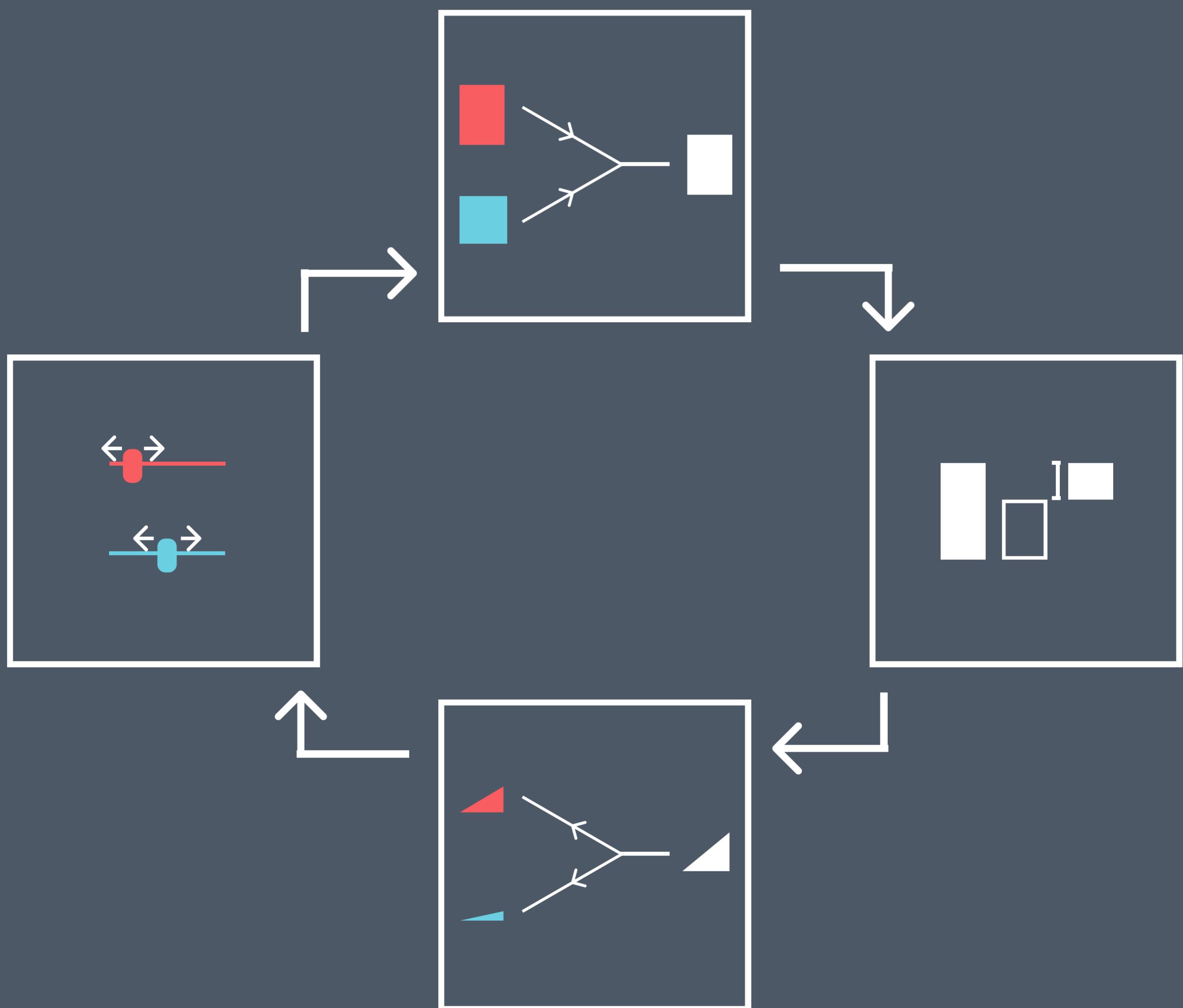


# A VISUAL INTRODUCTION TO DEEP LEARNING



## ABOUT THIS BOOK

Deep learning is the algorithm powering the current renaissance of artificial intelligence (AI). And its progress is not showing signs of slowing down. A McKinsey report estimates that by 2030, AI will potentially deliver \$13 trillion to the global economy, or 16% of the world's current GDP. This opens up exciting career opportunities in the coming decade.

But deep learning can be quite daunting to learn. With the abundance of learning resources in recent years has emerged another problem—information overload.

This book aims to compress this knowledge and make the subject approachable. By the end of this book, you will be able to build a visual intuition about deep learning and neural networks.

## WHO SHOULD READ THIS BOOK

If you are new to deep learning, or machine learning in general.

If you already know some background about deep learning but want to gain further intuition.

## BOOK FORMAT

This book uses a visuals-first approach. Each page of this book begins with a visual and is supported by concise text.

This book doesn't include math derivations and code examples. There are some parts where basic math is involved, but generally it is kept to a minimum.

## ABOUT THE AUTHOR

My journey into AI began in 2010 after my son was born with a limb difference. I became interested in machine learning for prosthetics and did an MSc at Imperial College London majoring in neurotechnology.

I have also worked in the telecoms data analytics space, serving clients in over 15 countries.

Above all, I am passionate about education and how we learn. I am currently working on projects that explore ways to create alternative learning experiences using visuals, storytelling, and games.



email: [contact@kdimensions.com](mailto:contact@kdimensions.com)

# TABLE OF CONTENTS

<b>INTRODUCTION</b>																																																																																																																																							
• PREDICTION & DECISION	4	• BACKPROPAGATION	119																																																																																																																																				
• MACHINE LEARNING	6	• ADJUST	129																																																																																																																																				
• DEEP LEARNING	12	• PERFORMANCE	135																																																																																																																																				
• ALGORITHM	13	• LINEAR ACTIVATION	137																																																																																																																																				
• DATA	14	• LINEARITY	139																																																																																																																																				
• COMPUTATION	15	• NON-LINEARITY	141																																																																																																																																				
• ROADMAP	19	• RELU ACTIVATION	142																																																																																																																																				
• KEY CONCEPTS	20	• PERFORMANCE	145																																																																																																																																				
• ACTIVATION FUNCTIONS		• ACTIVATION FUNCTIONS	148																																																																																																																																				
<b>FOUNDATIONS</b>																																																																																																																																							
• A NEURON	22	<b>3 - BINARY CLASSIFICATION</b>																																																																																																																																					
• WEIGHTED SUM	24	• WEIGHTS AND BIASES	25	• INTRODUCTION	150	• ACTIVATION	27	• CLASSIFICATION VS. REGRESSION	152	• DATA	30	• GOAL AND DATASET	153	• DATASET	34	• ARCHITECTURE	156	• TRAINING	38	• SIGMOID ACTIVATION	158	• TESTING	40	• BINARY CROSS ENTROPY	166			• ACCURACY	169			• PERFORMANCE	170			• CONFUSION MATRIX	174			• PRECISION-RECALL	176			• F1 SCORE	177	<b>1 - LINEAR REGRESSION</b>		<b>4 - MULTI-CLASS CLASSIFICATION</b>		• INTRODUCTION	43	• GOAL AND DATASET	45	• INTRODUCTION	180	• PREDICT-MEASURE-FEEDBACK-ADJUST	50	• GOAL AND DATASET	181	• WEIGHTED SUM AND ACTIVATION	52	• ONE-HOT ENCODING	183	• LOSS FUNCTION	58	• ARCHITECTURE	184	• MEAN SQUARED ERROR	59	• SOFTMAX ACTIVATION	188	• MINIMIZING LOSS	64	• CATEGORICAL CROSS ENTROPY	195	• GRADIENT	68	• PERFORMANCE	197	• GRADIENT DESCENT	73	• IMPROVING PERFORMANCE		• LEARNING RATE	76	• HYPERPARAMETERS	202	• EPOCH	81	• DATA TECHNIQUES	210	• COST AND METRIC	83			• PERFORMANCE	86	<b>THE BIGGER PICTURE</b>				• NEURAL NETWORKS		<b>2 - NON-LINEAR REGRESSION</b>		• FEEDFORWARD	217	• INTRODUCTION	93	• CONVOLUTIONAL	219	• GOAL AND DATASET	99	• RECURRENT	225	• ARCHITECTURE	102	• GENERATIVE ADVERSARIAL	230	• PREDICT	108	• OTHER ARCHITECTURES	232	• MEASURE	112	• CONCLUSION	233	• FEEDBACK	114	• KEY CONCEPTS REVISITED	234	• COMPUTATION GRAPH	117	• SUGGESTED RESOURCES	235
• WEIGHTS AND BIASES	25	• INTRODUCTION	150																																																																																																																																				
• ACTIVATION	27	• CLASSIFICATION VS. REGRESSION	152																																																																																																																																				
• DATA	30	• GOAL AND DATASET	153																																																																																																																																				
• DATASET	34	• ARCHITECTURE	156																																																																																																																																				
• TRAINING	38	• SIGMOID ACTIVATION	158																																																																																																																																				
• TESTING	40	• BINARY CROSS ENTROPY	166																																																																																																																																				
		• ACCURACY	169																																																																																																																																				
		• PERFORMANCE	170																																																																																																																																				
		• CONFUSION MATRIX	174																																																																																																																																				
		• PRECISION-RECALL	176																																																																																																																																				
		• F1 SCORE	177																																																																																																																																				
<b>1 - LINEAR REGRESSION</b>		<b>4 - MULTI-CLASS CLASSIFICATION</b>																																																																																																																																					
• INTRODUCTION	43	• GOAL AND DATASET	45	• INTRODUCTION	180	• PREDICT-MEASURE-FEEDBACK-ADJUST	50	• GOAL AND DATASET	181	• WEIGHTED SUM AND ACTIVATION	52	• ONE-HOT ENCODING	183	• LOSS FUNCTION	58	• ARCHITECTURE	184	• MEAN SQUARED ERROR	59	• SOFTMAX ACTIVATION	188	• MINIMIZING LOSS	64	• CATEGORICAL CROSS ENTROPY	195	• GRADIENT	68	• PERFORMANCE	197	• GRADIENT DESCENT	73	• IMPROVING PERFORMANCE		• LEARNING RATE	76	• HYPERPARAMETERS	202	• EPOCH	81	• DATA TECHNIQUES	210	• COST AND METRIC	83			• PERFORMANCE	86	<b>THE BIGGER PICTURE</b>				• NEURAL NETWORKS		<b>2 - NON-LINEAR REGRESSION</b>		• FEEDFORWARD	217	• INTRODUCTION	93	• CONVOLUTIONAL	219	• GOAL AND DATASET	99	• RECURRENT	225	• ARCHITECTURE	102	• GENERATIVE ADVERSARIAL	230	• PREDICT	108	• OTHER ARCHITECTURES	232	• MEASURE	112	• CONCLUSION	233	• FEEDBACK	114	• KEY CONCEPTS REVISITED	234	• COMPUTATION GRAPH	117	• SUGGESTED RESOURCES	235																																																		
• GOAL AND DATASET	45	• INTRODUCTION	180																																																																																																																																				
• PREDICT-MEASURE-FEEDBACK-ADJUST	50	• GOAL AND DATASET	181																																																																																																																																				
• WEIGHTED SUM AND ACTIVATION	52	• ONE-HOT ENCODING	183																																																																																																																																				
• LOSS FUNCTION	58	• ARCHITECTURE	184																																																																																																																																				
• MEAN SQUARED ERROR	59	• SOFTMAX ACTIVATION	188																																																																																																																																				
• MINIMIZING LOSS	64	• CATEGORICAL CROSS ENTROPY	195																																																																																																																																				
• GRADIENT	68	• PERFORMANCE	197																																																																																																																																				
• GRADIENT DESCENT	73	• IMPROVING PERFORMANCE																																																																																																																																					
• LEARNING RATE	76	• HYPERPARAMETERS	202																																																																																																																																				
• EPOCH	81	• DATA TECHNIQUES	210																																																																																																																																				
• COST AND METRIC	83																																																																																																																																						
• PERFORMANCE	86	<b>THE BIGGER PICTURE</b>																																																																																																																																					
		• NEURAL NETWORKS																																																																																																																																					
<b>2 - NON-LINEAR REGRESSION</b>		• FEEDFORWARD	217																																																																																																																																				
• INTRODUCTION	93	• CONVOLUTIONAL	219																																																																																																																																				
• GOAL AND DATASET	99	• RECURRENT	225																																																																																																																																				
• ARCHITECTURE	102	• GENERATIVE ADVERSARIAL	230																																																																																																																																				
• PREDICT	108	• OTHER ARCHITECTURES	232																																																																																																																																				
• MEASURE	112	• CONCLUSION	233																																																																																																																																				
• FEEDBACK	114	• KEY CONCEPTS REVISITED	234																																																																																																																																				
• COMPUTATION GRAPH	117	• SUGGESTED RESOURCES	235																																																																																																																																				



## PREDICTION AND DECISION

*Prediction is a key ingredient in decision-making under uncertainty.* — Prediction Machines book.

Much that goes on in our lives involves some form of prediction. These predictions differ in one way, namely, how sure we are of them. In some tasks, they don't feel like predictions because we feel so sure about them. In some others, we know next to nothing about them, so they become mere guesses.

All of this depends on how simple a task is and, more importantly, how much experience we have with it.

To illustrate this, let's look at some examples.

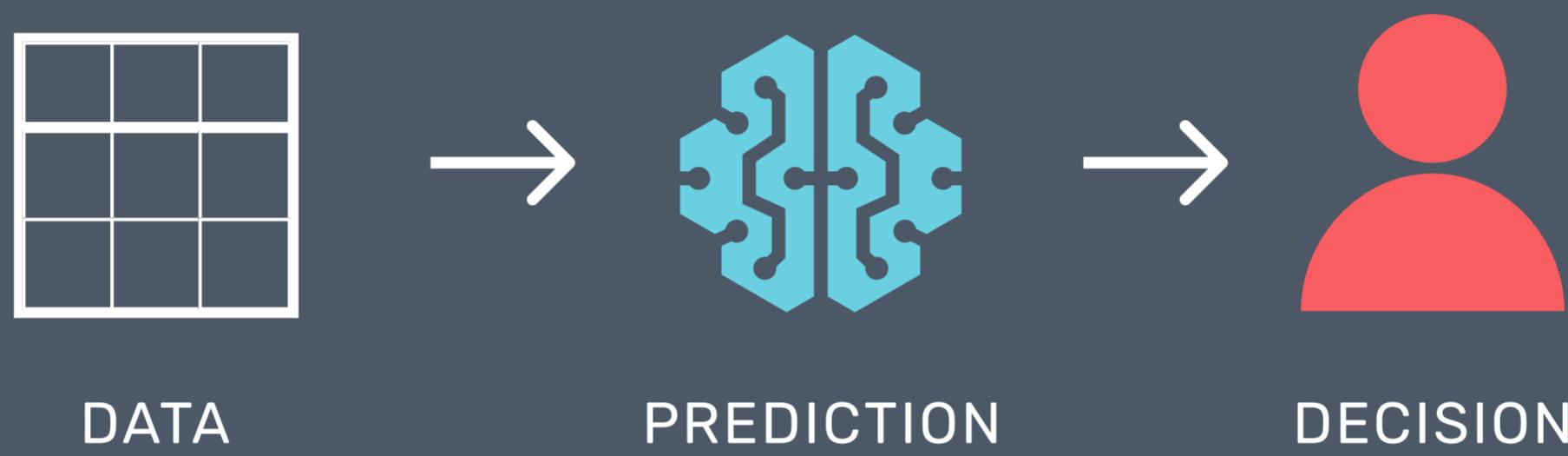


## EXAMPLES

Let's take the example of language translation. As we listen to someone speaking, we are predicting what the person means. The more experience we have with this language, the better our prediction becomes, and the better our decision, that is our reply, becomes.

Take another example in a business setting. Our experience dealing with customers can help us see patterns in their behavior, so we'll notice if they are likely to churn.

As for driving, the more miles we clock, the more skilled we become and the more adept we are at evaluating our surroundings.



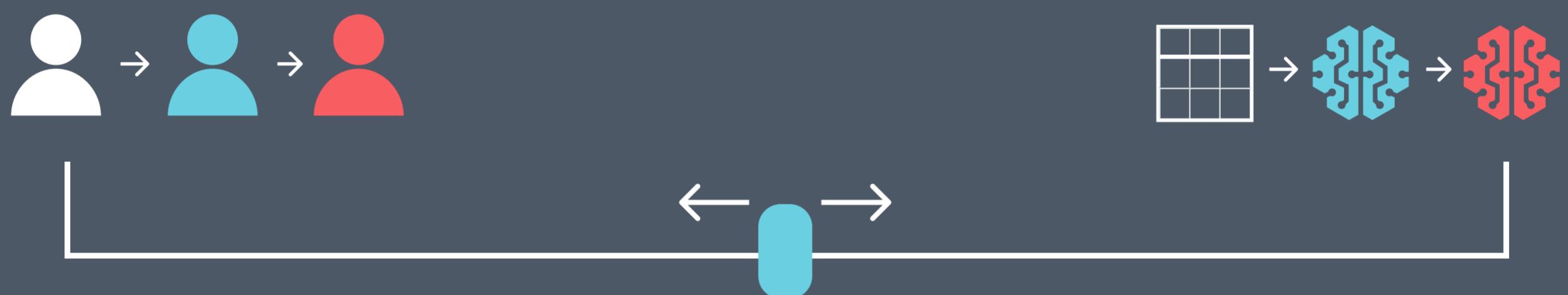
## WHAT IS MACHINE LEARNING?

In many of these tasks, machine learning can handle the prediction on our behalf.

In recent years, the adoption of machine learning has accelerated. Many industries and verticals are already deploying use cases that automate predictions using machine learning.

In the machine's world, the experience comes in the form of data. Just as we learn from experience, the machine learns from data.

That is what machine learning is all about—learning from the data and turning it into predictions.

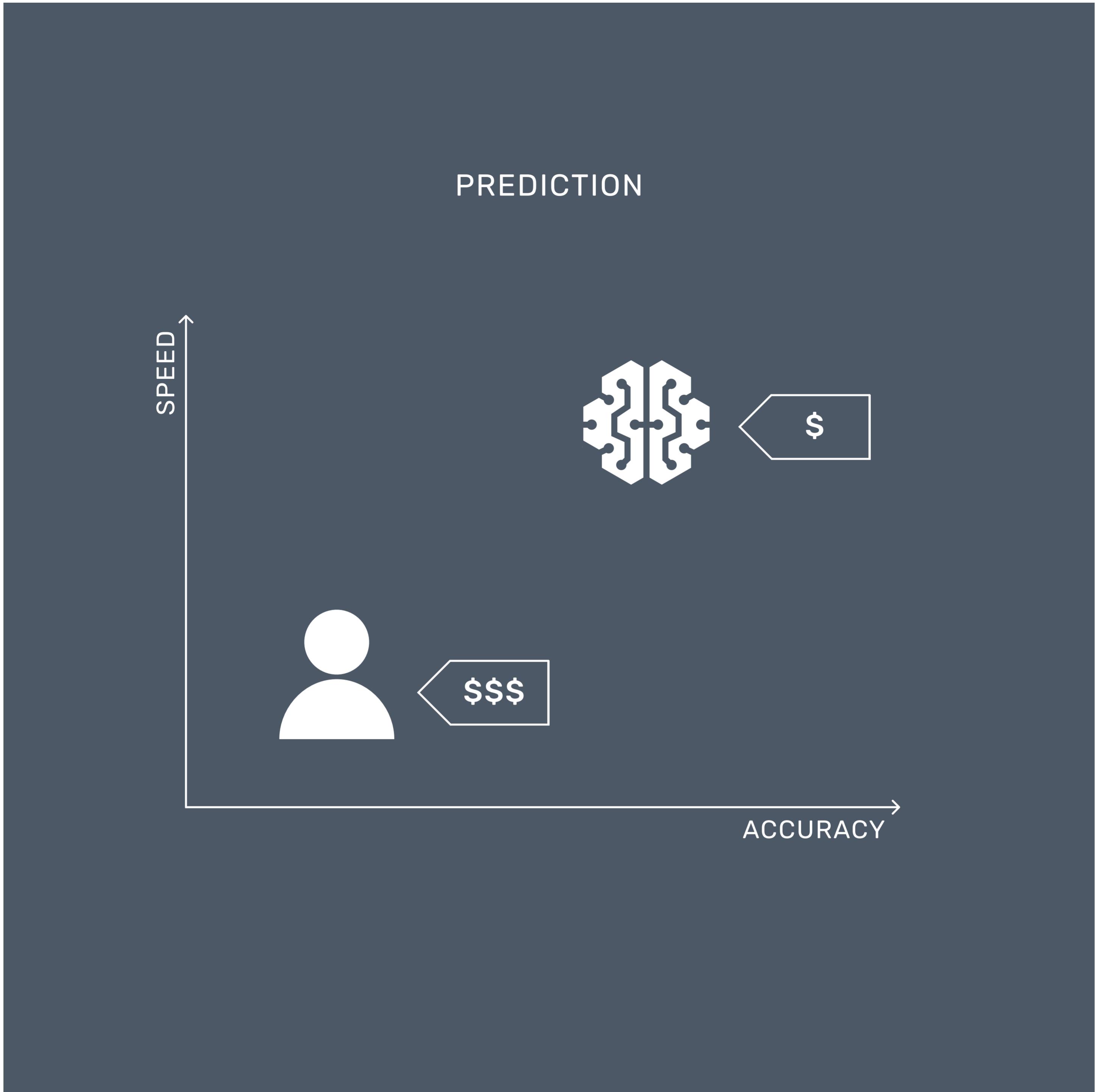


## MACHINE LEARNING IN THE REAL WORLD

In fact, machine learning can even handle the decision part. In some domains, most notably self-driving cars, we are not far from seeing full automation becoming the norm.

But in most other domains, this is still far from reality. For this reason, the focus of this book is on the prediction part.

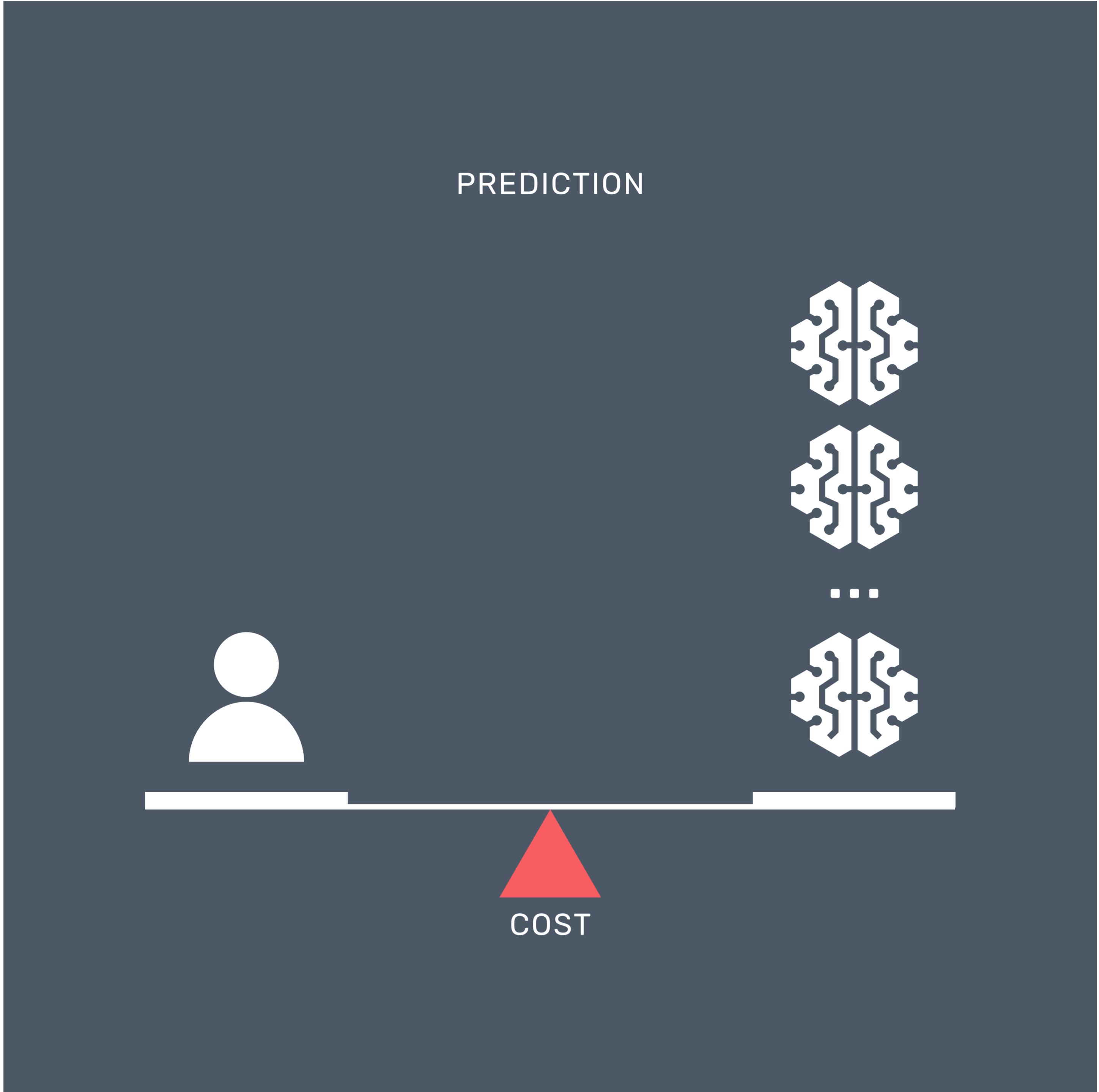
And indeed, it is upon us to ensure healthy technological progress, where people can thrive with the help of machines rather than being inhibited by them. That's the sweet spot that we are collectively trying to find.



## THE VALUE OF MACHINE LEARNING

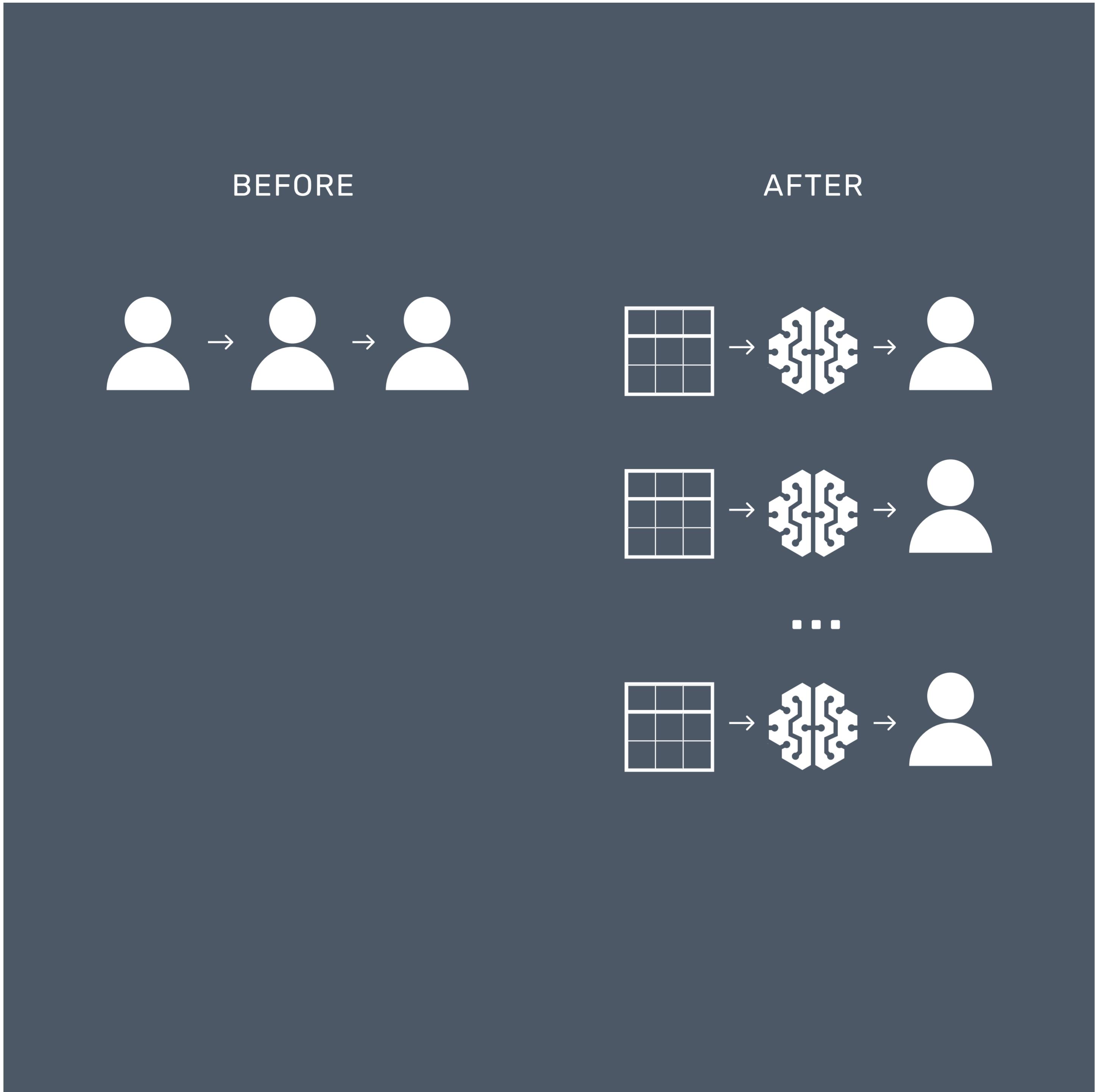
So, what is the value of having machines that can make predictions on our behalf?

In the book *Prediction Machines*, the authors argued for a few reasons why prediction machines are so valuable, the first being that 'they can often produce better, faster, and cheaper predictions than humans can'.



## ACCELERATING HUMAN PROGRESS

The cheaper the cost of prediction, the more tasks we can take on. The world is full of challenges waiting to be solved. Machine learning enables us to scale our efforts in ways that have not been possible before, presenting us with the opportunity to take on these challenges.

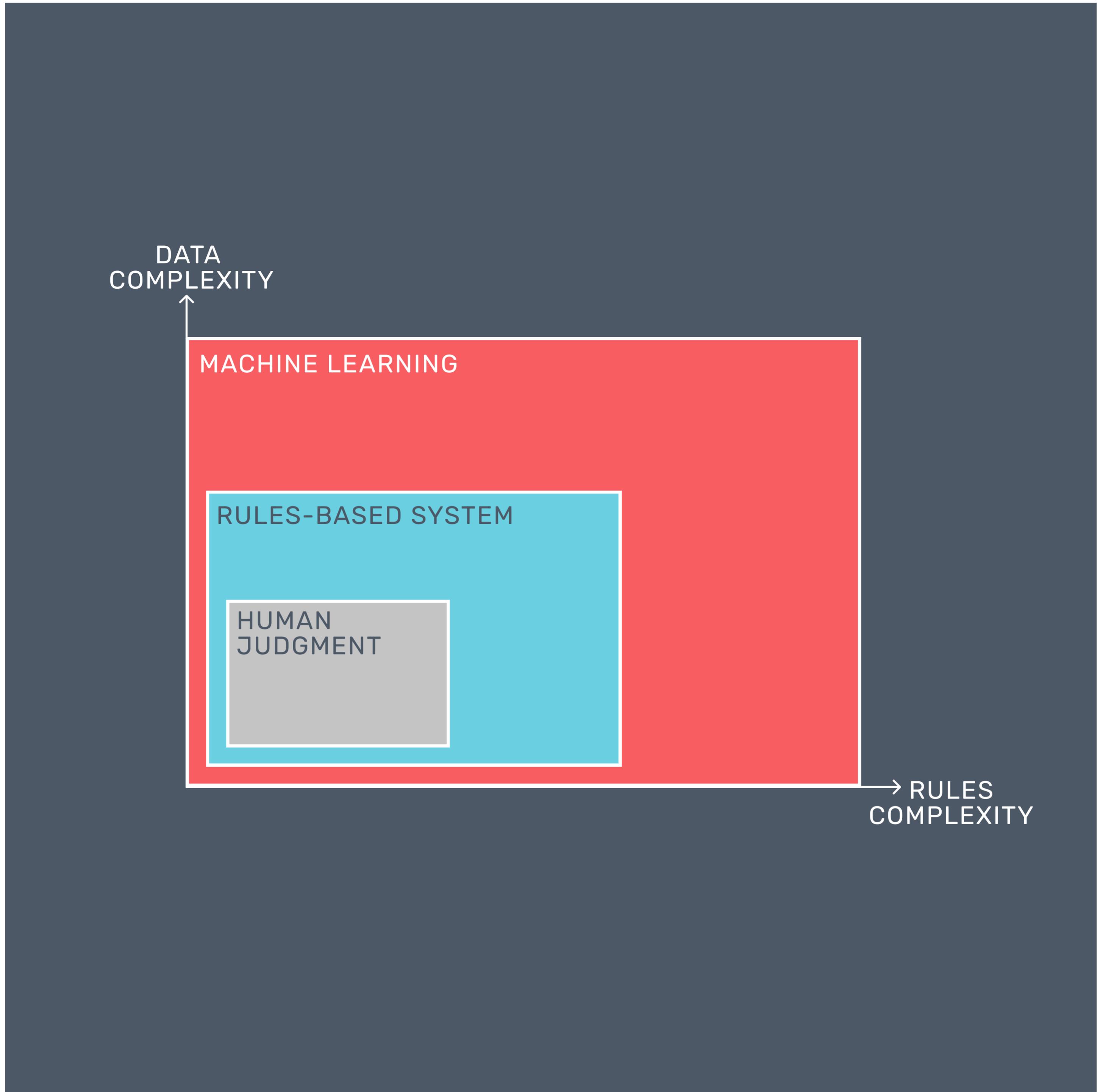


## EVOLUTION IN ROLES

Some may worry that this will spell the end of most jobs, and rightly so. But looking at the bigger picture, there will in fact be even more job opportunities.

The World Economic Forum's The Future of Jobs Report 2020 estimates that by 2025, 85 million jobs may be displaced. But on the other hand, 97 million new roles may emerge. This already takes into account the economic slowdown due to the pandemic, and still, the net effect is positive.

Job roles will evolve, and the machine's role is to serve us so we can pursue more creative and challenging endeavors.



## WHEN TO USE MACHINE LEARNING?

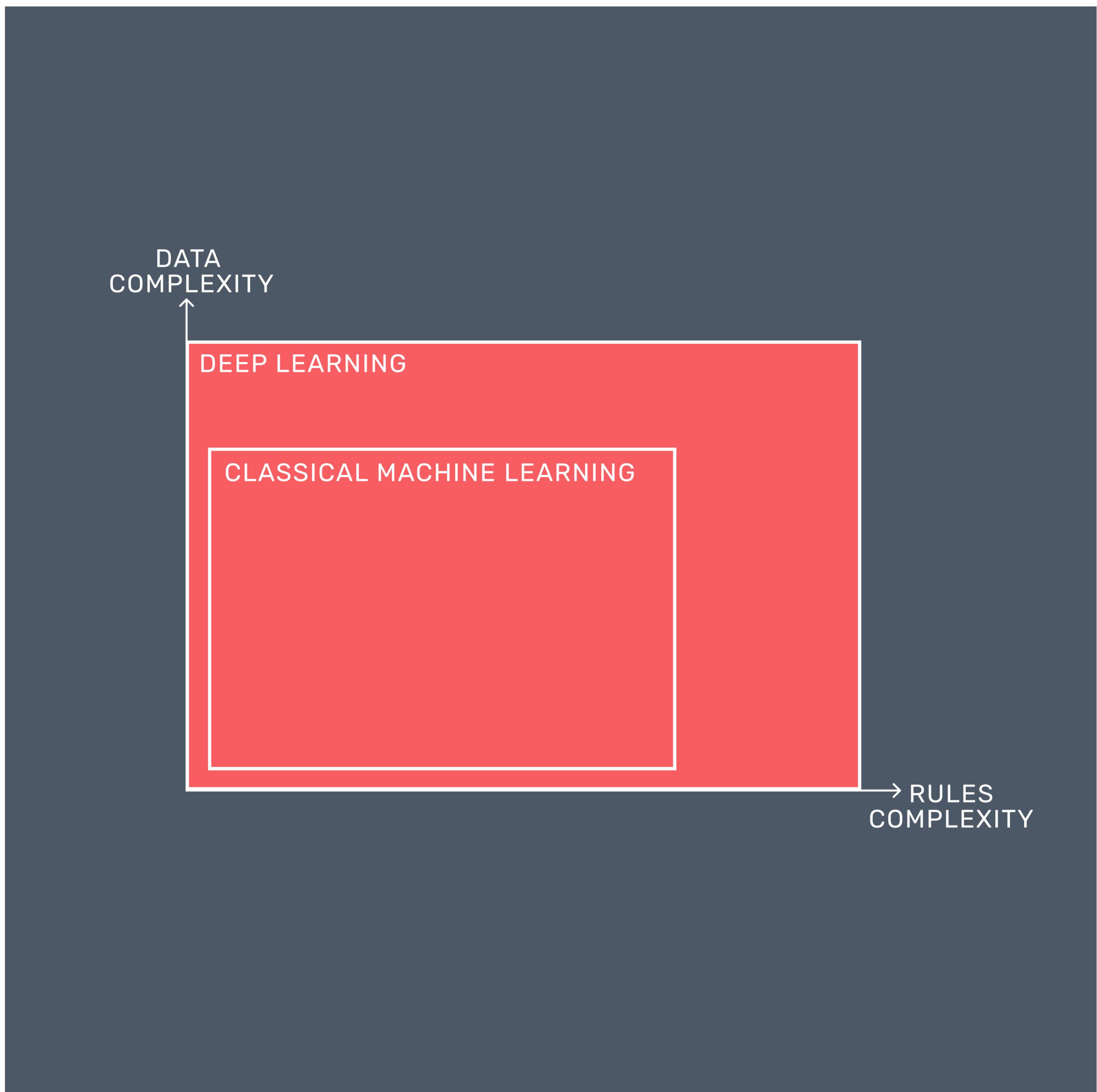
We can think of prediction automation in three phases.

The first, that is without automation, is relying on human judgment, either based on data or experience.

The second is using a rules-based system. We translate our experience into rules that software can understand and execute based on data as inputs.

The third is machine learning, which uses data to create its own rules, guided by the goal defined by humans.

As the data and rules become more complex, it makes sense to use machine learning. Otherwise, it may not be cost-effective to do so.



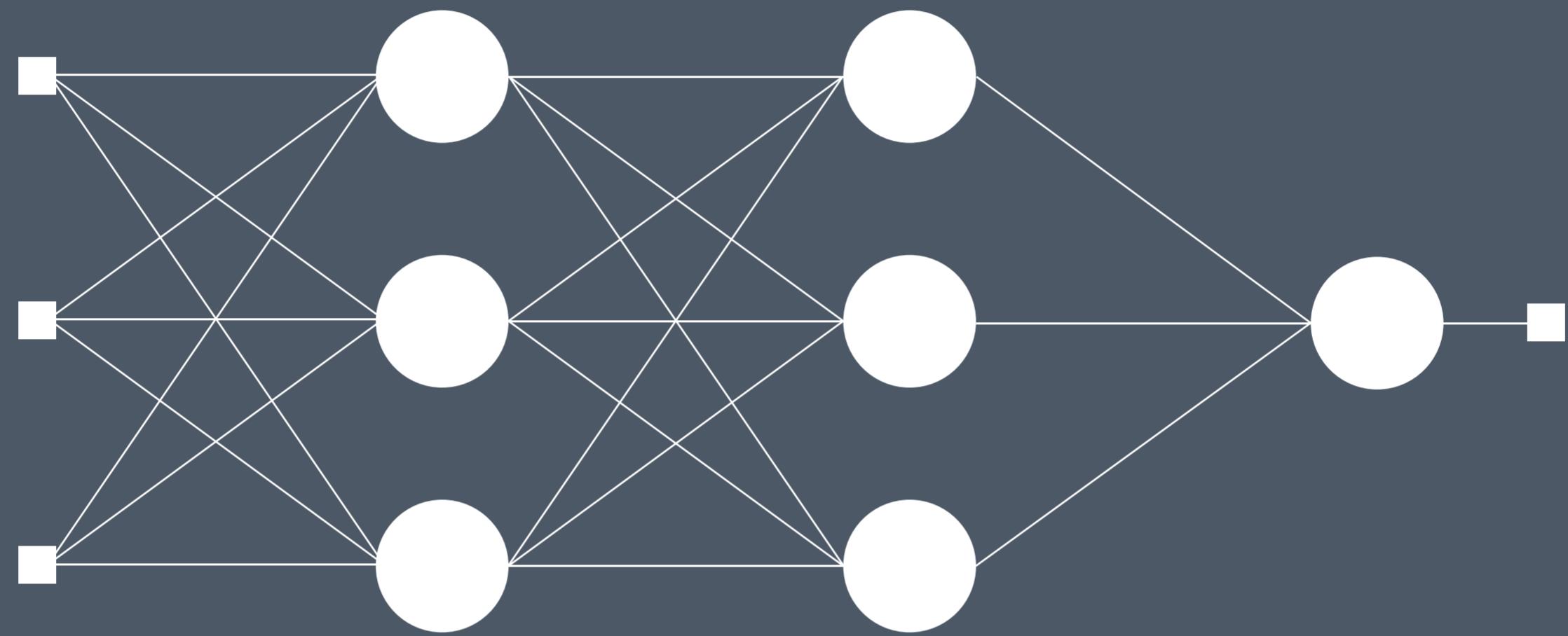
## WHAT IS DEEP LEARNING?

Within machine learning, there are various types of algorithms. Think of machine learning algorithms as competing techniques to get the best out of the data. Some algorithms are better in certain aspects, but there's not one that's the best in all departments.

Deep learning is a type of algorithm that's adaptable to varying complexities of data and rules.

It's not necessarily the most accurate, but it's extremely adaptable. And this comes from its modular and flexible form, which will become evident throughout this book.

## ALGORITHM



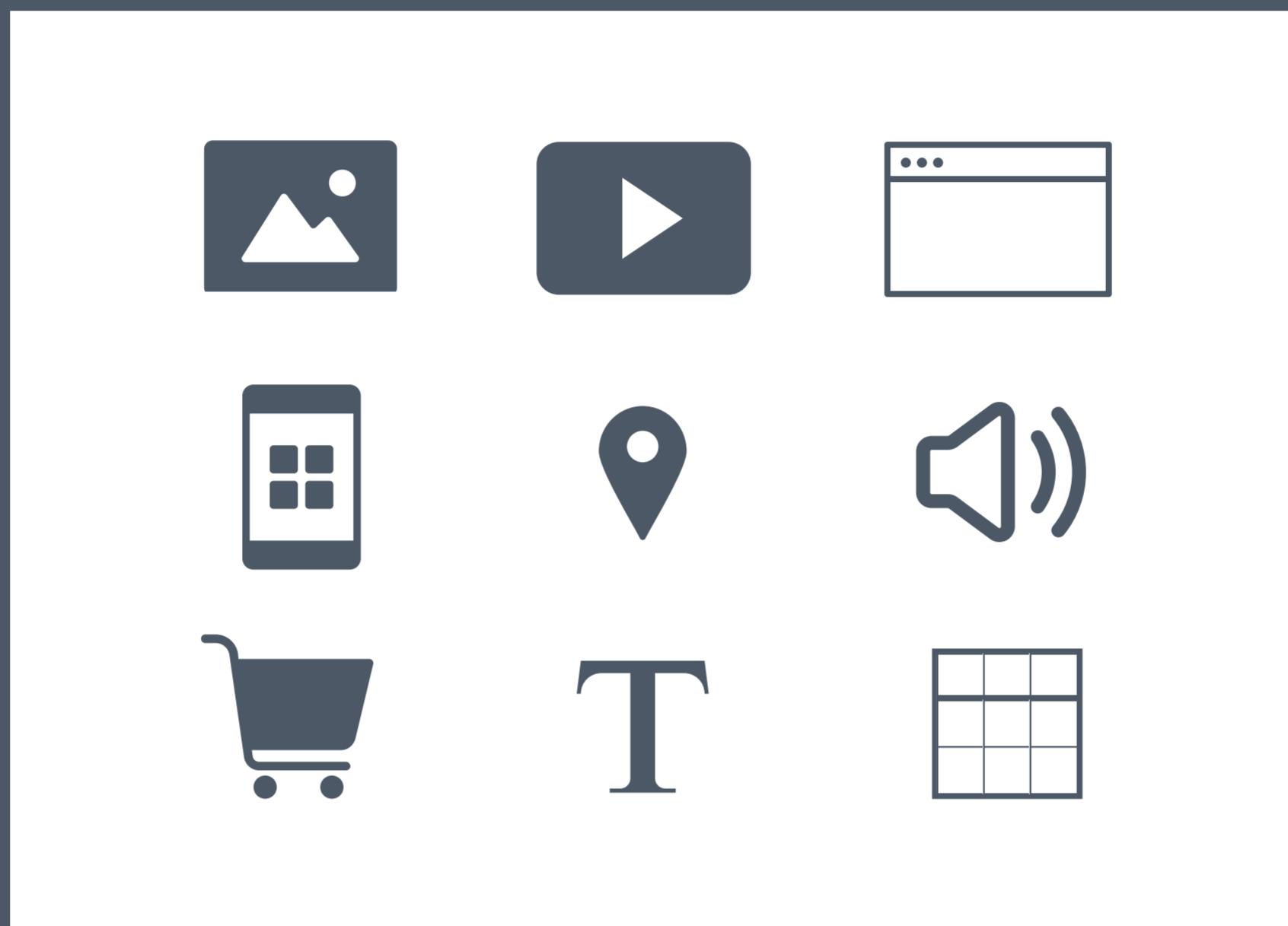
## ALGORITHM

In fact, deep learning has revived the push toward Artificial Intelligence (AI) over the past decade.

The progress is gathering pace now is because of three main reasons. The first is the algorithm, which in truth, has been around for many decades.

But that alone is not enough.

## DATA

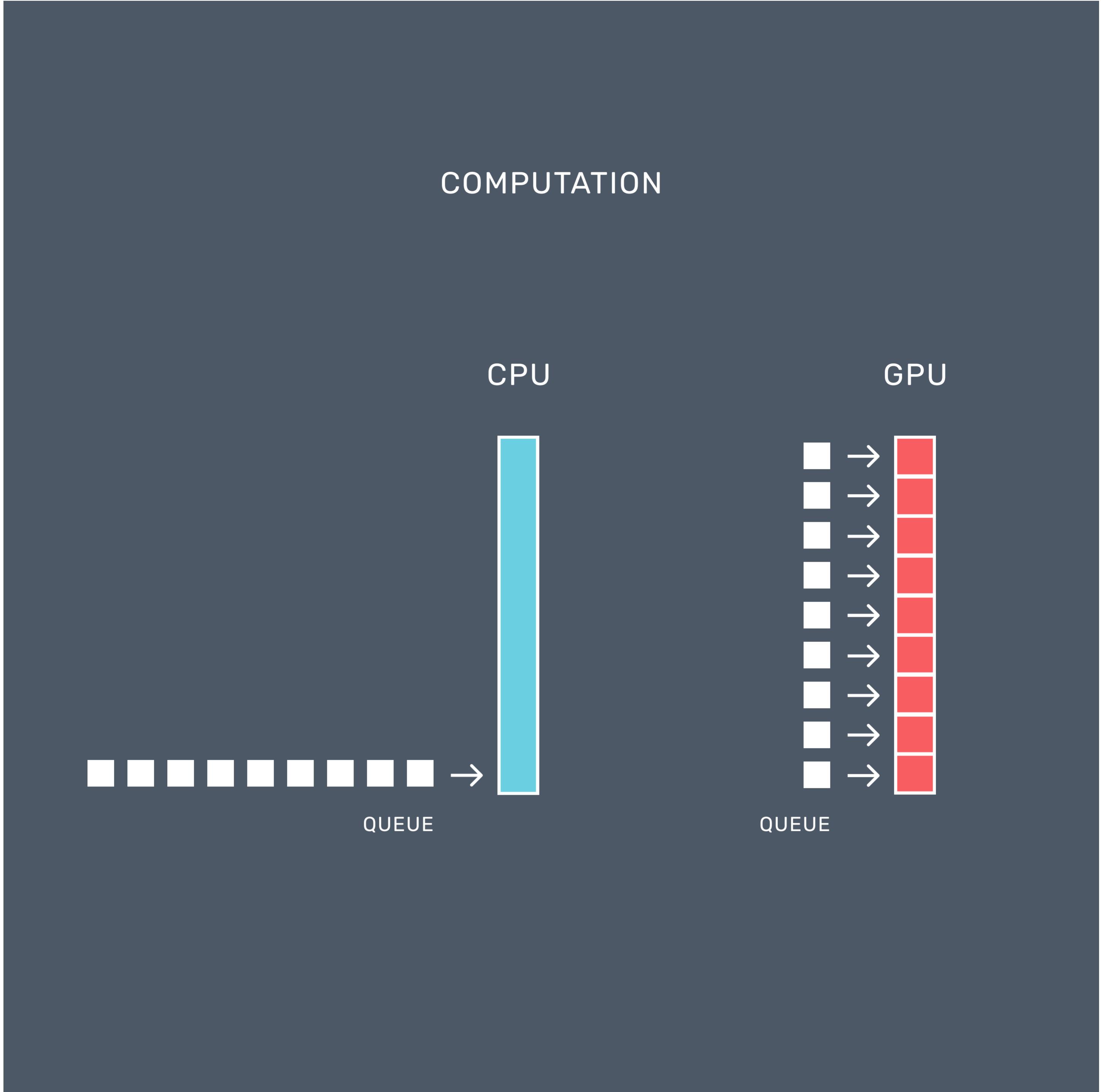


## DATA

The second reason is data.

The impetus came from the Internet and followed by social media, smartphones, digital transformation, and a long list of other waves of innovation. They produce new forms of data that we've never seen before, generated in large volumes.

This data contains invaluable information that we can now extract with the help of algorithms.



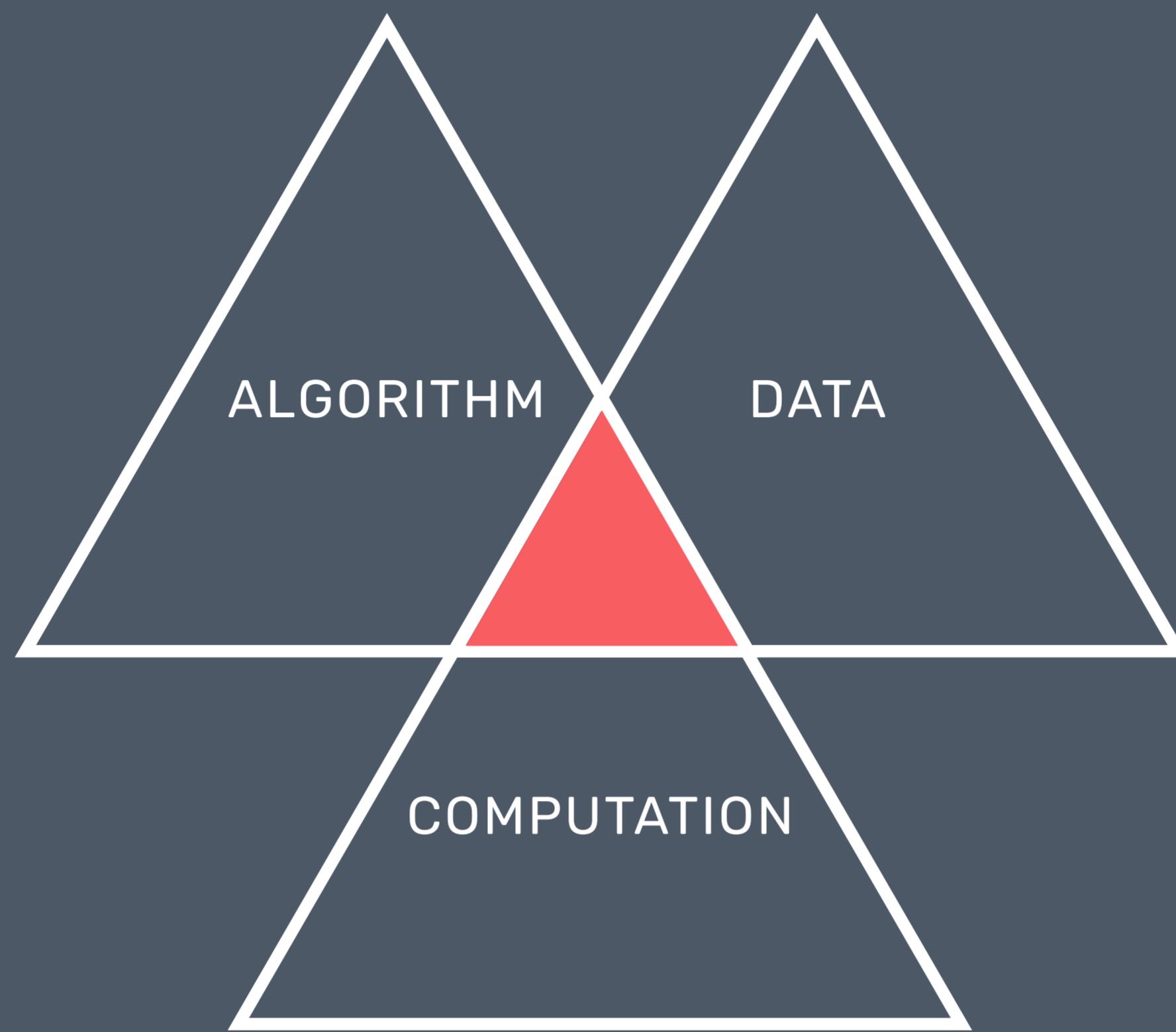
## COMPUTATION

The third reason is computational power.

Machine learning involves performing a significant amount of mathematical computation on the data. In deep learning, this is multiplied many times over. The standard Central Processing Unit (CPU) architecture is not capable of handling this task efficiently.

Enter the Graphics Processing Units (GPU). Originally designed for games, it has emerged as the perfect solution for deep learning.

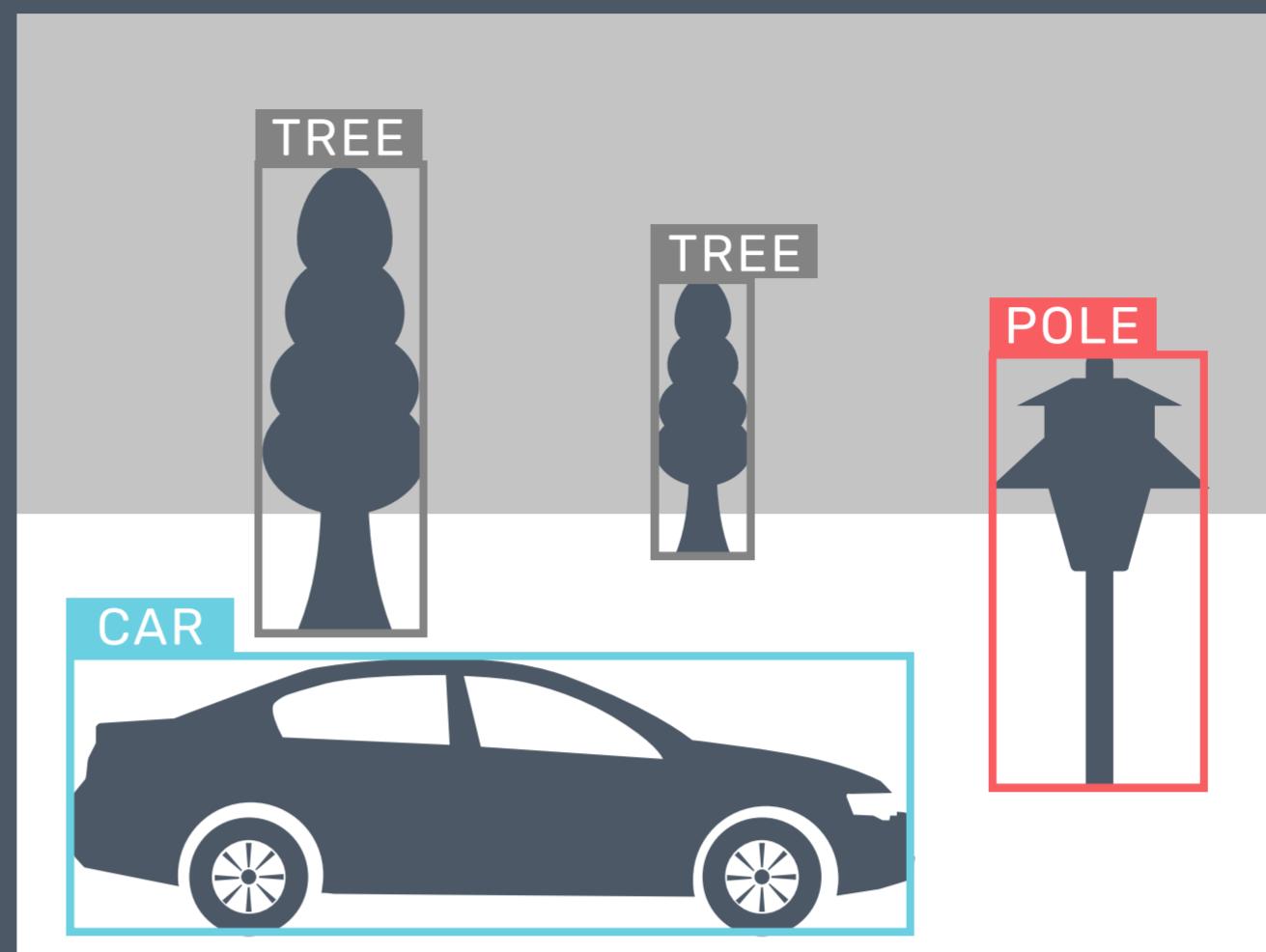
This is a hot area of research as we speak. Even more efficient hardware designs are yet to come.



## THE DRIVING FORCES

Together, these three factors are the driving forces behind today's rapid advances in deep learning.

## COMPUTER VISION



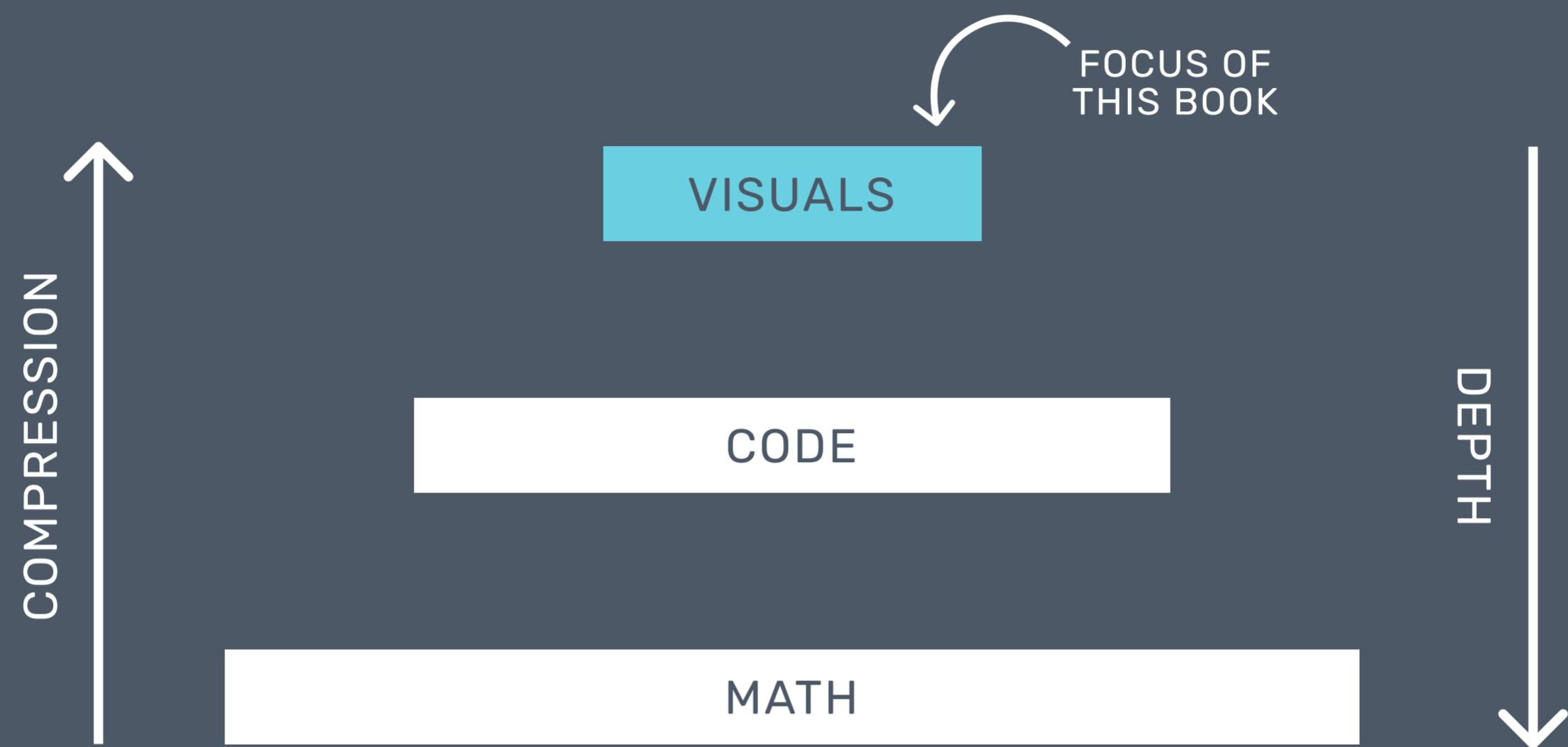
## NATURAL LANGUAGE PROCESSING

SENTENCE	SENTIMENT
IT'S A GREAT DAY	POSITIVE
I DON'T LIKE MONDAYS	NEGATIVE

• • •

## APPLICATIONS

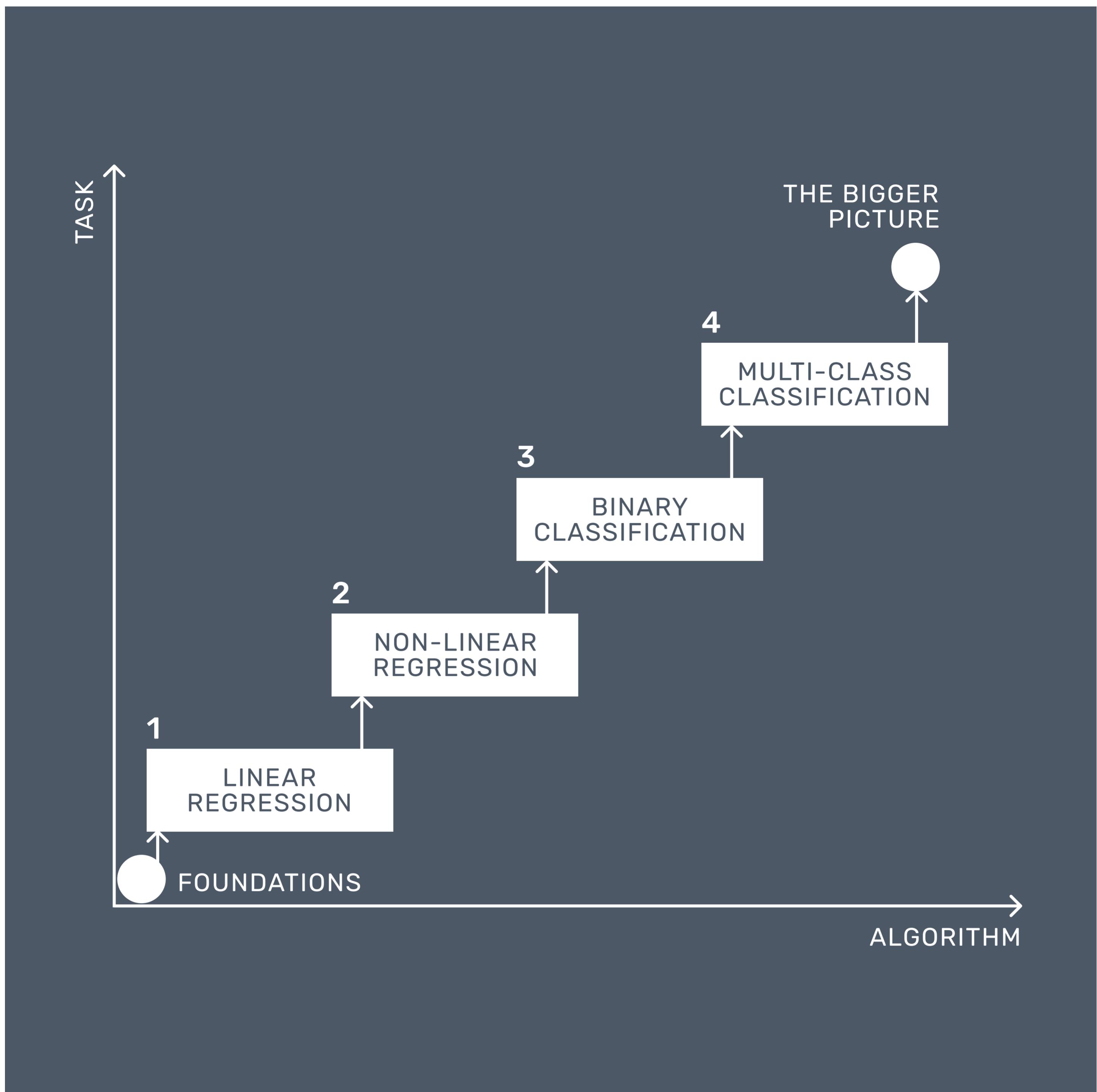
Today, there are widespread applications in computer vision, natural language processing, business automation, and beyond. And it is just the beginning.



## WHAT CAN YOU EXPECT FROM THIS BOOK?

By the end of this book, you will be able to build a visual intuition about deep learning and neural networks.

This book doesn't cover mathematical proofs and code examples. As you advance your learning further, these are the domains you should progress into. They will provide you with the depth you need to be successful in this field.

