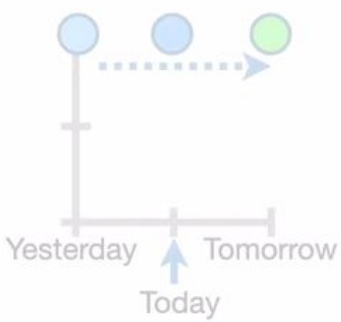


The first thing we'll do scale the prices so that **Low = 0**…

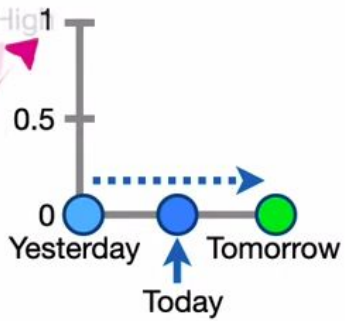

High
Medium
0
Yesterday
Tomorrow
Today


Yesterday
Tomorrow
Today


Yesterday
Tomorrow
Today

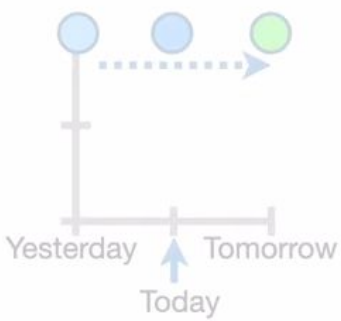

Yesterday
Tomorrow
Today

…Medium = 0.5…

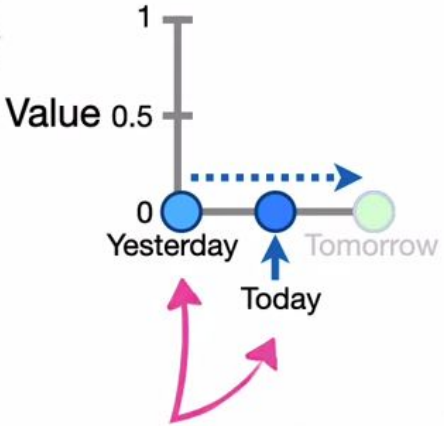

1
0.5
0
Yesterday
Tomorrow
Today


Yesterday
Tomorrow
Today

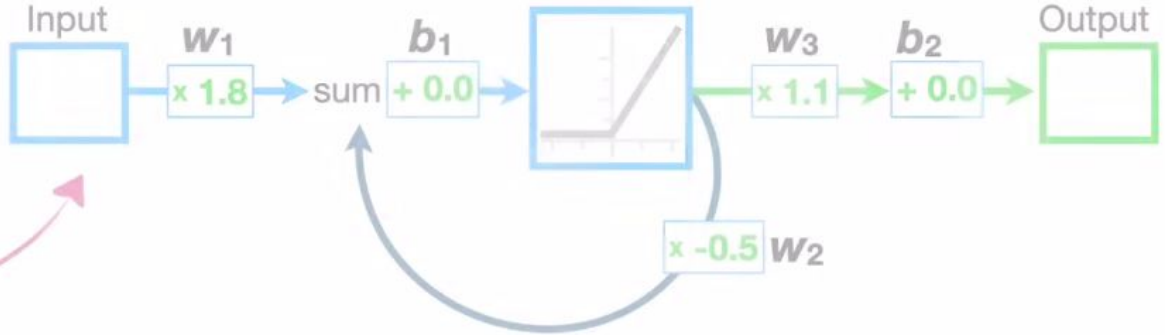

Yesterday
Tomorrow
Today

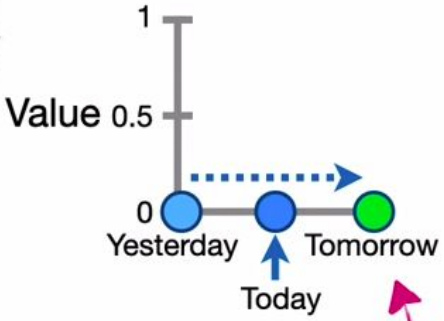

Yesterday
Tomorrow
Today

…and **High = 1**.

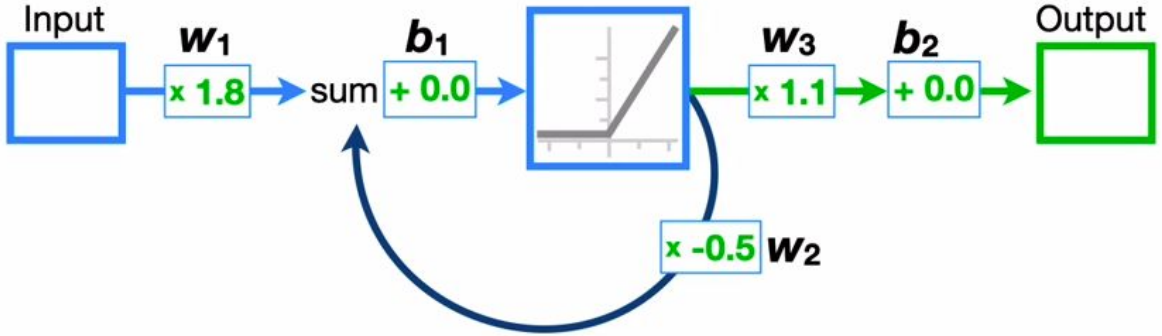


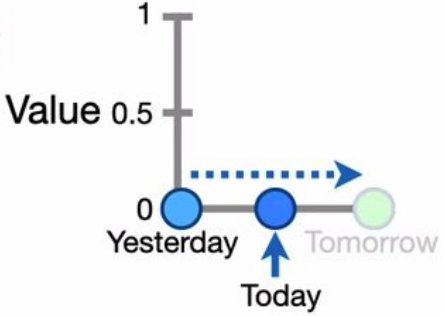
Now let's run the values for **yesterday** and **today**…



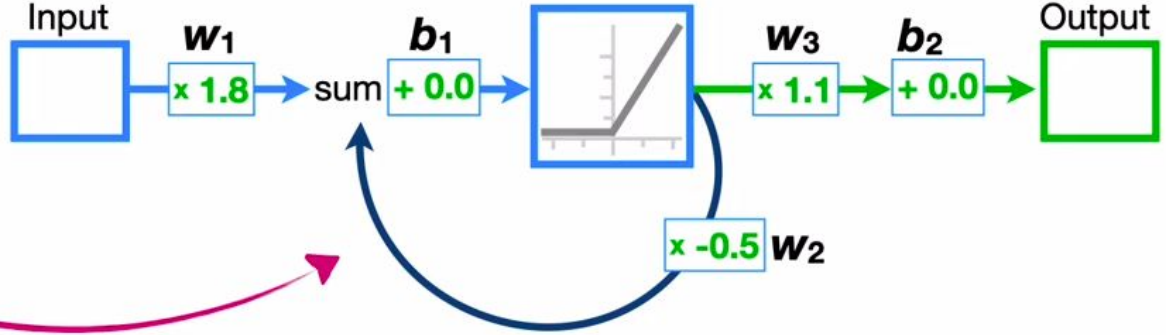


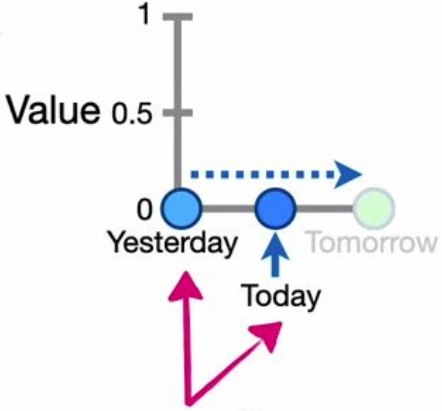
…and see if it can correctly predict **tomorrow's** value.



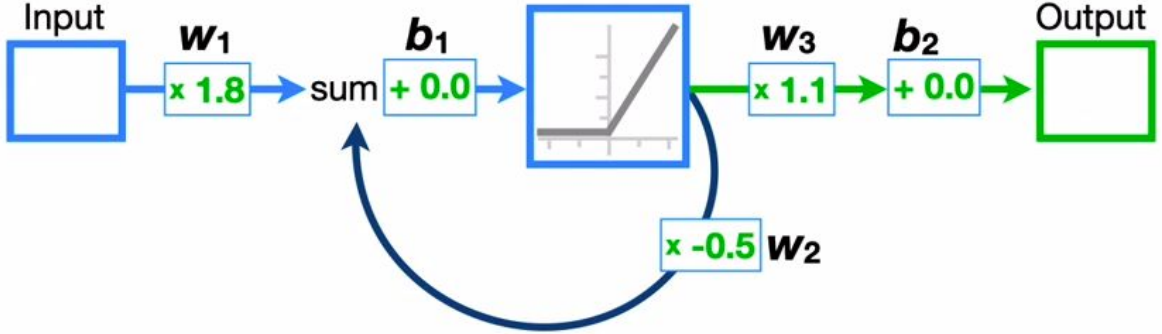


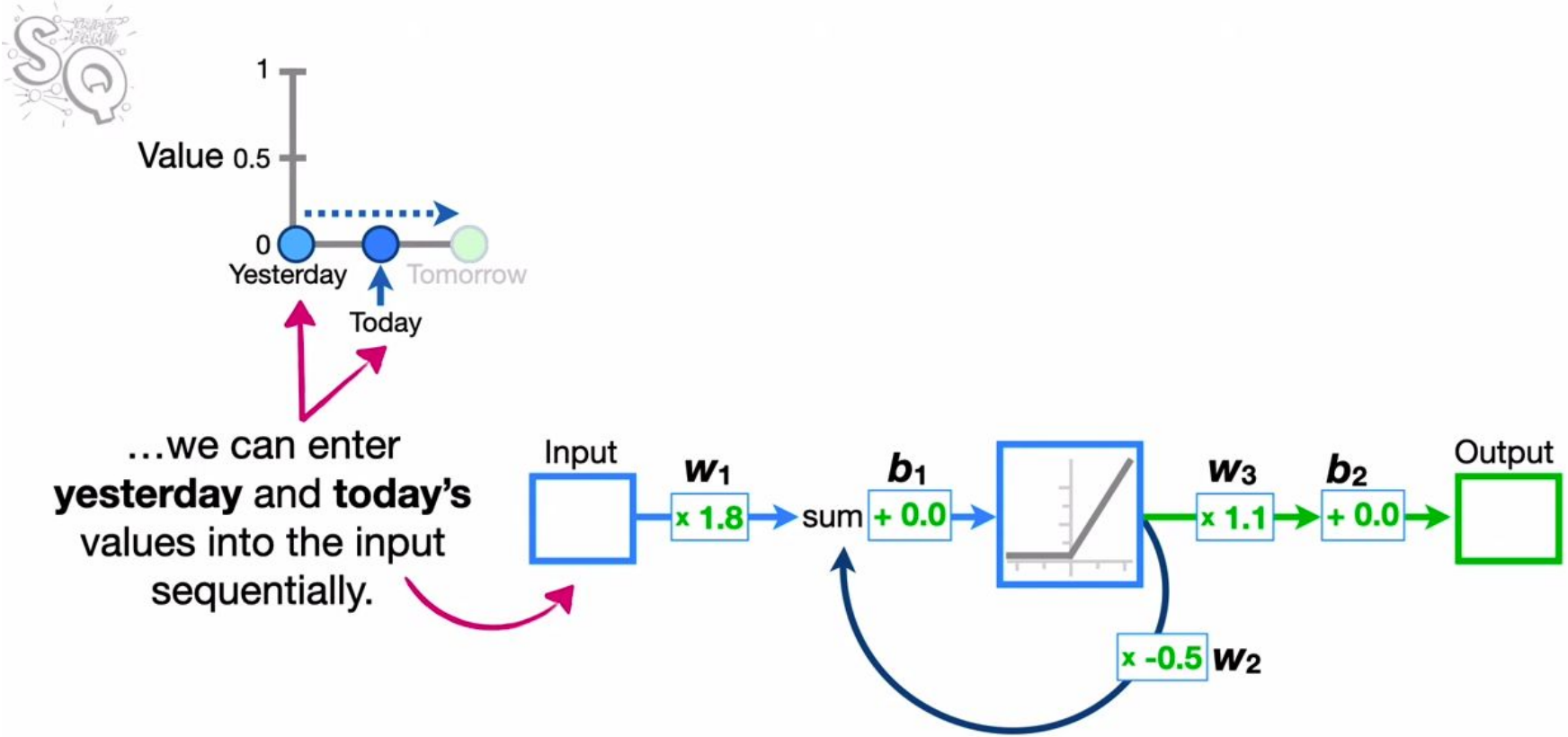
Now, because the recurrent neural network has a **feedback loop**…

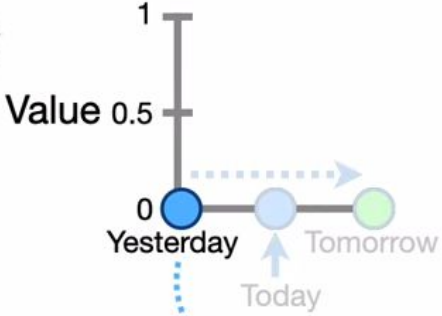




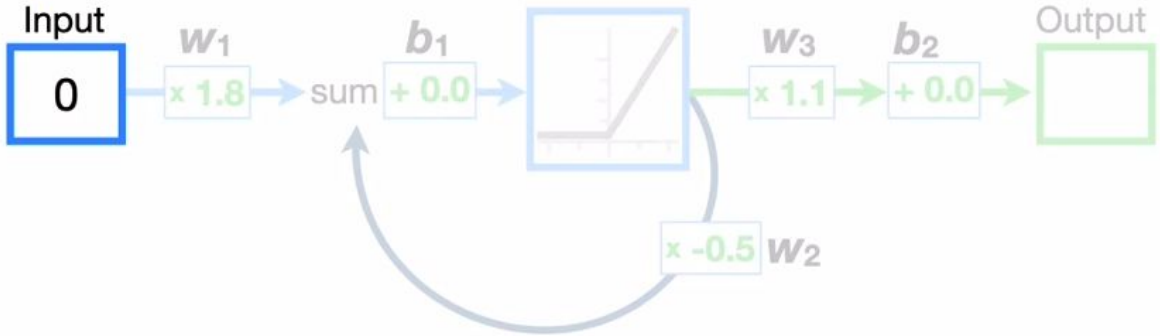
…we can enter **yesterday** and **today's** values into the input sequentially.

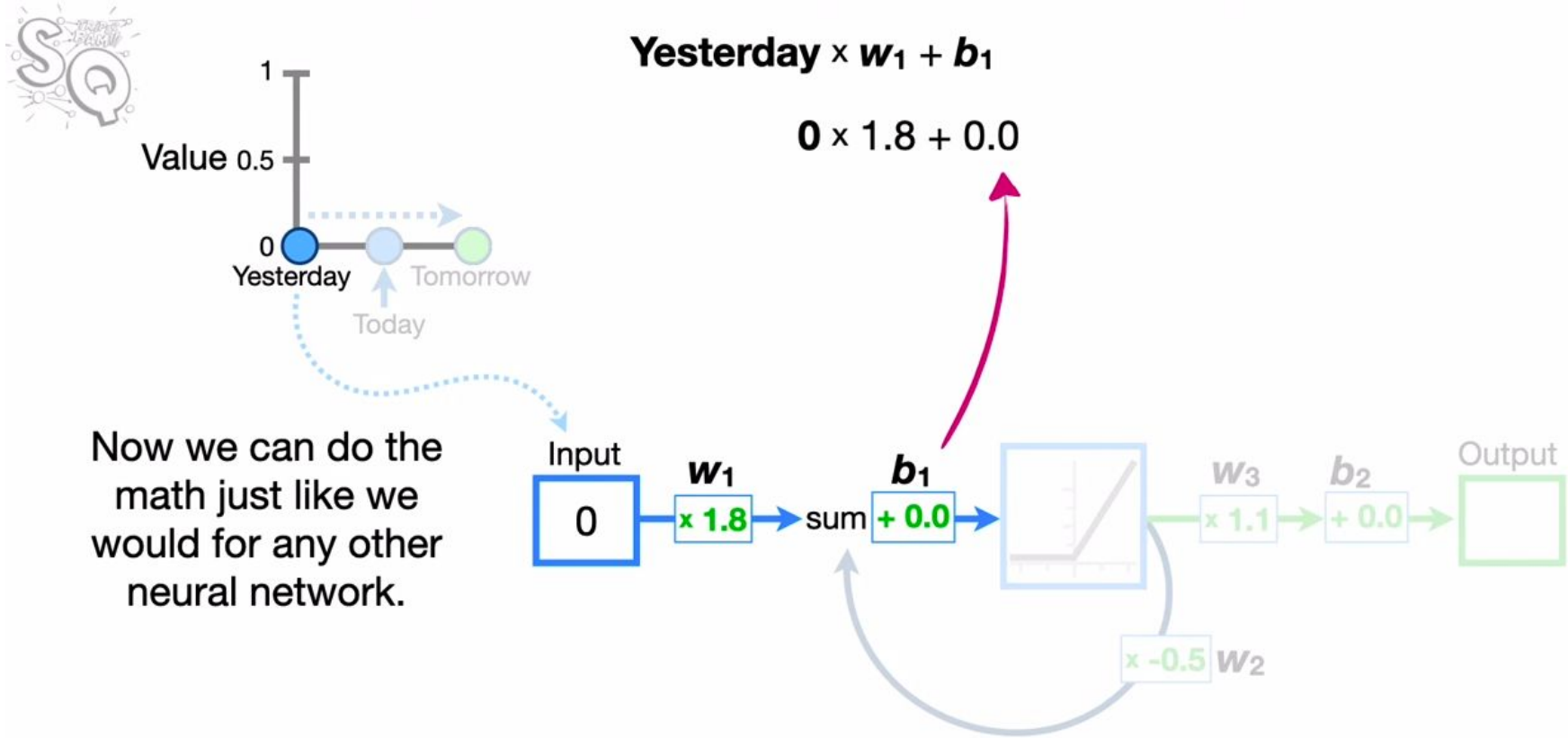


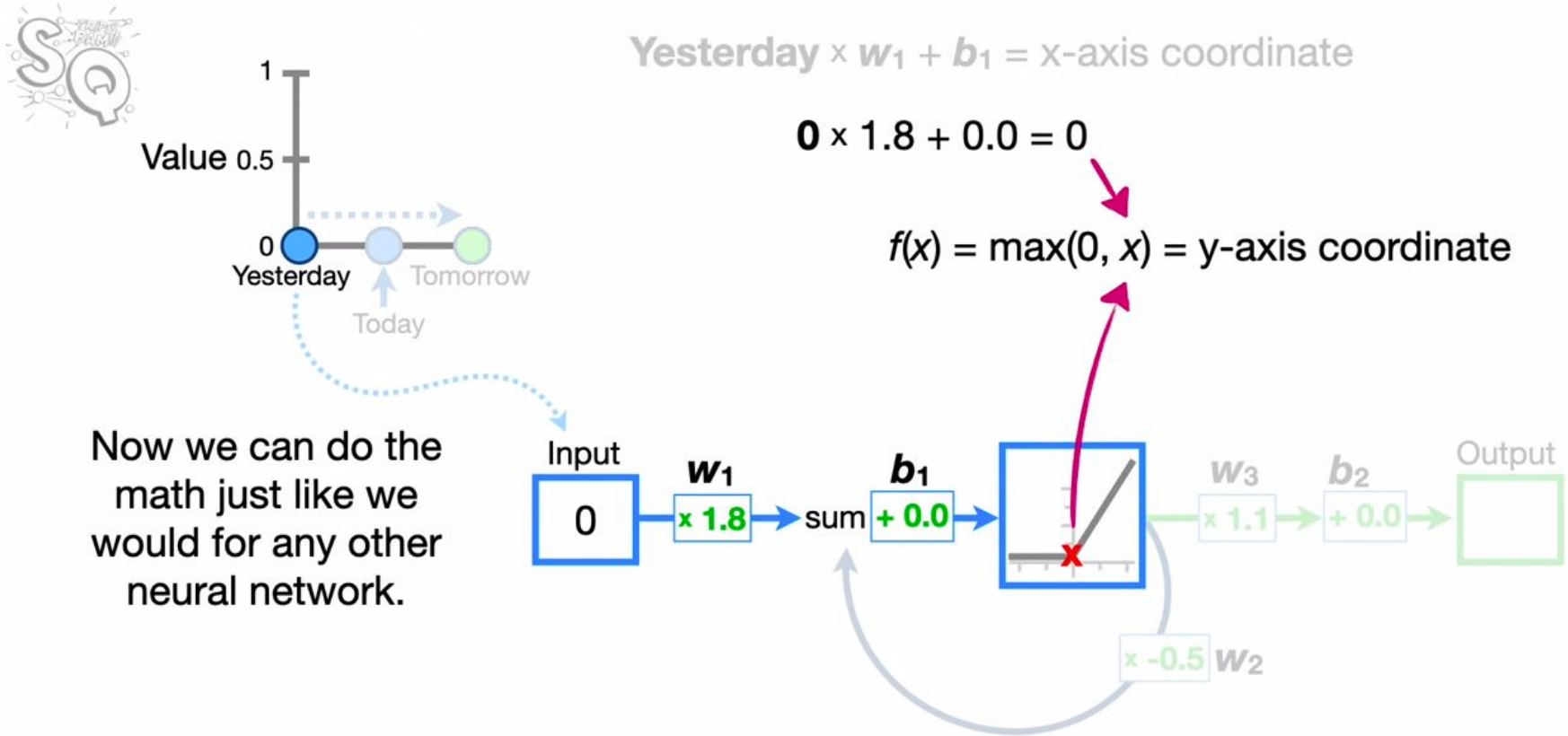

1
Value 0.5
0
Yesterday
Today
Tomorrow
…we can enter **yesterday** and **today's** values into the input sequentially.
Input
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output
x -0.5
w_2

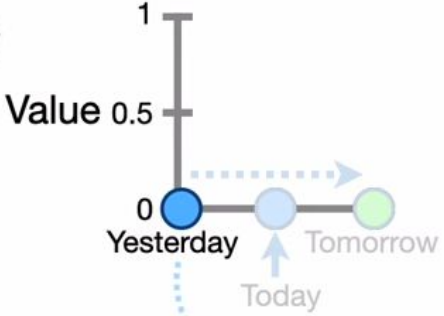


We'll start by plugging **yesterday's** value into the input.



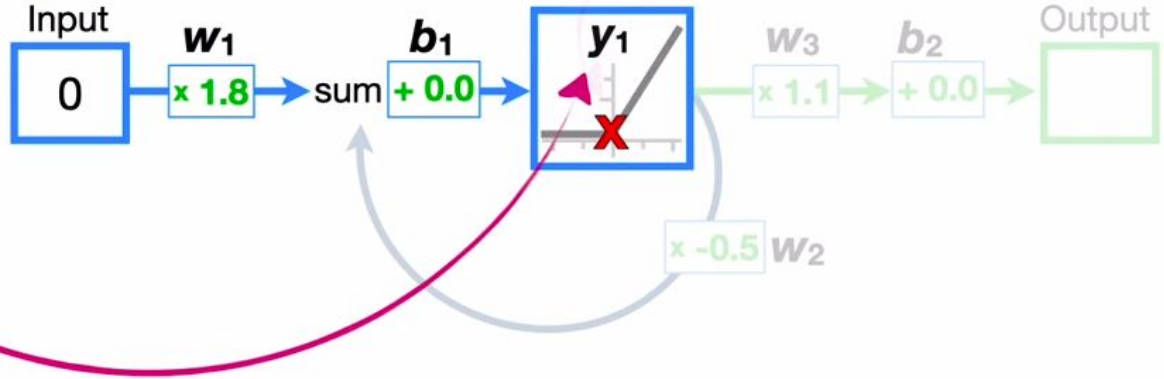

Yesterday x w_1 + b_1
0 x 1.8 + 0.0
1
Value 0.5
0
Yesterday
Tomorrow
Today
Now we can do the math just like we would for any other neural network.
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output
x -0.5 w_2

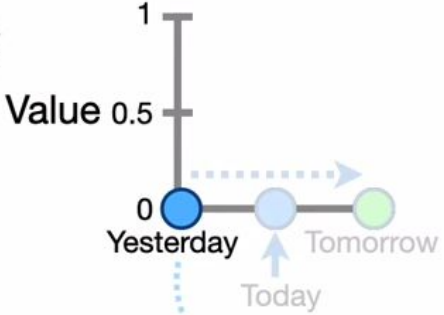

Yesterday x w_1 + b_1 = x-axis coordinate
1
Value 0.5
0
Yesterday
Tomorrow
Today
0 x 1.8 + 0.0 = 0
$f(x)$ = max(0, x) = y-axis coordinate
Now we can do the math just like we would for any other neural network.
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output
x -0.5
w_2



$f(0) = \max(0, 0) = 0 = y_1$

At this point, the output from the activation function, the y-axis coordinate that we'll call y_1…

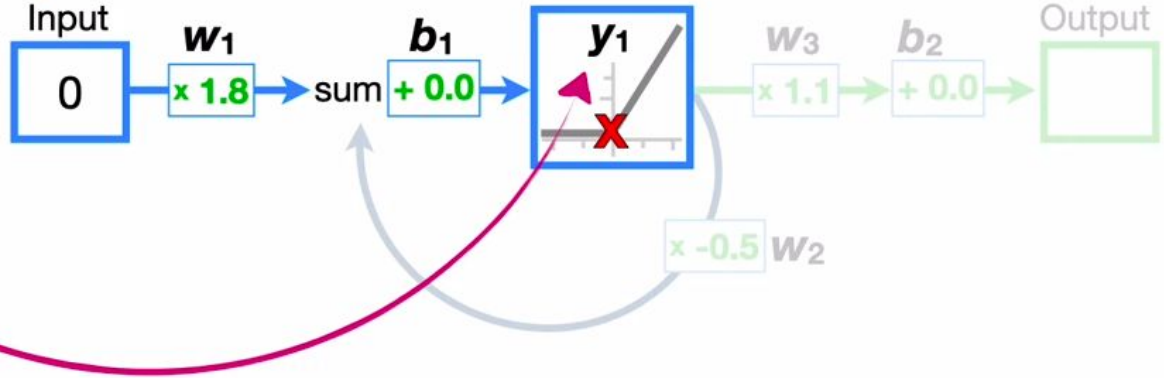


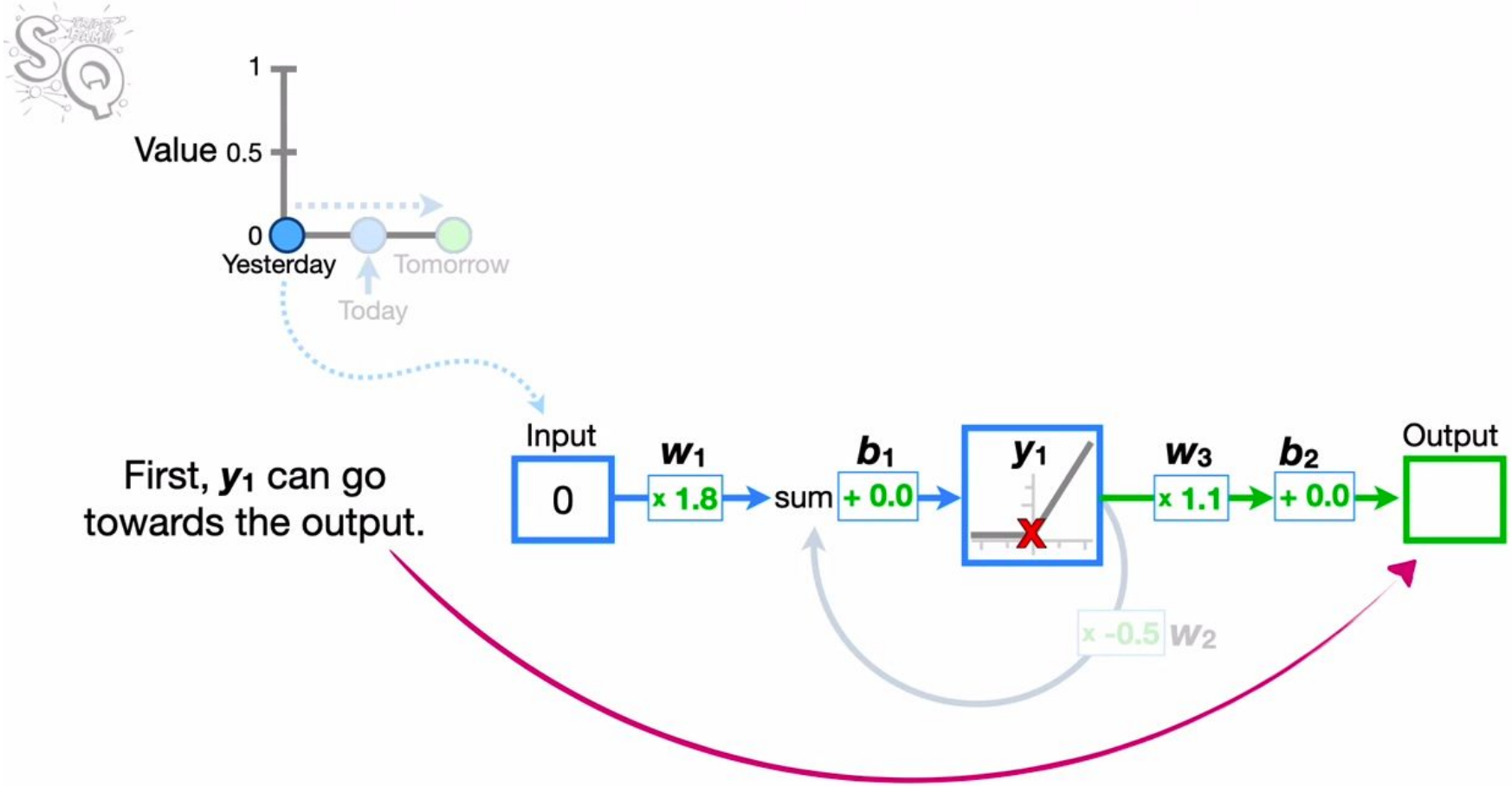


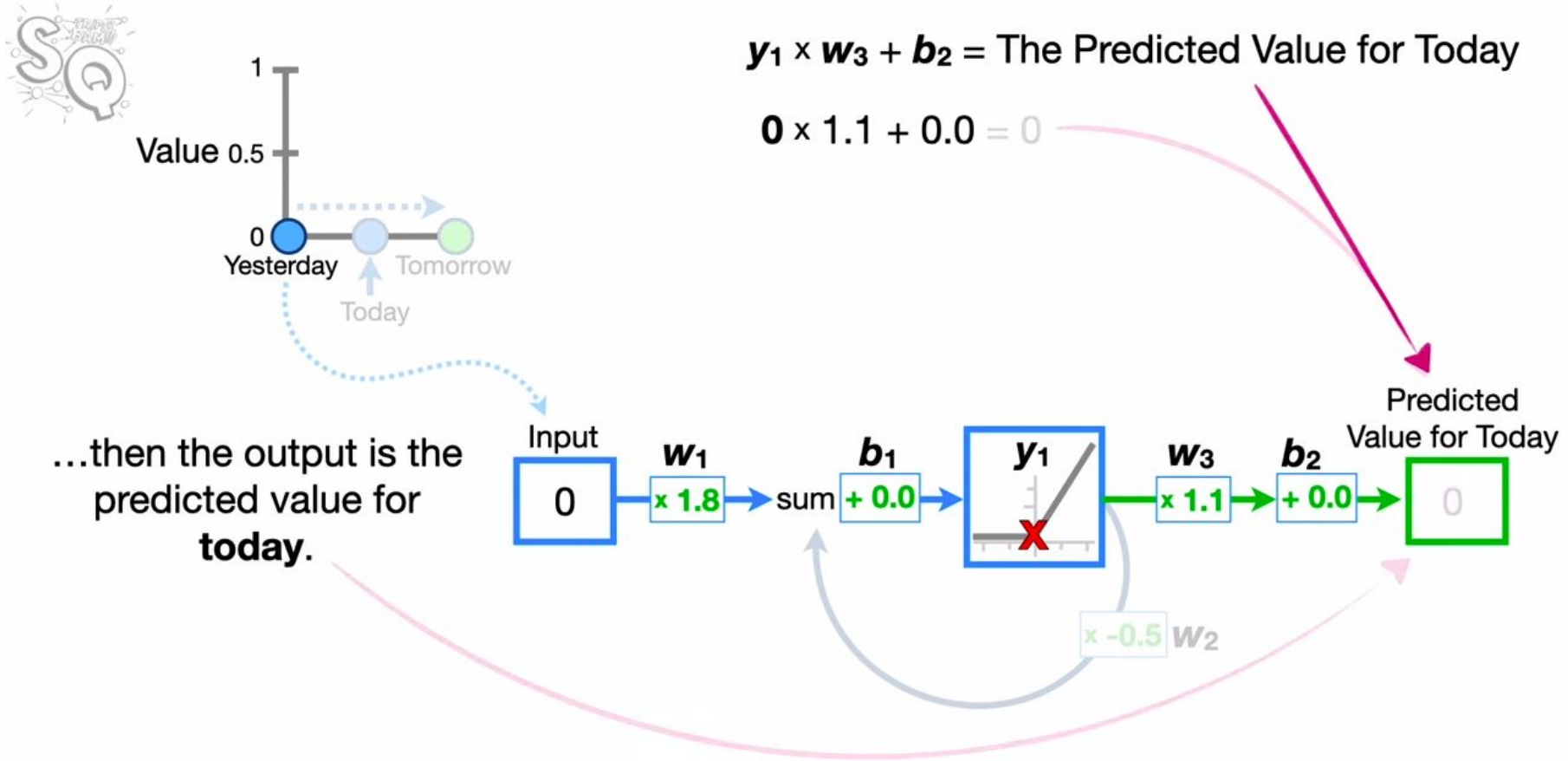
$f(0) = \max(0, 0) = 0 = y_1$

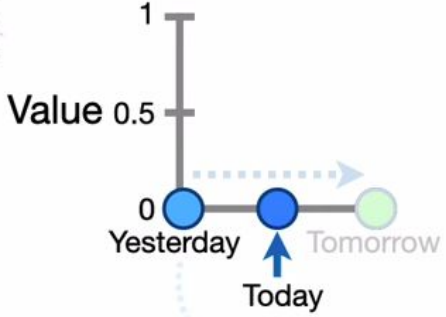
At this point, the output from the activation function, the y-axis coordinate that we'll call y_1…

…can go two places.




1
Value 0.5
0
Yesterday
Today
Tomorrow
First, y_1 can go towards the output.
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Output
x -0.5 w_2

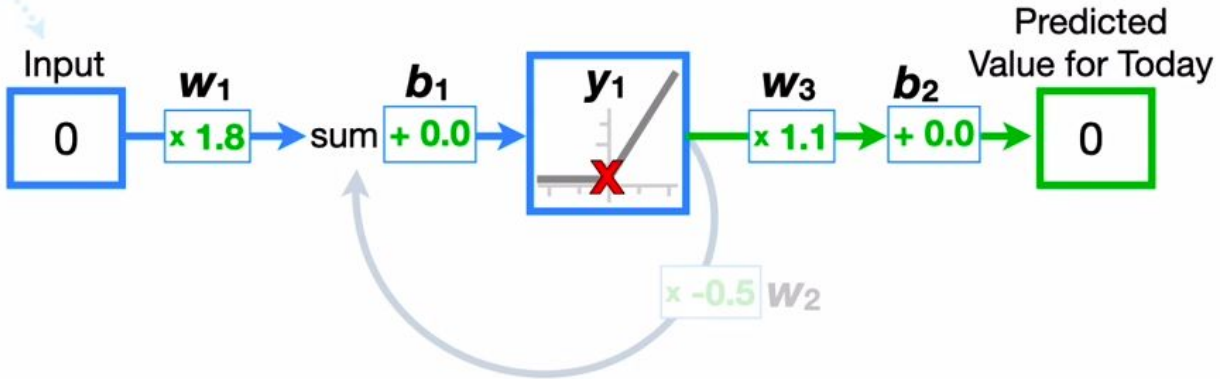

$y_1 \times w_3 + b_2$ = The Predicted Value for Today
0 x 1.1 + 0.0 = 0
1
Value 0.5
0
Yesterday
Today
Tomorrow
…then the output is the predicted value for **today**.
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
0
x -0.5 w_2

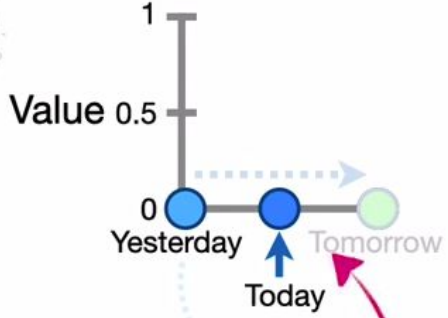


$y_1 \times w_3 + b_2$ = The Predicted Value for Today

0 × 1.1 + 0.0 = 0

However, we're not interested in the *predicted* value for **today** because we already have the *actual* value for **today**.