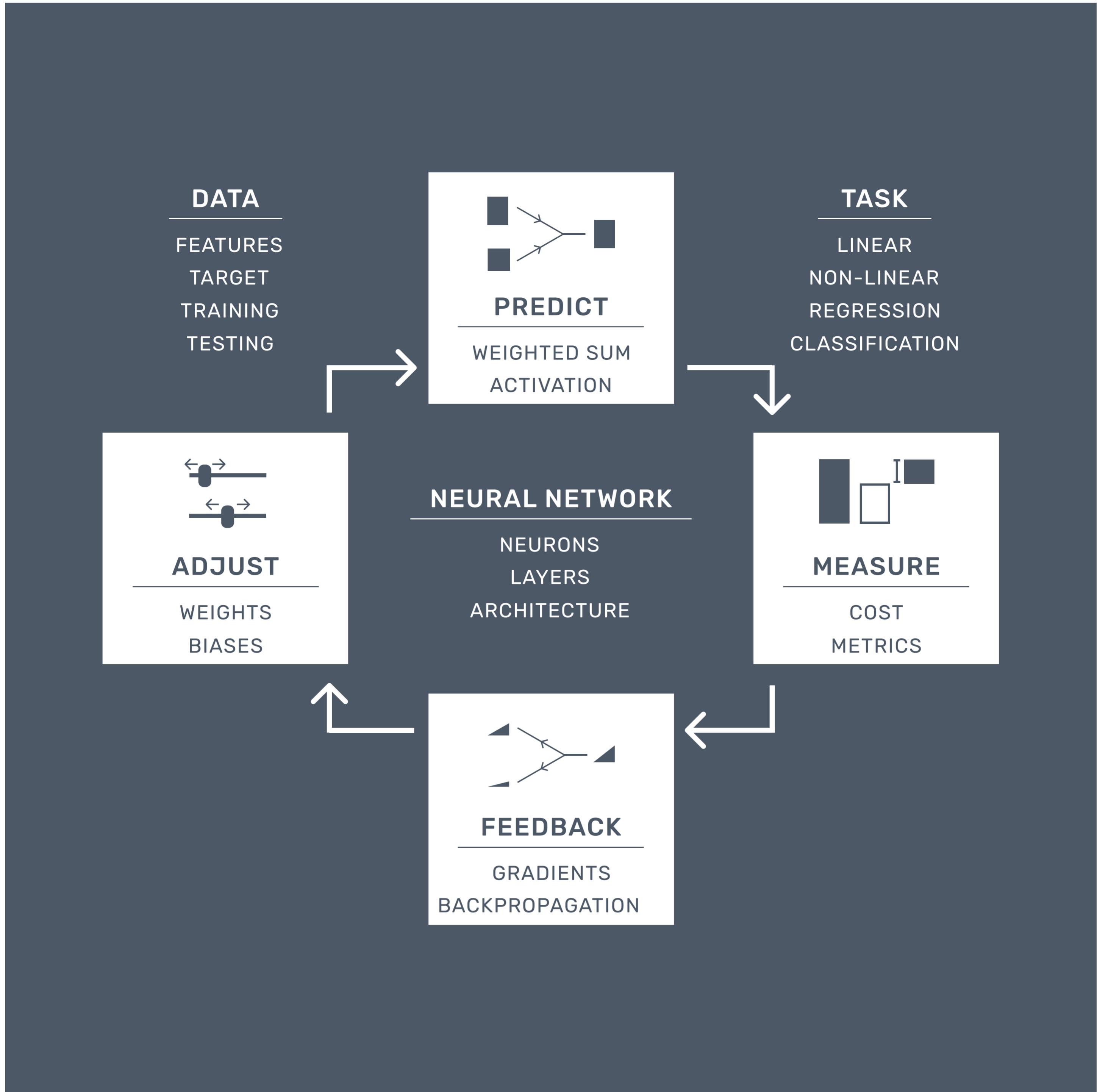


## ROADMAP

We'll see how deep learning works via four tasks - linear regression, non-linear regression, binary classification, multi-class classification.

They are correspondingly split into four chapters, in which new concepts are introduced one at a time and built upon the previous ones. Therefore, it is recommended that you read the chapters in sequence.

On either side of these four chapters, we'll have a short section for foundations and a final section where we take a brief look beyond those covered in the four chapters.

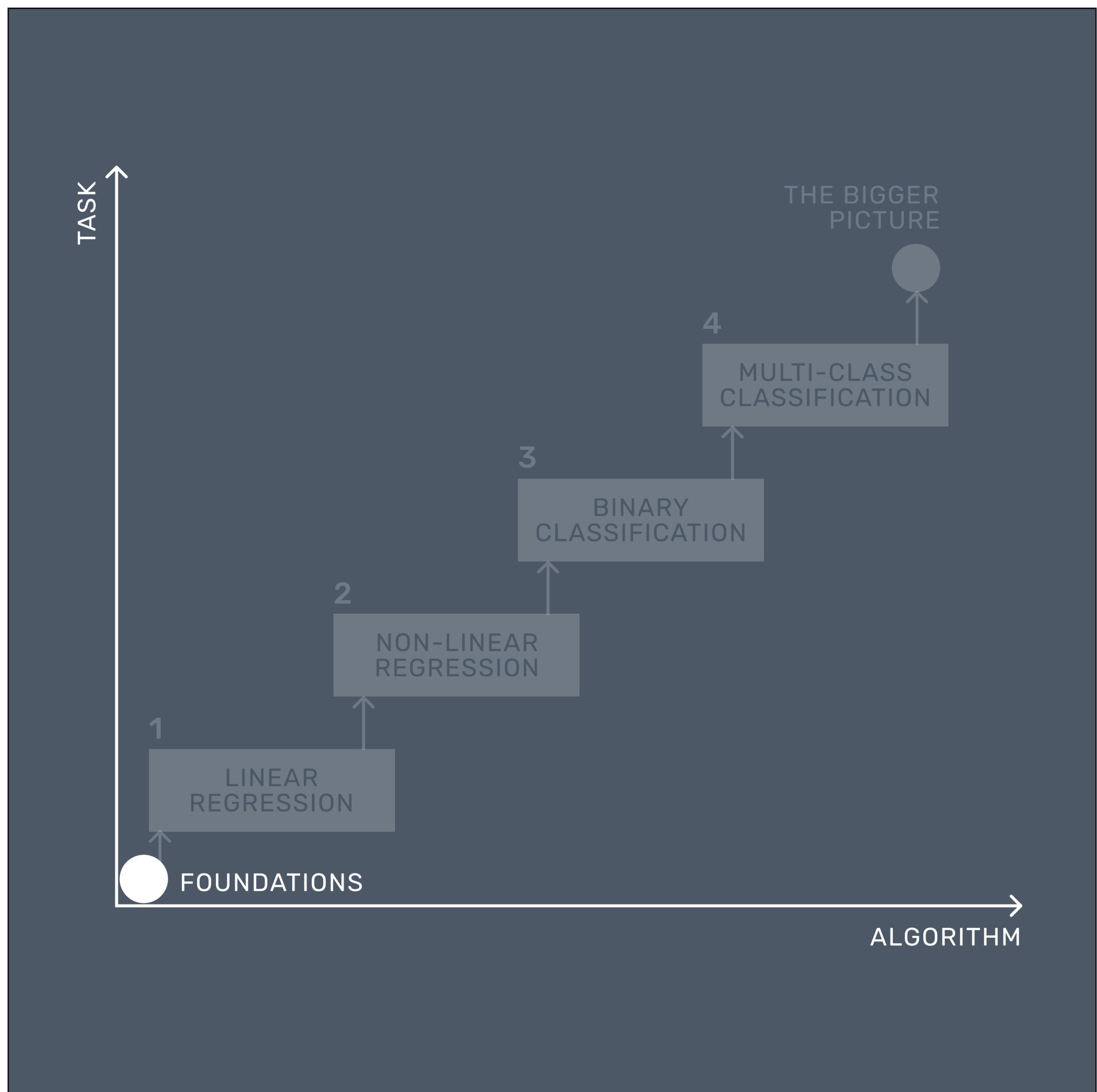


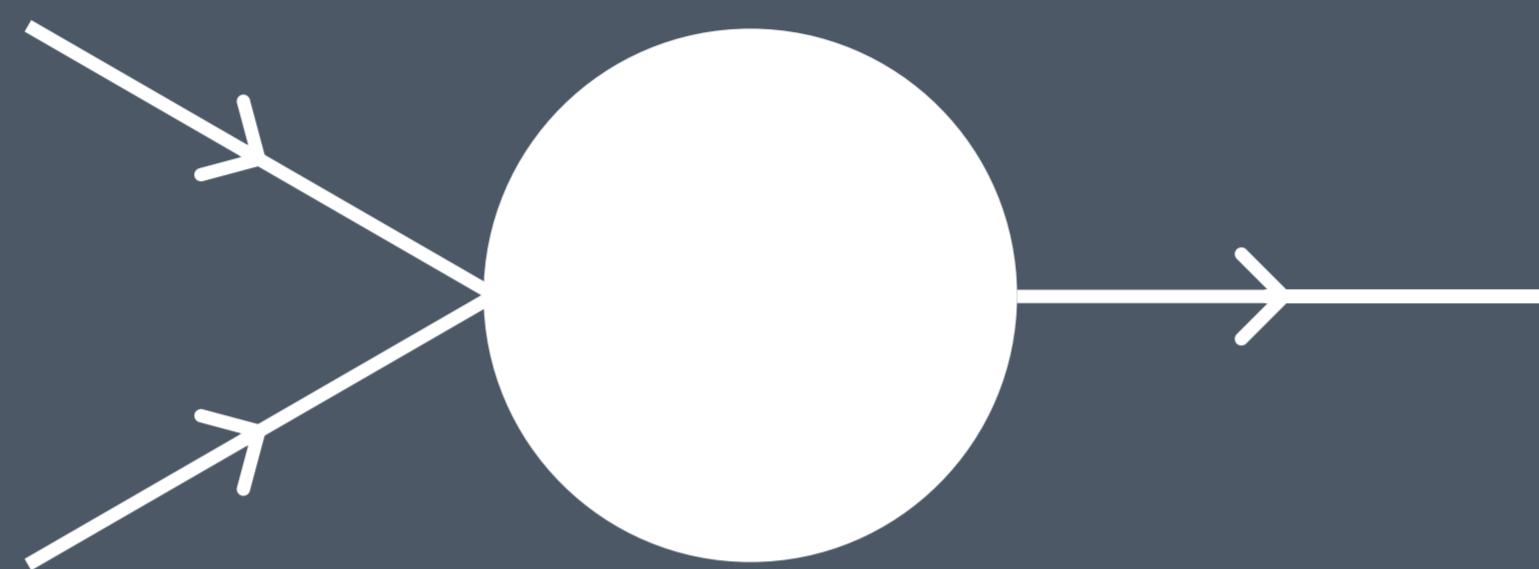
## KEY CONCEPTS

Here is a summary of the key concepts that we'll explore in this book. As you go through the book, it'll be useful to return to this page from time to time to keep track of what you have learned.

Let's begin!

# FOUNDATIONS



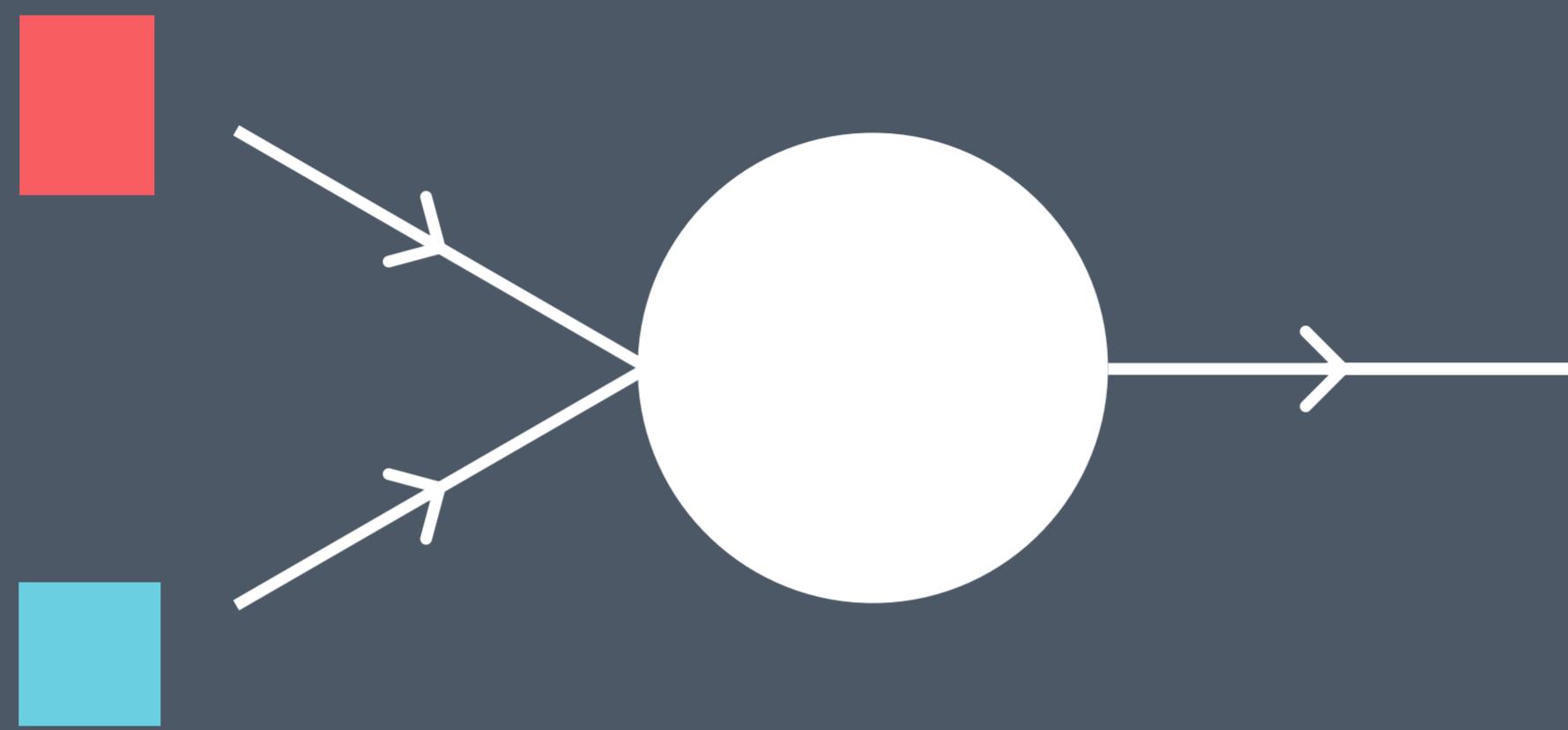


## A NEURON

We have so far used the term deep learning, but from now on, we'll use *neural network* instead. These terms are used interchangeably and refer to the same thing. But as we start to go into the inner workings, neural network is a more natural term to use.

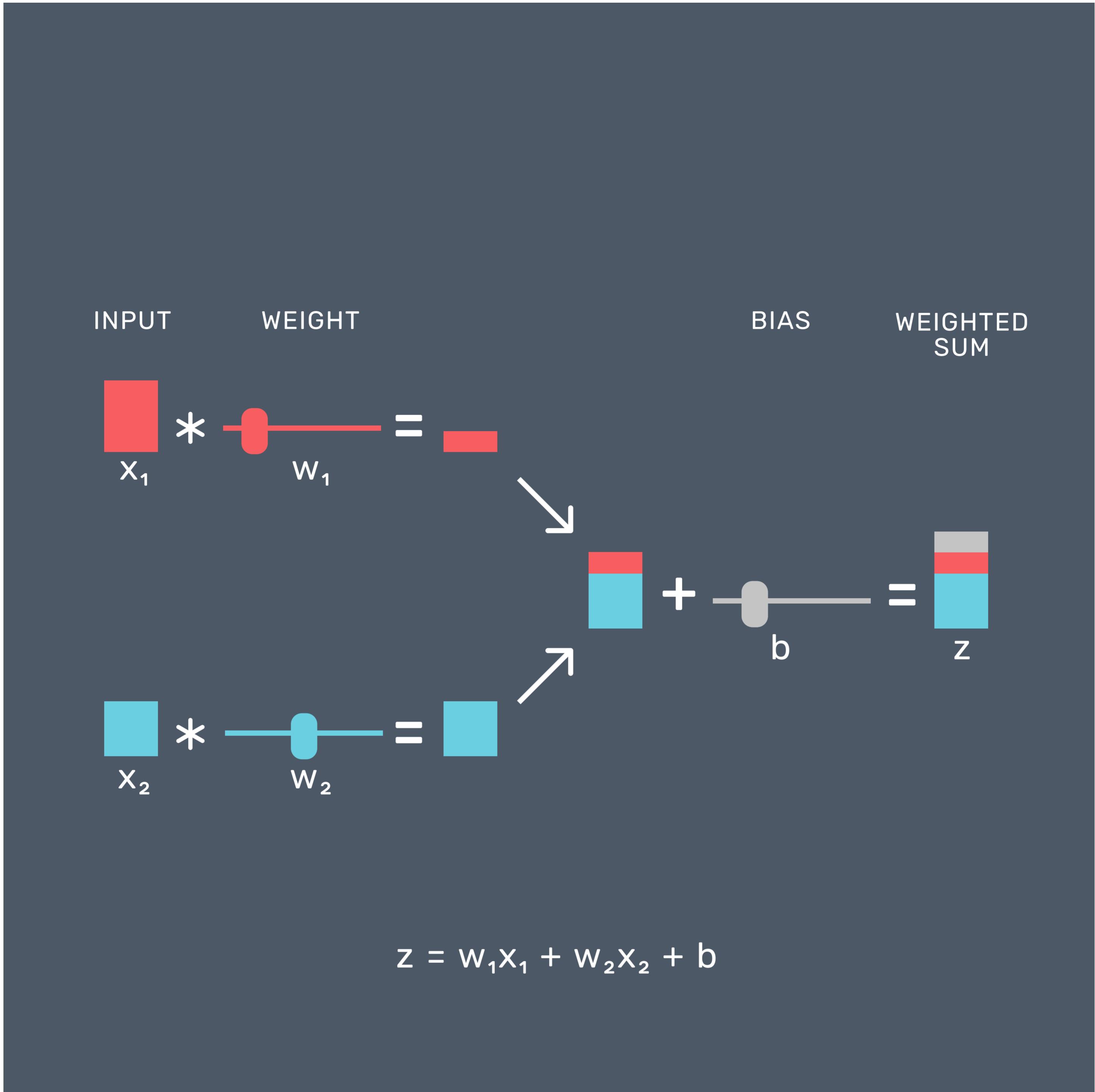
To begin our journey, let's start with a *neuron*. The neuron is the smallest unit and the building block of a neural network.

A neuron takes a set of inputs, performs some mathematical computations, and gives an output.



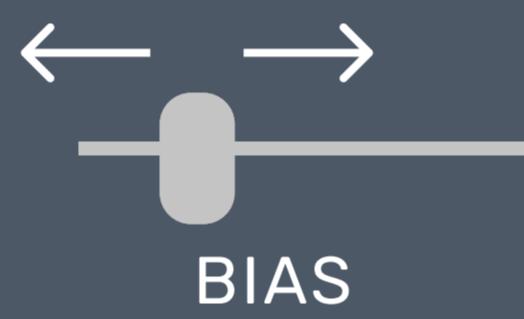
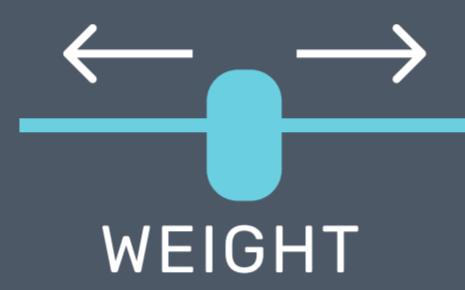
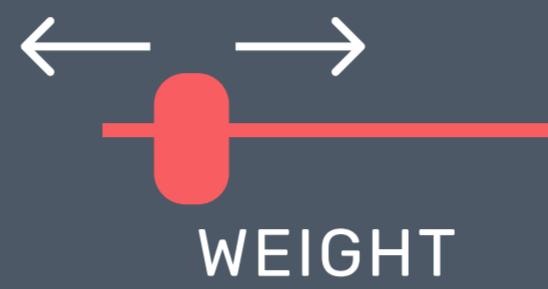
## INPUTS

The inputs and outputs are numbers, either positive or negative. In this example, the neuron takes two inputs. However, there is no limit to the number of inputs a neuron can take.



## WEIGHTED SUM

The first computation that a neuron performs is the *weighted sum*. It multiplies each input by its corresponding *weight*. Then all the inputs are summed and a term called *bias* is added.



## WEIGHTS AND BIASES

These weights and biases are called the *parameters* of the neural network.

These adjustable parameters are the medium through which a neural network learns, which we'll explore in detail in this book.

### EXAMPLE #1

$$\begin{array}{ccc} 3.0 & 0.5 & 1.5 \\ \text{---} * \text{---} = \text{---} \end{array}$$

$$\begin{array}{ccc} 2.0 & 1.5 & 3.0 \\ \text{---} * \text{---} = \text{---} \end{array}$$



$$\begin{array}{ccc} 4.5 & 1.0 & 5.5 \\ \text{---} + \text{---} = \text{---} \end{array}$$



### EXAMPLE #2

$$\begin{array}{ccc} 3.0 & -0.5 & -1.5 \\ \text{---} * \text{---} = \text{---} \end{array}$$

$$\begin{array}{ccc} 2.0 & 1.0 & 2.0 \\ \text{---} * \text{---} = \text{---} \end{array}$$

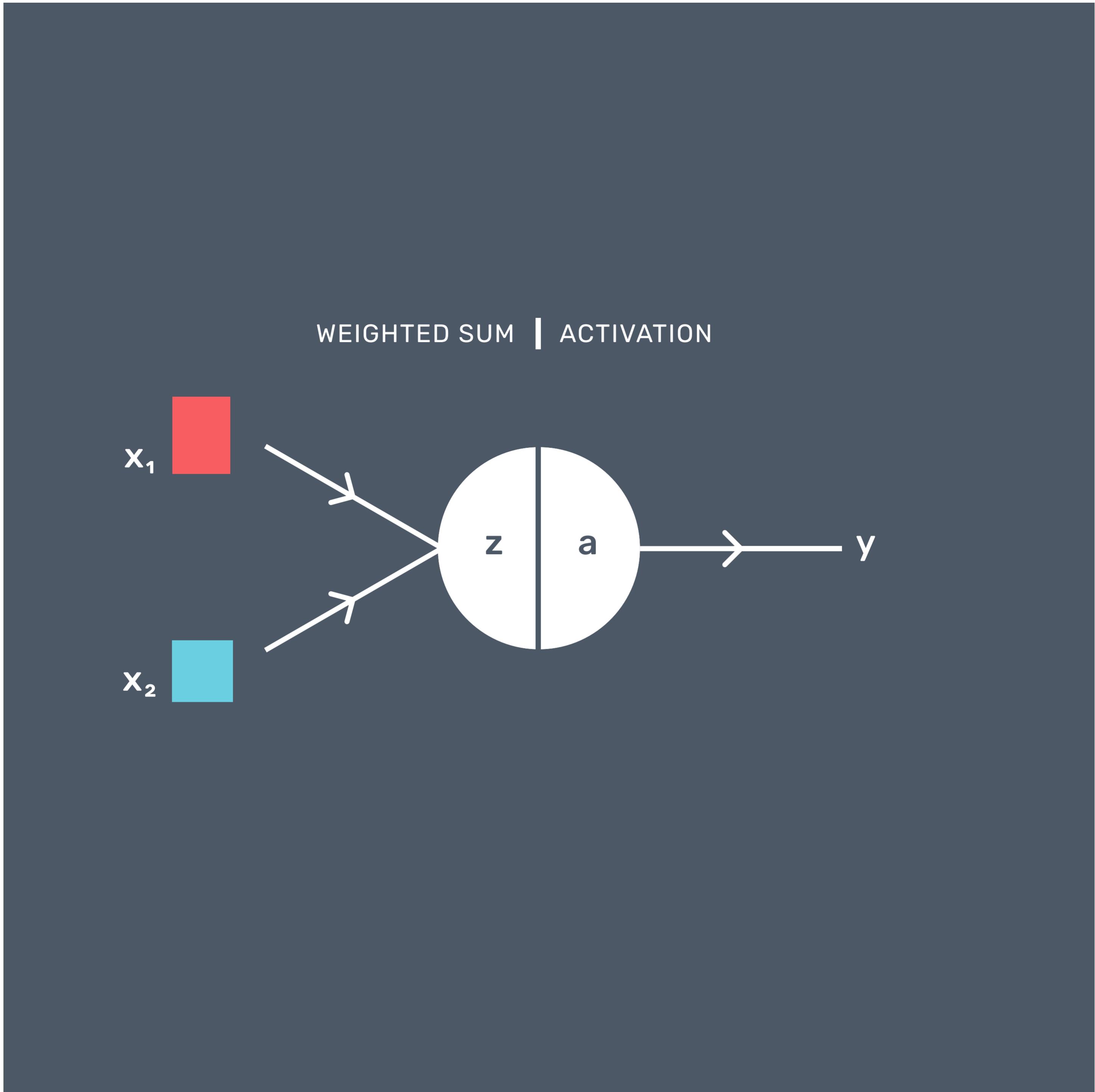


$$\begin{array}{ccc} 0.5 & -2.0 & -1.5 \\ \text{---} + \text{---} = \text{---} \end{array}$$



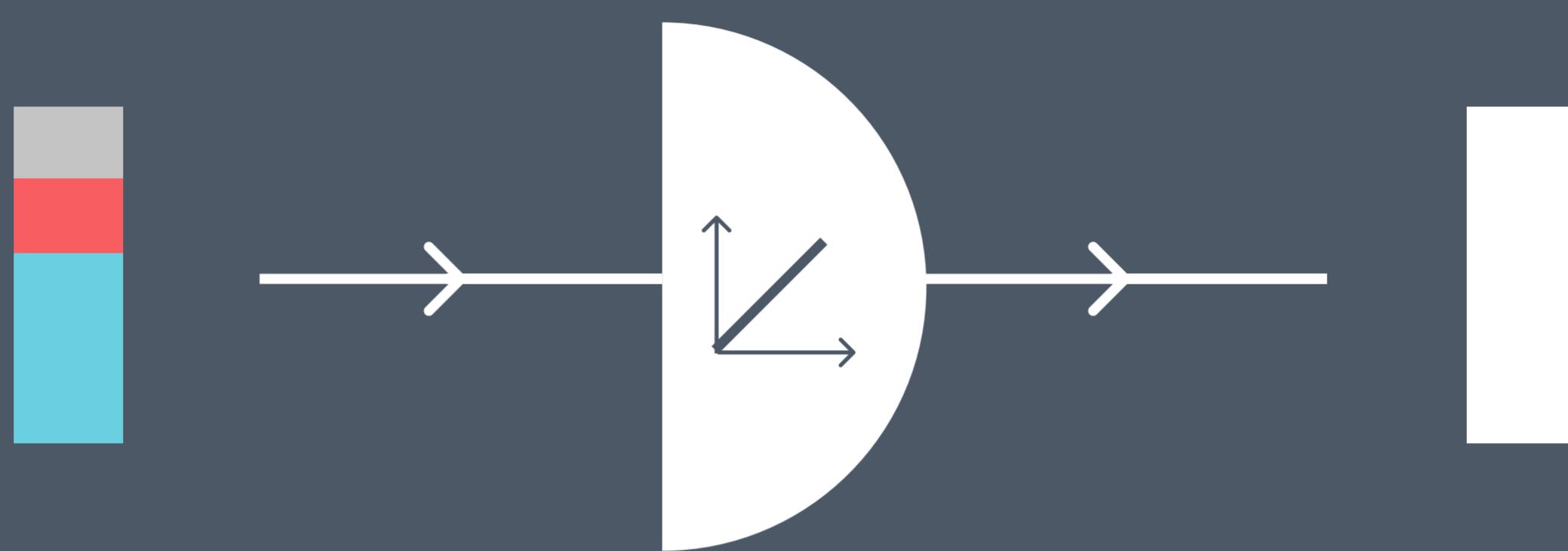
### EXAMPLE

Here we have a neuron with two inputs, 3.0 and 2.0. Given different weight values, it will correspondingly output different values.



## ACTIVATION

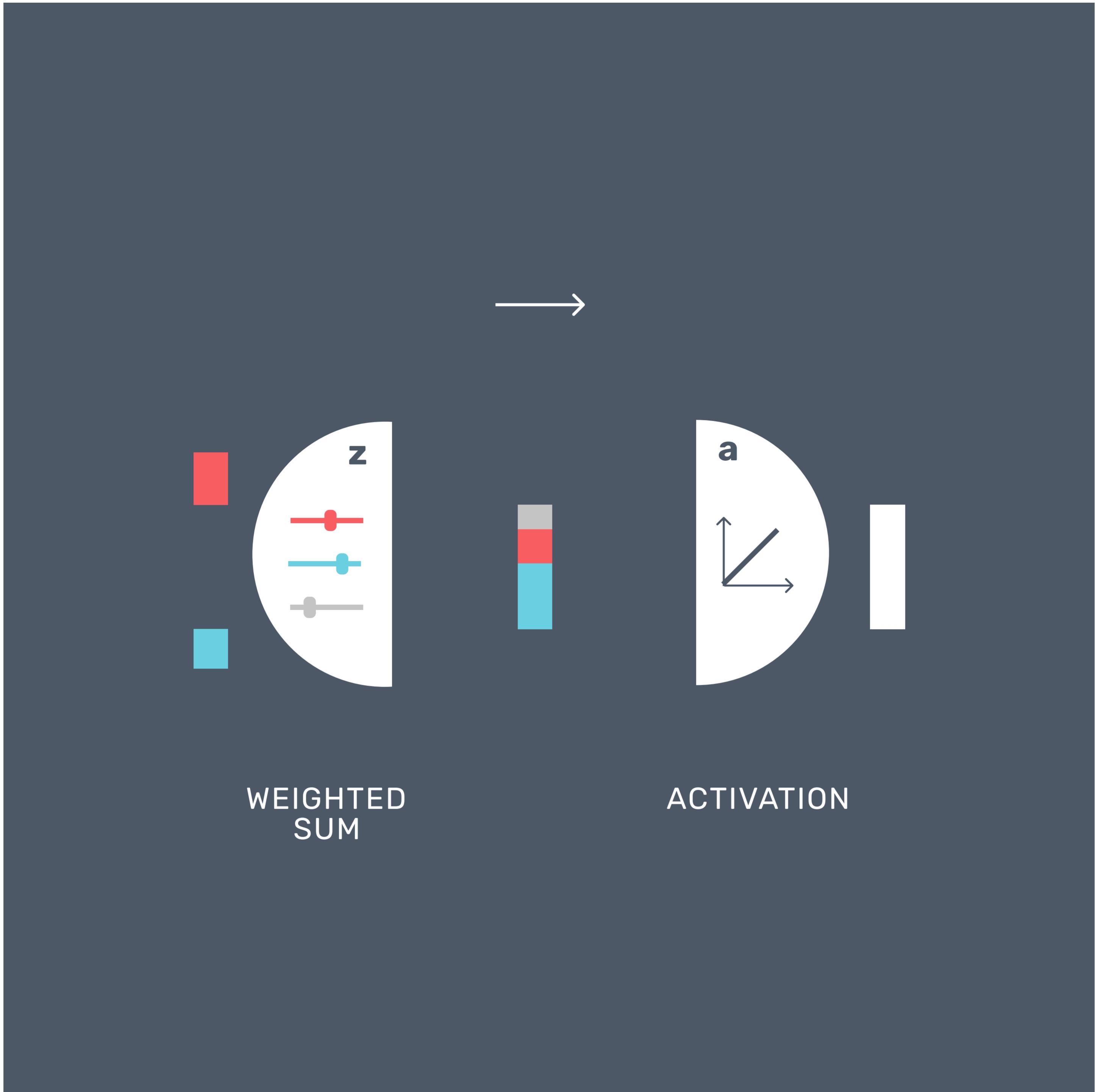
The second computation performed by the neuron is called an *activation*. This is done by taking the output of the weighted sum and passing it through an *activation function*.



## LINEAR ACTIVATION

The activation function gives the neural network the ability to express itself. This will not make much sense now but will become clear by the end of this book.

There are a few common activation functions. To start with, here we have a linear activation function. It's as basic as it gets - it simply outputs the same input it receives. Plotted on a chart, it gives a straight-line relationship between the input and the output.

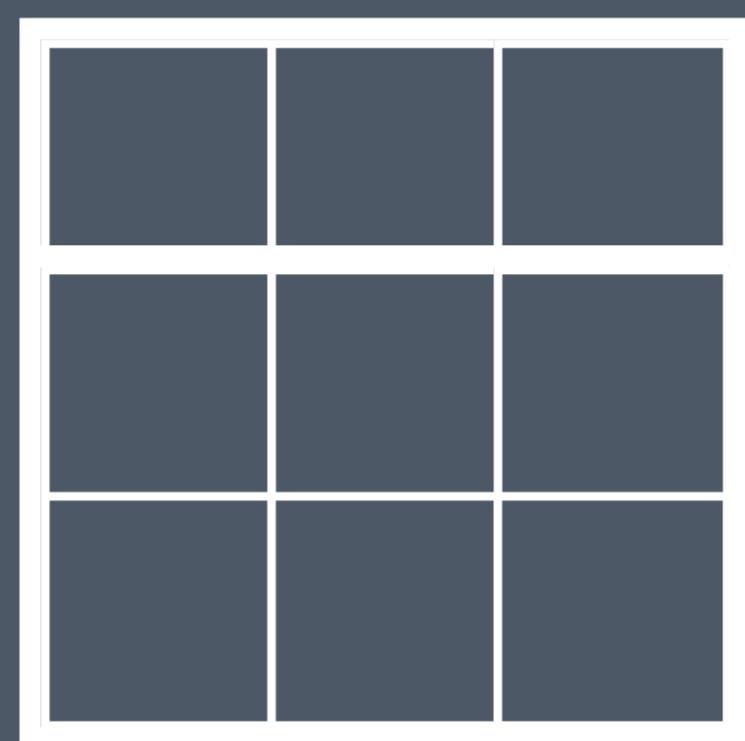


## RECAP

Let's do a quick recap. When inputs are passed through a neuron, it performs a sequence of computations.

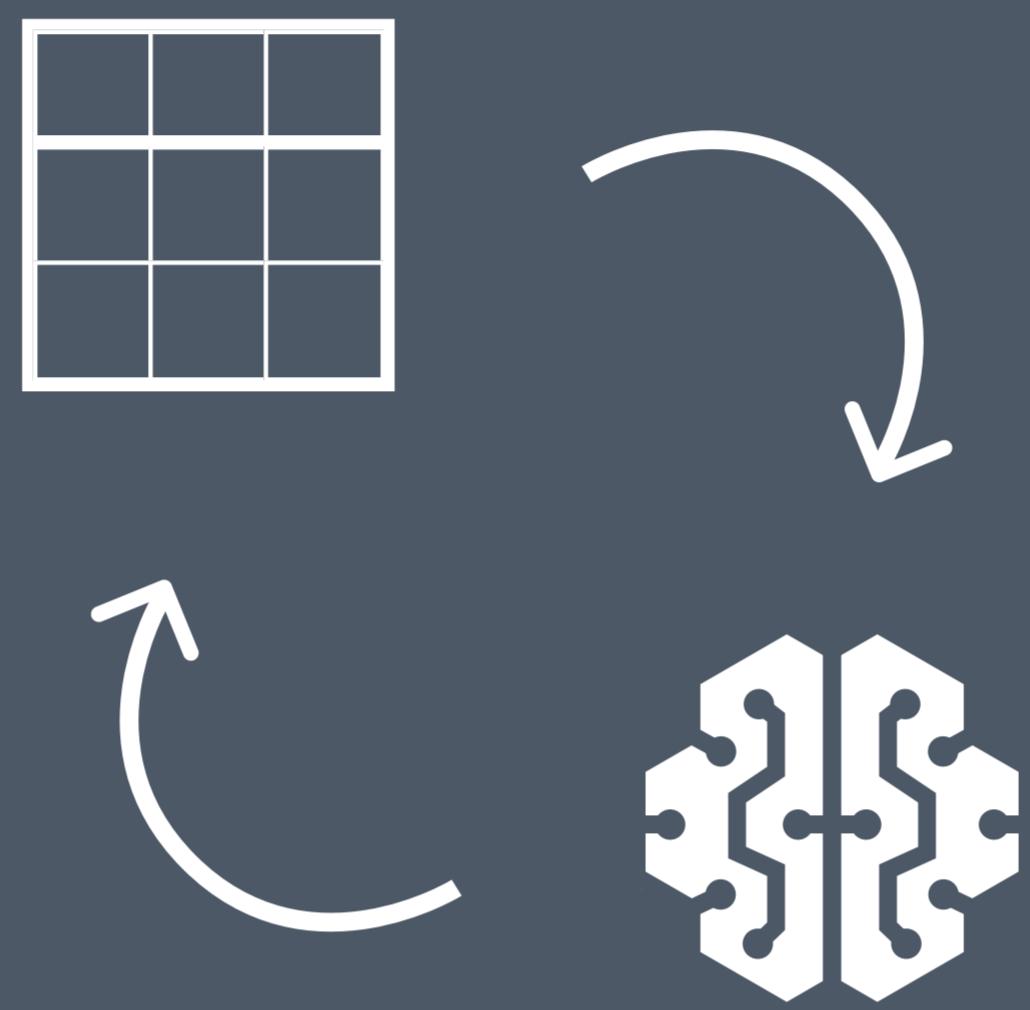
First it performs a weighted sum by multiplying each input by its corresponding weight, summing them, and finally adding a bias term.

Then it performs an activation via an activation function, which in our case, is a linear function.



## DATA

Neural networks are nothing without data. Let's now turn our attention to data and what it brings.



## LEARNING

Data is to the neural network as experience is to humans.

A machine learning algorithm, in this case a neural network, uses data to find useful patterns and relationships. It uses these insights to learn and update itself.

### STRUCTURED DATA

XX	X	X
X	XX	X
XX	X	X

### SEMI-STRUCTURED DATA

```
xx : {  
    xx : x,  
    x :xxxxx,  
    xxx : {  
        xx : xxx,  
        xxxx : xx,  
    }  
}
```

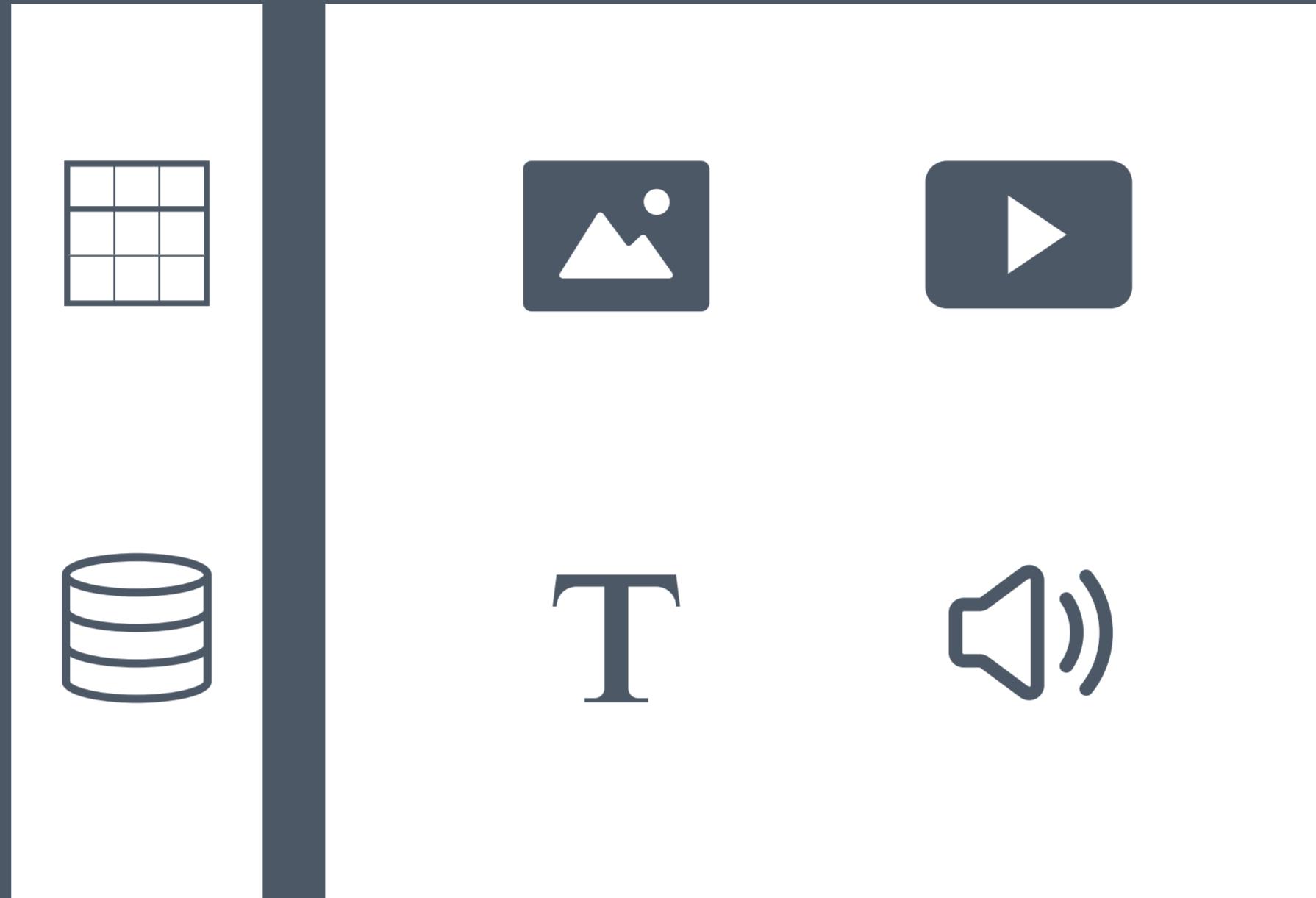
### UNSTRUCTURED DATA

xxx xx xxxxxx x xx  
xx x xxx x xx xxxx.  
xxx, x xxx.  
  
x xxx xxxx, x xx  
xxxxx xx, xxxx x xx x.  
xxx x xx xx xxx x xx  
xx xx.

## TYPES OF DATA

Data can come in many different forms. The most obvious form of data is the tabular format. This is an example of *structured* data, where each data point and its properties can be deciphered in a straightforward manner.

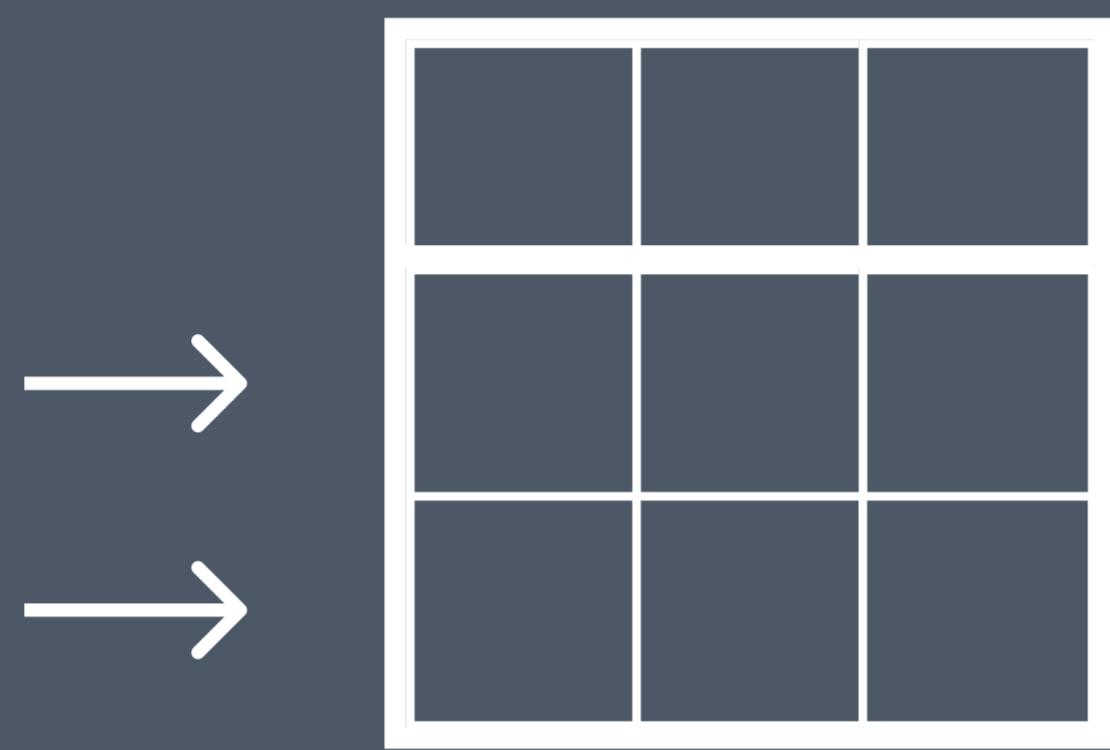
But data can come in other forms too.



## SOURCES OF DATA

In fact, most of the data around us are in the *unstructured* form. According to projections from IDC, 80 percent of worldwide data will be unstructured by 2025.

And indeed, most of the exciting innovations in deep learning today come from unstructured data, such as text, images, videos, and so on.



## A DATASET

Now let's look at how we should prepare a *dataset* for the neural network.

A dataset is composed of many data points. A *data point* is a single observation that we collect and record.



DISTANCE (MI)	RATING
1.5	3.6

## EXAMPLE

Let's take the example of hotel room rates, a dataset we'll use throughout this book.

Each data point represents a hotel. Here we have a hotel with a distance of 1.5 miles from the city center and a guest rating of 3.6 stars.

## FEATURES

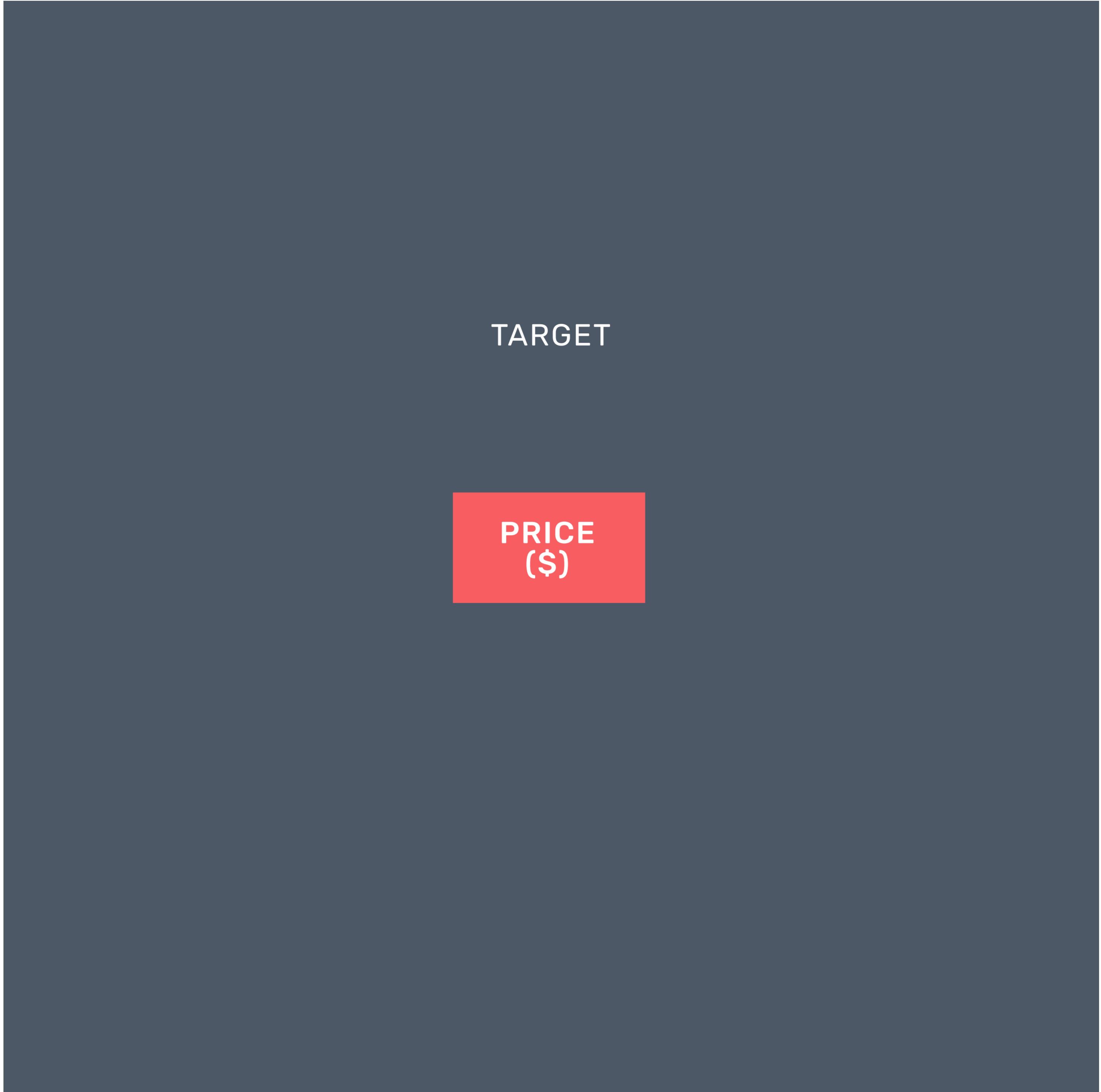
DISTANCE (MI)	RATING
------------------	--------

## FEATURES

These two pieces of information are called *features*. Features describe the properties of a data point.

Each data point in a dataset is described with the same features, of course with different values, making each of them unique.

From now on, we'll refer to these two features as *distance* and *rating* for short.



## TARGET

Recall that the goal of a neural network is to make predictions.

In this example, we want to predict the average daily room rate (or *price* for short) for any given hotel. This means, given the two features earlier, we want to predict how much each hotel will cost.

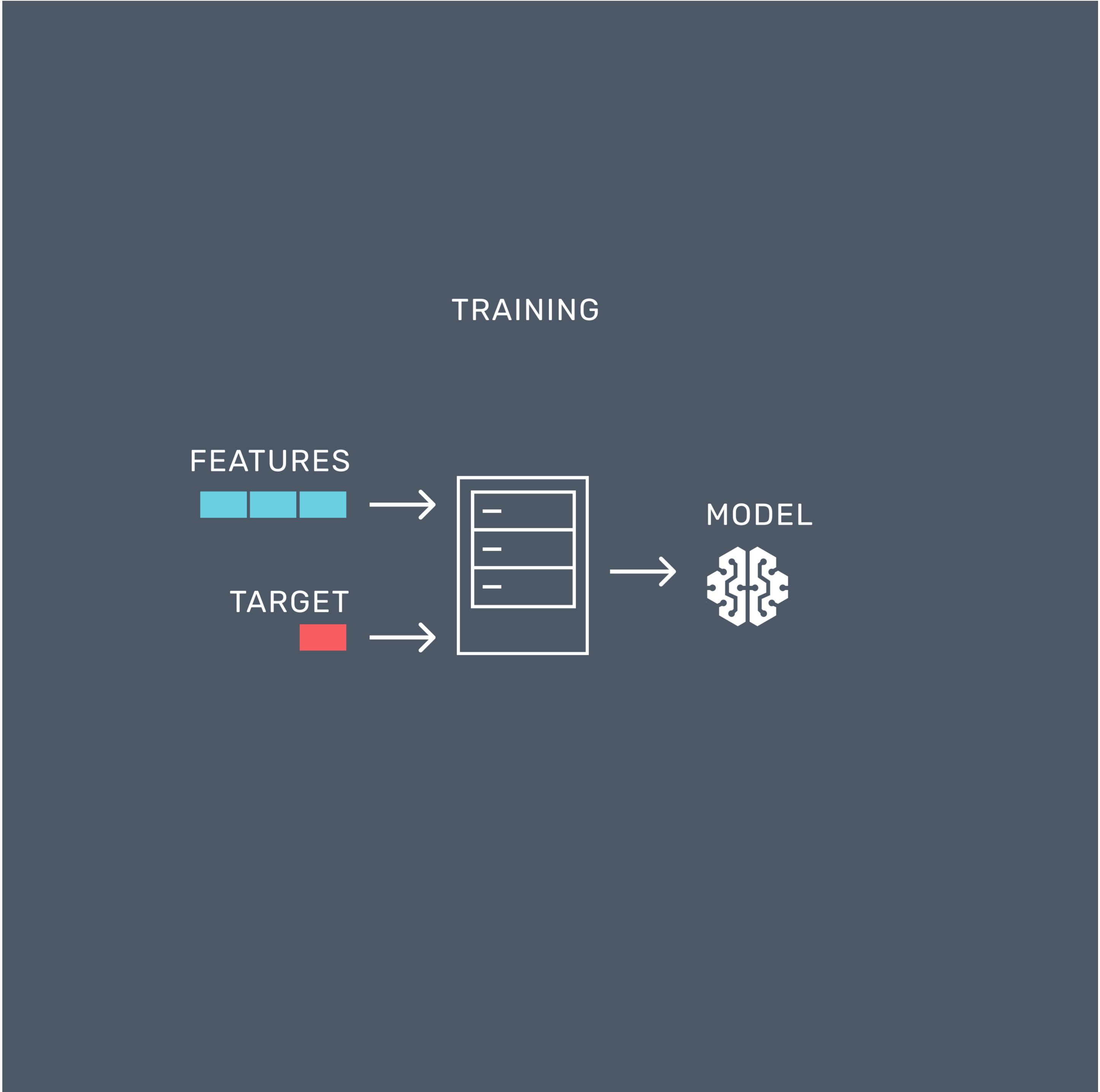
The is called the *target*. In other resources, you may also find the term *label* being used.

DIST (MI)	RATING	PRICE (\$)
0.8	2.7	147
1.5	3.6	136
...	...	...
19.4	4.8	209

## TRAINING

We'll give the neural network enough data points containing the features and target values, which it will learn from.

A machine learning task where we specify the target is called *supervised learning*, which will be our focus in this book.



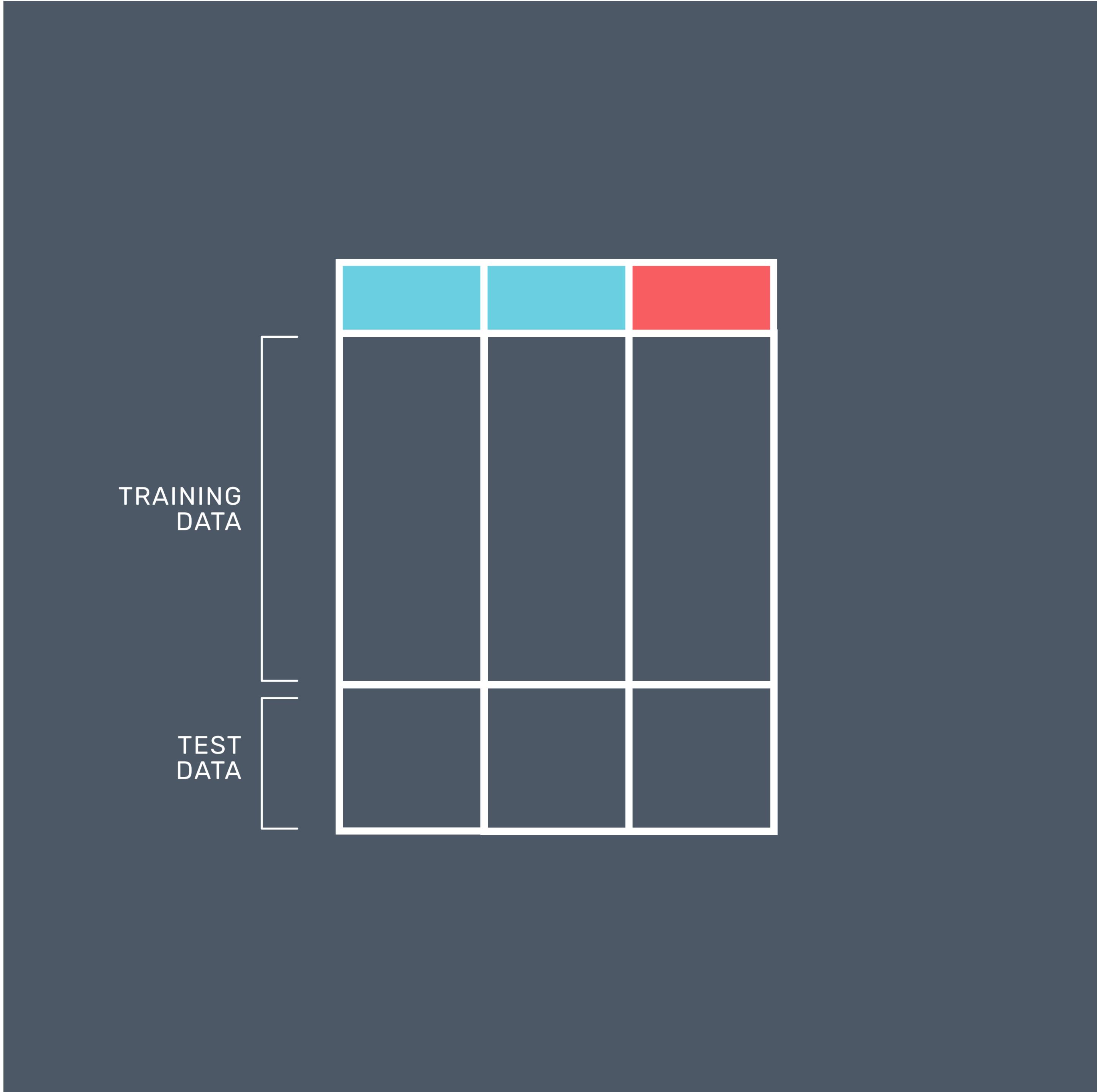
## TRAINING

We have just described a process called *training*.

During training, the neural network learns from the data and updates its parameters. By this point, we'll have a trained *model*.

In short, given a dataset containing features and target, we get a model.

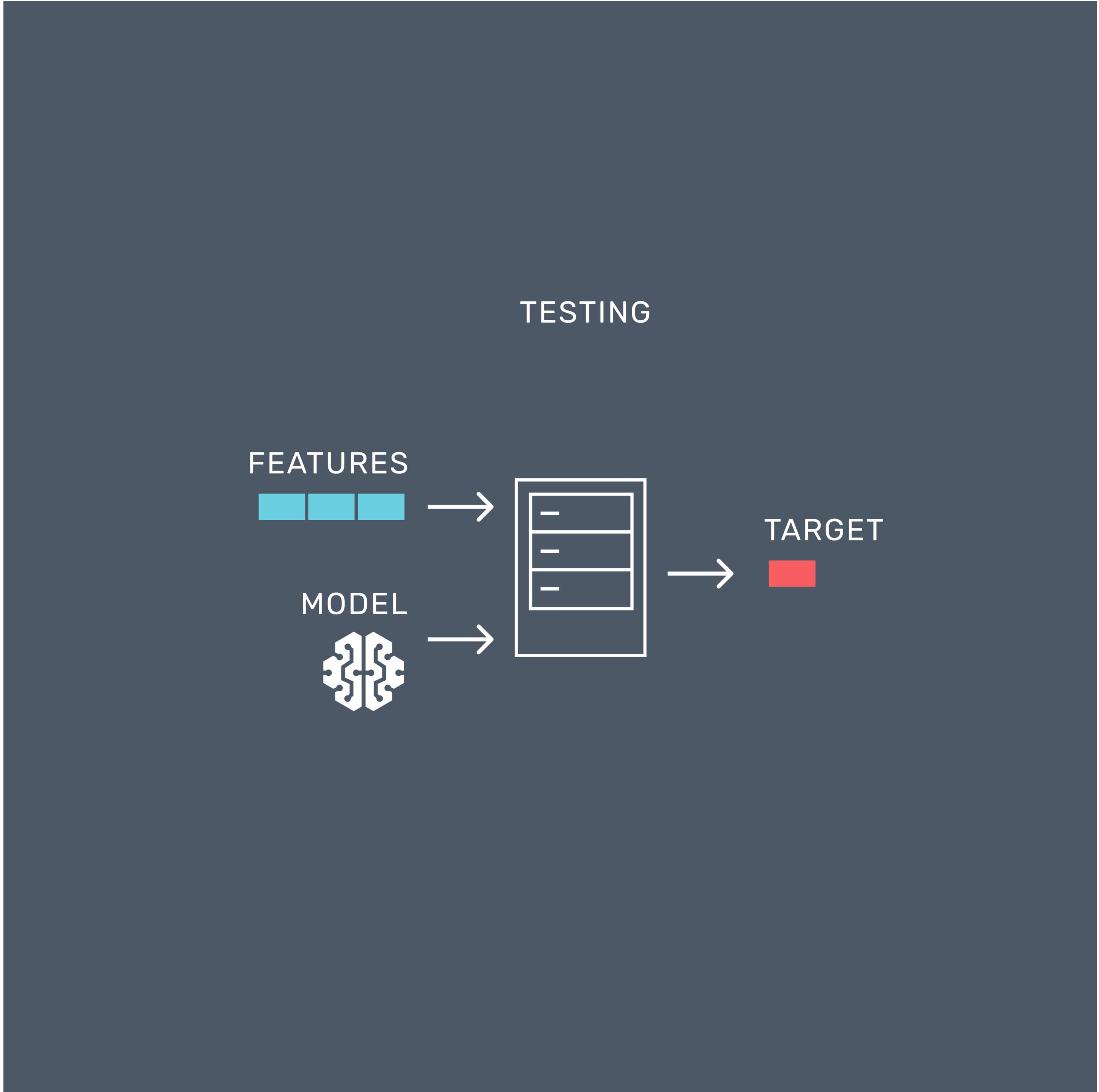
That is why the training process is sometimes also called ‘fitting the model to the data’.



## TESTING

Once the training is complete, we need a way to know how the model is performing.

For this, we'll keep a portion of the dataset for *testing*.

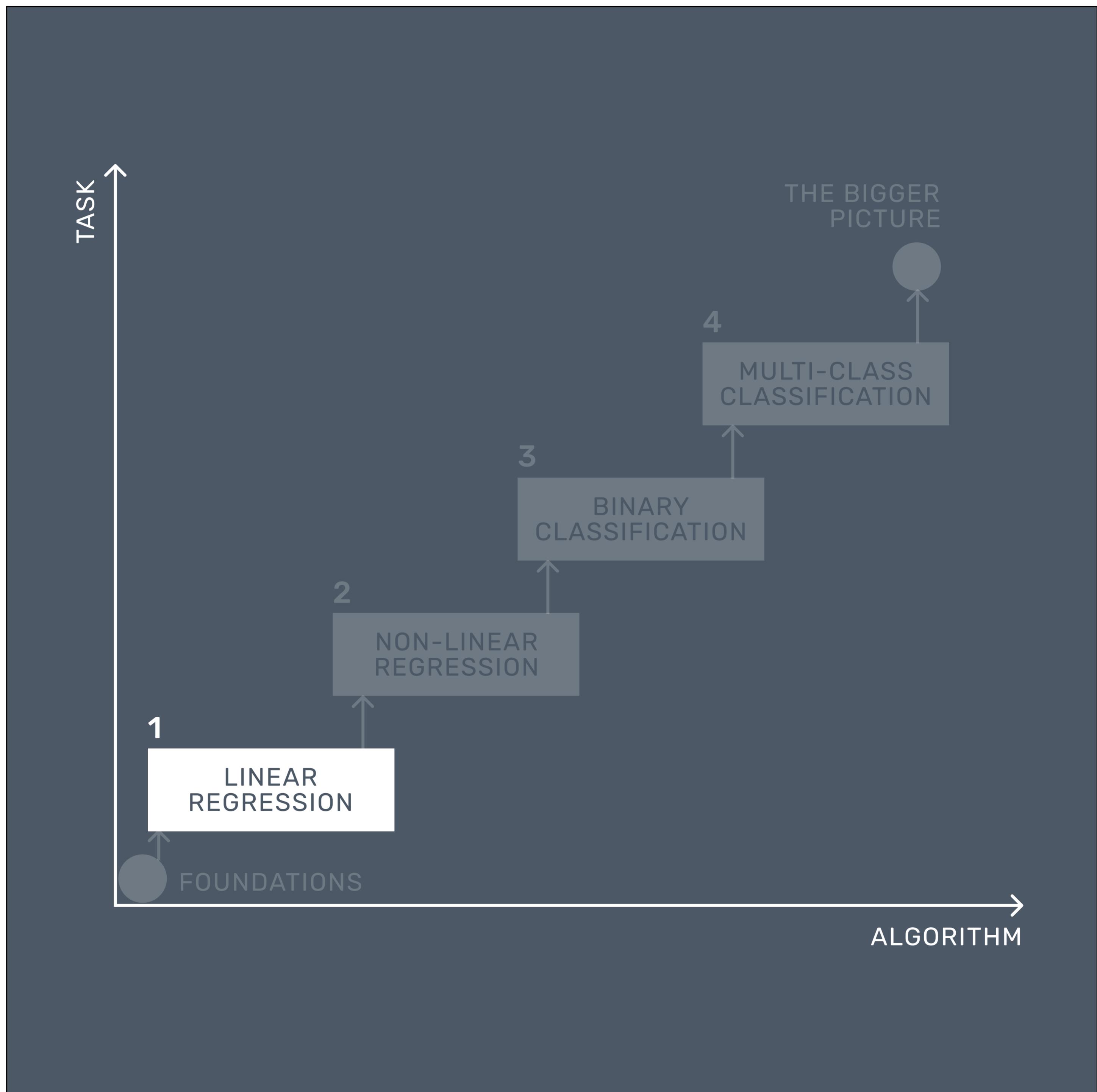


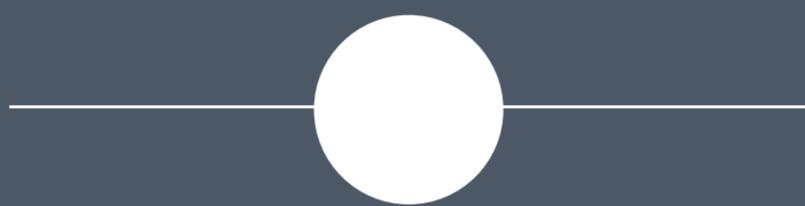
## TESTING

During testing, we'll provide the neural network with just the features, without the target. Now that it's already trained, its job is to predict the target values.

In the coming four chapters, we'll revisit these training and testing steps.

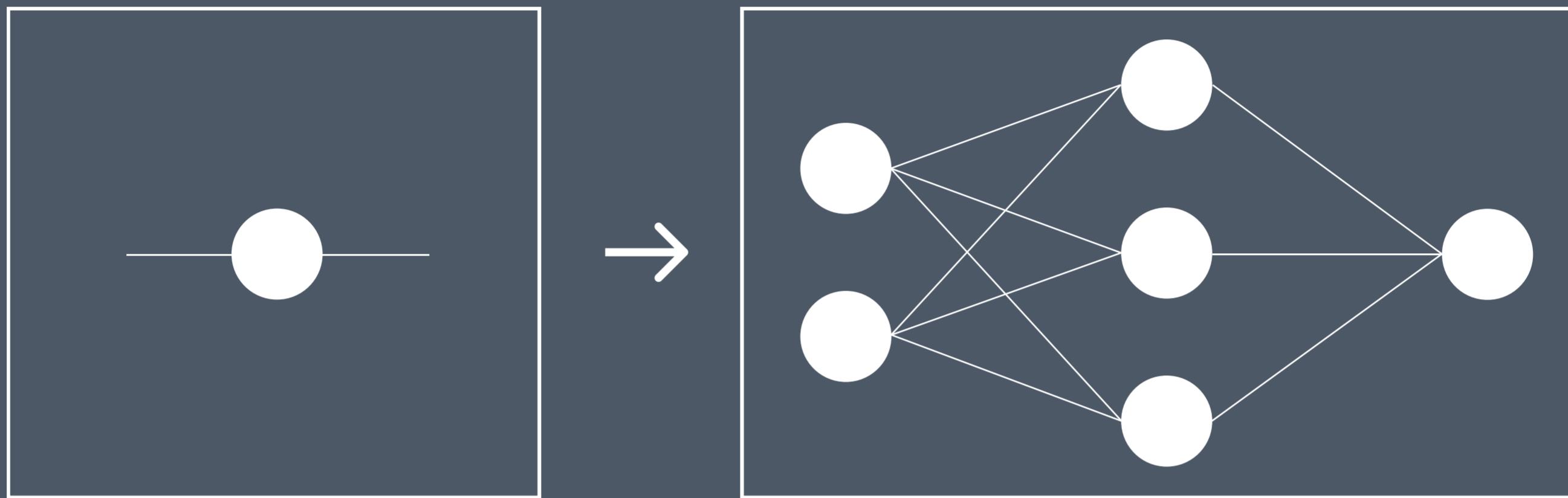
# 1 - LINEAR REGRESSION





## A SINGLE-NEURON NEURAL NETWORK

Now let's look at how the neural network works. We'll start with its simplest possible version—a network with only one neuron and one input!



## THE PLAN

We'll lay the necessary foundation in this chapter and use that in the subsequent chapters when we start building larger neural networks.



## THE GOAL

Let's revisit the dataset from the previous chapter, which contains a list of hotels in a city.

Our goal is to predict the average daily room rate for a given hotel (i.e. *price*) based on the features.

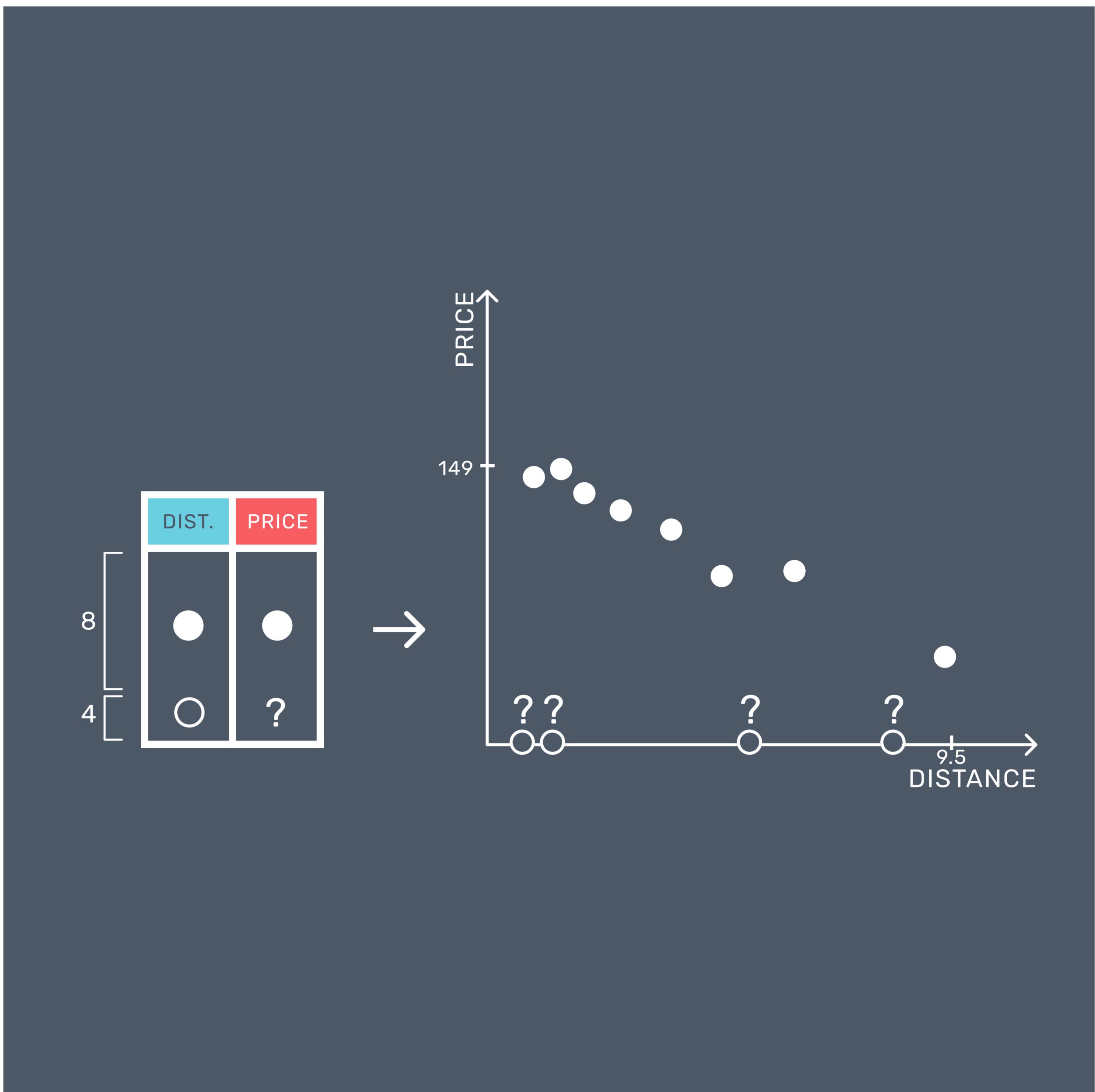
In this chapter, we'll use only one of the features—the distance from the city center (i.e. *distance*).

DISTANCE (MI)	PRICE (\$)
0.5	146.00
1.1	149.00
1.6	140.00
2.4	134.00
3.5	127.00
4.6	110.00
6.2	112.00
9.5	81.00
0.3	156.00
0.7	168.00
4.9	116.00
8.5	99.00

## THE DATASET

This is what the dataset looks like. It contains twelve data points, one feature, and one target.

The distance and price values are *continuous values*—numeric values that can take any value within a range.

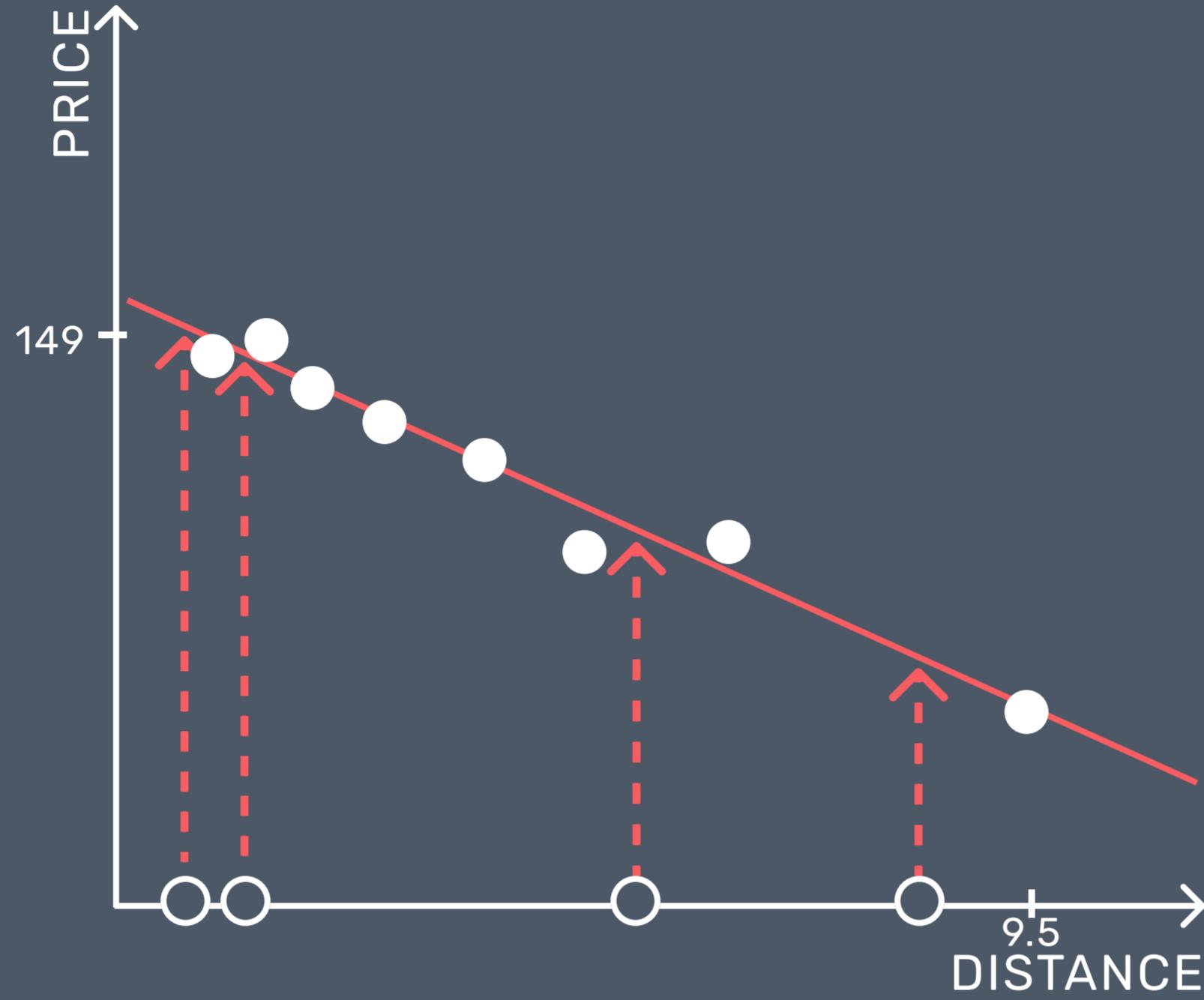


## REGRESSION

This is a type of task called *regression*. In regression tasks, the target value to be predicted is a continuous value.

We'll split the dataset into the training and testing datasets and train the model using the training data.

Ultimately, we want the model to give good predictions for the four test data points.



## LEARNING

By visual inspection, it's clear that there is a straight-line relationship between the feature and the target. This is called *linear regression*.

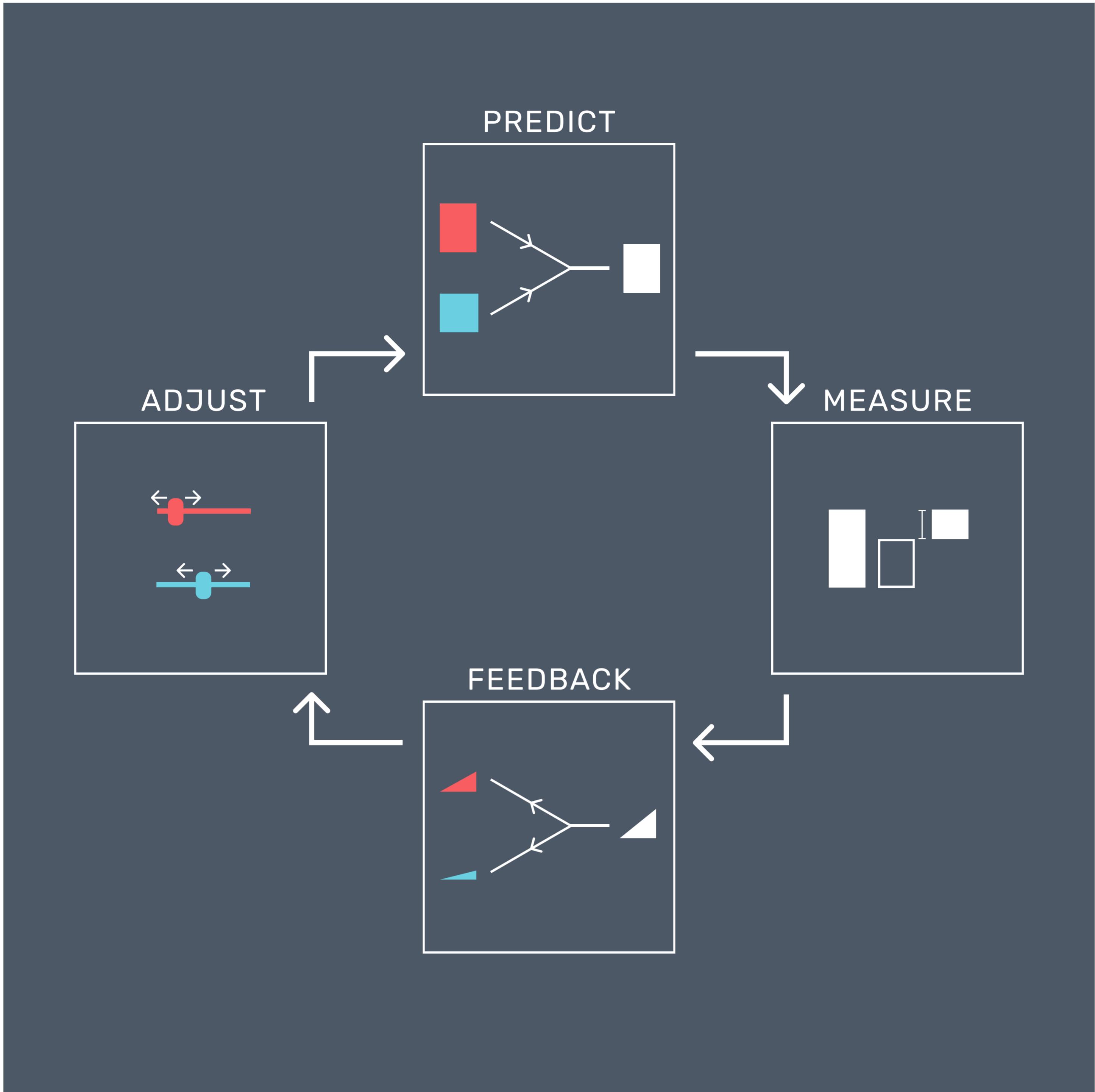
This is the relationship that we want our single-neuron network to capture. Let's see if it can do that.

TRAINING DATA

	DISTANCE (MI)	PRICE (\$)
	0.5	146.00
	1.1	149.00
	1.6	140.00
	2.4	134.00
	3.5	127.00
	4.6	110.00
	6.2	112.00
	9.5	81.00
	0.3	156.00
	0.7	168.00
	4.9	116.00
	8.5	99.00

## TRAINING DATA

We'll take the first eight data points as training data and leave the other four for testing later.



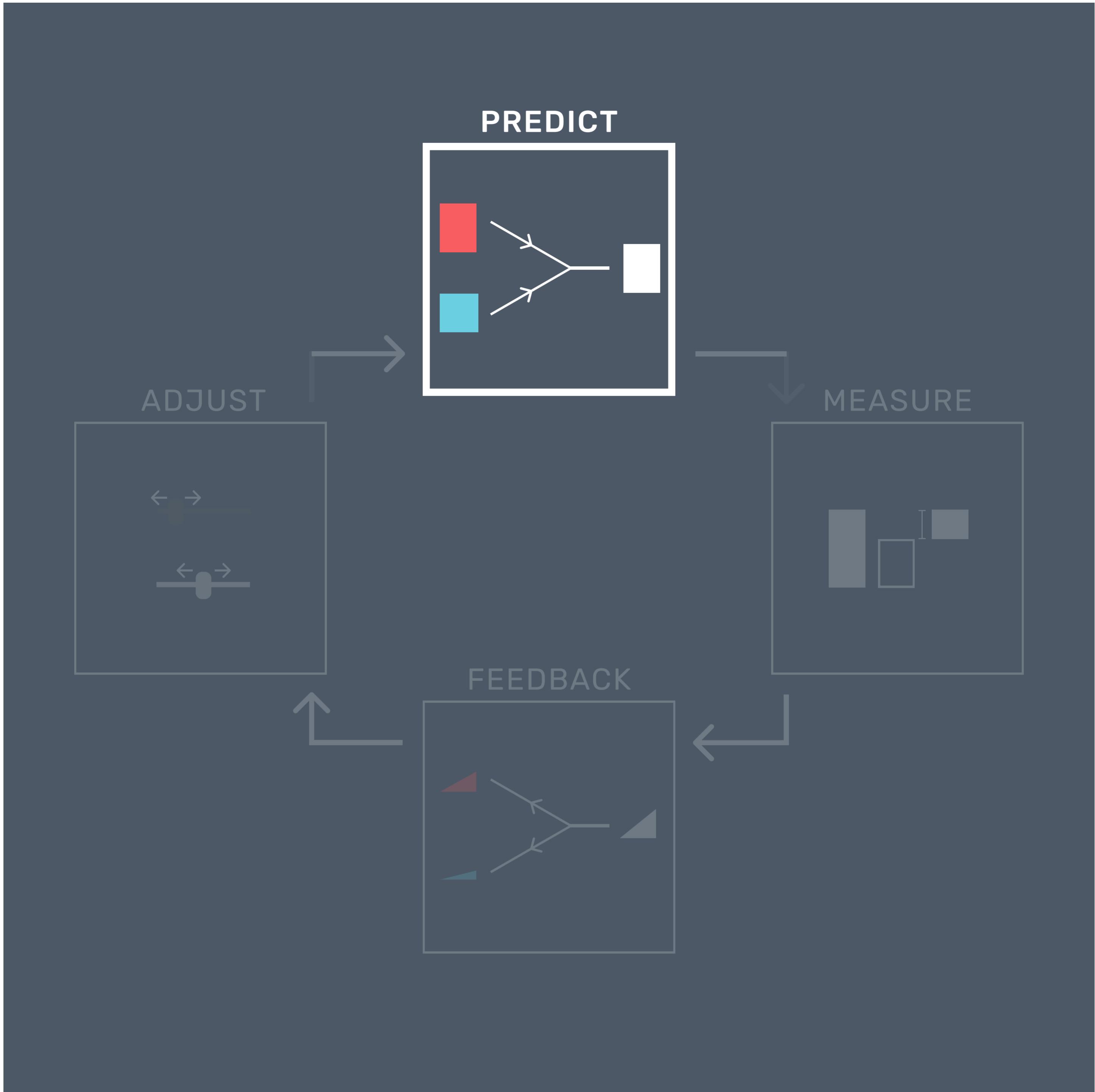
## THE FOUR STEPS OF TRAINING

We can think of training a neural network as a four-step cycle: *Predict - Measure - Feedback - Adjust*.

One cycle represents one training round. This is repeated for a number of rounds until the neural network becomes good at the task it's training for.

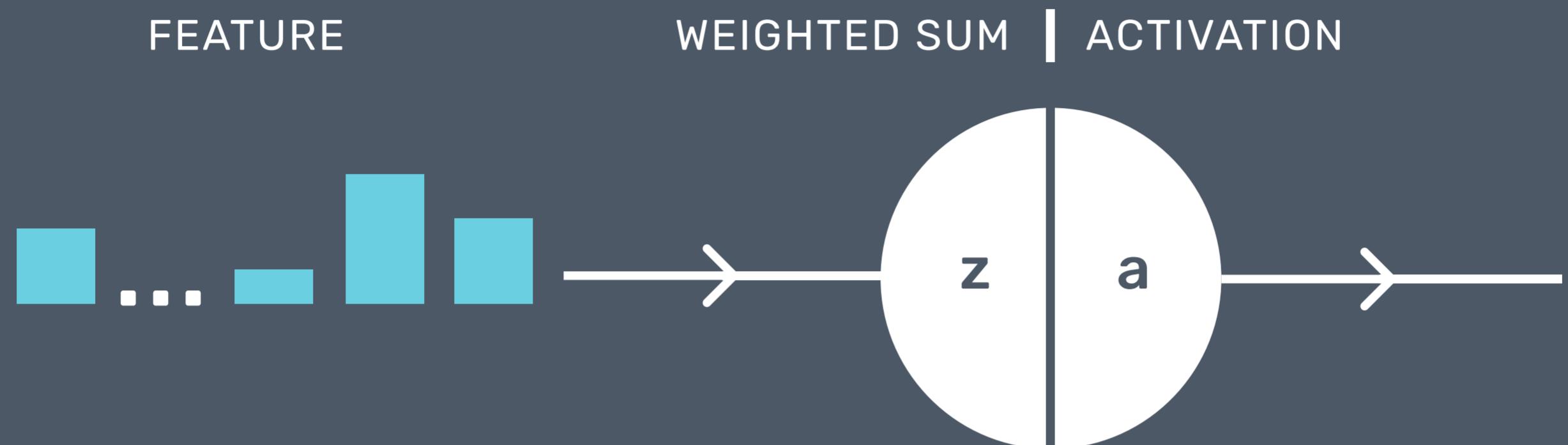
None of this will make sense to you yet, but that's exactly what we'll learn about next!

Also note that these four terms were chosen for this book to make it easy for someone new to deep learning. In other resources, you will find other terms used (e.g. *forward* instead of predict, *backward* instead of feedback, *update* instead of adjust). They refer to the same concepts.



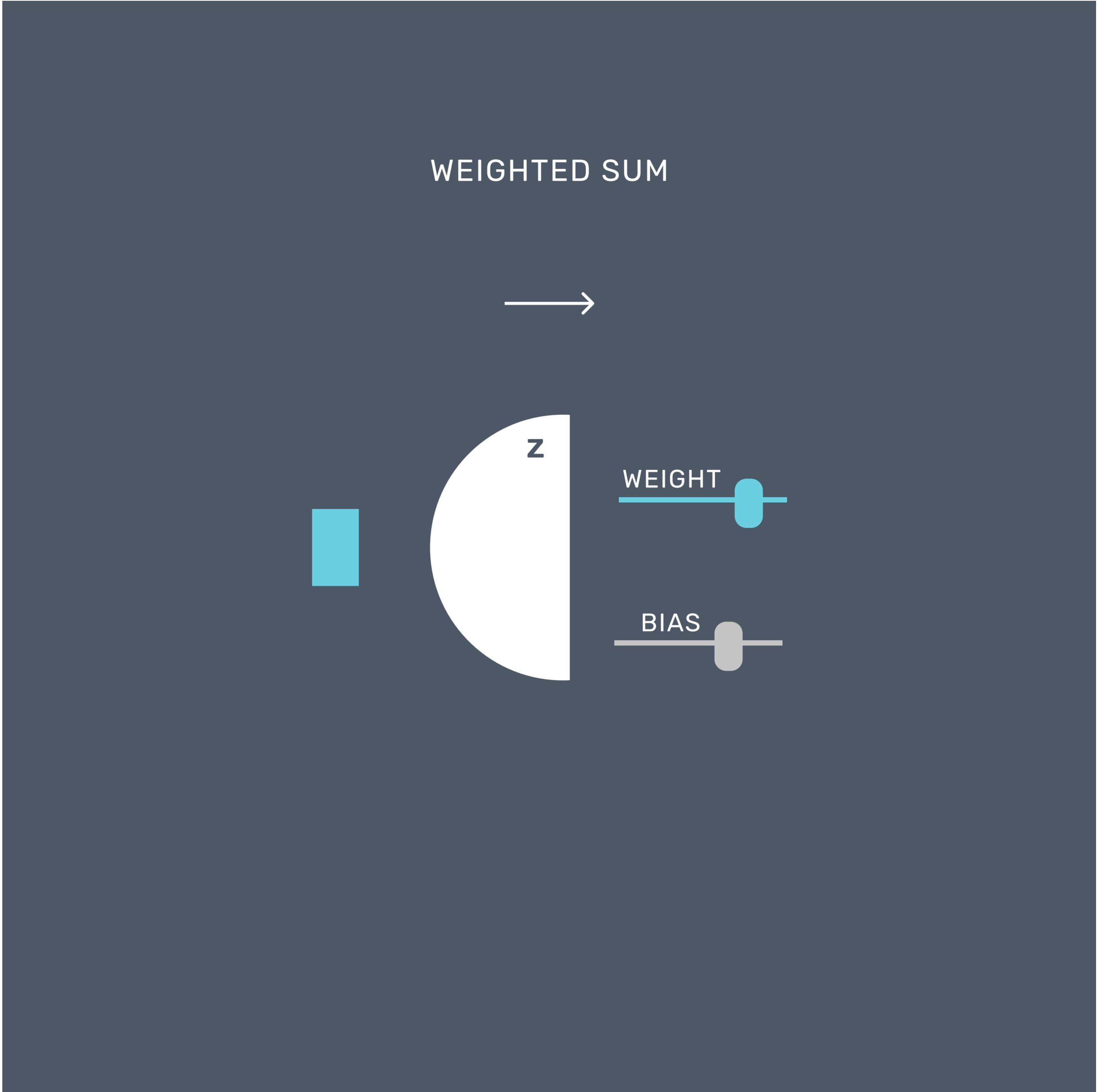
## PREDICT

In the first step of the cycle, *predict*, we'll pass the training data through the neuron.



## NEURON COMPUTATIONS

Recall that this means going through two steps of computations - weighted sum and activation, one data point at a time.



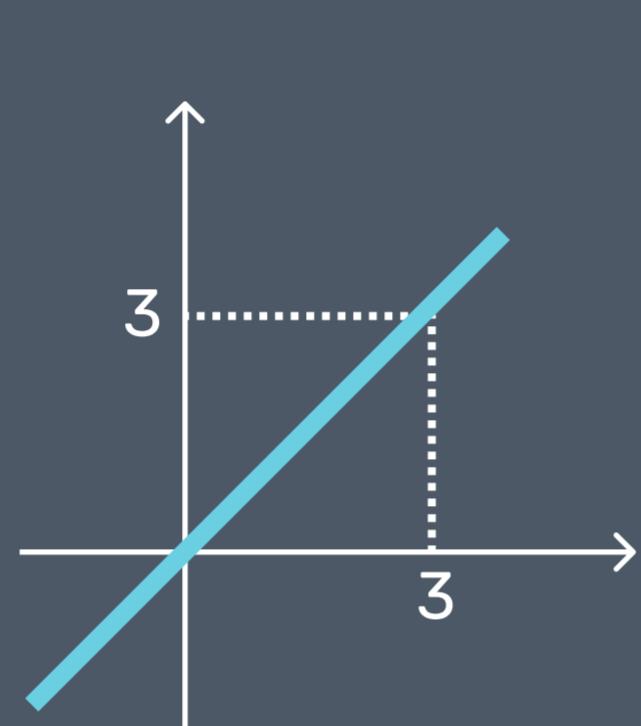
## PARAMETER COUNT

We've already seen that the number of weights of a neuron is equal to the number of inputs. The inputs are the dataset's features. And since we have only one feature, there is going to be only one input, and hence, one weight.

We also saw that on top of that, a neuron has one bias value.

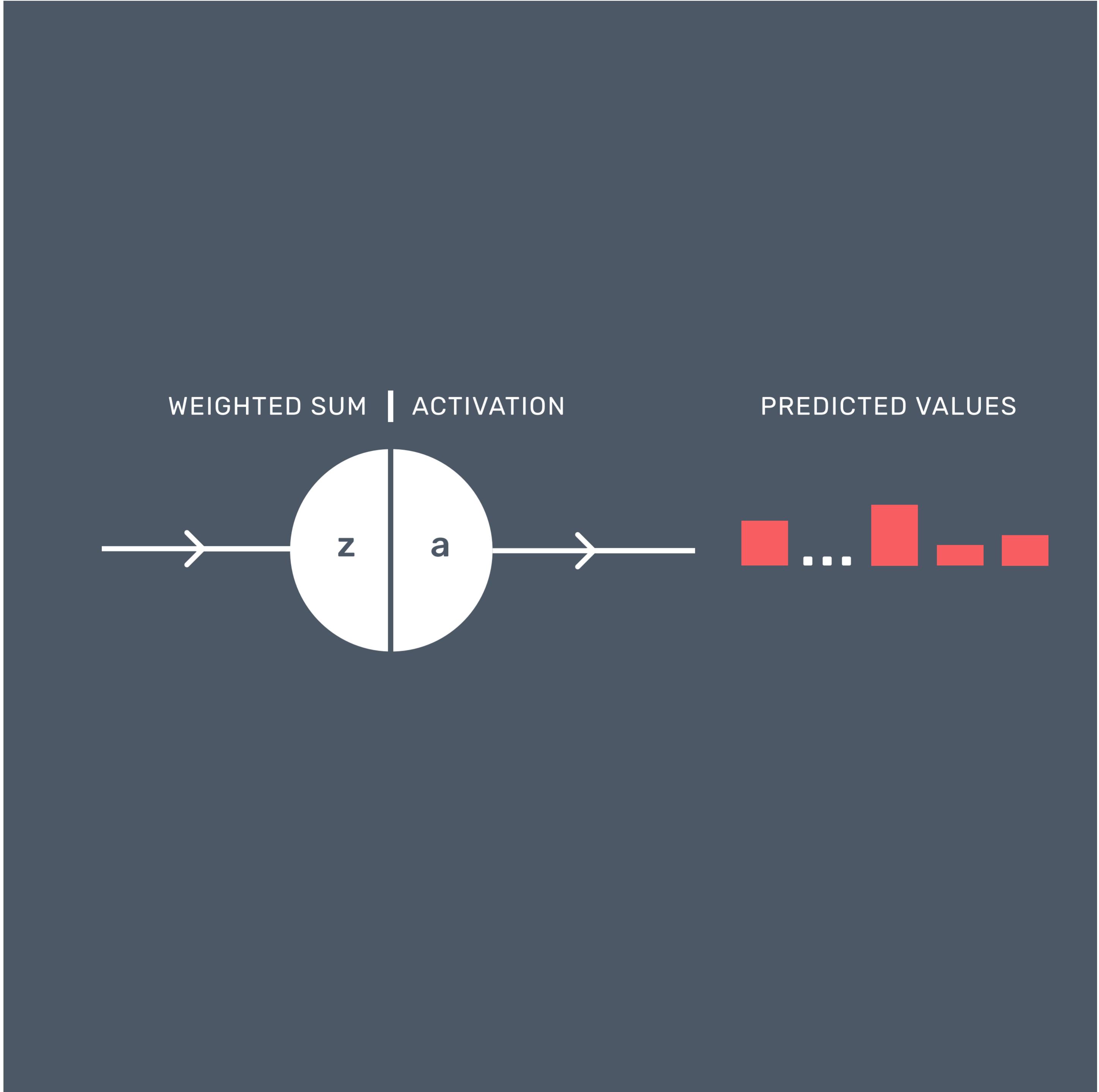
We'll assign initial values for these parameters, in which there are a number of initialization techniques we can choose from. These techniques help the neural network learn more effectively compared to simply assigning random initial values. However, this book doesn't cover this topic as it is quite mathematically involved.

## ACTIVATION



## ACTIVATION

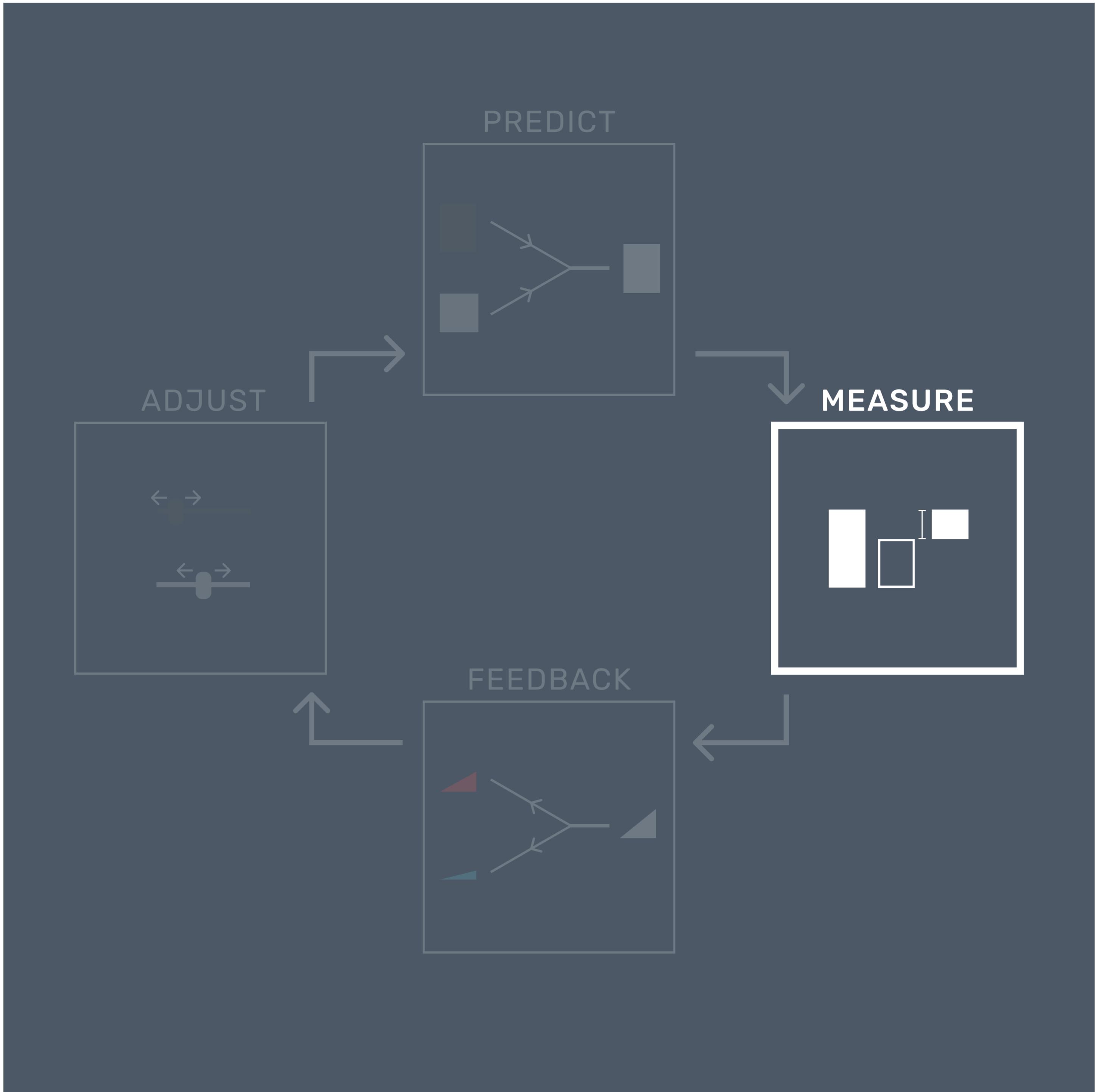
For this task, we'll stick to the linear activation function.



## OUTPUT

By now, we will have the neuron successfully outputting eight values. They represent the prices that the neuron predicted.

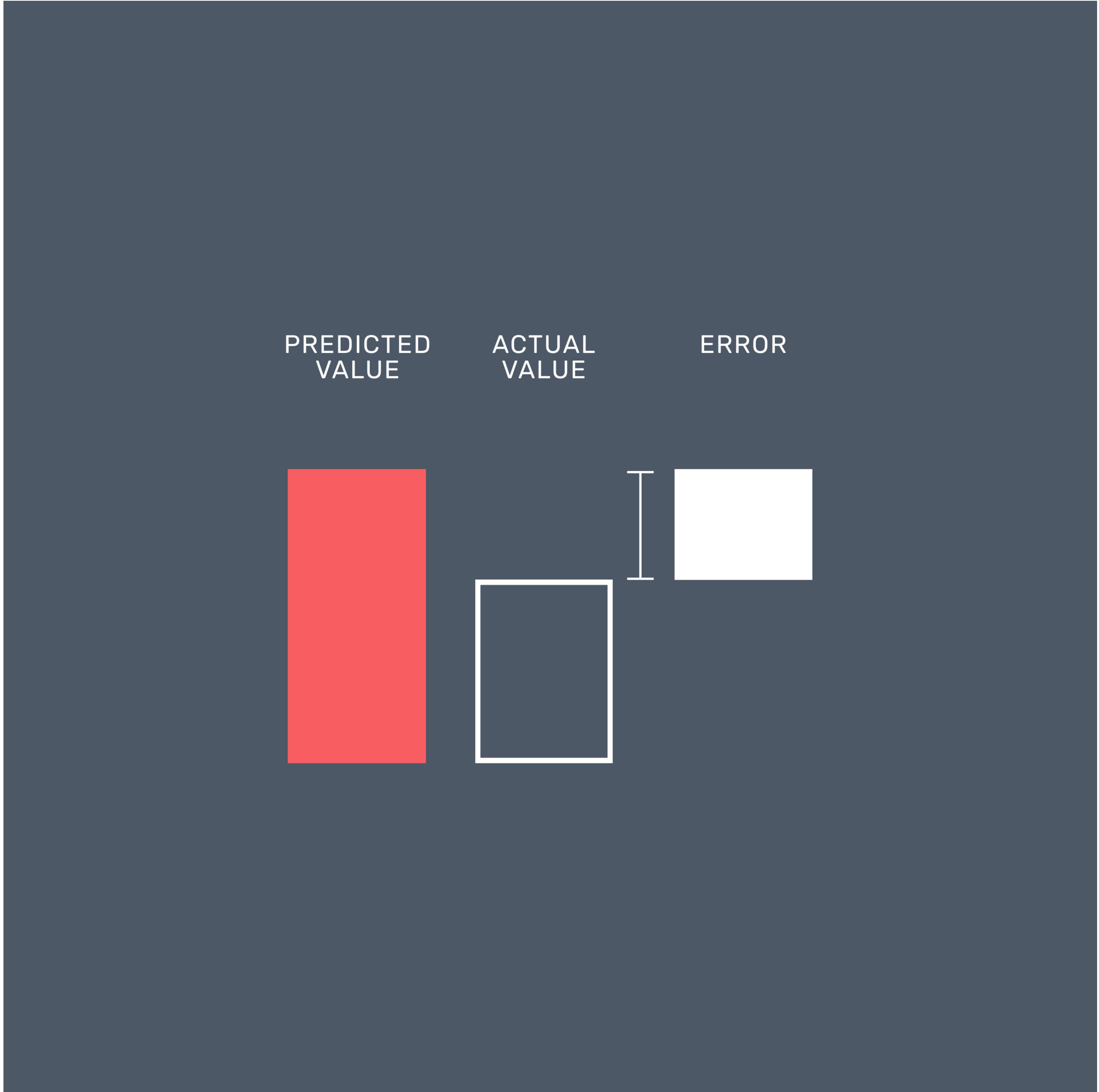
The problem, however, is that the neuron hasn't learned anything yet. As a result, its predictions will be completely wide of the mark.



## MEASURE

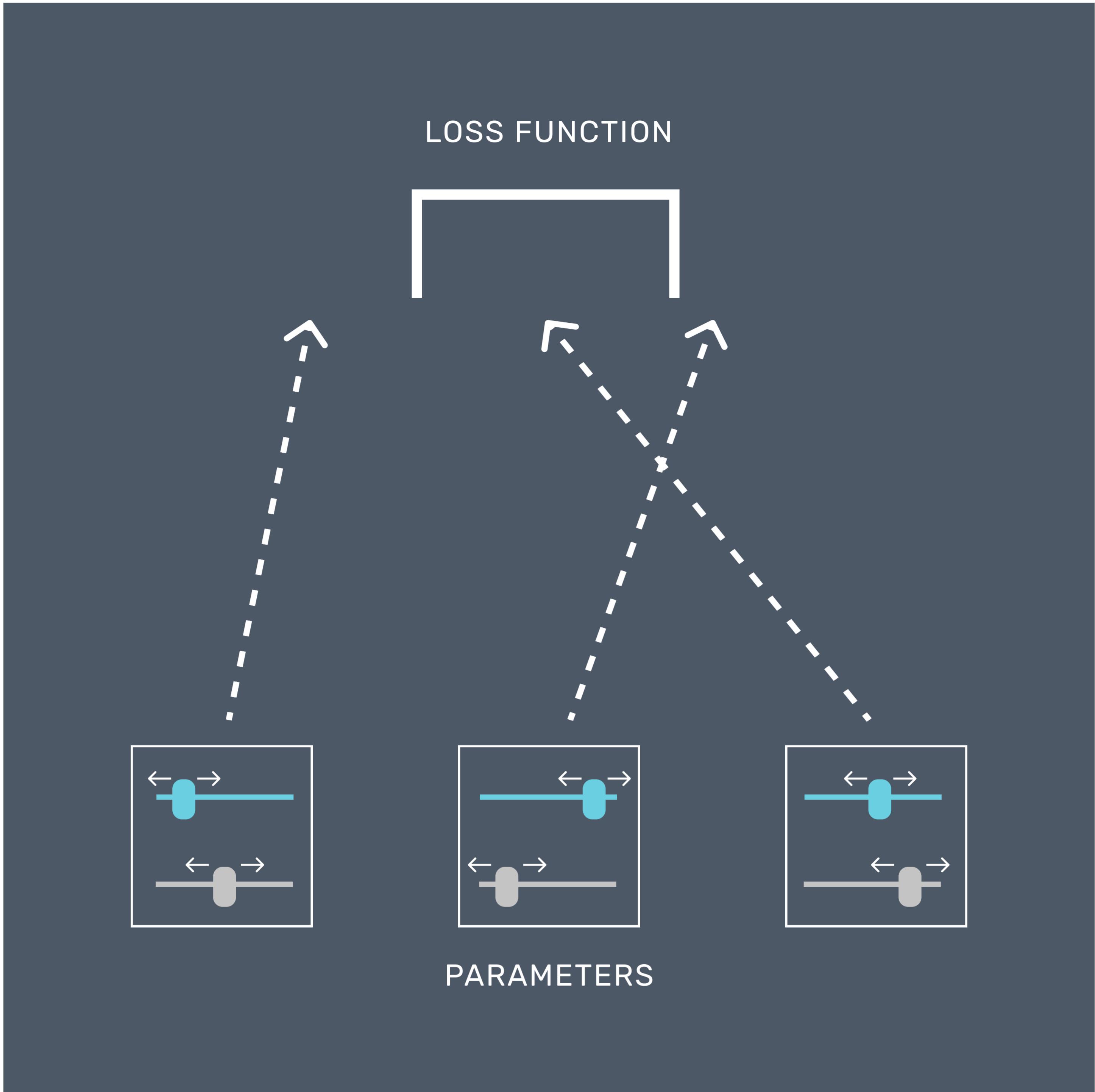
But how do we actually know if the neuron's prediction is good or bad?

This is when we move to the second step, *measure*, where we'll quantify its performance.



## ERROR VALUE

Since we know the actual value of the target, we can quantify the performance by computing the difference between the predicted and actual prices. This is called the *error value*.

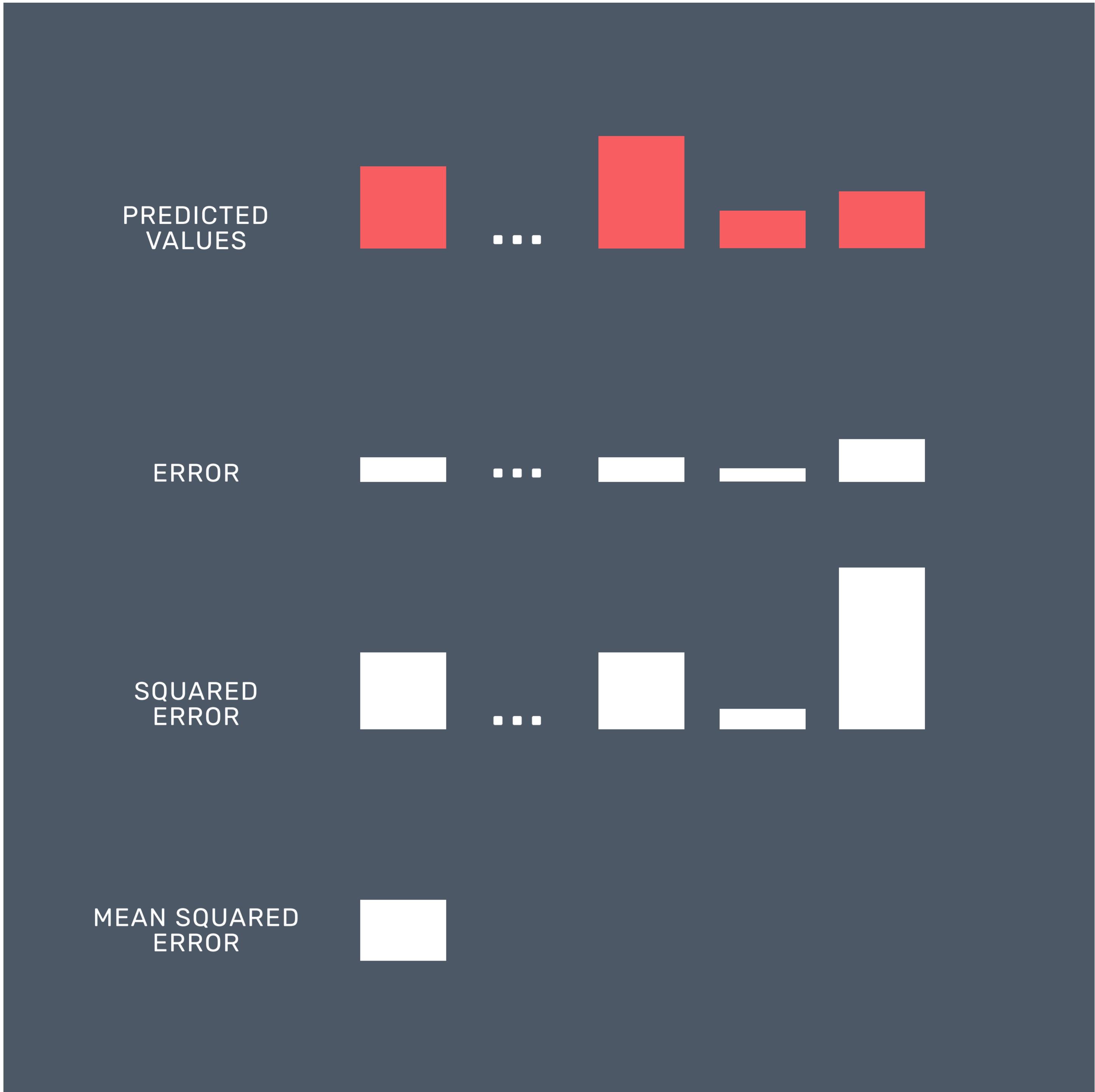


## LOSS FUNCTION

This brings us to one of the most crucial parts of designing a neural network—choosing its *loss function*.

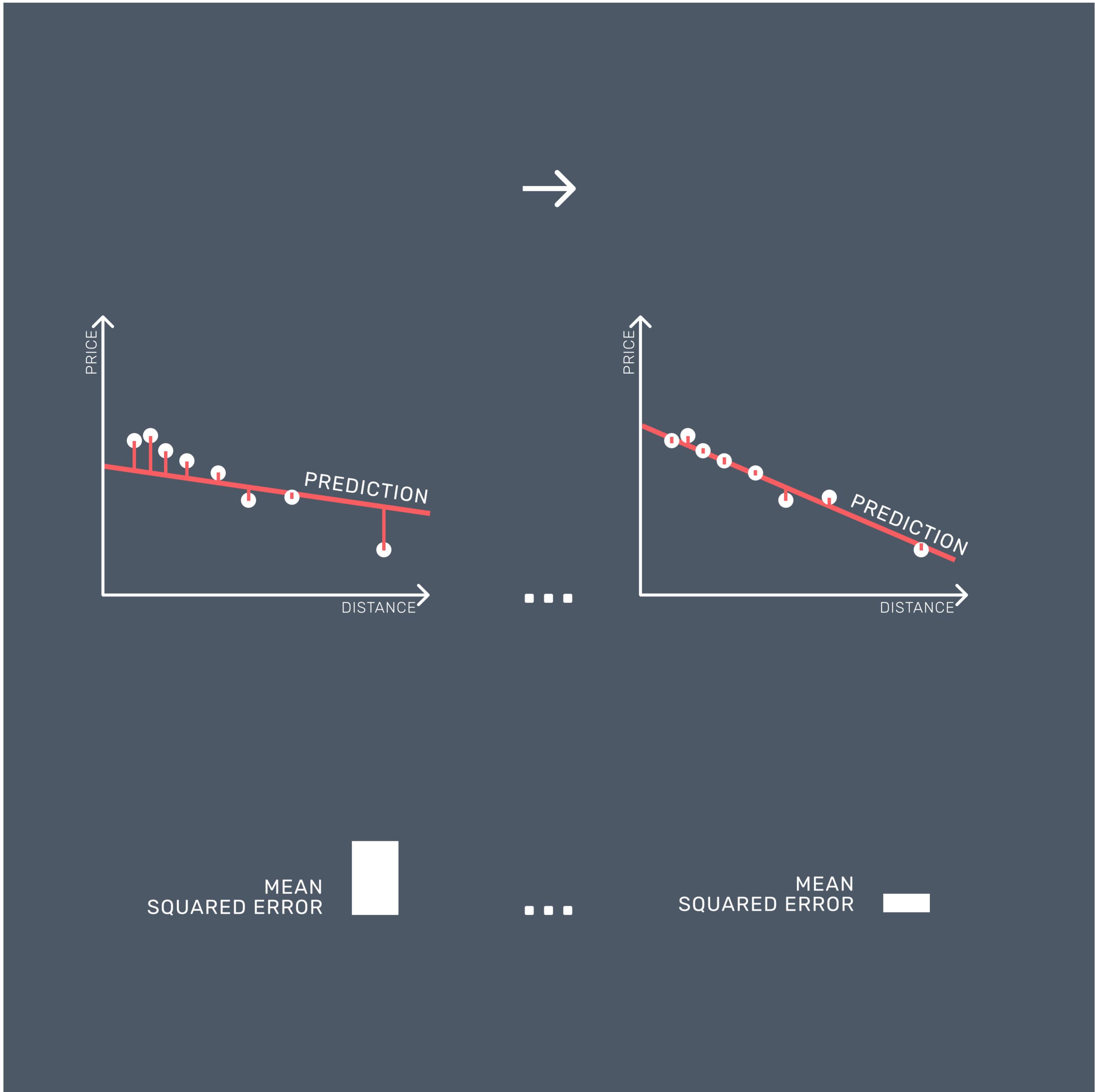
While the parameters are the dials that the network adjusts to reach its goal, the loss function is the goal itself.

The loss function comes in various forms and it all depends on the nature of the task. This will become clearer in Chapters 3 and 4, where we'll use other kinds of loss functions.



## MEAN SQUARED ERROR

The loss function we'll use for this task is called the mean squared error, or MSE for short. Each of the eight error values is squared to get the squared error. Then they are averaged to get the MSE.

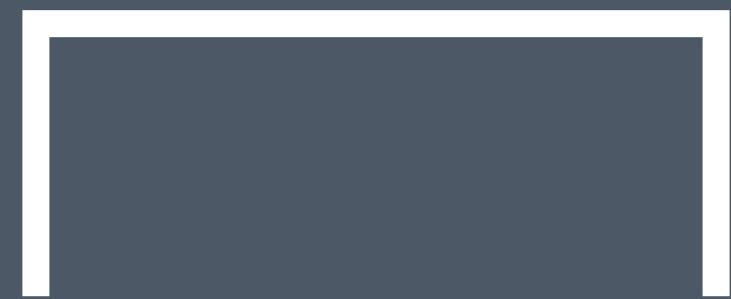


## MINIMIZE LOSS

The MSE is a measure of error. That means the smaller the MSE, the better the network is doing.

In other words, the neuron's goal is to minimize its loss over many training rounds.

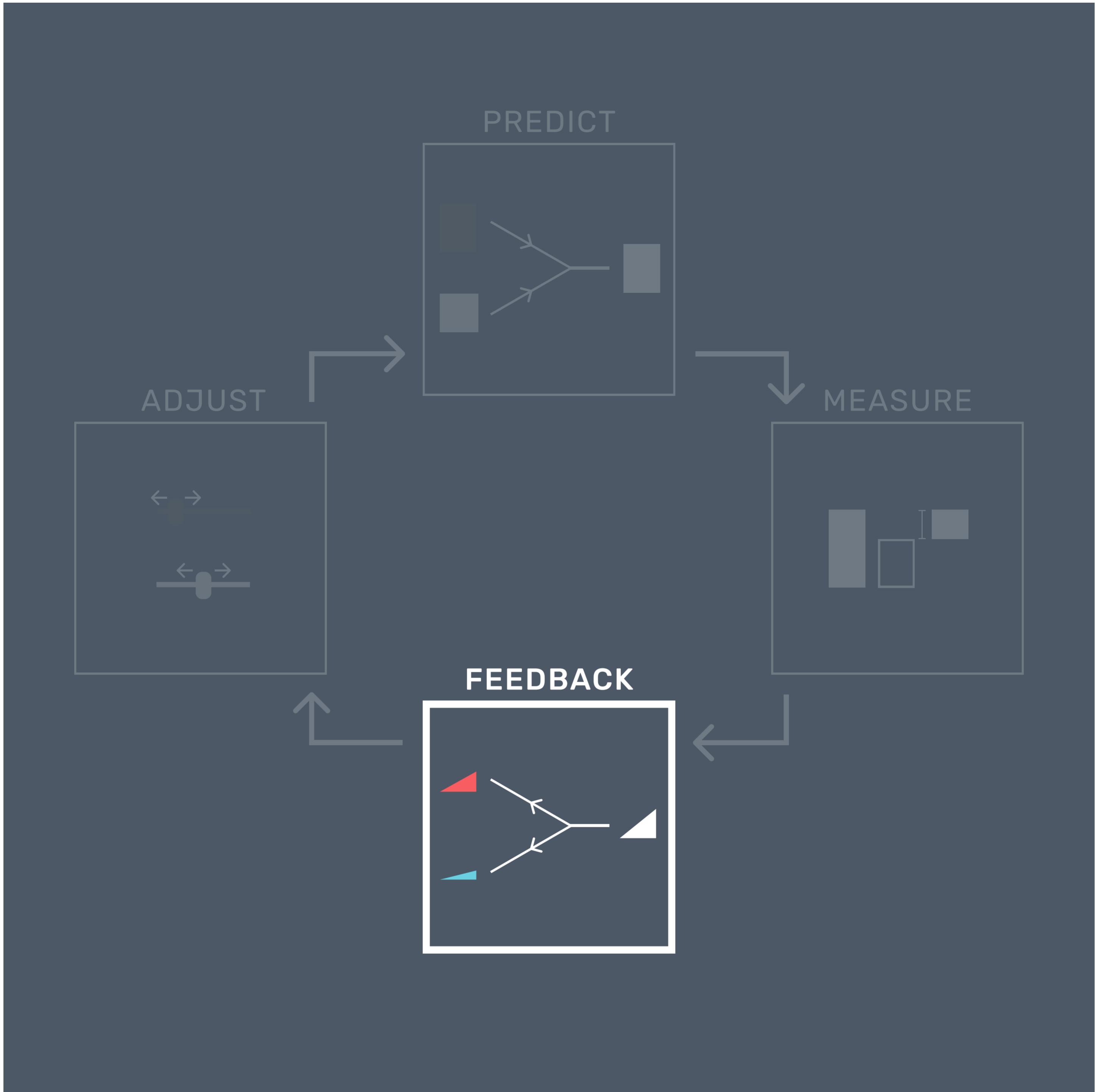
## LOSS FUNCTION



## WEIGHT VS. LOSS

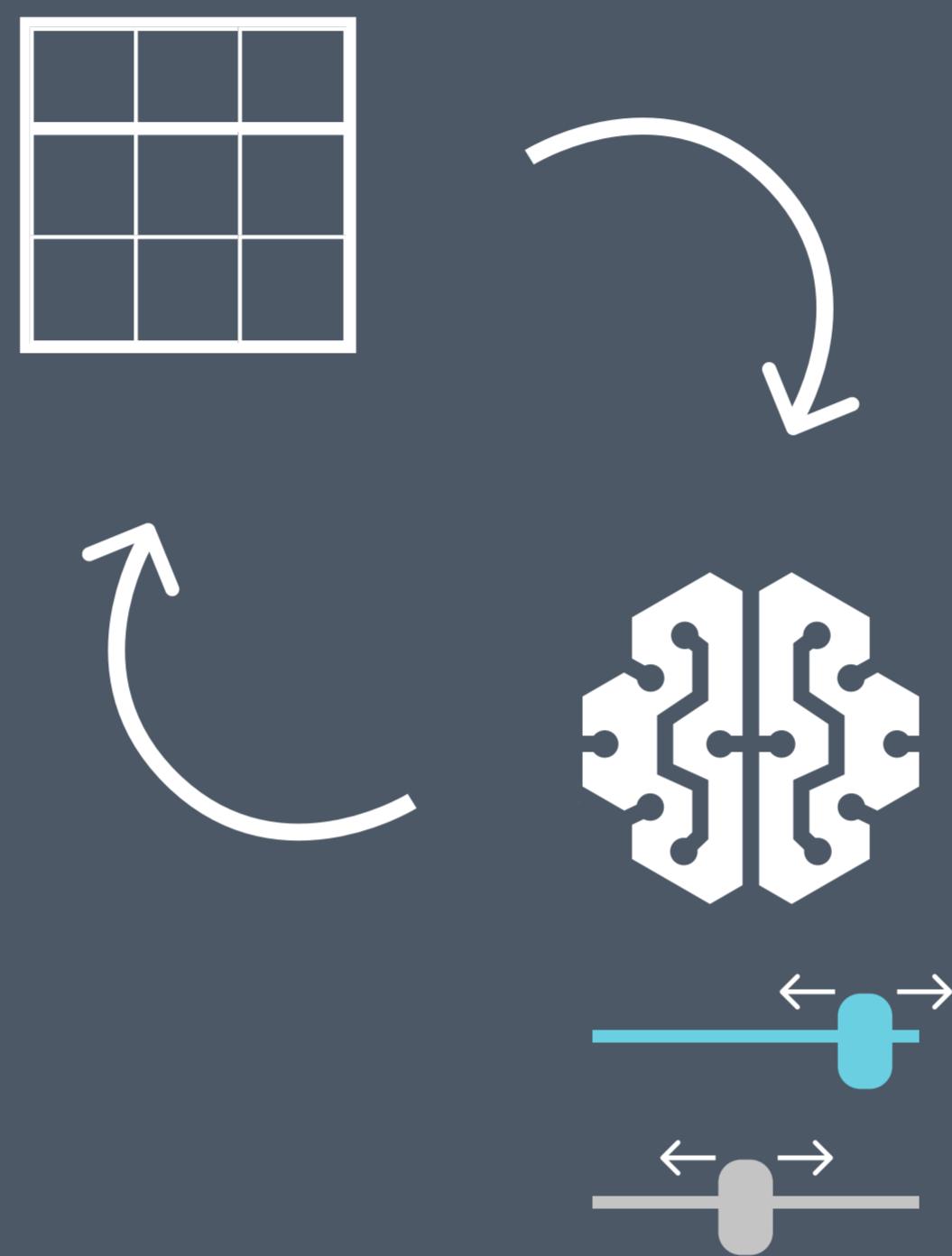
Recall that a neural network learns by adjusting its parameters - weights and biases. Let's first focus on weights since this is where most of the learning takes place. We'll come back to biases later.

We want to find out how changing the weights affects the loss.



## FEEDBACK

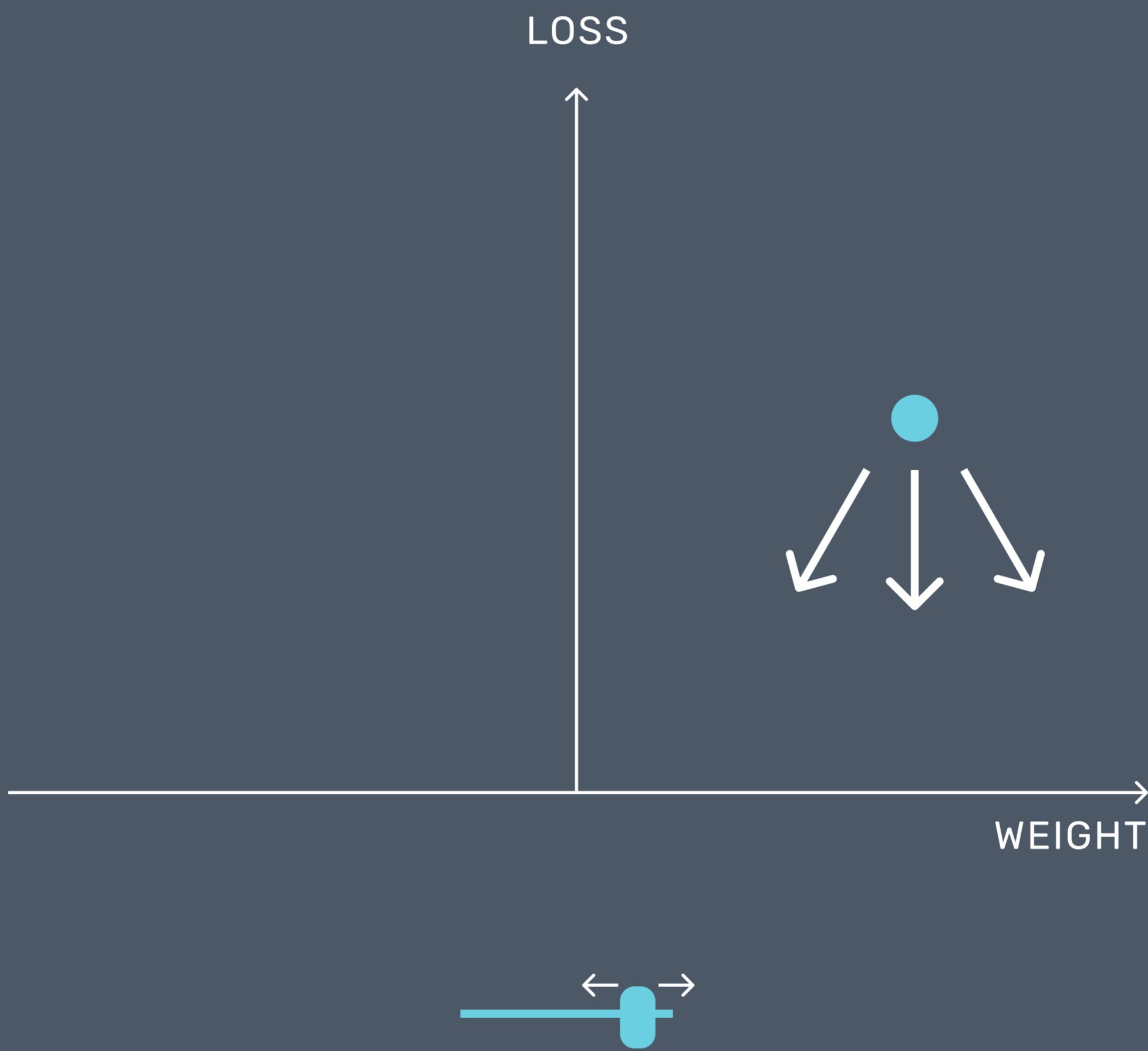
At this point, the neuron hasn't learned anything yet. And learning is exactly what is going to happen in the third step, *feedback*.



## LEARNING

We have established that the neuron's goal is to minimize the training loss by adjusting its parameters.