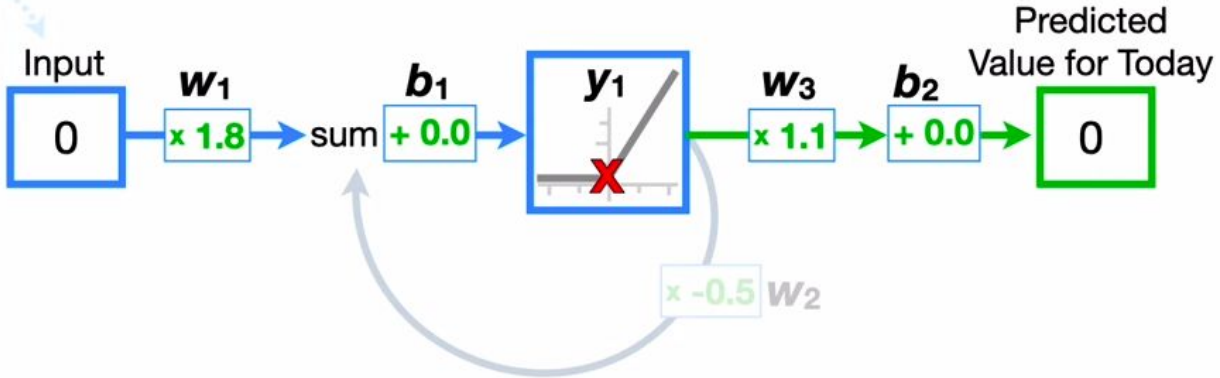




$y_1 \times w_3 + b_2$ = The Predicted Value for Today

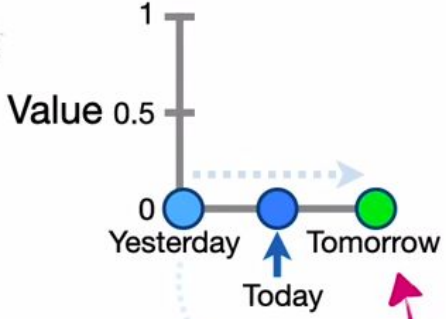
0 × 1.1 + 0.0 = 0

However, we're not interested in the *predicted* value for **today** because we already have the *actual* value for **today**.

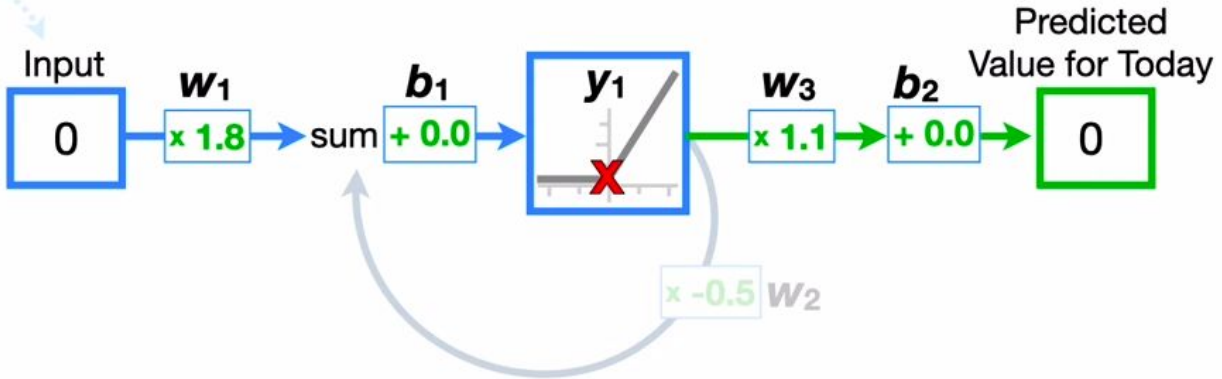


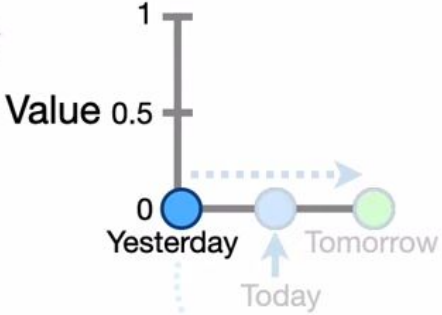
$y_1 \times w_3 + b_2 = \text{The Predicted Value for Today}$

$0 \times 1.1 + 0.0 = 0$

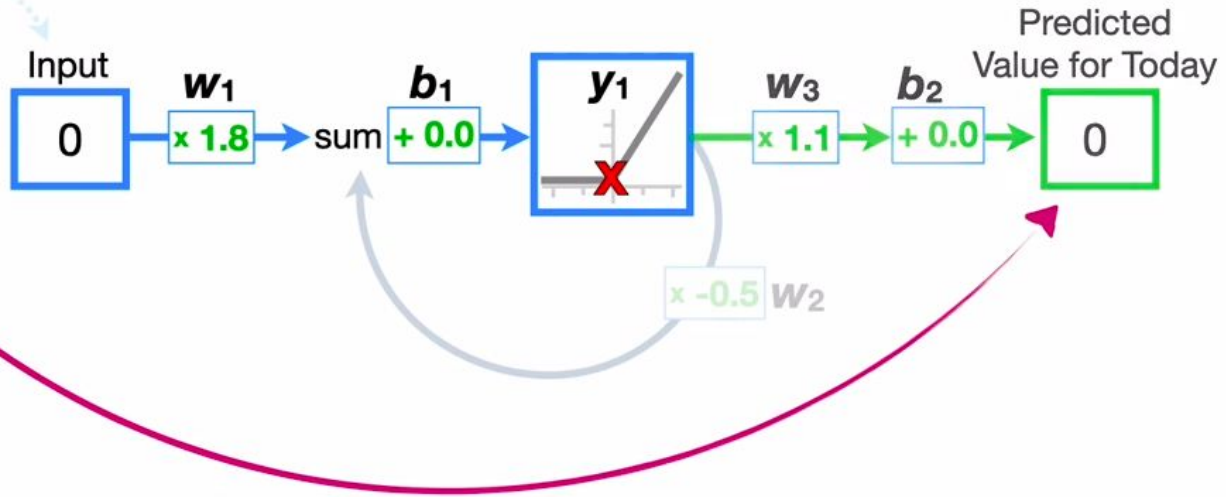


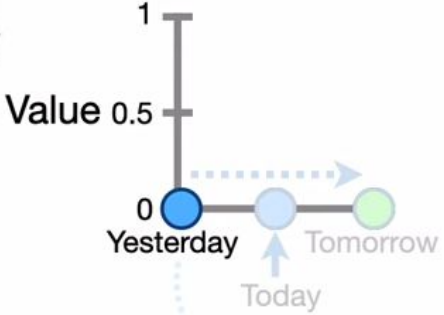
Instead, we want to use both **yesterday** and **today's** value to predict **tomorrow's** value.



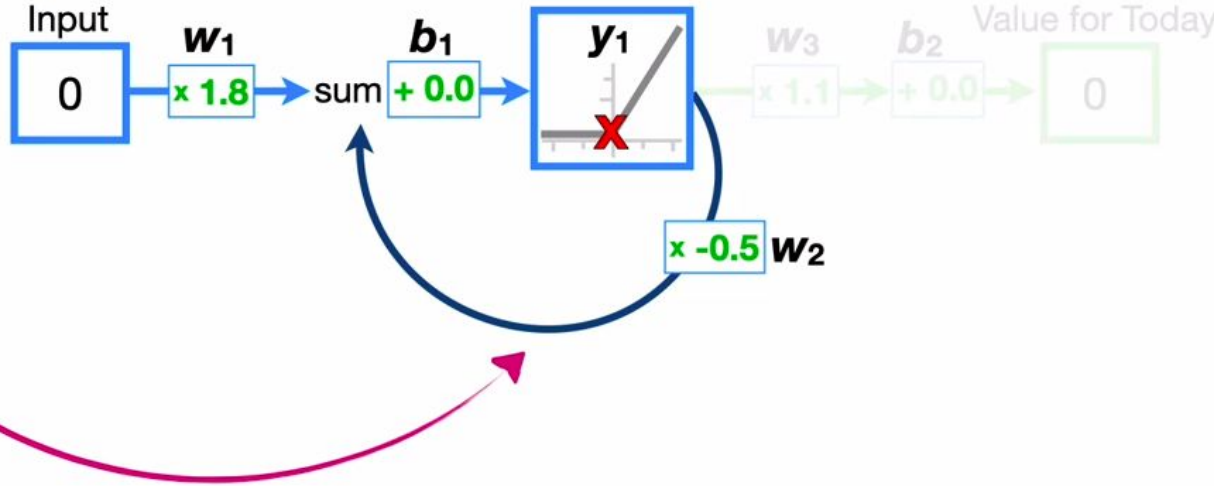


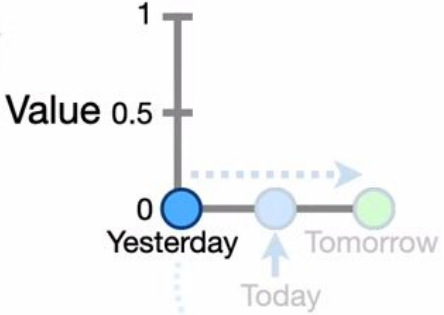
So, for now, we'll ignore this output…



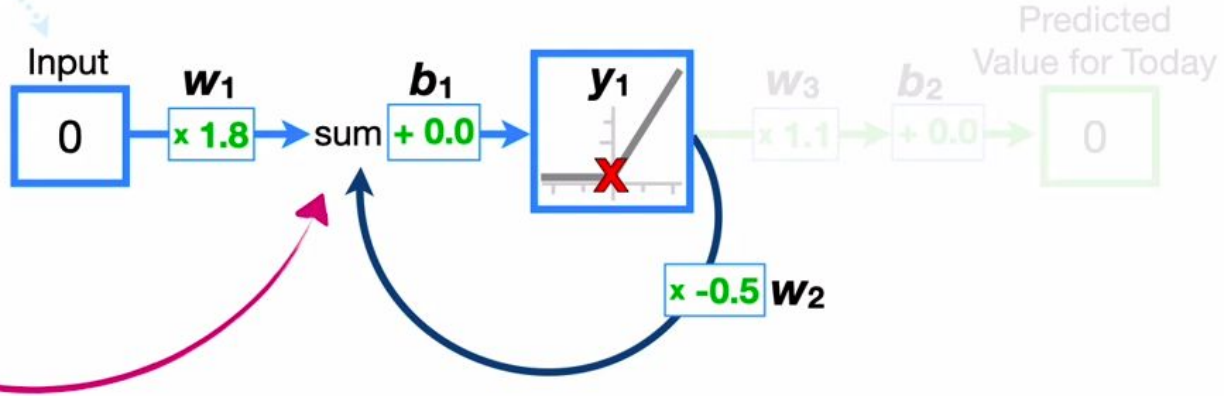


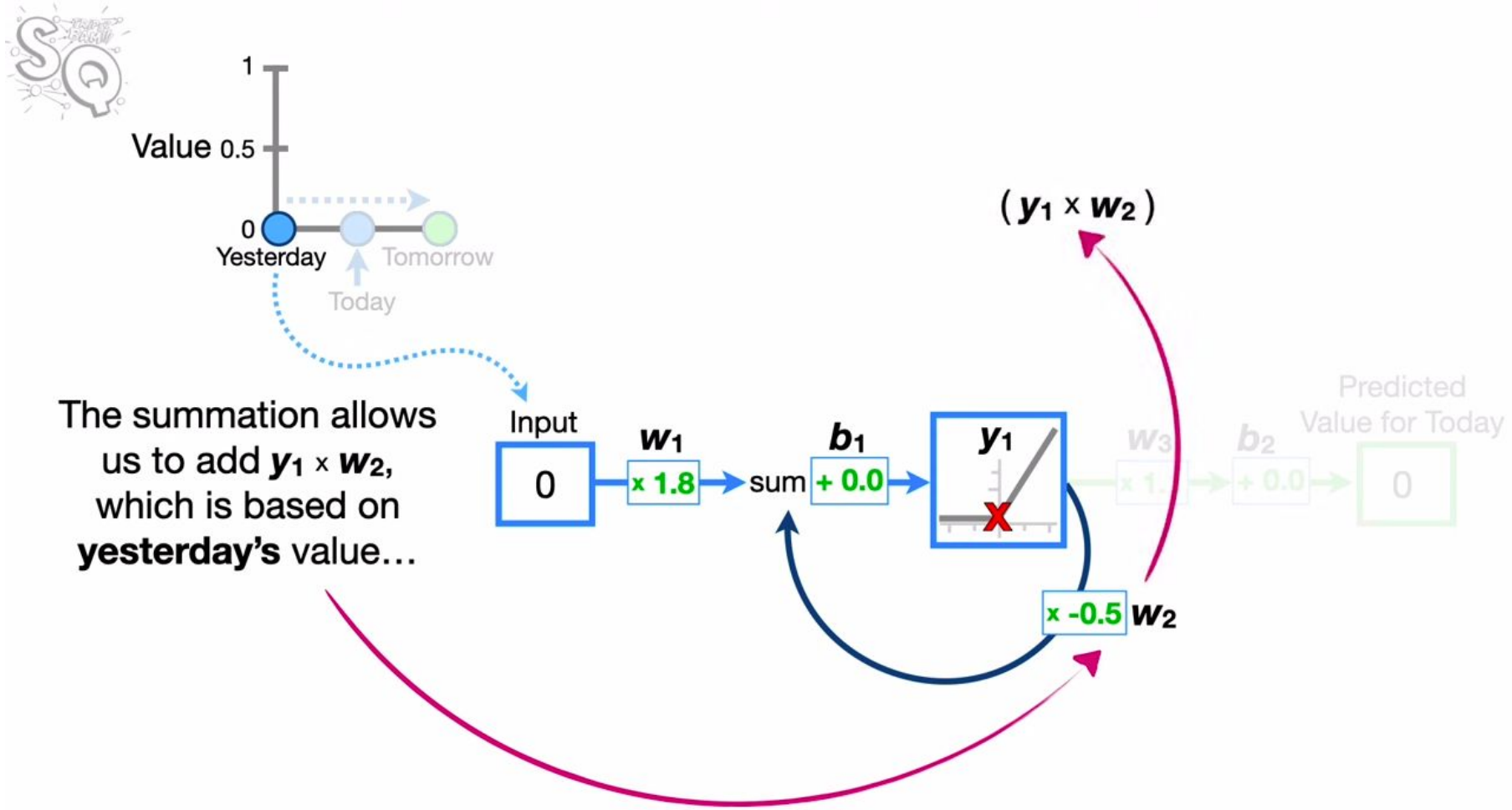
…and, instead, focus on what happens with this **feedback loop**.

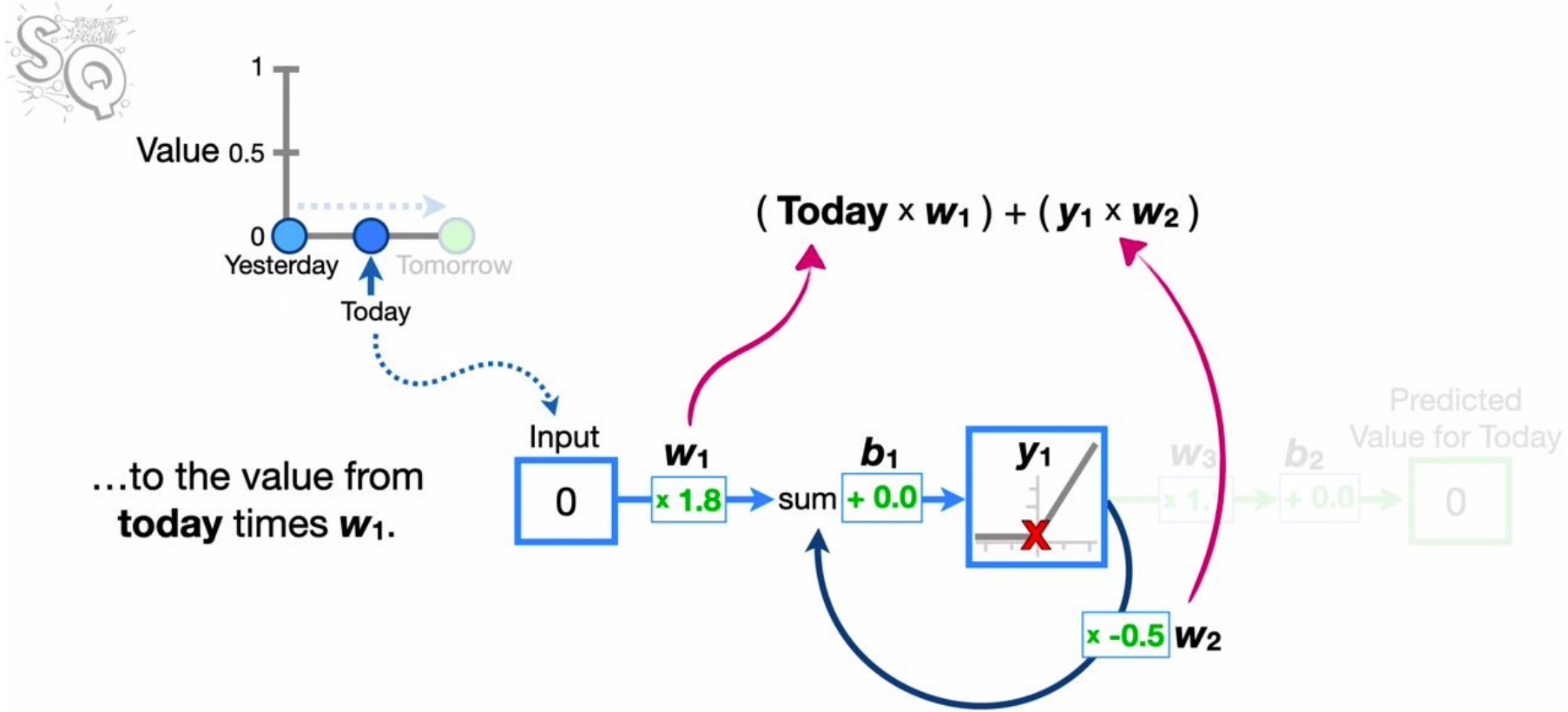


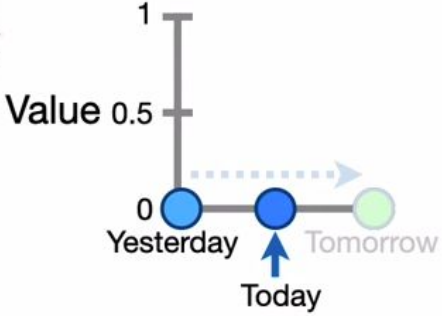


The key to understanding how the feedback loop works is this summation.

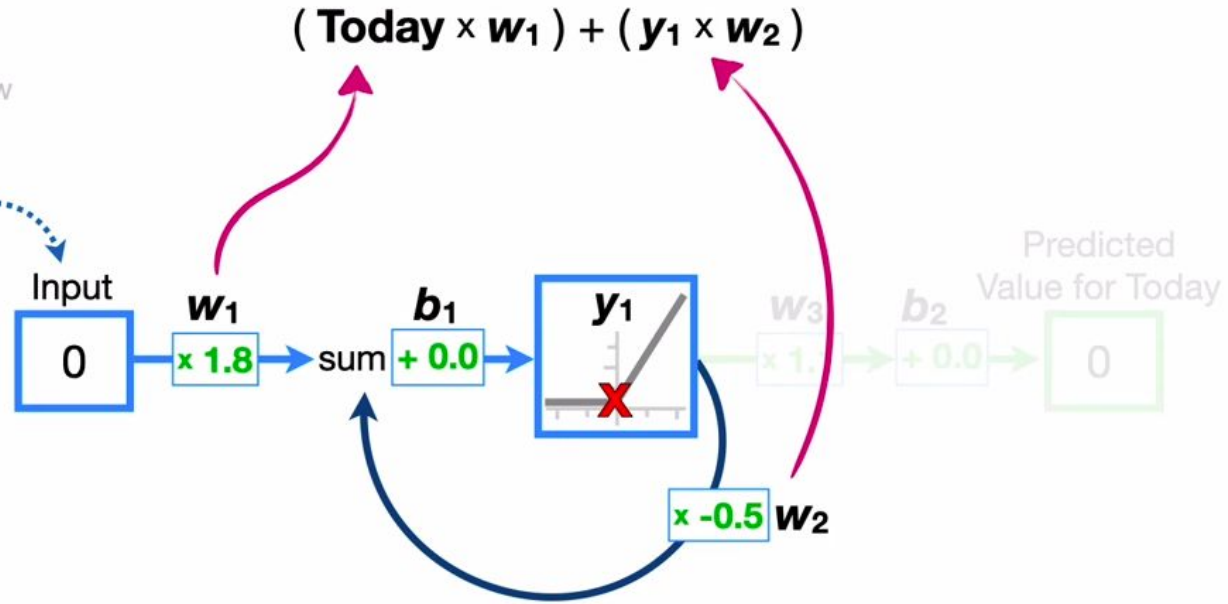


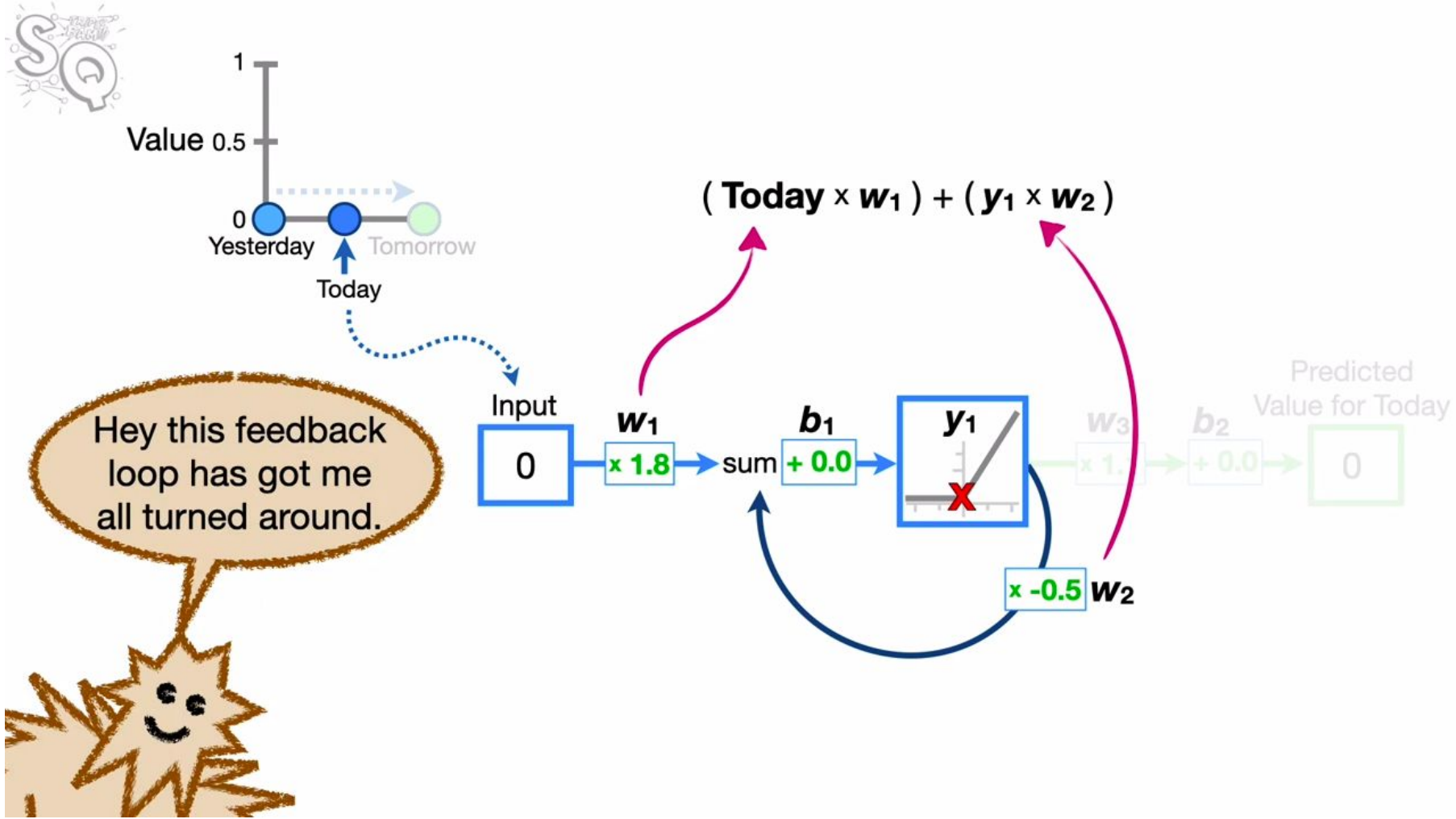

1
Value 0.5
0
Yesterday
Today
Tomorrow
The summation allows us to add $y_1 \times w_2$, which is based on **yesterday's** value…
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
$(y_1 \times w_2)$
x -0.5
w_2
w_3
b_2
Predicted Value for Today
0

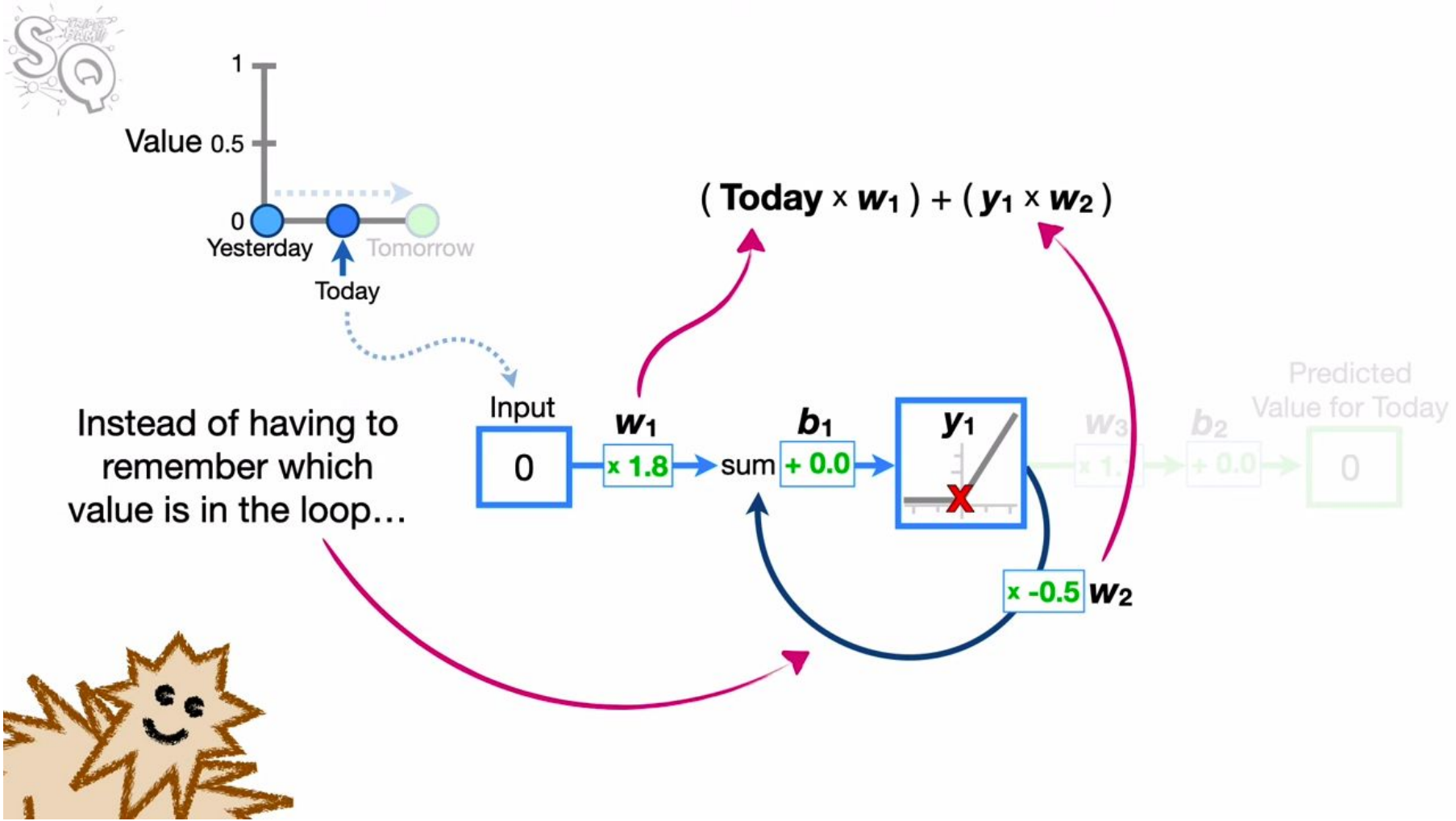

1
Value 0.5
0
Yesterday
Today
Tomorrow
…to the value from **today** times w_1.
(Today × w_1) + (y_1 × w_2)
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
b_2
Predicted Value for Today
0
x -0.5
w_2

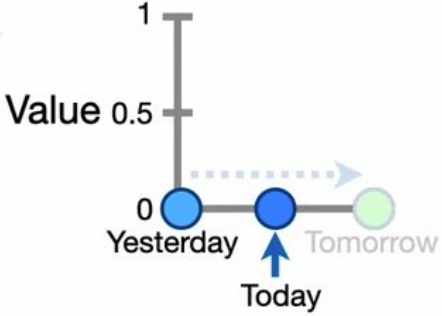


In other words, the **feedback loop** allows both **yesterday** and **today's** values to influence the prediction.

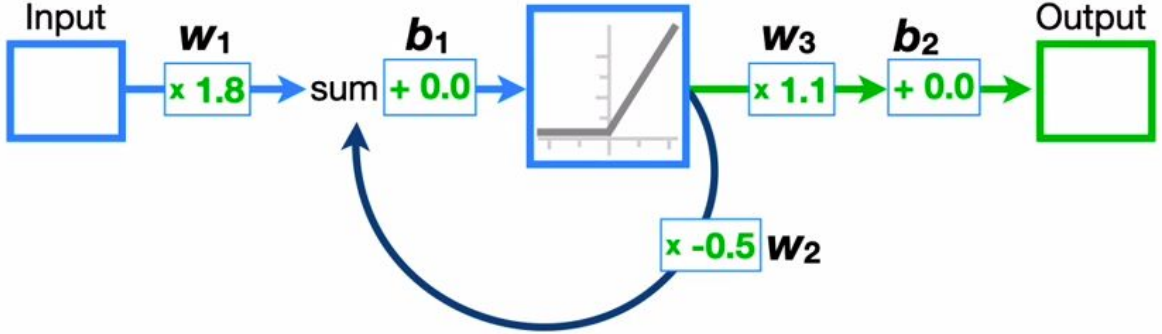



1
Value 0.5
0
Yesterday
Today
Tomorrow
(Today × w_1) + (y_1 × w_2)
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
b_2
+ 0.0
Predicted Value for Today
0
x -0.5
w_2
Hey this feedback loop has got me all turned around.

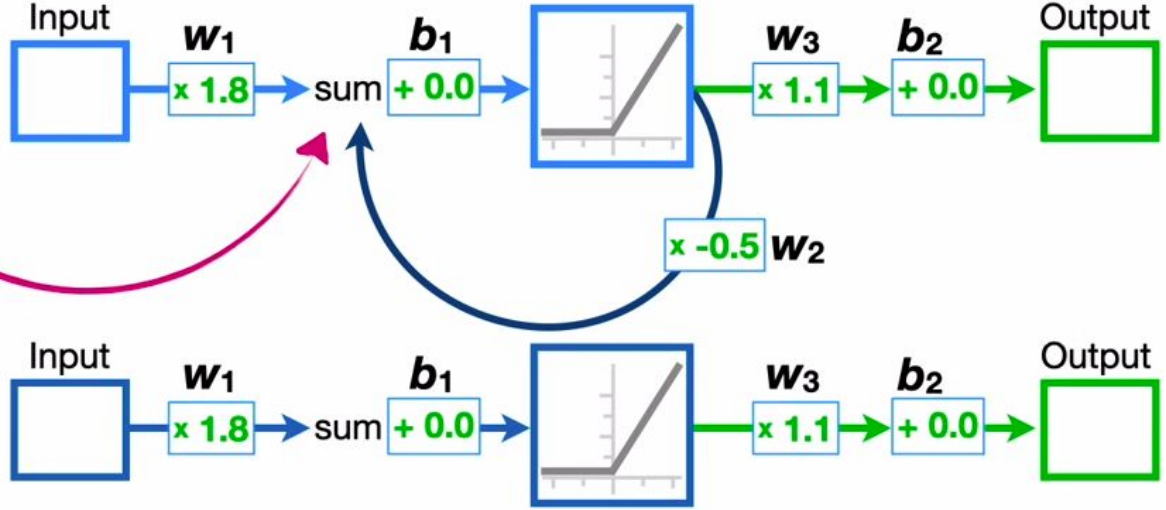

1
Value 0.5
0
Yesterday
Today
Tomorrow
(Today × w_1) + (y_1 × w_2)
Instead of having to remember which value is in the loop…
Input
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
x -0.5
w_2
w_3
b_2
+ 0.0
Predicted Value for Today
0



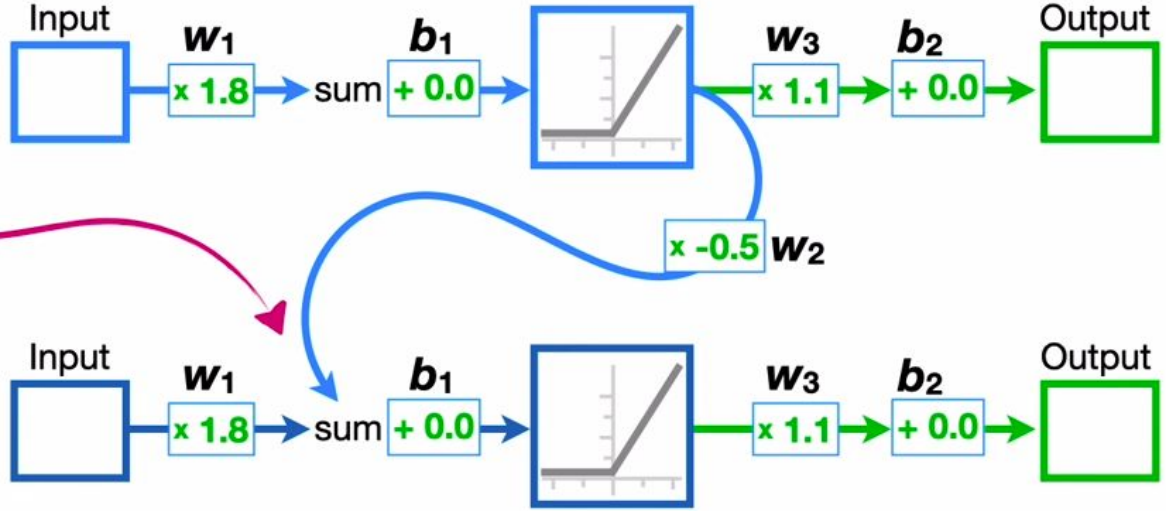
…we can **unroll** the feedback loop by making a copy of the neural network for each input value.



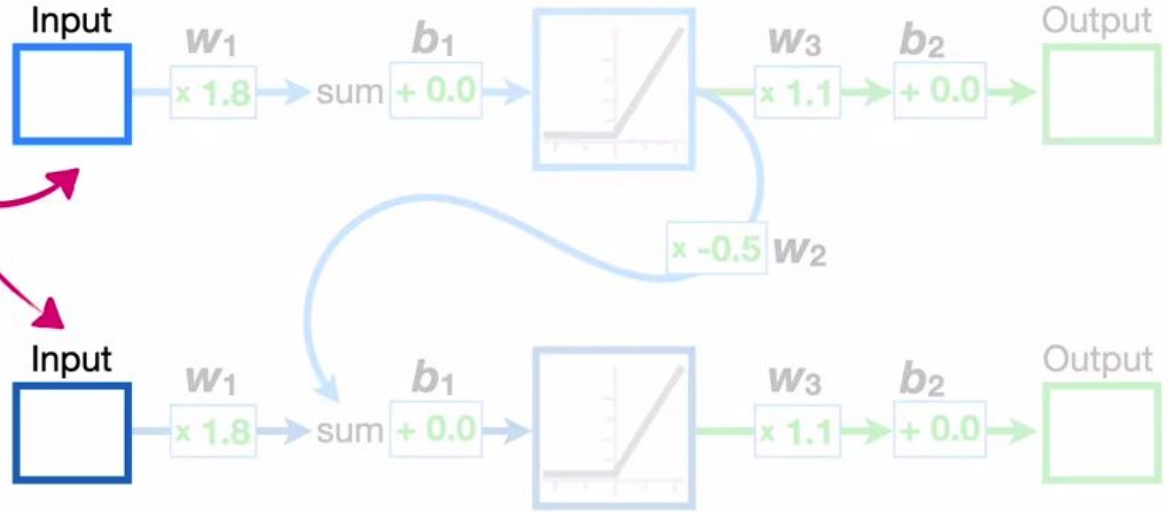
Now, instead of pointing the **feedback loop** to the sum in the first copy…

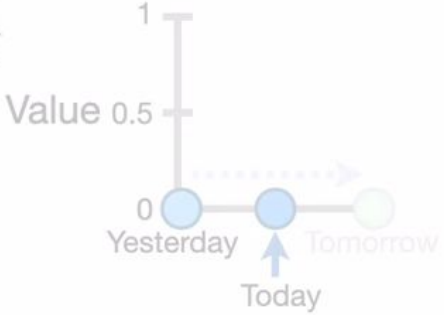


…we can point it to the sum in the second copy.