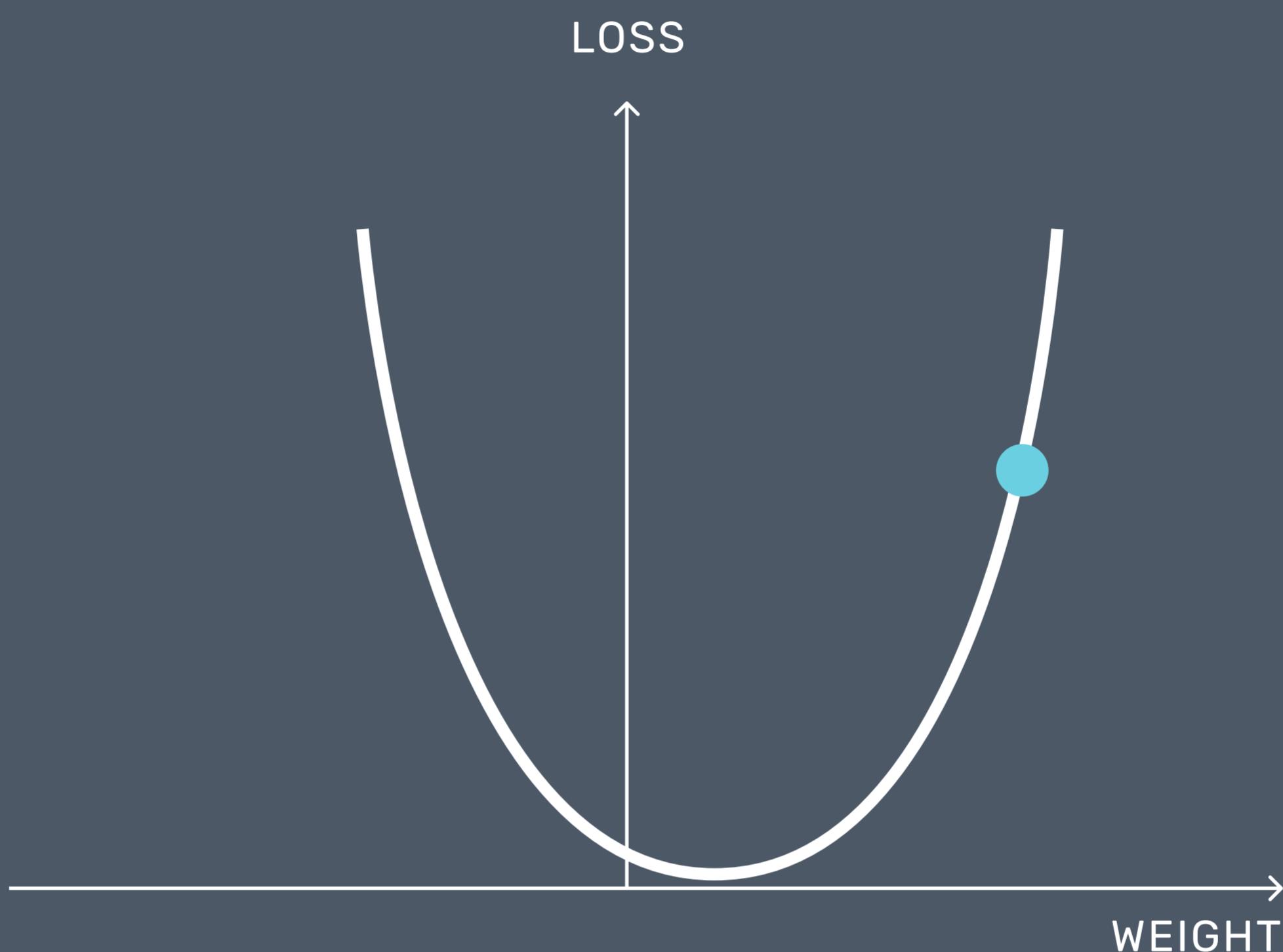
This is the essence of learning in a neural network. Let's find out how this works.



## MINIMIZING LOSS

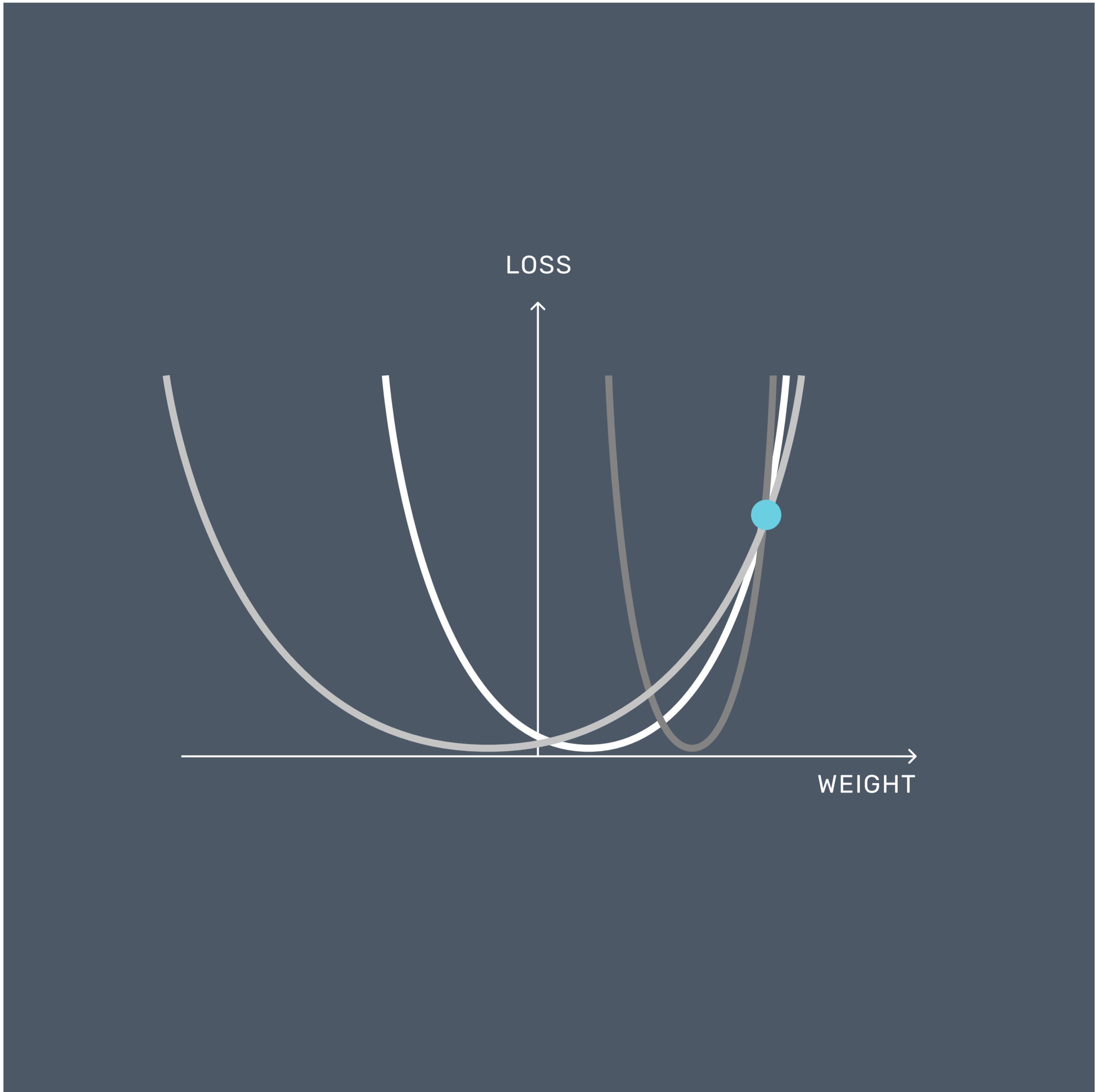
Let's start with one training cycle and plot the loss (i.e. MSE) on a chart.

Now, we want to bring this MSE down to be as close to zero as possible. What we need is to find the weight value that gets us there. But how do we do this?



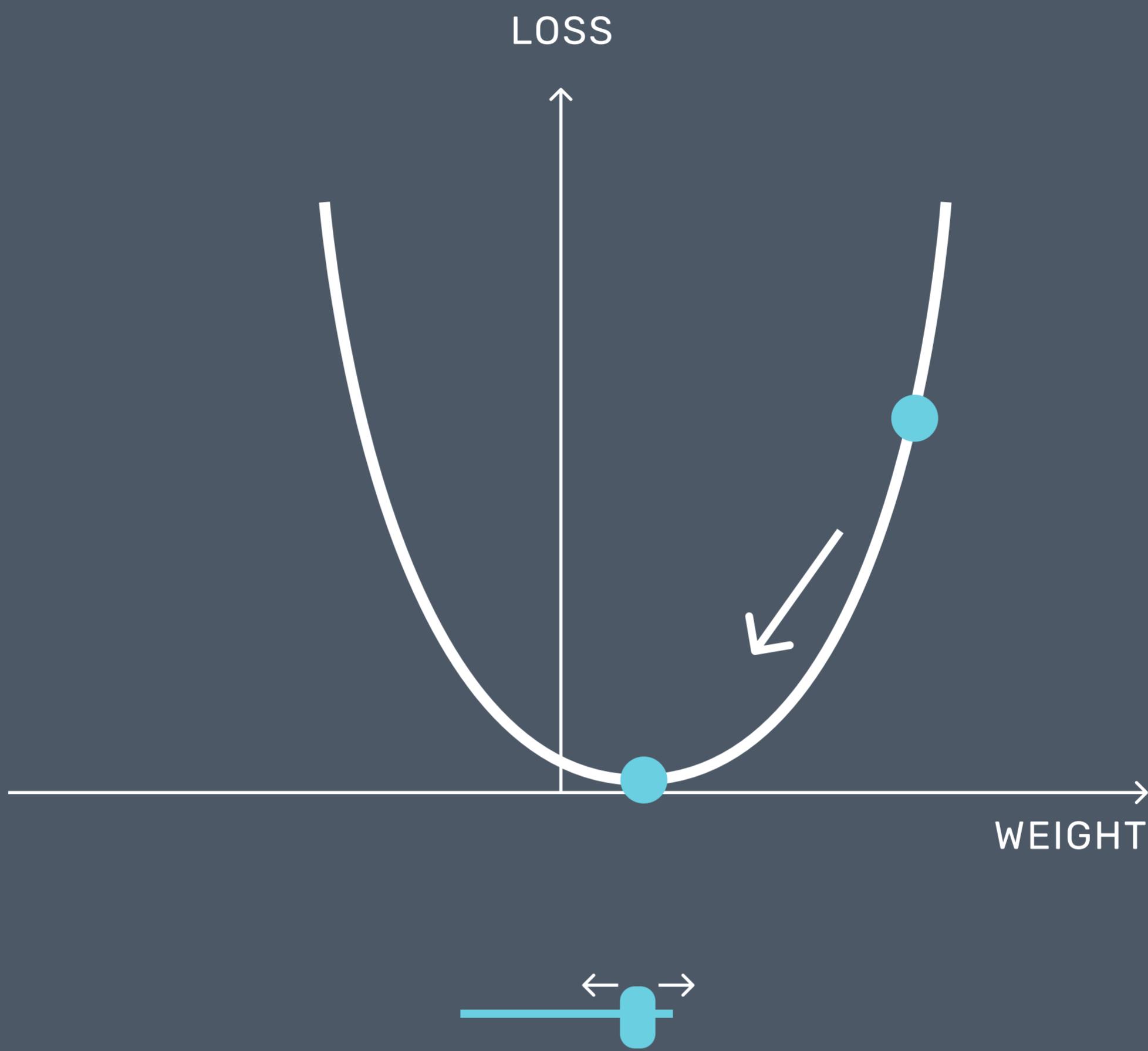
## THE LOSS CURVE

As a loss function, the MSE gives us a very desirable property. If we tried plotting all possible weight values and the corresponding MSEs, we'd get a U-shaped curve. This comes from the squaring effect of the MSE.



## MINIMUM POINT

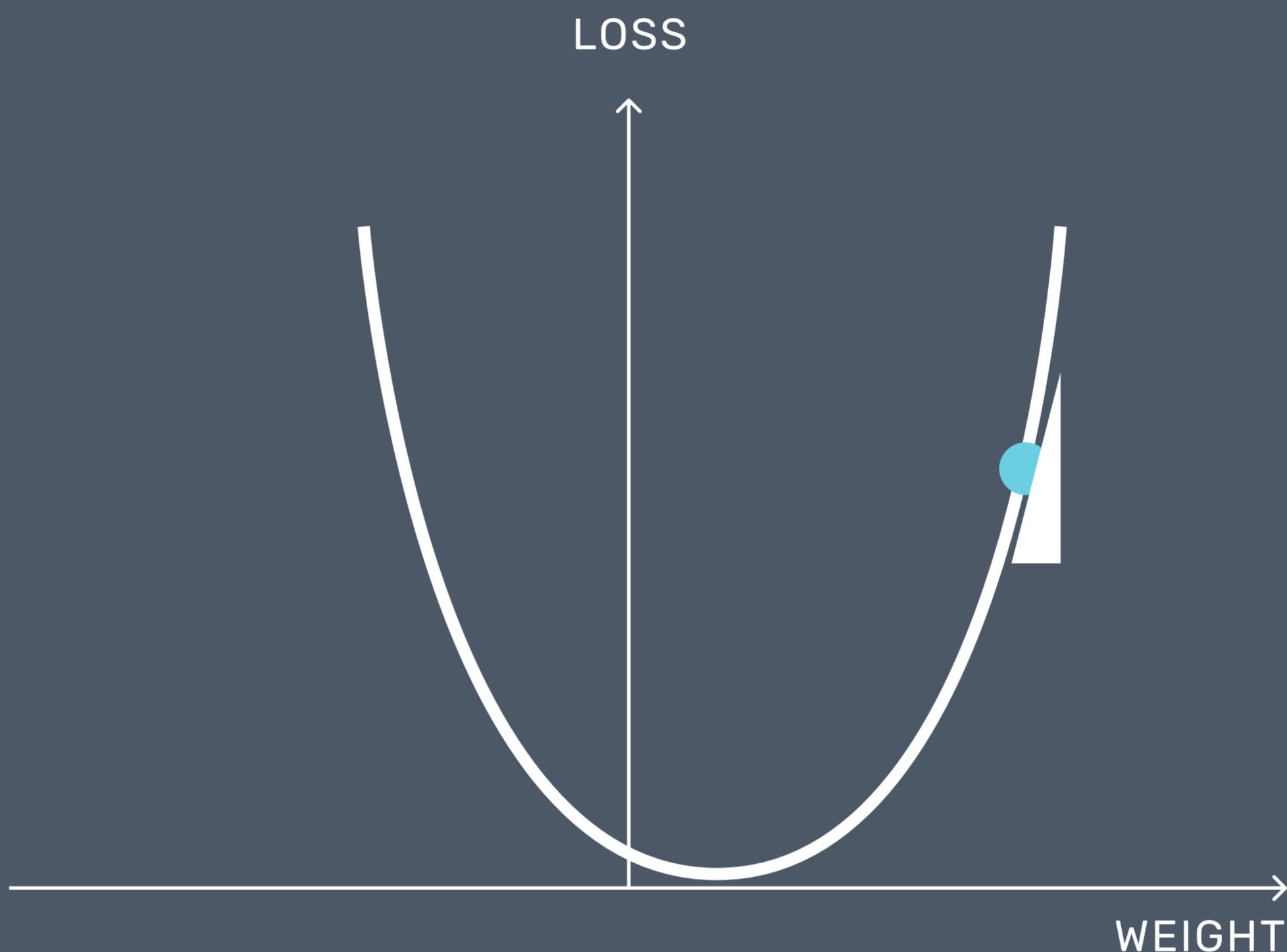
Its width and position may vary, but its shape will always be the same - there is a single point where the curve reaches its minimum. And that is what we are after!



## GOAL

And that is our goal - to get the neuron to find the weight value that will bring the MSE to its minimum.

In practice, we won't be able to get exactly to the lowest point. But we can get very close, and that's good enough for most tasks.

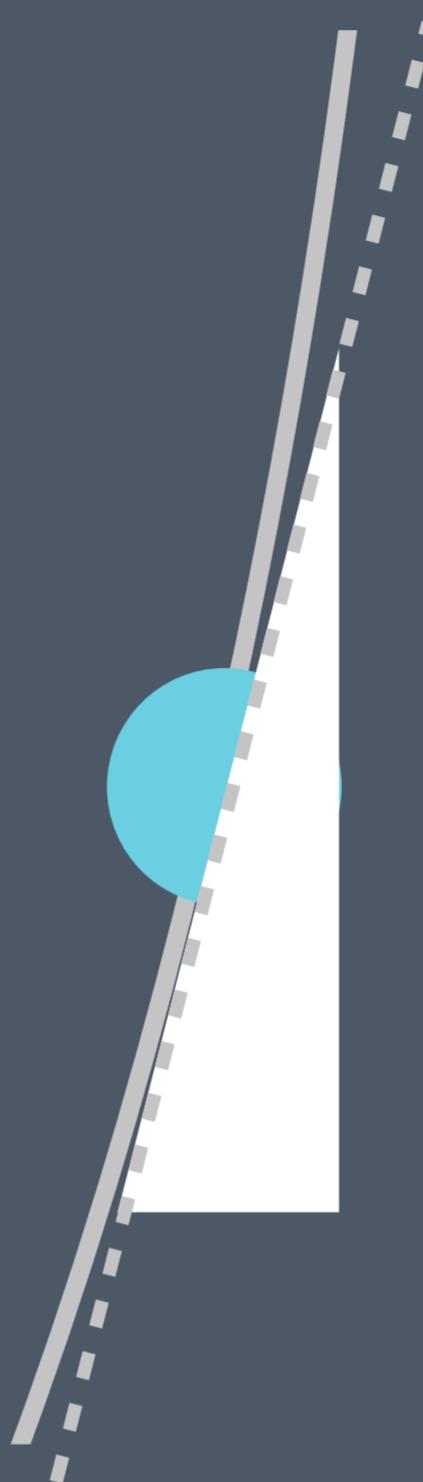


## GRADIENT

The next question then is, how does the neuron know by how much to adjust its weight? The answer is to find the *weight gradient*.

A gradient is the derivative of an output with respect to its input. In our case, the output is the loss function, while the input is the weight. In Chapter 2, we'll find gradients of the loss function with respect to other types of inputs, so it's worth keeping this in mind.

We won't go deep into the math, but let's understand why this is useful for the neural network.



## STEEPNESS

The gradient is a measure of the steepness of the curve of the loss function. And where we are now, it's very steep. The steeper the curve, the greater the gradient.

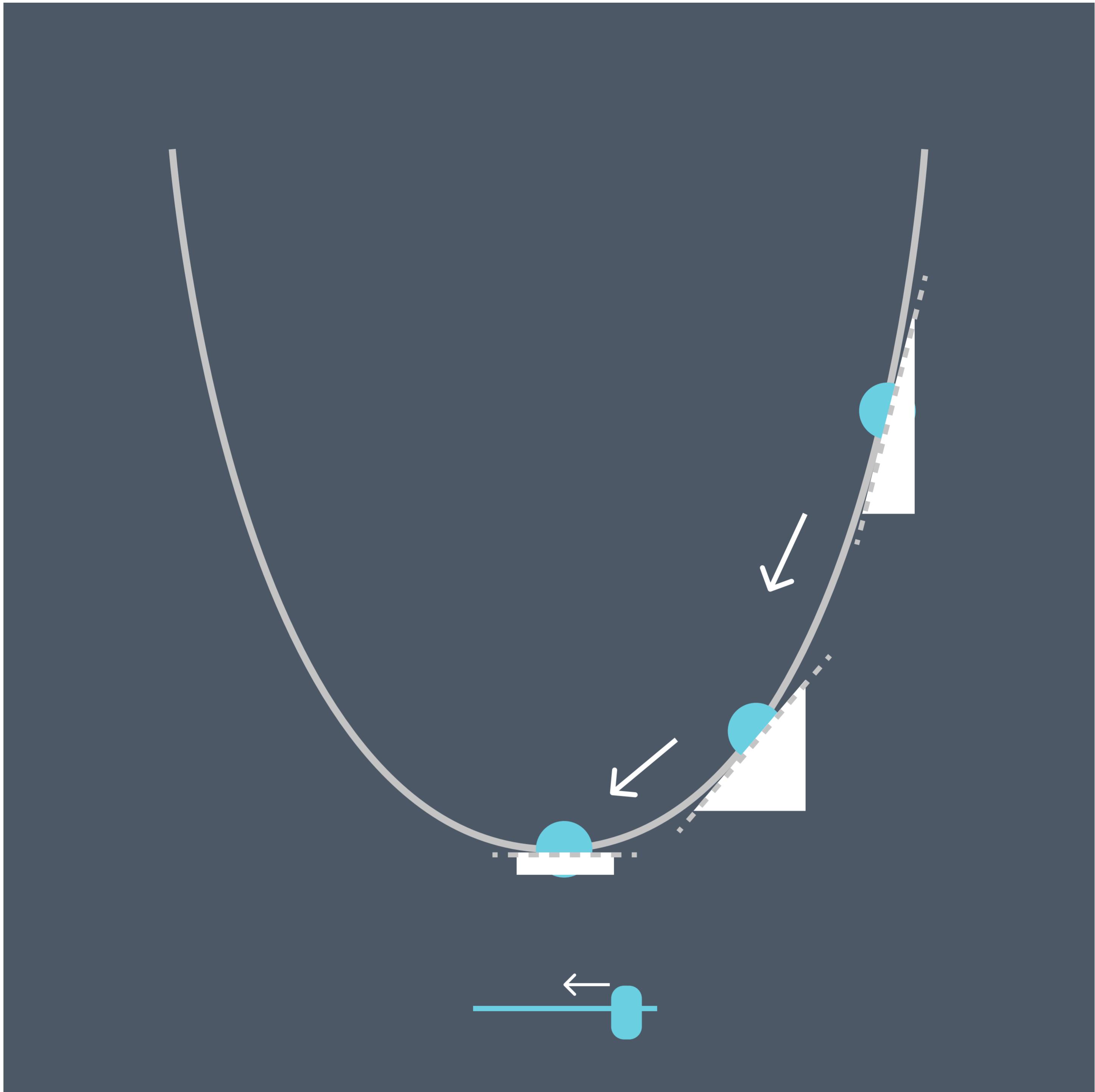
A large gradient indicates that the weight is still far from the optimal value, so we'll need to adjust it by some amount.



## MINIMUM GRADIENT

But why is this so? To better understand, let's pretend that we've succeeded in finding the ideal weight value that brings the loss to the bottom of the curve.

The gradient here is zero, which means that we no longer need to adjust the weight.

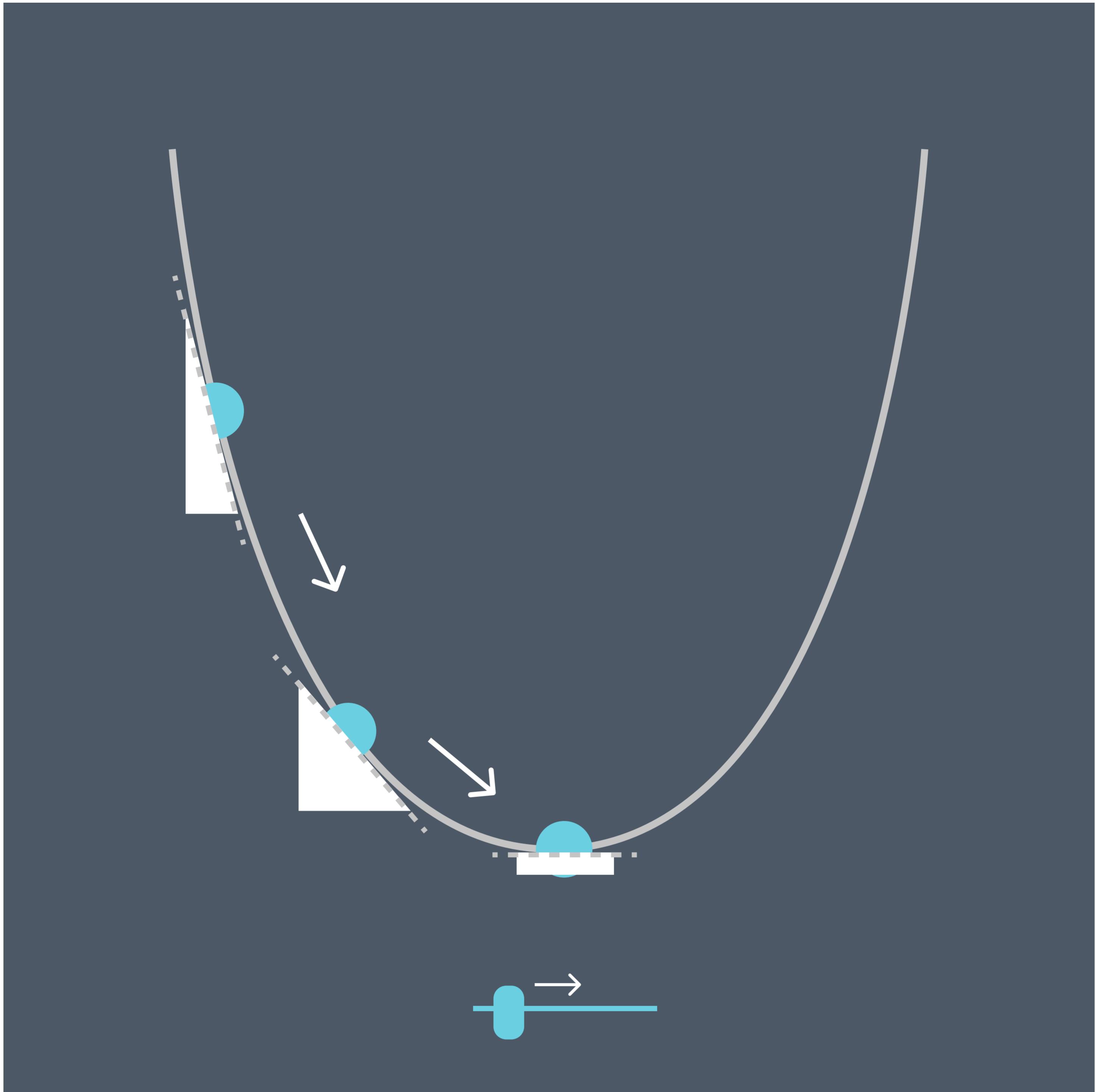


## MAGNITUDE

Notice that as we decrease the weight from its initial position to the bottom of the curve, the gradient continues to decrease until it reaches zero.

This is the first property of the weight gradient - its *magnitude*.

The magnitude of the gradient informs the neuron how far its prediction is from the actual. And by the same token, it also informs how much the neuron needs to adjust its weights.



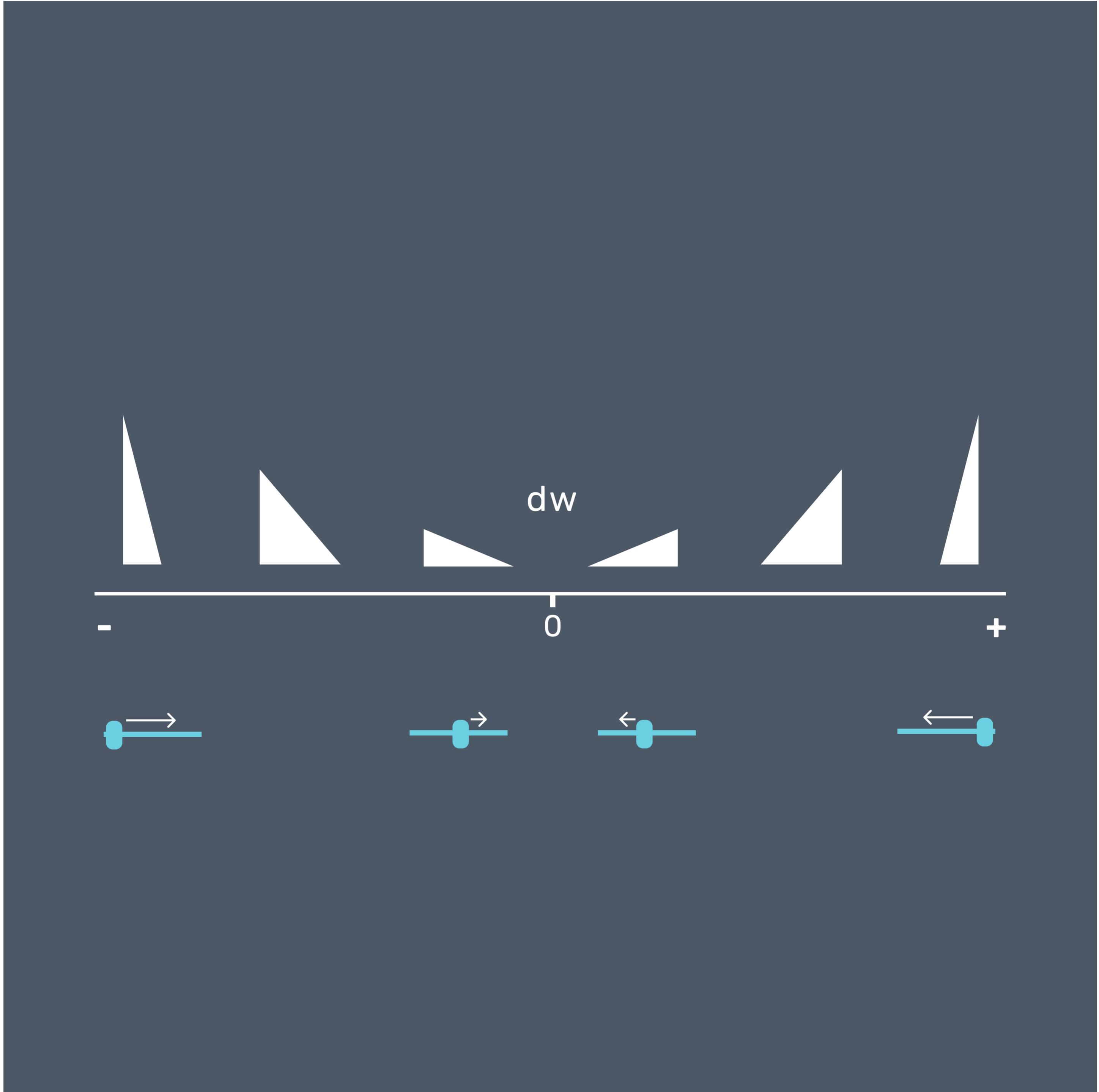
## DIRECTION

The second property of the weight gradient is its *direction*.

Suppose the starting weight is on the other side of the curve. This causes the sign of the gradient to become negative.

This indicates that the gradient is too small rather than high. Instead of decreasing, we'll need to increase the weight to reach the bottom of the curve.

Therefore, the sign of the weight gradient informs the neuron about the direction of weight adjustment.



## GRADIENT DESCENT

The magnitude and direction of the weight gradient are the two types of feedback returned to the network.

As the network goes through multiple training cycles, we want the weight to move down the curve toward its minimum point.

For this reason, this method is called *gradient descent*.

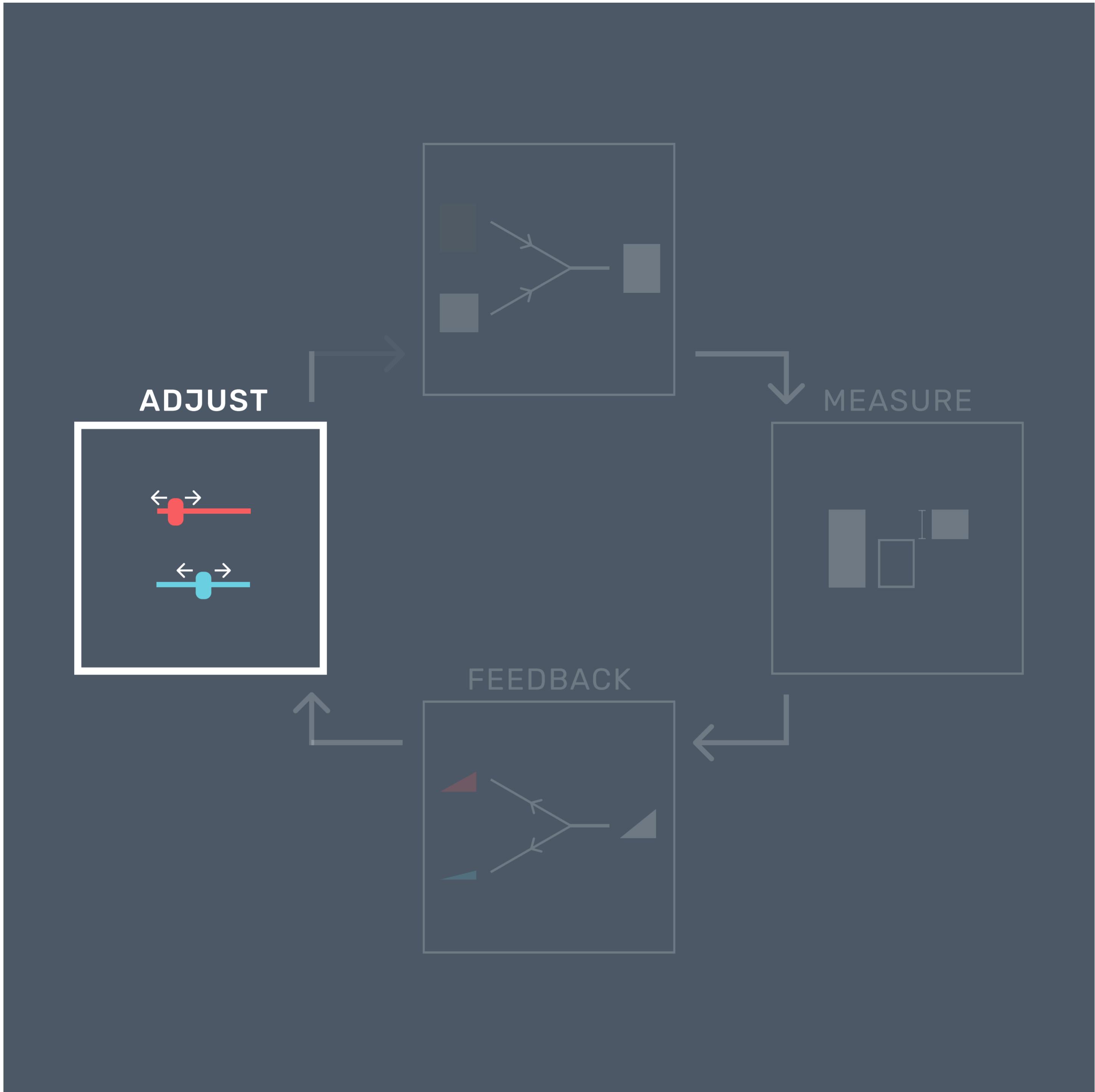
$$dw = \text{input} * \text{error}$$



## WEIGHT GRADIENT

We've seen that the weight gradient is the derivative of the loss function with respect to the weight. We won't go into the mathematical proof, but the result is the input value multiplied by the error. We'll represent the weight gradient as  $dw$  for short.

The final gradient to be passed back to the network is the average gradient from all the training data points.



## ADJUST

We have now reached the fourth and final step of the training cycle, *adjust*. In this step, the neuron will adjust its weight according to the gradient that it receives.

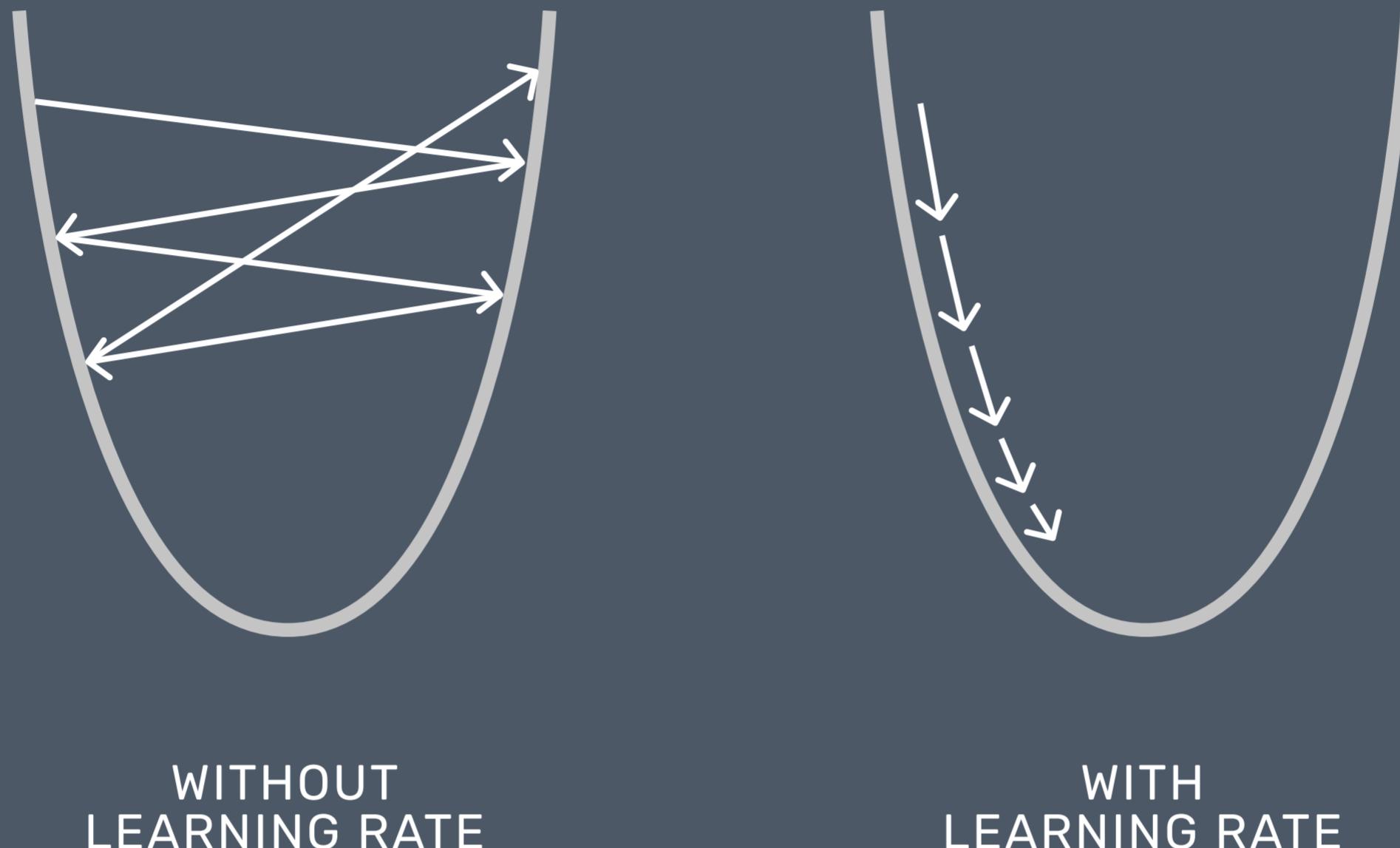
$$W_{\text{new}} = W_{\text{previous}} - \alpha * dw$$



## LEARNING RATE

If the gradient is a positive value, the previous weight is reduced commensurate to the magnitude. On the other hand, if the gradient is negative, the previous weight is increased.

Here we introduce another term called the *learning rate*, represented by  $\alpha$  for short. It is a value multiplied by the gradient before making the weight adjustment.

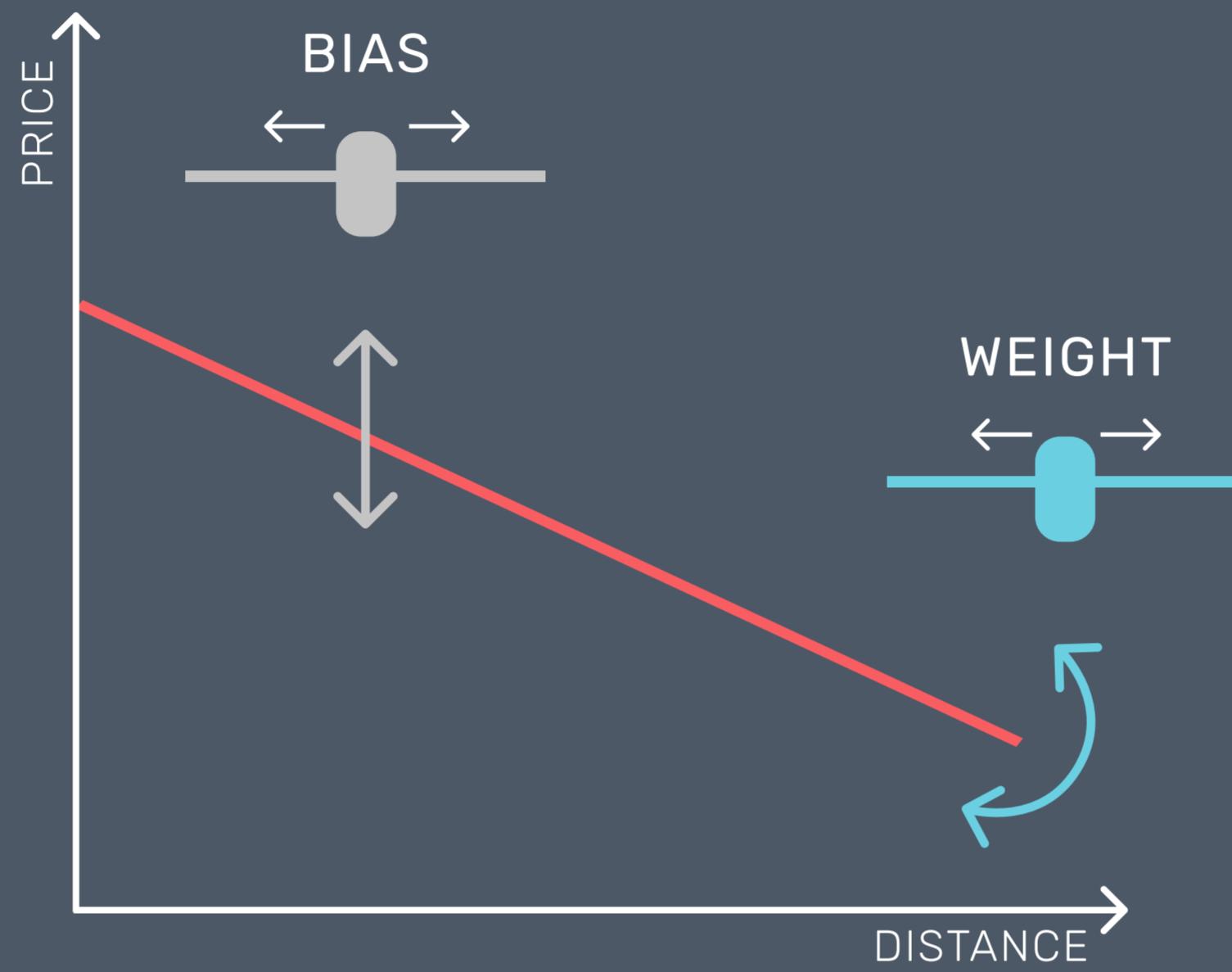


## THE ROLE OF LEARNING RATE

We want the gradient descent process to be a smooth one. And to ensure that, we need to apply the learning rate. It is usually a very small number that scales the gradient down. In our task, we'll use 0.08.

Without the learning rate, the descent may become unstable, or worse, never reach the bottom of the curve.

Choosing the right values is an art in and of itself. Too small and the neuron learns too slowly. Too high and the neuron never finds the minimum point.



## BIAS

Let's now bring the bias parameter into the discussion.

The weight's role is to adjust the shape of the prediction line or curve. Meanwhile, the bias' role is to adjust the position of the function, shifting it up or down.

$$dw = \text{input} * \text{error}$$



$$db = \text{error}$$



## BIAS GRADIENT

We get the bias gradient  $db$  by taking the derivative of the loss function with respect to the bias. Without going into the mathematical proof, the result is the error value itself.

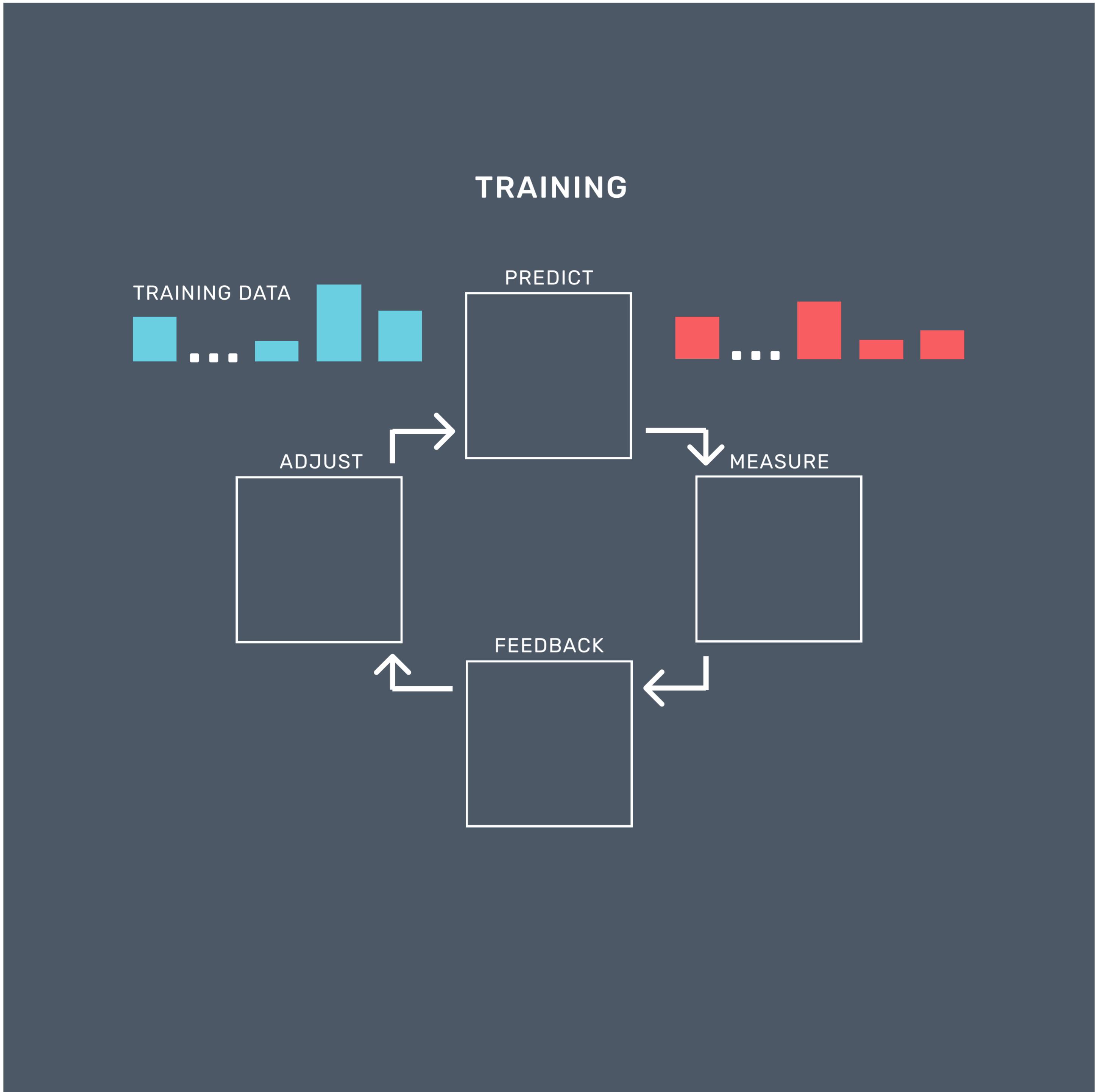
The final gradient to be passed back to the network is the average gradient from all the training data points.

$$b_{\text{new}} = b_{\text{previous}} - \text{alpha} * db$$



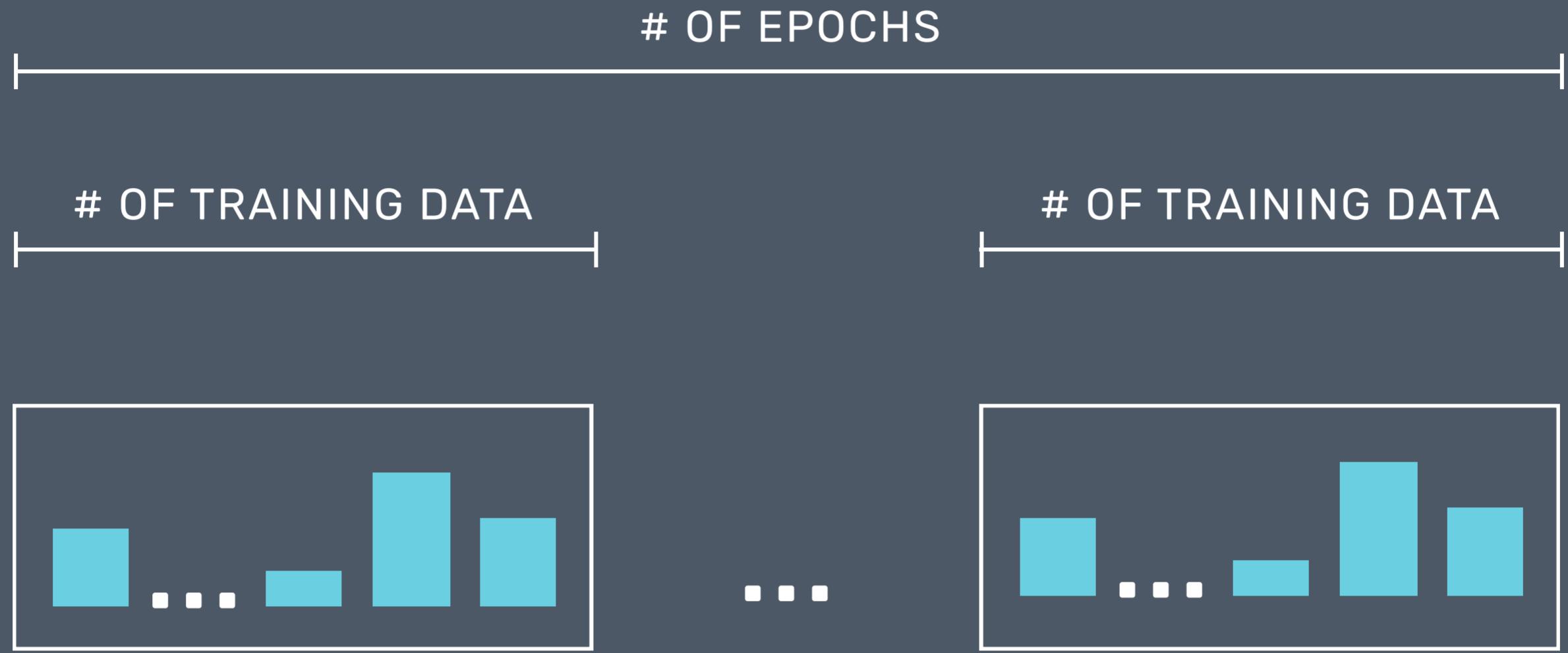
## ADJUST

As in the weight, the bias gradient is multiplied by the learning rate before making the bias adjustment.



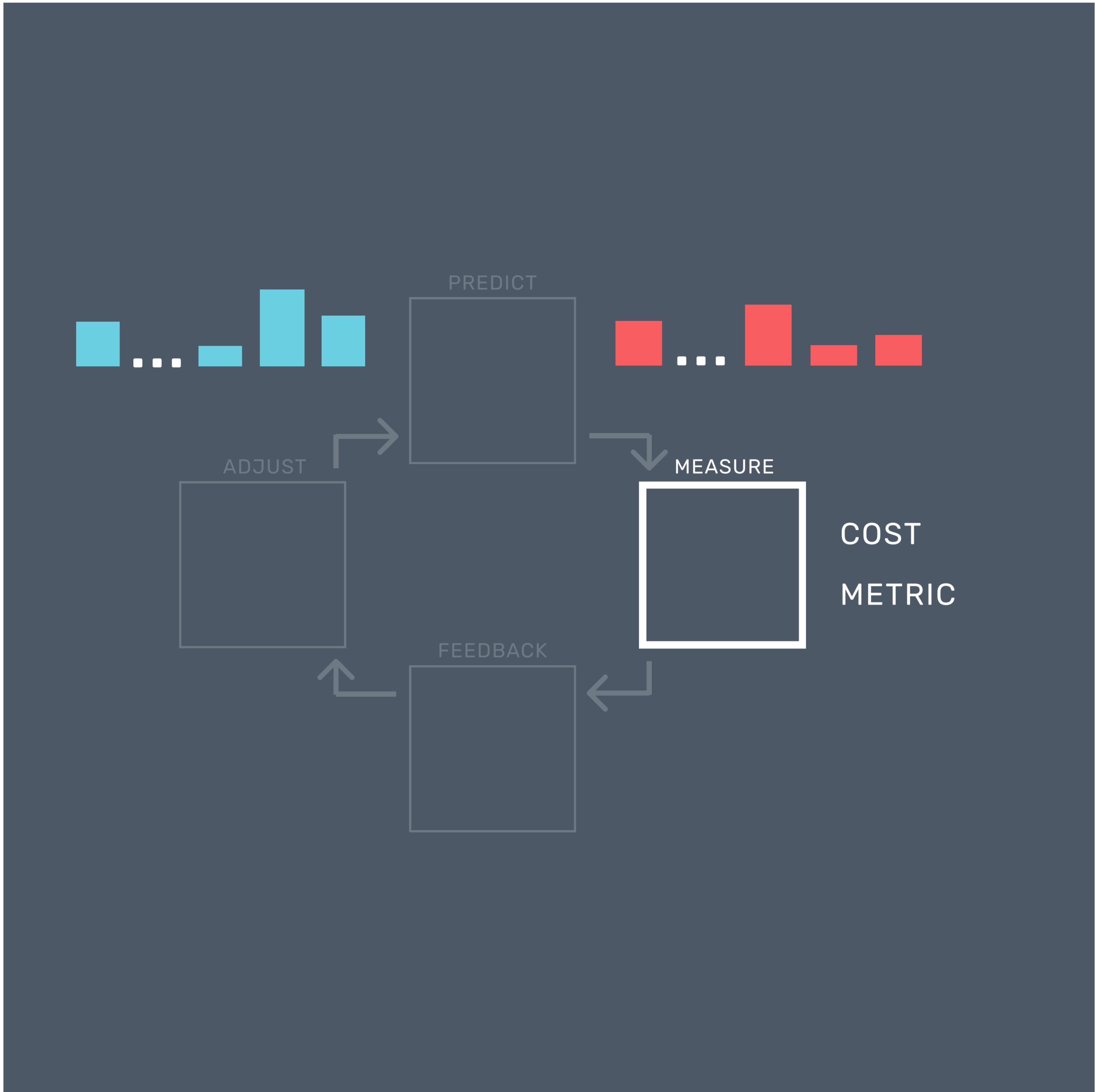
## EPOCH

This completes one round of training, also called an *epoch*.



## COMPLETE TRAINING

We'll repeat the four steps for 100 epochs. And once we've gone through all the epochs, training will be complete!

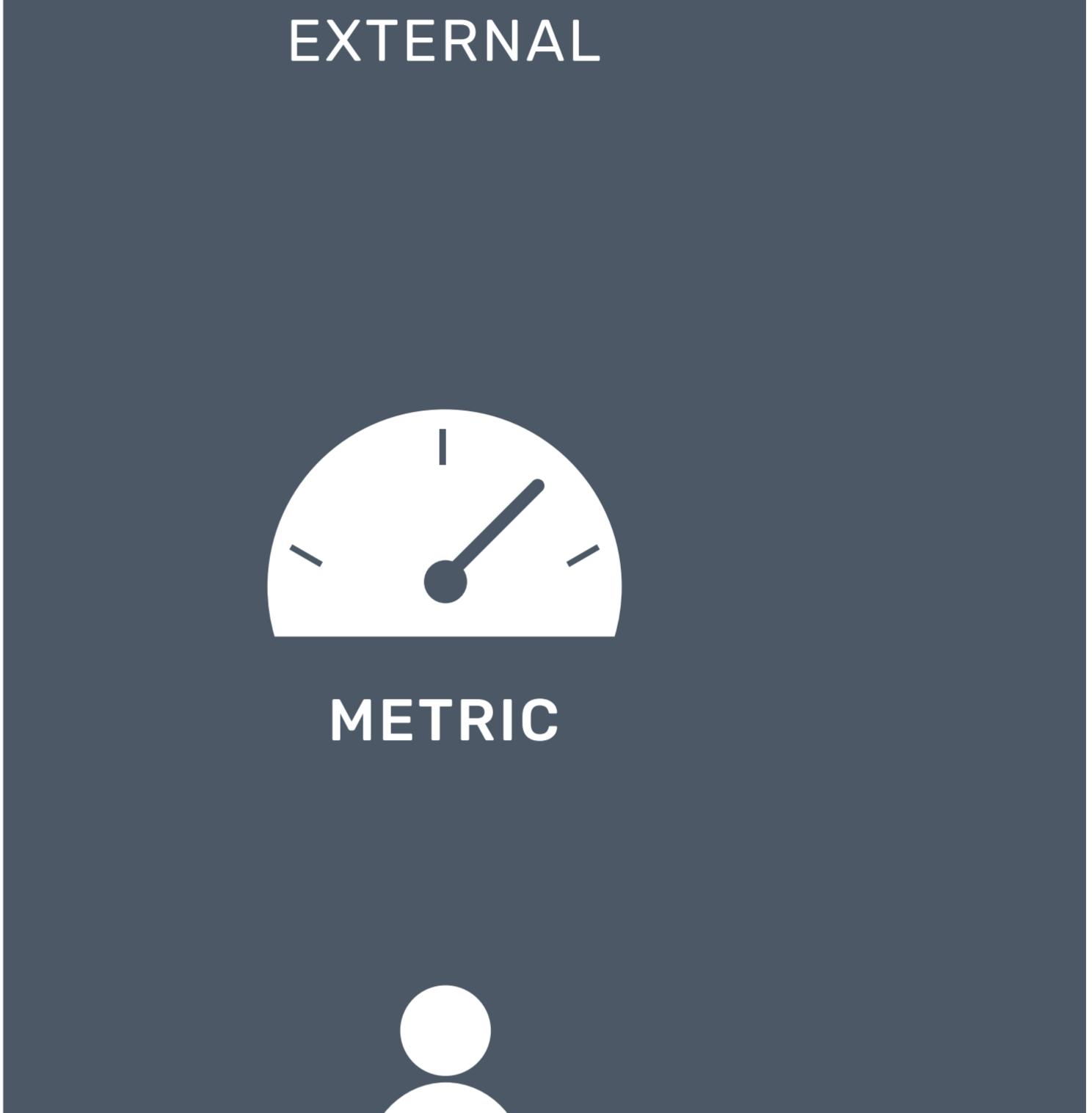


## COST AND METRIC

During the *measure* step of the cycle, we actually measure two things after each epoch.

The first is *cost*. Cost is simply the average loss value over the training data points (i.e. the MSE). So far, we've only used the term *loss* for simplicity, but cost is the more precise term. In practice though, it's not uncommon to see these two terms used interchangeably.

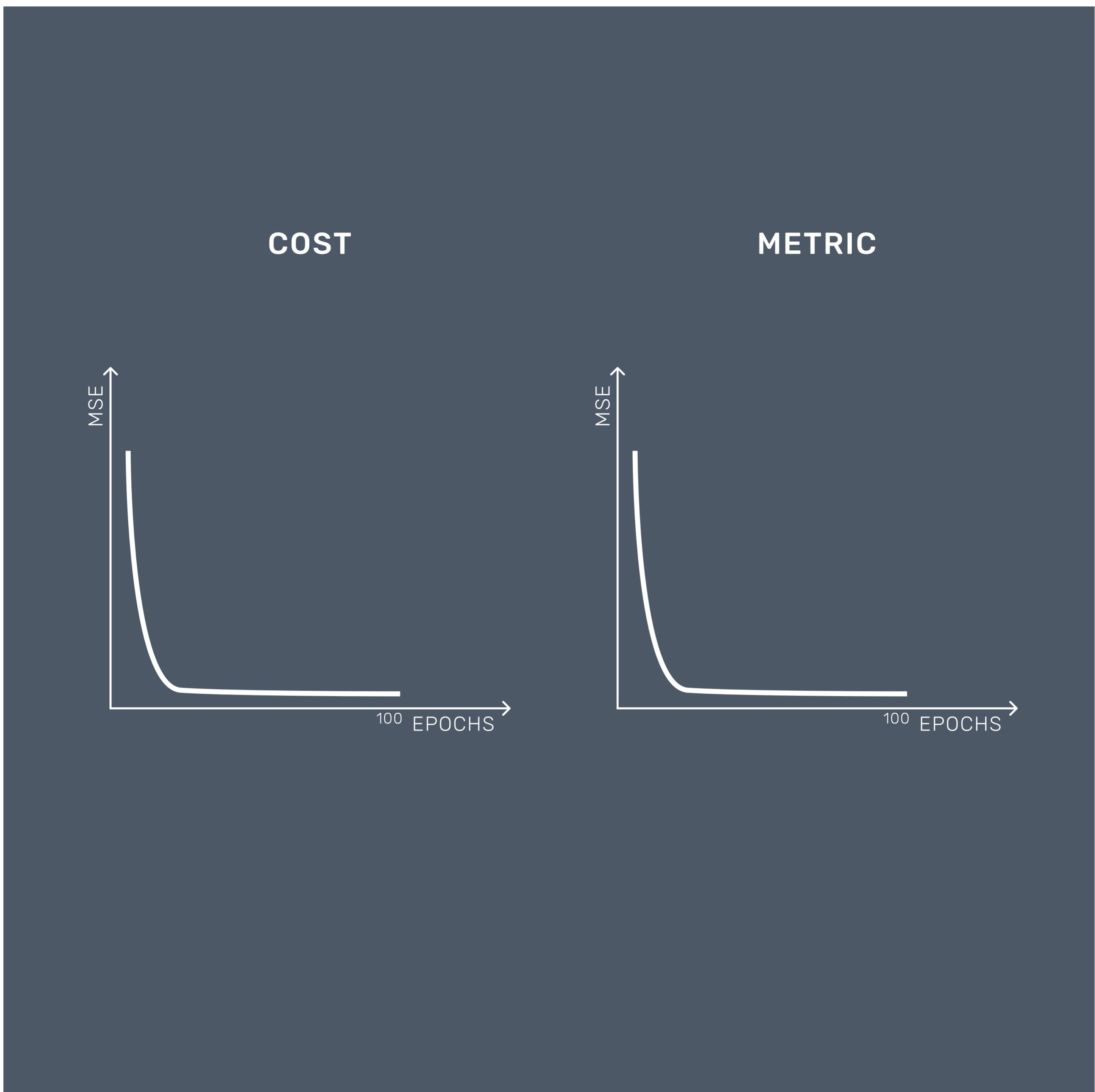
The second is *metric*. For this task, it's also equivalent to the MSE. It may not make sense now why we need two measures for the same thing, but it will become apparent when we get to Chapters 3 and 4, where the cost and metric are different.



## PERFORMANCE MEASURE

In other words, the cost is an internal performance measure - that is, for the algorithm.

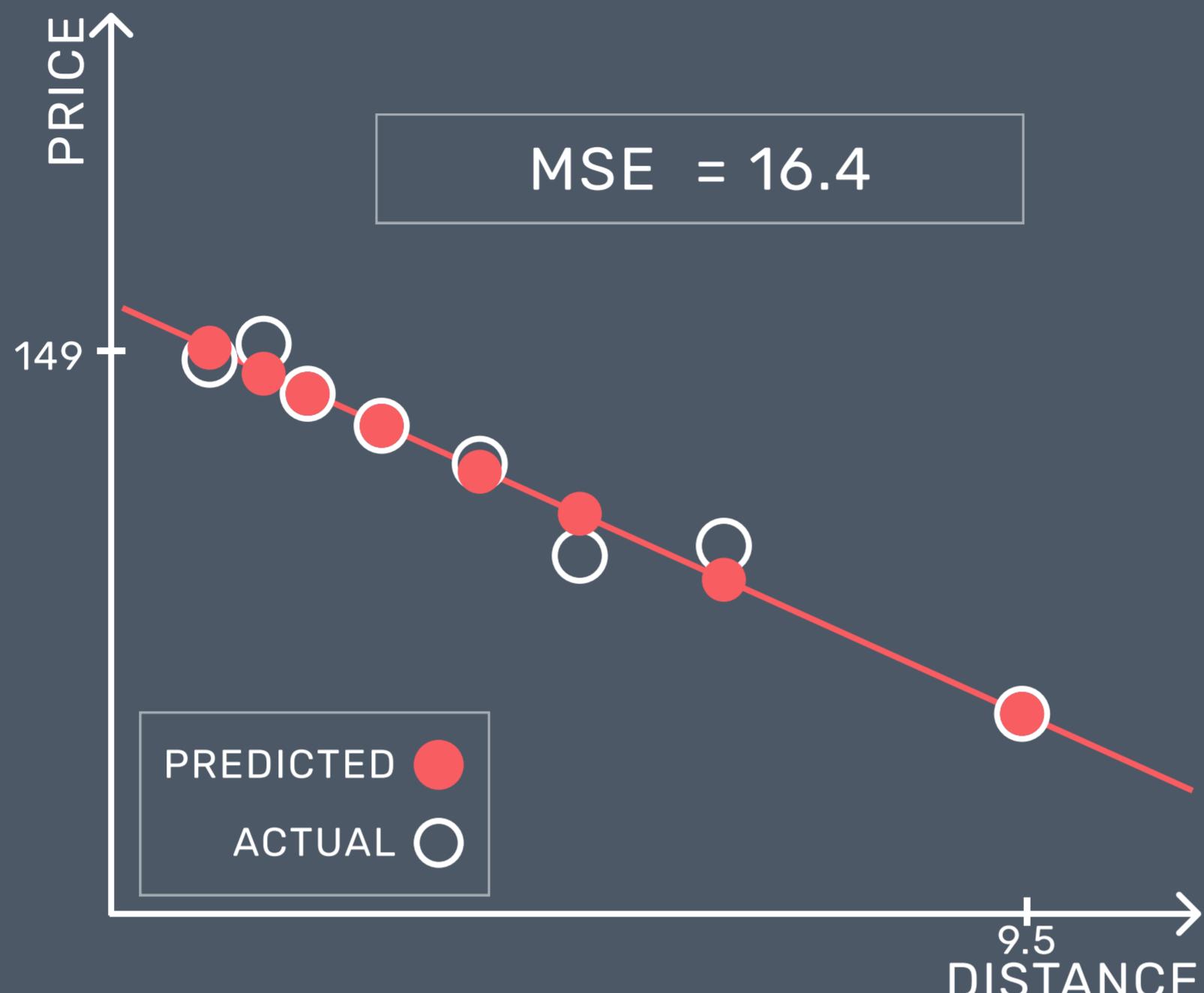
Conversely, the metric is an external performance measure - that is, for the human.



## MONITORING

Over many epochs, we can see that the MSE continues to improve and converge toward zero.

## PREDICTION - TRAINING DATA



## TRAINING PERFORMANCE

At the end of the 100 epochs, we have a trained model that gives a respectable MSE of 16.4.

Plotting the predicted values on a chart gives us a linear regression line that's defined by the learned parameters.

Note that the MSE can only go to zero if the actual training data are perfectly aligned along a straight line. In this case, they aren't.

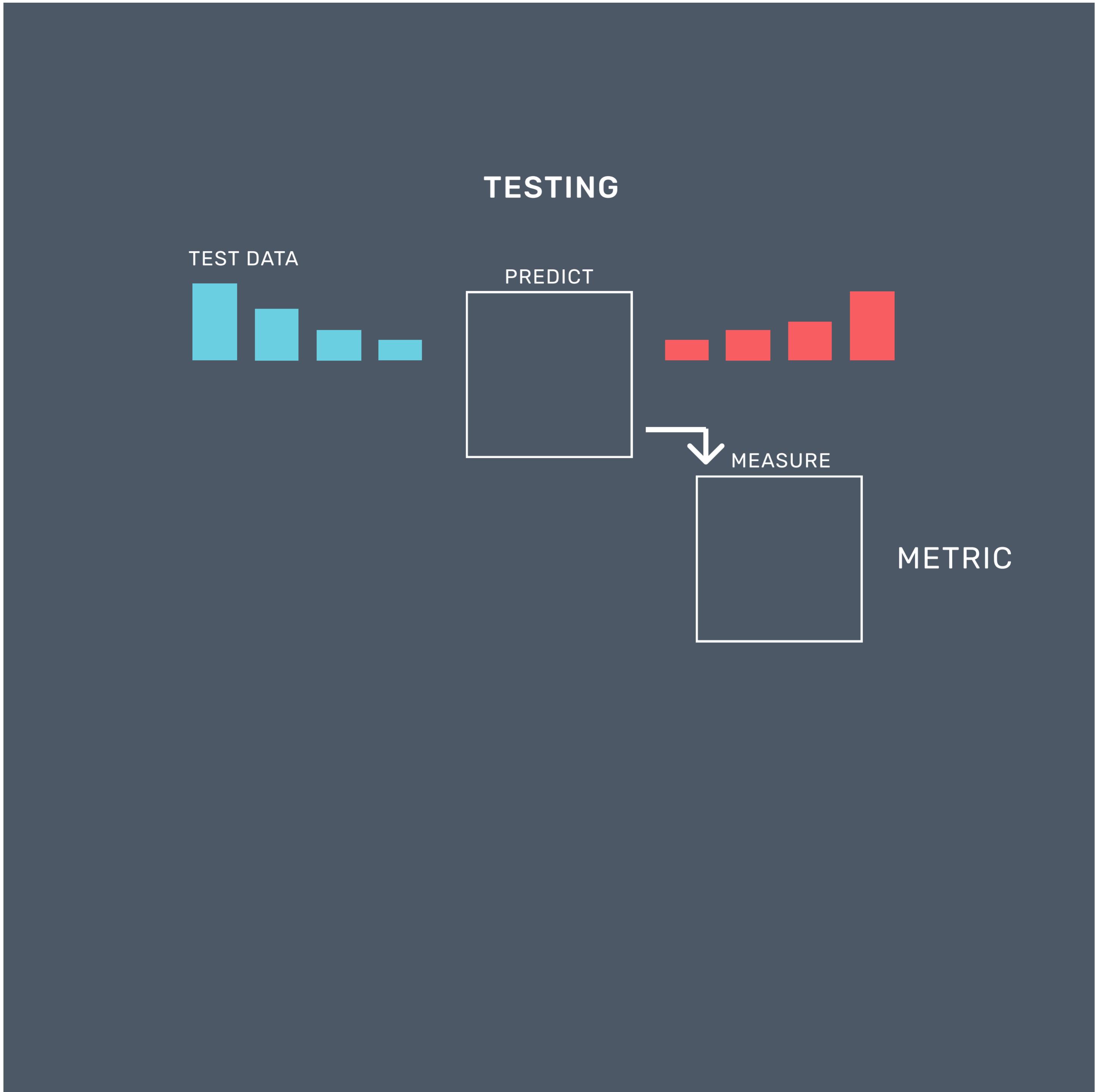
TEST DATA

DISTANCE (MI)	PRICE (\$)
0.5	146.00
1.1	149.00
1.6	140.00
2.4	134.00
3.5	127.00
4.6	110.00
6.2	112.00
9.5	81.00
0.3	156.00
0.7	168.00
4.9	116.00
8.5	99.00

## TESTING

Measuring the performance of a model based on training data is a good indicator of success, but it is far from the true measure. The reason is that we are measuring its performance based on the data it has seen.

We need to measure its performance based on data it has never seen. For this, we use the four test data points that we set aside.



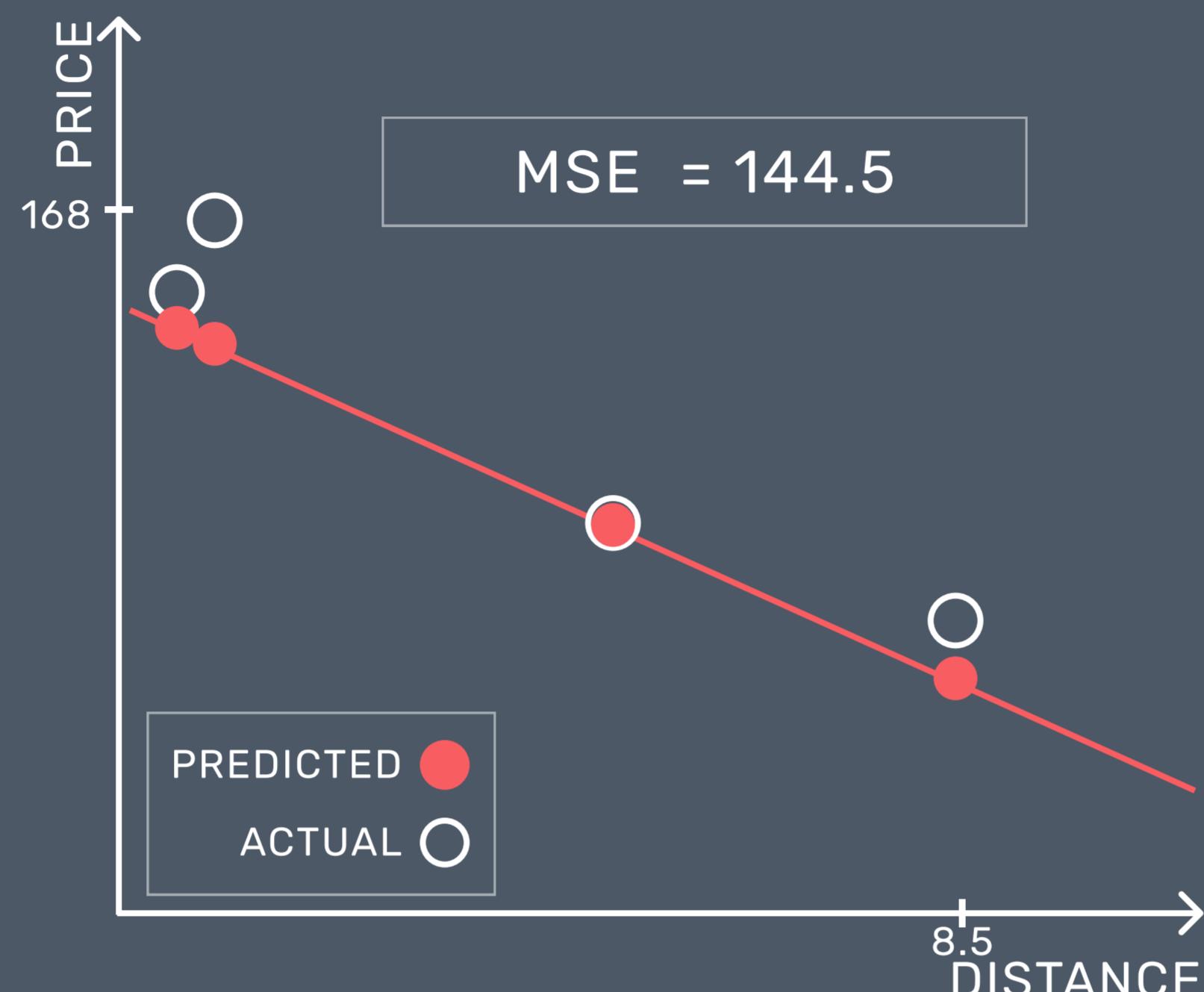
## TESTING

For the test data, we don't need to go through all four steps of the cycle. We only need the *predict* and *measure* steps.

In *predict*, we pass through the features (distance) through the neural network and get the prediction (price) at the other end.

In *measure*, we compute the metric (the MSE) of the prediction. The cost is internal to the model and it's used only during training, so we won't need to consider that.

## PREDICTION - TEST DATA

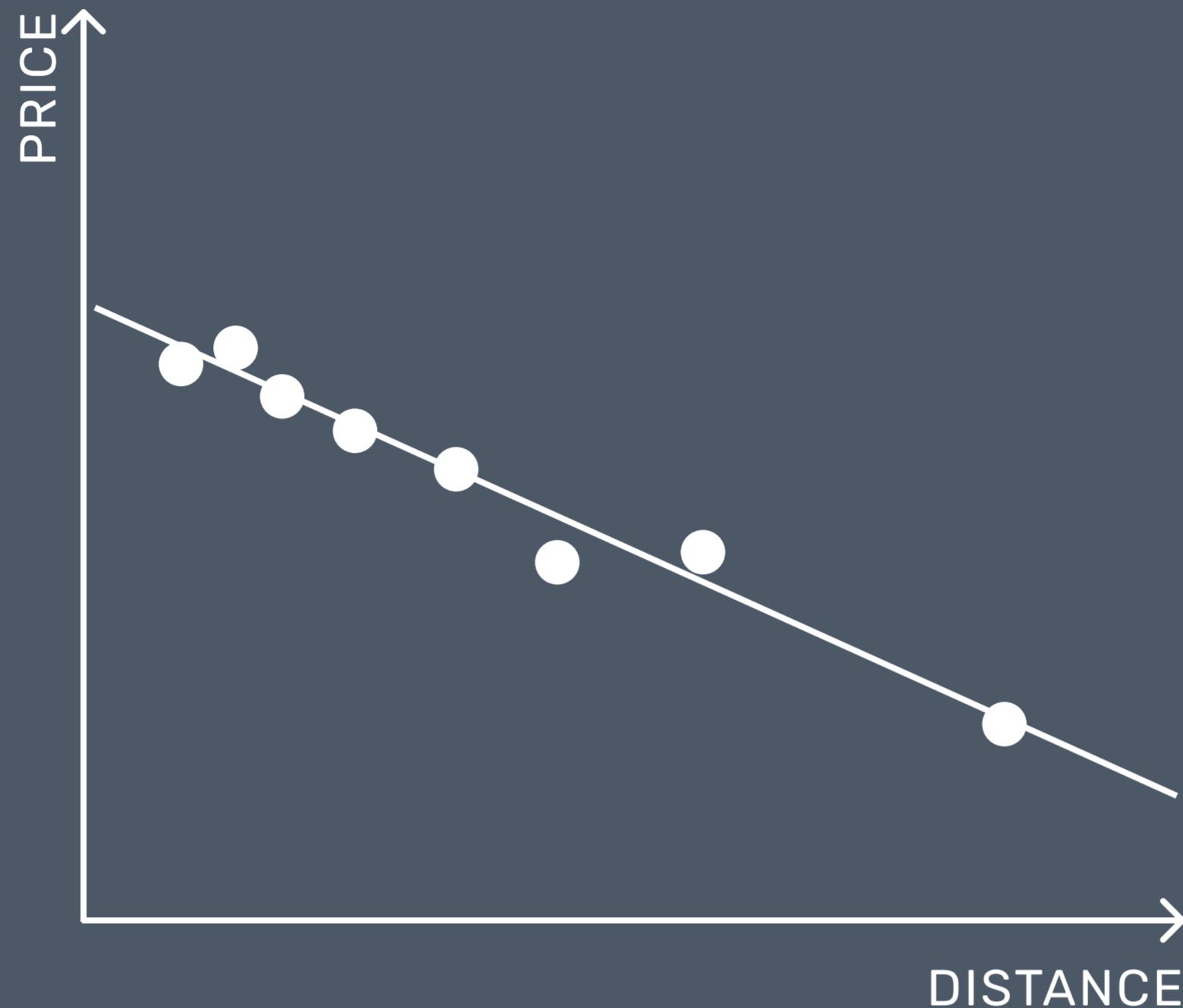


## TEST PERFORMANCE

We get an MSE that is substantially worse than that of the training data. Examining this visually, this is because the test data are more sporadic compared to the training data. This indicates a slightly different distribution of the test data compared to the training data.

However, for this amount of data, we can't really confirm that, and this performance is fine.

In general, the performance with the test data will inevitably not be at the same level as with the training data. But our job is to get them as close as possible.



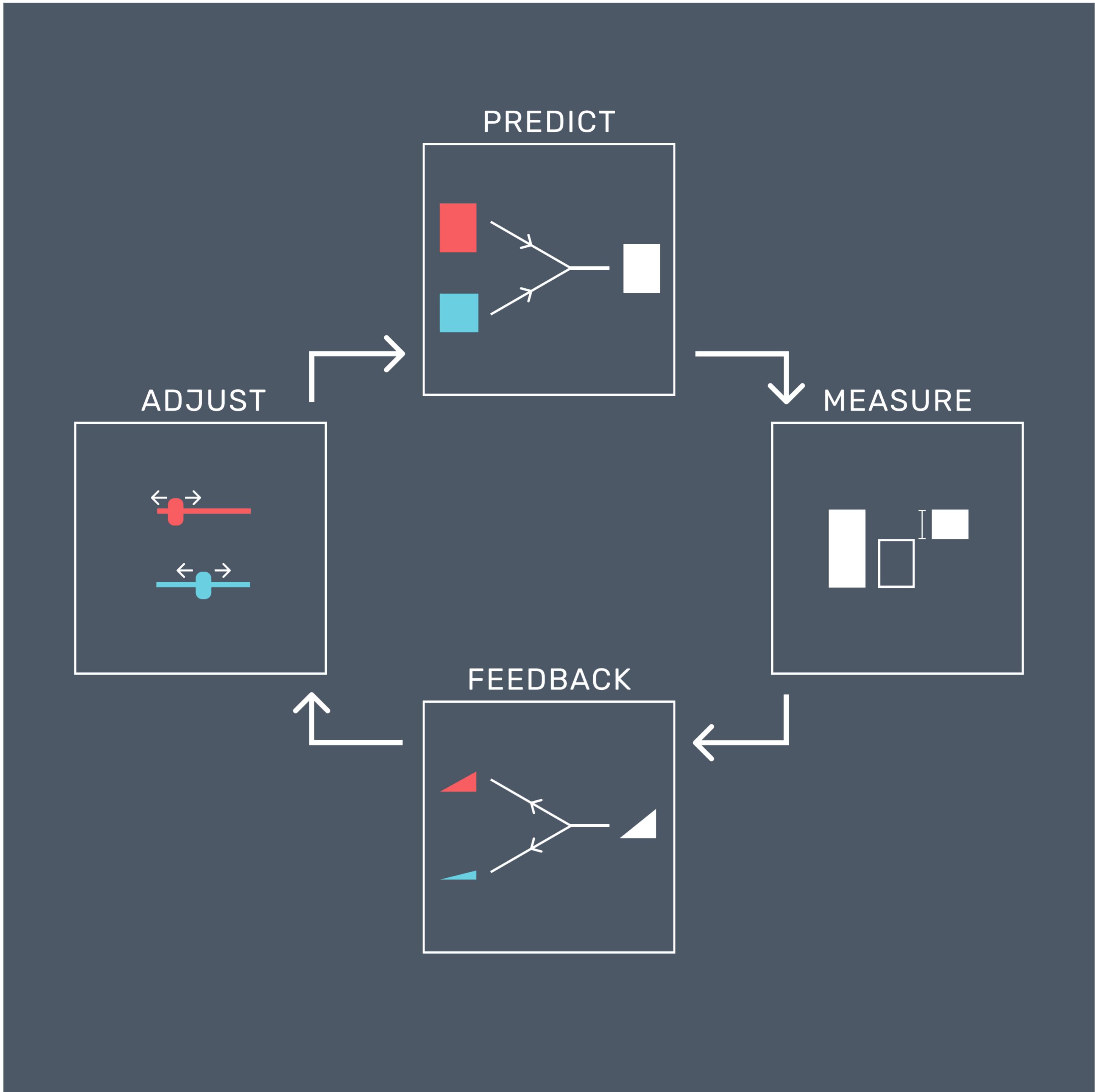
## LINEAR REGRESSION

And that's the end of our first example. If you're familiar with linear regression, you might be wondering why we're taking all the trouble to use a neural network to build a linear regression model?

You are right! There are other simpler algorithms that would have achieved the same outcome.

But this is where the similarity ends. In the following chapters, we'll start to build complexities and non-linearities, which is when the neural network shines.

This chapter is all about building a foundational understanding of neural networks via a simple example. In the coming chapters, we'll continue to build on it.



## RECAP

Let's do a recap of the four-step training cycle: *Predict - Measure - Feedback - Adjust*.

In *predict*, the dataset is passed through the neuron to generate predictions.

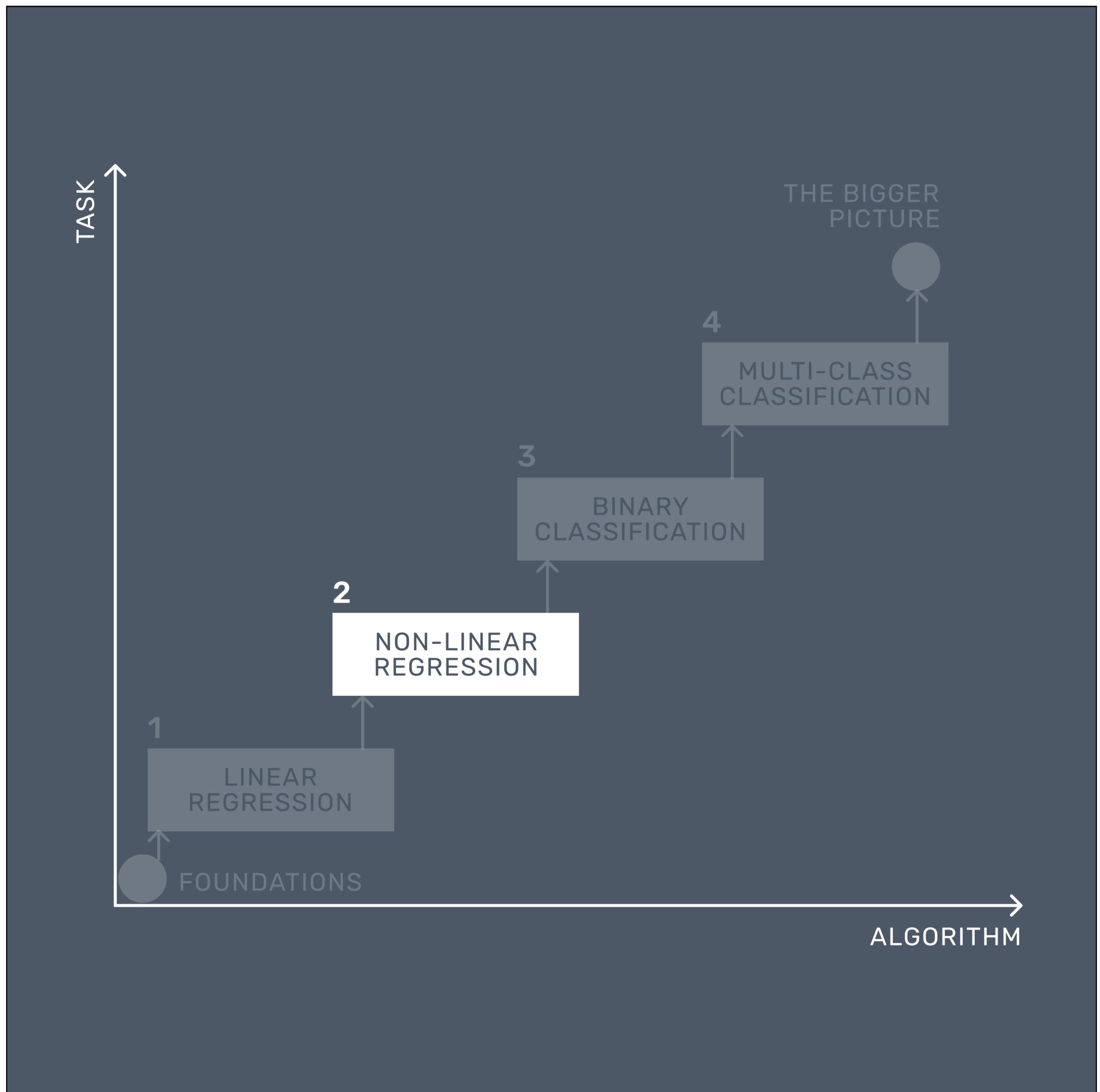
In *measure*, we measure the cost, that is how far are the predictions from the actual values.

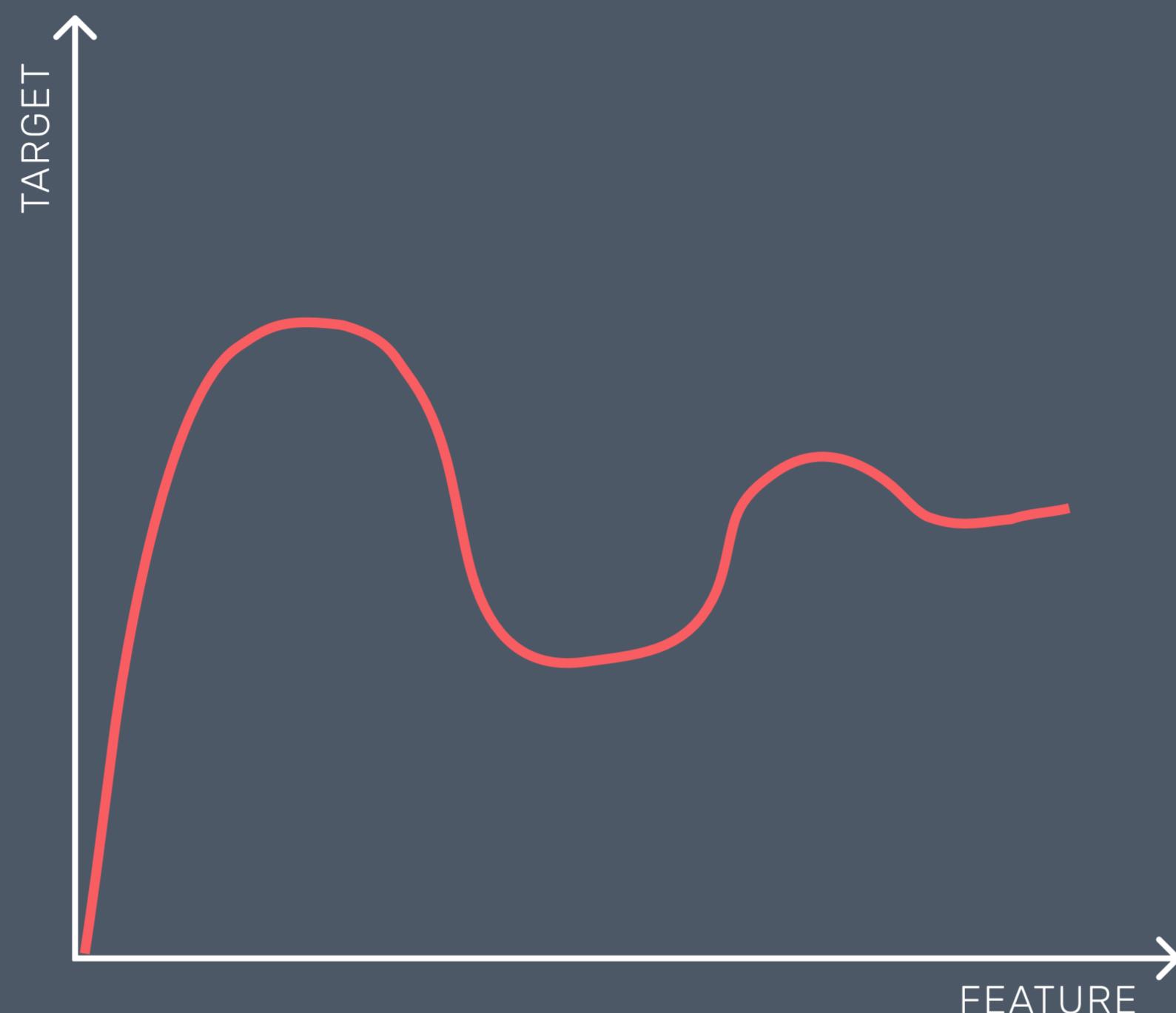
In *feedback*, we compute the parameter gradients and feed them back to the neuron.

Finally, in *adjust*, we increase or decrease the parameters based on the gradients.

Repeat through many epochs and we get a trained model.

## 2 - NON-LINEAR REGRESSION



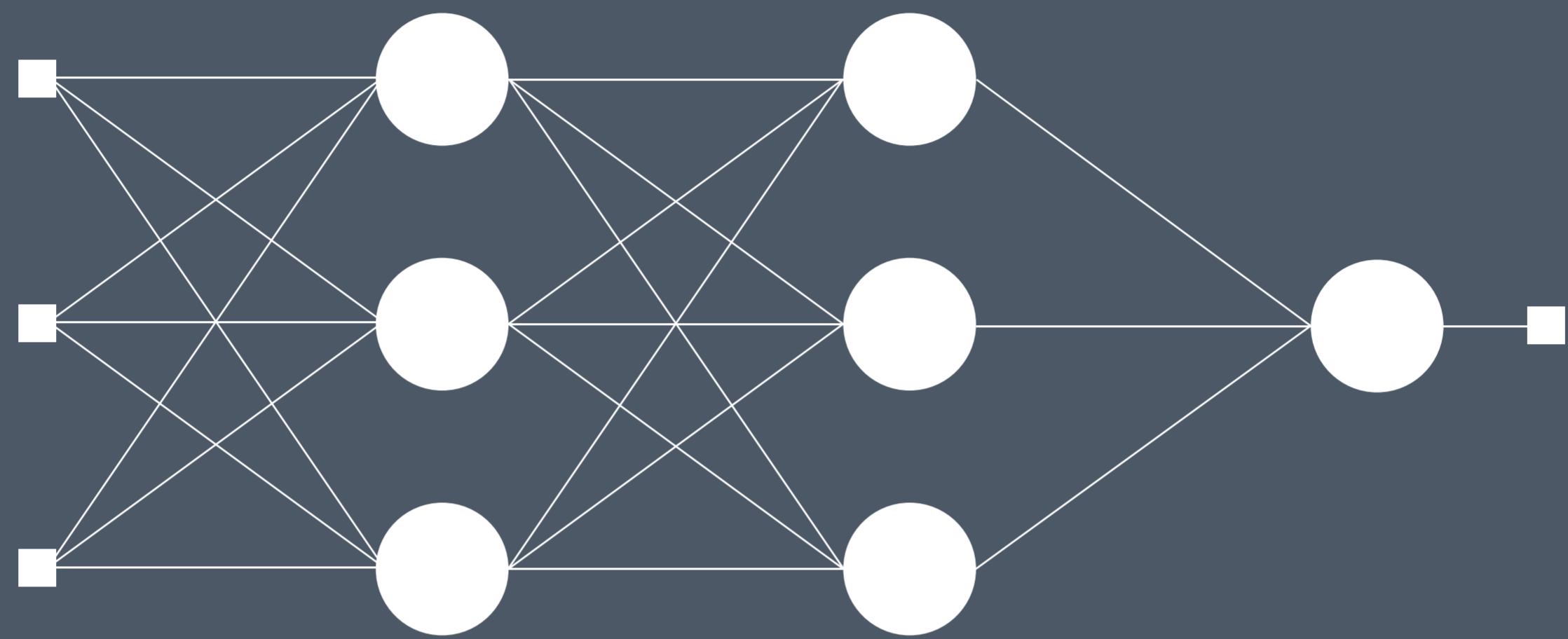


## NON-LINEARITY

In the last chapter, our single-neuron neural network had the task of modeling a linear relationship.

In practice, however, we are more likely to encounter non-linear relationships. The datasets will be more complex. Instead of one, there will be many features at play.

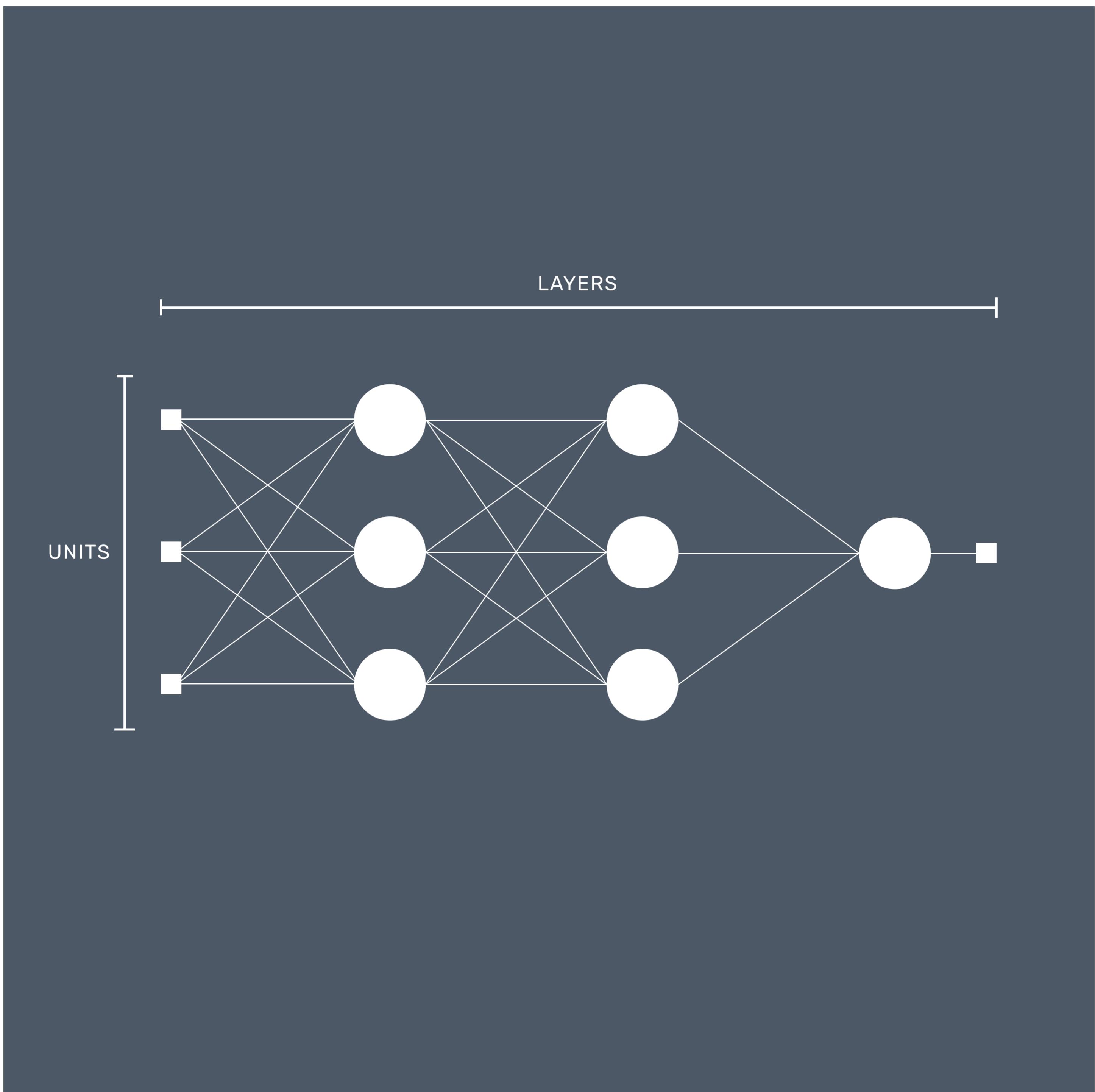
A single neuron won't be able to handle these scenarios.



## NEURAL NETWORK

This is when we need a proper neural network.

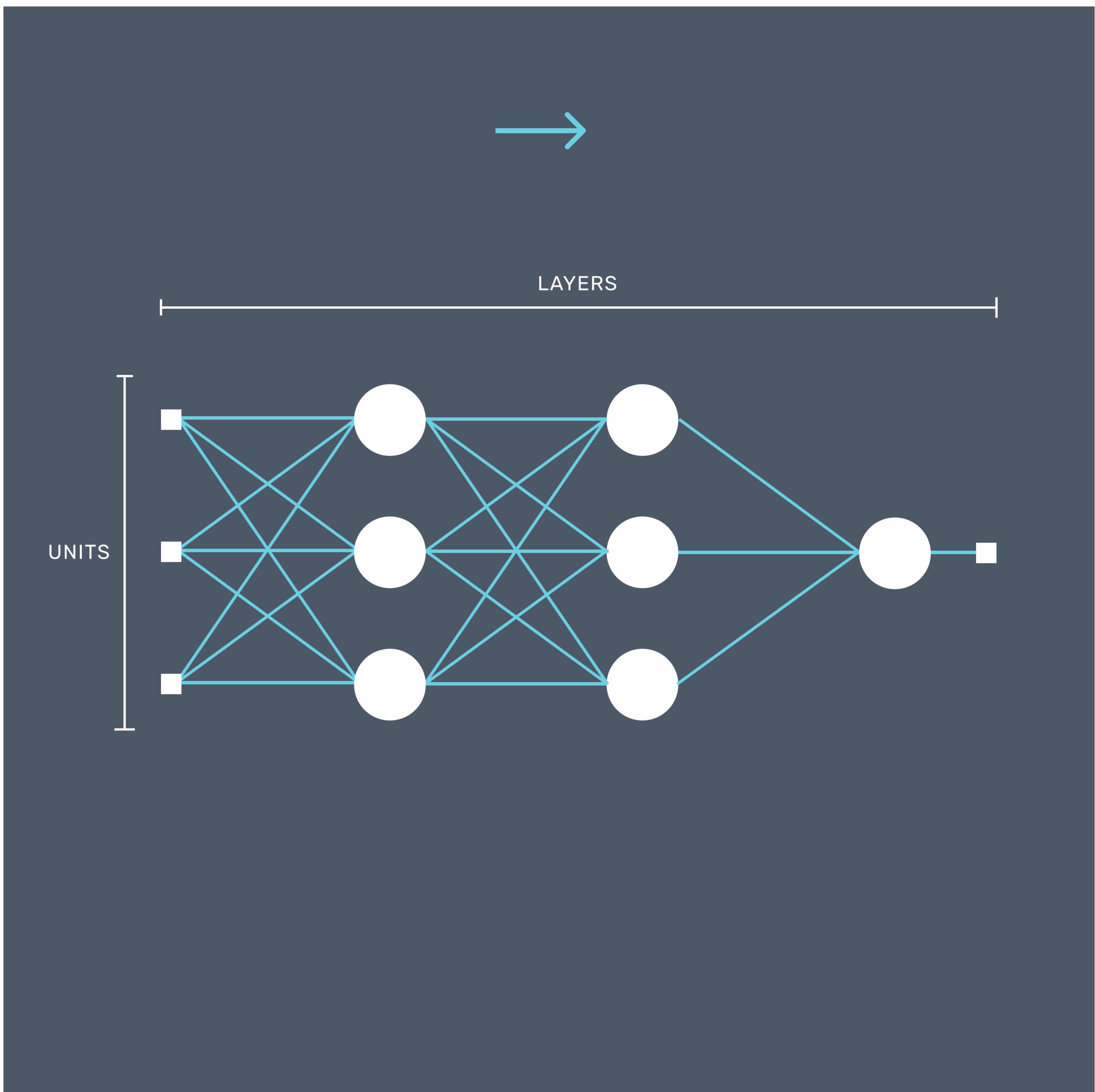
A neural network consists of its building blocks - neurons. These neurons learn uniquely at the individual level and synergistically at the collective level.



## LAYERS AND UNITS

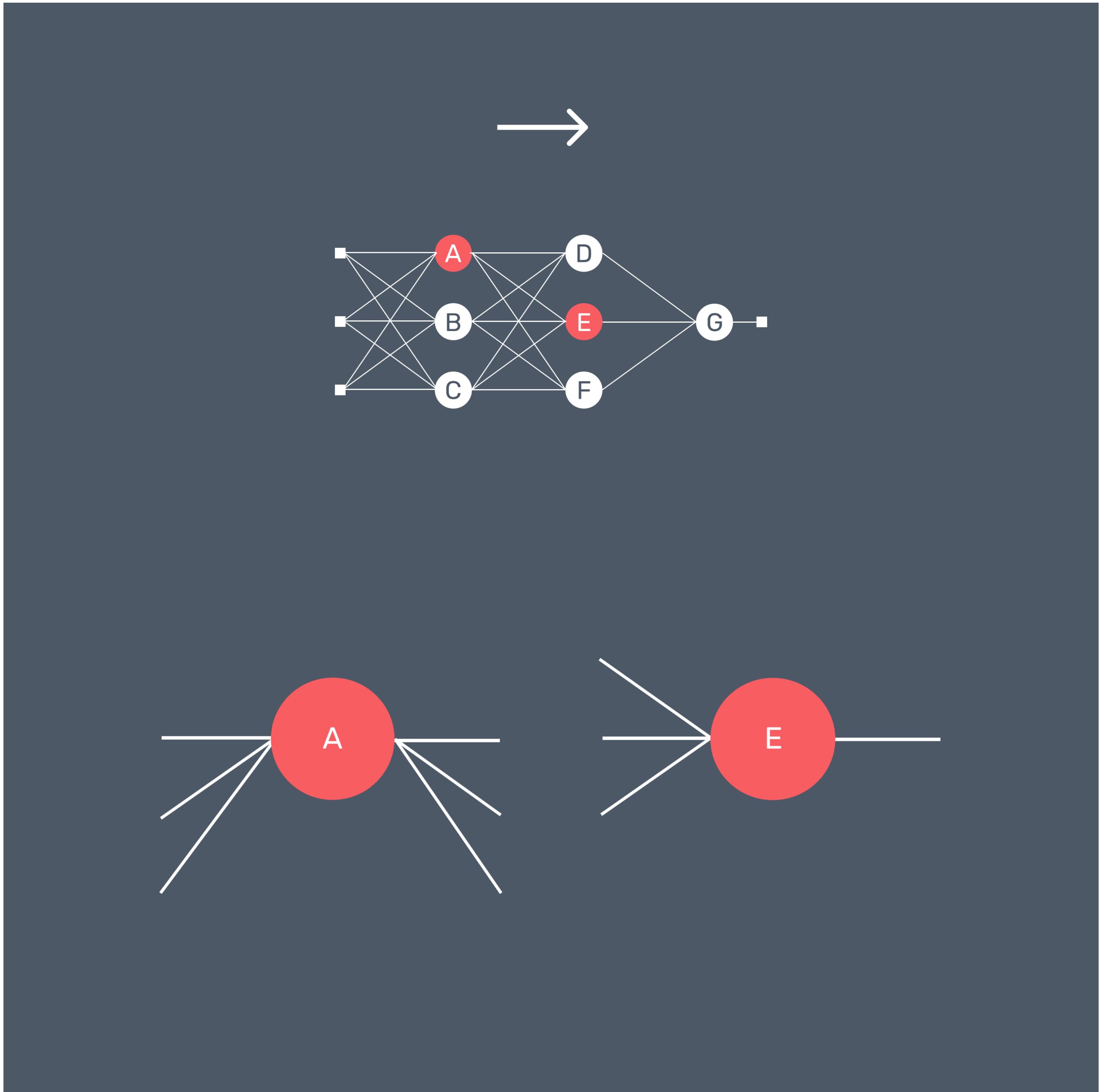
Neural networks come in various configurations called *architectures*.

In its basic architecture, a neural network consists of *layers*. Each layer has its own number of neurons, or *units*.



## DATA FLOW

The lines represent the flow of data. Each neuron receives inputs from all the neurons in the preceding layers. Then it sends its output to all neurons in the next layer.

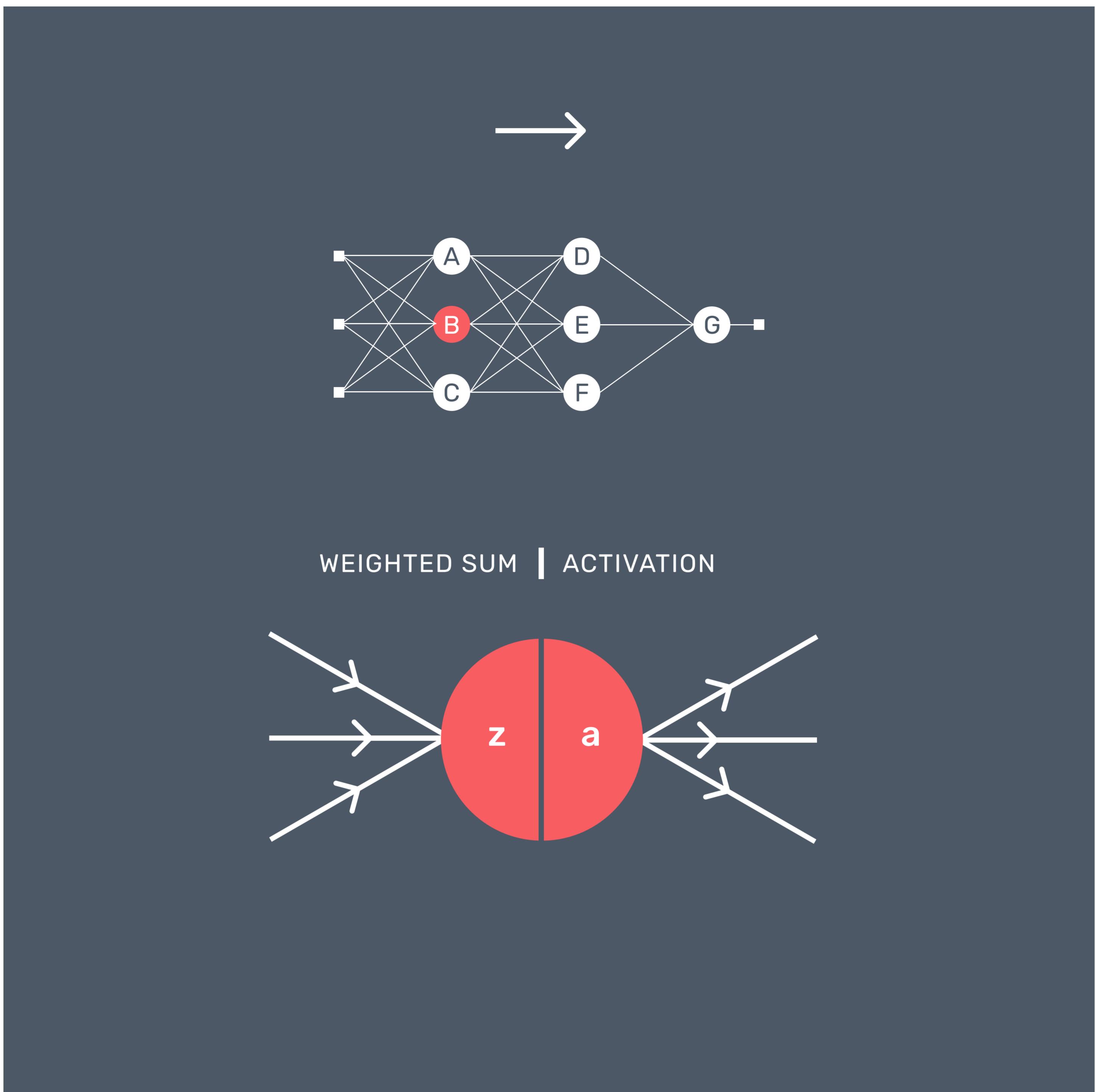


## EXAMPLE

The neurons and connections can make the neural network look complicated, so let's break it down and look at a couple of examples.

Example 1: Neuron A receives three inputs directly from the data and sends its output to 3 neurons in the next layer.

Example 2: Neuron E receives three, which are the outputs of neurons A, B, and C in the previous layer, and sends its output to one neuron in the next layer.



## NEURON COMPUTATIONS

Regardless of how many inputs and outputs a neuron is connected to, or in which layer the neuron is located, it goes through the same set of computations—weighted sum and activation.

Take Neuron B, for example. It takes three inputs, computes the weighted sum on these inputs, and then performs the activation. The output of the activation is then passed to three neurons in the next layer.



## THE GOAL

Let's now build the neural network architecture we need for this task.

Before that, we'll define the goal for this task, which is the same as in Chapter 1—to predict a hotel's average daily room rates (i.e. *price*).



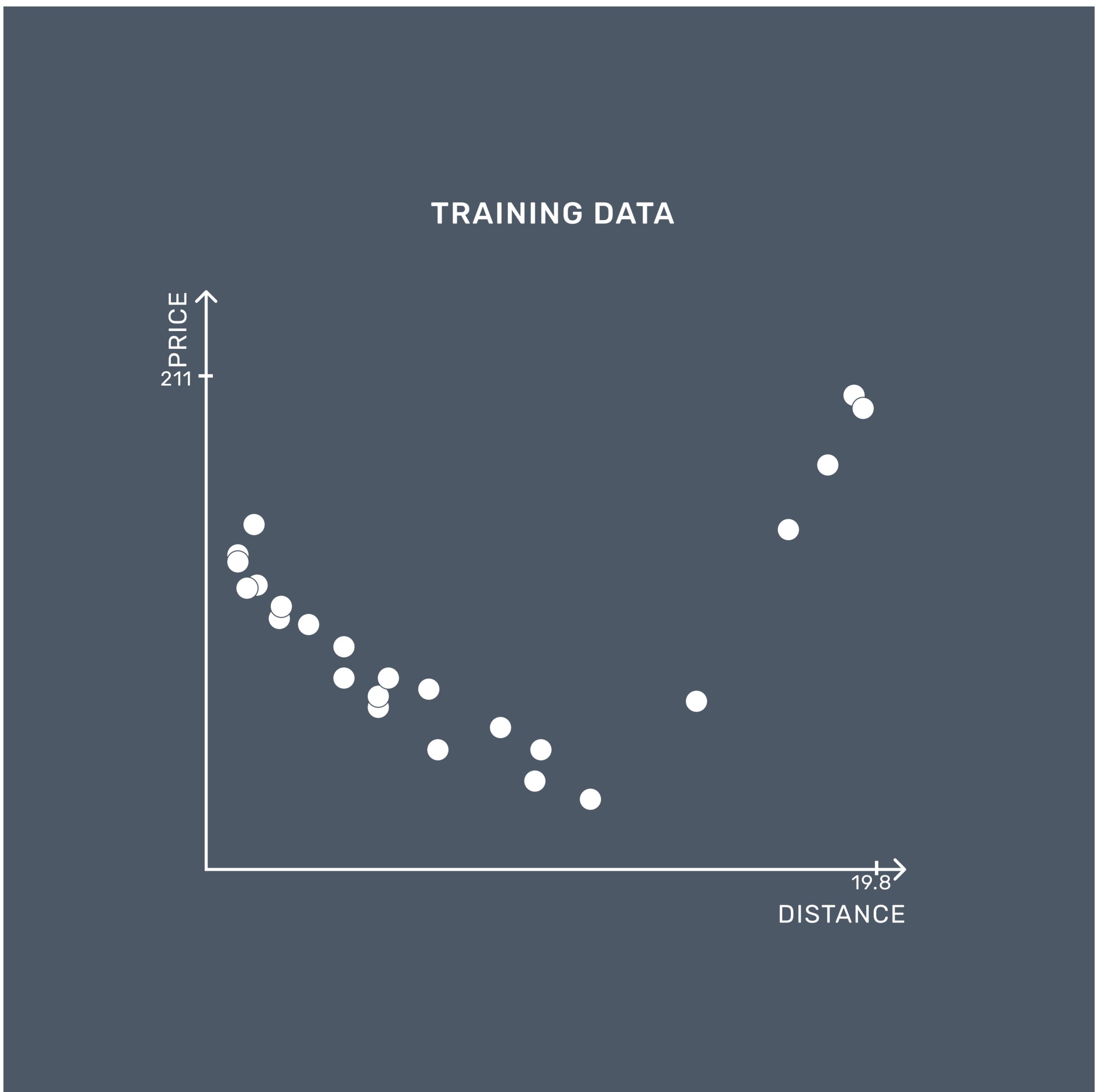
	DIST (MI)	RATING	PRICE (\$)
TRAINING DATA (24)	0.2	3.5	157.00
	0.2	4.8	155.00
	0.5	3.7	146.00
	0.7	4.3	168.00
	0.8	2.7	147.00
	1.5	3.6	136.00
	1.6	2.6	140.00
	2.4	4.7	134.00
	3.5	4.2	116.00
	3.5	3.5	127.00
	4.6	2.8	106.00
	4.6	4.2	110.00
	4.9	3.8	116.00
	6.2	3.6	112.00
	6.5	2.4	92.00
	8.5	3.1	99.00
	9.5	2.1	81.00
	9.7	3.7	92.00
	11.3	2.9	75.00
	14.6	3.8	108.00
	17.5	4.6	166.00
	18.7	3.8	188.00
	19.5	4.4	211.00
	19.8	3.6	207.00
TEST DATA (8)	0.3	4.6	156.00
	0.5	4.2	162.00
	1.1	3.5	149.00
	1.2	4.7	145.00
	2.7	2.7	123.00
	3.8	4.1	118.00
	7.3	4.6	82.00
	19.4	4.8	209.00

## THE DATASET

The difference is this time, we have two features instead of one. Here, we bring back the rating feature that we left out in Chapter 1.

Another difference is the size of the dataset. We had only 12 data points in Chapter 1. For this task, we are adding 20 more, making up a total of 32 data points. We'll use 24 for training and 8 for testing.

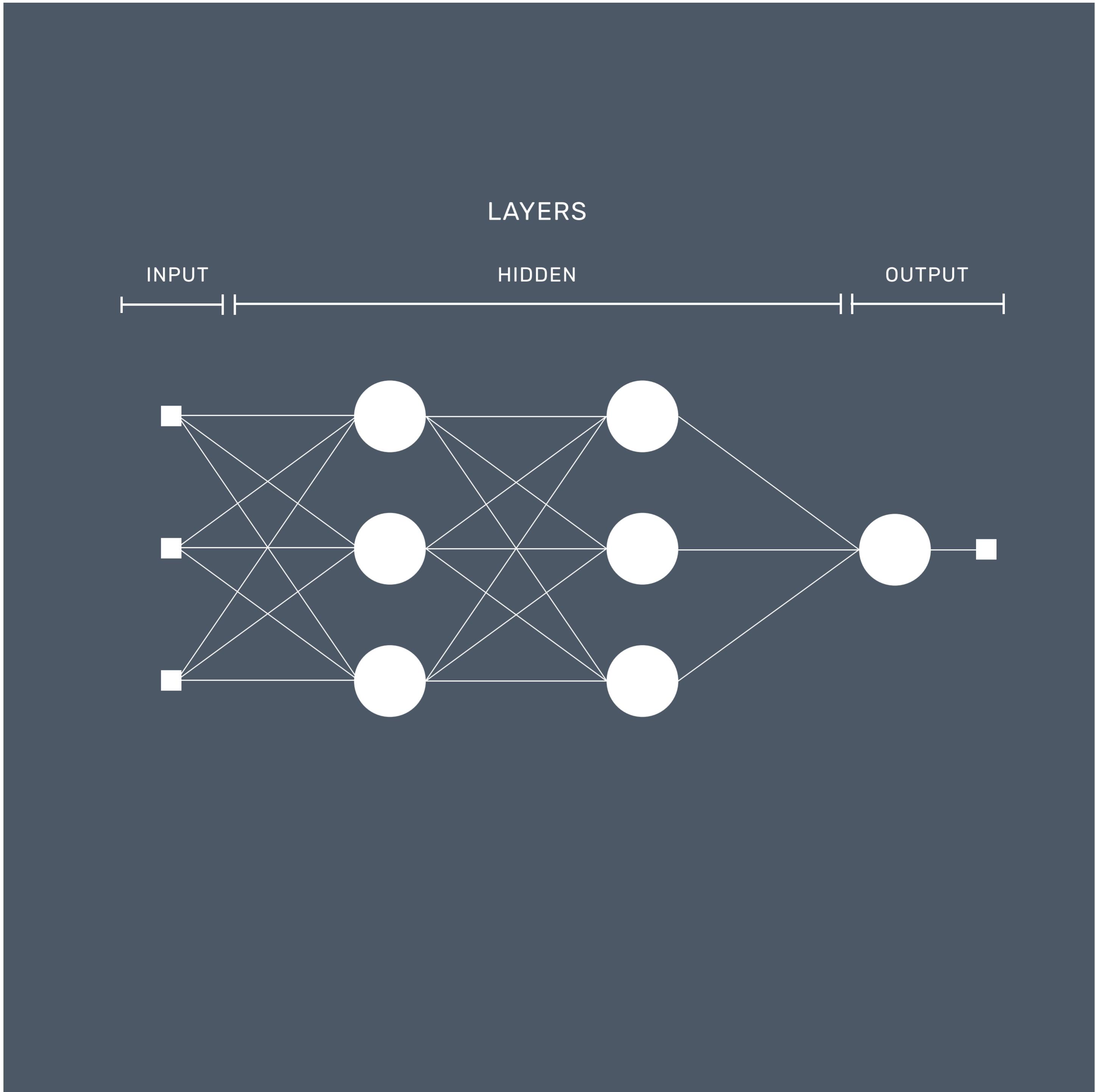
The result is, instead of linear, our task now becomes a *non-linear regression* task. Let's see why this is so.



## THE DATASET

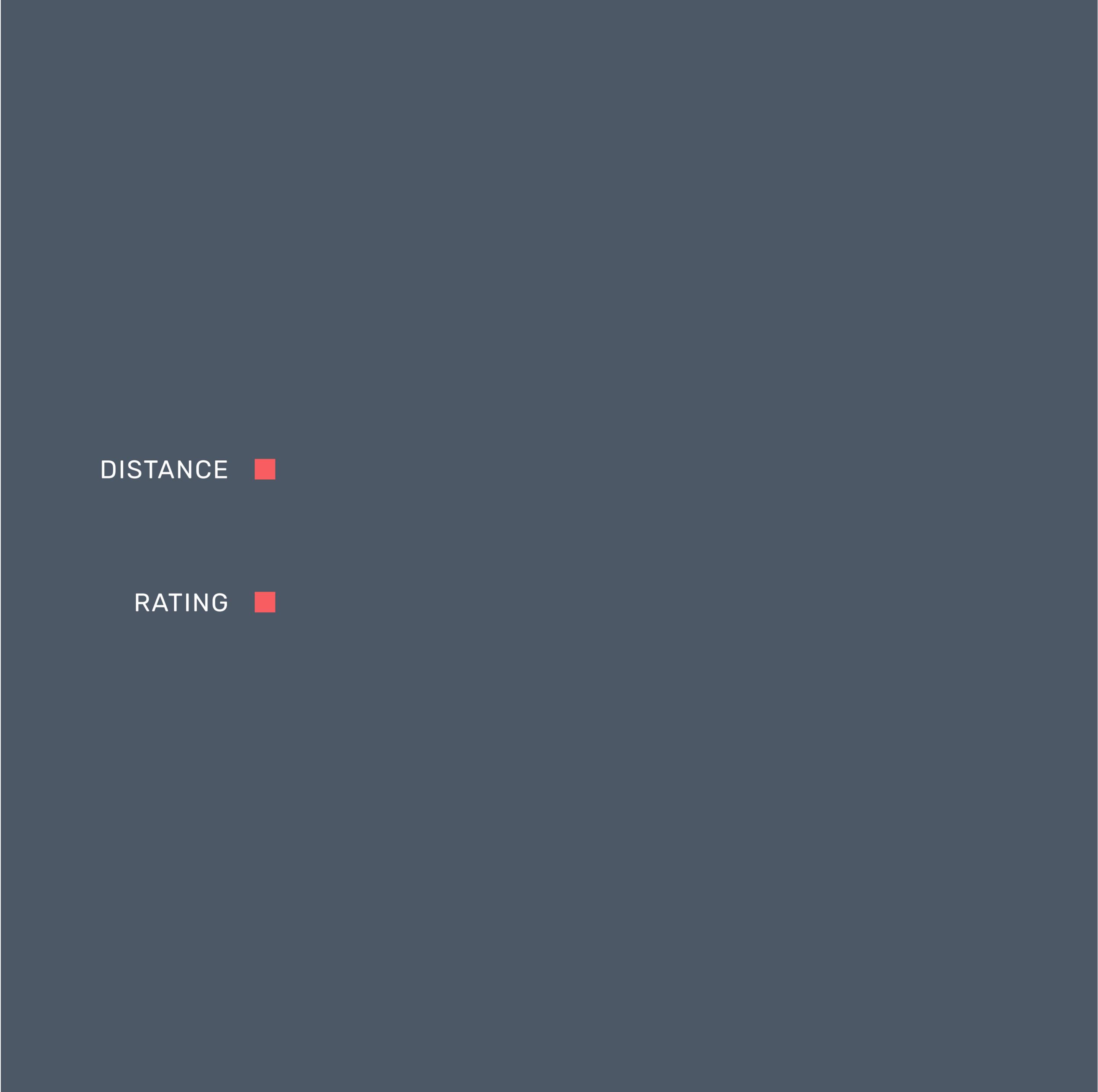
The data points we used in Chapter 1 are on the left side of this curve. As we add more hotels to the dataset, we find that the dynamic changes. In the beginning, the farther we get from the city center, the cheaper the prices become. This is expected because there will be a higher demand for hotels closer to the center. But there is a point in the middle where the room rates get more expensive the further away we get. The reason is that these are the resort-type hotels that charge similar, if not higher, prices.

This dataset no longer has a linear relationship. The distance-price relationship now has a bowl shape, which is *non-linear*. This is what we want our neural network to produce.



## LAYERS

We have seen that a neural network consists of layers. A typical neural network, like the one we are building, has one *input layer* and one *output layer*. Everything in between is called the *hidden layer*.



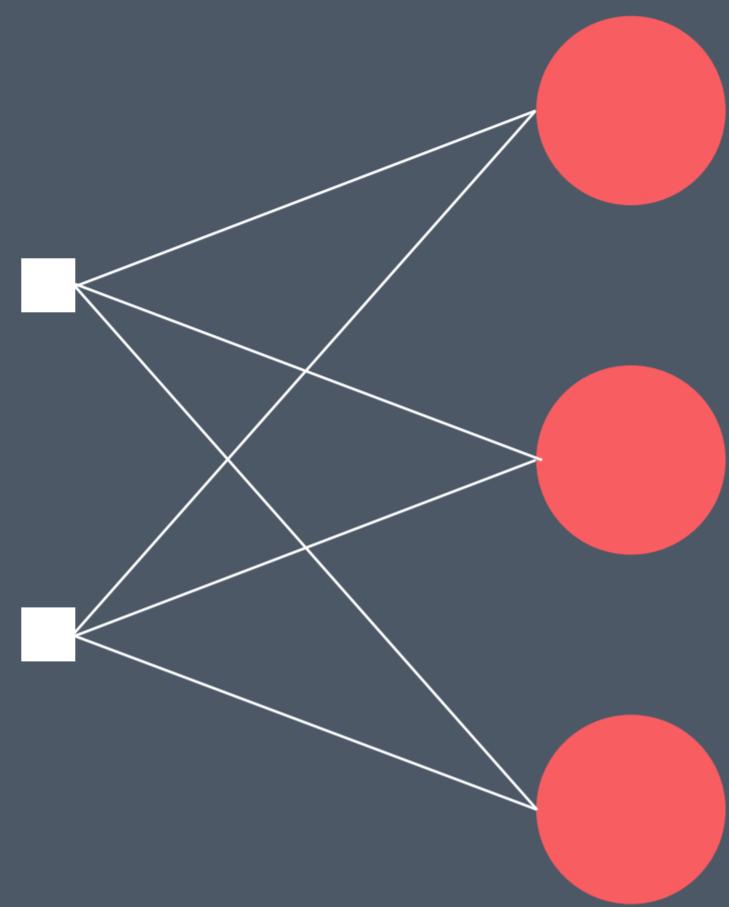
DISTANCE ■

RATING ■

## INPUT LAYER

Let's get started with the architecture.

The number of inputs is equal to the number of features, which means we'll have two inputs.



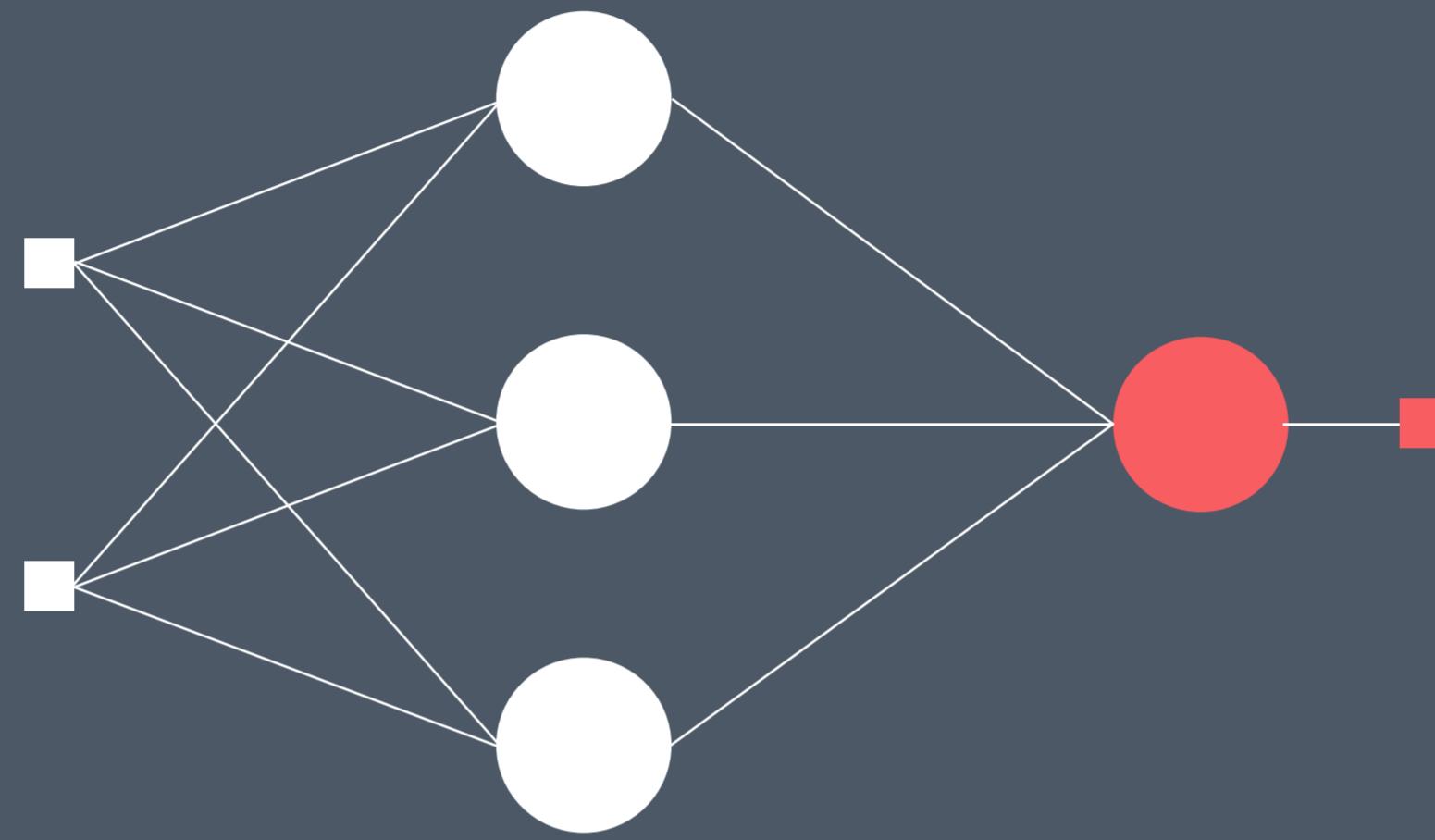
## HIDDEN LAYER

We'll have one hidden layer consisting of three units of neurons.

The choice of the number of layers and units depends on the complexity of the data and the task. In our case, we have a small dataset, so this configuration is sufficient.