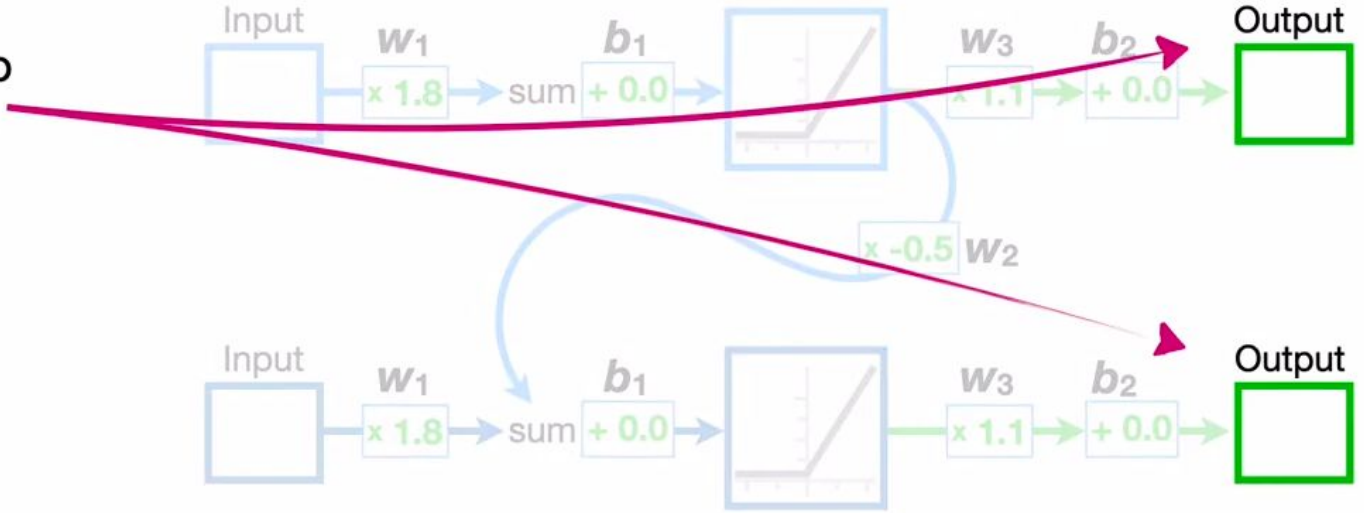


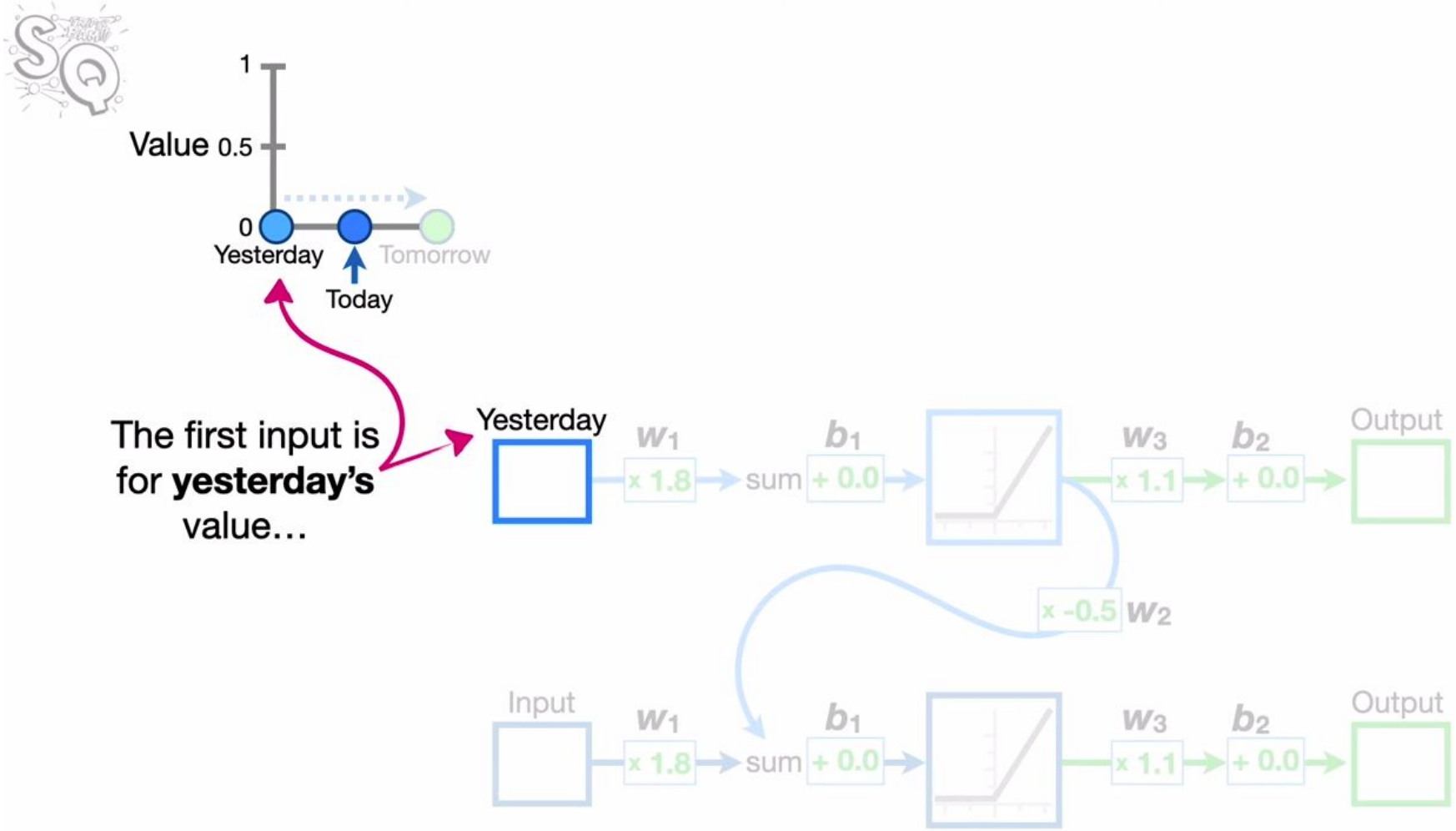
By **unrolling** the recurrent neural network, we end up with an new network that has two inputs…

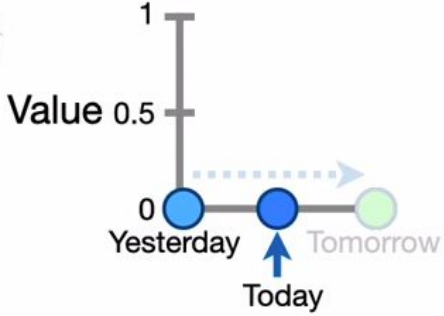




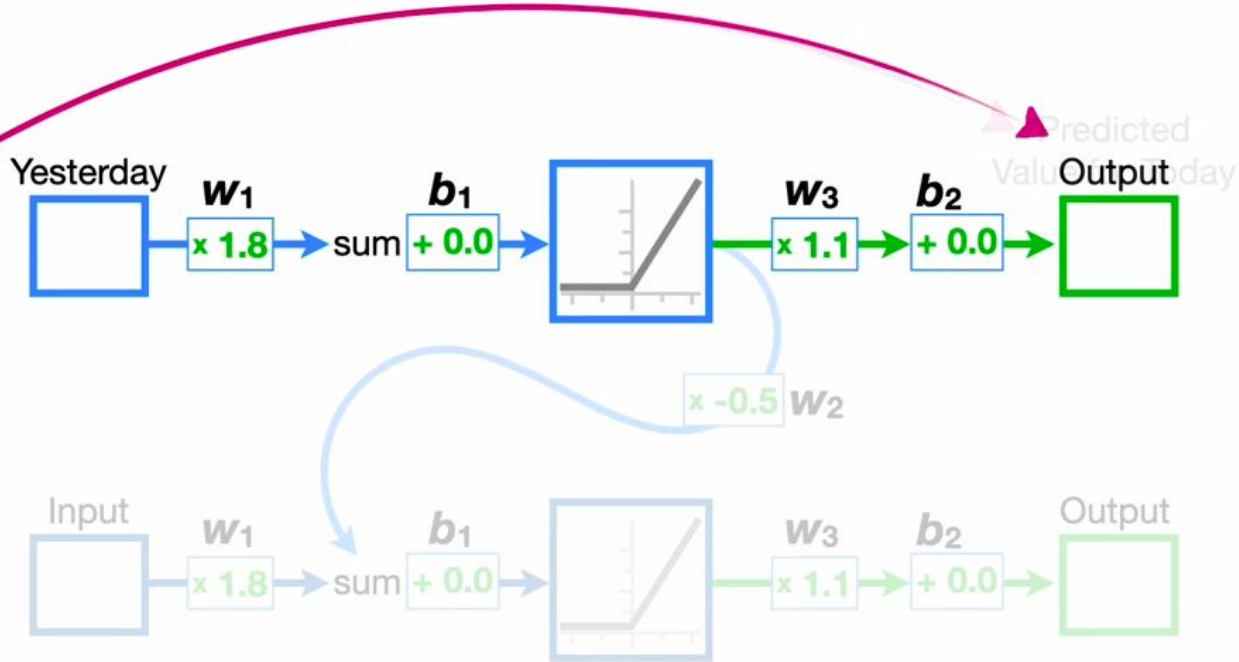
…and two outputs.

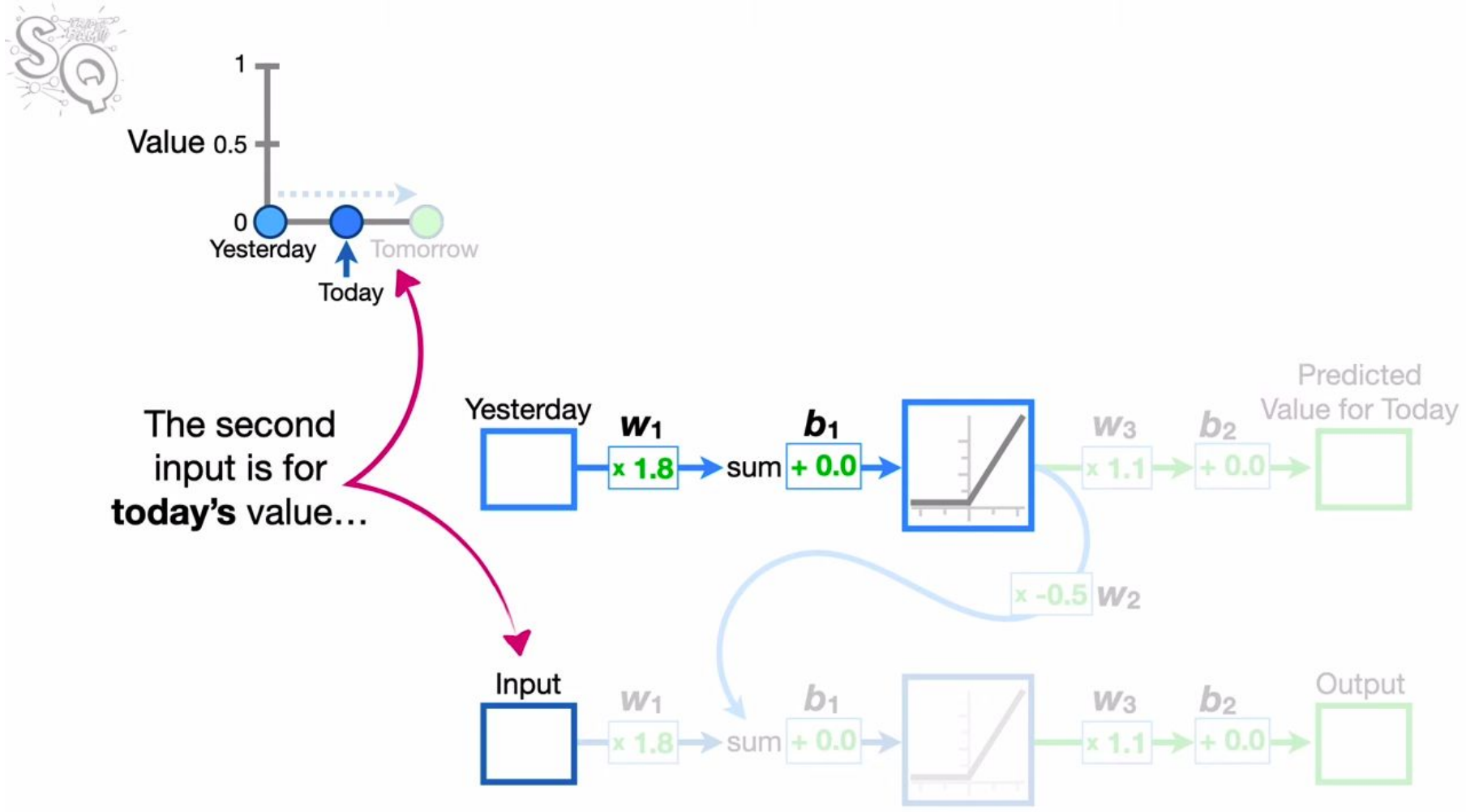


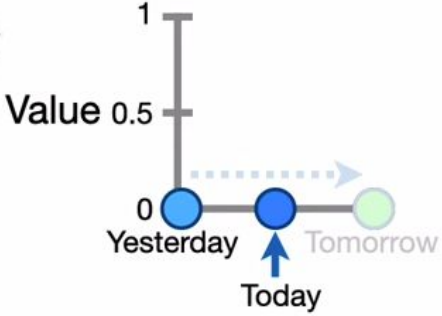

1
Value 0.5
0
Yesterday
Today
Tomorrow
The first input is for **yesterday's** value…
Yesterday
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output
x -0.5 w_2
Input
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output



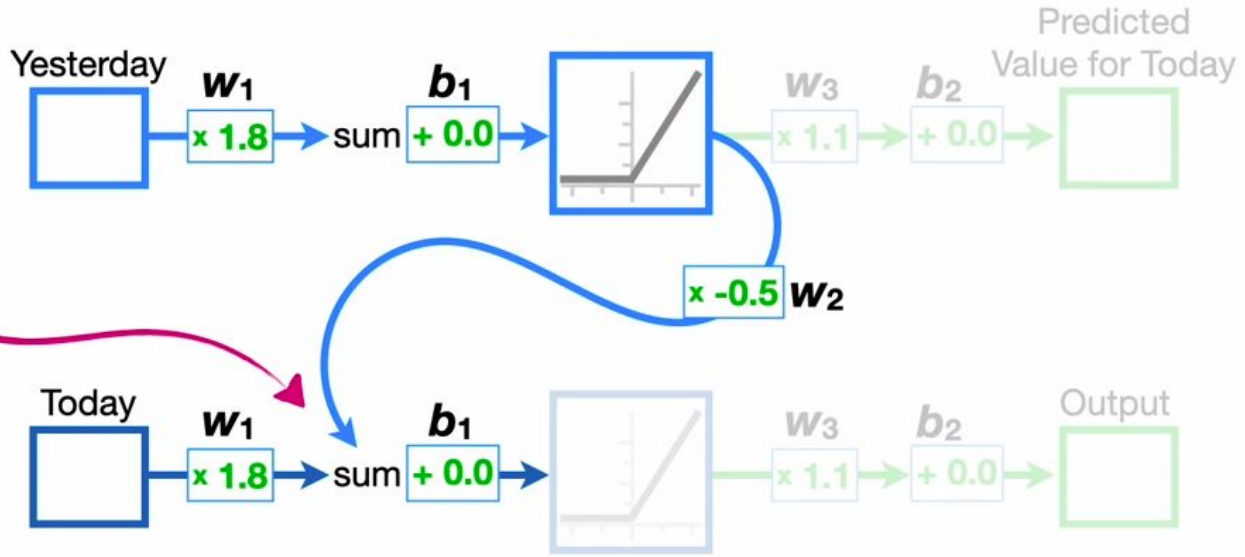
…we get the predicted value for **today**.

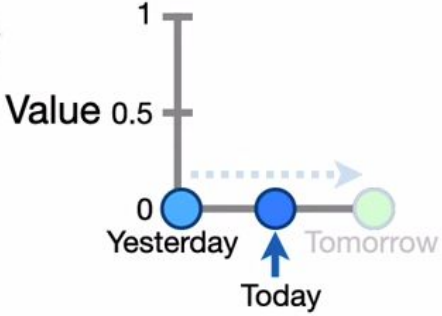



1
Value 0.5
0
Yesterday
Tomorrow
Today
The second input is for **today's** value…
Yesterday
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Input
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Output

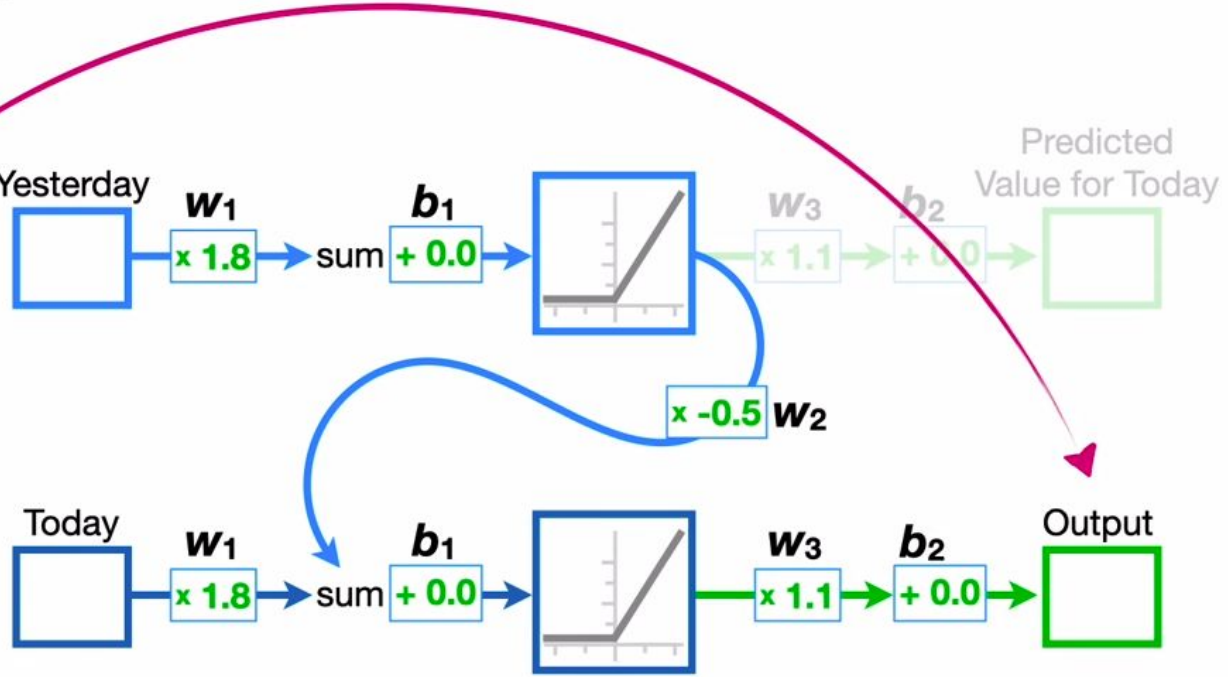


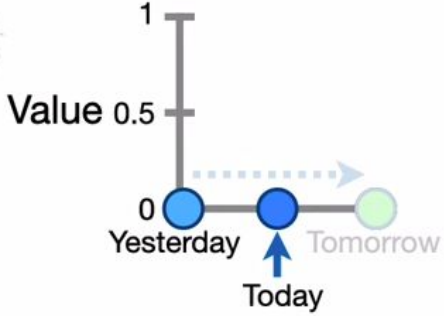
…and the connection between the first activation function and the second summation…



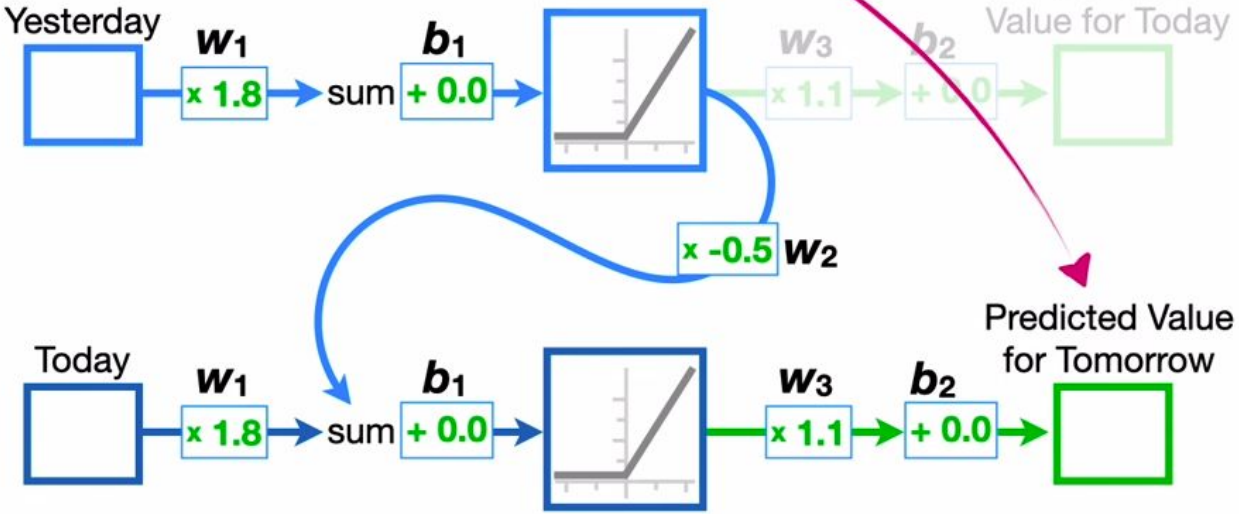


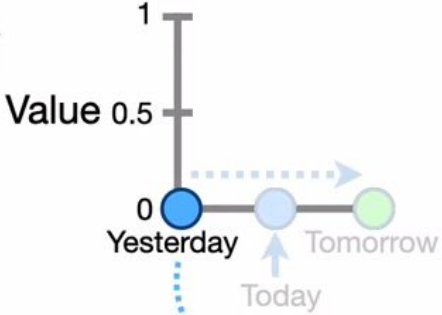
…allows both **yesterday** and **today's** values to influence the final output…



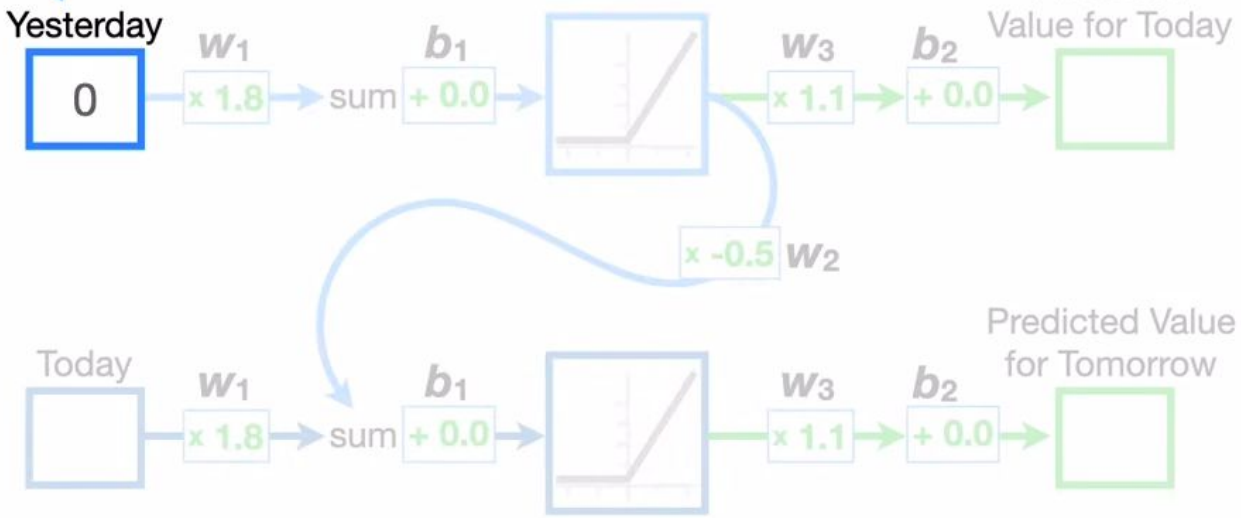


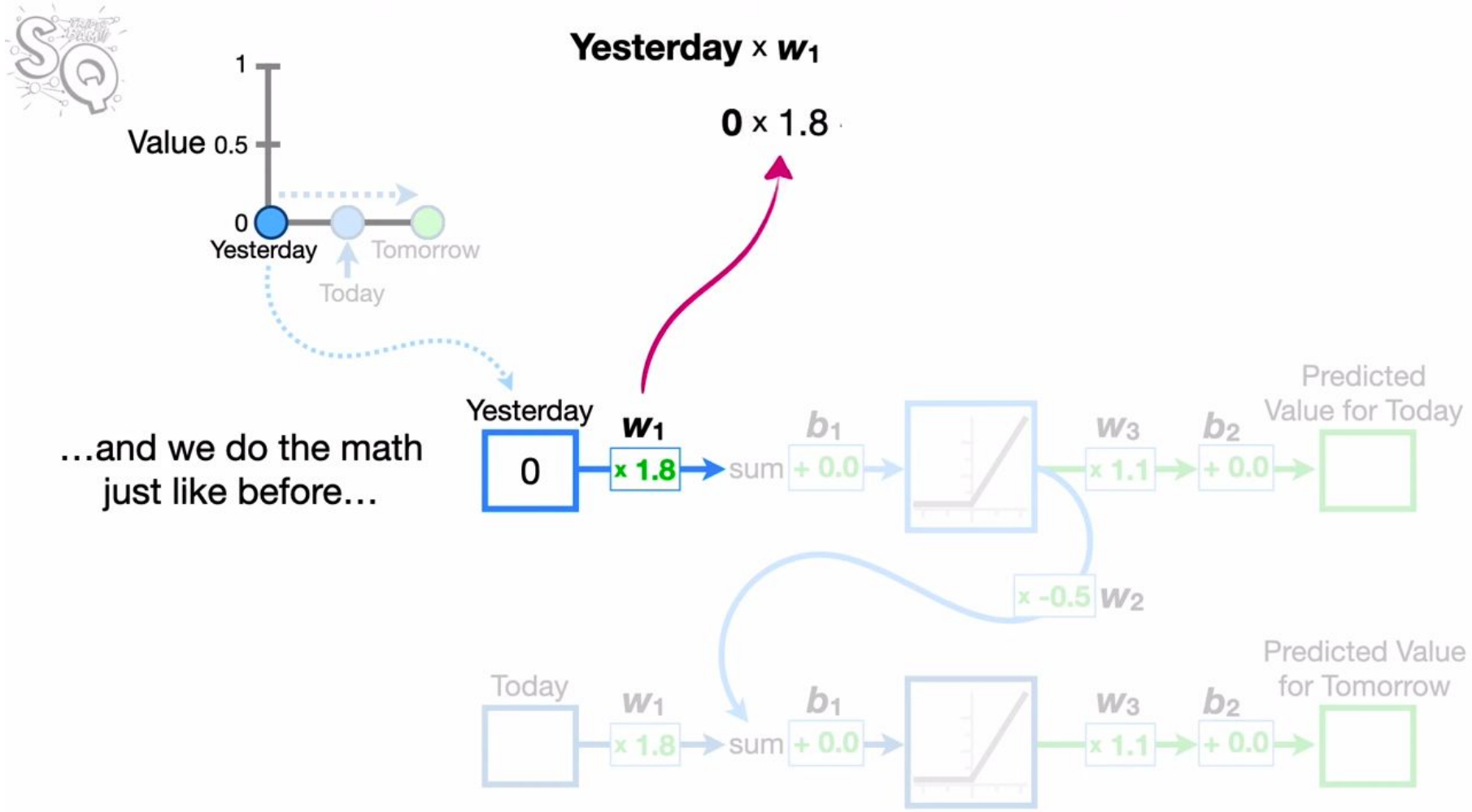
…which gives us the predicted value for **tomorrow**.

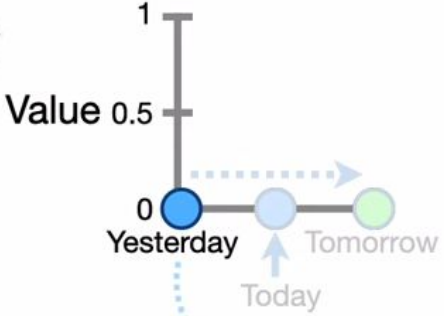




Now, when we put **yesterday's** value into the first input…

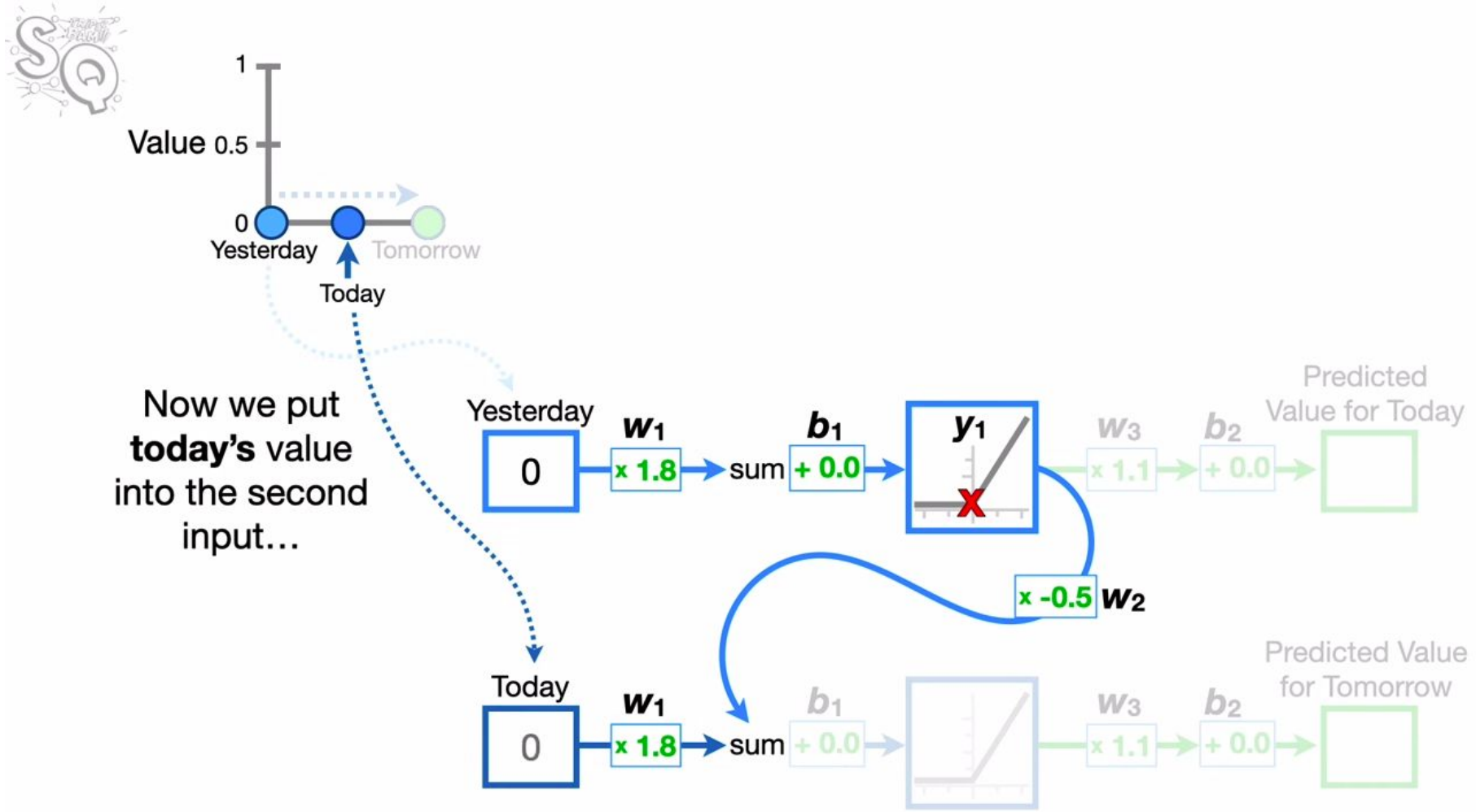


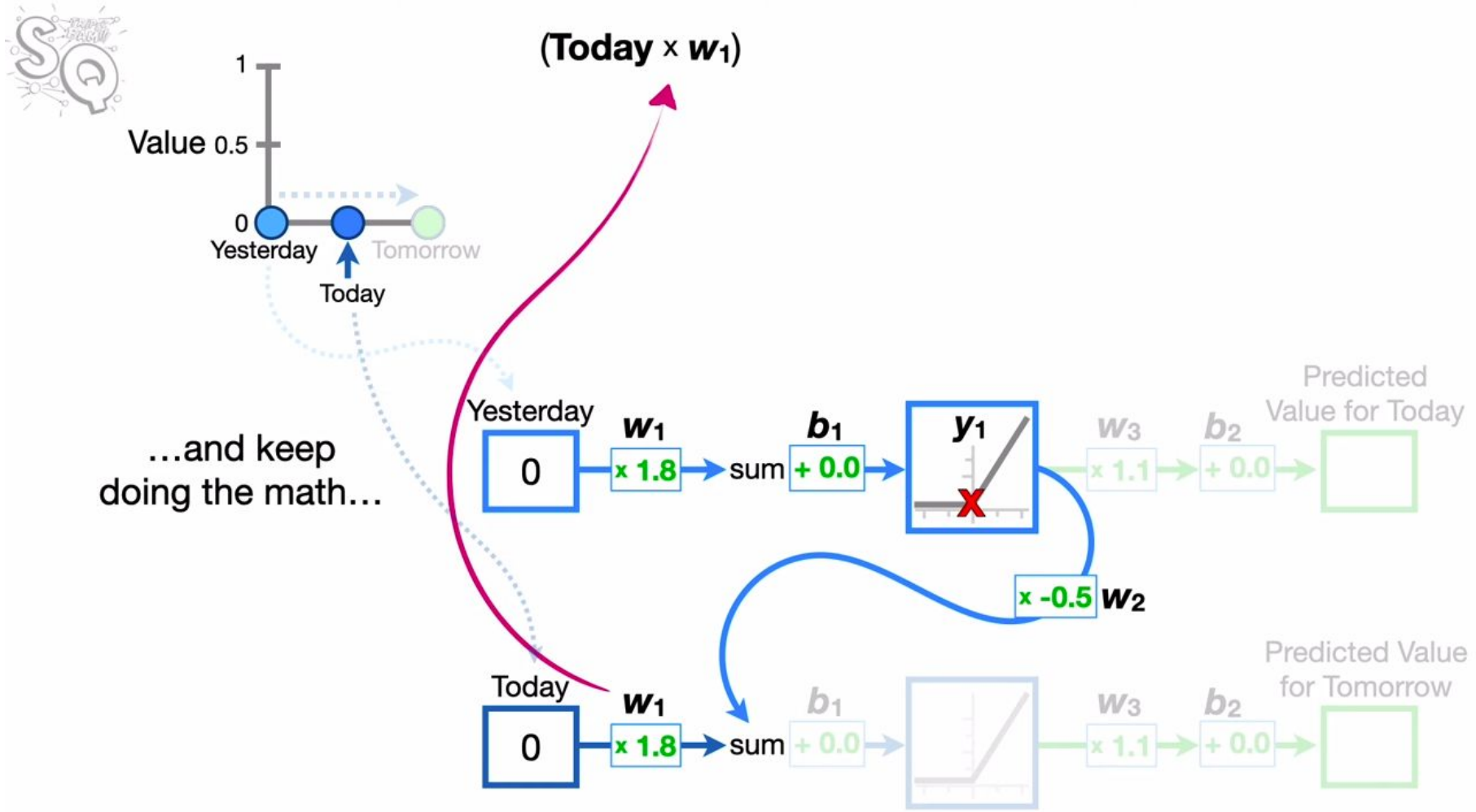

Yesterday x w_1
0 x 1.8
1
Value 0.5
0
Yesterday
Tomorrow
Today
…and we do the math just like before…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