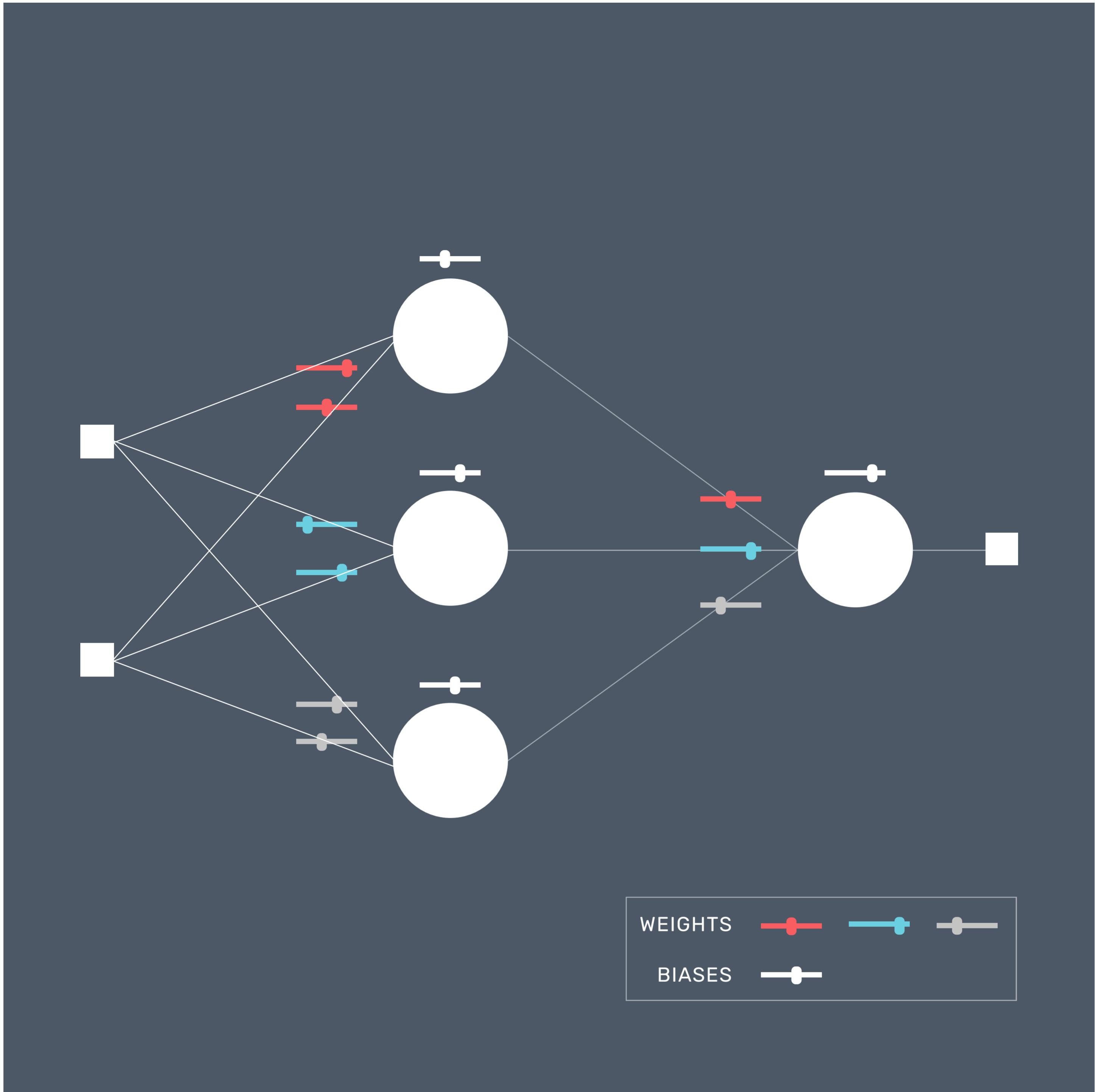
What do hidden layers do? A hidden layer transforms the information it receives from the previous layer into useful forms. Guided by the goal of the task, it looks for patterns and signals and decides which ones are important.

This cascades across the layers and up to the output layer, which will have received a summarized and relevant piece of information to aid its predictions.



## OUTPUT LAYER

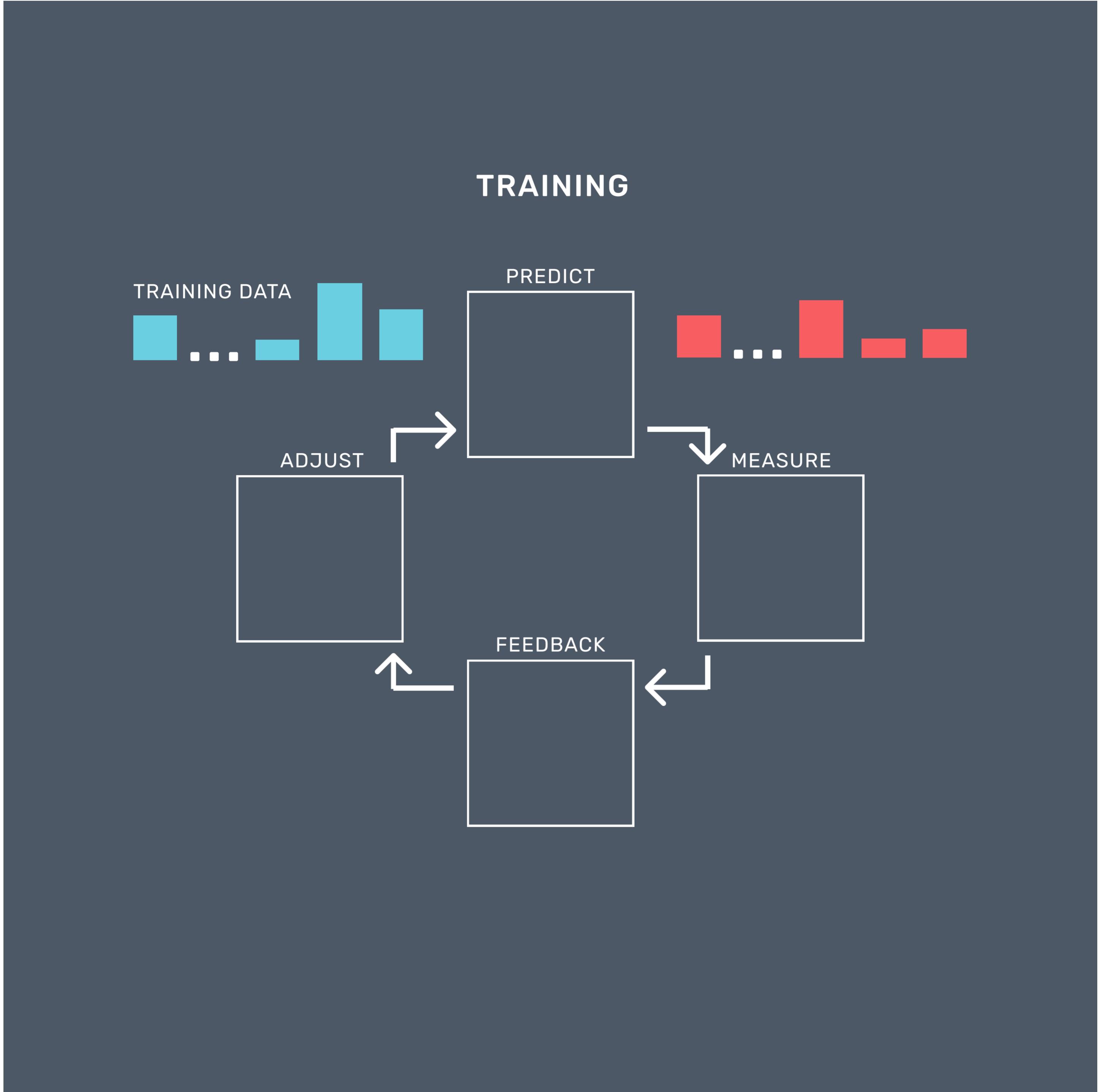
We complete the neural network by adding one unit of neuron in the output layer, whose job is to output the predicted prices.



## WEIGHTS AND BIASES

As for the parameters, recall that each neuron has weights equal to the number of its inputs and one bias.

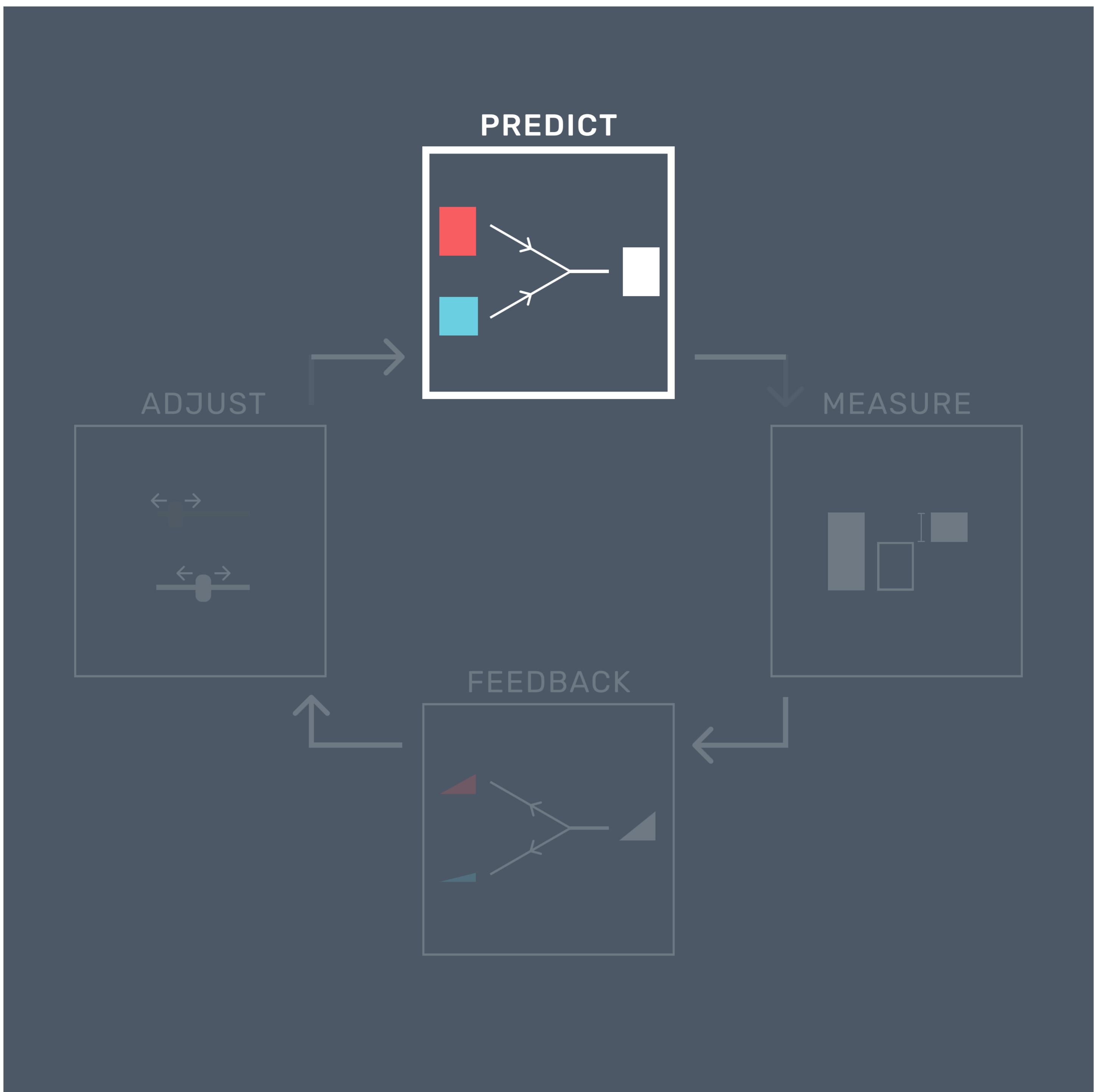
So, in our case, we have a total of nine weights and four biases. And as in Chapter 1, we'll assign initial values for these parameters.



## TRAINING

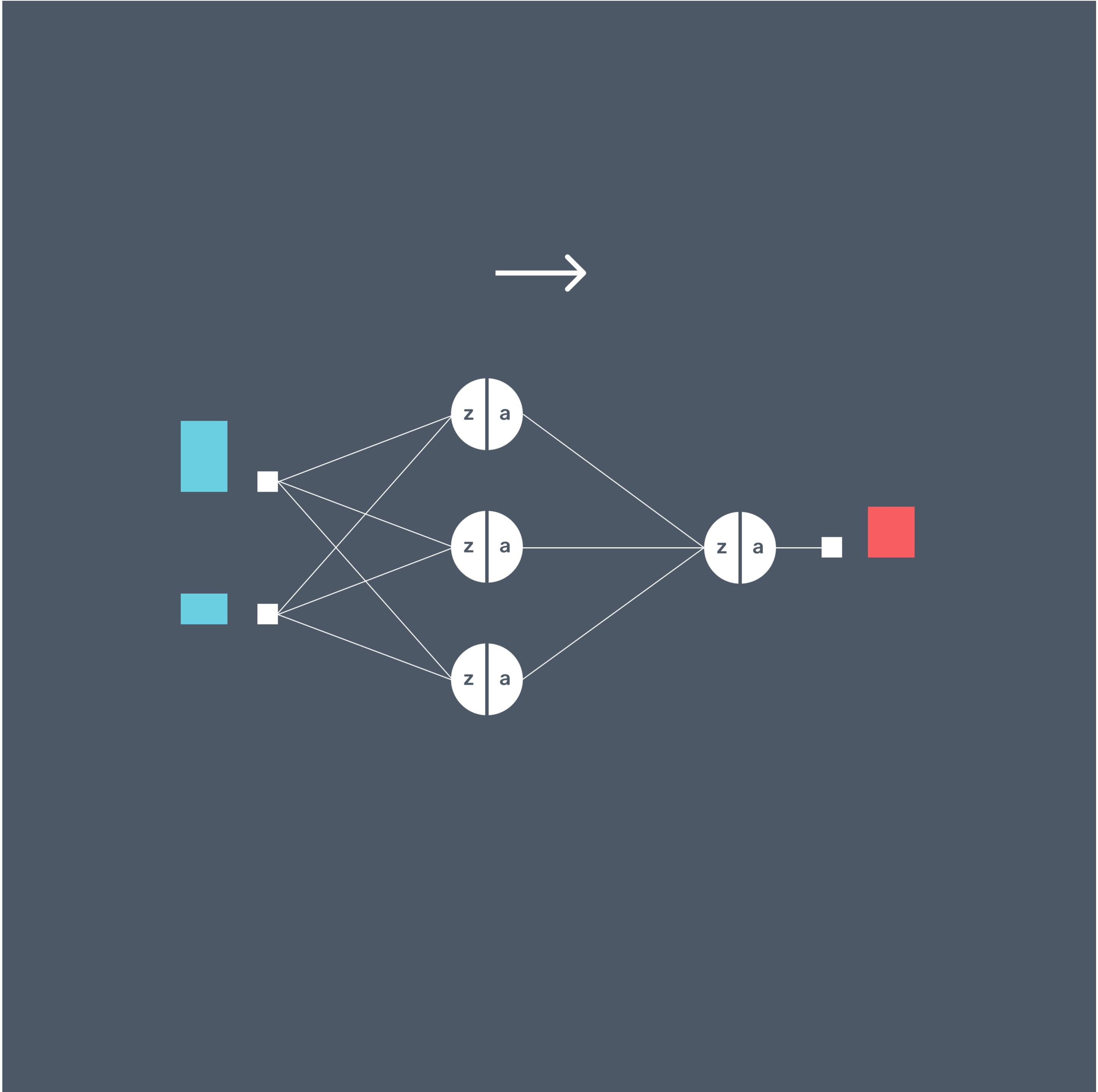
Now that the data and architecture are in place, it's time to start training.

Recall the four-step cycle: *Predict - Measure - Feedback - Adjust*.



## PREDICT

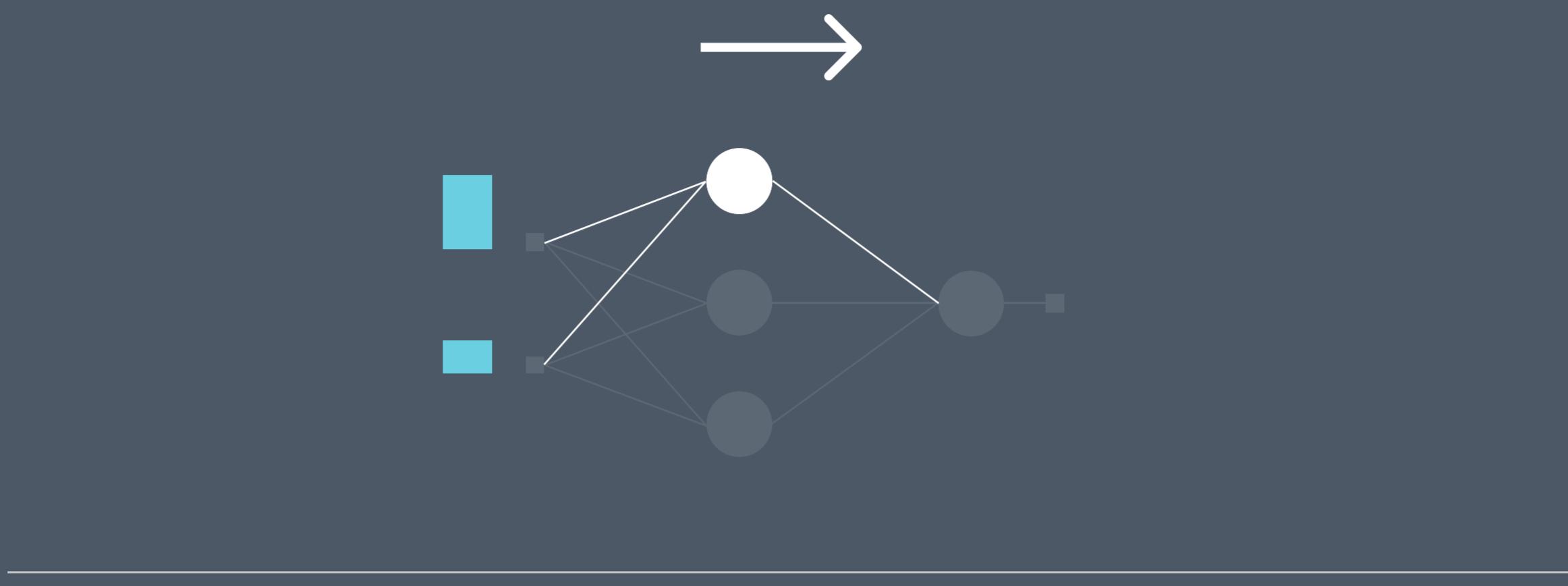
Let's begin with the first step, *predict*.



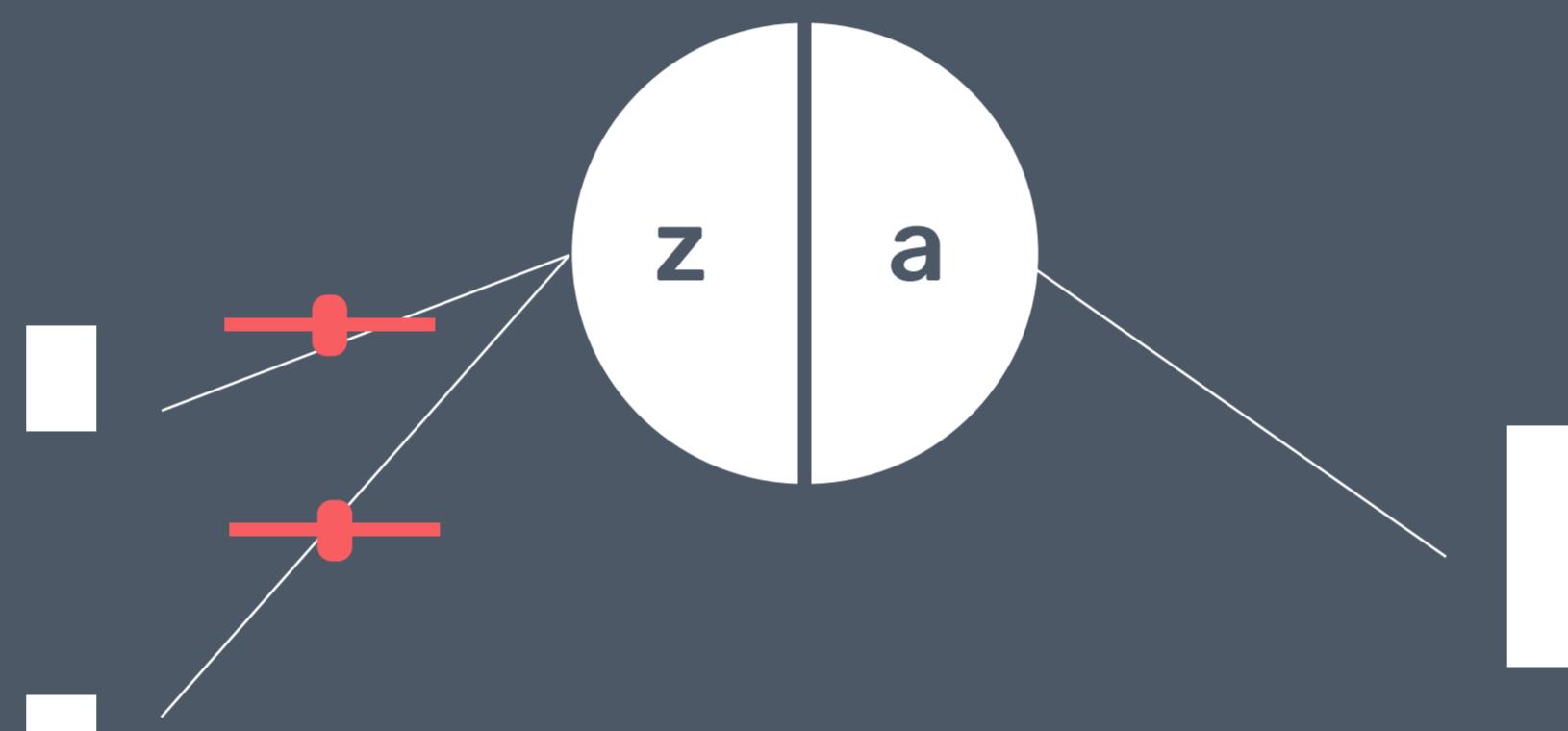
## PREDICT

Recall that in this step, each data point is passed through the neural network and prediction is generated on the other side.

Now that we have more neurons, the weighted sum and activation computations will take place at each neuron. Let's look at a couple of examples.



WEIGHTED SUM | ACTIVATION

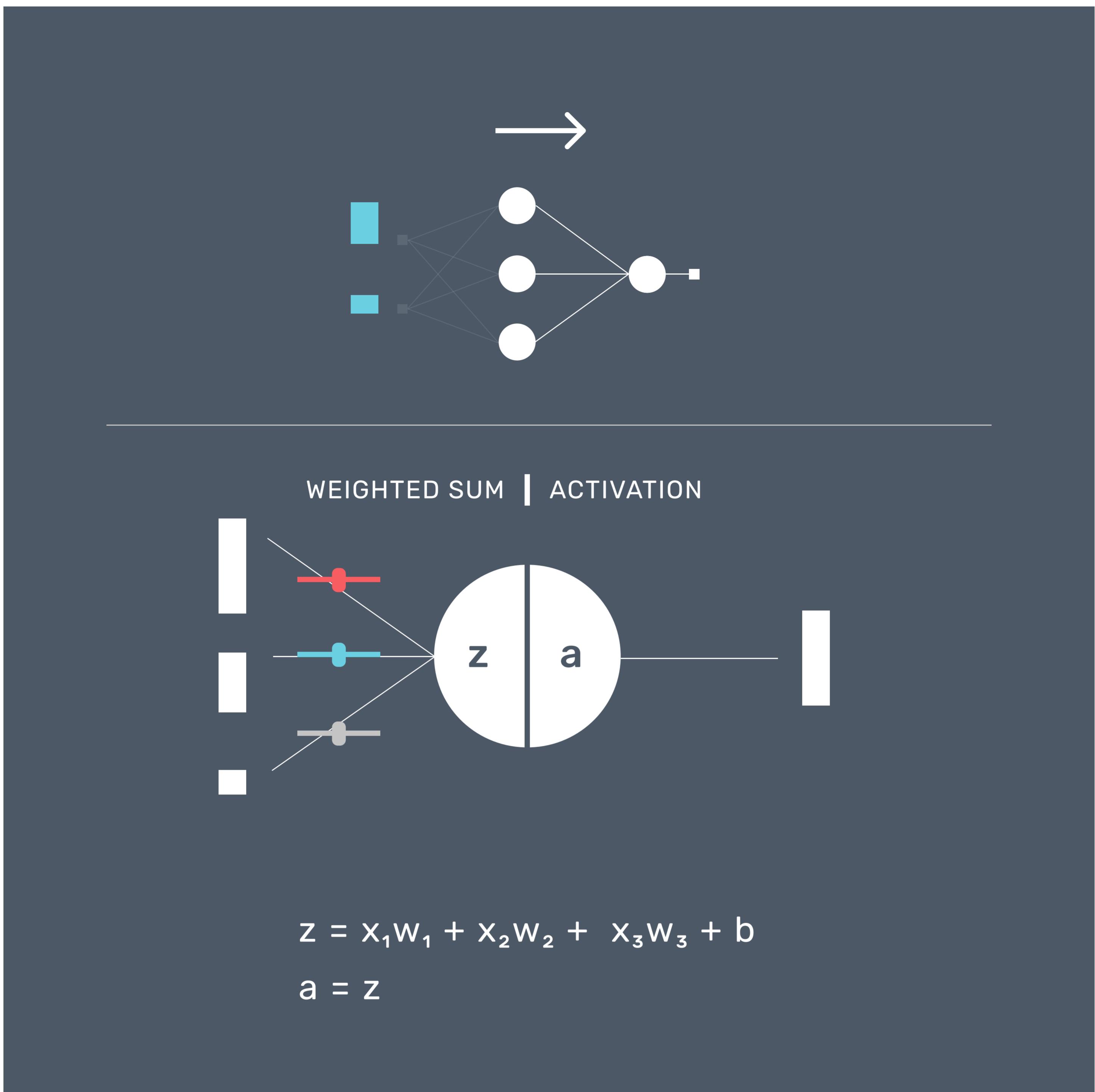


$$z = x_1 w_1 + x_2 w_2 + b$$

$$a = z$$

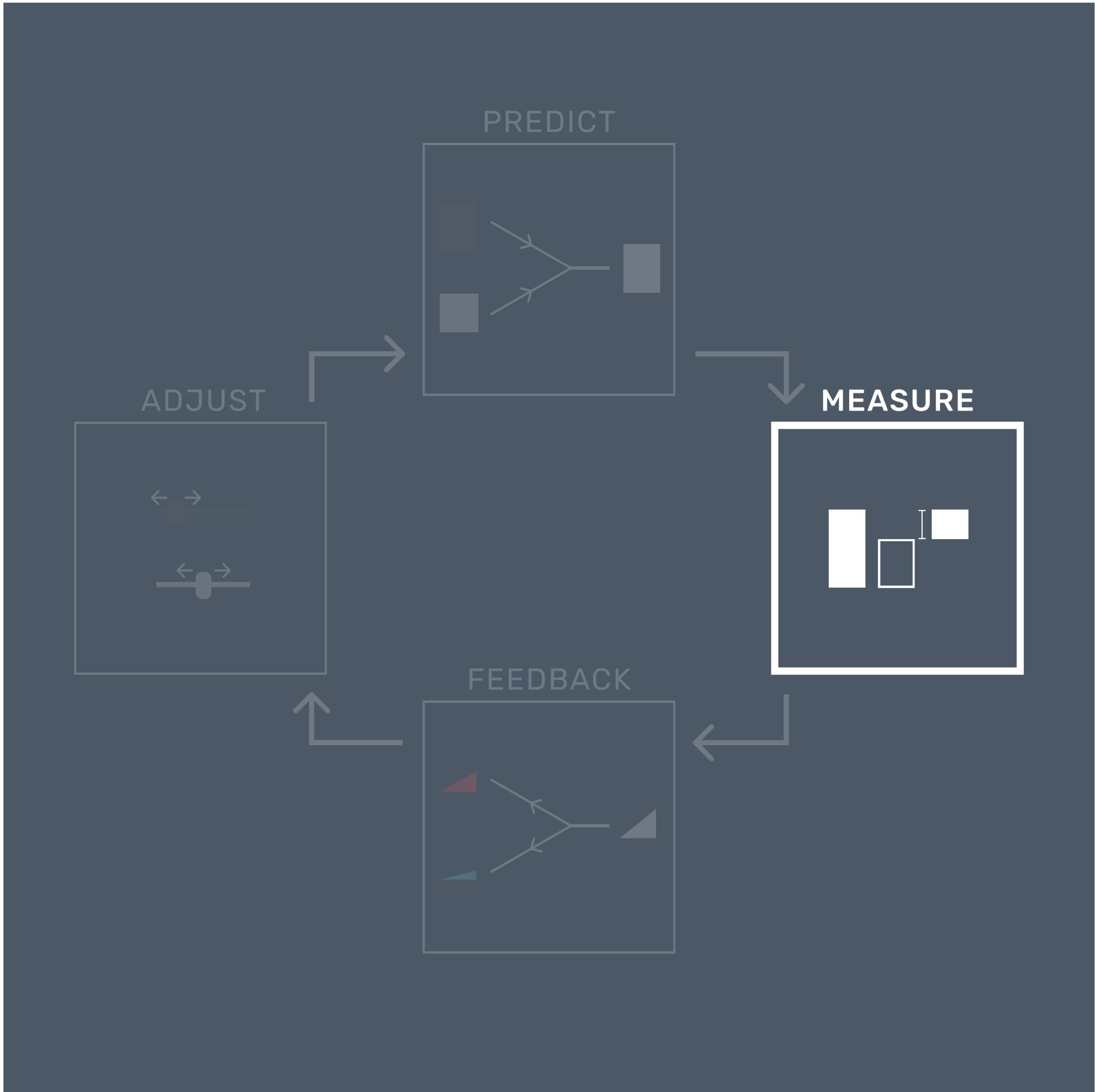
## EXAMPLE 1

The first example is the first neuron in the hidden layer. It takes the original data's features as inputs, performs the weighted sum, and adds a bias value. Then it goes through a linear activation function, which returns the same output as the input.



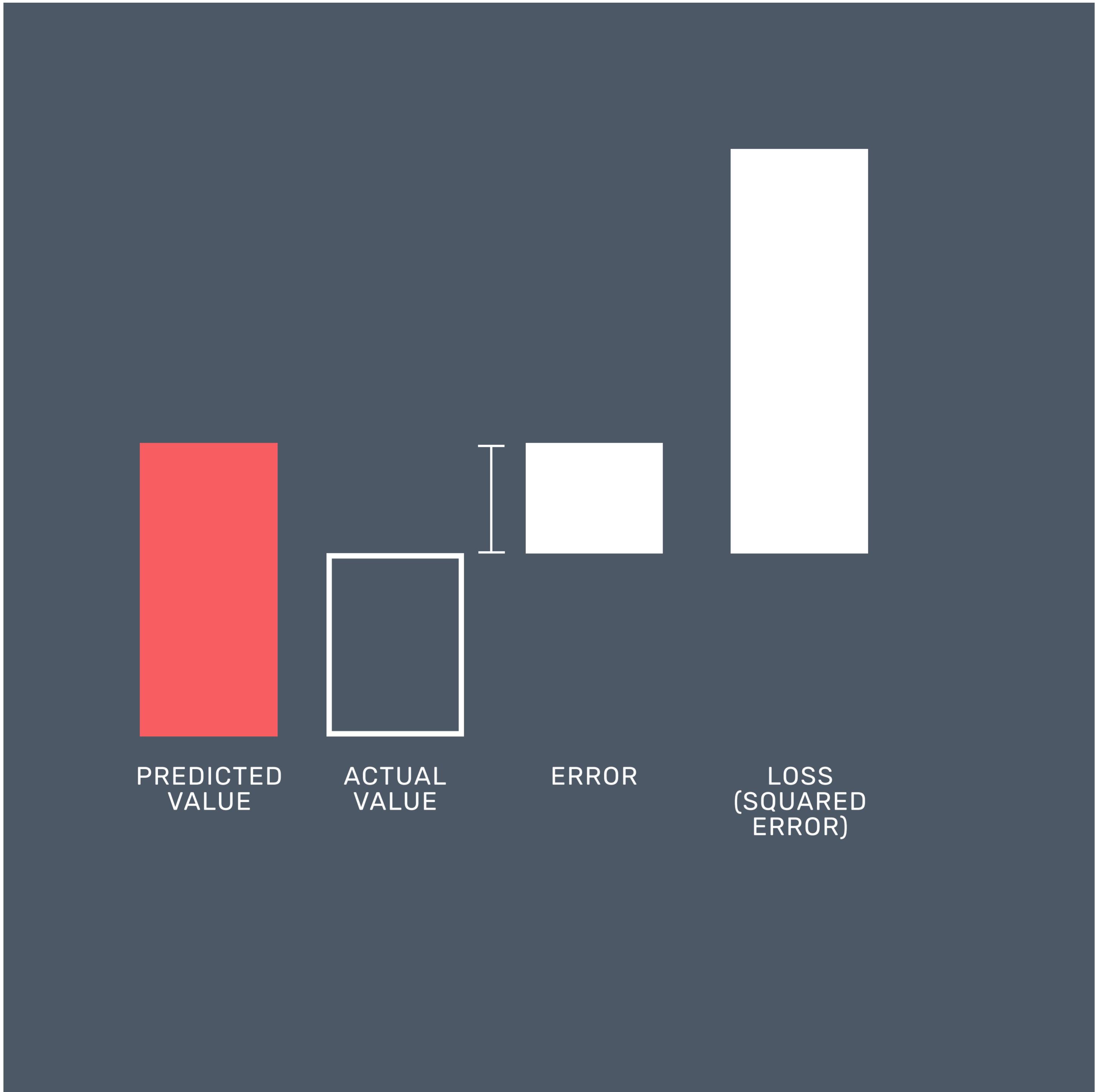
## EXAMPLE 2

The second example is the neuron in the output layer. It takes the three outputs of the previous layer as inputs. As in the first example, it then performs the weighted sum, adds a bias, and performs the linear activation.



## MEASURE

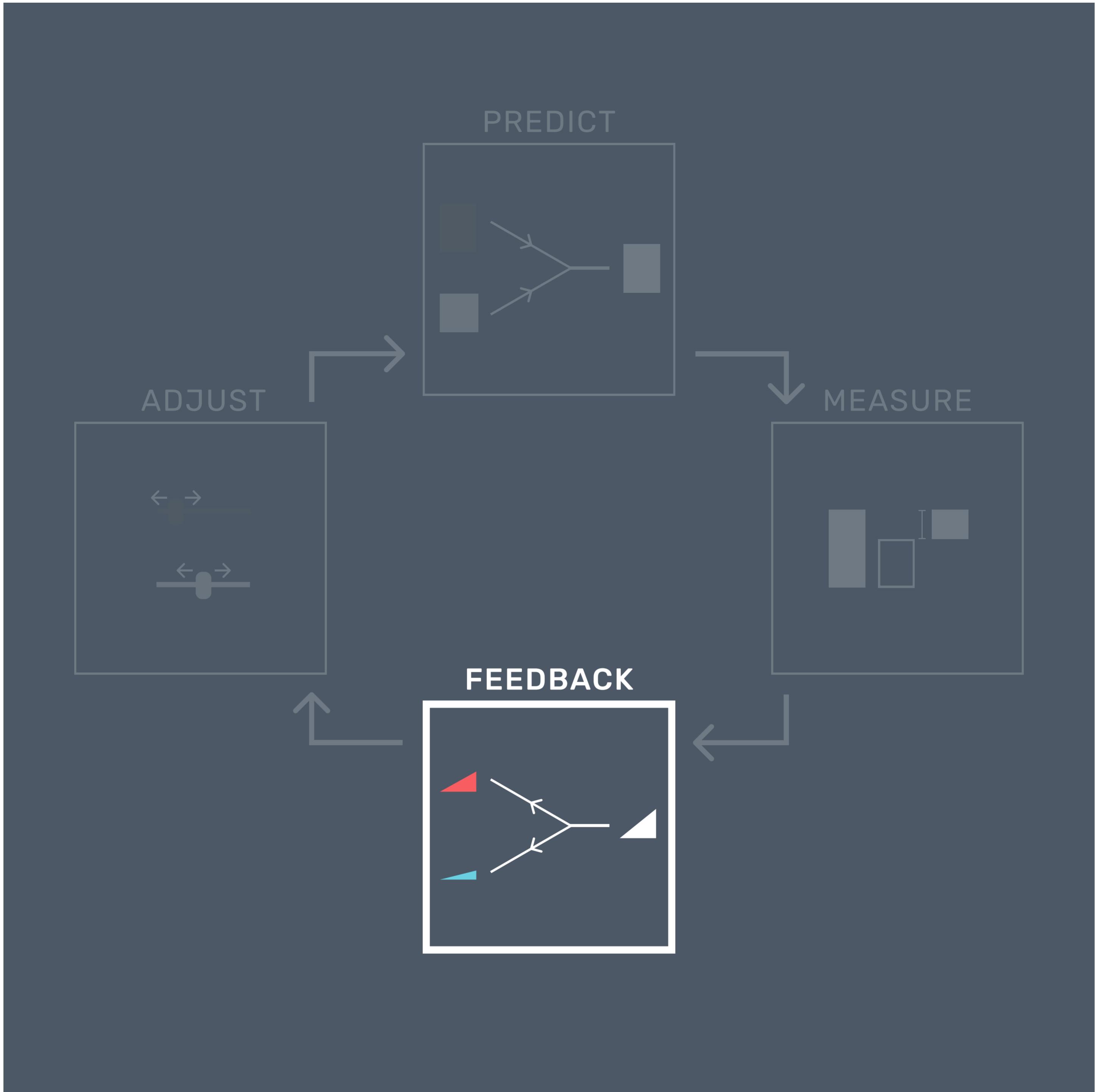
In the second step, *measure*, we quantify the performance of the prediction.



## MEASURE

This is still a regression task, so we can use the same loss function as in Chapter 1—the MSE.

Averaging the squared error over all twenty-four training data points gives us the MSE.

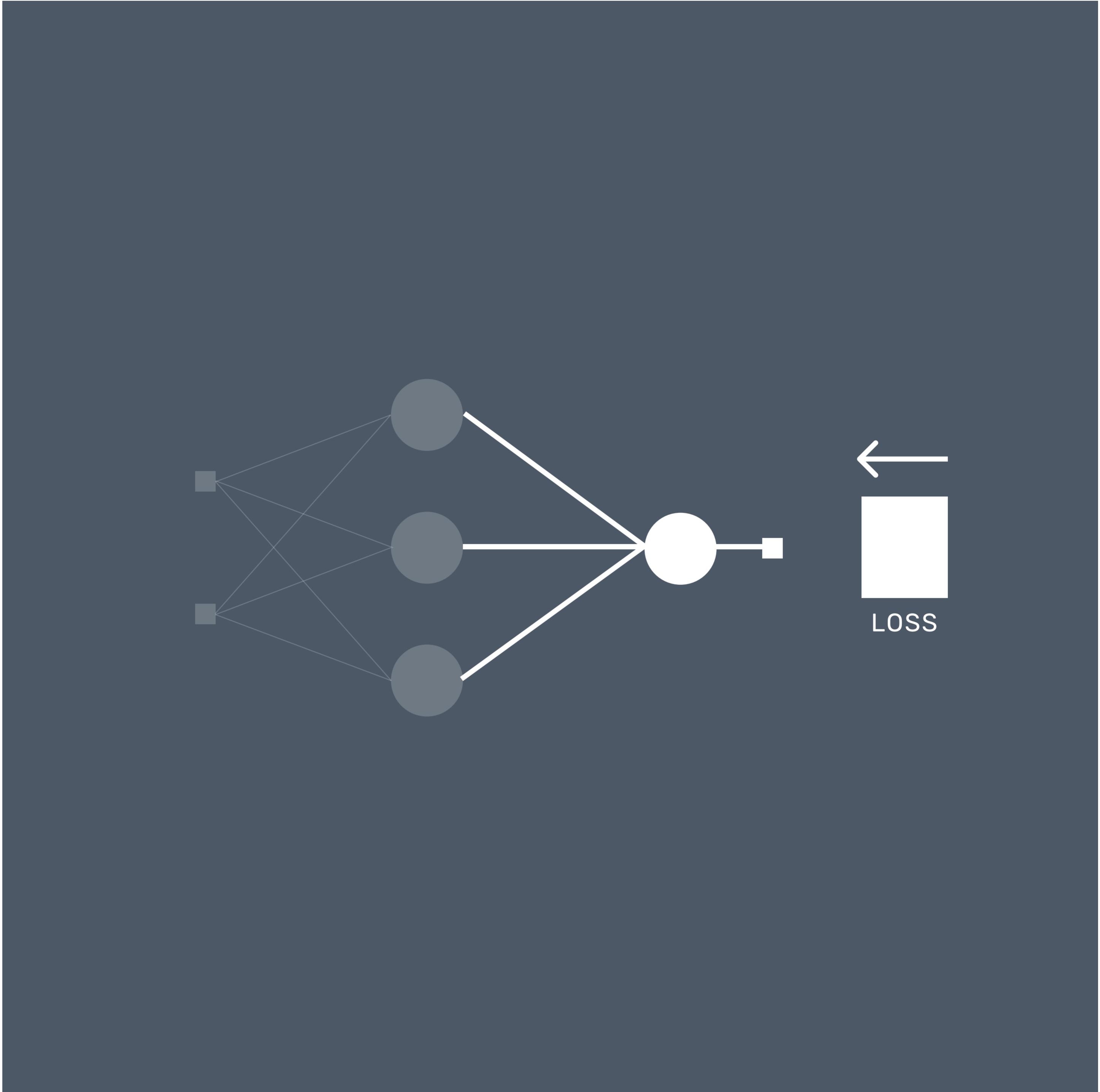


## FEEDBACK

The third step, *feedback*, is where it gets interesting. Here, we'll find a lot more things going on compared to the single-neuron case.

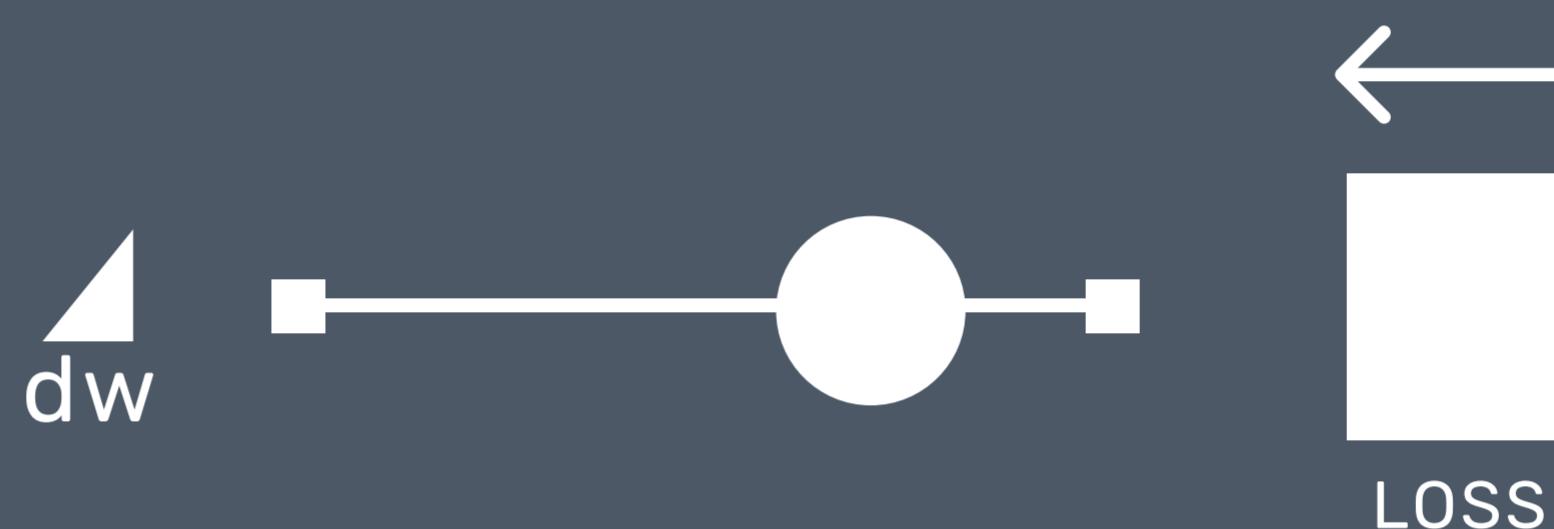
This part of the book will be quite dense. For this, it is helpful to keep in mind the goal of this step, which is to find the parameter gradients so the neural network can adjust its parameters.

Let's dive into it.



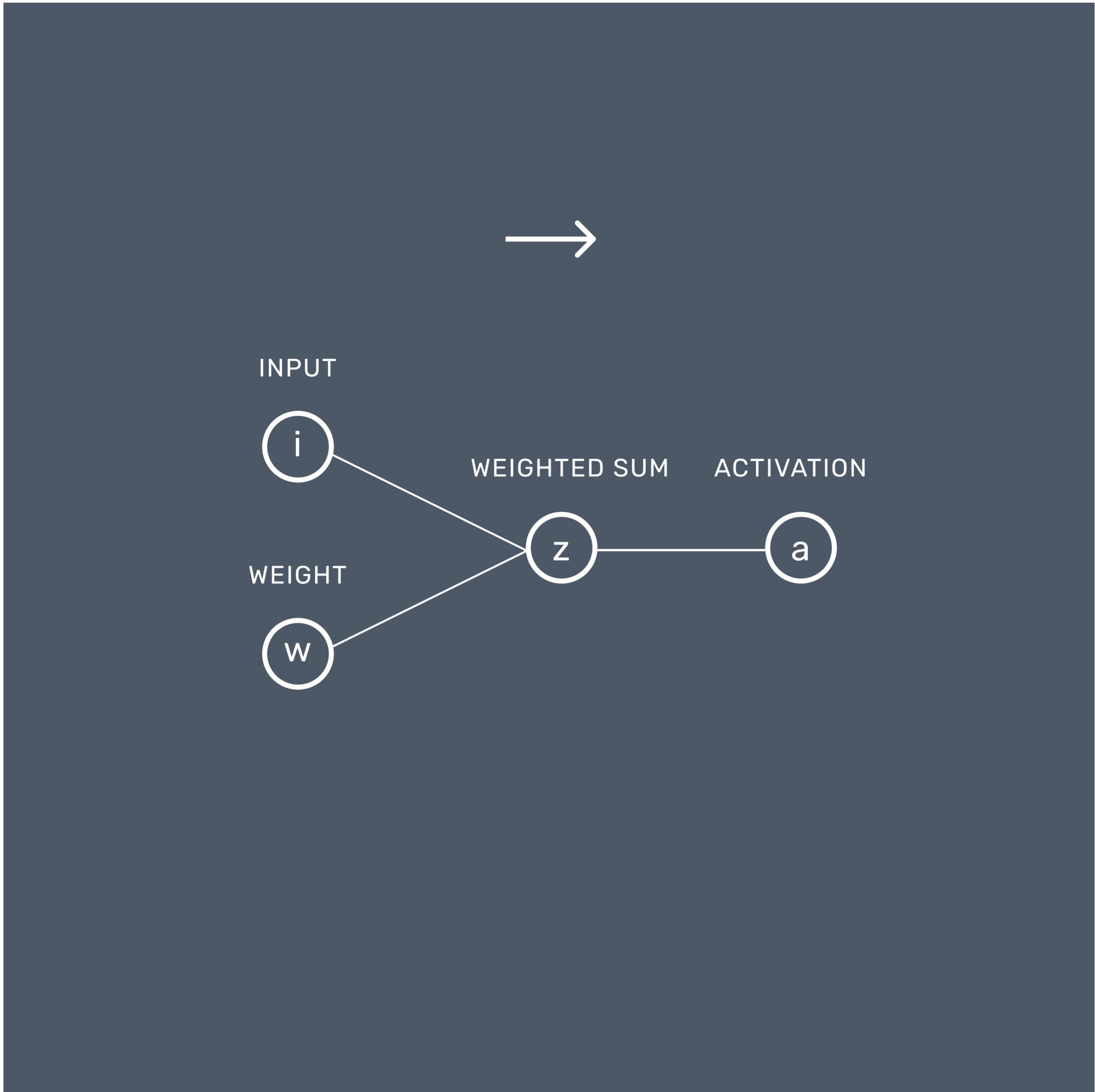
## FEEDBACK

We'll start with the output layer and move backward to the input layer. Again, for simplicity, we'll focus on the weights for now and come back to the biases later.



## SINGLE NEURON

In Chapter 1, with a single-neuron network, we computed the weight gradient based on the loss (MSE). But what really happened under the hood? Let's now see how the loss was fed back to the neural network.

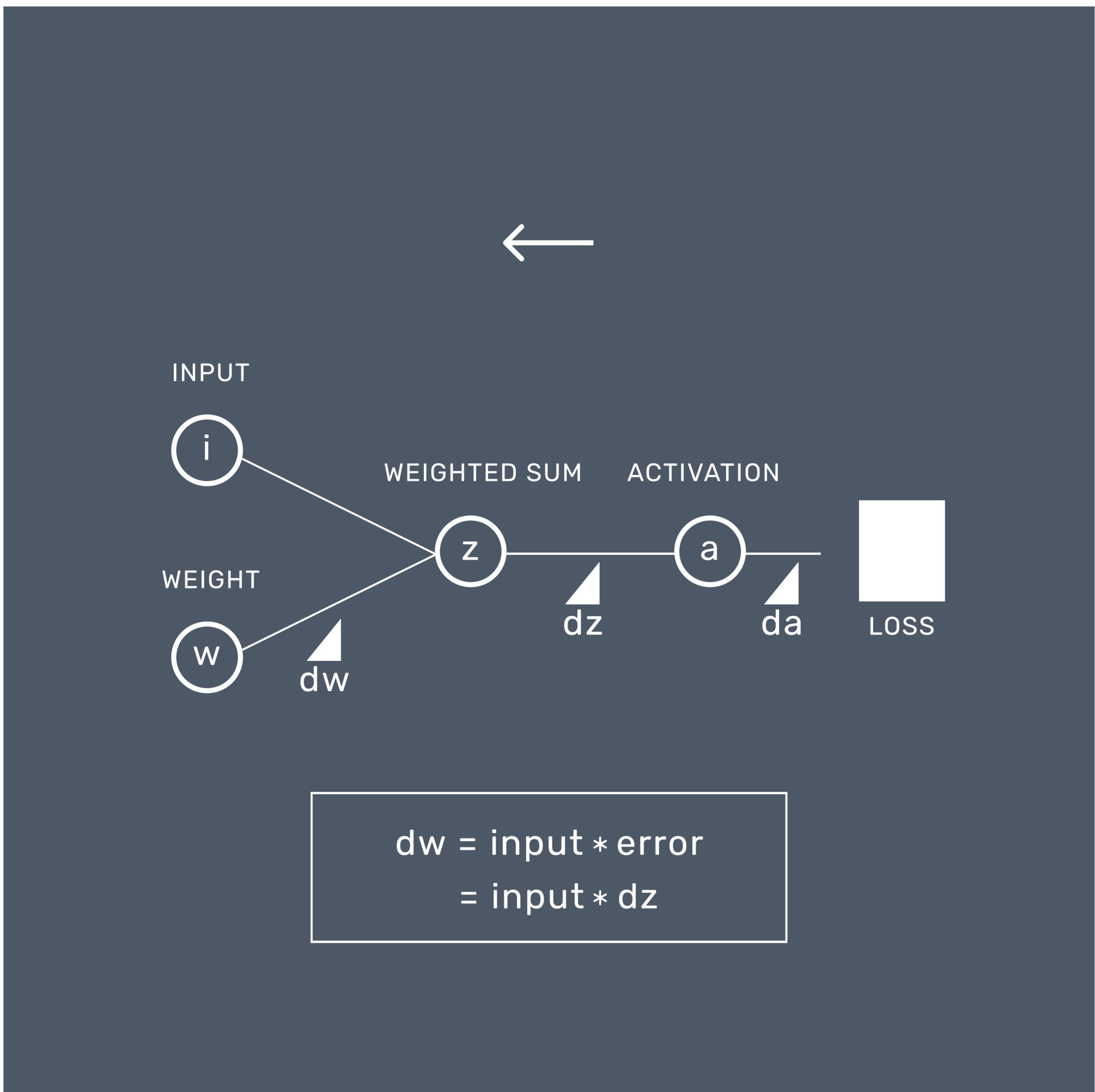


## FORWARD COMPUTATION GRAPH

Keeping to the single-neuron example for now, we can picture the flow of data using a *computation graph*. It provides a way to visualize how information traverses the neural network.

Here we have the forward computation graph, which represents the *predict* step of the training cycle.

Note that the graph you are seeing is a slightly simplified version, sufficient to aid our discussion.

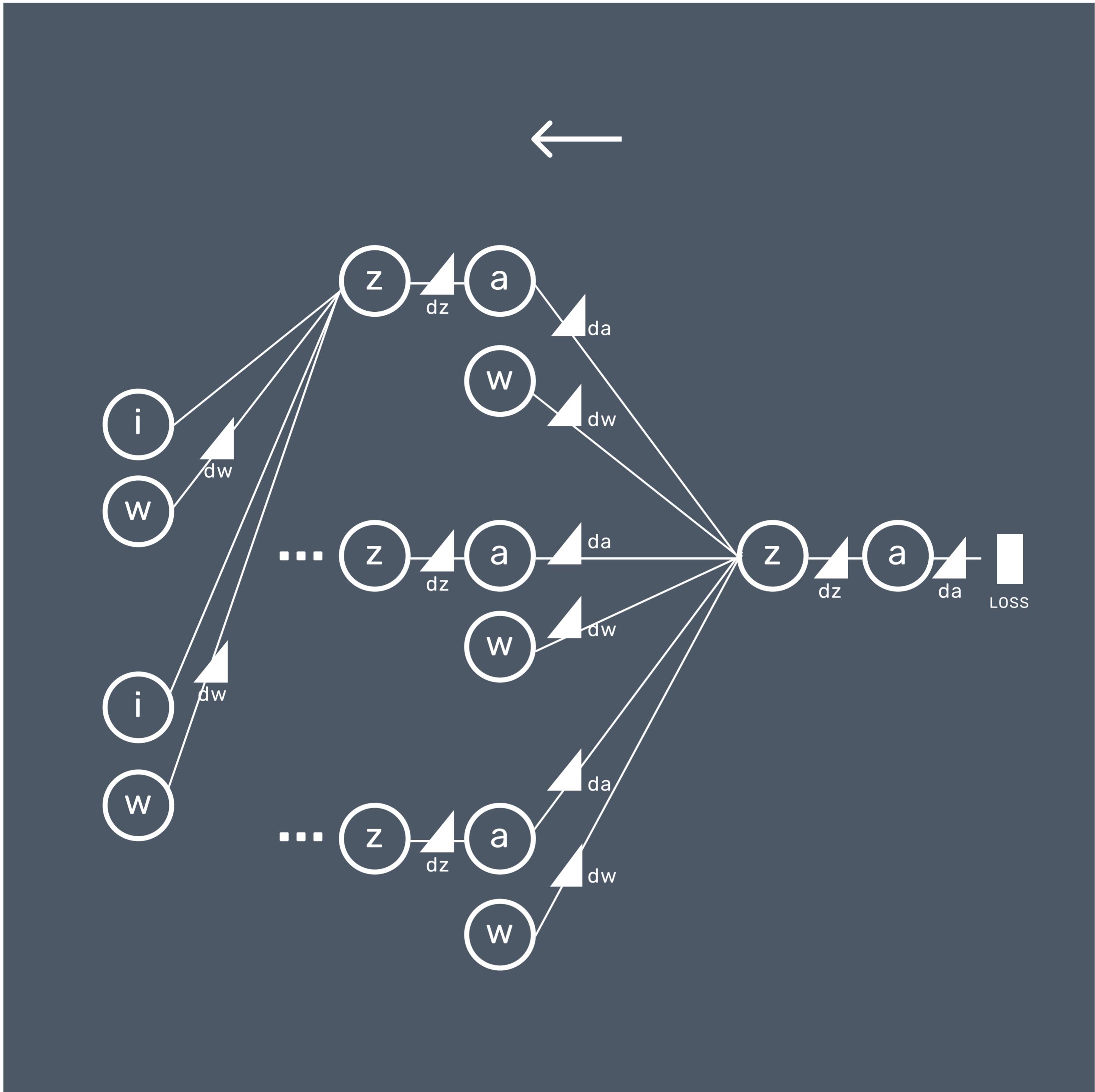


## BACKWARD COMPUTATION GRAPH

Meanwhile, the backward computation graph represents the *feedback* step. Here, we find new terms representing the activation gradient ( $da$ ) and the weighted sum gradient ( $dz$ ). They have been there all along, but for simplicity, were not shown in the earlier examples.

We need these other gradients to arrive at  $dw$ . The concept is called the *chain rule*. We won't cover the math, but the idea is this: We can compute a particular gradient if we know the gradient adjacent to it. Here, we can compute  $da$  from the loss value, which means we can compute  $dz$ , which means we can compute  $dw$ .

In fact, whenever *error* was mentioned in Chapter 1, it was referring to  $dz$ , which is the gradient adjacent to  $dw$ .

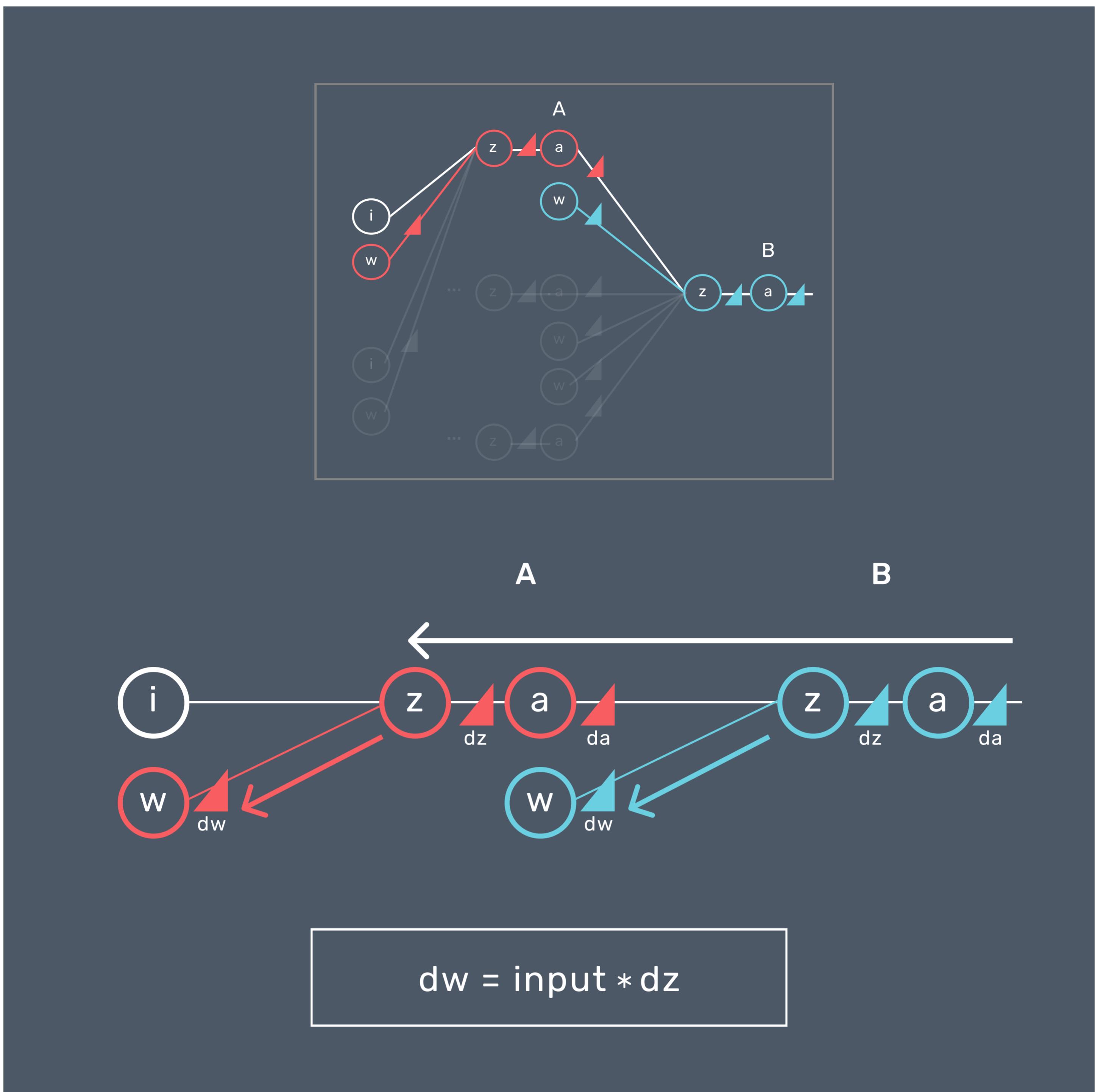


## BACKPROPAGATION

Let's return to this chapter's neural network. The backward computation graph is shown here for all four neurons. Starting from the loss value, information flows back to all neurons so that each weight receives its gradient,  $dw$ .

In deep learning, this process is called *backpropagation*.

This graph looks pretty complex, so let's pick one example.



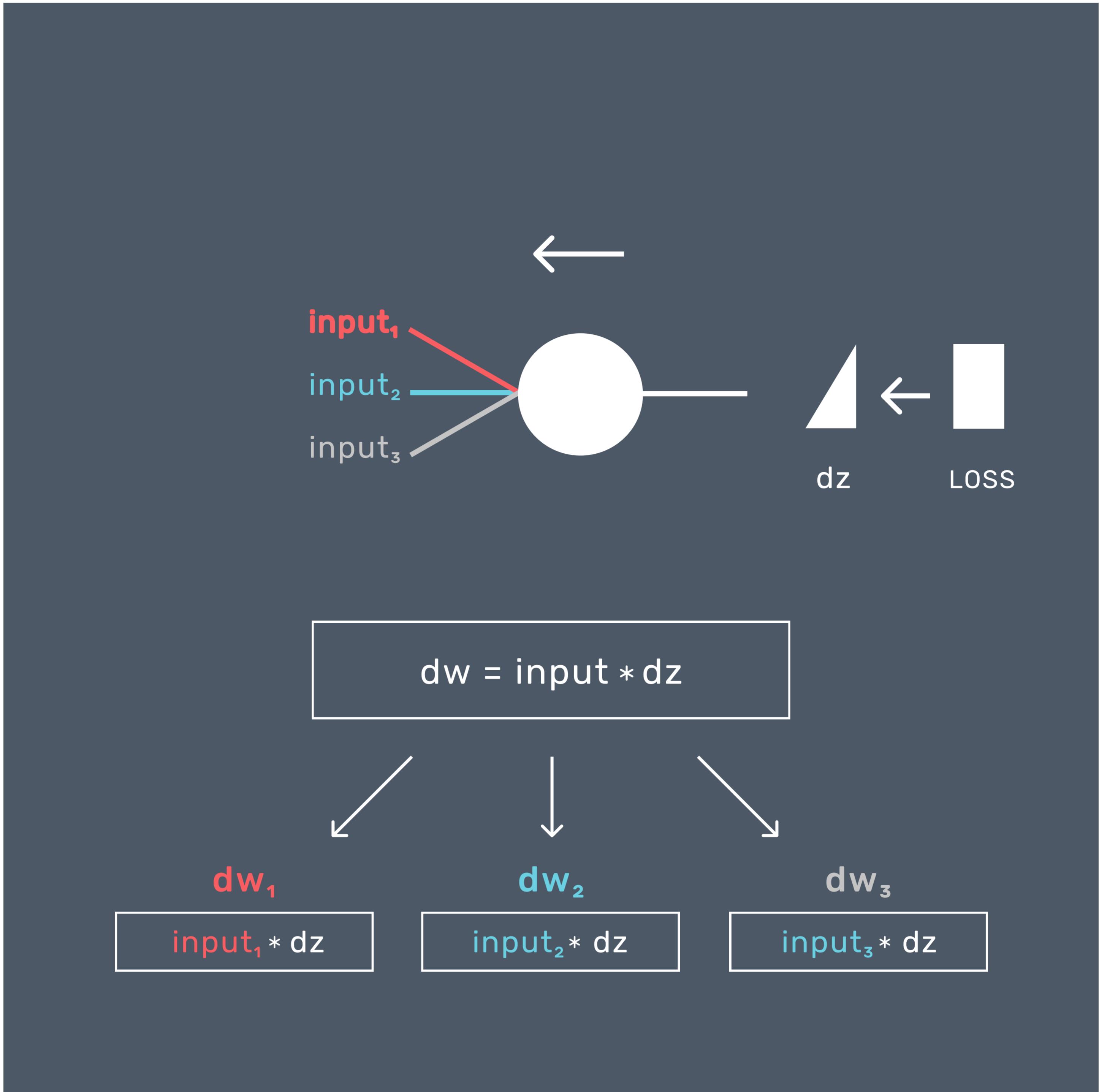
## EXAMPLE

In this example, we'll focus on two weight gradients, red and blue, involving two neurons, A and B.

Tracing the lines along the graph, we obtain the red  $dw$  by multiplying the input data  $i$  by neuron A's  $dz$ .

Meanwhile, we obtain the blue  $dw$  by multiplying neuron A's output (which becomes neuron B's input) by neuron B's  $dz$ .

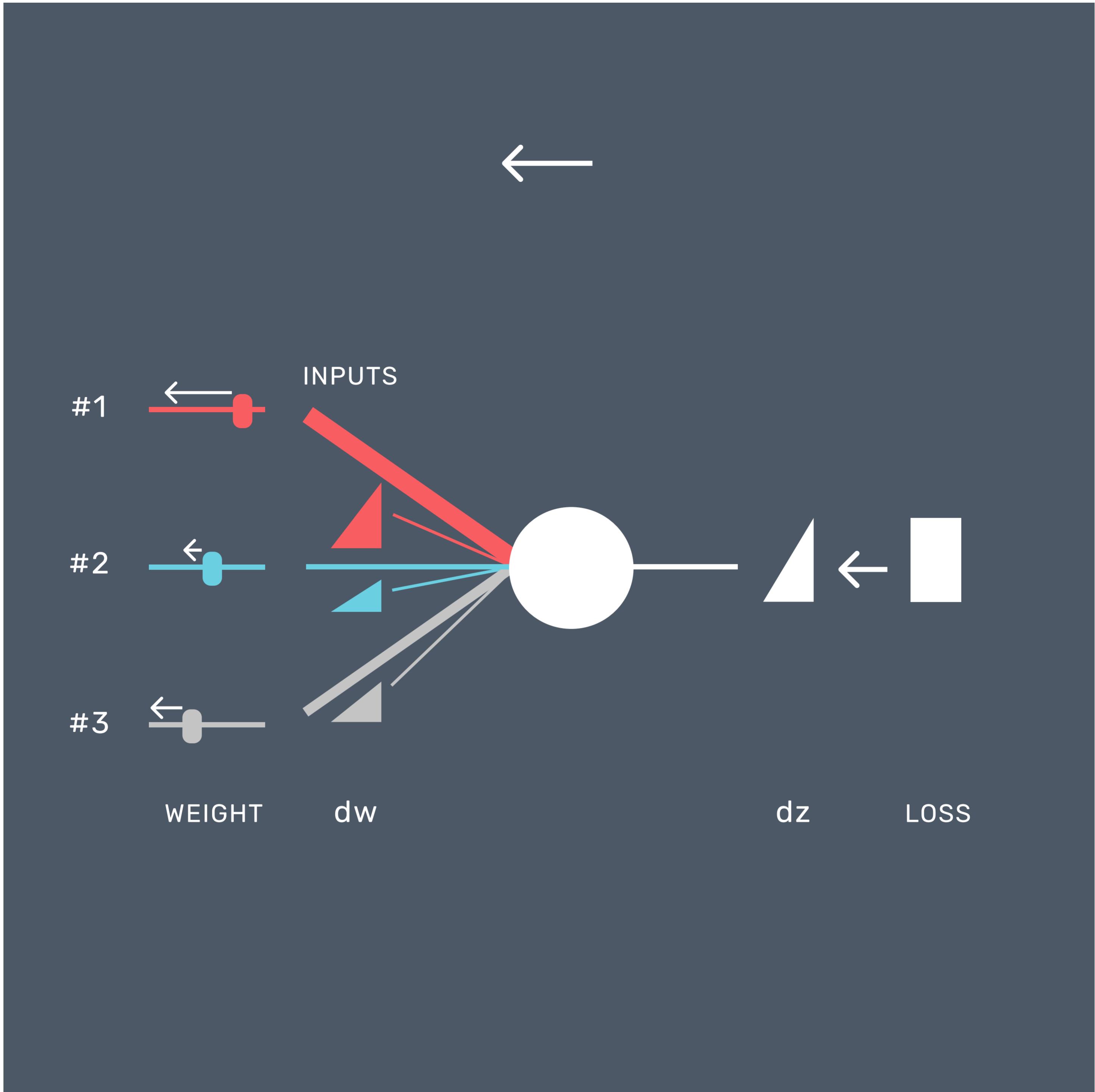
You may be wondering, how did we know the formula for  $dw$  in the first place, and what about the formulas for  $dz$  and  $da$ ? Unfortunately, we won't cover them in this book, but if you are curious, do check out other resources to understand the math derivations.



## MAGNITUDE

Let's take a closer look at the magnitude of the weight gradients, starting with the neuron in the output layer.

Inspecting the formula shows that the larger the input, the larger the corresponding weight gradient. But what does this mean?

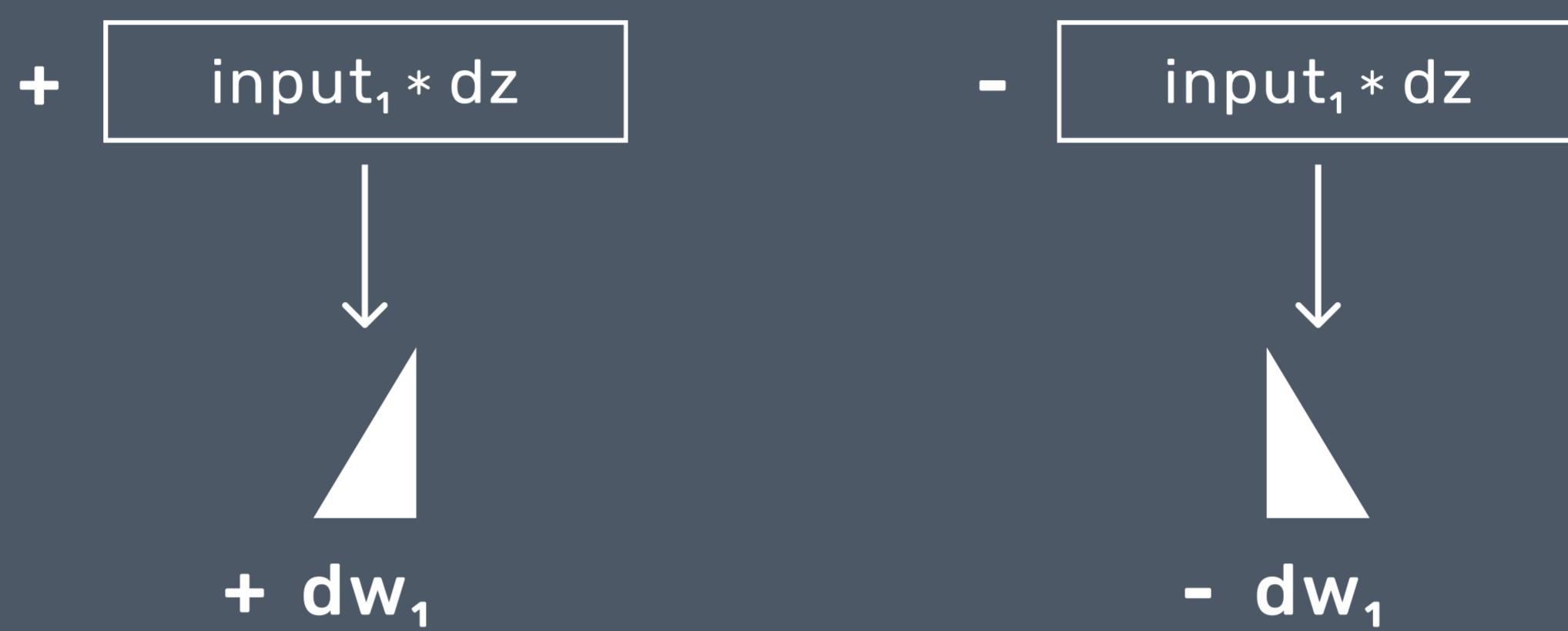


## ERROR CONTRIBUTION

This means that the larger inputs have a greater influence on the output value, and thus on the prediction outcome. Therefore, this formula enables the network to assign the larger weight adjustments to inputs that make the bigger difference.

We can also think of this in terms of error contribution.

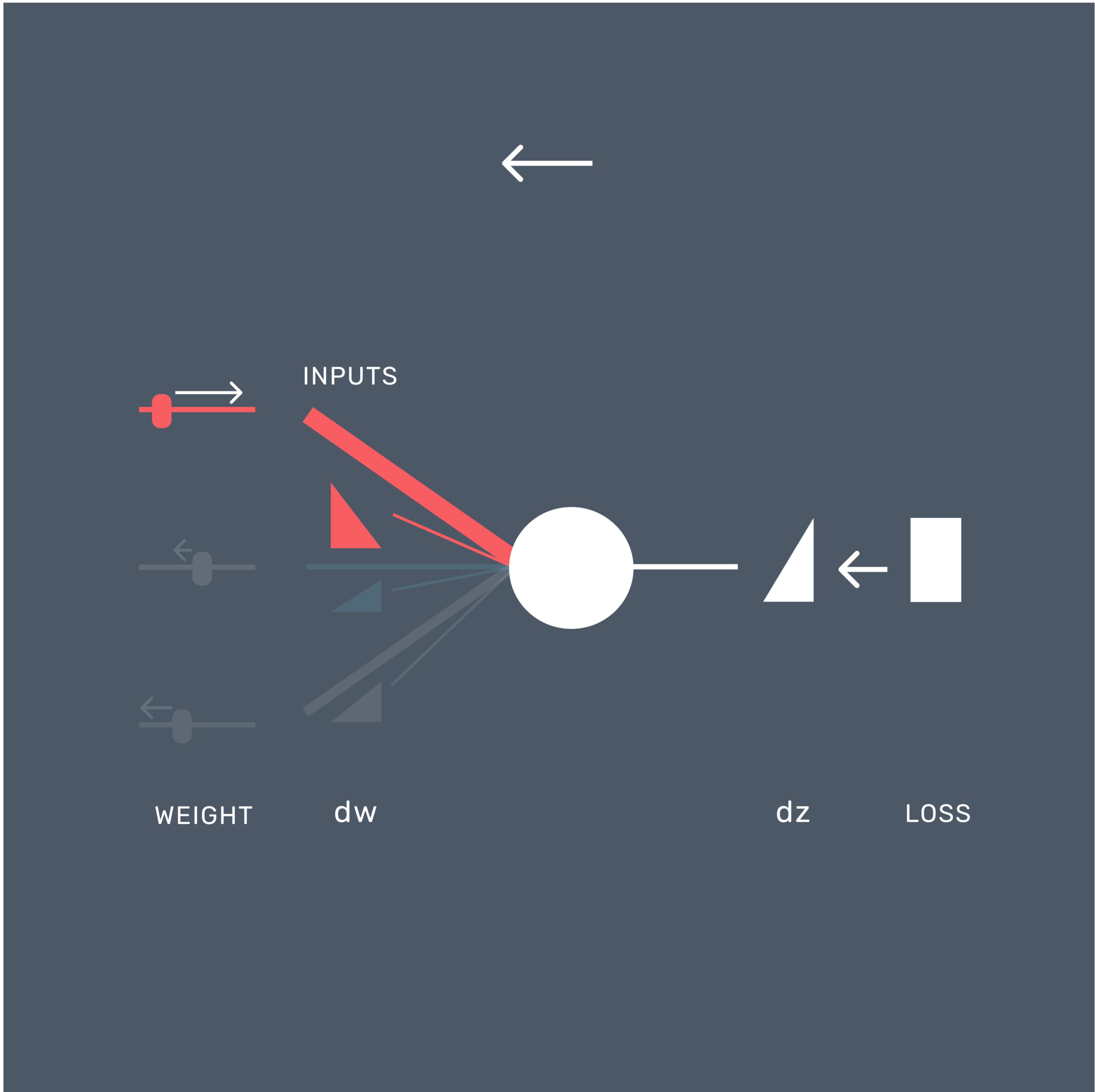
Suppose input #1 is the largest. This tells us that input #1 is the one that contributed the most toward the error. So, we want to tell input #1 to make the biggest weight adjustment, and we do that by giving it the biggest weight gradient.



## DIRECTION

Recall that weight gradients have both magnitude and direction. As the gradients backpropagate, their directions can change too.

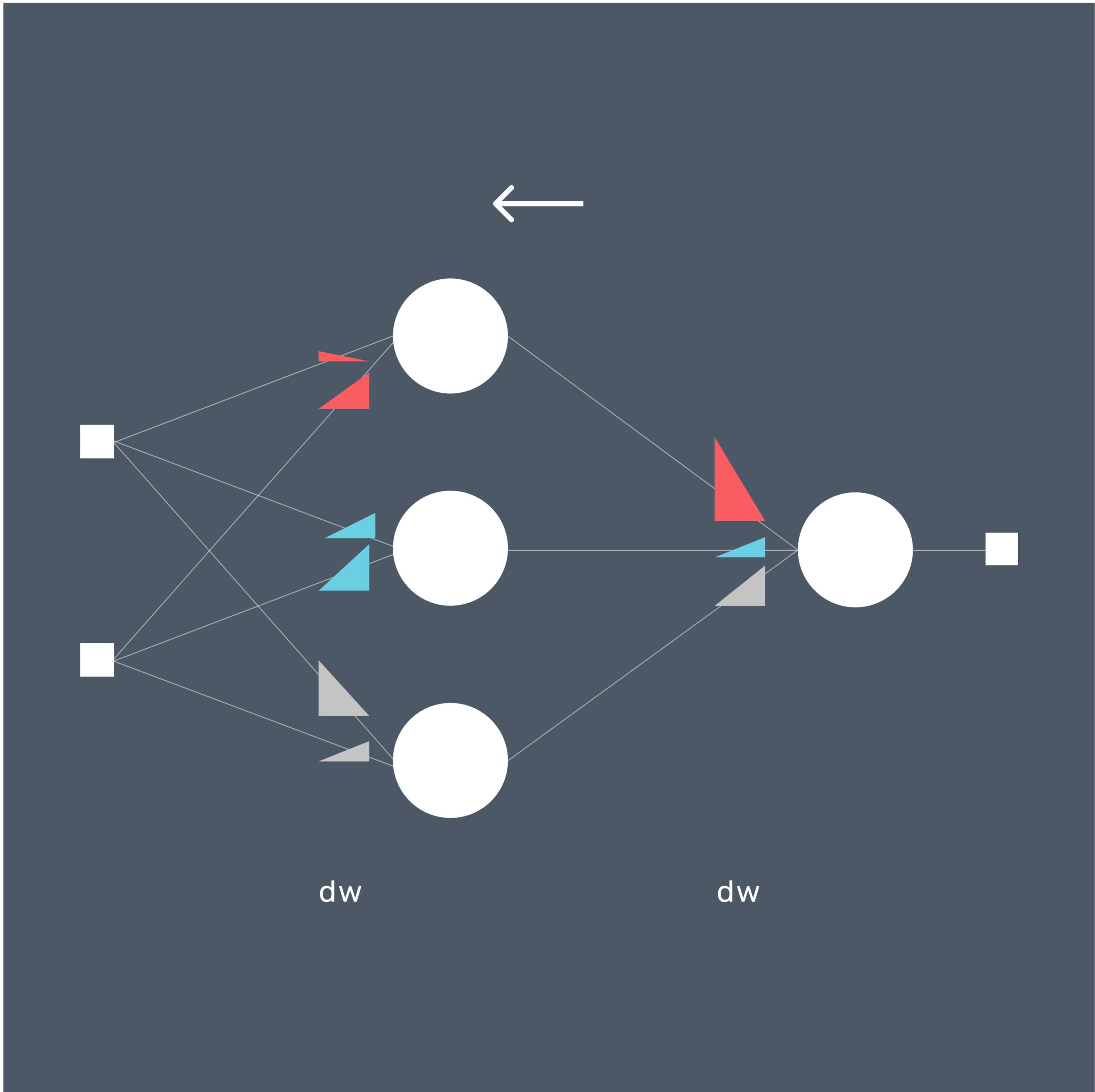
If either the input or  $dz$  is negative, then the weight gradient will be negative.



## NEGATIVE GRADIENT

A negative weight gradient means that the network will need to increase its weight instead.

If you recall the gradient descent discussion in Chapter 1, a negative value causes the gradient to flip its shape.

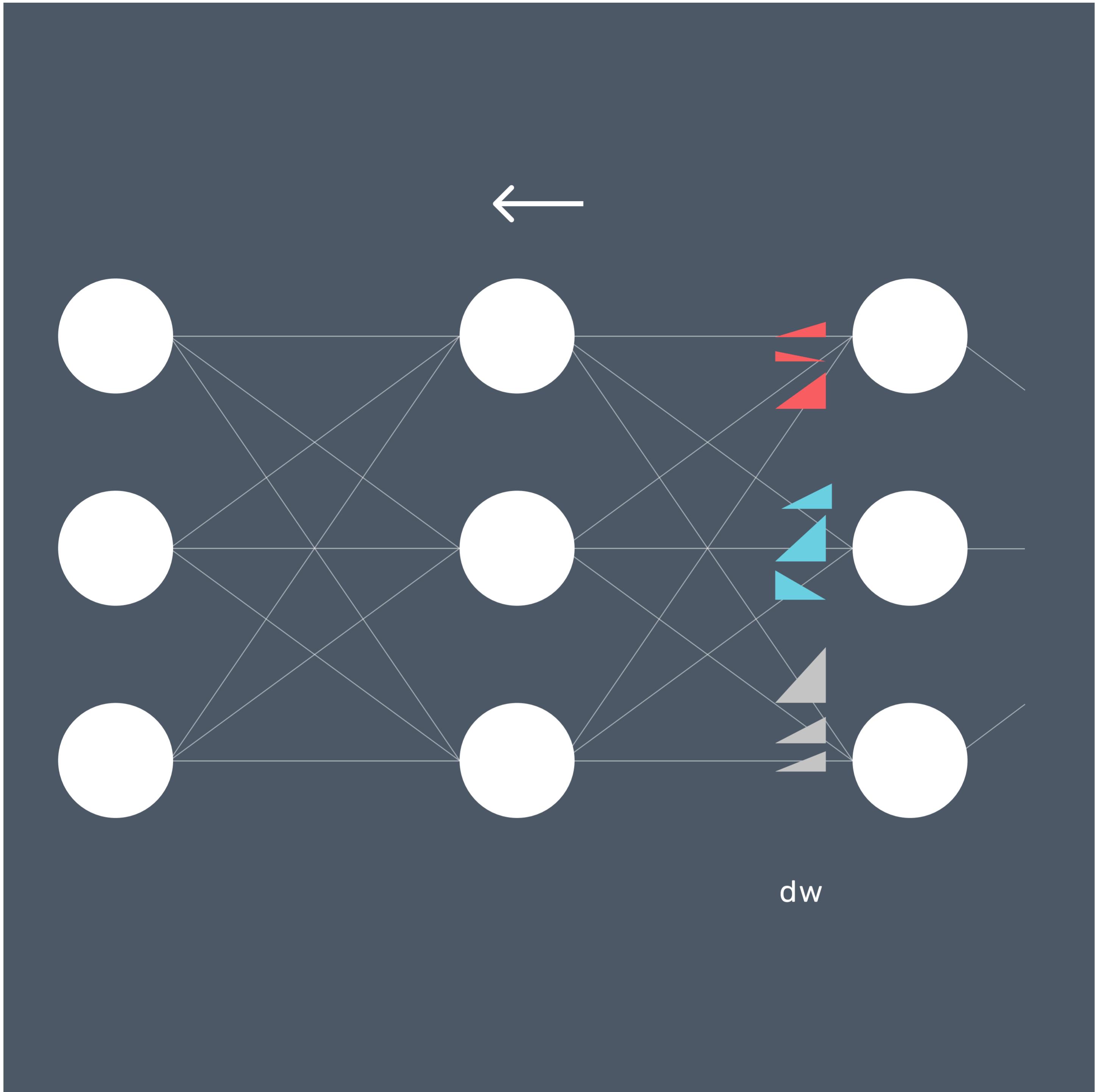


## REPEAT FOR THE WHOLE NETWORK

The same principle applies to the rest of the neural network. Backpropagation continues until all the weights have received their gradients.

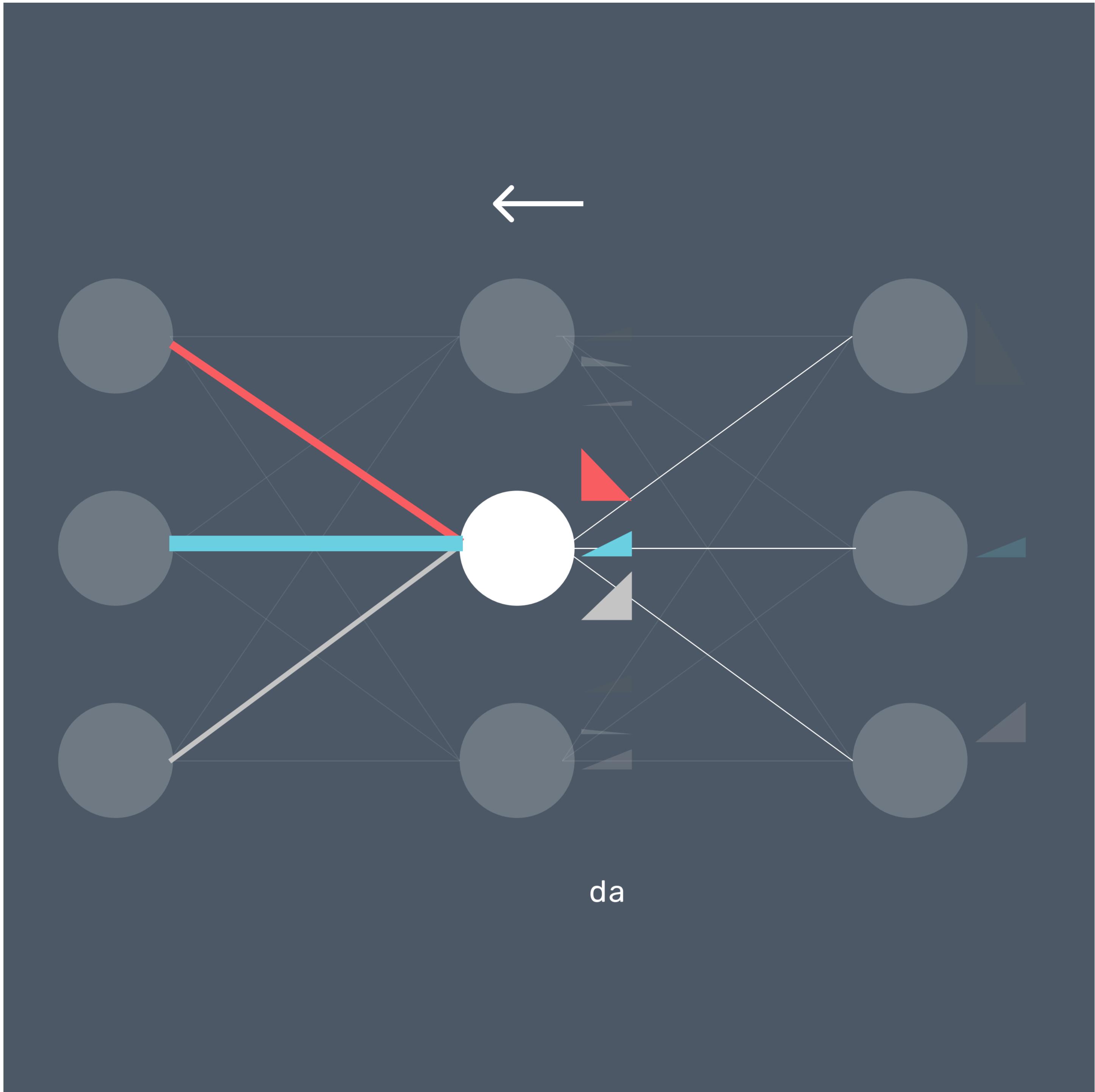
The good news is, in practice, there are well-established deep learning frameworks such as PyTorch and Tensorflow that handle all the tedious computations on our behalf!

Nevertheless, the value of understanding how it all works is immense.



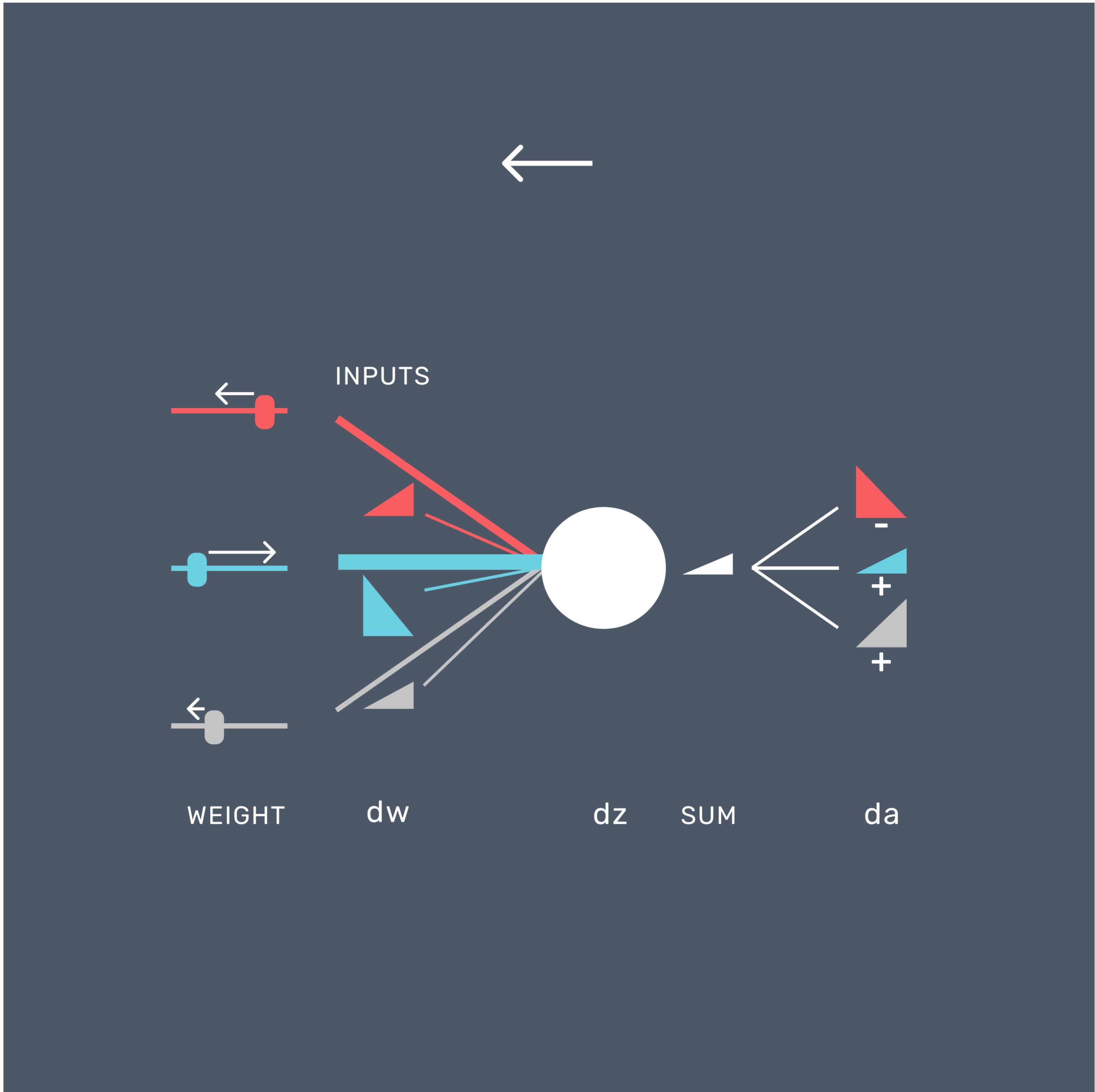
## EXAMPLE WITH A BIGGER NETWORK

Now let's firm up our understanding of how backpropagation works. Suppose we had a larger network with a few more hidden layers.



## EXAMPLE NEURON

Take this neuron in the middle for example. It has just received the  $da$  gradients from its outputs. But what does it do with them?

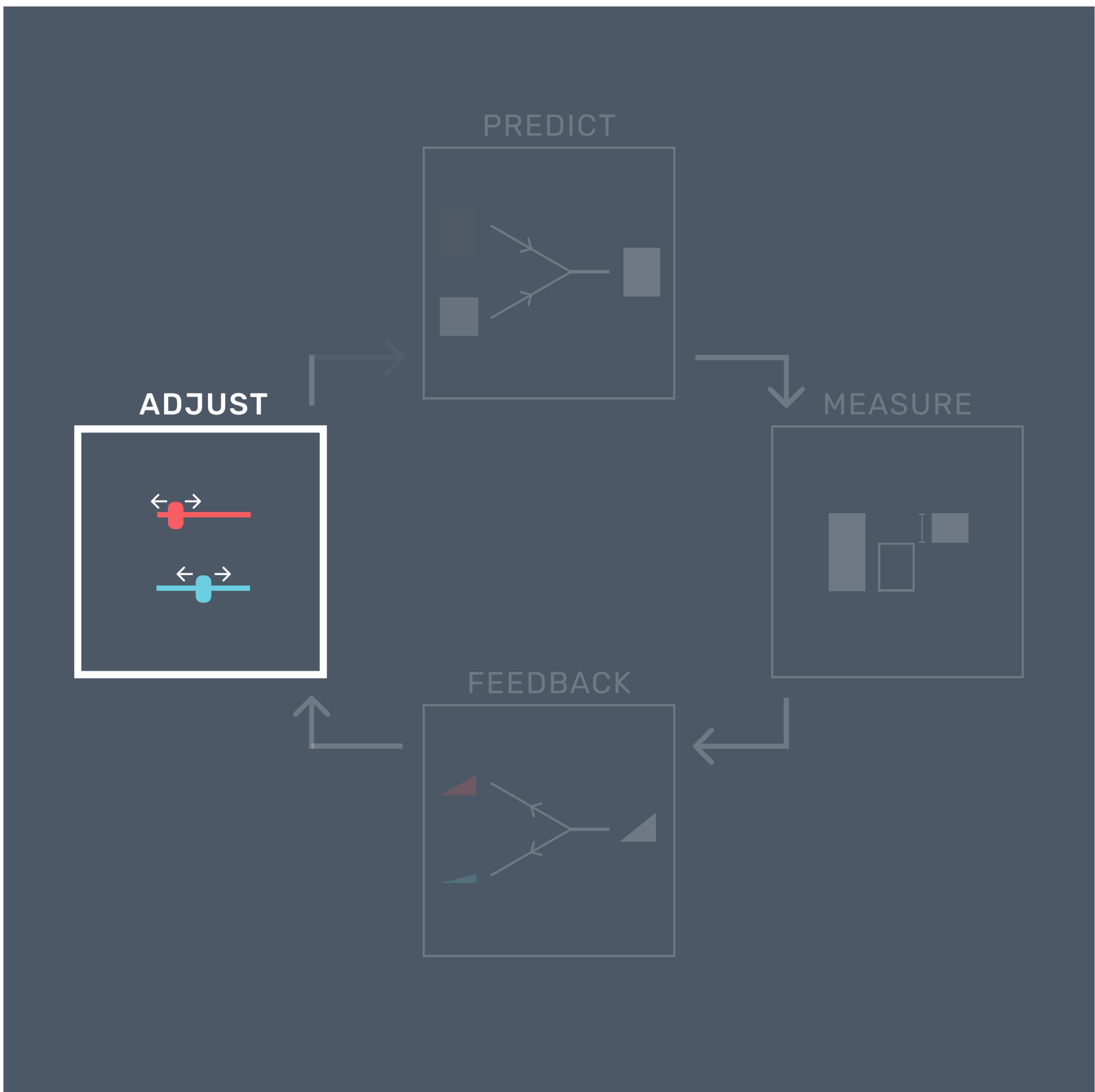


## EXAMPLE NEURON

The first thing it does is to sum up these  $da$  values. This represents the net gradient coming from the three outputs.

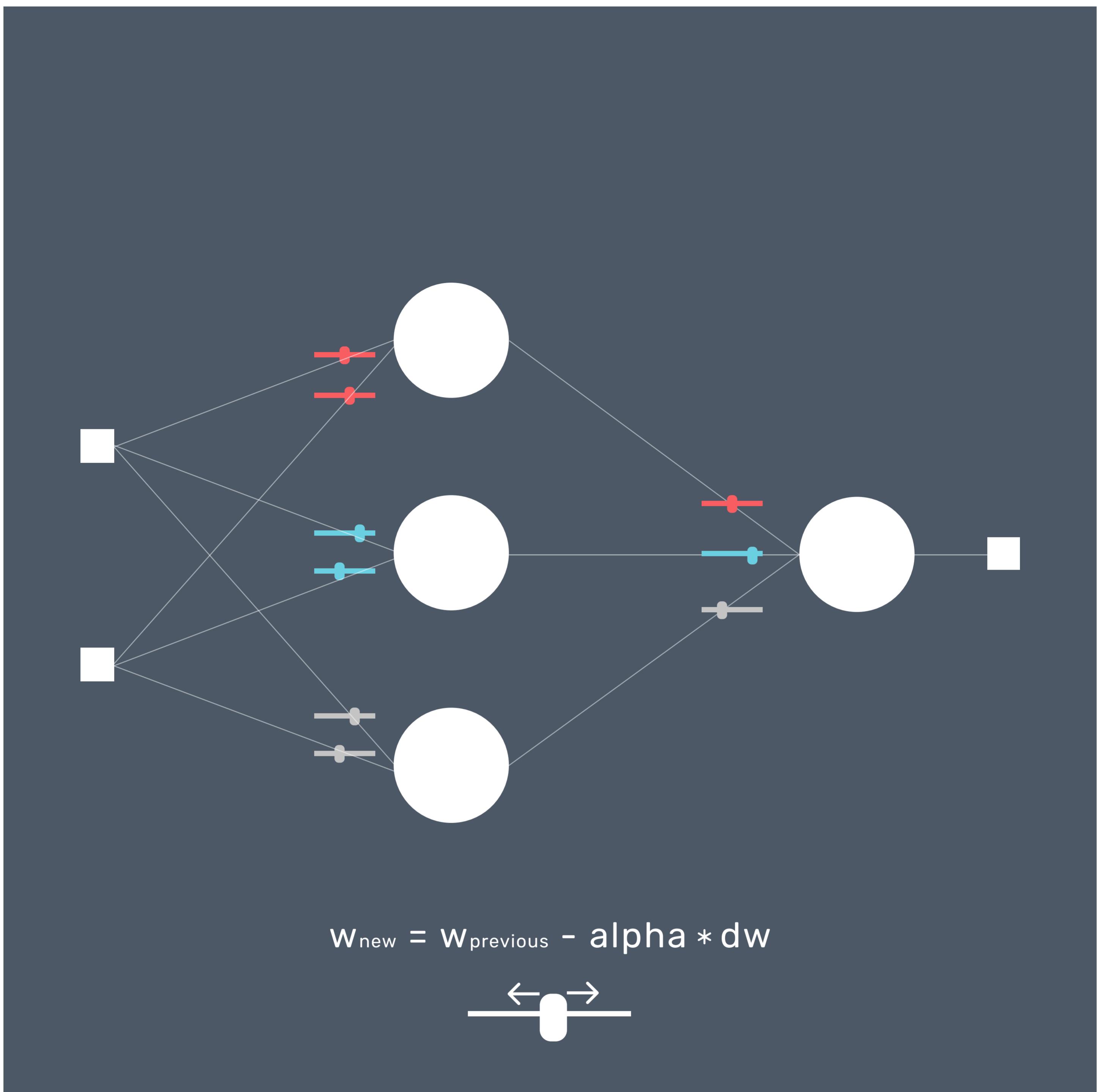
As per the chain rule,  $da$  gives rise to  $dz$ .

And then, we get the  $dw$  for each input by multiplying  $dz$  by the inputs.



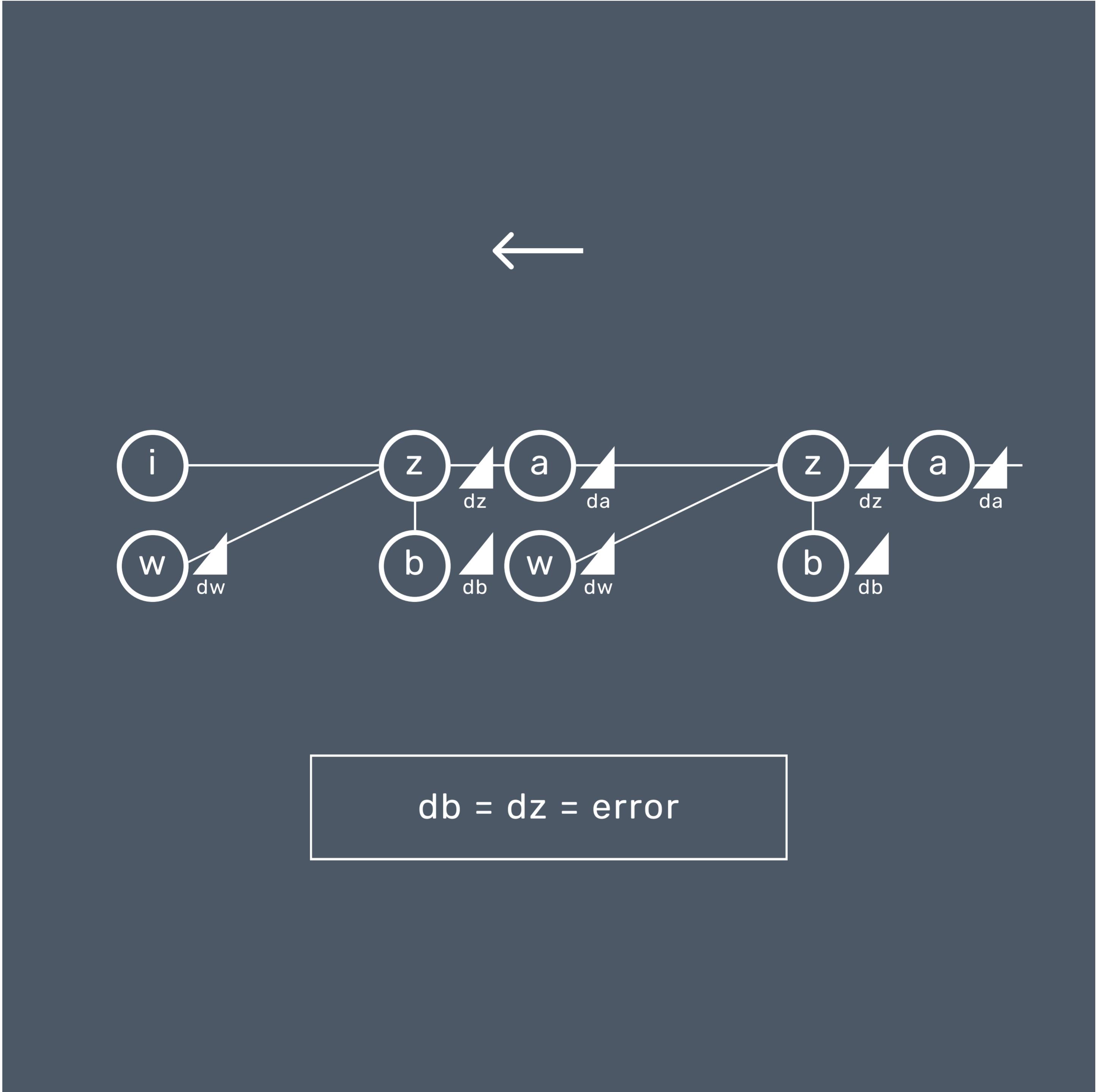
## ADJUST

The fourth step, *adjust*, is where we perform the weight adjustments.



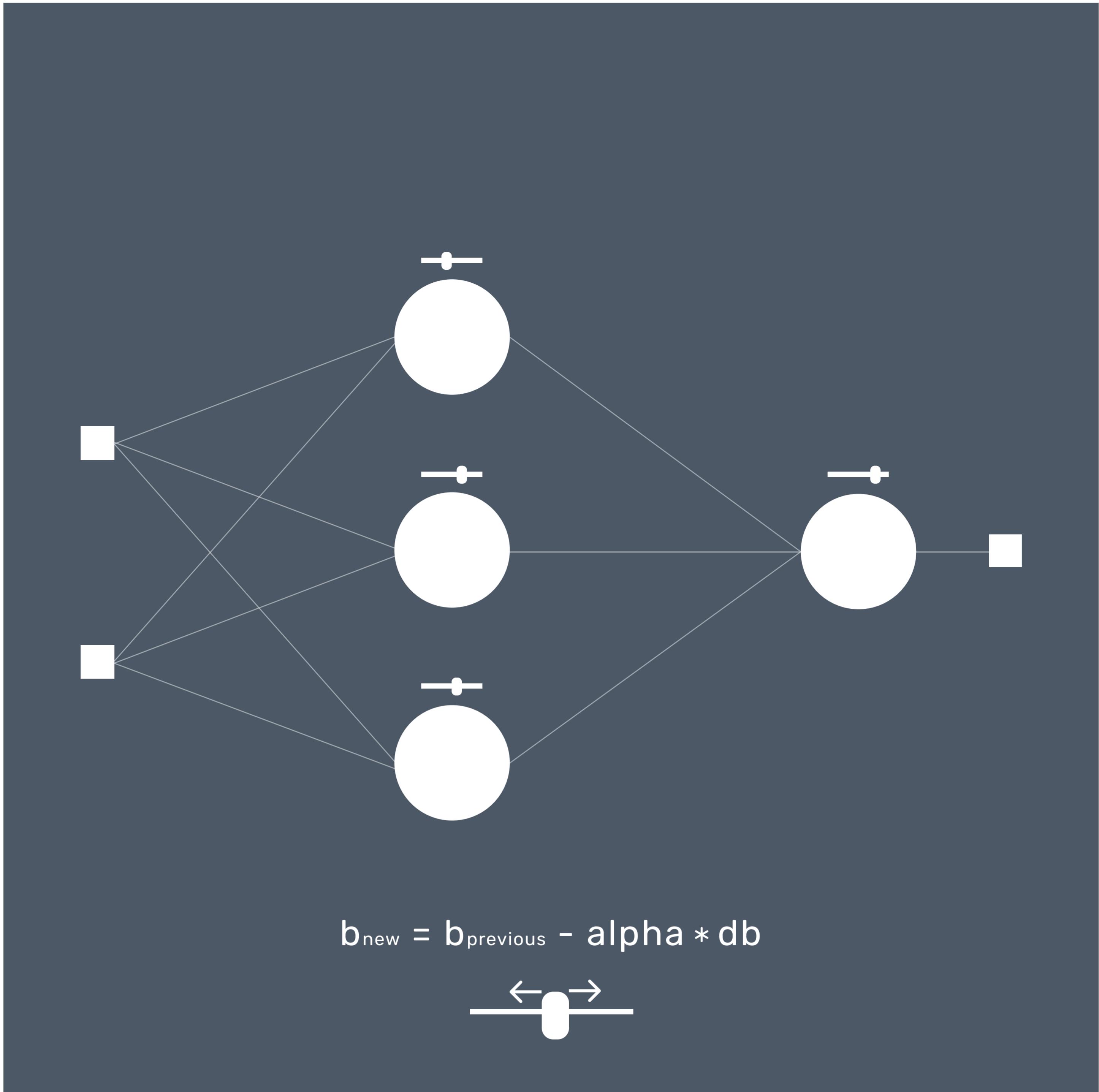
## WEIGHT ADJUSTMENT

As in Chapter 1, we can now adjust the nine weights according to the gradients that they receive.



## BIAS

Let's now bring the bias term back into the picture. As before, the formula for the bias gradient  $db$  is simply equal to the error, which we now know is given by  $dz$ .



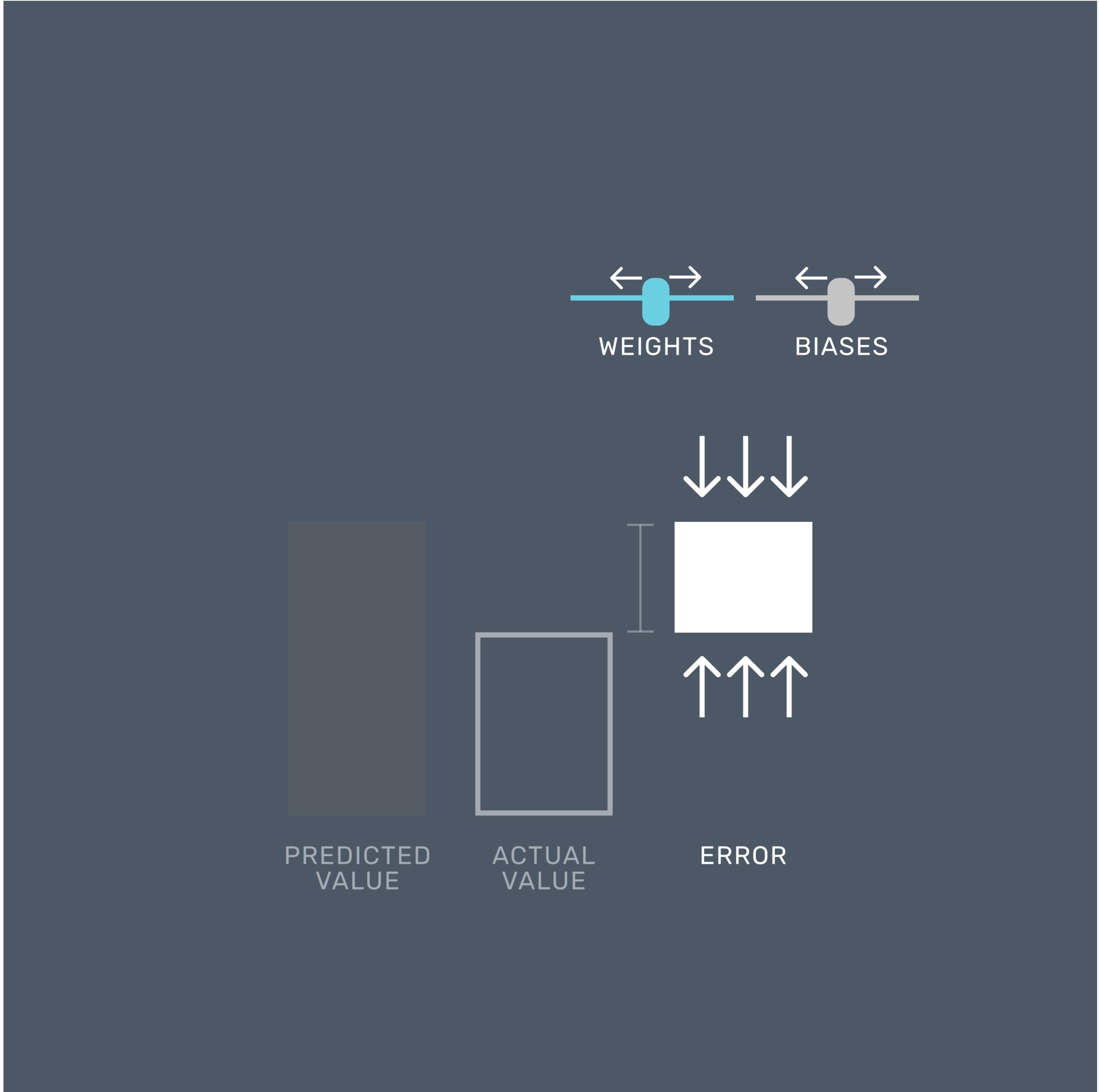
## BIAS ADJUSTMENT

The neural network can now adjust the biases based on their gradients.



## EPOCHS

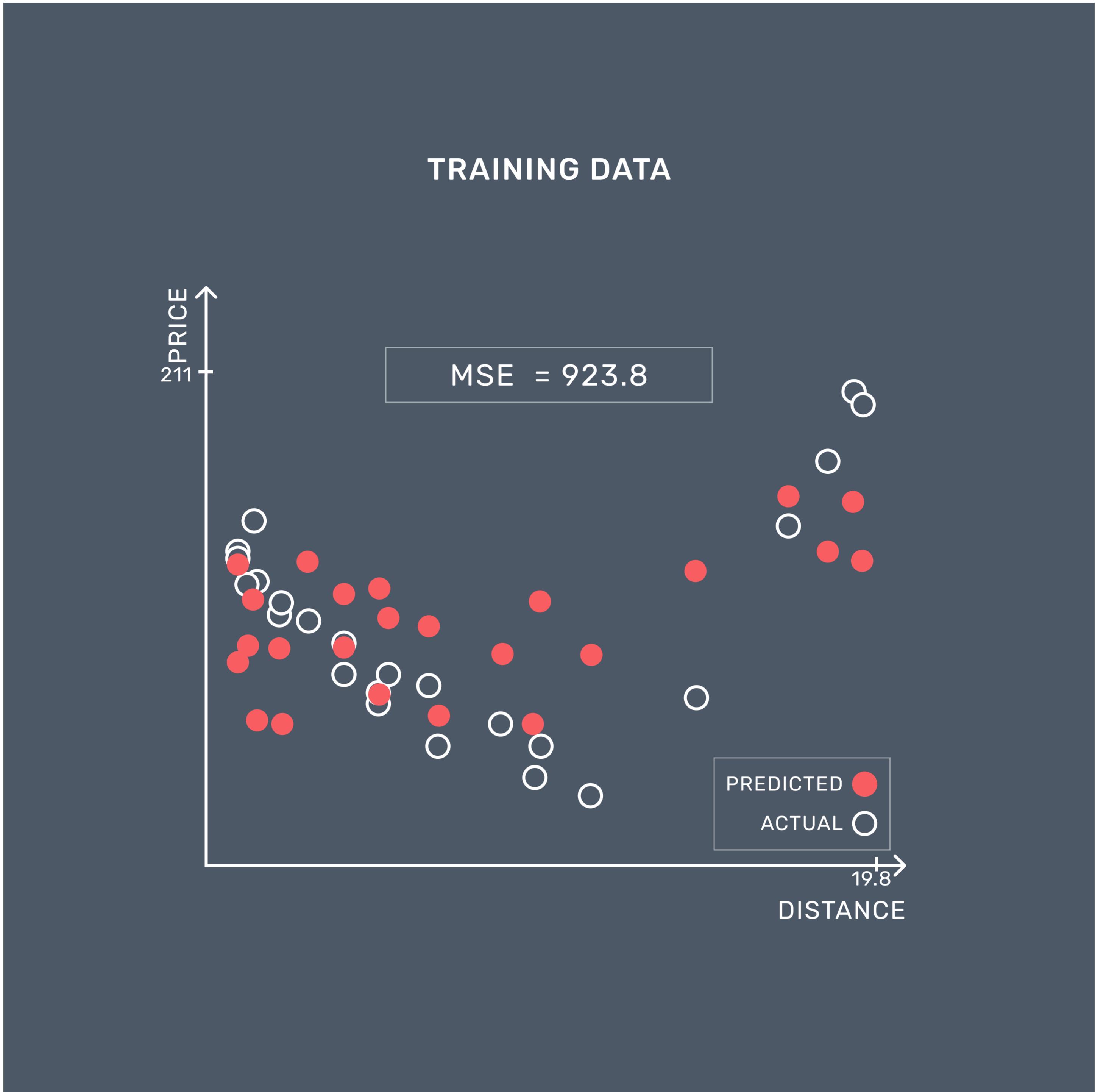
We'll repeat the training cycle for 800 epochs. This task requires more epochs compared to Chapter 1. This is because of the non-linearity, which will take a longer time to learn than a linear case.



## REVISITING OUR GOAL

With all the details, it's easy to lose sight of the goal. It's a good time to remind ourselves that all this boils down to finding the optimal weights that give the most accurate predictions.

Now that training is complete, the neural network should have learned enough to produce decent predictions.



## TRAINING PERFORMANCE

However, this doesn't seem to be the case. The MSE of the training data is very poor. Meanwhile, when plotted, the predictions seem as good as random. It appears that the neural network hasn't learned much!

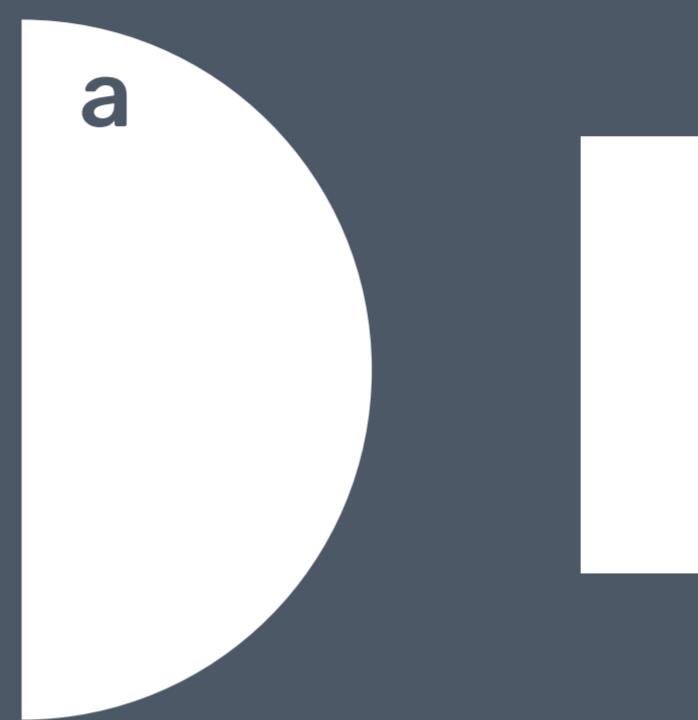
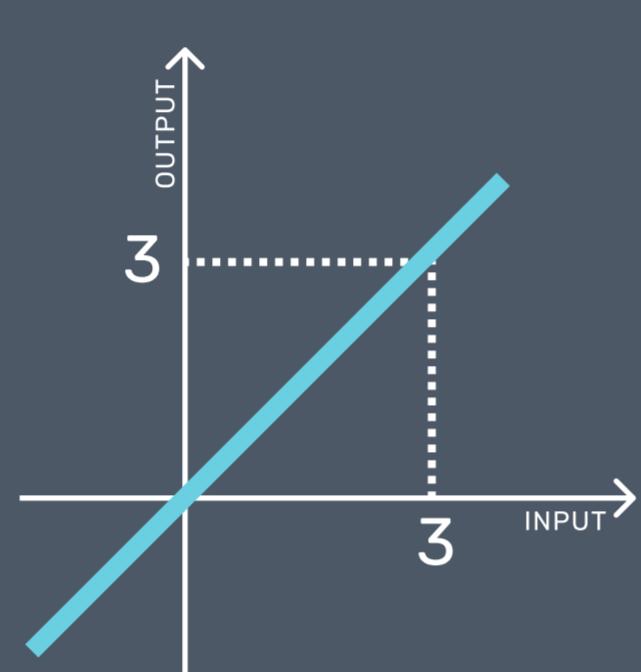


# TEST PERFORMANCE

The same is true with the test data. The predictions don't seem anywhere close to the actual values.

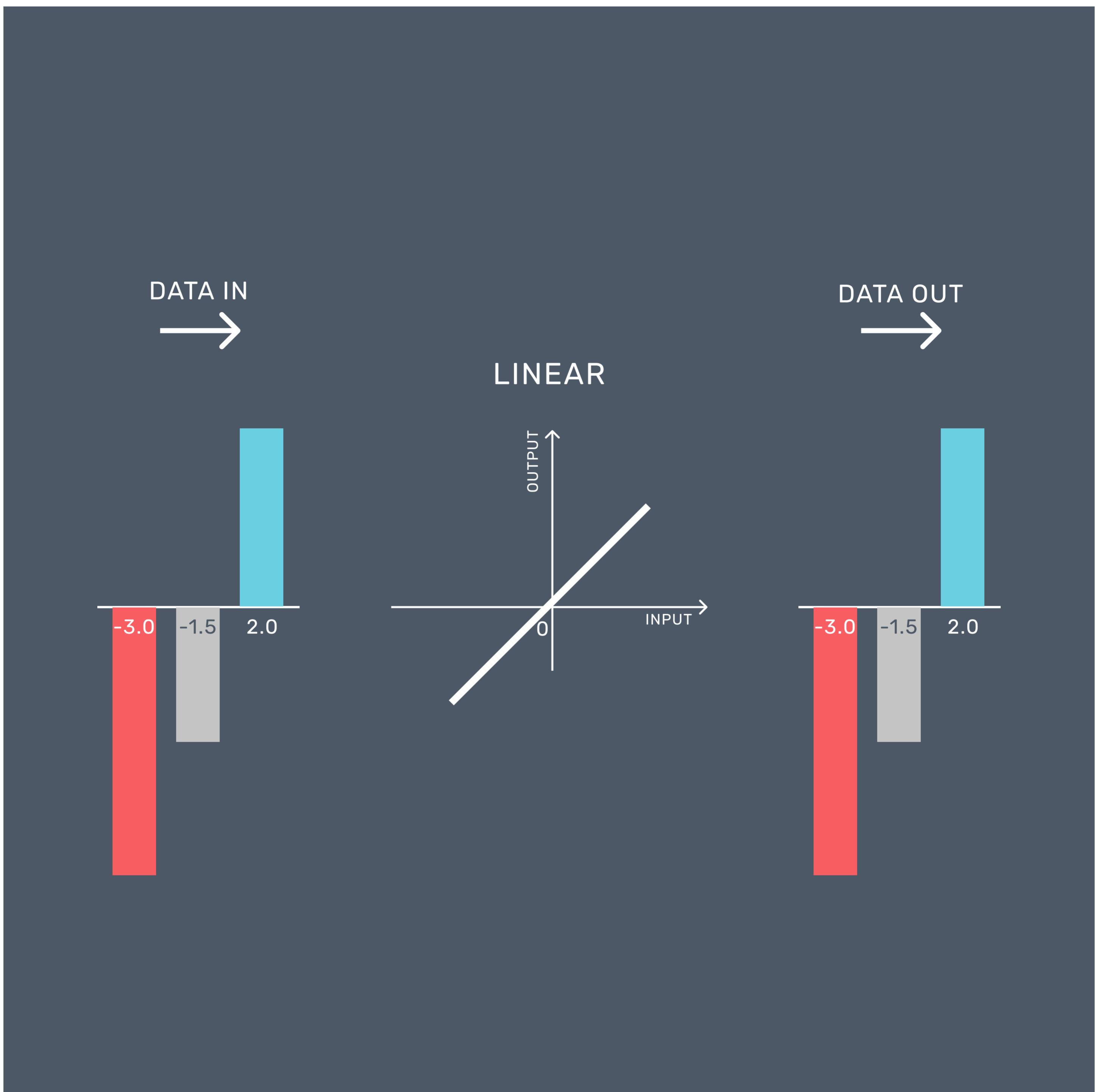
# So, where did this go wrong?

## LINEAR ACTIVATION



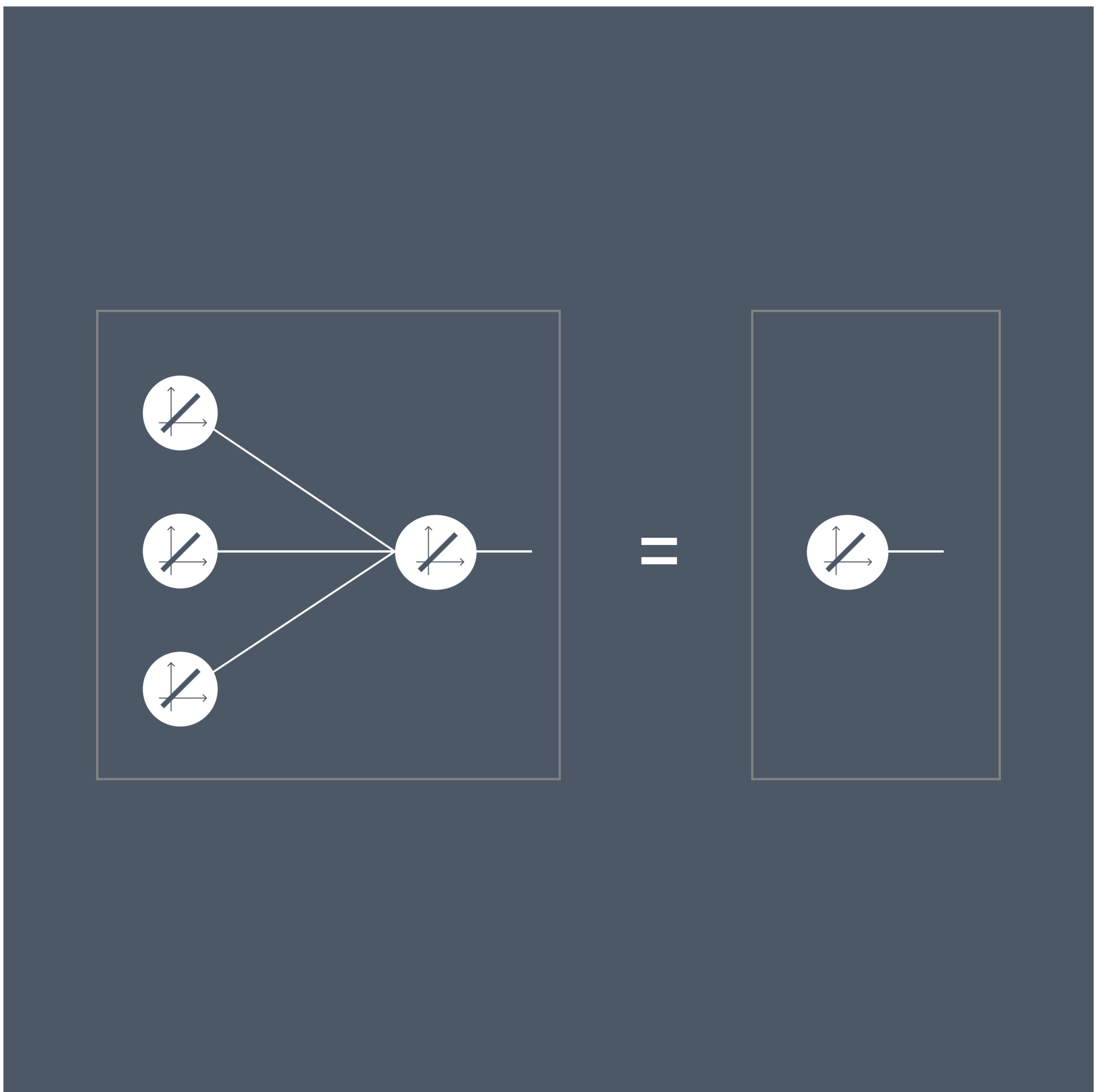
## ACTIVATION FUNCTION

The answer lies in the activation function that we used. All neurons were using linear activation.



## LINEAR ACTIVATION

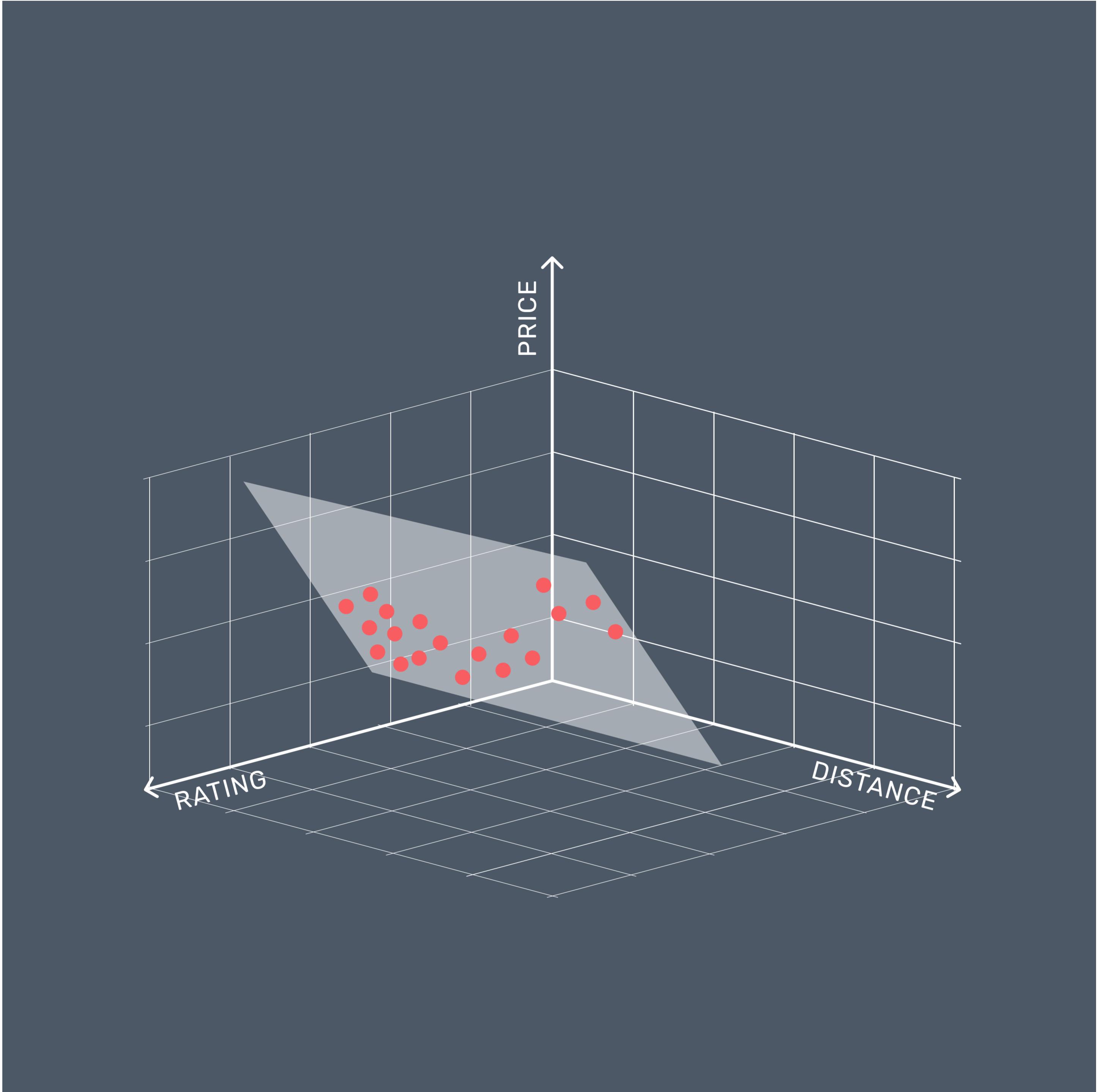
As we've seen, a linear activation function returns exactly the same value of the input it receives.



## LINEARITY

It can be shown mathematically that if all neurons in the hidden layer are using linear activation, they make no difference to the output. It doesn't matter how many neurons there are. It is effectively the same as having no hidden layer, which is equivalent to what we had in Chapter 1.