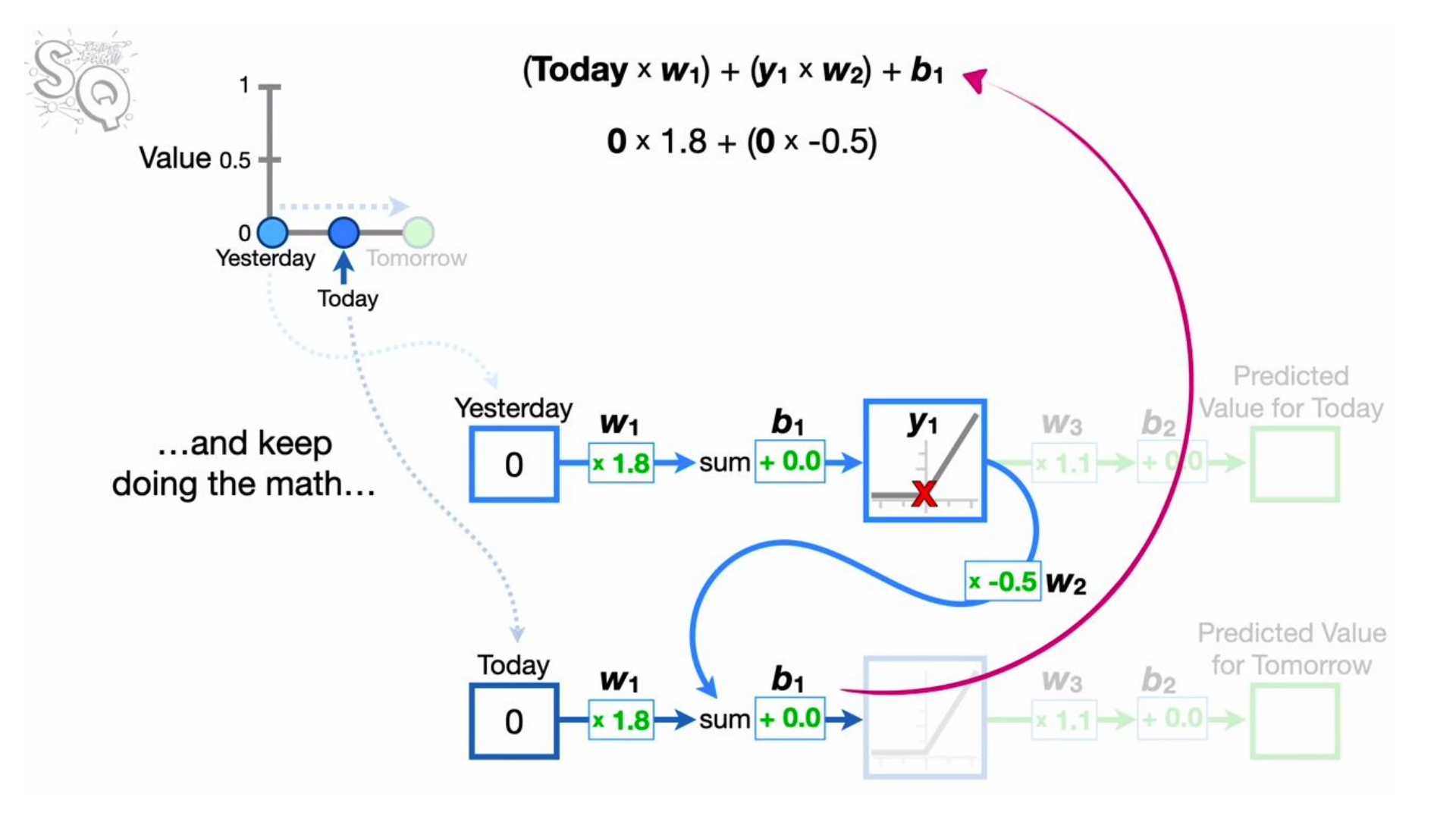


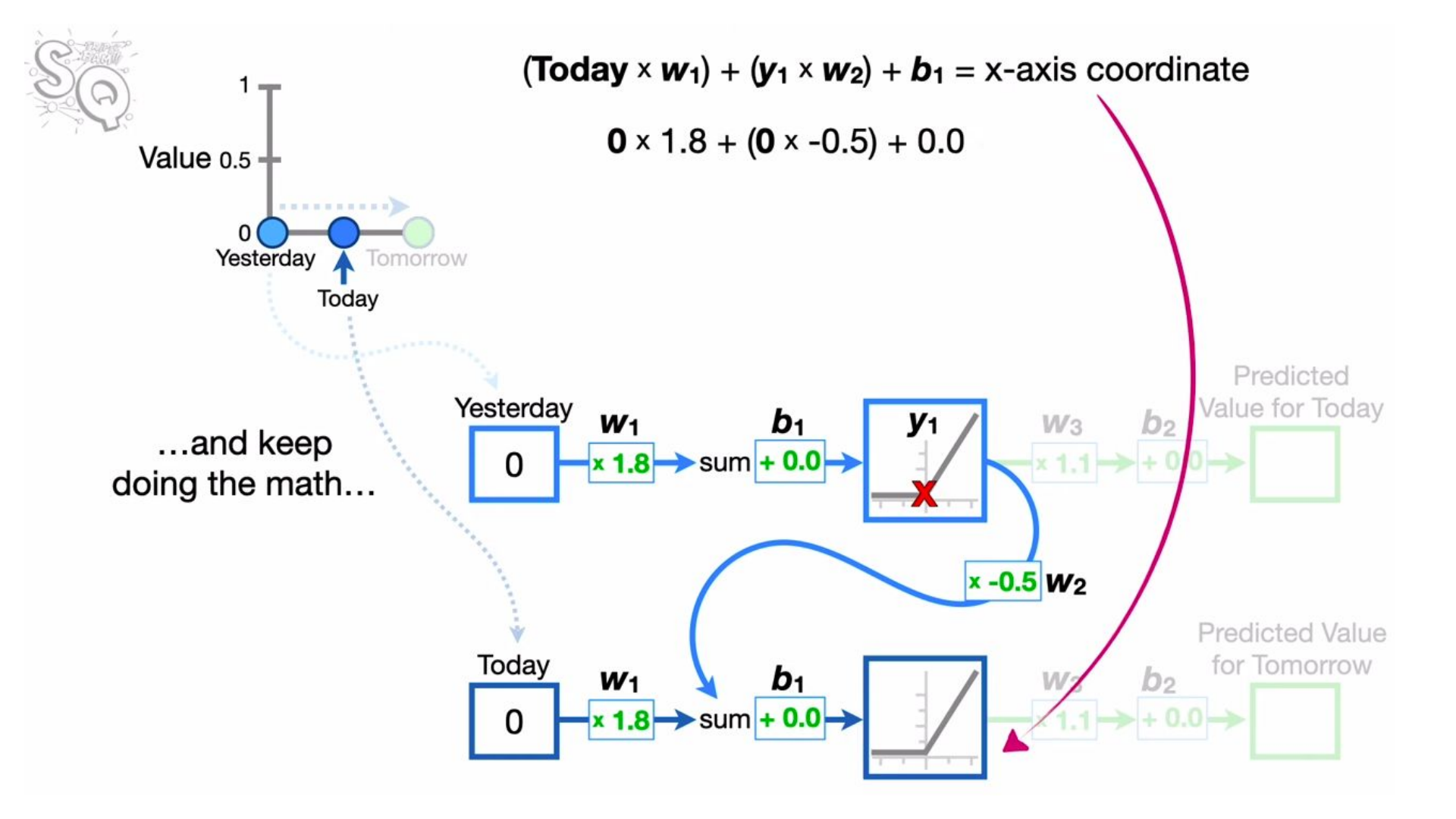
…then we follow the connection from the first activation function to the summation in the second copy of the neural network.

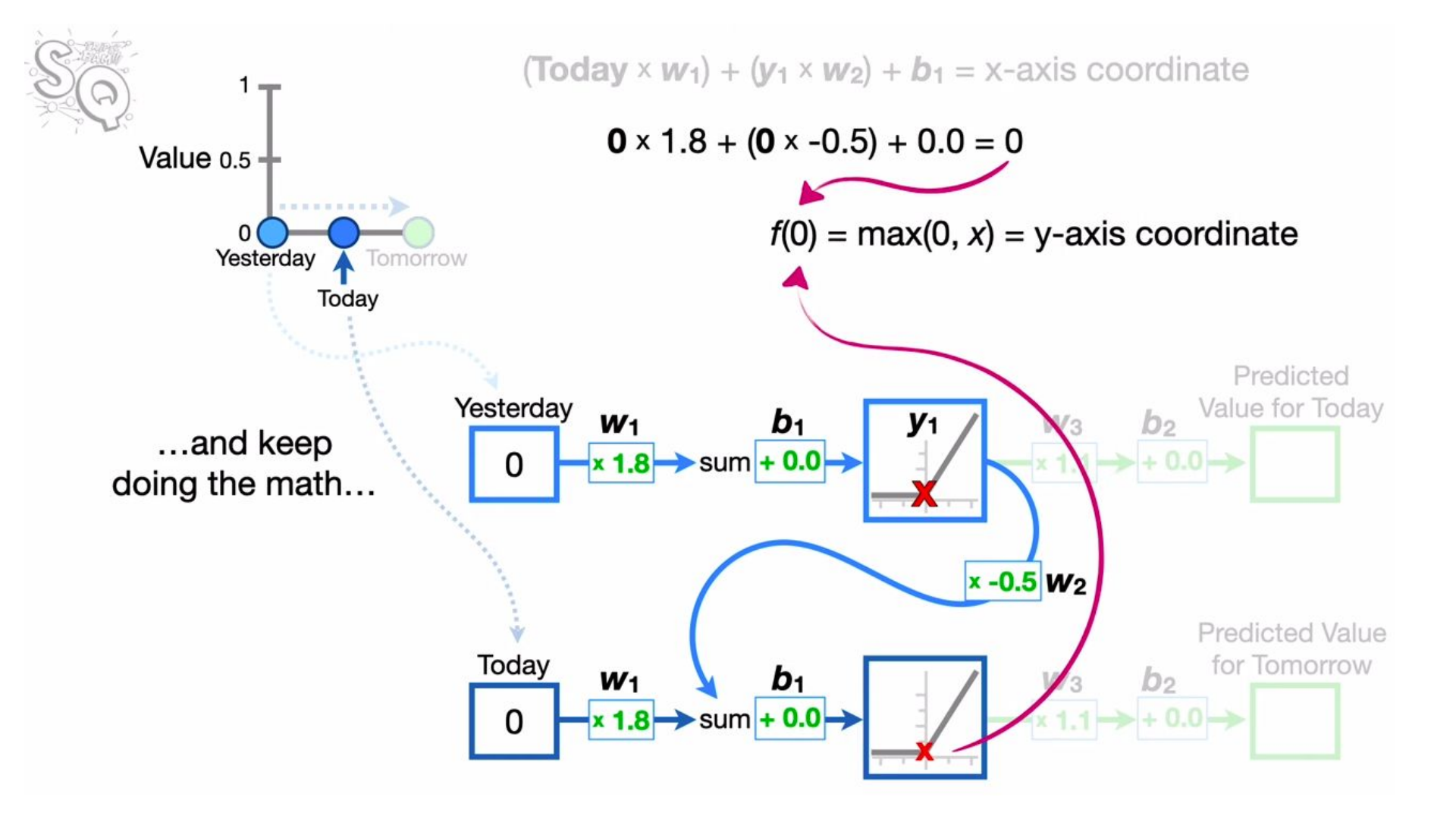
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5
w_2
Today
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

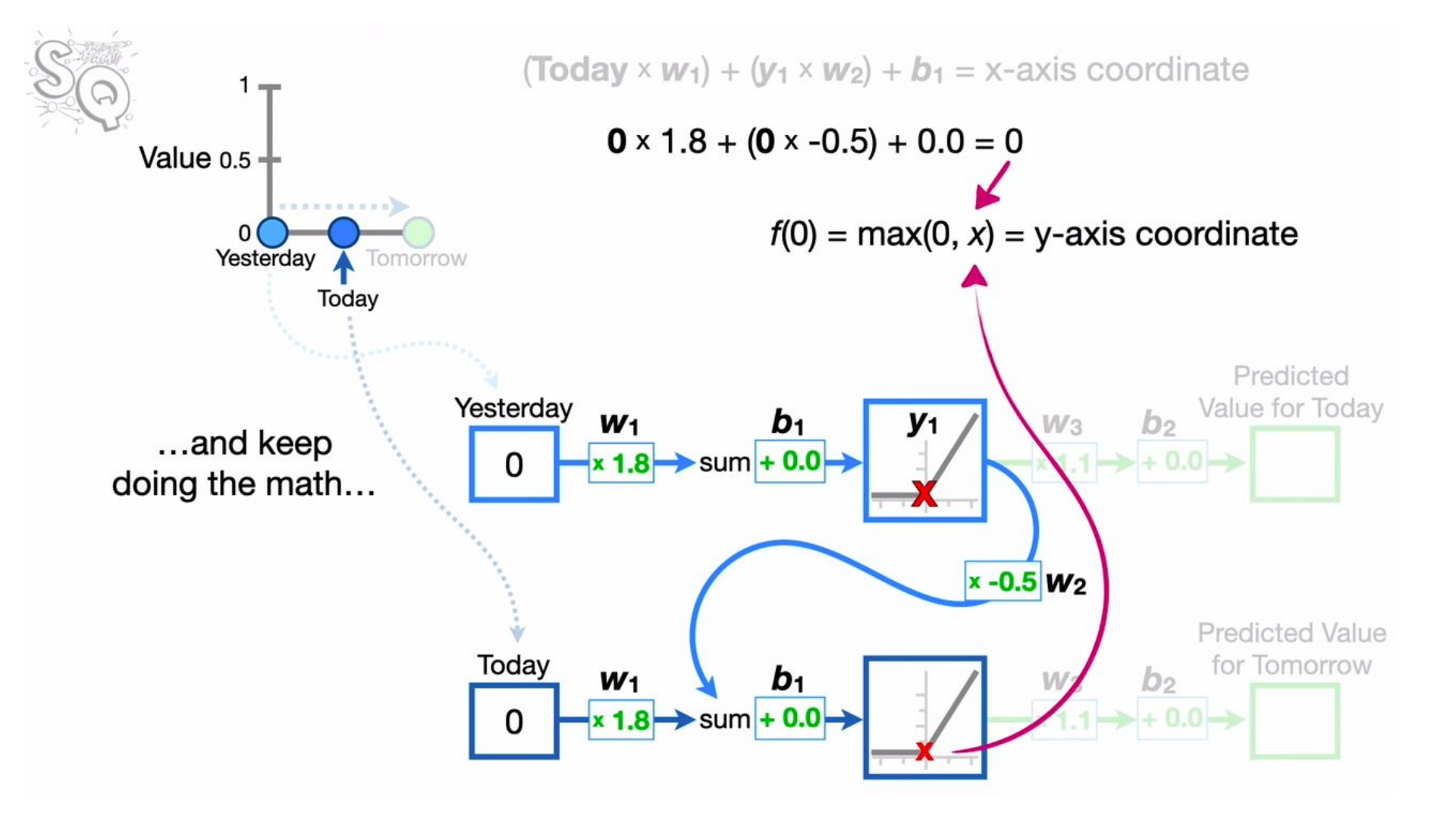

1
Value 0.5
0
Yesterday
Tomorrow
Today
Now we put **today's** value into the second input…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5
w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

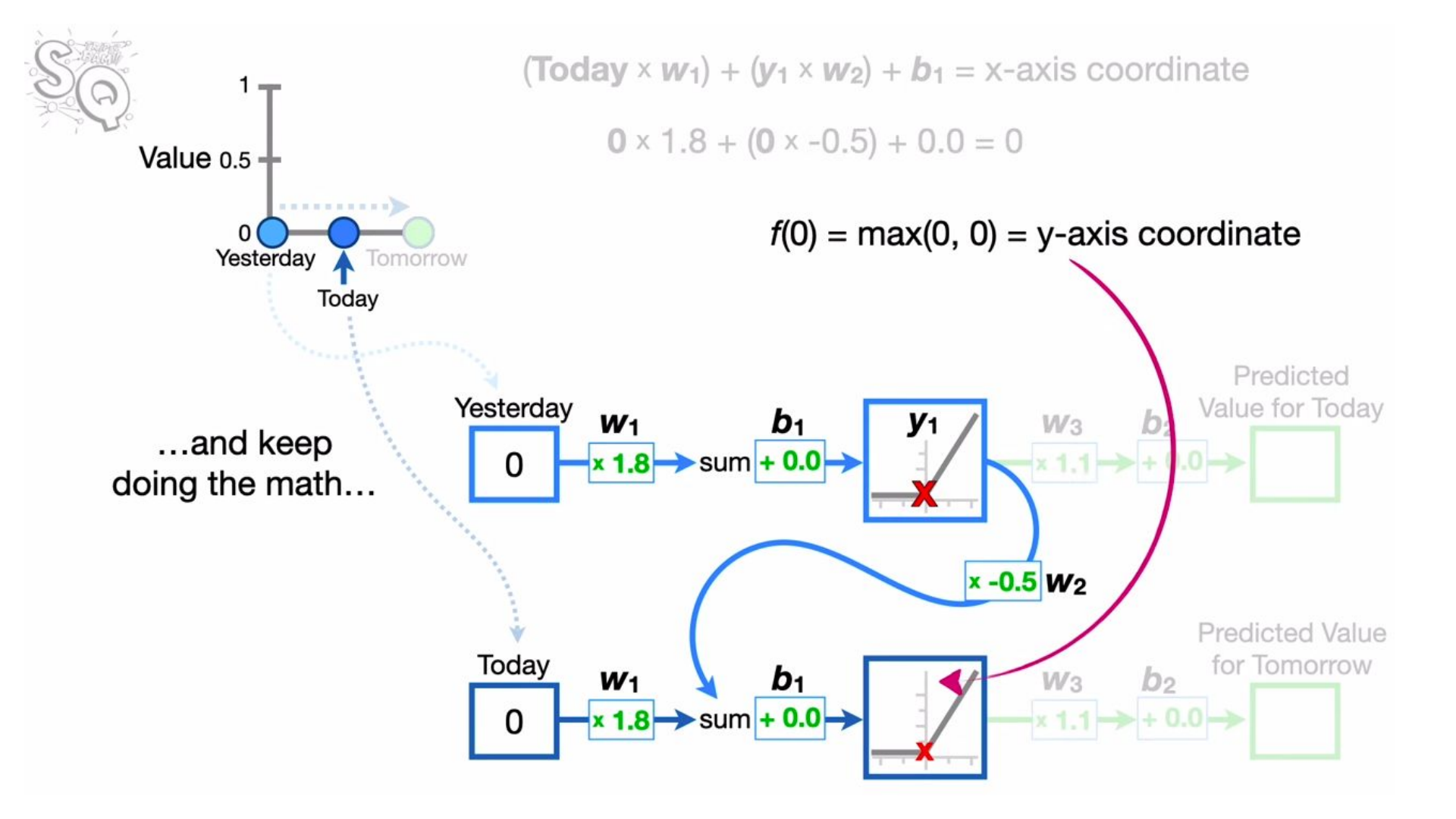

(Today x w_1)
1
Value 0.5
0
Yesterday
Tomorrow
Today
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

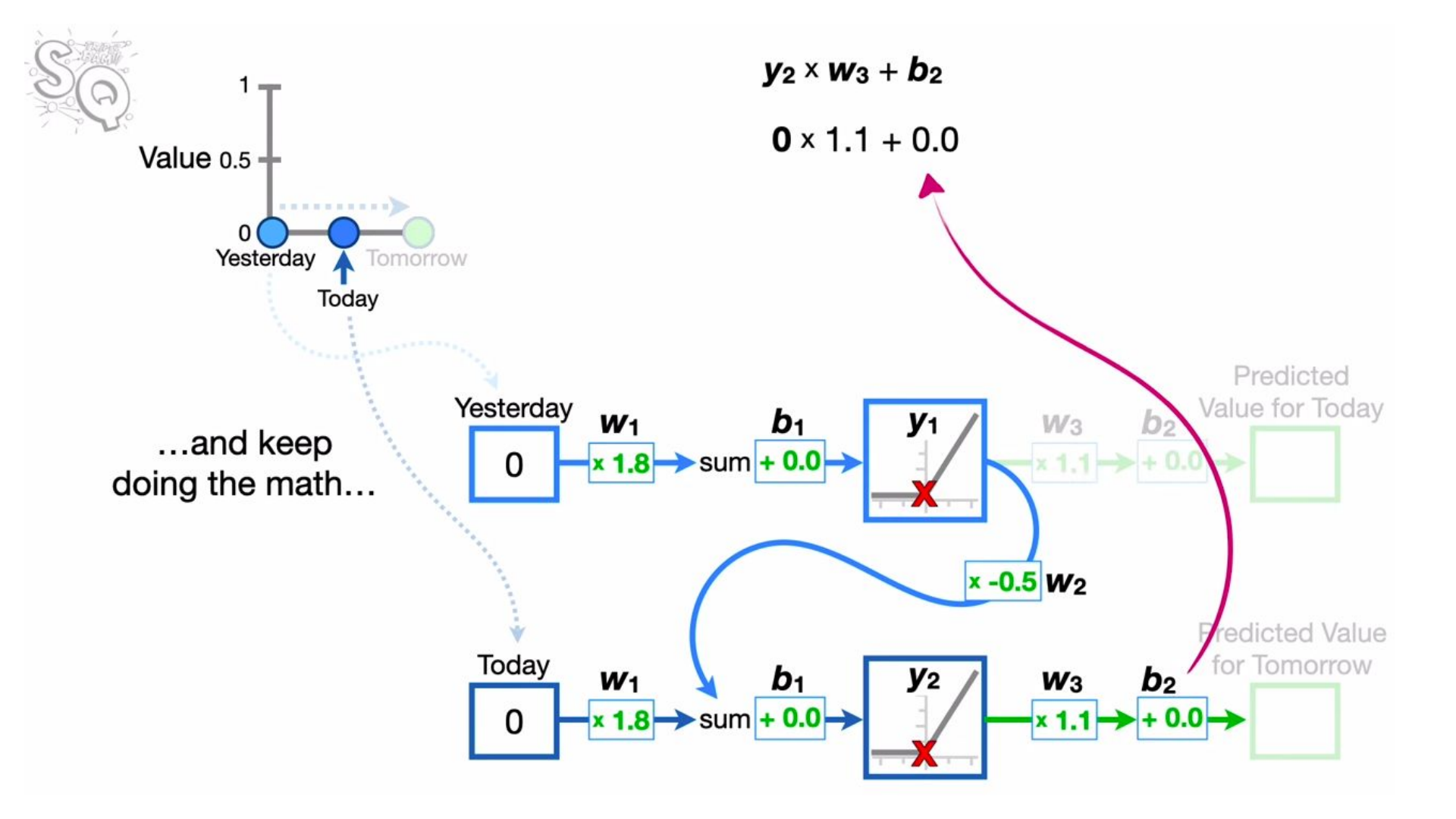

(Today x w_1) + (y_1 x w_2) + b_1
0 x 1.8 + (0 x -0.5)
1
Value 0.5
0
Yesterday
Tomorrow
Today
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

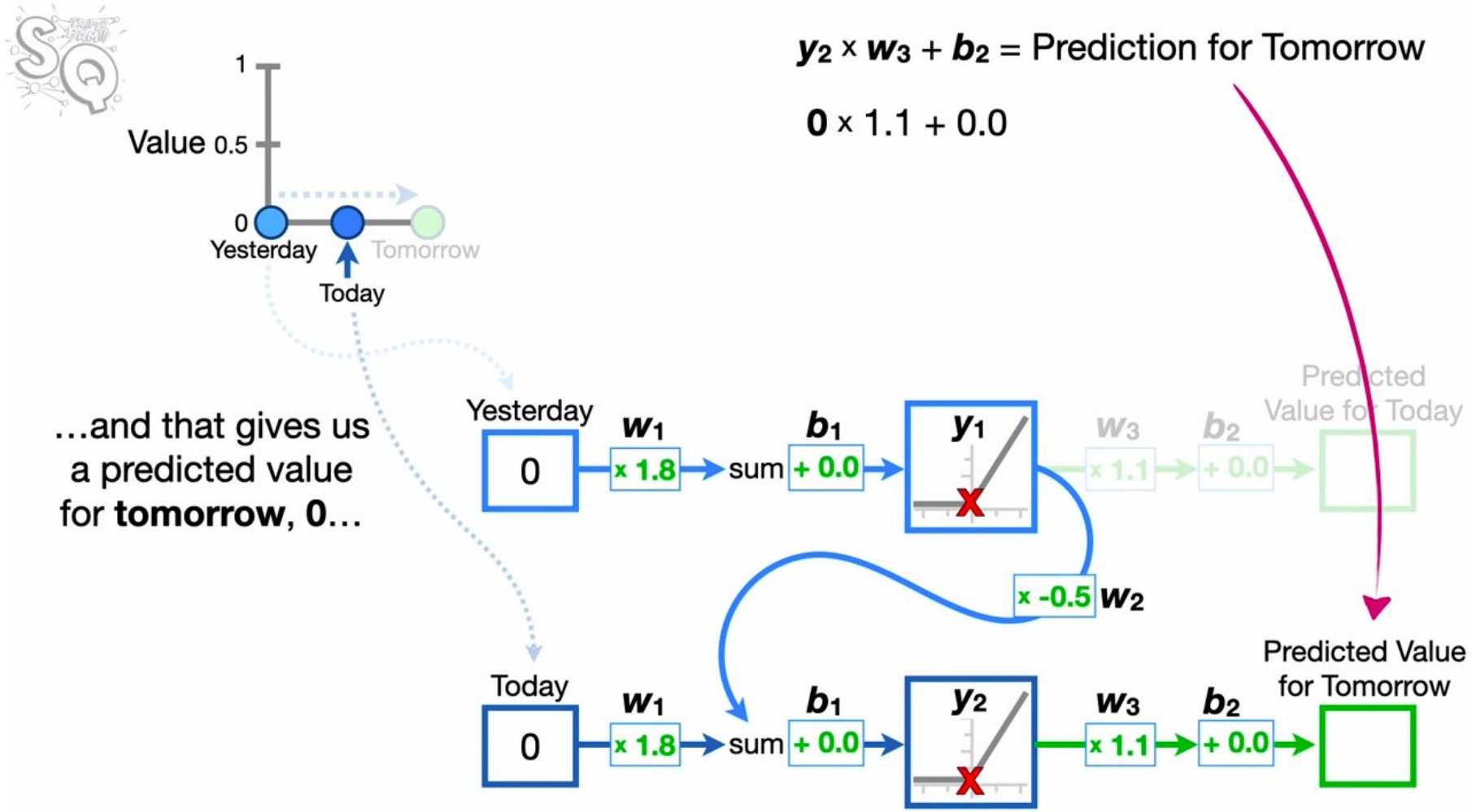

(Today x w_1) + (y_1 x w_2) + b_1 = x-axis coordinate
0 x 1.8 + (0 x -0.5) + 0.0
1
Value 0.5
0
Yesterday
Tomorrow
Today
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

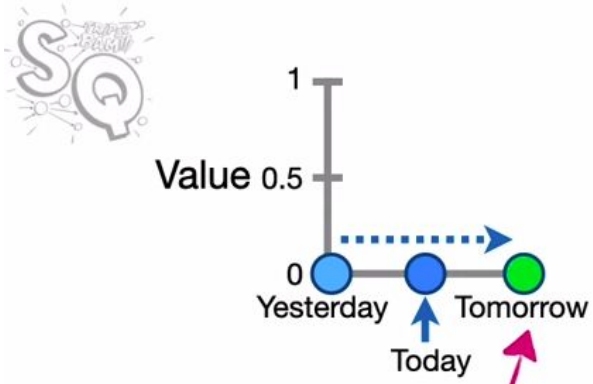

(Today x w_1) + (y_1 x w_2) + b_1 = x-axis coordinate
0 x 1.8 + (0 x -0.5) + 0.0 = 0
f(0) = max(0, x) = y-axis coordinate
1
Value 0.5
0
Yesterday
Tomorrow
Today
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow


(Today x w_1) + (y_1 x w_2) + b_1 = x-axis coordinate
0 x 1.8 + (0 x -0.5) + 0.0 = 0
f(0) = max(0, x) = y-axis coordinate
1
Value 0.5
0
Yesterday
Today
Tomorrow
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow


(Today x w_1) + (y_1 x w_2) + b_1 = x-axis coordinate
0 x 1.8 + (0 x -0.5) + 0.0 = 0
1
Value 0.5
0
Yesterday
Tomorrow
Today
$f(0)$ = max(0, 0) = y-axis coordinate
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow


1
Value 0.5
0
Yesterday
Today
Tomorrow
$y_2 \times w_3 + b_2$
0 x 1.1 + 0.0
…and keep doing the math…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

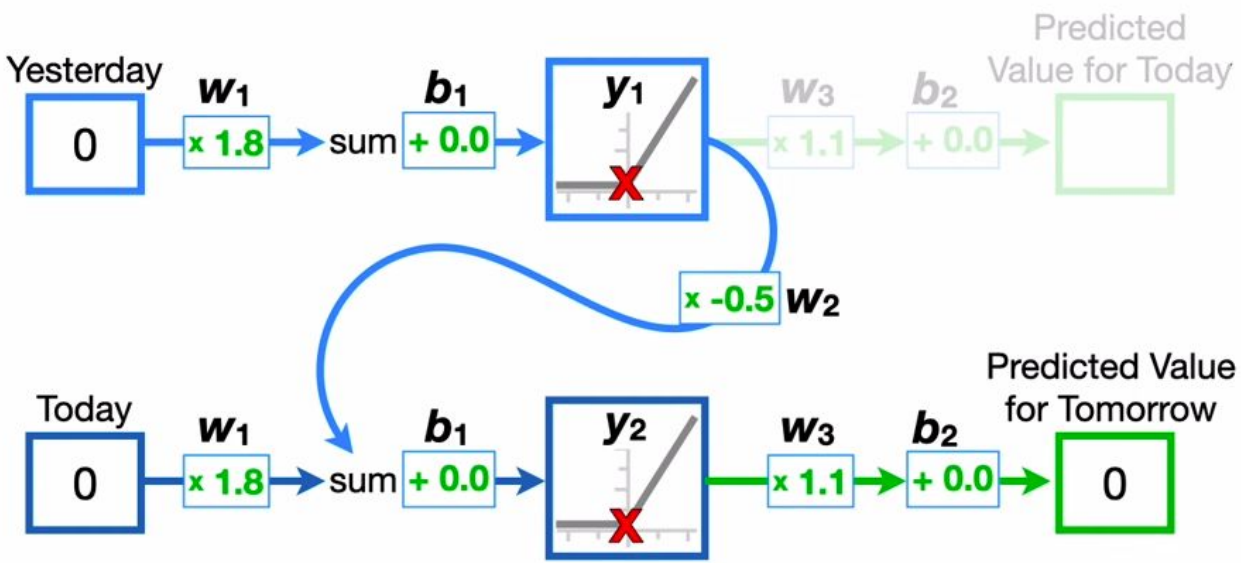

$y_2 \times w_3 + b_2$ = Prediction for Tomorrow
0 x 1.1 + 0.0
1
Value 0.5
0
Yesterday
Today
Tomorrow
…and that gives us a predicted value for **tomorrow**, **0**…
Yesterday
0
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

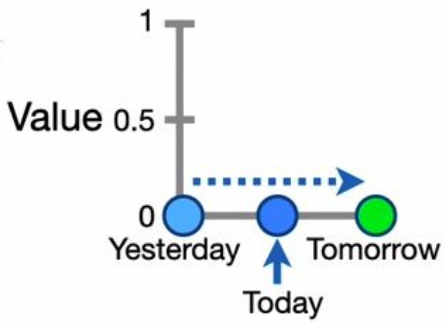


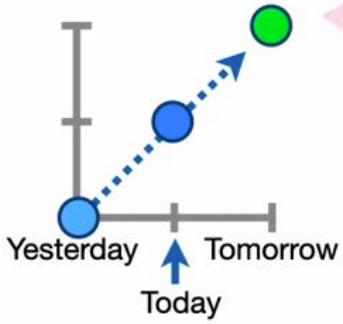
…which is consistent with the original observation.

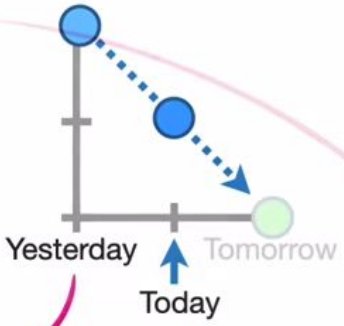
$y_2 \times w_3 + b_2$ = Prediction for Tomorrow

$0 \times 1.1 + 0.0 = 0$

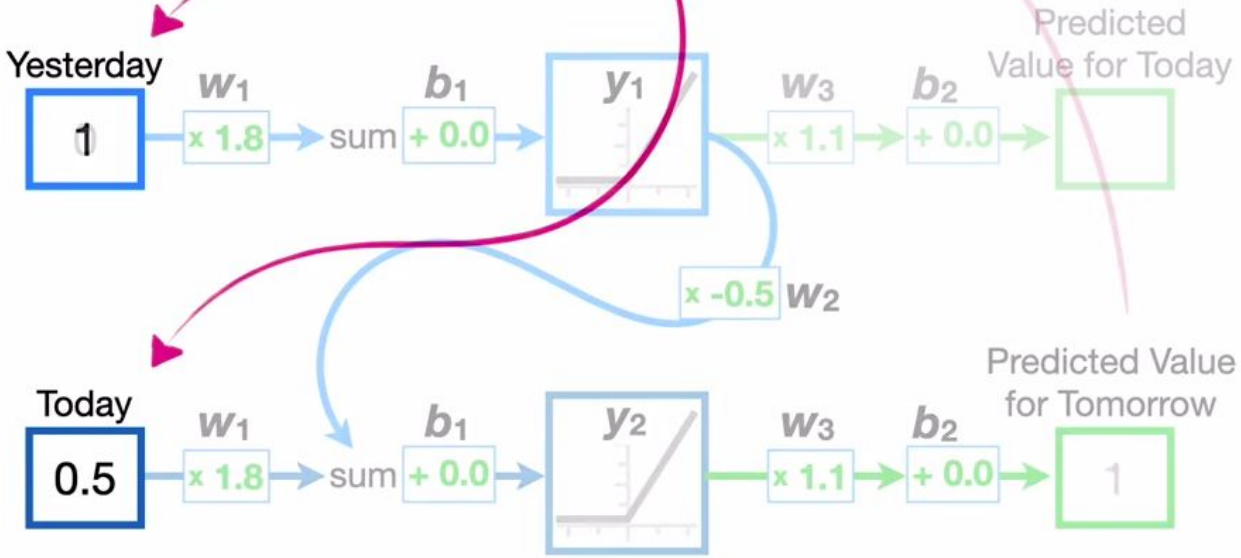


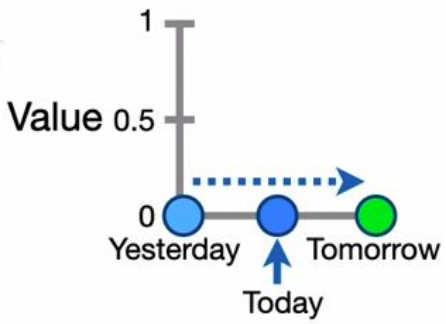


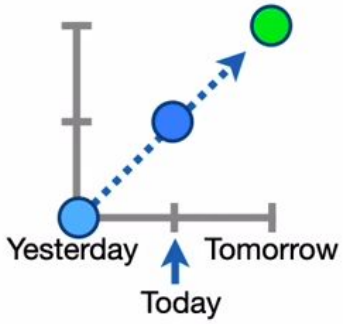


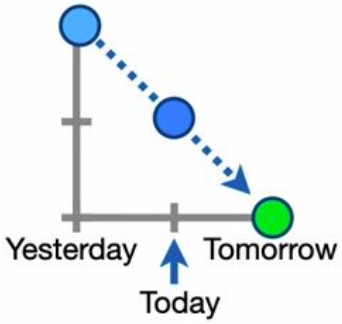


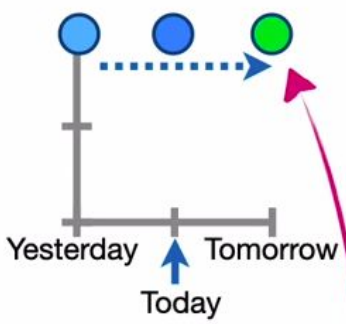
Likewise, when we run **yesterday** and **today's** values for the other scenarios through the recurrent neural network we predict the correct values for **tomorrow**.



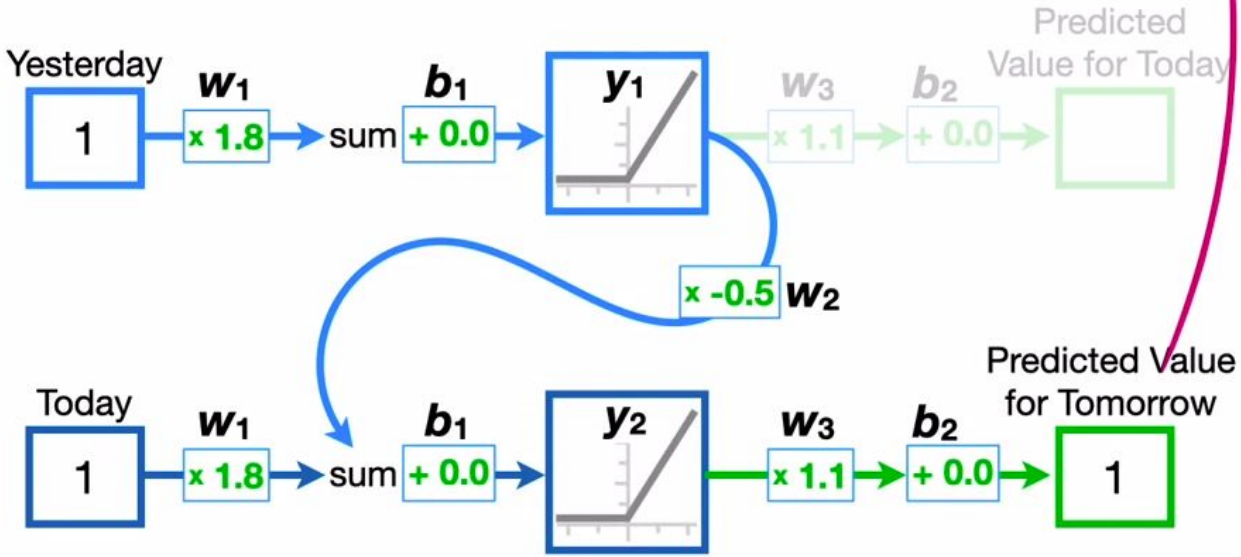


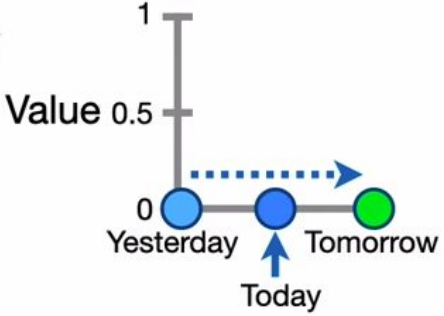




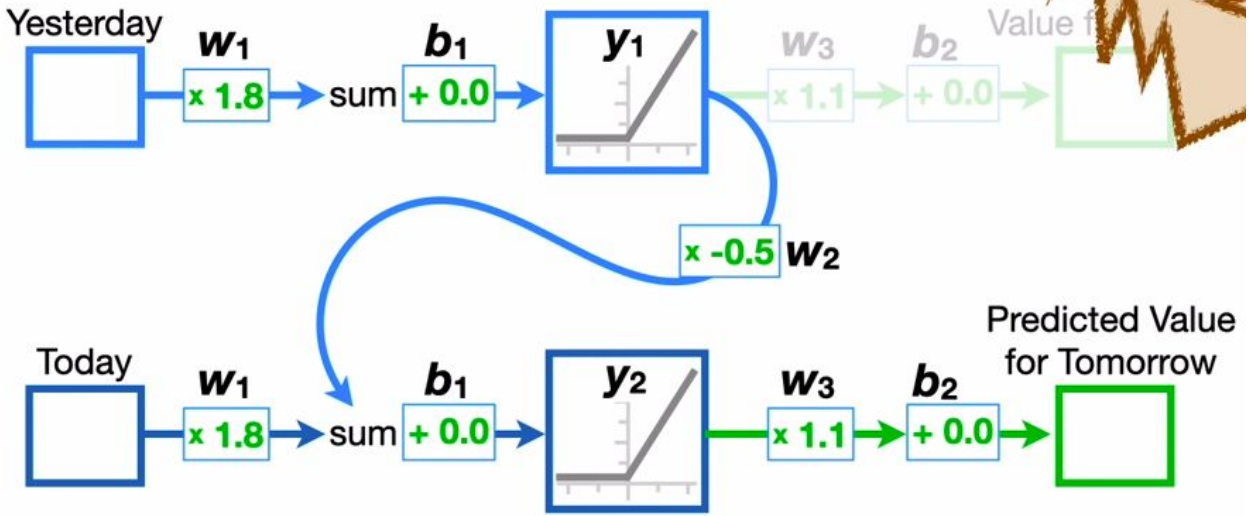


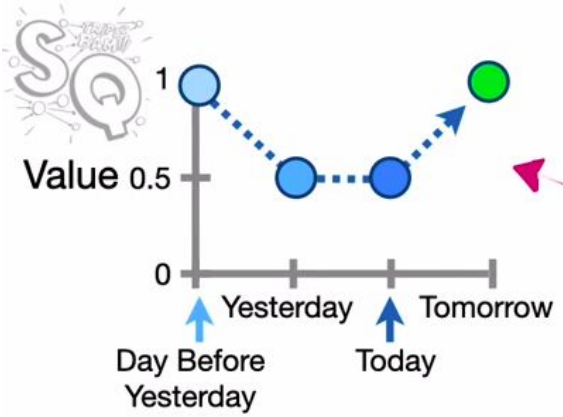
Likewise, when we run **yesterday** and **today's** values for the other scenarios through the recurrent neural network we predict the correct values for **tomorrow**.



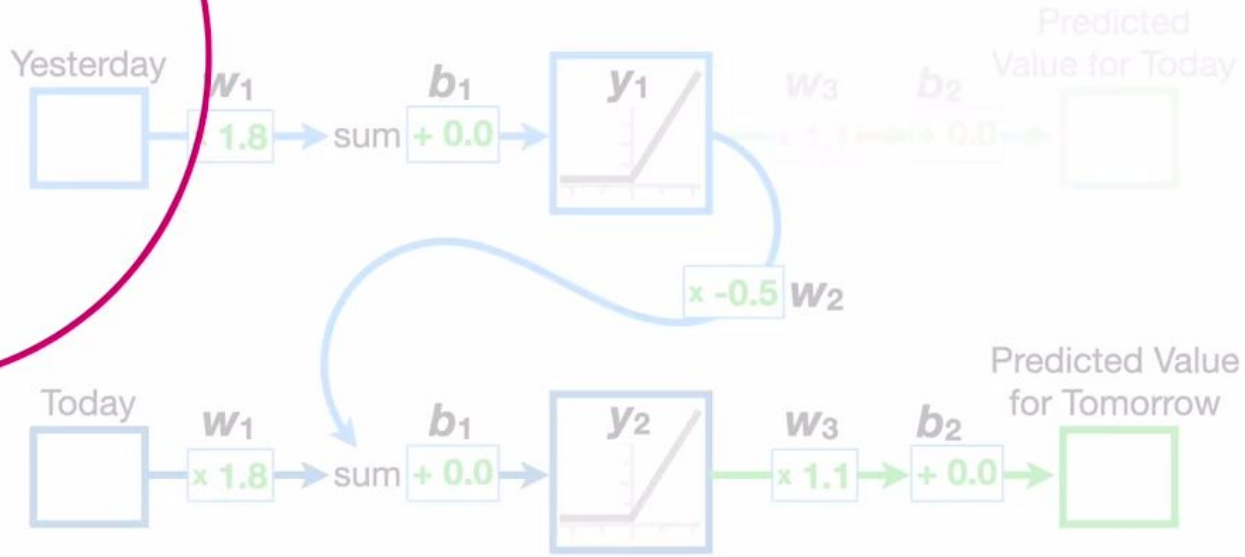


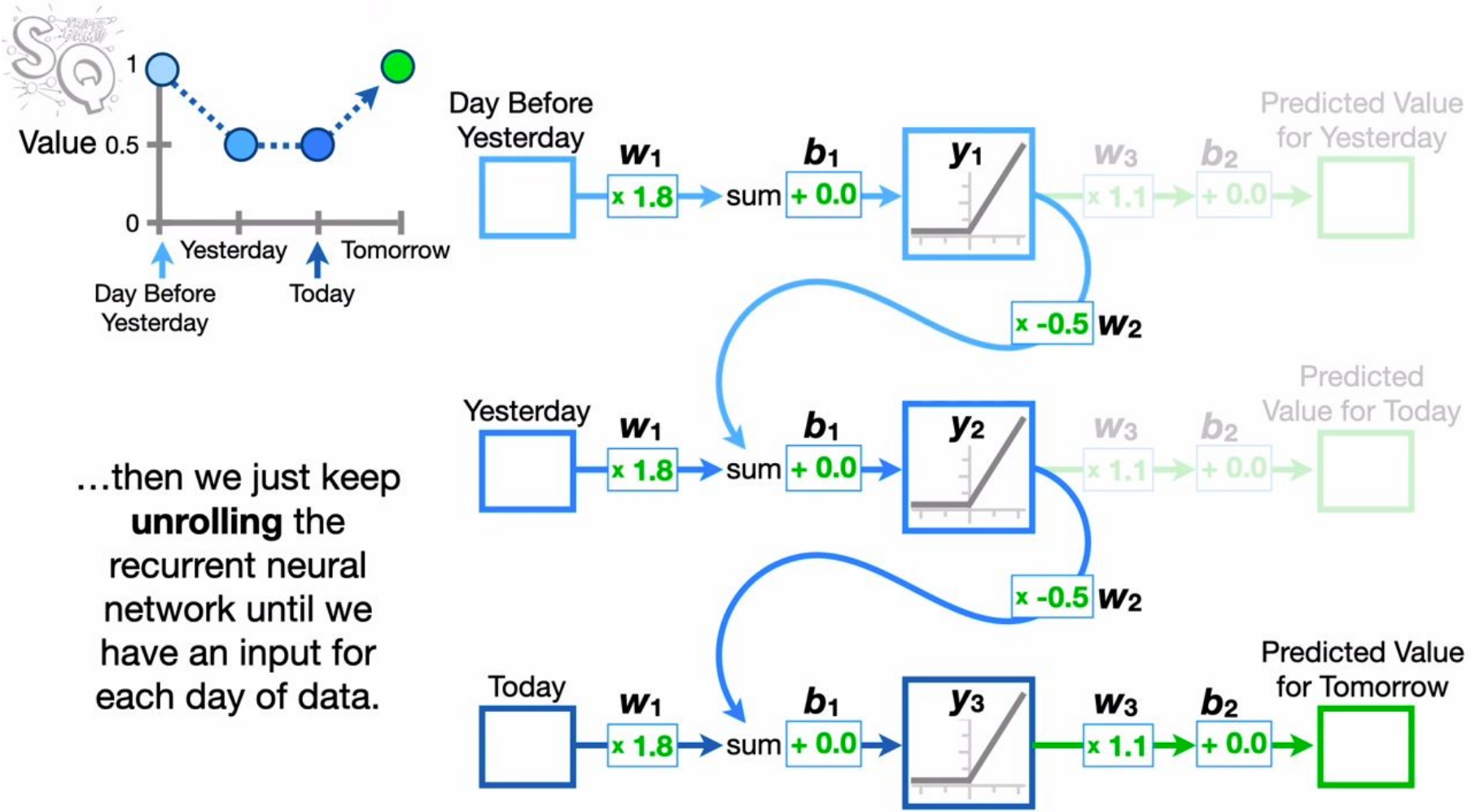
This recurrent neural network performs great with **2** days worth of data.



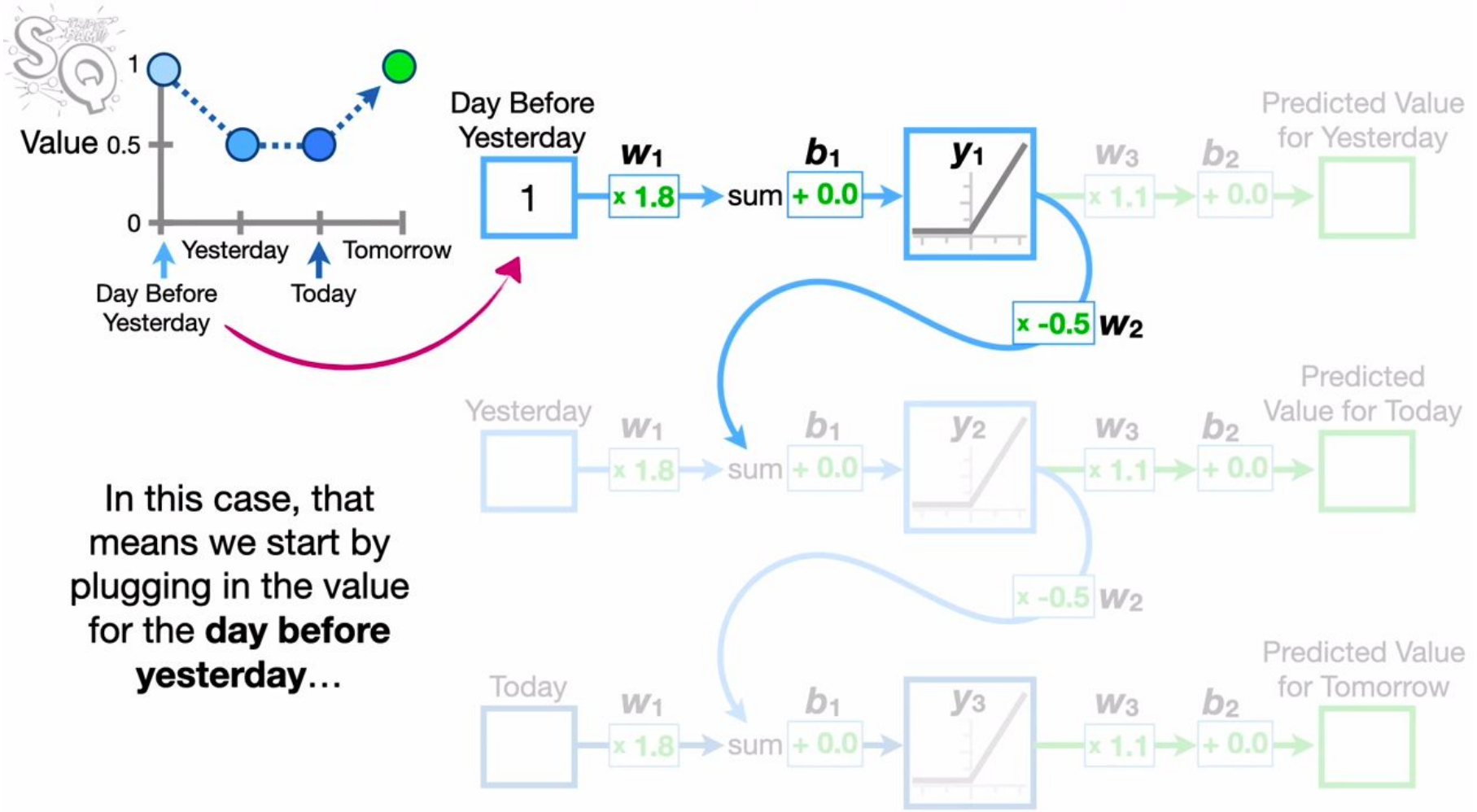


When we want to use **3** days of data to make a prediction about **tomorrow's** price, like this…

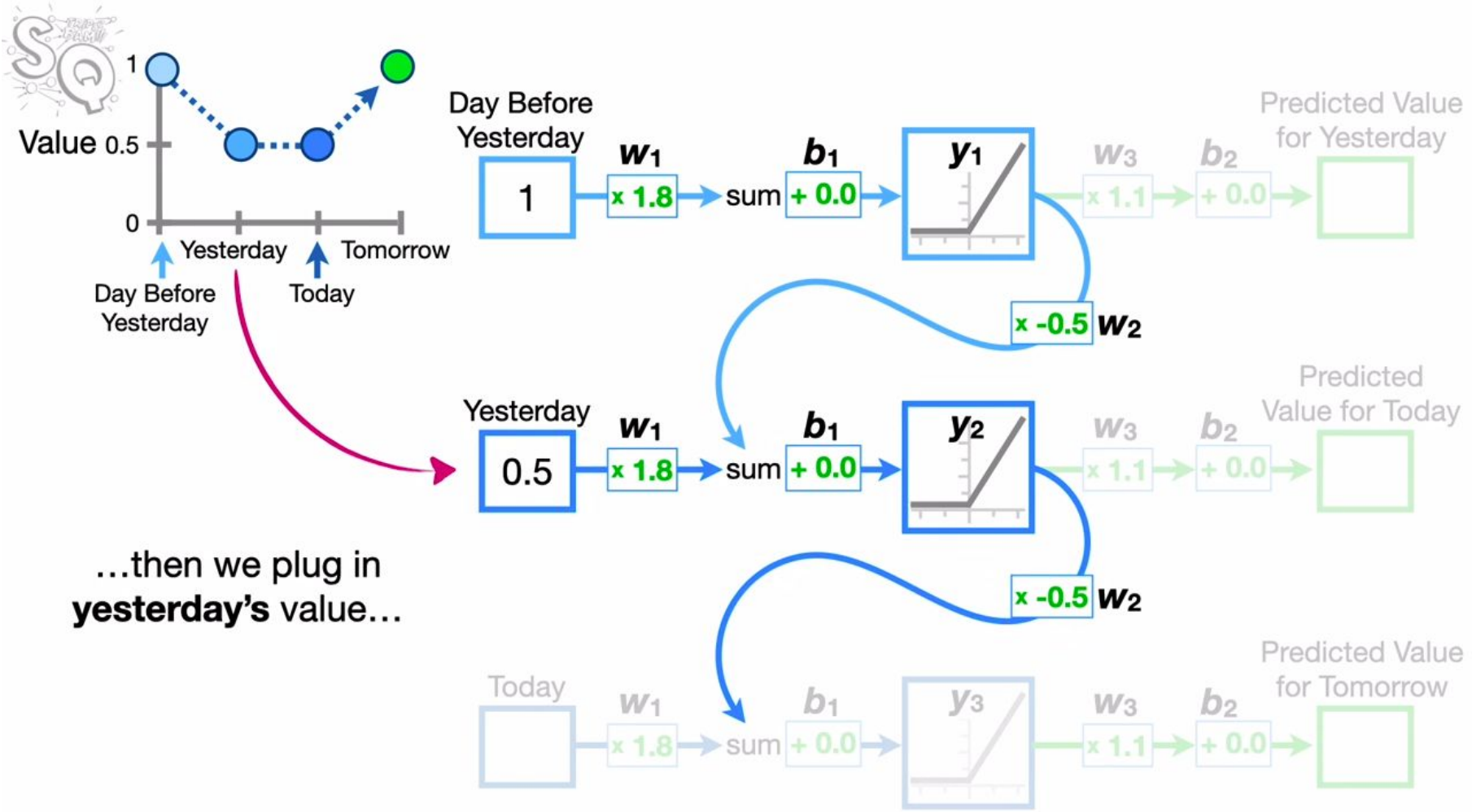


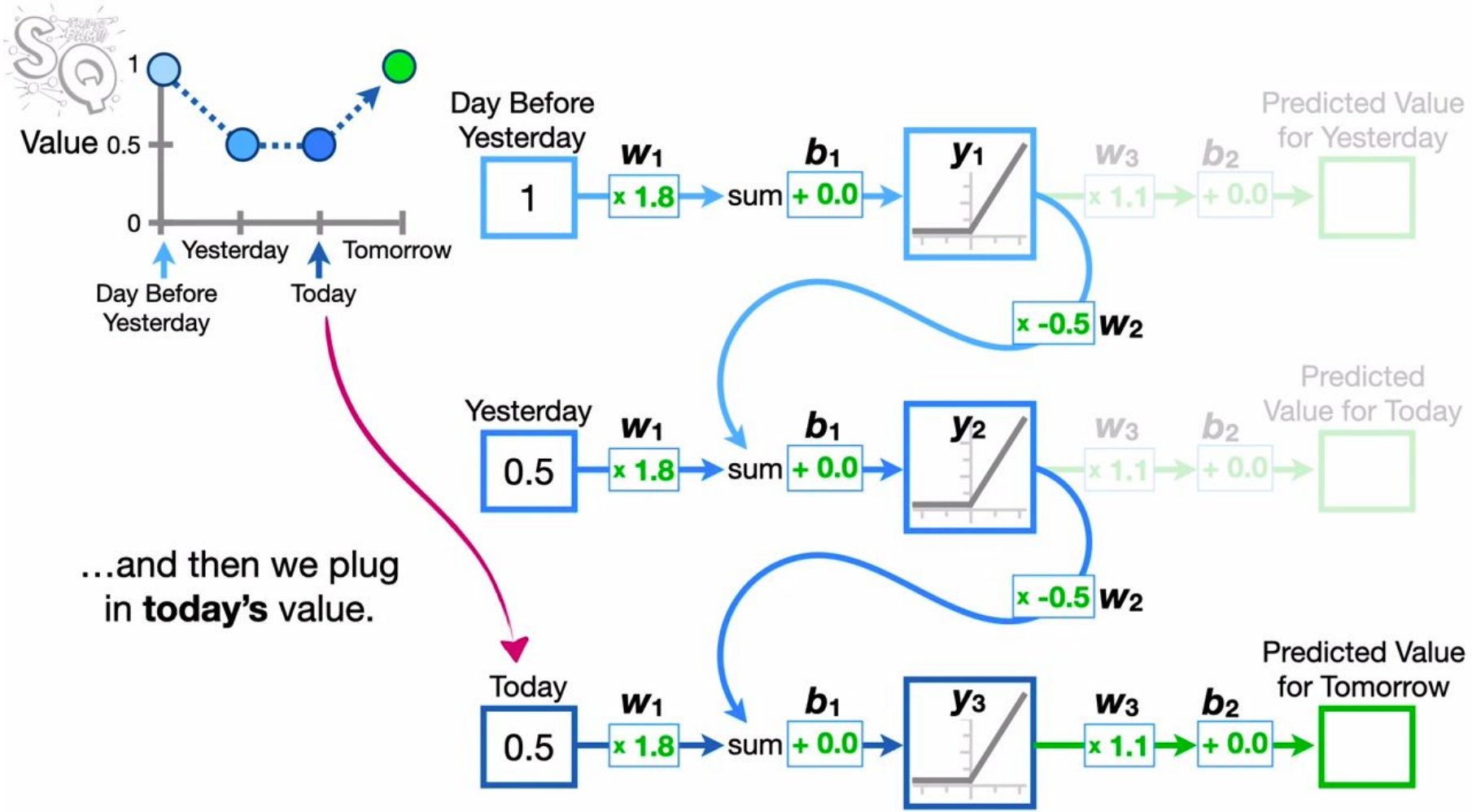


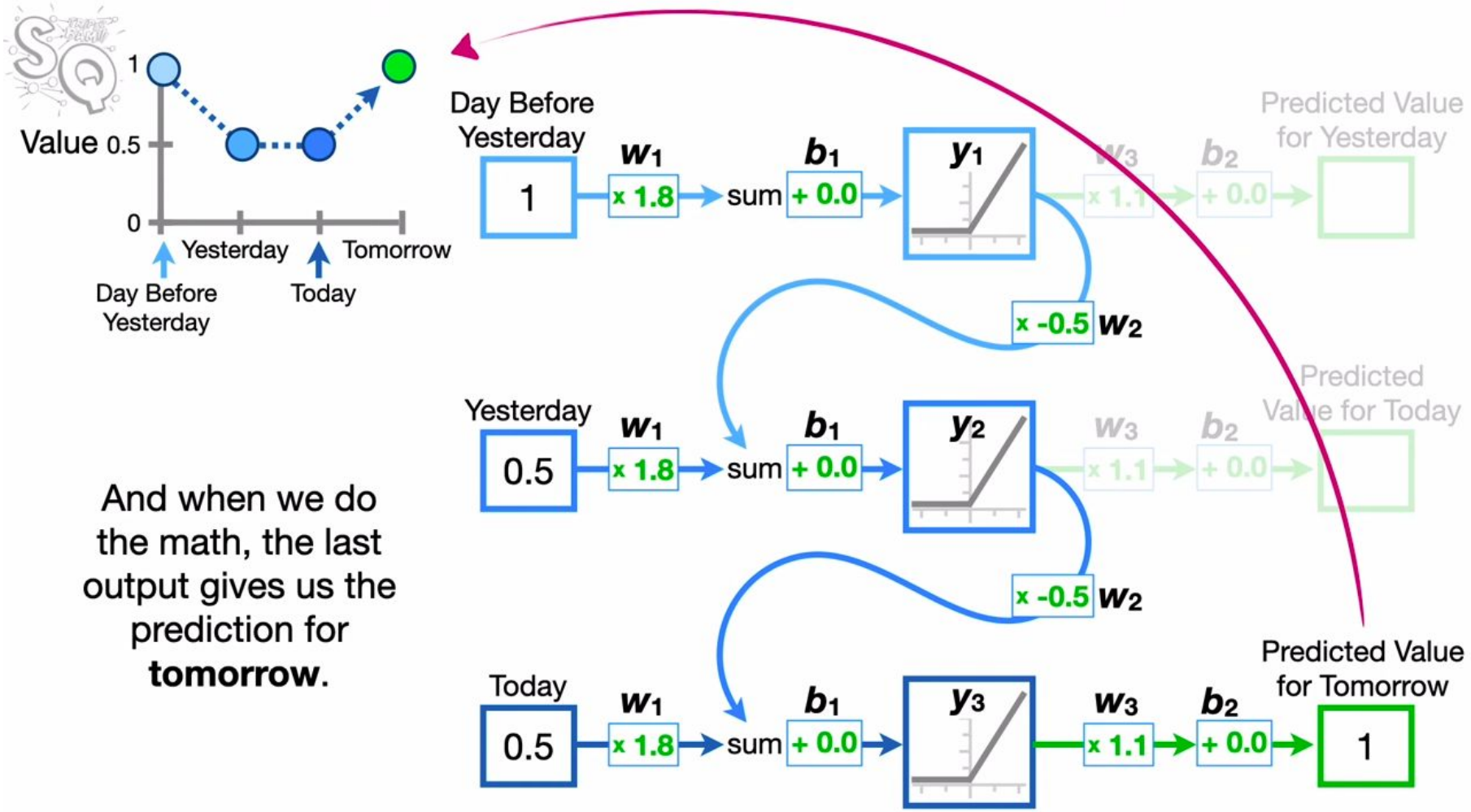
…then we just keep **unrolling** the recurrent neural network until we have an input for each day of data.



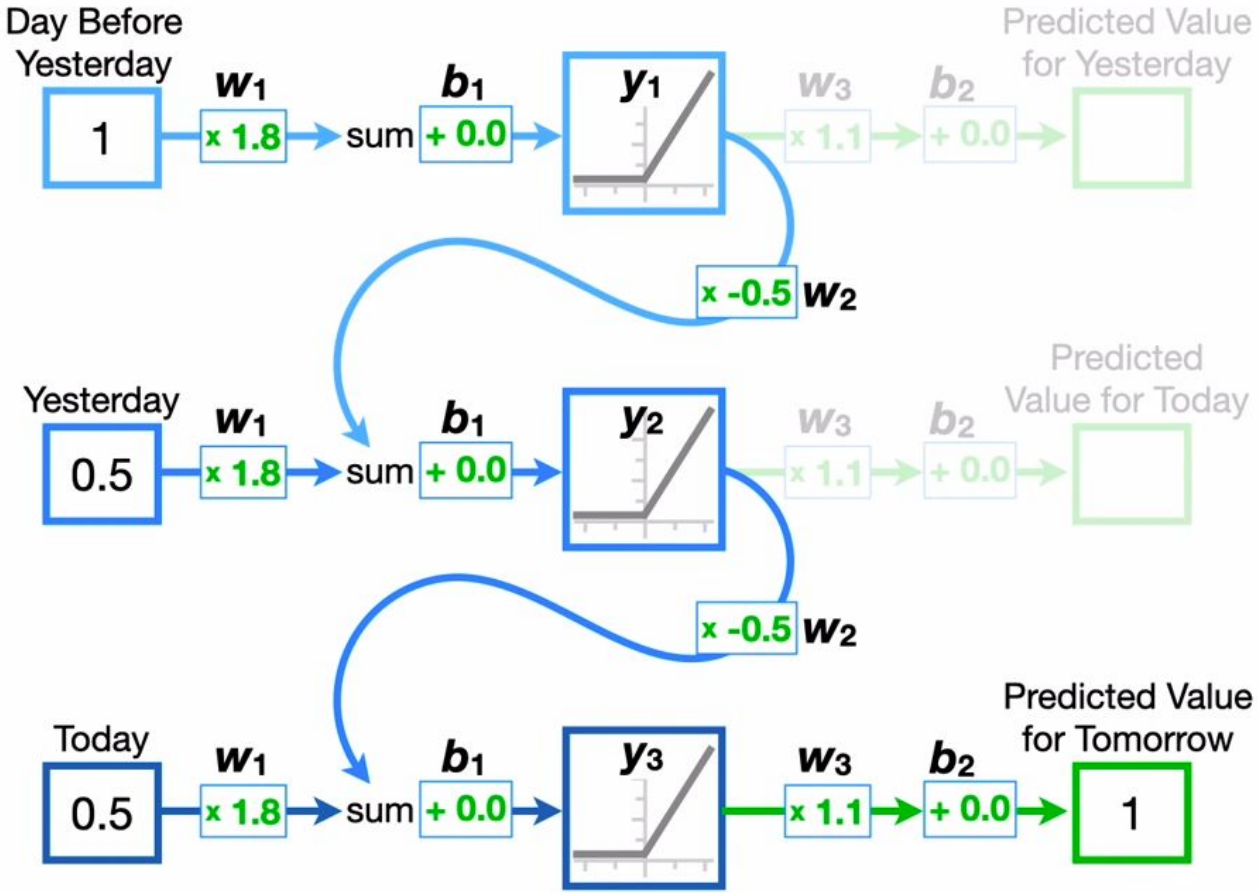
In this case, that means we start by plugging in the value for the **day before yesterday**…

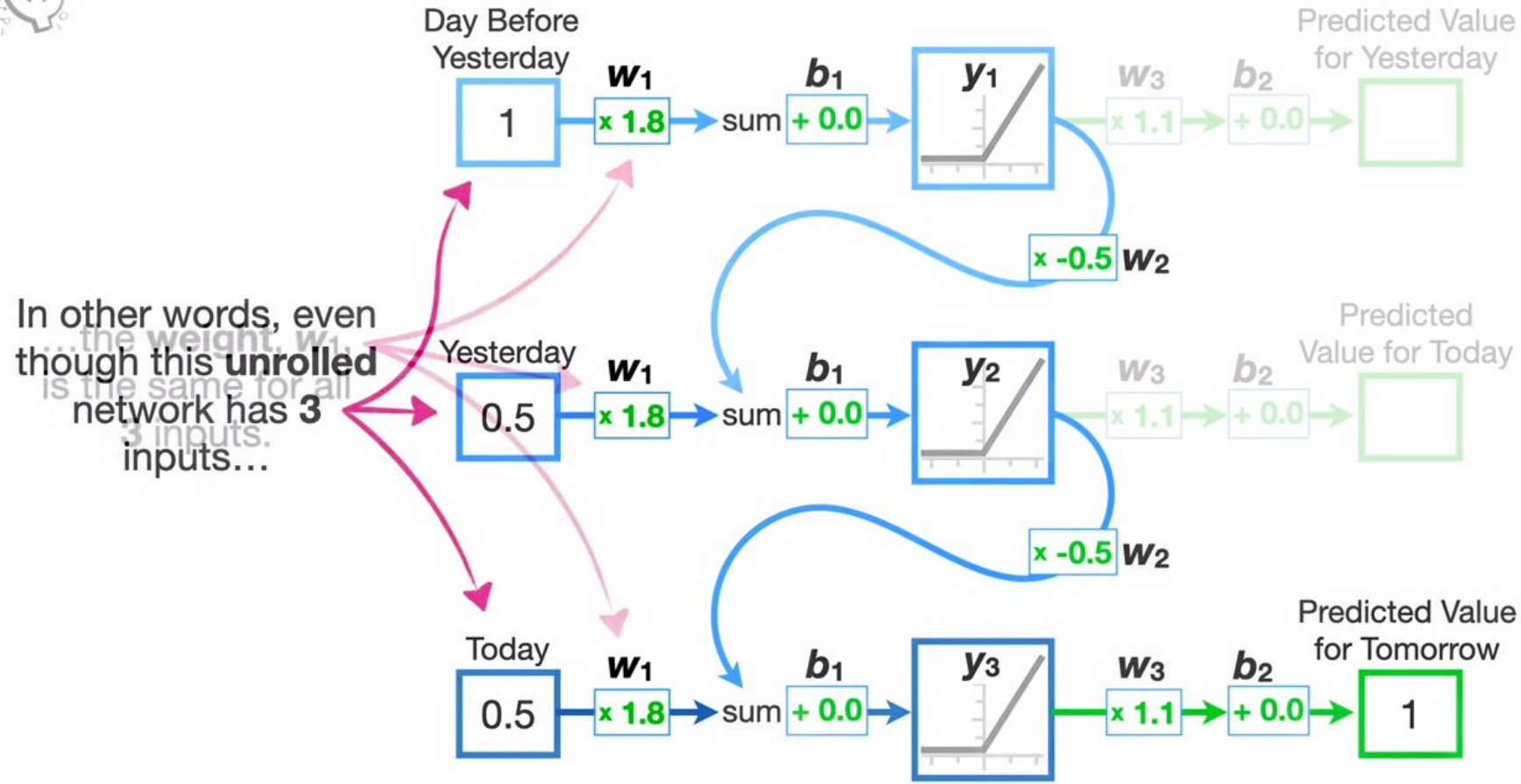

1
Value 0.5
0
Yesterday
Tomorrow
Day Before Yesterday
Today
Day Before Yesterday
1
w1
x 1.8
sum
b1
+ 0.0
y1
w3
x 1.1
b2
+ 0.0
Predicted Value for Yesterday
x -0.5
w2
Yesterday
0.5
w1
x 1.8
sum
b1
+ 0.0
y2
w3
x 1.1
b2
+ 0.0
Predicted Value for Today
x -0.5
w2
…then we plug in yesterday's value…
Today
w1
x 1.8
sum
b1
+ 0.0
y3
w3
x 1.1
b2
+ 0.0
Predicted Value for Tomorrow

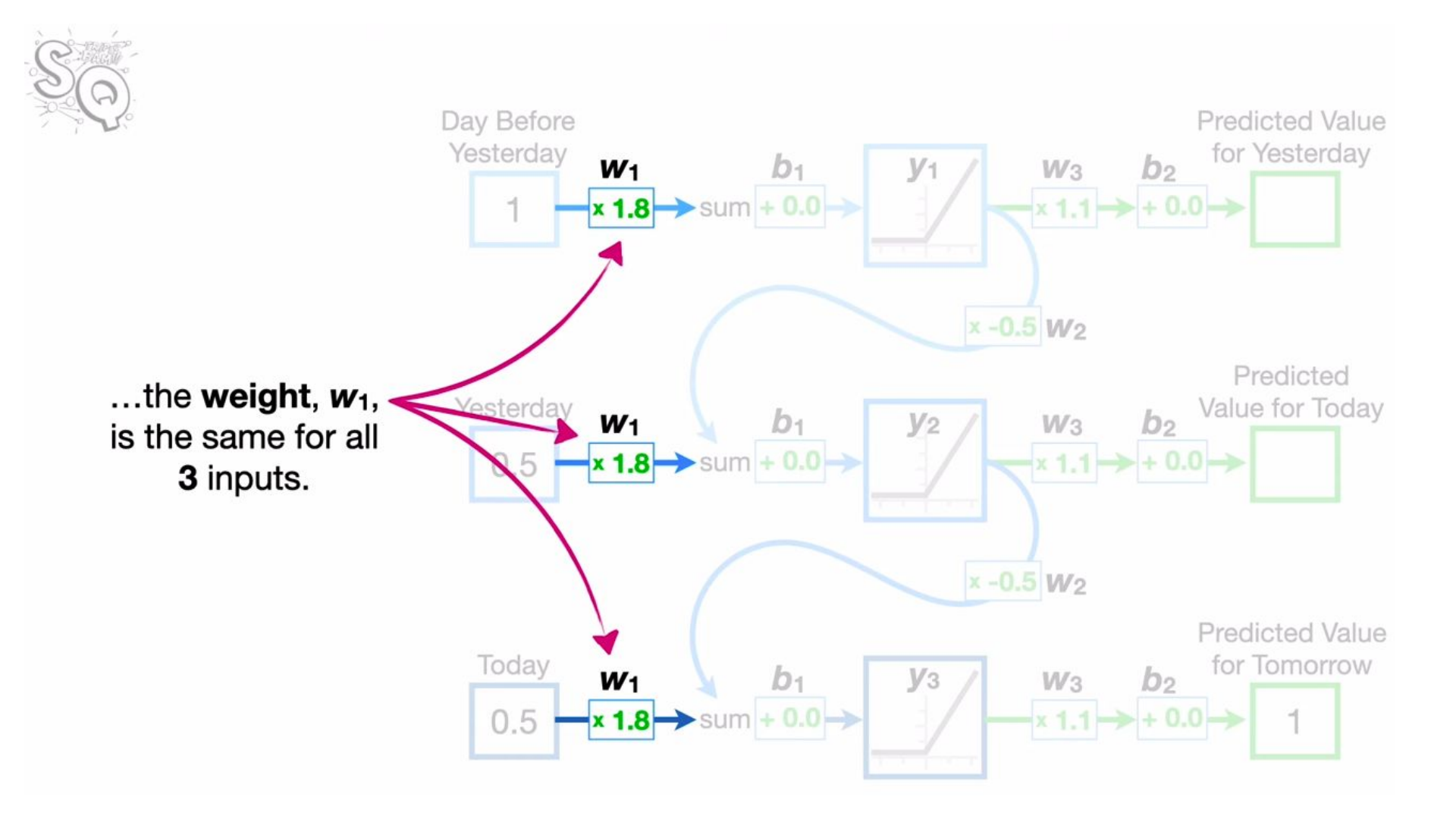

1
Value 0.5
0
Yesterday
Tomorrow
Day Before Yesterday
Today
Day Before Yesterday
1
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Yesterday
x -0.5 w_2
Yesterday
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
…and then we plug in **today's** value.
Today
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_3
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow

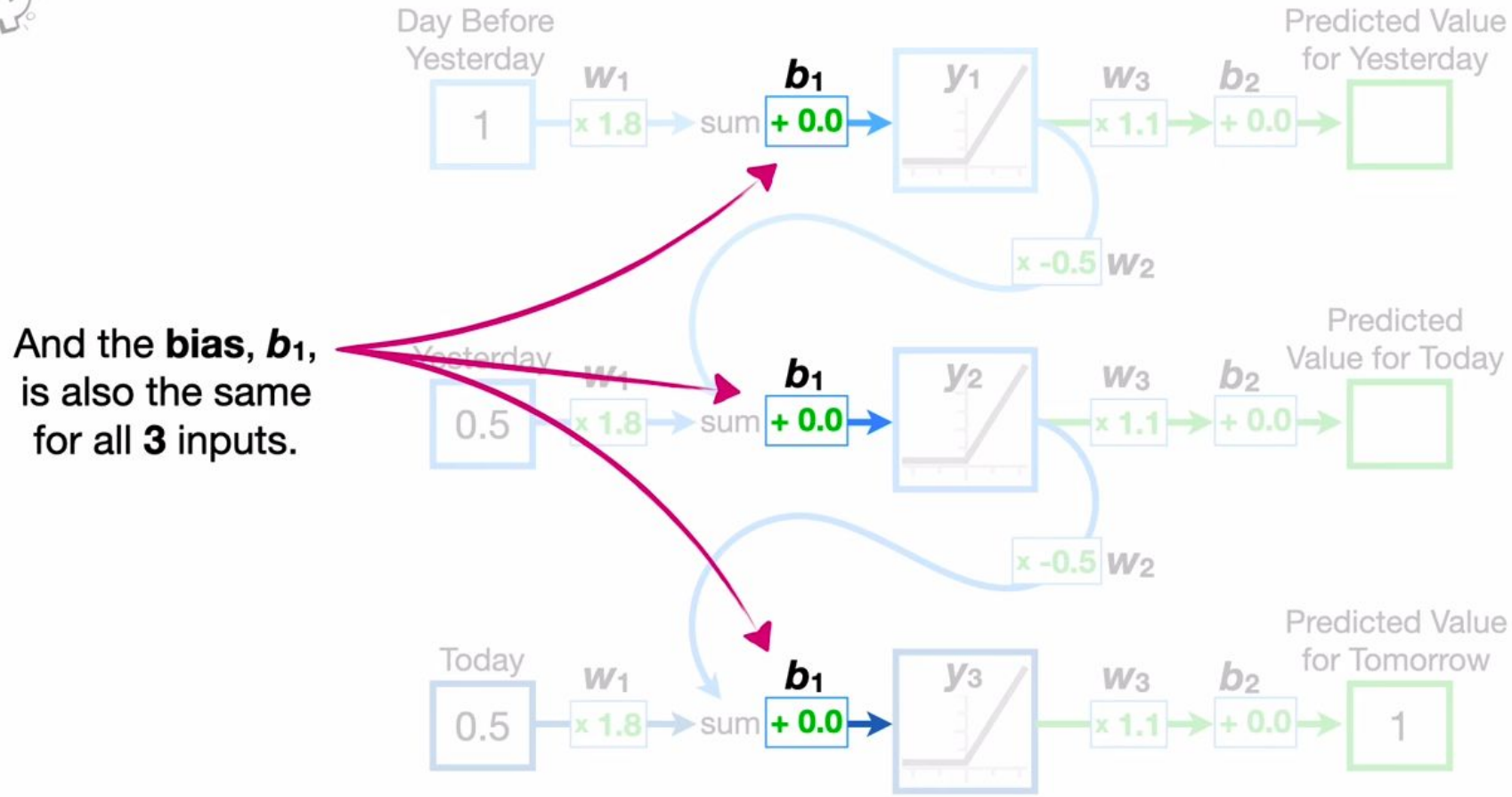

1
Value 0.5
0
Yesterday
Tomorrow
Day Before Yesterday
Today
Day Before Yesterday
1
w1
x 1.8
sum
b1
+ 0.0
y1
w3
x 1.1
b2
+ 0.0
Predicted Value for Yesterday
x -0.5 w2
Yesterday
0.5
w1
x 1.8
sum
b1
+ 0.0
y2
w3
x 1.1
b2
+ 0.0
Predicted Value for Today
x -0.5 w2
Today
0.5
w1
x 1.8
sum
b1
+ 0.0
y3
w3
x 1.1
b2
+ 0.0
Predicted Value for Tomorrow
1
And when we do the math, the last output gives us the prediction for **tomorrow**.

NOTE: Regardless of how many times we **unroll** a recurrent neural network, the **weights** and **biases** are shared across every input.

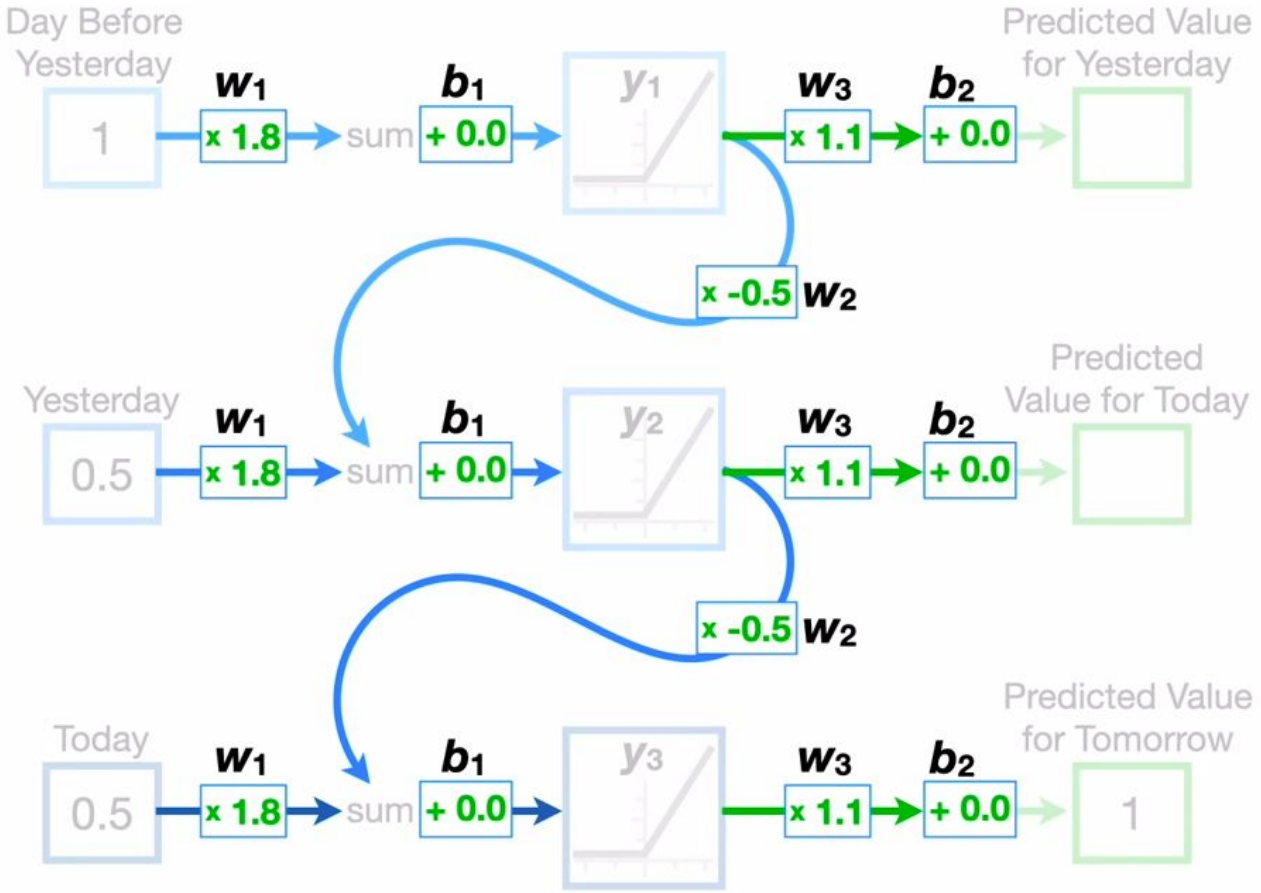



Day Before Yesterday
1
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Yesterday
x -0.5 w_2
In other words, even though this unrolled network has 3 inputs…
Yesterday
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_3
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow
1

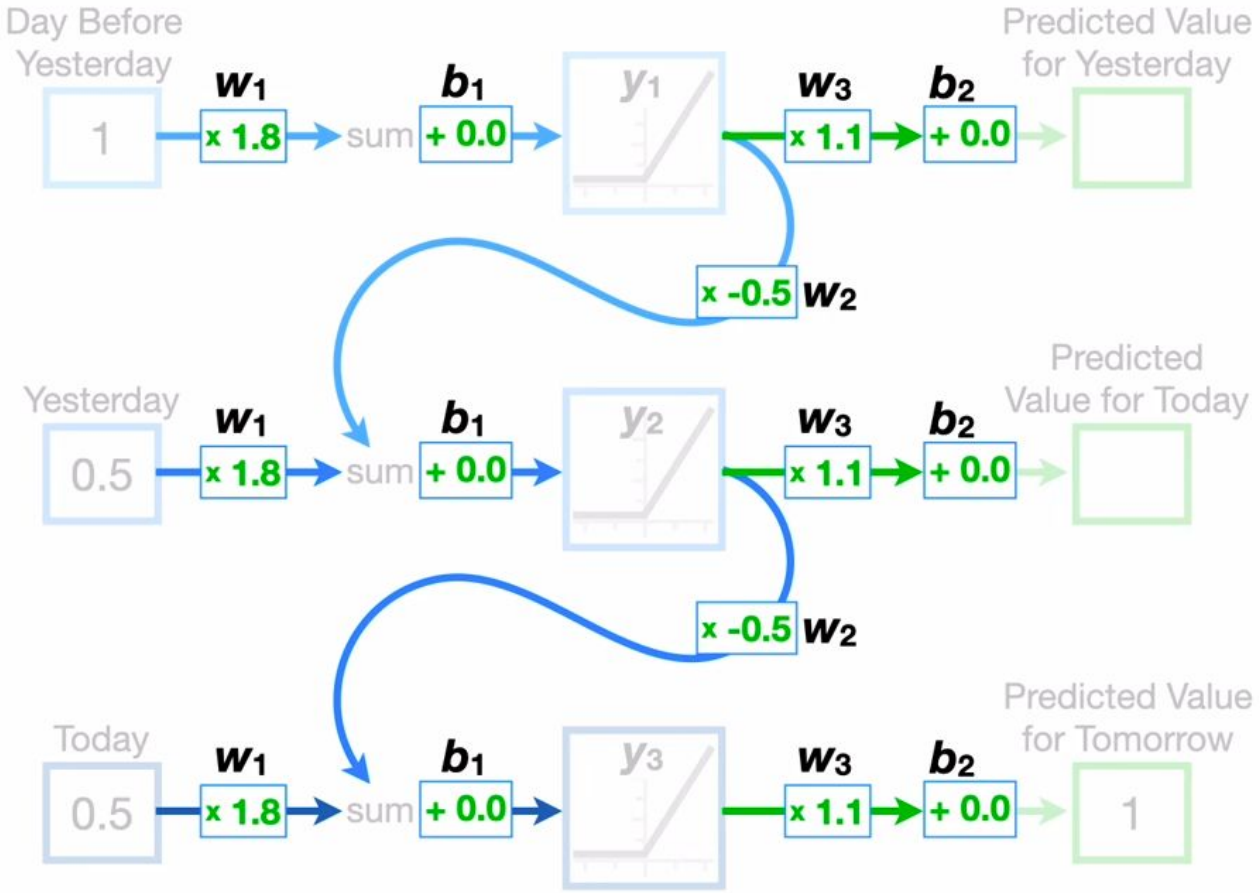

…the **weight**, w_1, is the same for all **3** inputs.
Day Before Yesterday
1
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Yesterday
x -0.5 w_2
Yesterday
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_3
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow
1


Day Before Yesterday
1
w_1
x 1.8
sum
b_1
+ 0.0
y_1
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Yesterday
x -0.5 w_2
Yesterday
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_2
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Today
x -0.5 w_2
Today
0.5
w_1
x 1.8
sum
b_1
+ 0.0
y_3
w_3
x 1.1
b_2
+ 0.0
Predicted Value for Tomorrow
1
And the **bias**, b_1, is also the same for all **3** inputs.

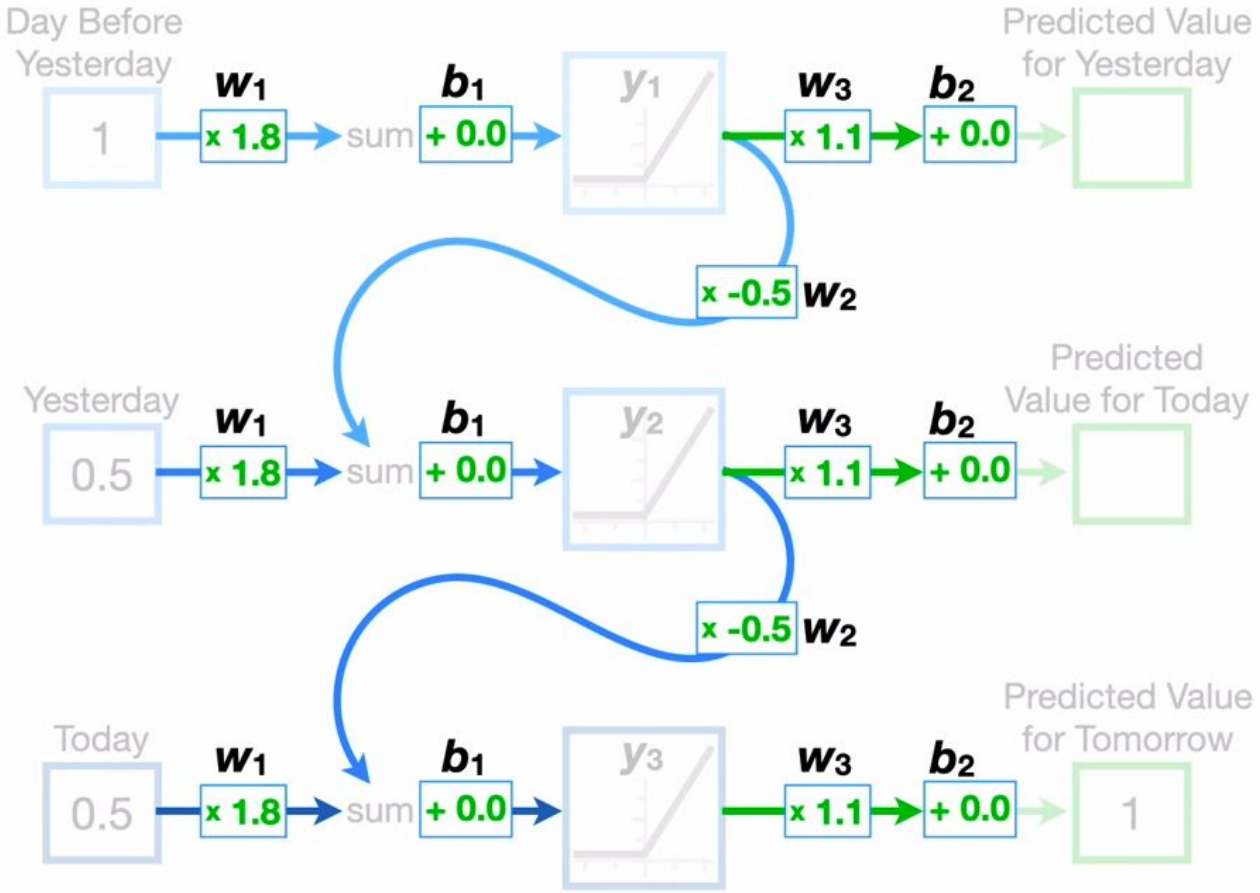
Likewise, all of the other **weights** and **biases** are shared.



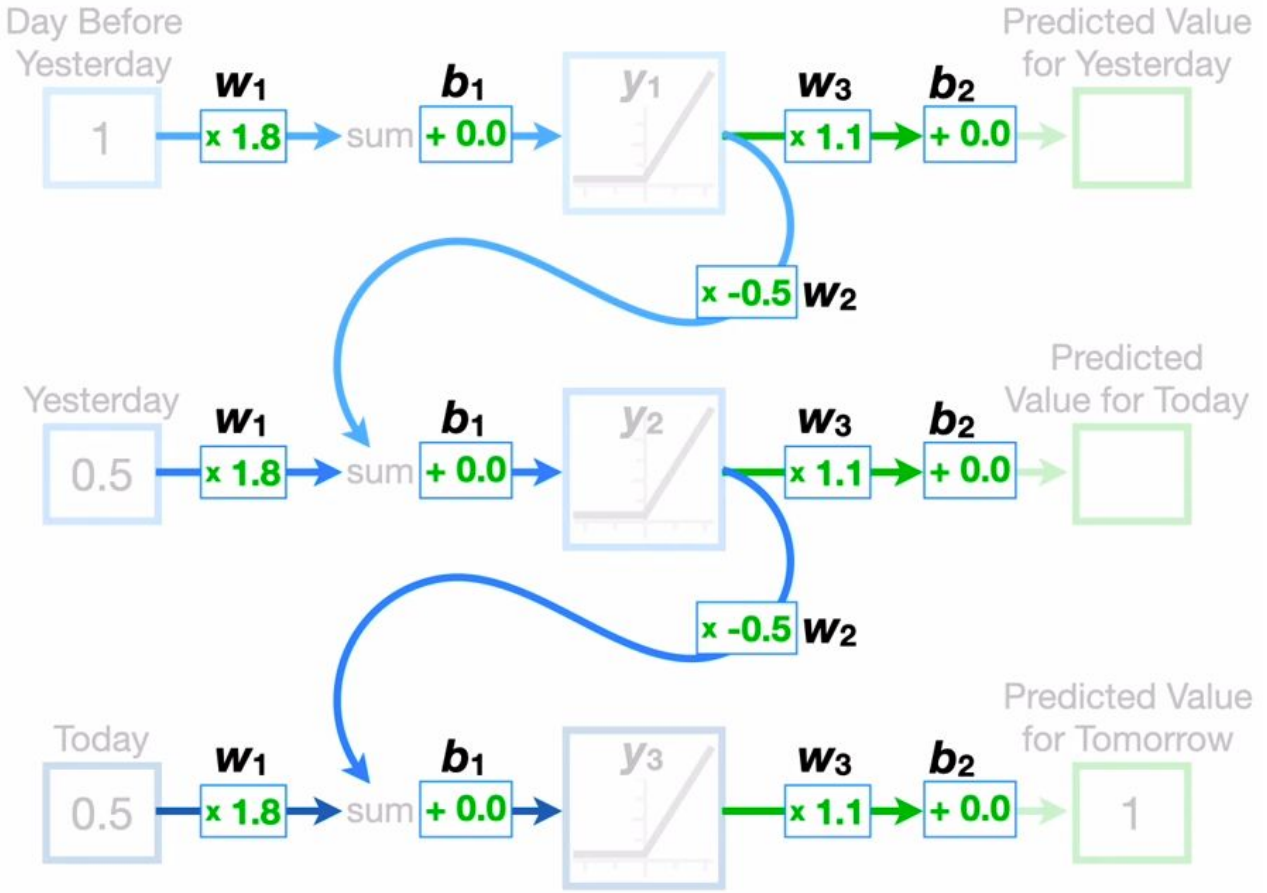
So, no matter how many times we **unroll** a recurrent neural network, we never increase the number of **weights** and **biases** that we have to train.



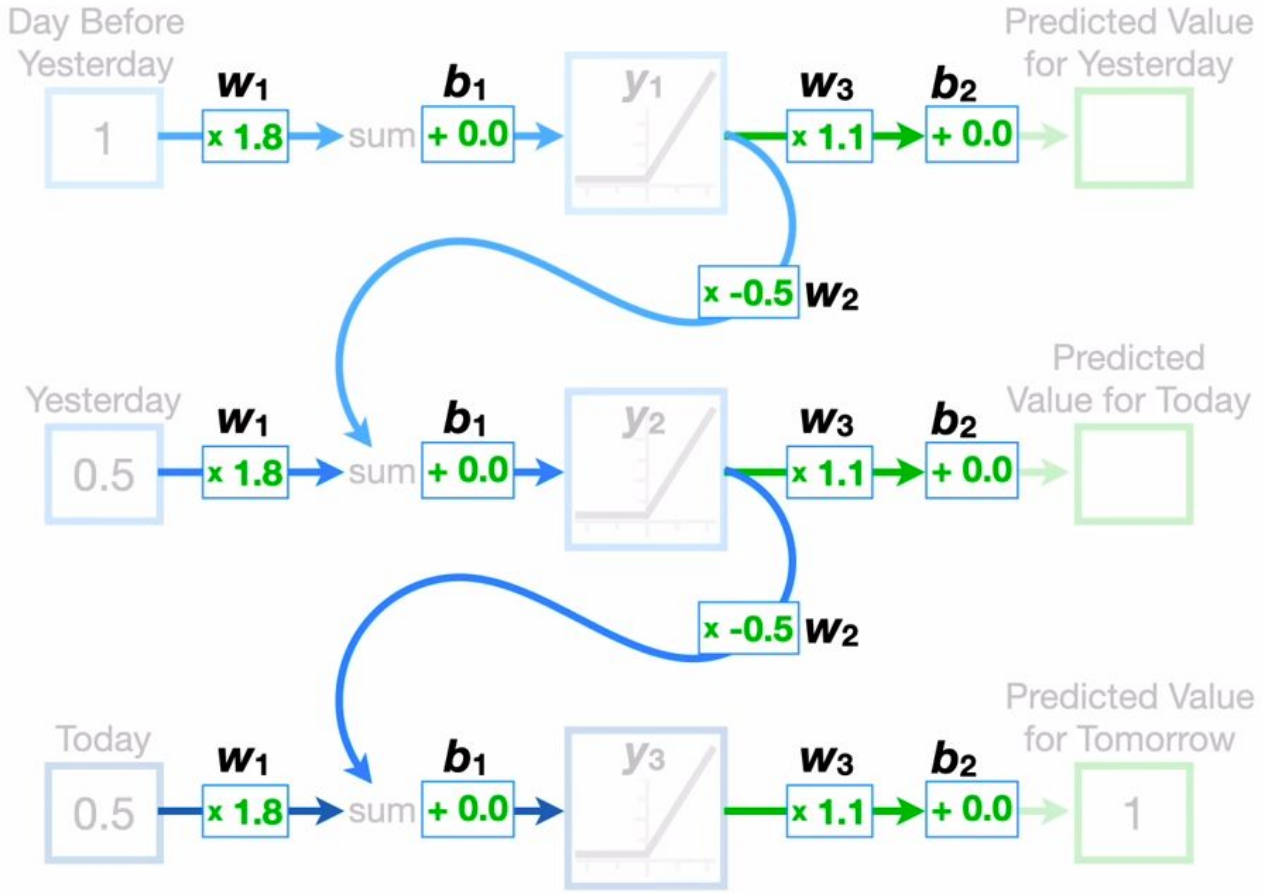
So, no matter how many times we **unroll** a recurrent neural network, we never increase the number of **weights** and **biases** that we have to train.



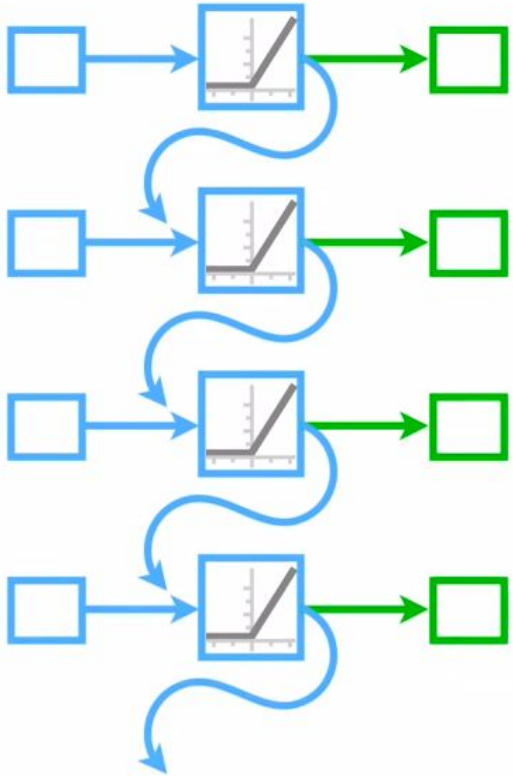
OK, now that we've talked about what makes ***basic*** recurrent neural networks so cool…



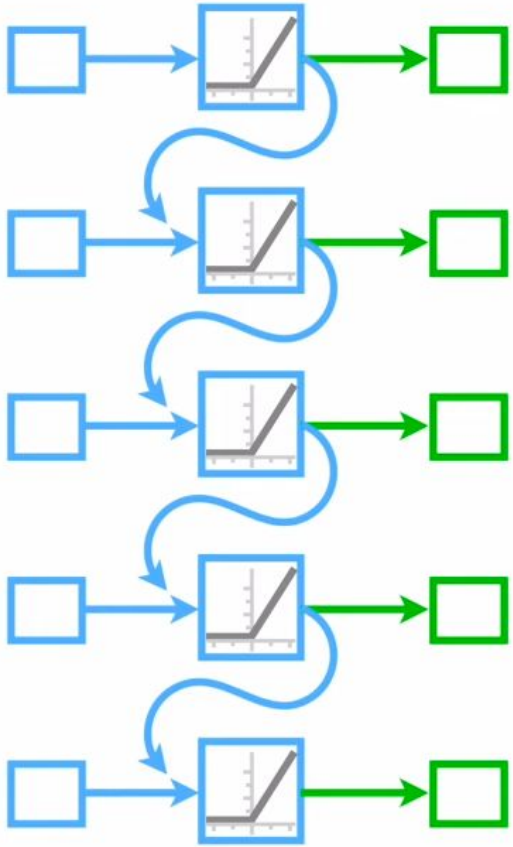
…let's briefly talk about why they are not used very often.



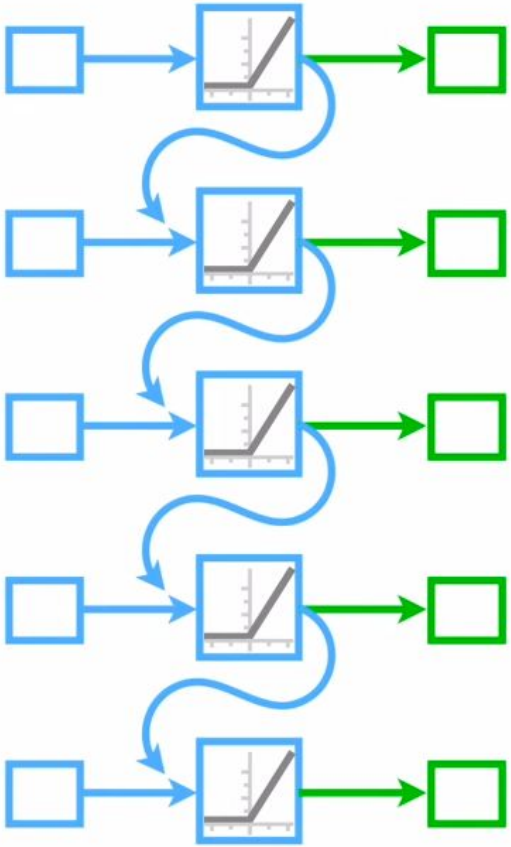
One big problem is that the more we **unroll** a recurrent neural network, the harder it is to train.



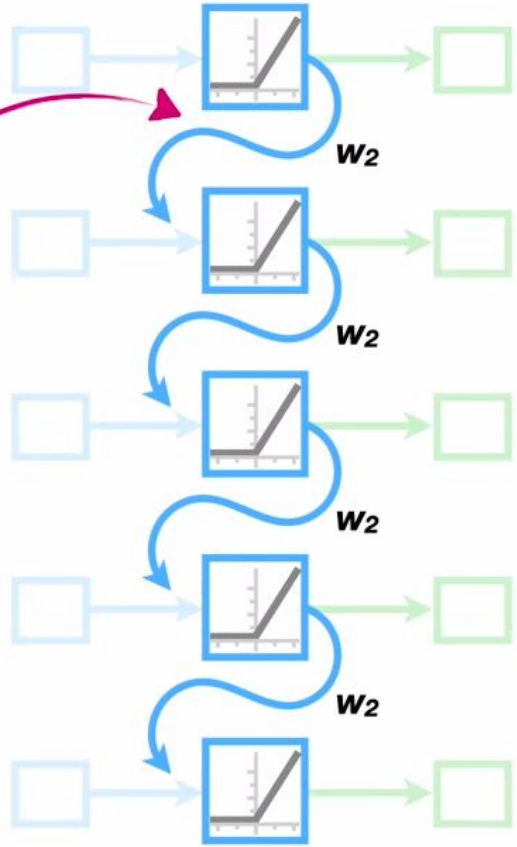
This problem is called **The Vanishing/Exploding Gradient Problem.**



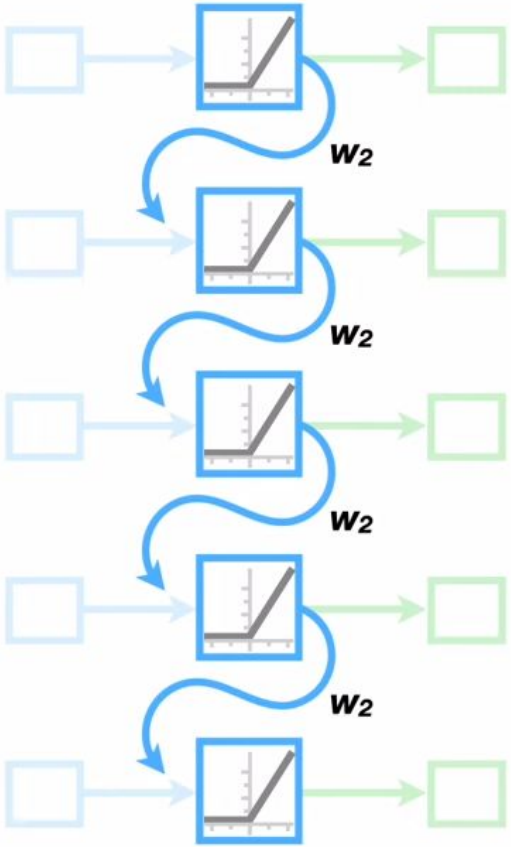
Which is also
known as the…



In our example, **The Vanishing/Exploding Gradient** problem has to do with the **weight** along the squiggle that we copy each time we **unroll** the network.



NOTE: To make it easier to understand **The Vanishing/ Exploding Gradient Problem**, we're going to ignore the other **weights** and **biases** in this network and just focus on w_2.

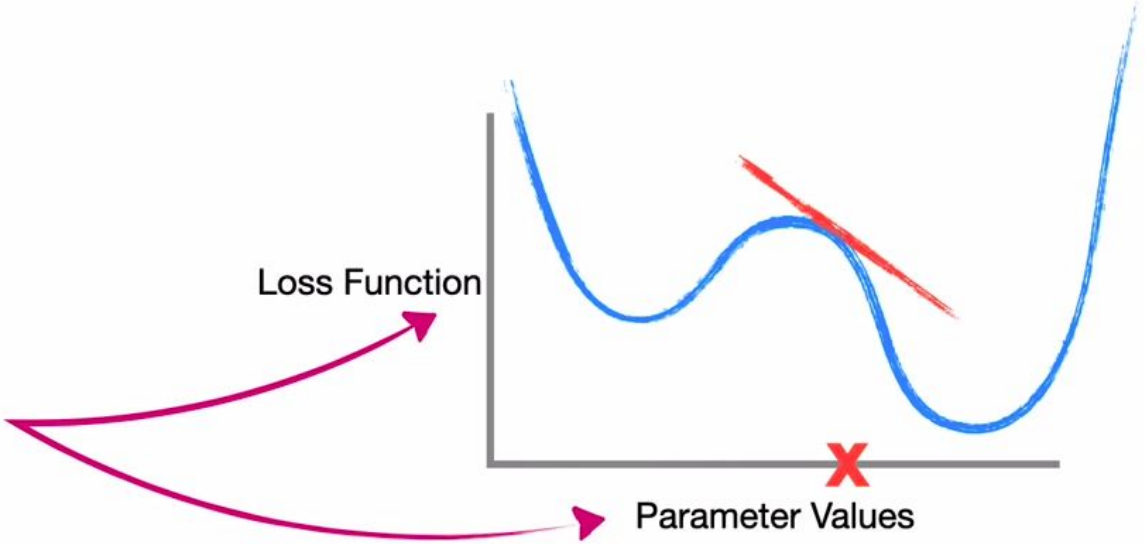


$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_2} \times \frac{d\,y_2}{d\,x_2} \times \frac{d\,x_2}{d\,w_1} + \ldots$$

Also, just to remind you, when we optimize neural networks with **Backpropagation**, we first find the derivatives, or **Gradients**, for each parameter.

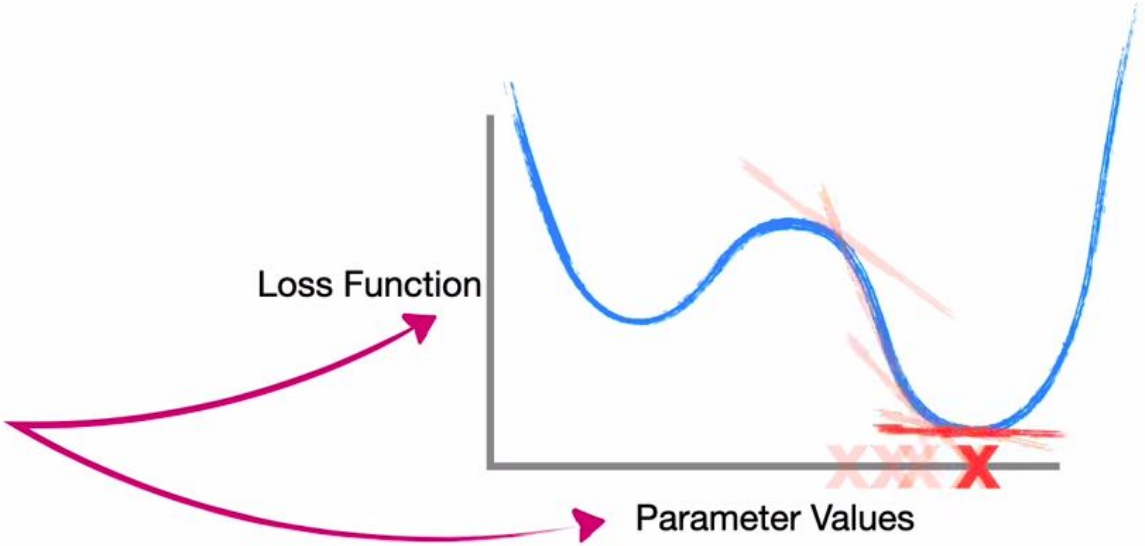
$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_2} \times \frac{d\,y_2}{d\,x_2} \times \frac{d\,x_2}{d\,w_1} + \ldots$$

We then plug those **Gradients** into the **Gradient Descent** algorithm to find the parameter values that minimize a **Loss Function**, like the **sum of the squared residuals**.

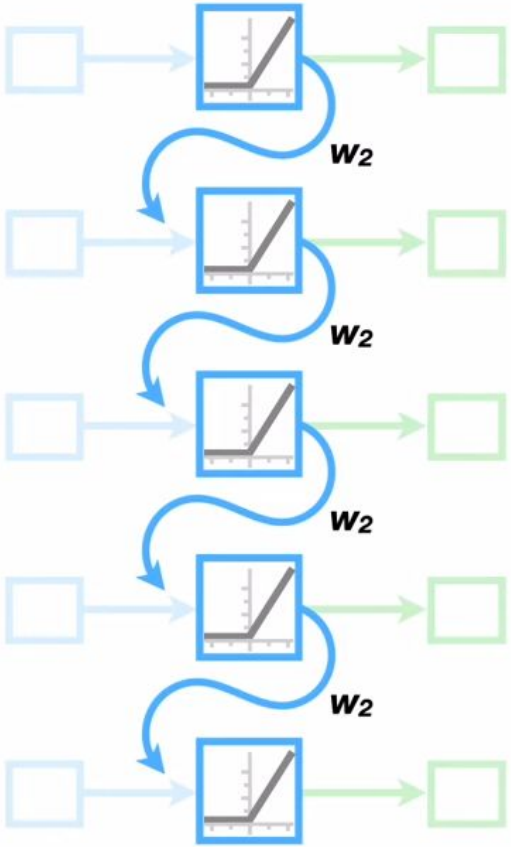


$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_2} \times \frac{d\,y_2}{d\,x_2} \times \frac{d\,x_2}{d\,w_1} + \ldots$$

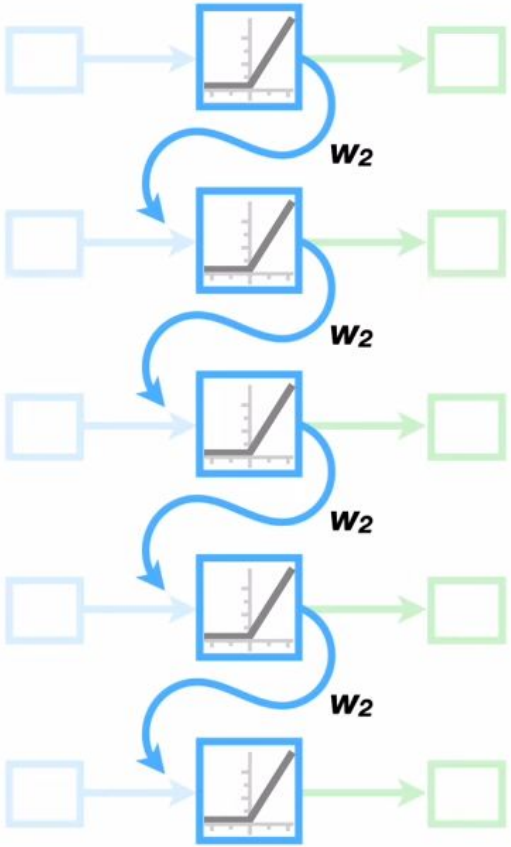
We then plug those **Gradients** into the **Gradient Descent** algorithm to find the parameter values that minimize a **Loss Function**, like the **sum of the squared residuals**.



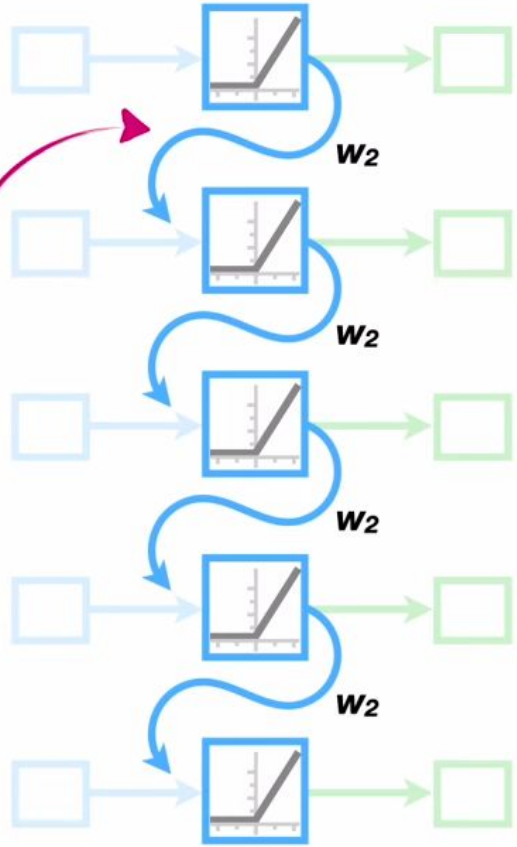
Now, even though **The Vanishing/ Exploding Gradient Problem** starts with ***Vanishing***…



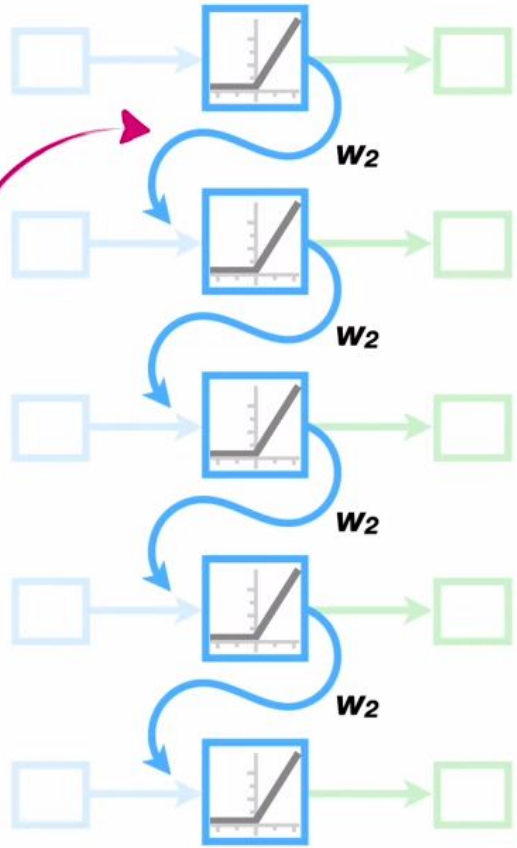
…we're going to start by showing how a gradient can ***Explode***.



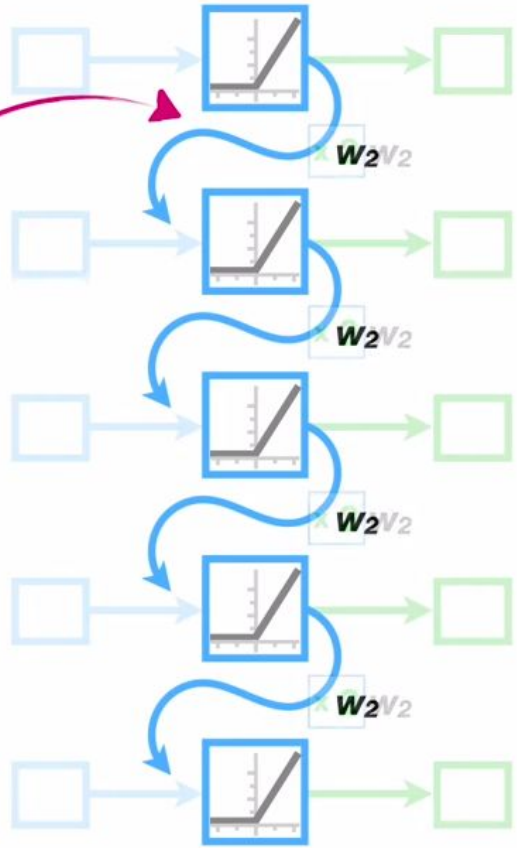
In our example, the gradient will ***explode***, when we set w_2 to any value larger than **1**.

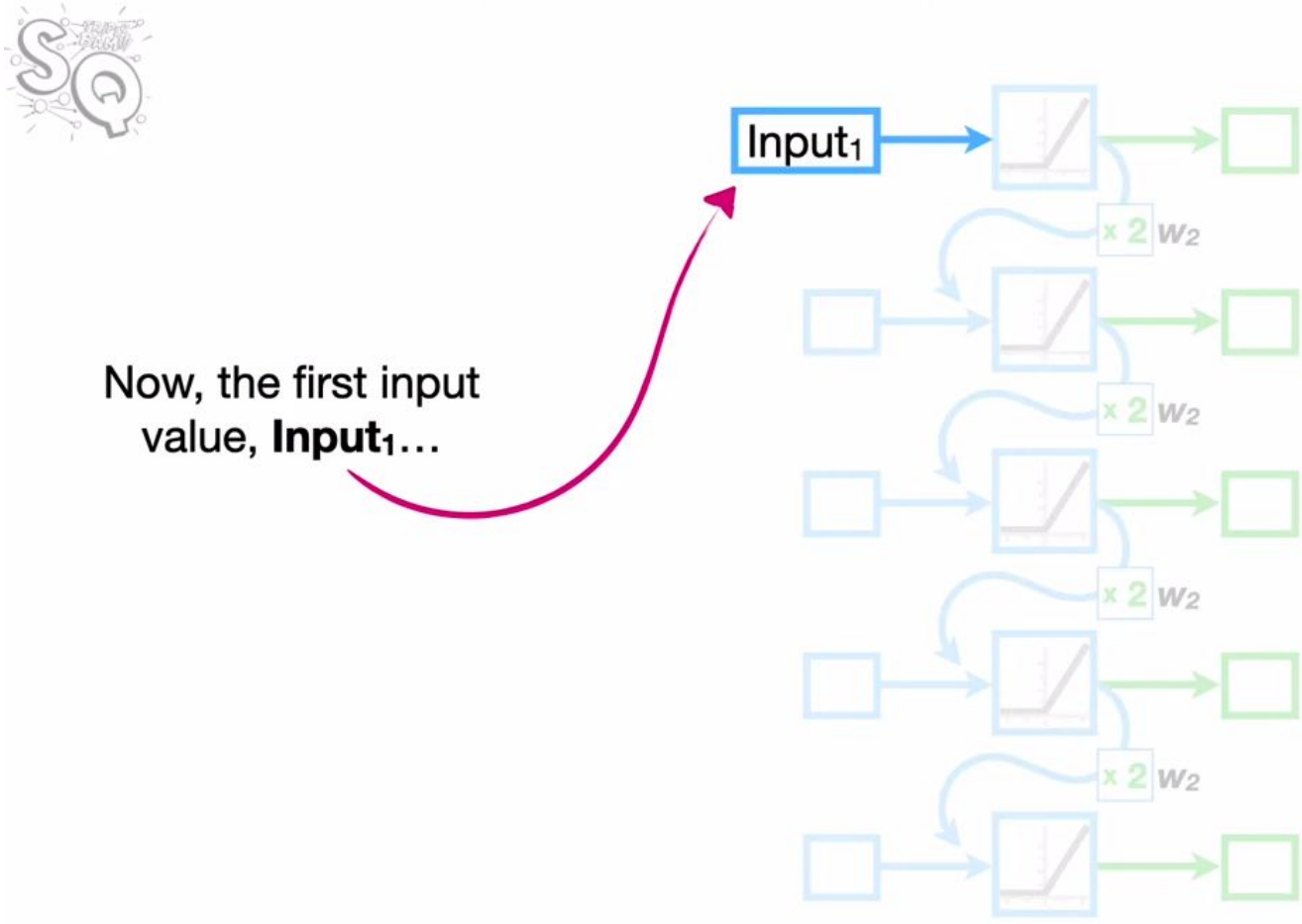


In our example, the gradient will ***explode***, when we set w_2 to any value larger than **1**.

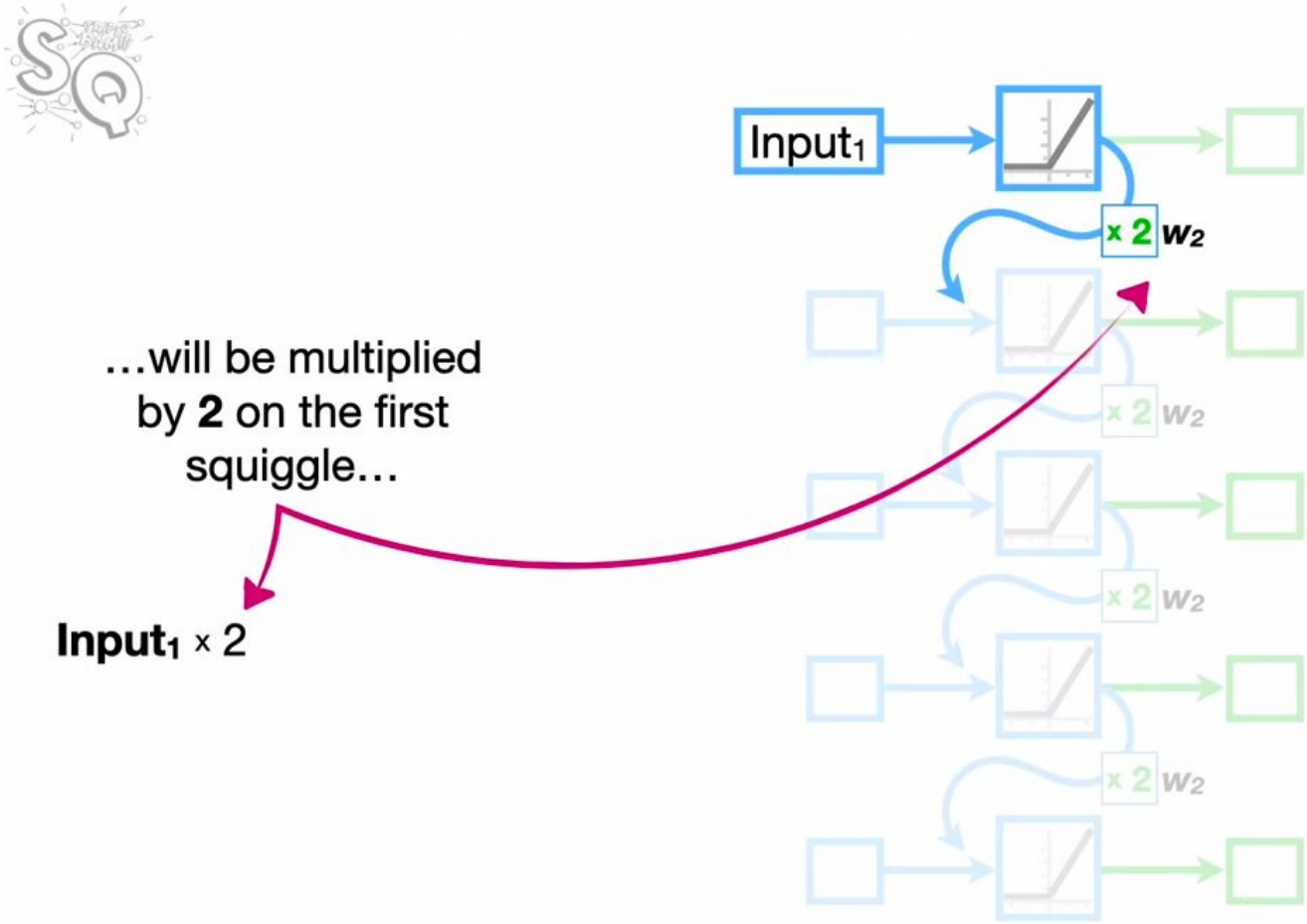


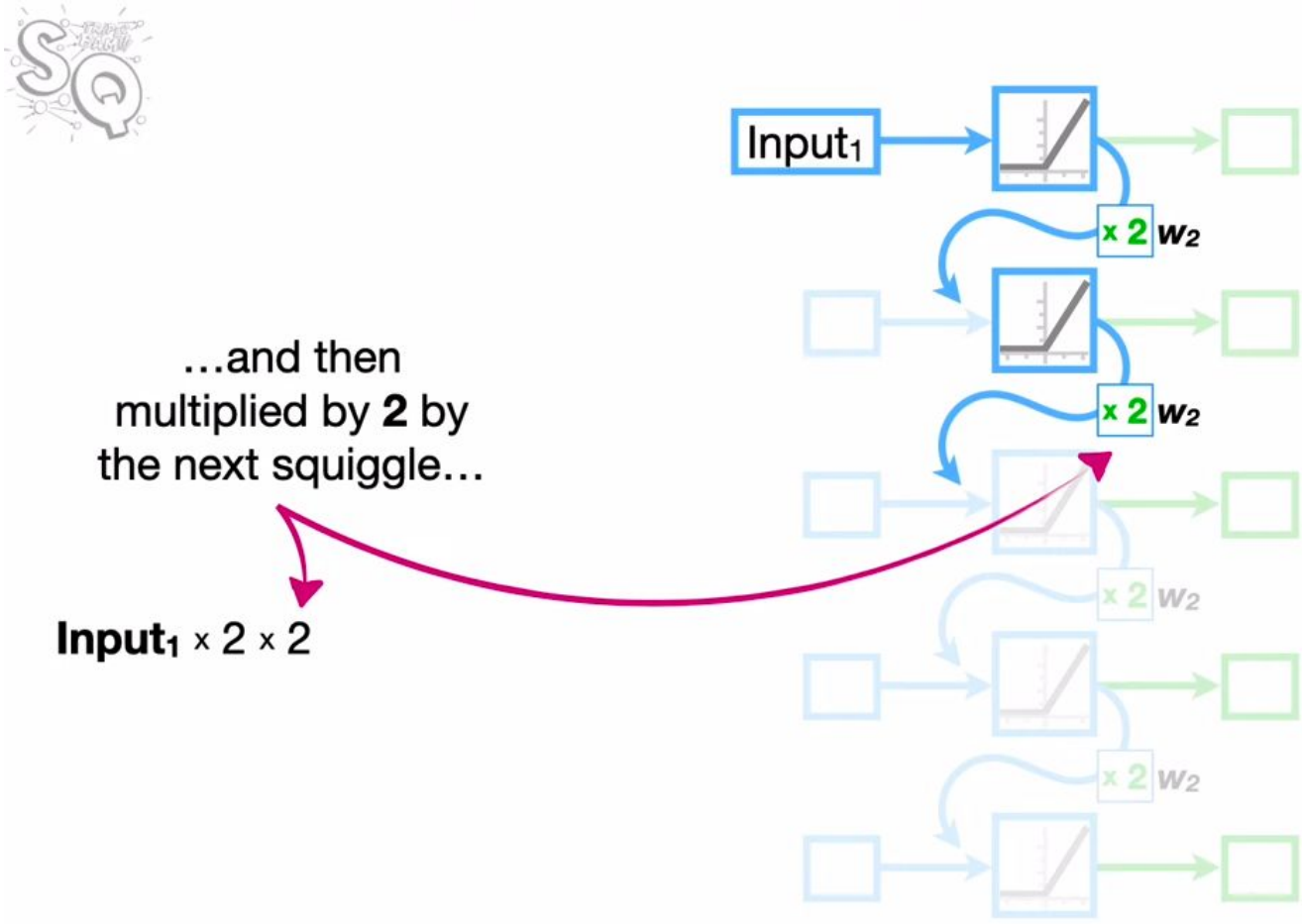
So let's set
$w_2 = 2$.

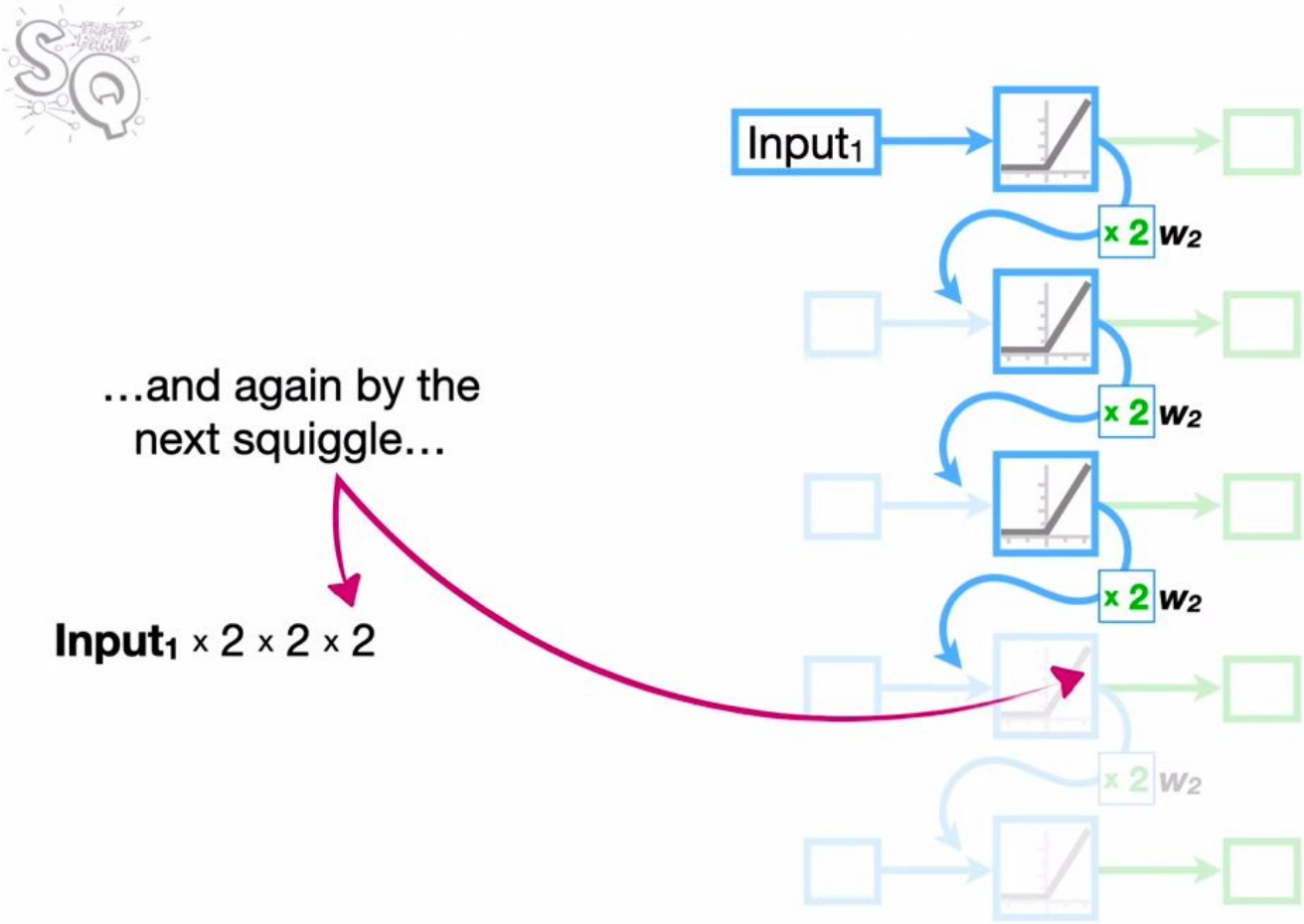


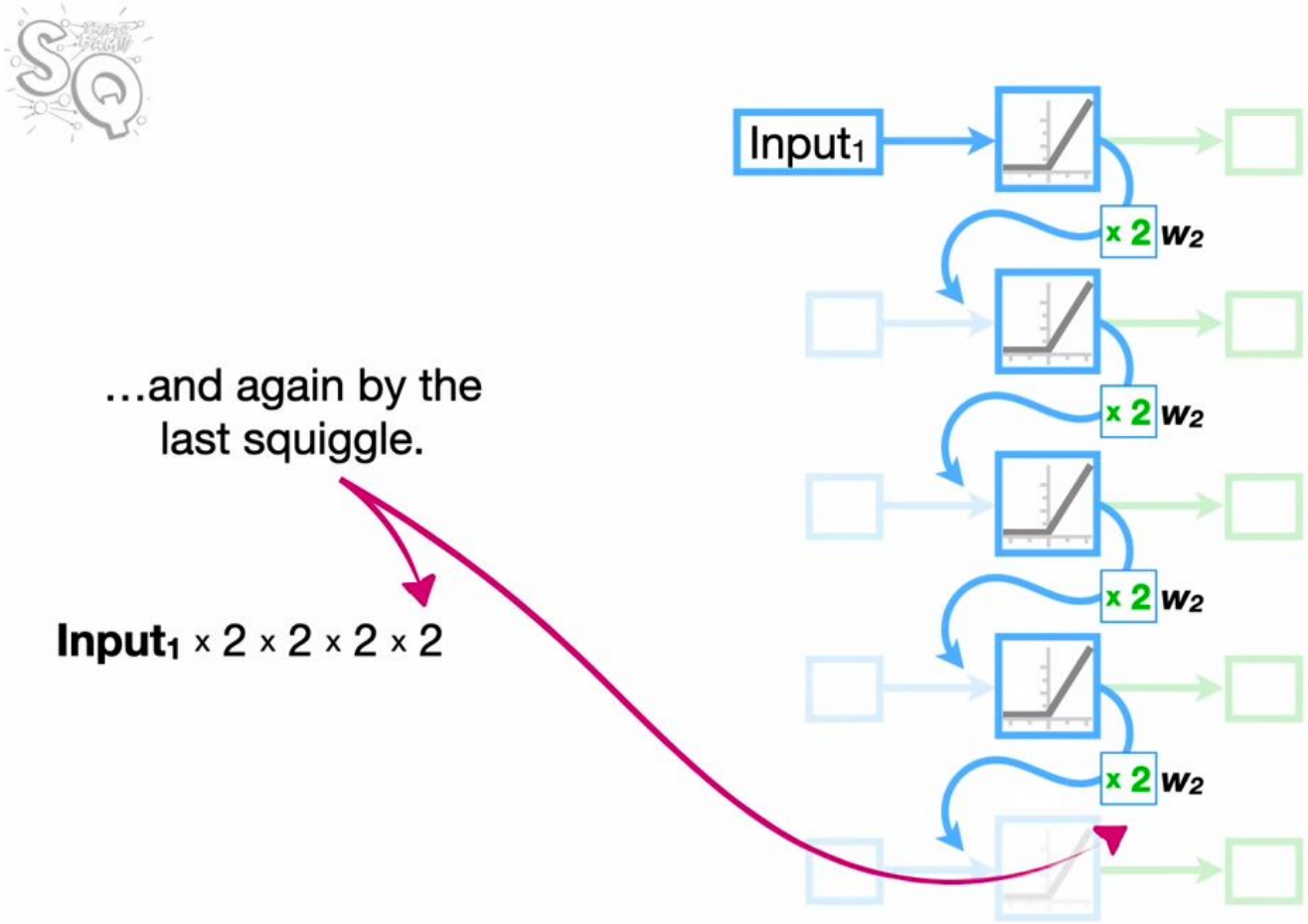


Now, the first input value, **Input$_1$**…


Input1
x 2 w2
x 2 w2
x 2 w2
x 2 w2
…will be multiplied by 2 on the first squiggle…
Input1 x 2

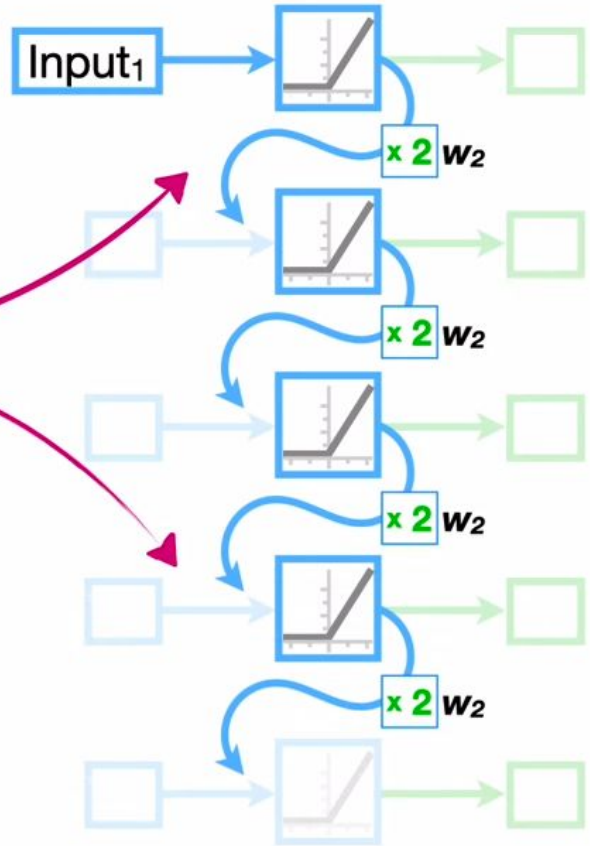

Input1
x 2 w2
x 2 w2
x 2 w2
x 2 w2
…and then multiplied by 2 by the next squiggle…
Input1 x 2 x 2


Input1
x 2 w2
x 2 w2
x 2 w2
x 2 w2
…and again by the next squiggle…
Input1 x 2 x 2 x 2


Input1
x 2 w2
x 2 w2
x 2 w2
x 2 w2
…and again by the last squiggle.
Input1 x 2 x 2 x 2 x 2

In other words, since we **unrolled** the recurrent neural network **4** times…

Input$_1$ x 2 x 2 x 2 x 2



…we multiply the input value by w_2, which is **2**, raised to the number of times we **unrolled**, which is **4**.

$$\text{Input}_1 \times 2 \times 2 \times 2 \times 2$$

$$= \text{Input}_1 \times 2^4$$

$$= \text{Input}_1 \times w_2^{\text{Num. Unroll}}$$

And that means the first input value is amplified **16** times before it gets to the final copy of the network.

$\mathbf{Input_1} \times 2 \times 2 \times 2 \times 2$

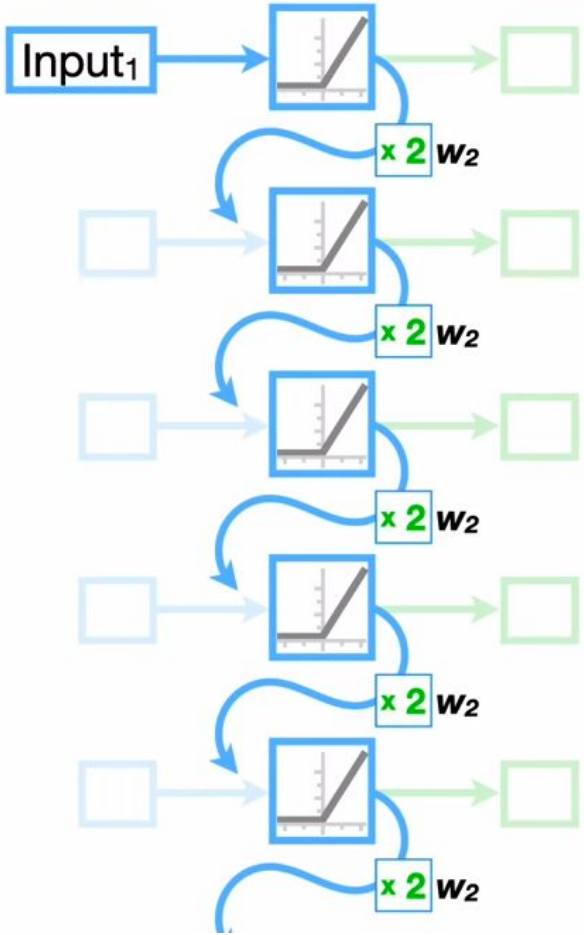
$= \mathbf{Input_1} \times 2^4$

$= \mathbf{Input_1} \times \boldsymbol{w_2}^{\text{Num. Unroll}}$

Now, if we had **50** sequential days of stock market data, which, to be honest, really isn't that much data…

= **Input**$_1$ × 2^{50}

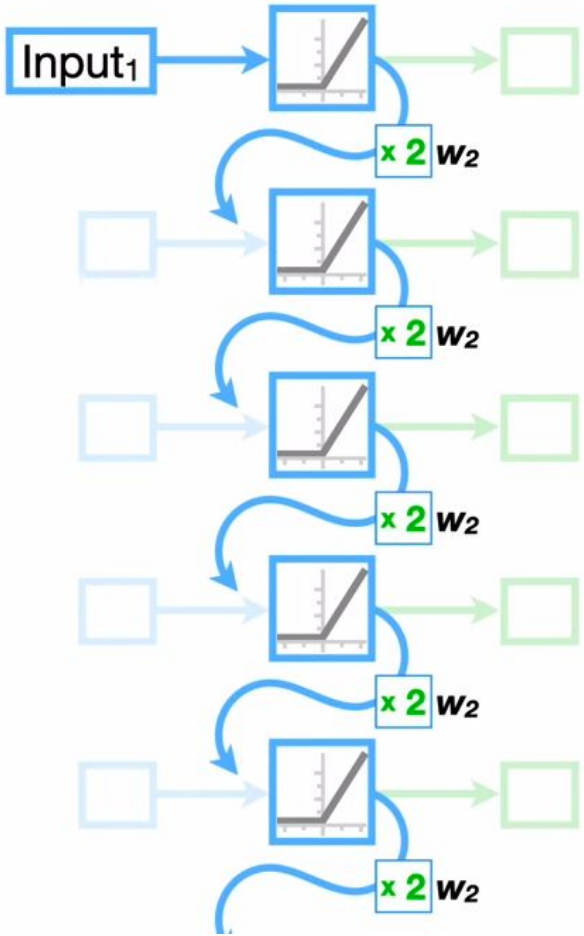
= **Input**$_1$ × $w_2^{\text{Num. Unroll}}$



…then we would **unroll** the network **50** times…

= **Input**$_1$ × 2^{50}

= **Input**$_1$ × $w_2^{\text{Num. Unroll}}$



…and 2^{50} is
A HUGE NUMBER.

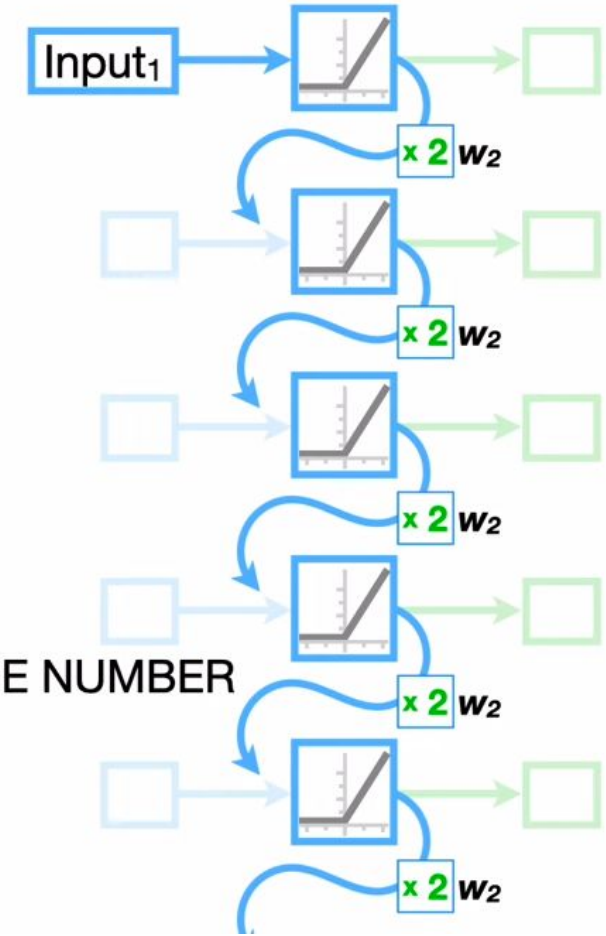
$= \textbf{Input}_1 \times 2^{50} = \textbf{Input}_1 \times$ A HUGE NUMBER

$= \textbf{Input}_1 \times w_2^{\text{Num. Unroll}}$

And this **HUGE NUMBER** is why they call this an **Exploding Gradient Problem.**

= **Input**$_1$ × 2^{50} = **Input**$_1$ × A HUGE NUMBER

= **Input**$_1$ × $w_2^{\text{Num. Unroll}}$



$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_1} \times \frac{d\,y_1}{d\,x_1} \times \frac{d\,x_1}{d\,w_1} + \ldots$$

If we tried to train this recurrent neural network with backpropagation, this **HUGE NUMBER** would find its way into some of the gradients…

$= \mathbf{Input_1} \times 2^{50} = \mathbf{Input_1} \times$ A HUGE NUMBER

$= \mathbf{Input_1} \times \boldsymbol{w_2}^{\text{Num. Unroll}}$

$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_1} \times \frac{d\,y_1}{d\,x_1} \times \frac{d\,x_1}{d\,w_1} + \ldots$$

If we tried to train this recurrent neural network with backpropagation, this **HUGE NUMBER** would find its way into some of the gradients…

$$\frac{d\,x_1}{d\,w_1} = \frac{d}{d\,w_1}\,\text{Input1} \times w_1 \times w_2^{50} \ldots$$

$$= \textbf{Input}_1 \times 2^{50} = \textbf{Input}_1 \times \text{A HUGE NUMBER}$$

$$= \textbf{Input}_1 \times w_2^{\text{Num. Unroll}}$$

$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_1} \times \frac{d\,y_1}{d\,x_1} \times \frac{d\,x_1}{d\,w_1} + \ldots$$

If we tried to train this recurrent neural network with backpropagation, this **HUGE NUMBER** would find its way into some of the gradients…

$$\frac{d\,x_1}{d\,w_1} = \frac{d}{d\,w_1}\,\text{Input1} \times w_1 \times w_2^{50}\ldots$$

$$= \frac{d}{d\,w_1}\,\text{Input1} \times w_1 \times \text{A HUGE NUMBER}\ldots$$

$$= \textbf{Input}_1 \times 2^{50} = \textbf{Input}_1 \times \text{A HUGE NUMBER}$$

$$= \textbf{Input}_1 \times w_2^{\text{Num. Unroll}}$$

$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_1} \times \frac{d\,y_1}{d\,x_1} \times \frac{d\,x_1}{d\,w_1} + \ldots$$

…and that would make it hard to take small steps to find the optimal **weights** and **biases**.

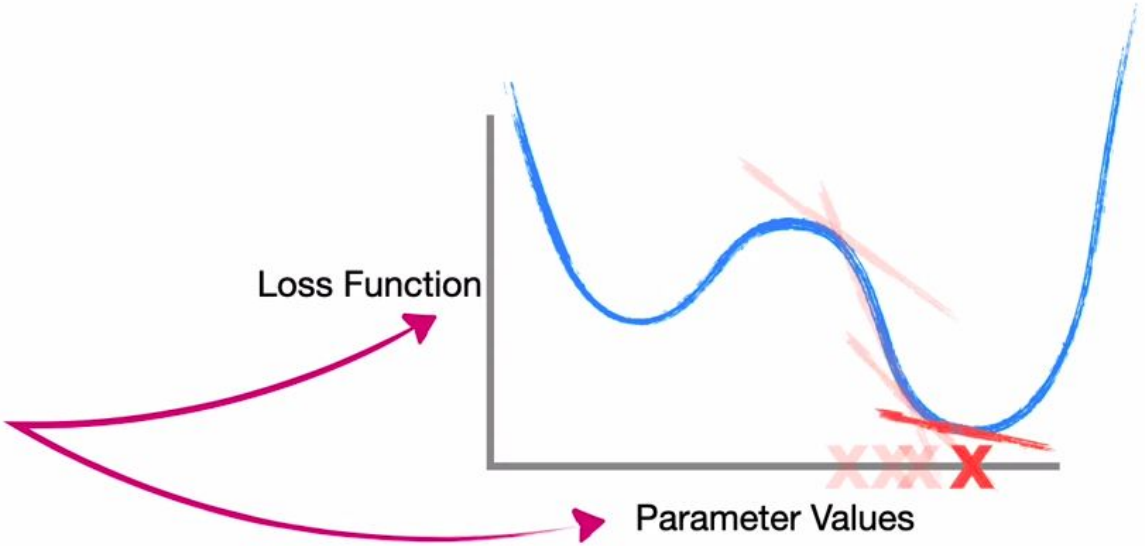
$$\frac{d\,x_1}{d\,w_1} = \frac{d}{d\,w_1}\,\text{Input1} \times w_1 \times w_2^{50}\ldots$$

$$= \frac{d}{d\,w_1}\,\text{Input1} \times w_1 \times \text{A HUGE NUMBER}\ldots$$

$= \textbf{Input}_1 \times 2^{50} = \textbf{Input}_1 \times \text{A HUGE NUMBER}$

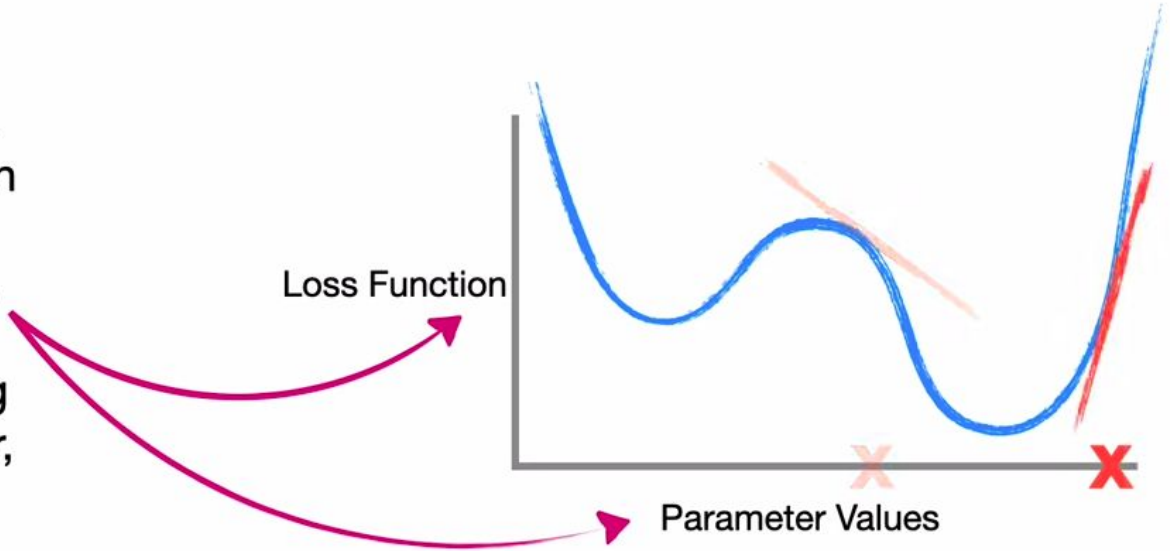
$= \textbf{Input}_1 \times w_2^{\text{Num. Unroll}}$

In other words, in order to find the parameter values that give us the lowest value for the **Loss Function**, we usually want to take relatively small steps.



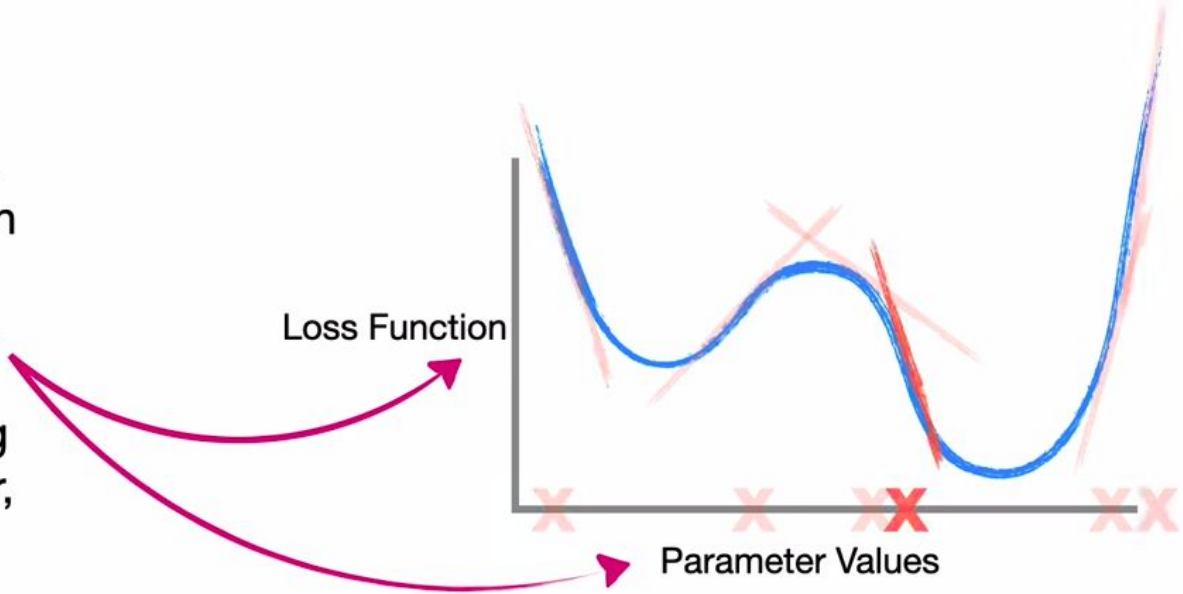
However, when the **Gradient** contains **A HUGE NUMBER**, then we'll end up taking relatively large steps.

And instead of finding the optimal parameter, we'll just bounce around a lot.



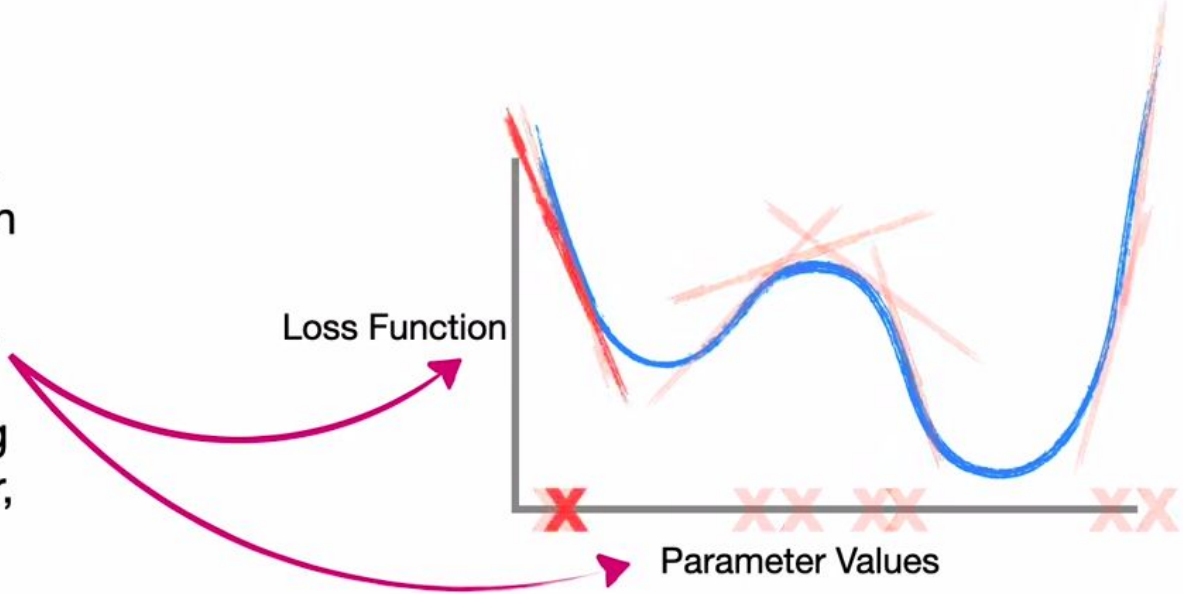
However, when the **Gradient** contains **A HUGE NUMBER**, then we'll end up taking relatively large steps.

And instead of finding the optimal parameter, we'll just bounce around a lot.

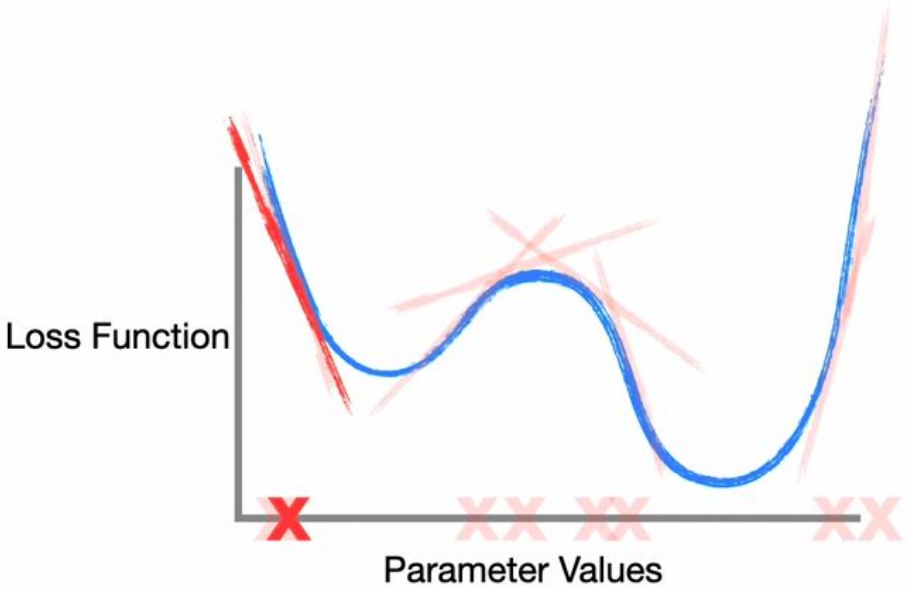


However, when the **Gradient** contains **A HUGE NUMBER**, then we'll end up taking relatively large steps.

And instead of finding the optimal parameter, we'll just bounce around a lot.

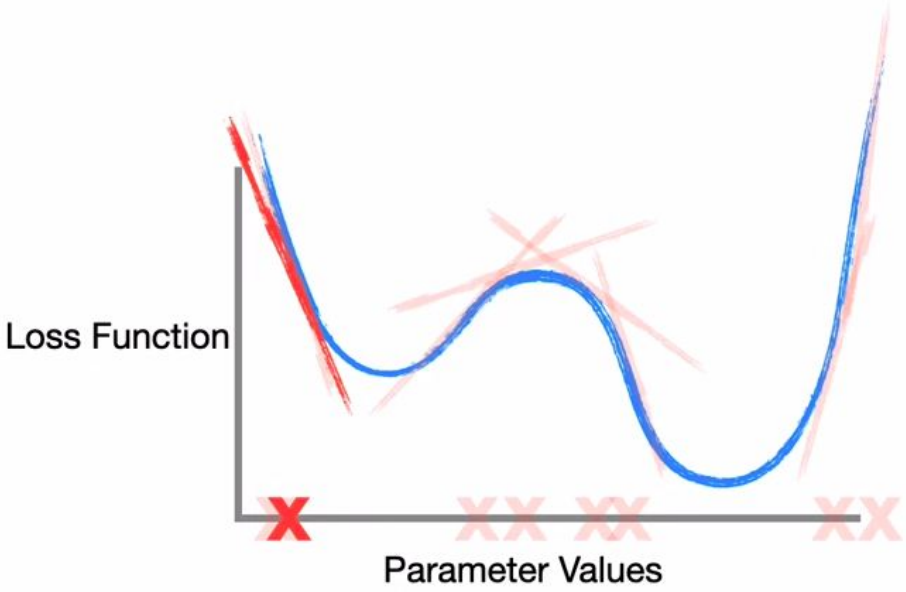


One way to prevent **The Exploding Gradient Problem** would be to limit w_2 to values **< 1**.



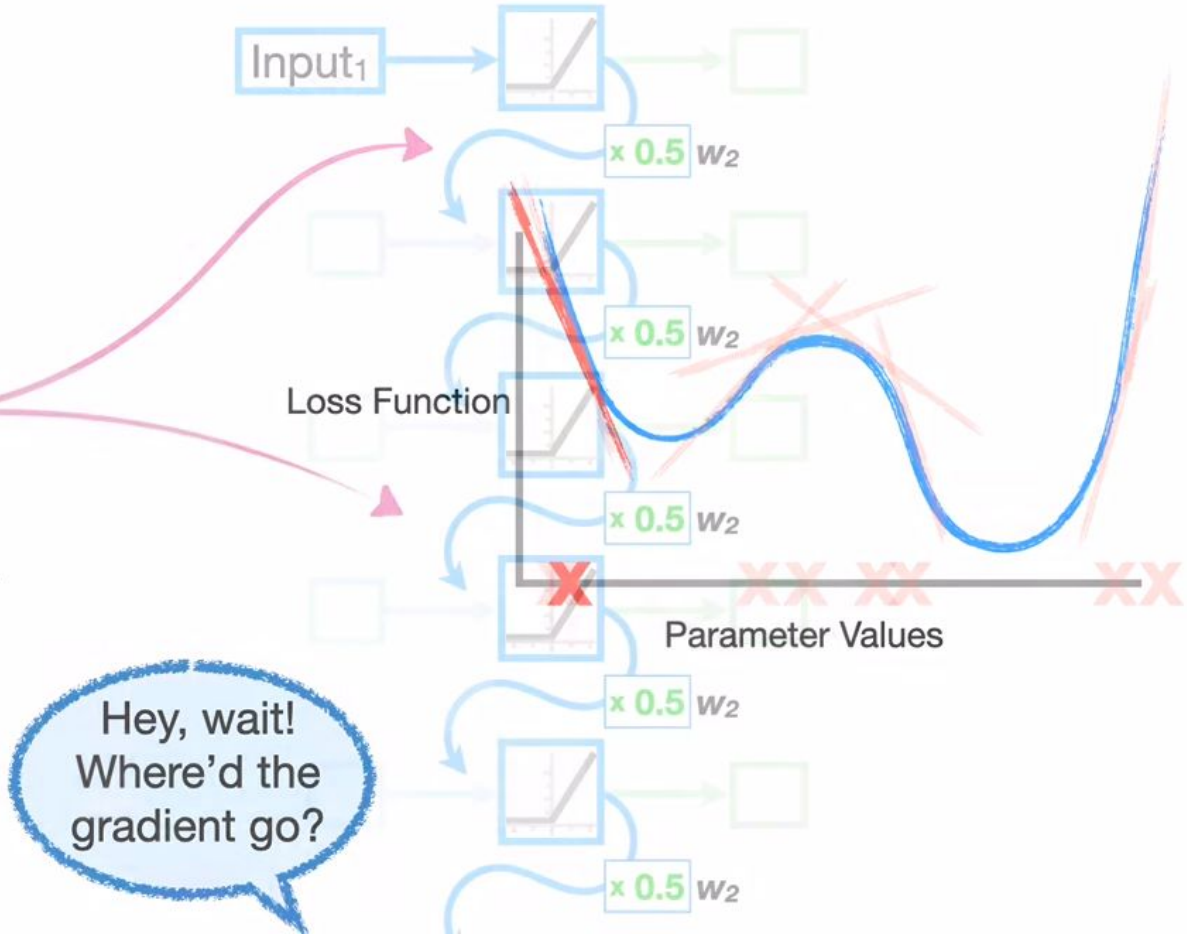
One way to prevent **The Exploding Gradient Problem** would be to limit w_2 to values **< 1**.

However, this results in **The Vanishing Gradient Problem**.

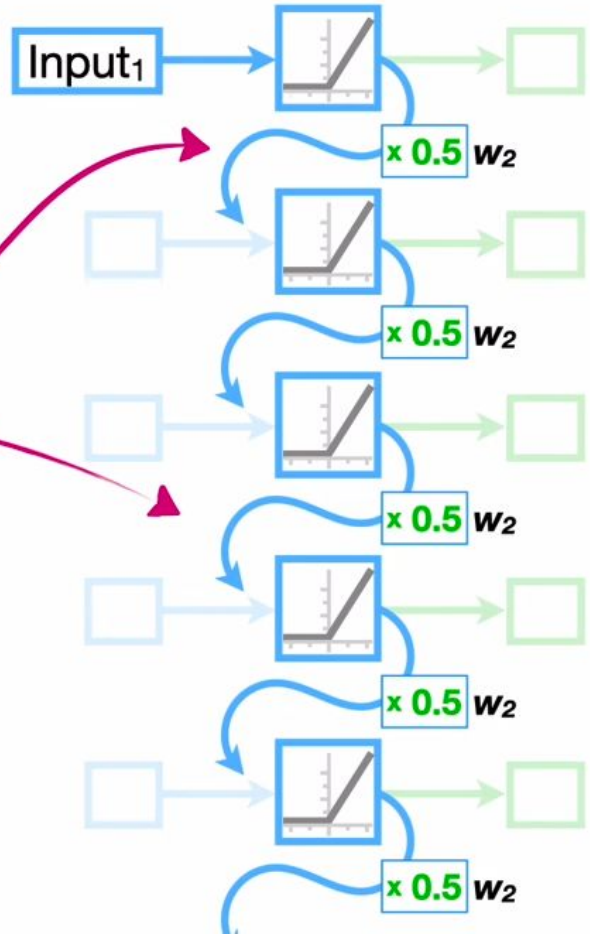


One way to prevent **The Exploding Gradient Problem** would be to limit w_2 to values **< 1**.

However, this results in **The Vanishing Gradient Problem**.



To illustrate **The Vanishing Gradient Problem**, let's set $w_2 = 0.5$.



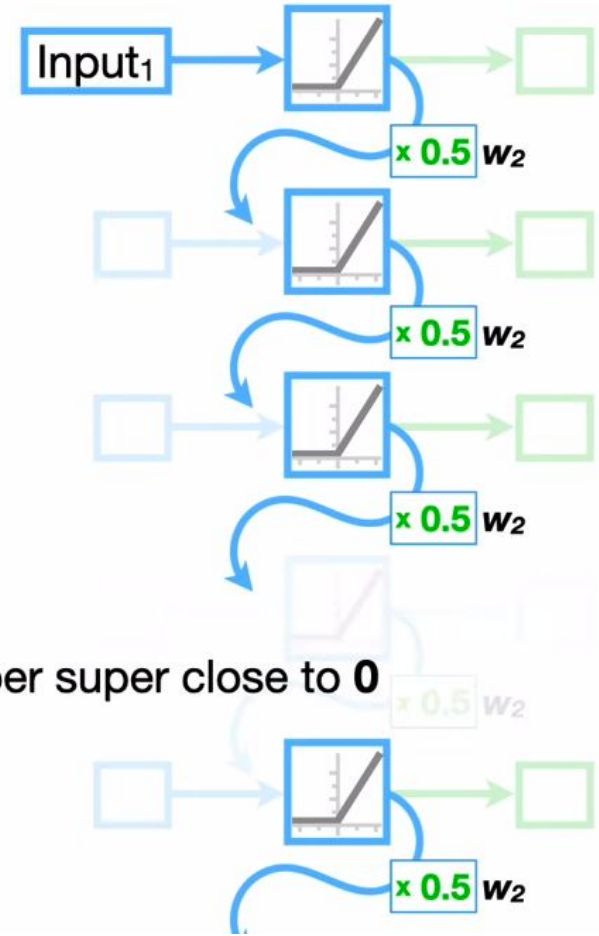
Now, just like before, we multiply the first input by w_2 raised to the number of times we **unroll** the network.

$$\textbf{Input}_1 \times w_2^{\text{Num. Unroll}}$$

So if we have **50** sequential input values, that means multiplying **Input$_1$** by **0.5^{50}**…

Input$_1$ × 0.5^{50}

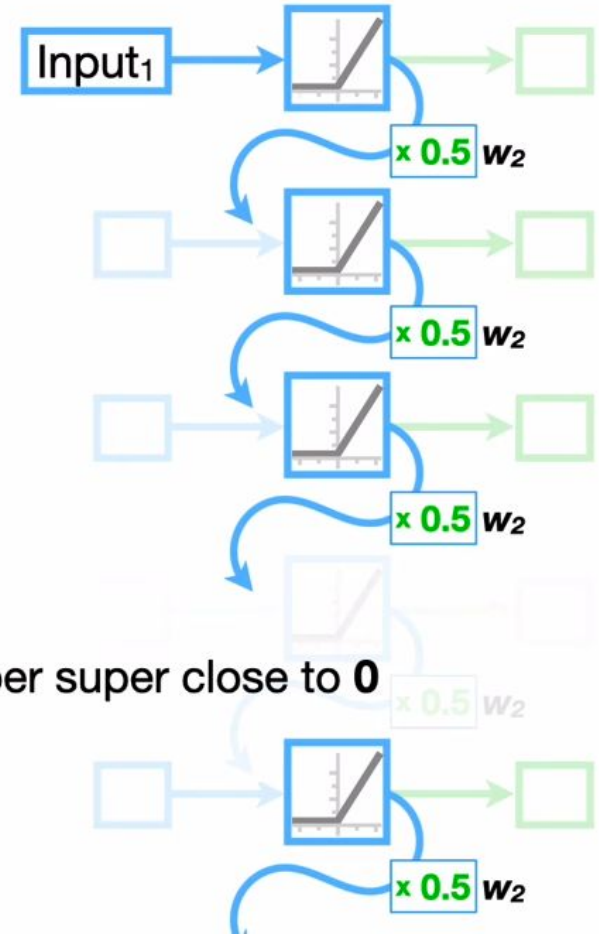
Input$_1$ × $w_2^{\text{Num. Unroll}}$



…and **0.5^{50}** is **a number super close to 0**.

$Input_1$ × 0.5^{50} = **$Input_1$** × a number super close to **0**

$Input_1$ × $w_2^{\text{Num. Unroll}}$

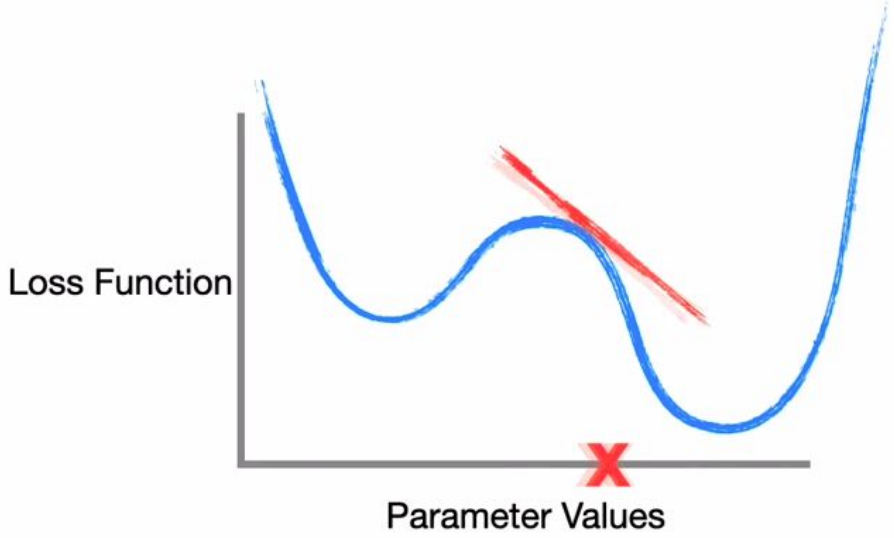


Because this number is **super close to 0**, this is called the **Vanishing Gradient Problem**.

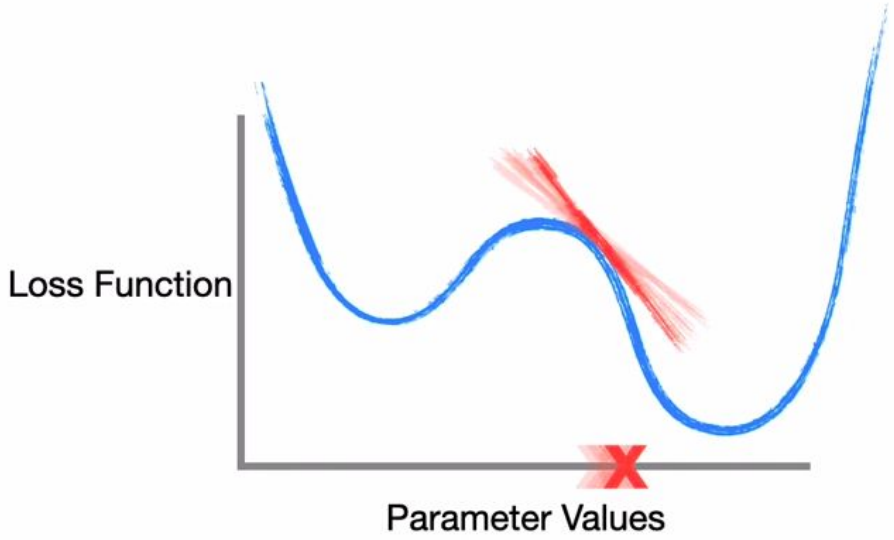
Input$_1$ × 0.5^{50} = **Input**$_1$ × a number super close to **0**

Input$_1$ × $w_2^{\text{Num. Unroll}}$

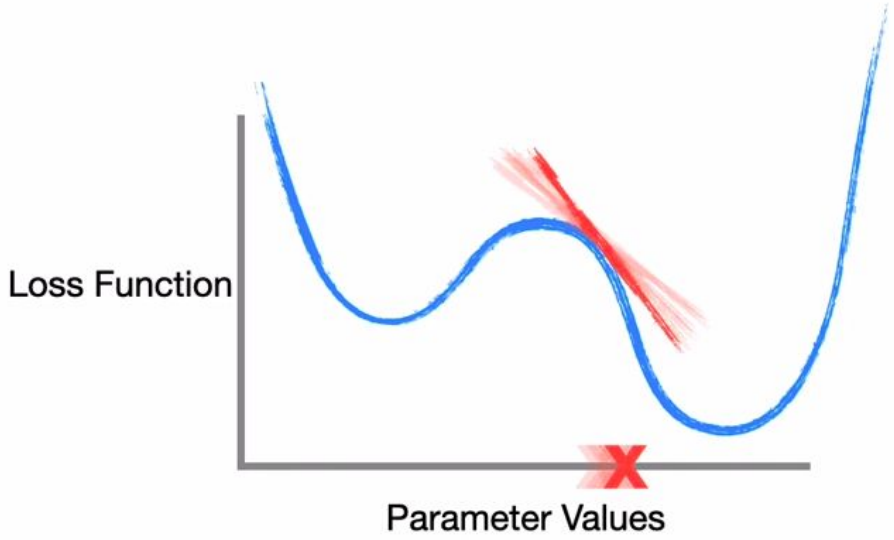
Now, when optimizing a parameter, instead of taking steps that are too large, we end up taking steps that are too small.



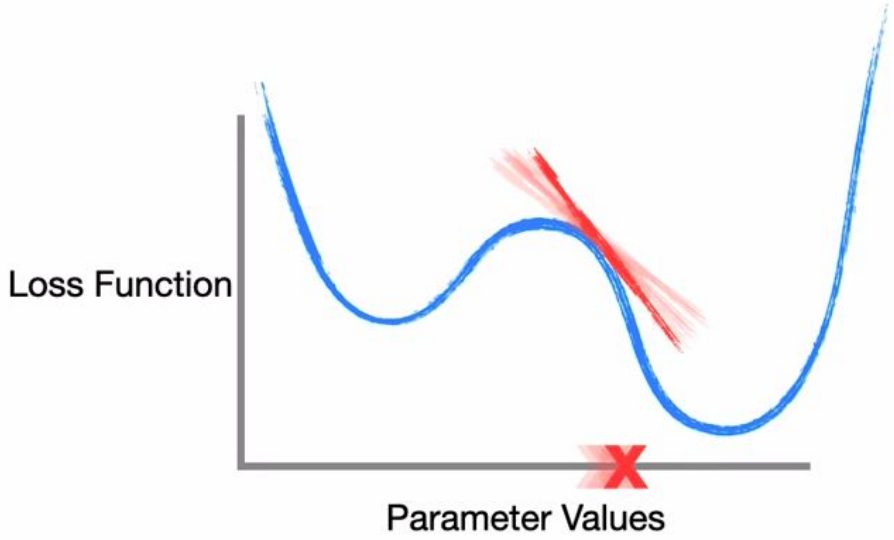
Now, when optimizing a parameter, instead of taking steps that are too large, we end up taking steps that are too small.



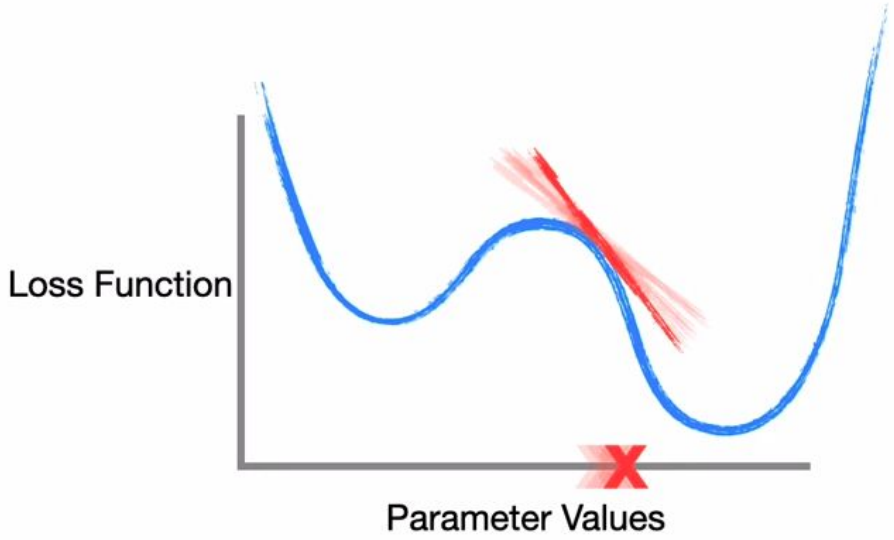
Now, when optimizing a parameter, instead of taking steps that are too large, we end up taking steps that are too small.



And as a result, we end up hitting the maximum number of steps we are allowed to take before we find the optimal value.

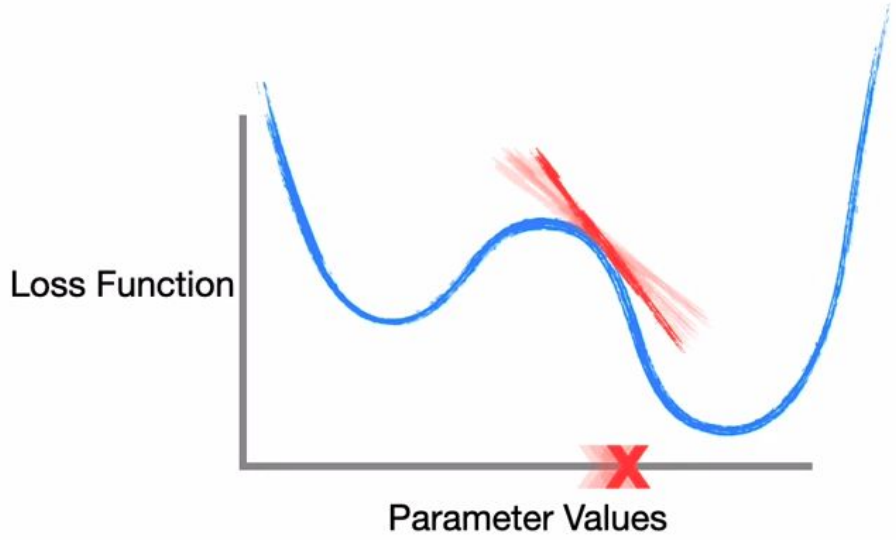


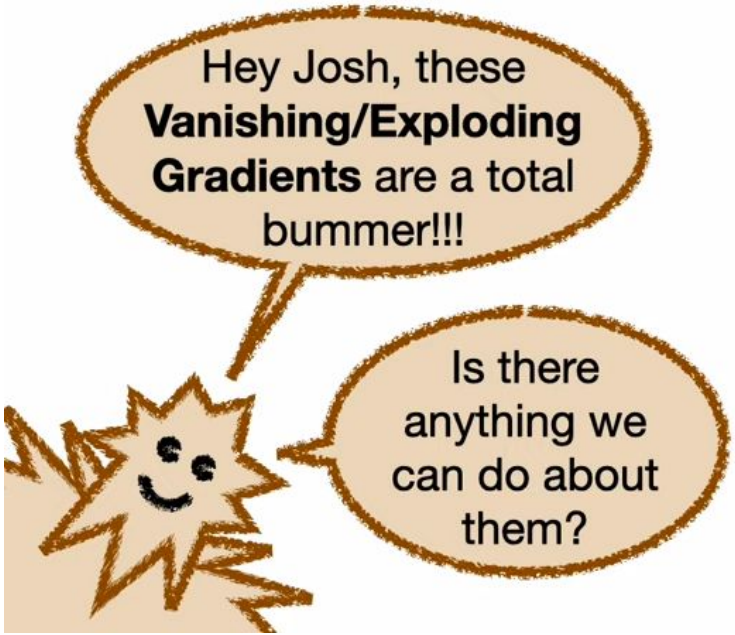
And as a result, we end up hitting the maximum number of steps we are allowed to take before we find the optimal value.



And as a result, we end up hitting the maximum number of steps we are allowed to take before we find the optimal value.

:(





YES! And we'll talk about a popular solution called **Long Short-Term Memory Networks** in the next **StatQuest!!!**