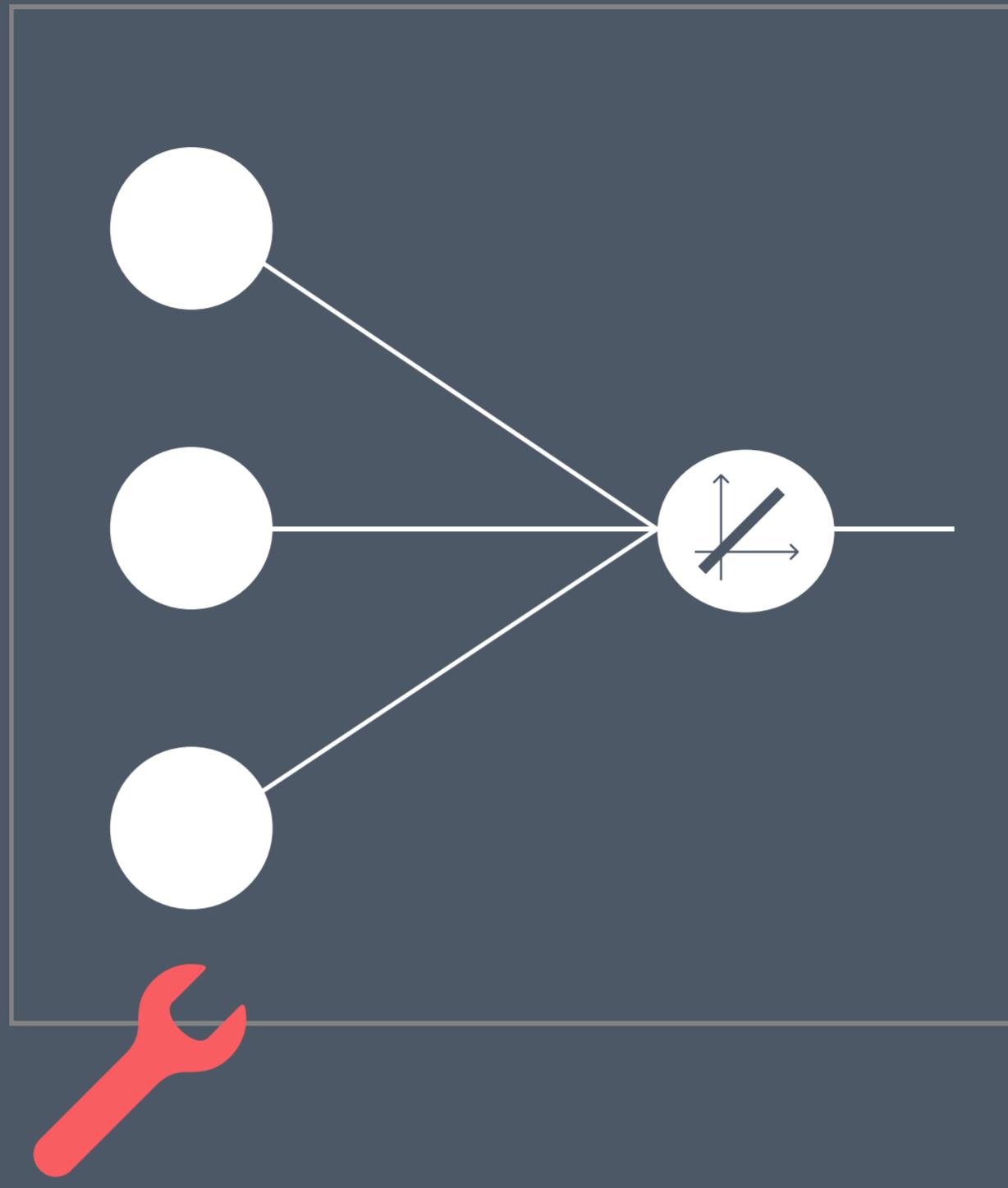
For this reason, our neural network was unable to capture the non-linearities in the data.



## LINEAR PLANE

Returning to our earlier plot of training predictions, it may not appear that the results were linear.

But indeed, they were. If we were to plot the predictions on a 3D chart, we would see them falling on a linear 2D plane.

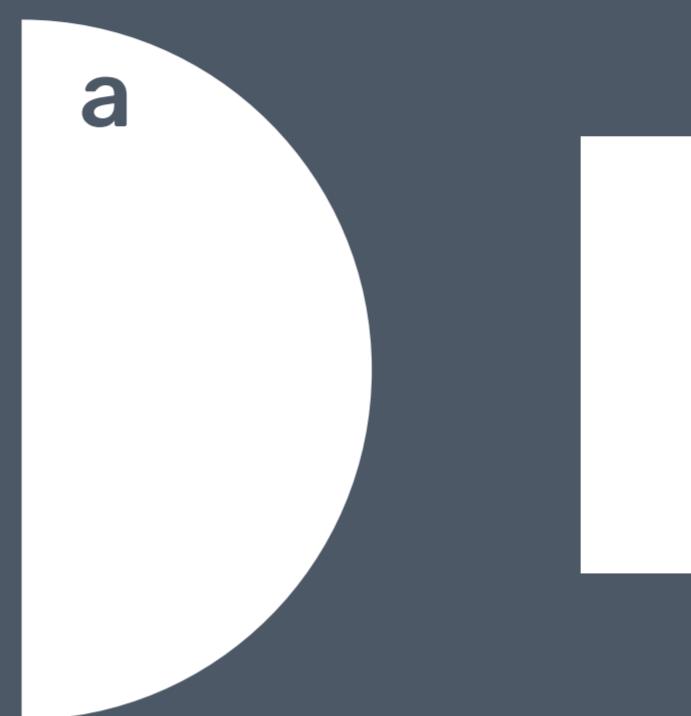
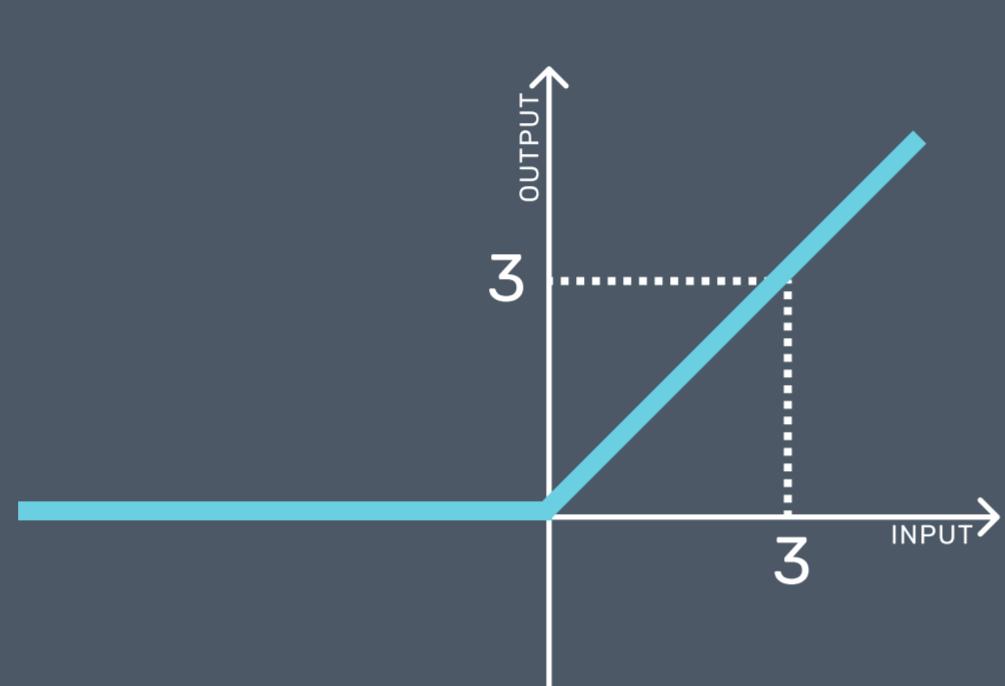


## NON-LINEARITY

The solution to our problem is to introduce non-linearity in the hidden layers. With non-linearity, having hidden layers now makes a huge difference.

The more layers and neurons the neural network has, the more complex relationships in the data it can capture.

## RELU ACTIVATION

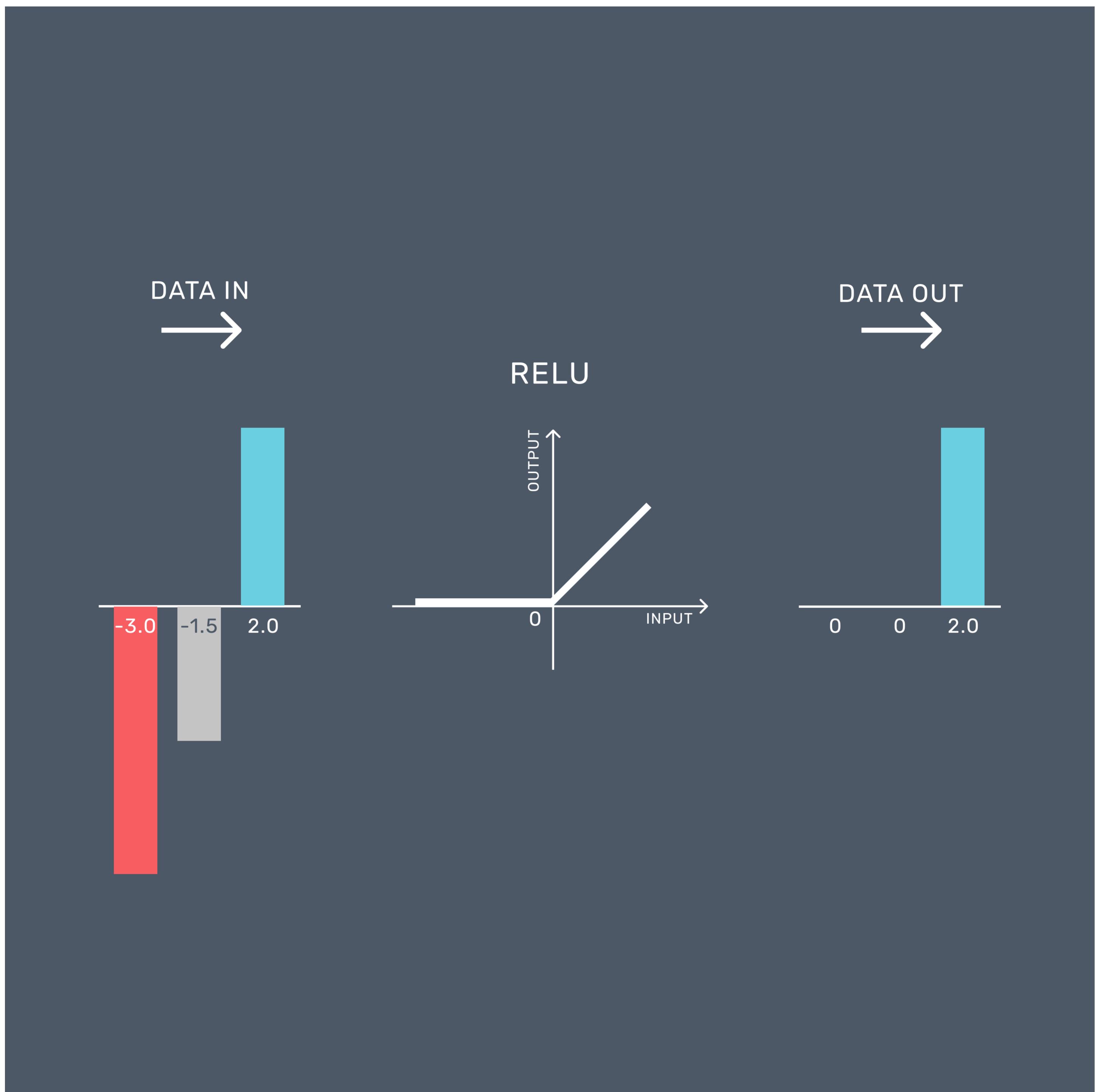


## RELU ACTIVATION FUNCTION

To add the much-needed non-linearity, we turn to an activation function called the *rectified linear unit (ReLU)*. It is an activation function used widely in neural networks today. It's a simple change from the linear function, but it serves its purpose and does its job very well.

For any positive input, the ReLU activation outputs the same value, just like the linear activation.

The difference is for the negative inputs. If the input is negative, the ReLU activation will output zero.

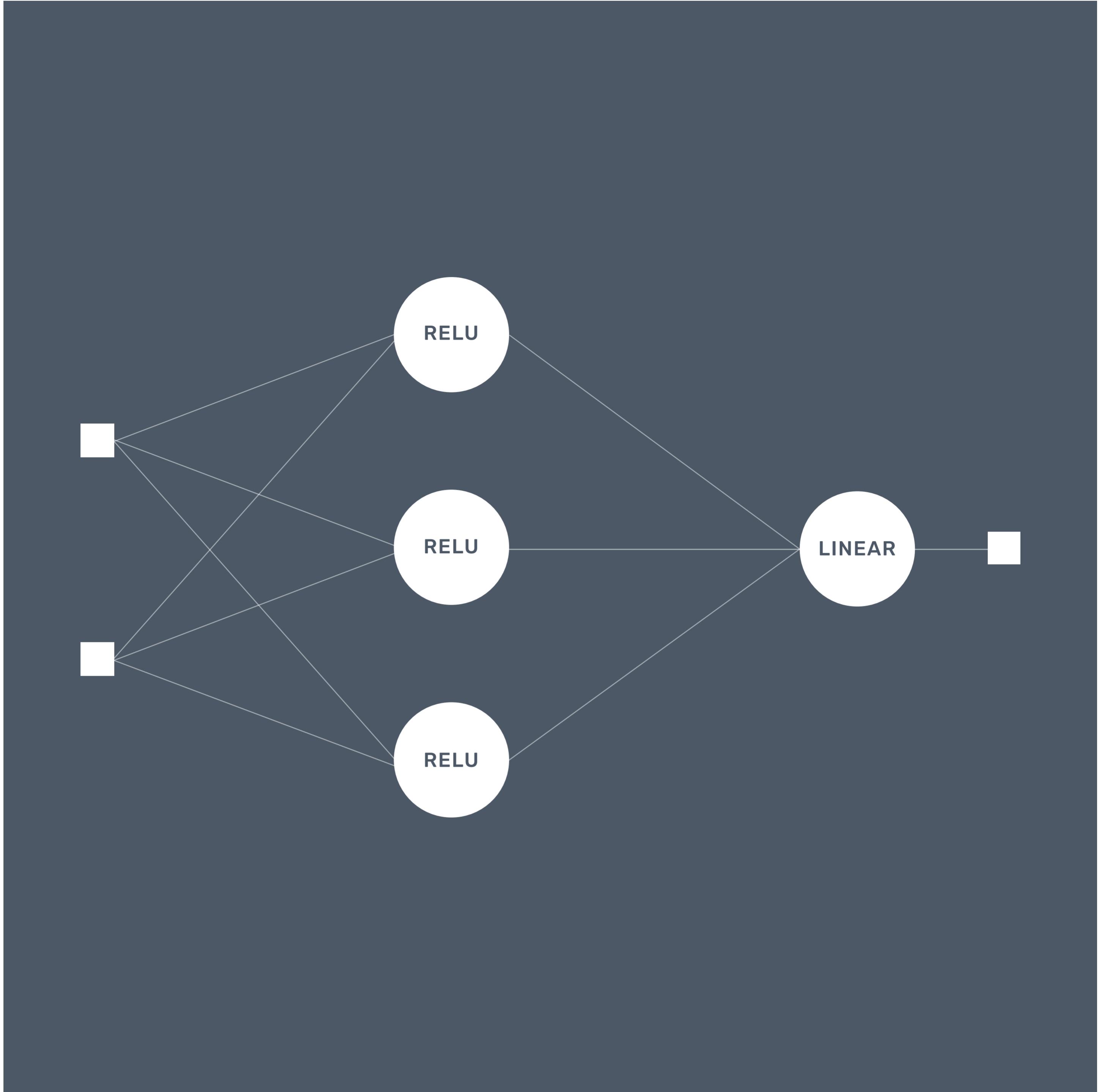


## CHOICE

The ReLU effectively works like a gate - it turns on whenever the input is above zero and turns off otherwise. As simple as it may look, it enables the neuron to have a pretty powerful ability—a choice.

With linear activation, it doesn't matter whether it thinks a piece of information is important or not, it just lets it through.

But with ReLU, every neuron can make a difference. It can now choose to only ‘activate’ when the input is positive.



## RECONFIGURE

We'll now replace the linear activation functions in the hidden layer with ReLU.

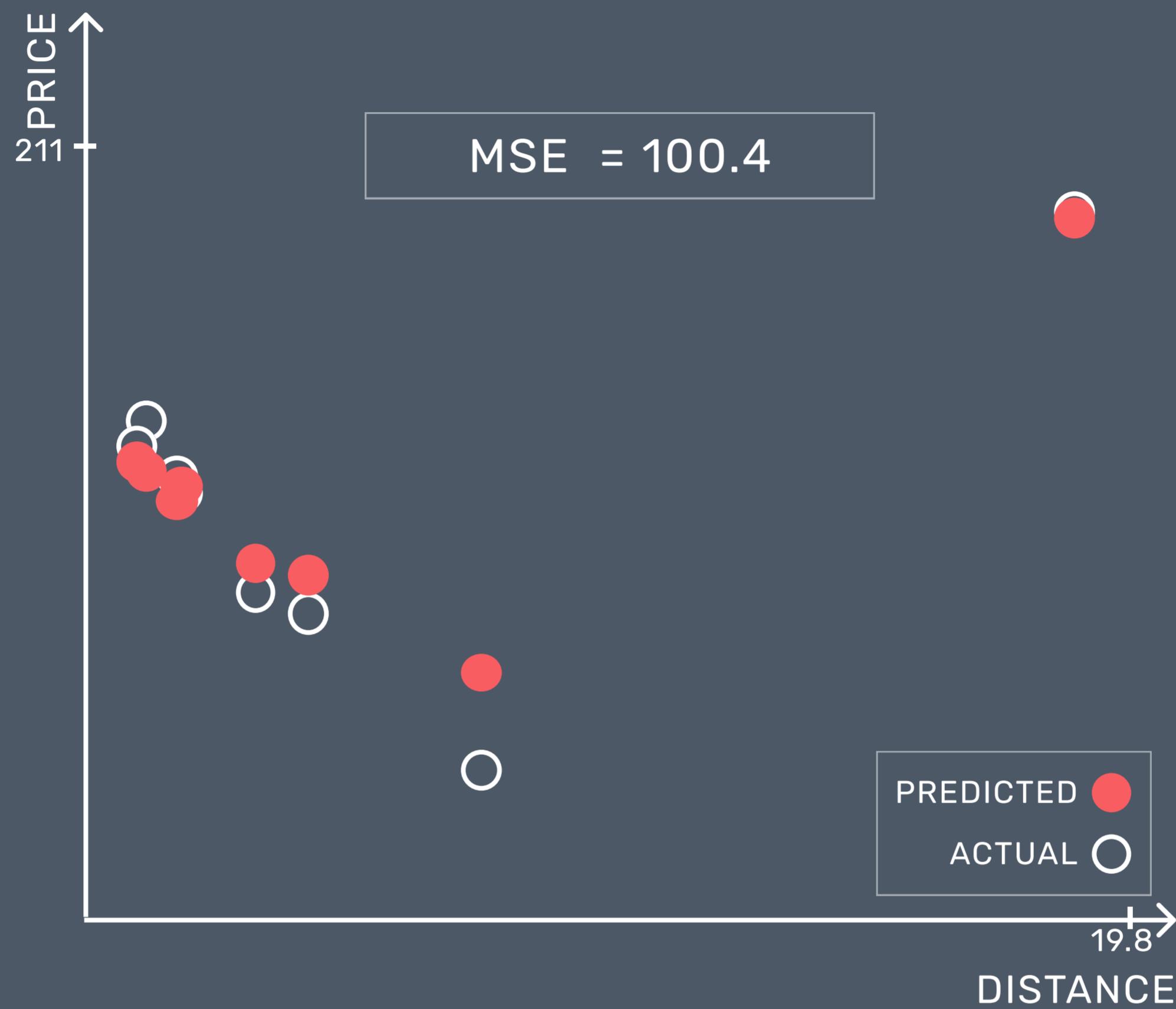
In the output layer, it's fine to keep the linear activation for this task. In the next chapter, we'll see a task where we do need to change the activation function in the output layer.



### TRAINING PERFORMANCE

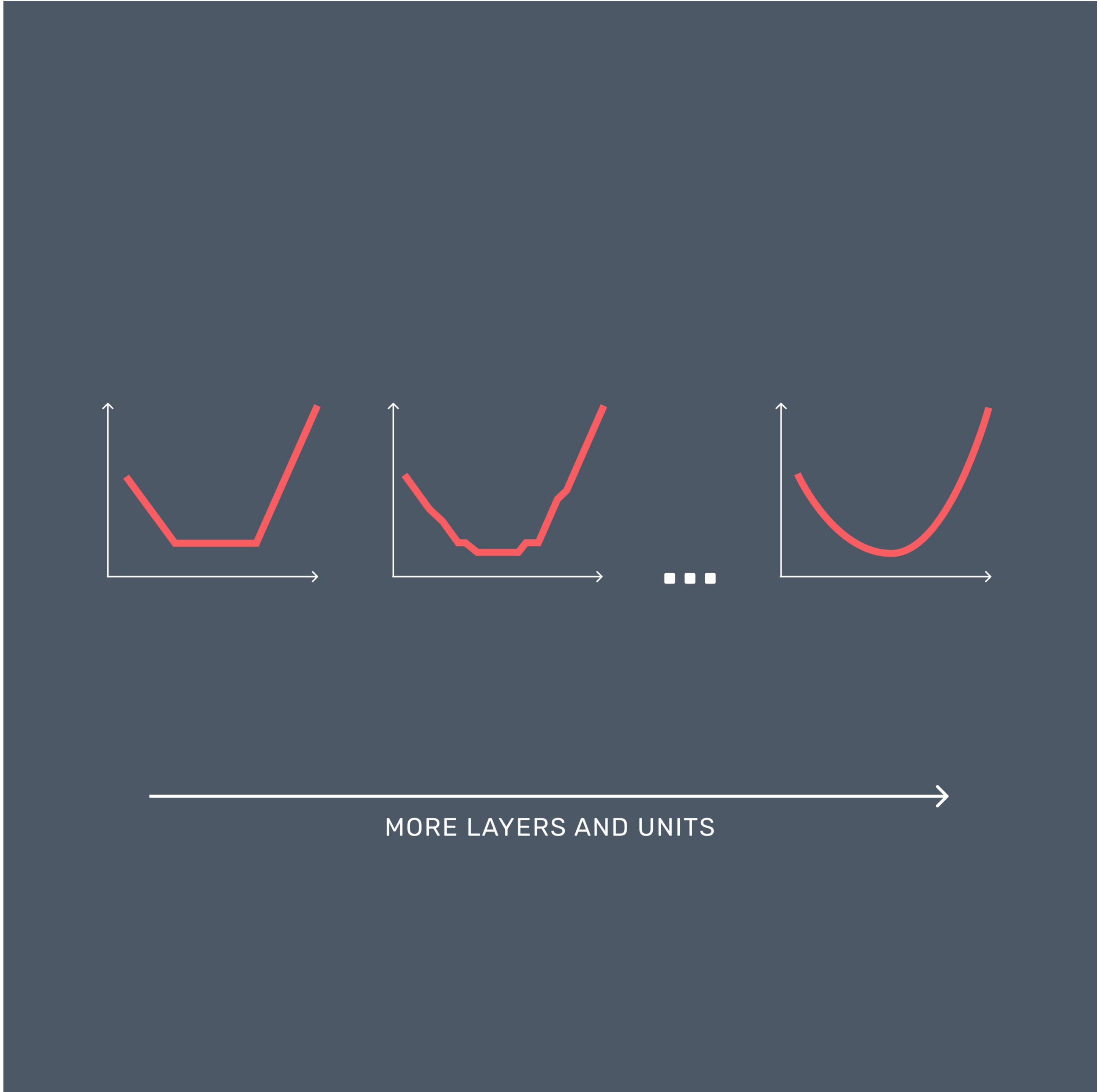
This time, it looks like we are heading in the right direction. The MSE of the training predictions is much better now.

## TEST DATA



## TEST PERFORMANCE

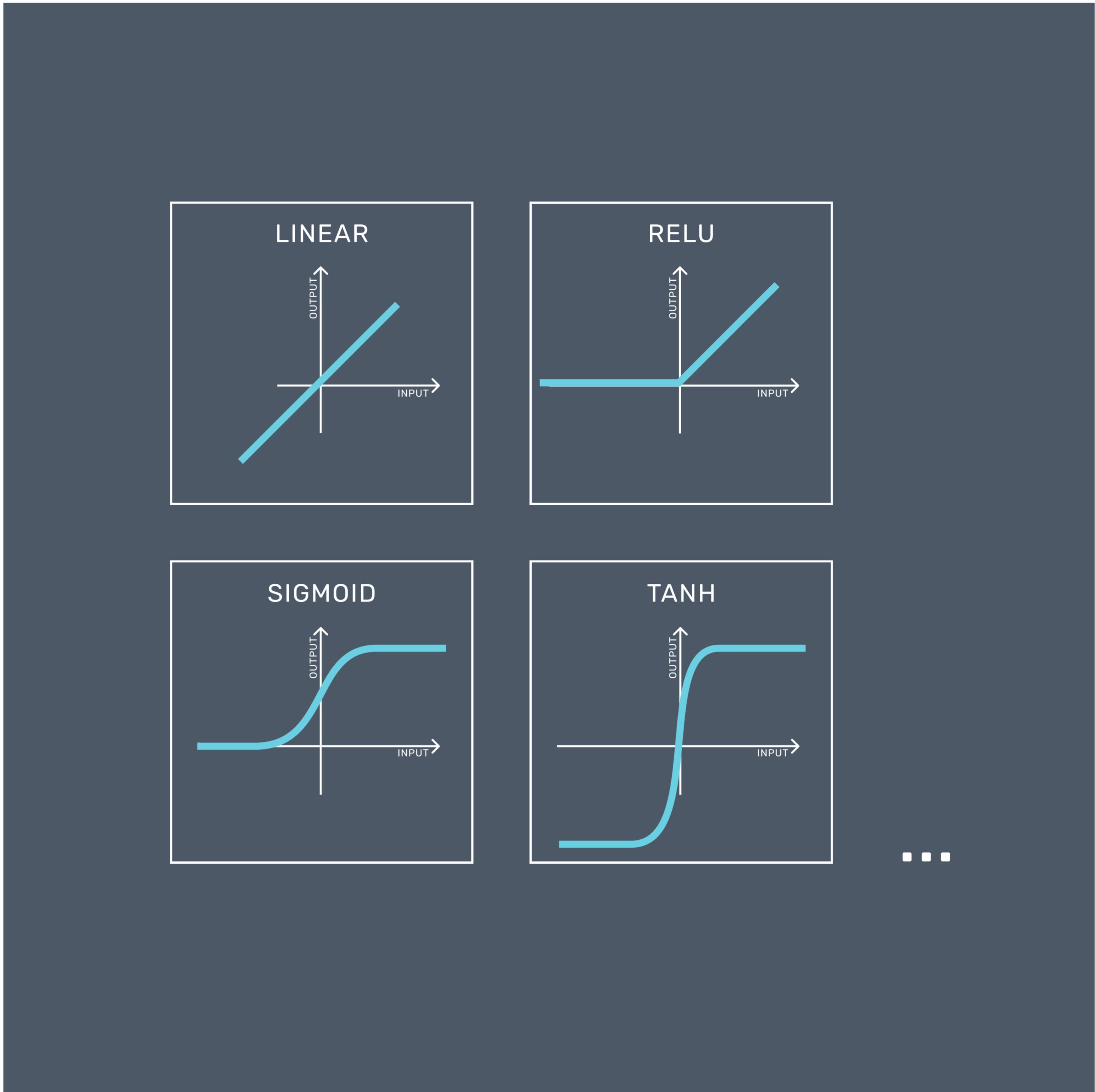
And the MSE of the test predictions is not too bad either. As in Chapter 1, we can bring it closer to the training MSE by having more data points and ensuring that the training and test distributions are similar.



## MORE NEURONS

If we want to, we can further improve the performance by adding more layers and units.

As we add more neurons, the neural network will be able to increase the granularity of its predictions, resulting in a prediction curve that is smoother and more well-defined.

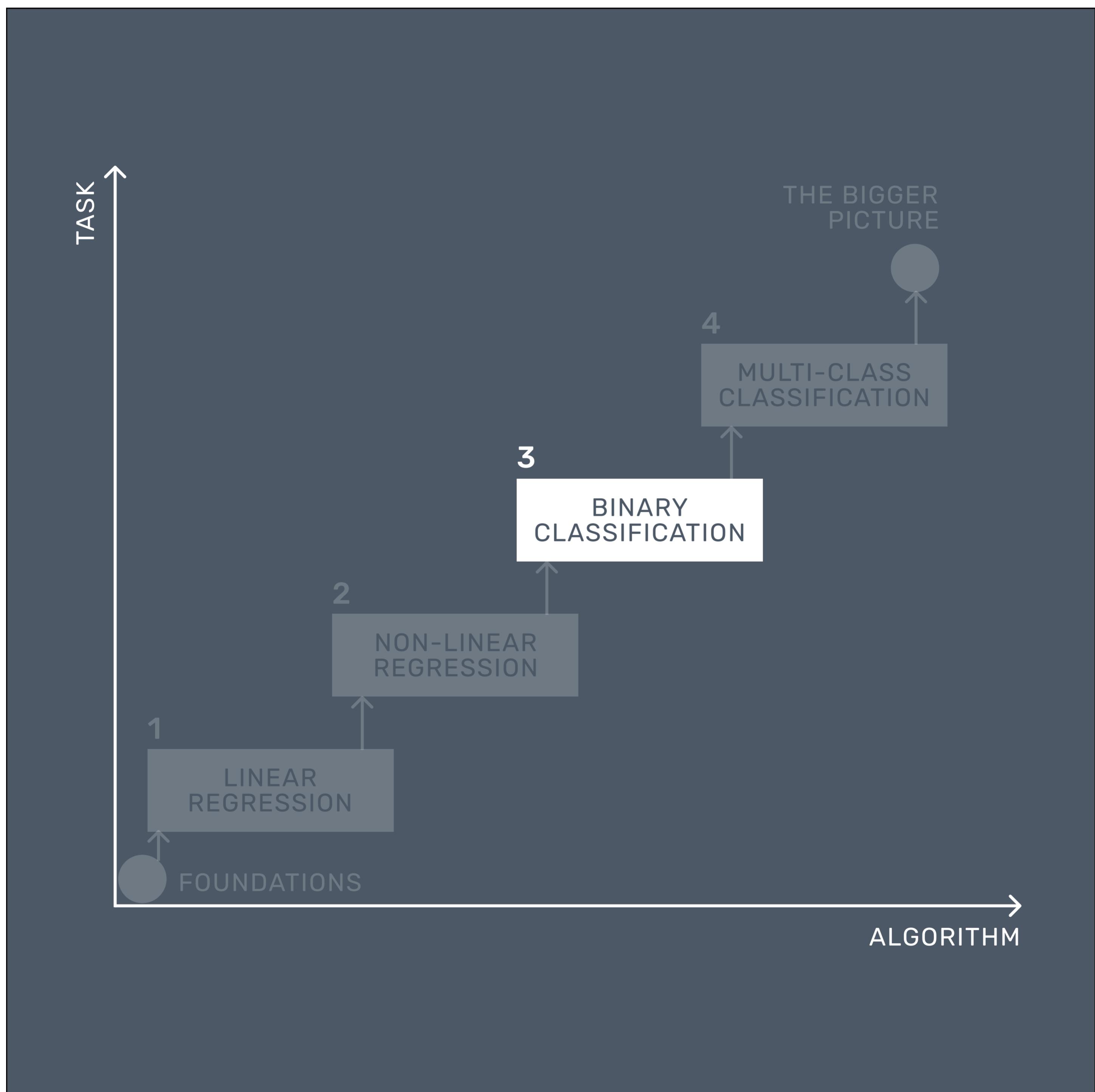


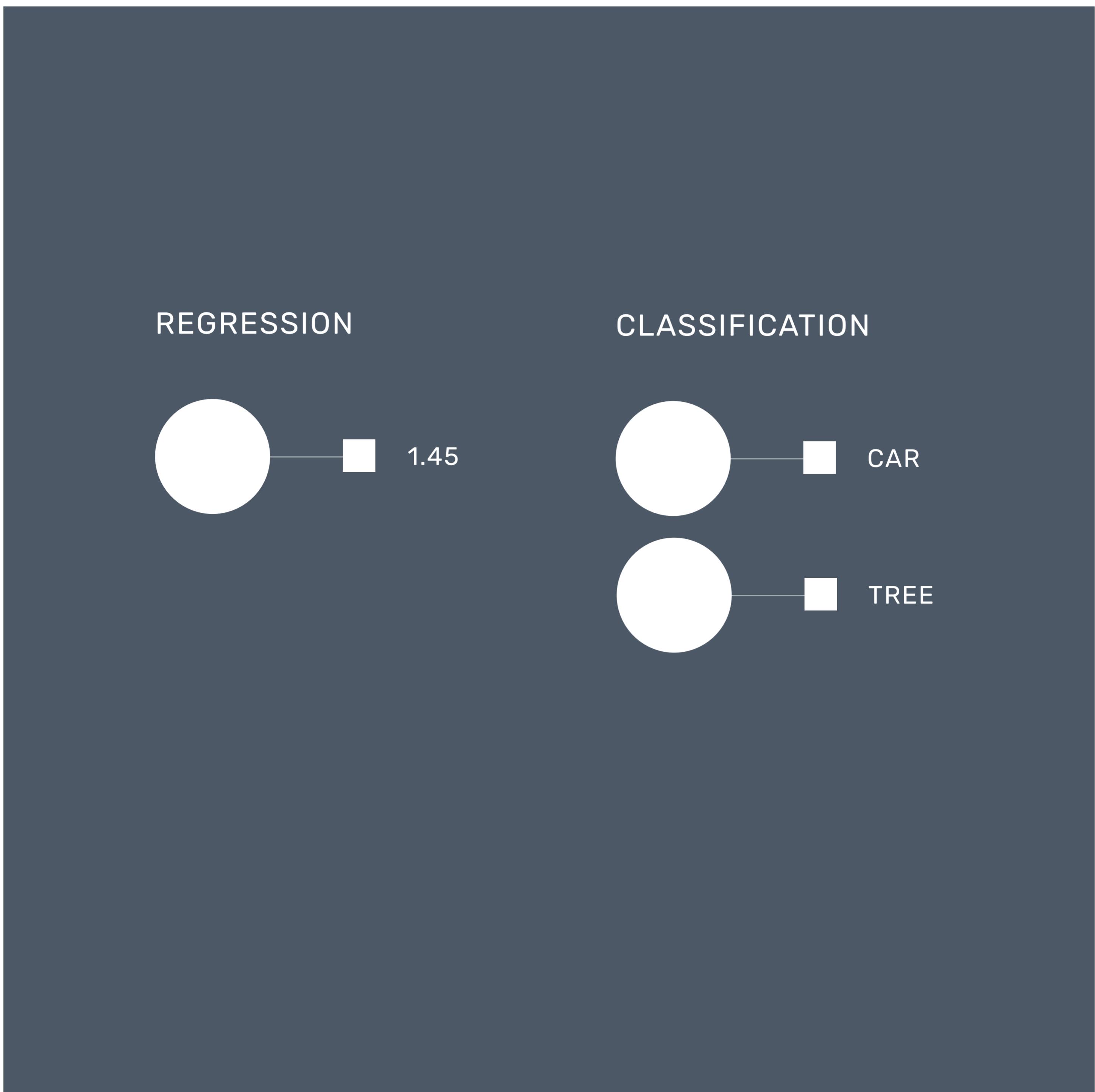
## ACTIVATION FUNCTIONS

There are many other types of activation functions. Some of the more commonly used ones are shown here. Each activation function allows the neural network to exhibit a different kind of quality compared to the others.

We'll look at another type, *sigmoid*, in the next chapter.

# 3 - BINARY CLASSIFICATION



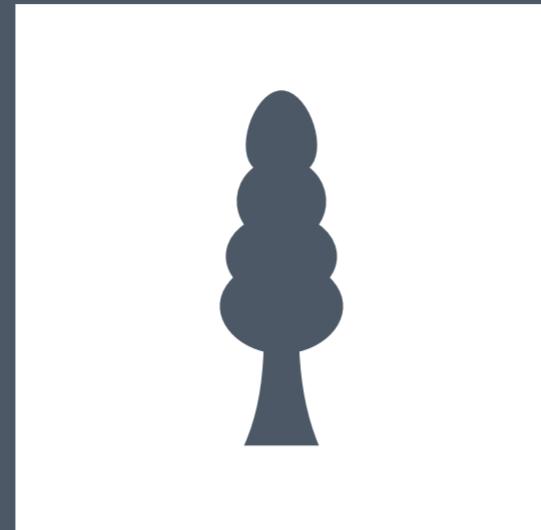


## CLASSIFICATION

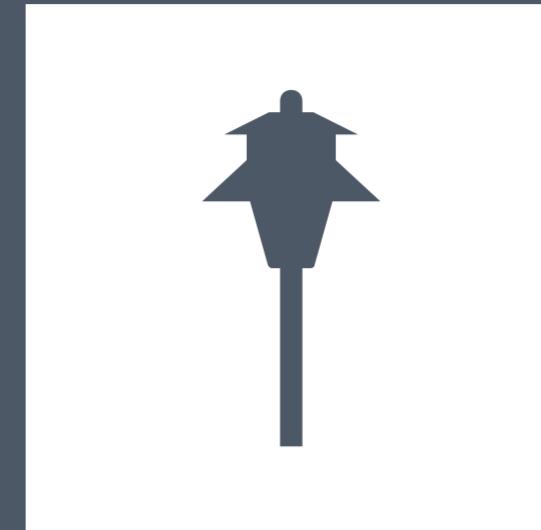
We have seen how the neural network performs regression tasks. In this chapter, we'll see how it performs another type of task—*classification*.

While regression is about predicting continuous values, classification is about predicting groups.

## COMPUTER VISION



TREE



NOT TREE

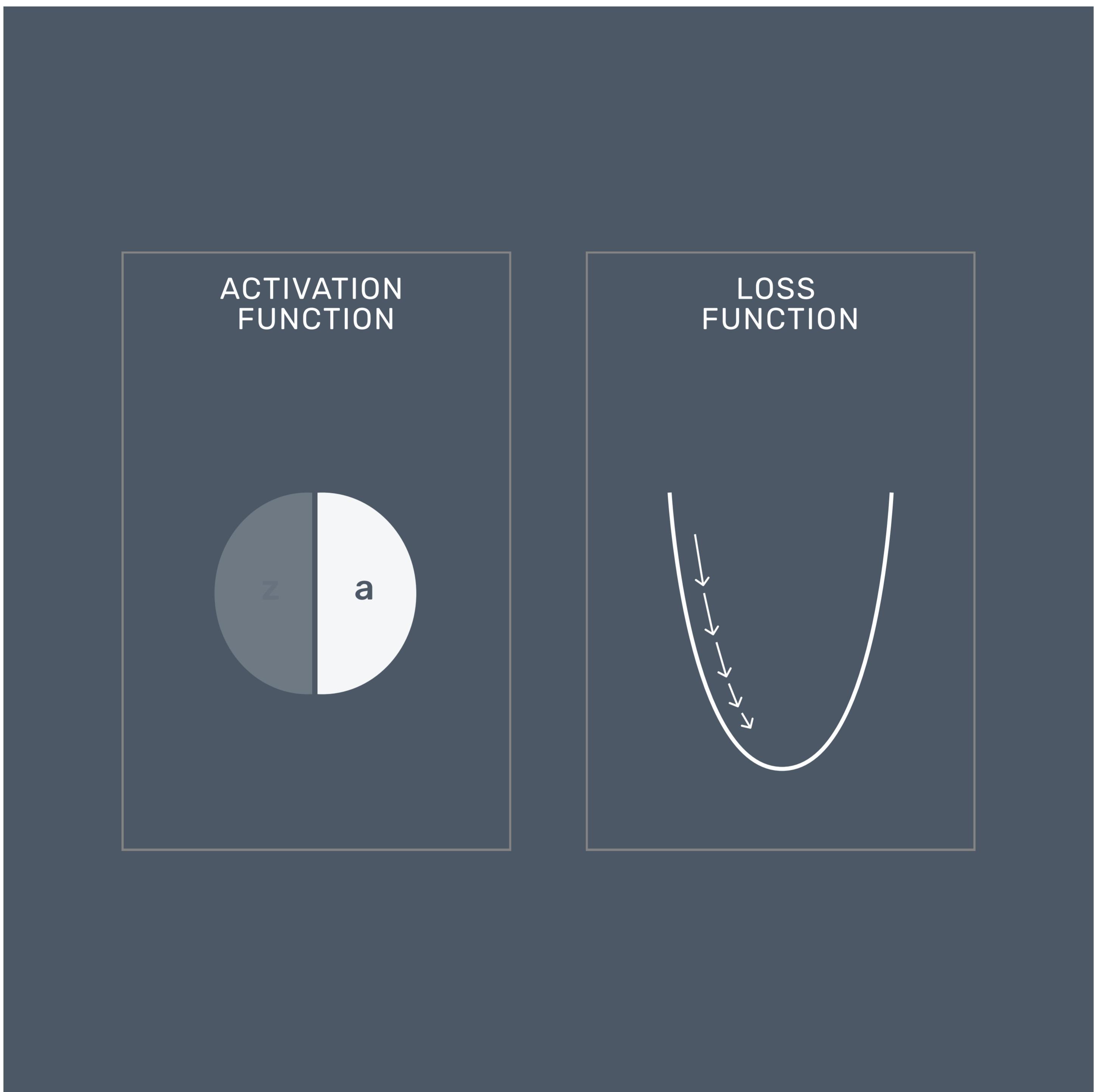
## NATURAL LANGUAGE PROCESSING

SENTENCE	SENTIMENT
IT'S A GREAT DAY	POSITIVE
I DON'T LIKE MONDAYS	NEGATIVE

■ ■ ■

## CLASSIFICATION USE CASES

There is an endless list of deep learning use cases in the real world that involve classification. The classic examples are image classification in computer vision and sentiment analysis in natural language processing (NLP).



## CLASSIFICATION VS. REGRESSION

The good news is, as we move from regression to classification, the basic principles of the neural network remain the same.

In fact, there are only two major differences to note. The first is the type of activation function and the second is the type of loss function. We'll learn more about them in this chapter.



	DIST (MI)	RATING	HOT
TRAINING DATA (24)	0.2	3.5	1
	0.2	4.8	1
	0.5	3.7	1
	0.7	4.3	1
	0.8	2.7	0
	1.5	3.6	1
	1.6	2.6	0
	2.4	4.7	1
	3.5	4.2	0
	3.5	3.5	1
	4.6	2.8	0
	4.6	4.2	0
	4.9	3.8	0
	6.2	3.6	0
	6.5	2.4	0
	8.5	3.1	0
	9.5	2.1	0
	9.7	3.7	0
	11.3	2.9	0
	14.6	3.8	0
	17.5	4.6	1
	18.7	3.8	1
	19.5	4.4	1
	19.8	3.6	1
TEST DATA (8)	0.3	4.6	1
	0.5	4.2	1
	1.1	3.5	1
	1.2	4.7	1
	2.7	2.7	0
	3.8	4.1	0
	7.3	4.6	0
	19.4	4.8	1

## THE DATASET

We'll use the same dataset as in Chapter 2 with one difference—we have replaced the target with a new one.

Let's suppose that we are tracking the hotels with the highest demand, segregating them from those which are not. This is represented by the new target called *hot* which has two possible values - yes and no.

The target is no longer a continuous variable but is a *categorical variable* instead. Categorical variables have a known, fixed number of values, or more precisely, *classes*. Unlike continuous variables, these classes do not imply any order (for example, whether one class is better than the other).

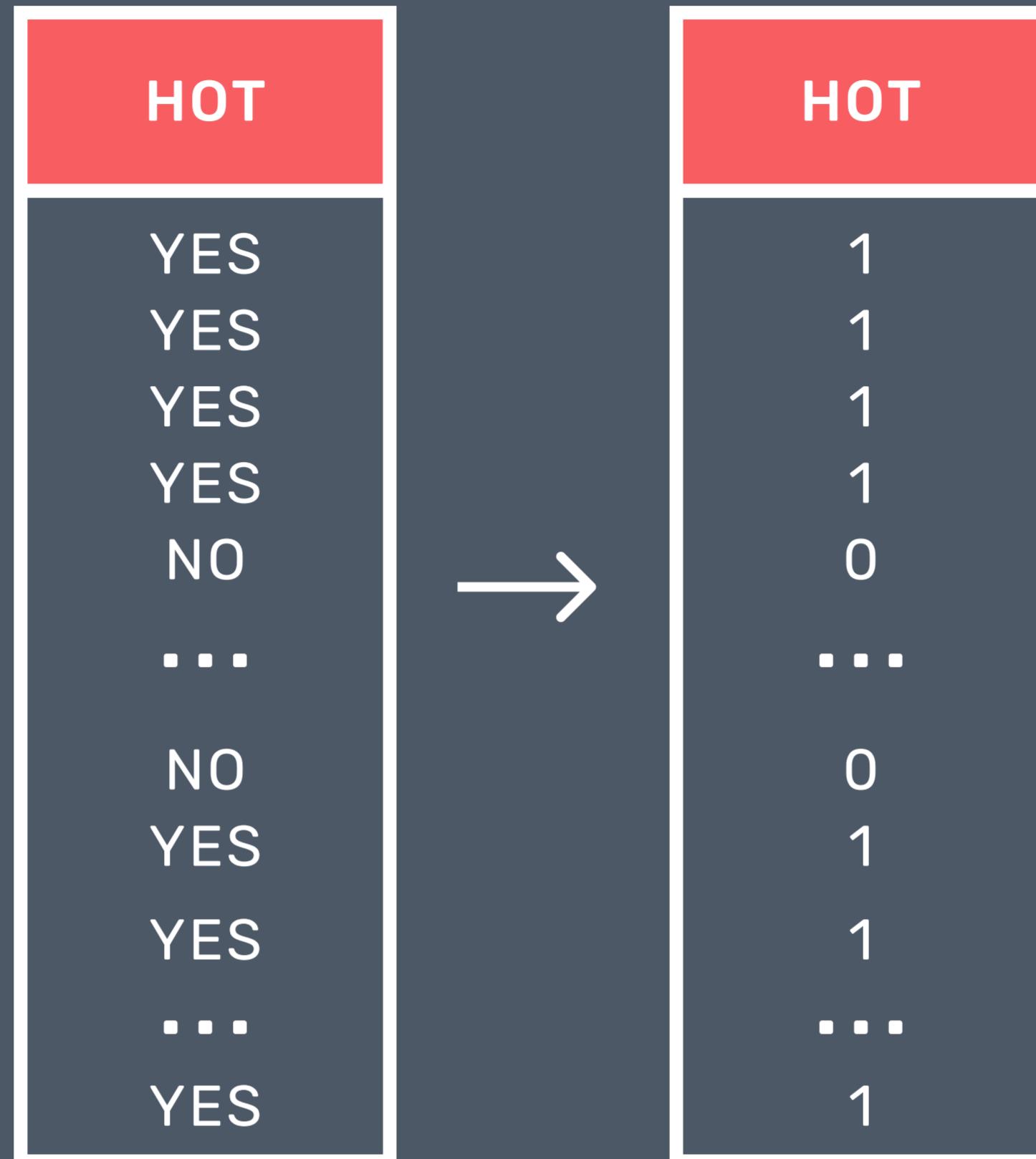


## THE DATASET

Recall that for regression, we wanted to model the line (for a single-feature scenario) or plane (for a two-feature scenario) where the predicted values would fall on.

As for classification, we want to model the classification *boundary* that separates the classes. For classification, it's more common to use the term *label* instead of *target*, so we will use that term from now on.

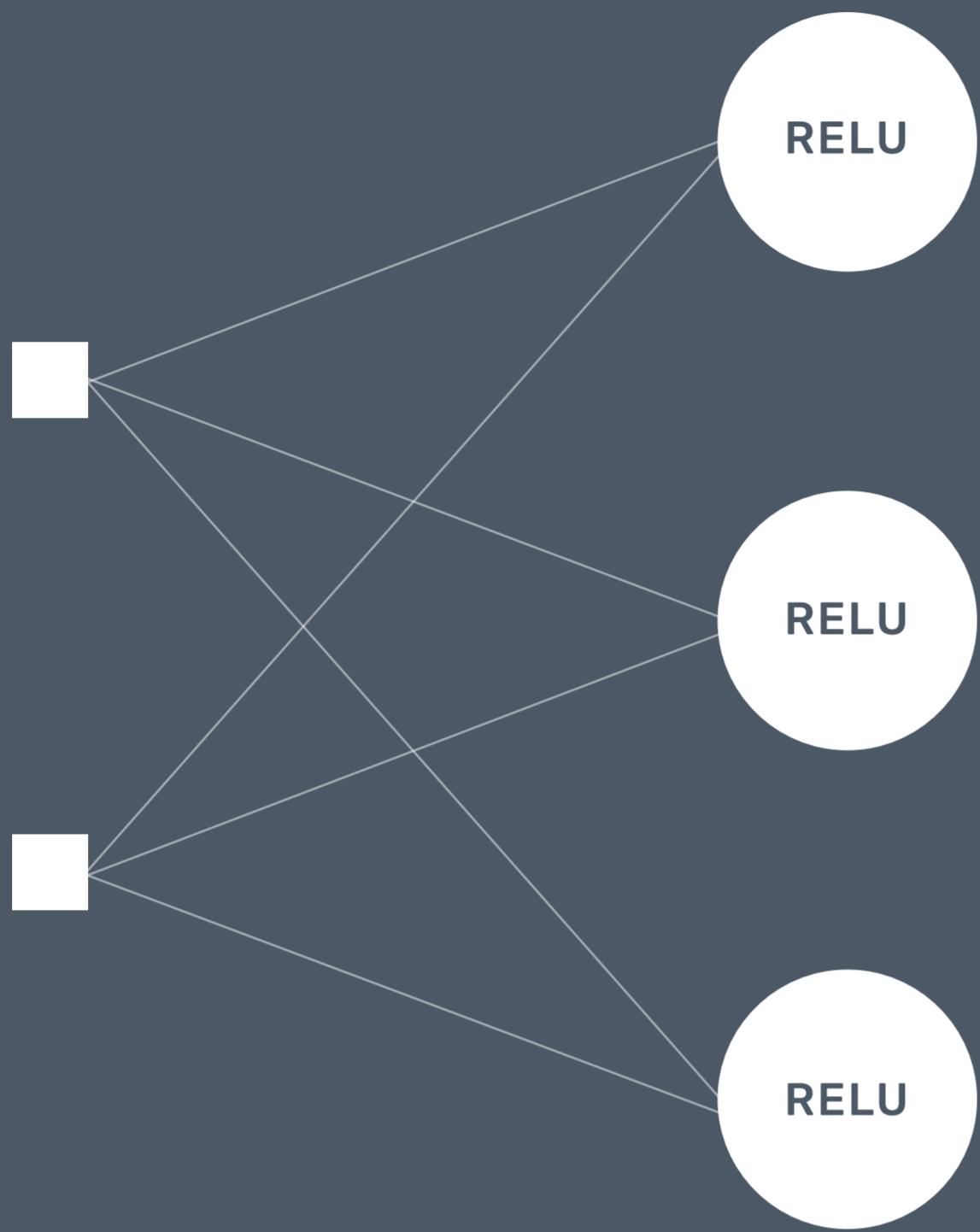
Here is shown the plot of the actual class for each training data point. Also shown is an example of a hand-drawn boundary that separates the two classes. This is what we want our neural network to produce.



## BINARY CLASSIFICATION

Our task is called a *binary classification* task because the outcome will be either one or the other - yes or no.

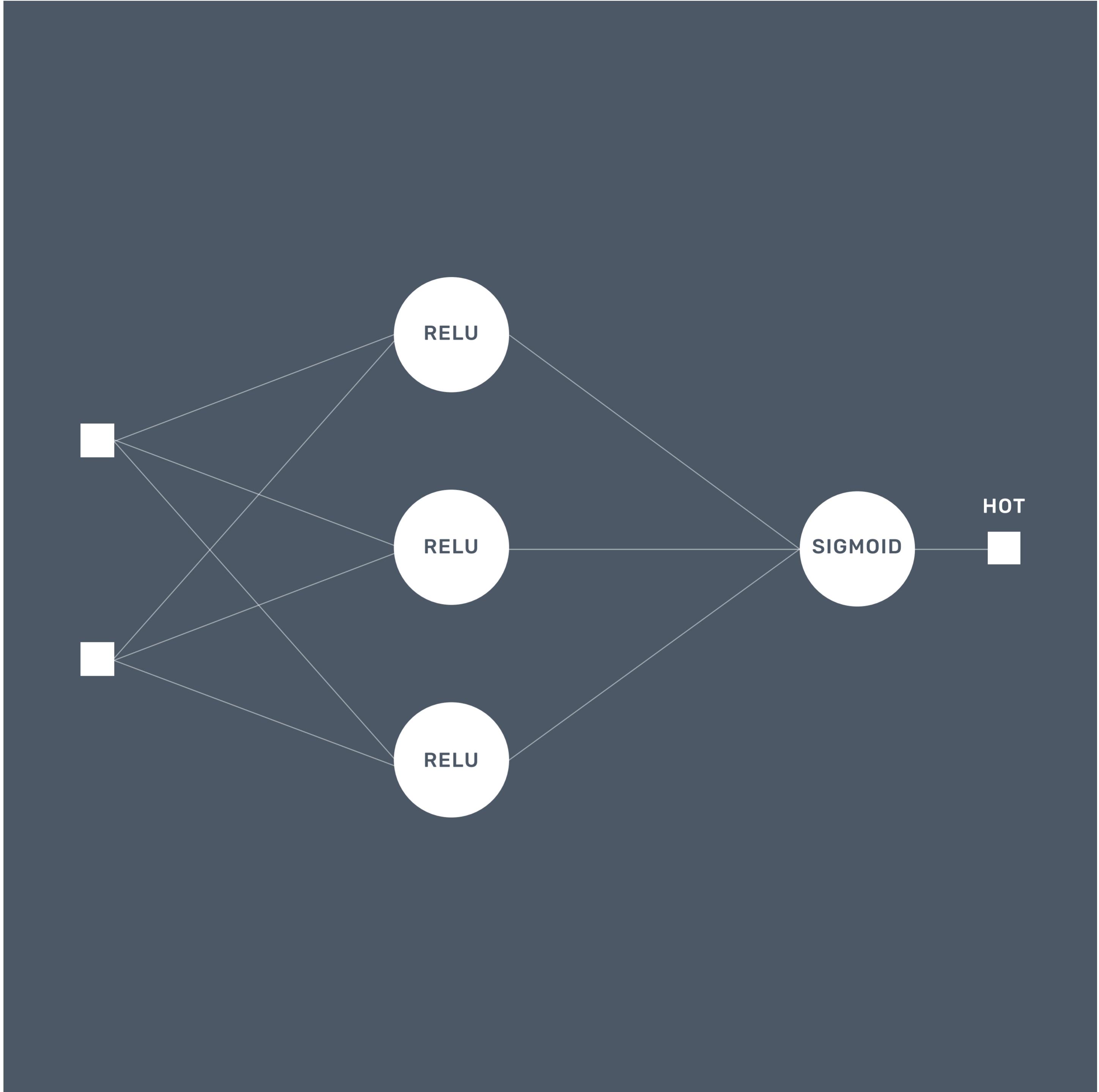
Before we can proceed with training, we need to convert the labels into a numeric format. For this purpose, we'll assign discrete values 1 for yes and 0 for no.



## INPUT & HIDDEN LAYERS

Let's start building the neural network architecture that we need for this task.

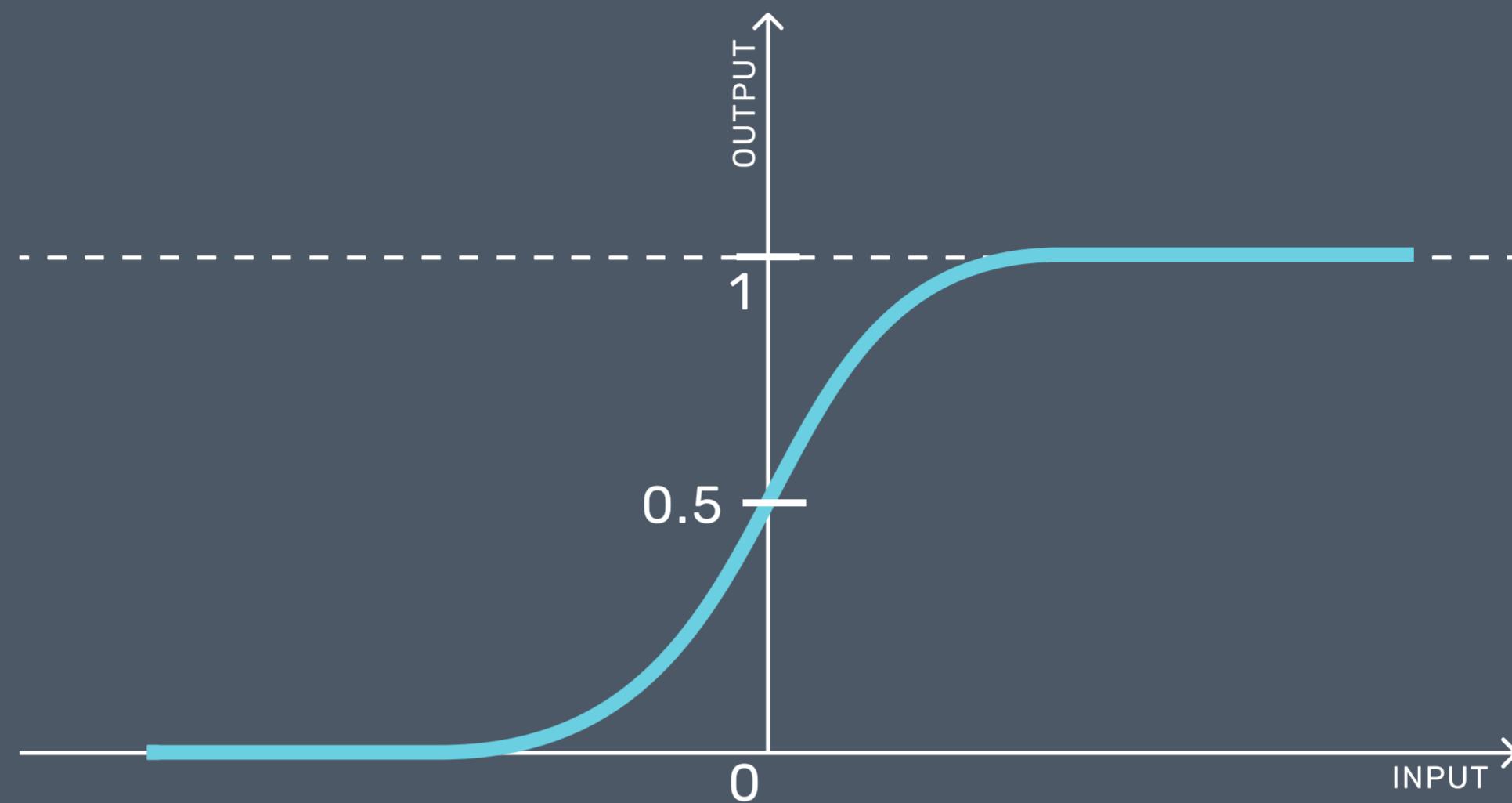
We'll stick with the same number of layers and units as in the previous task. The activation function in the hidden layer also remains unchanged as ReLU.



## OUTPUT LAYER

In fact, the overall architecture remains the same.

The only difference is in the output, where we'll replace the linear activation function with a new one called *sigmoid*. Let's take a look at how it works.

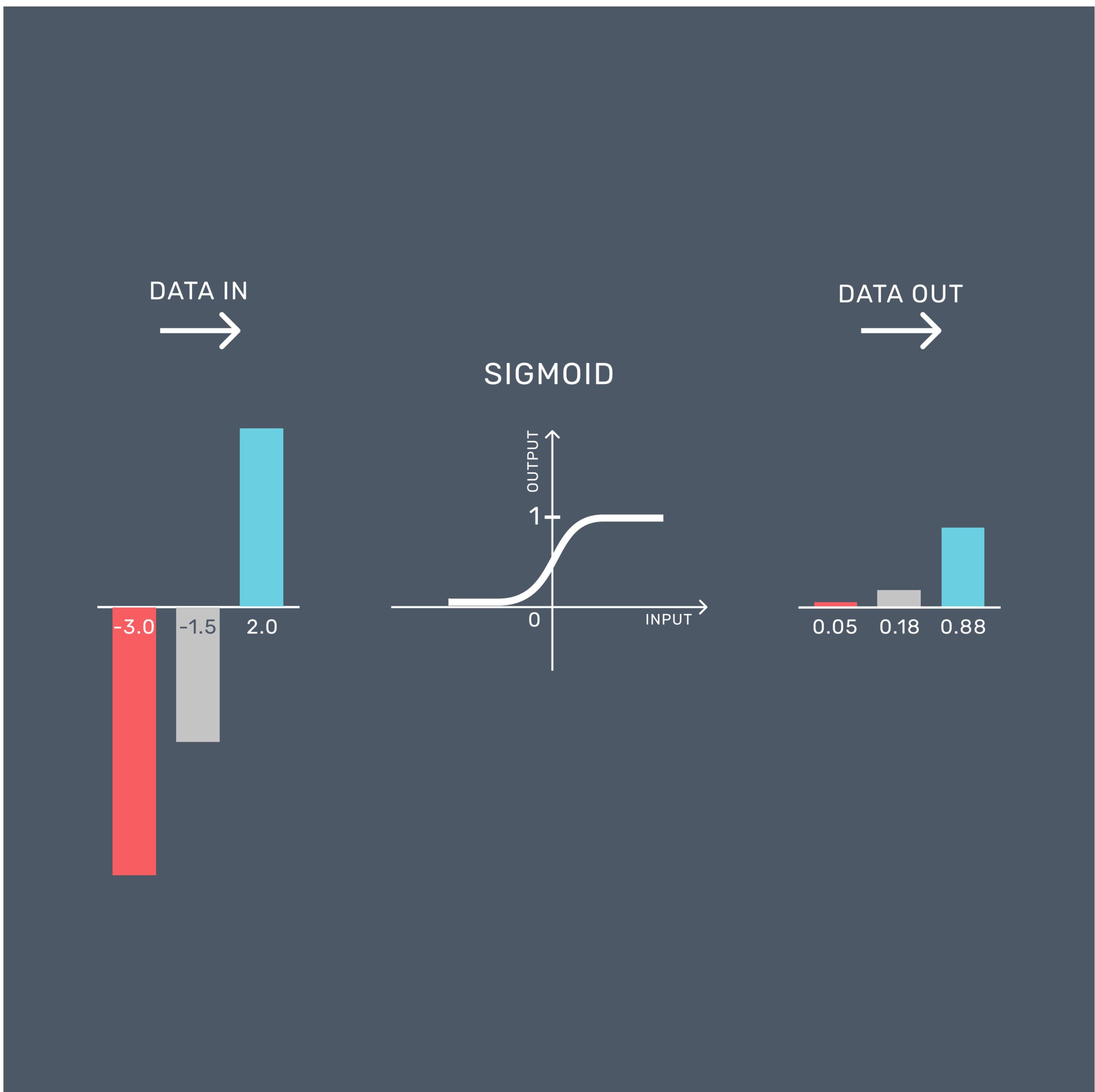


## SIGMOID ACTIVATION FUNCTION

Since the labels are discrete numbers of 0s and 1s, we must set up the neural network so that the predictions return either 0 or 1 and nothing else.

We can't achieve that with the current configuration. This is because the linear activation outputs continuous values instead of discrete.

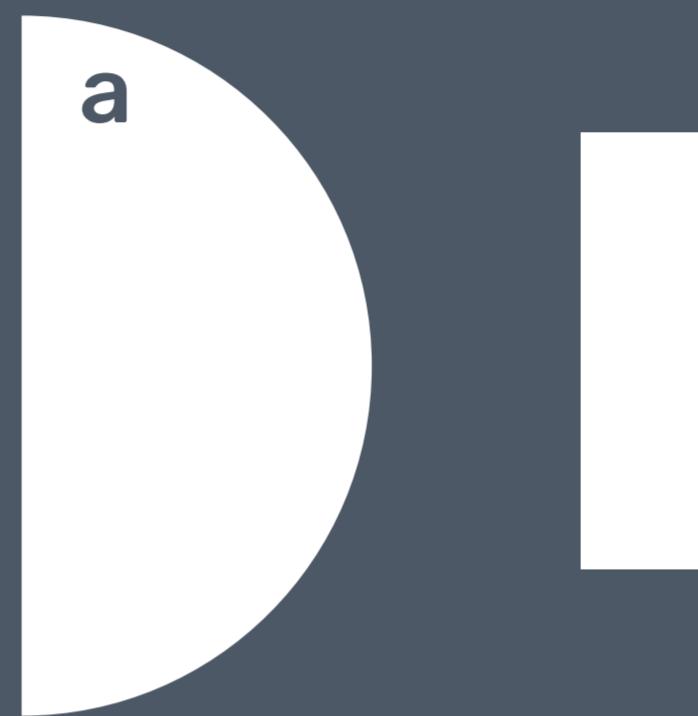
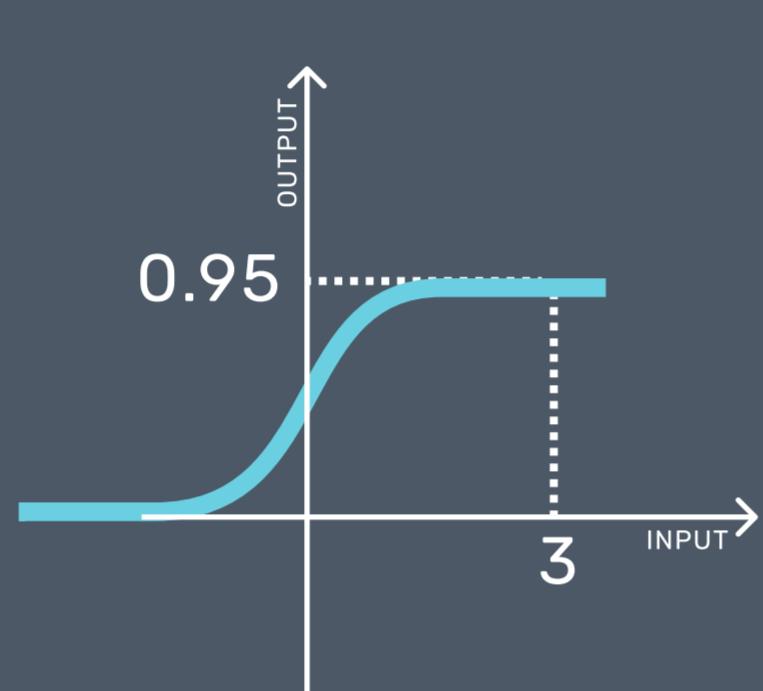
A sigmoid function solves this problem by squeezing its input into a value between 0 and 1.



## SIGMOID ACTIVATION FUNCTION

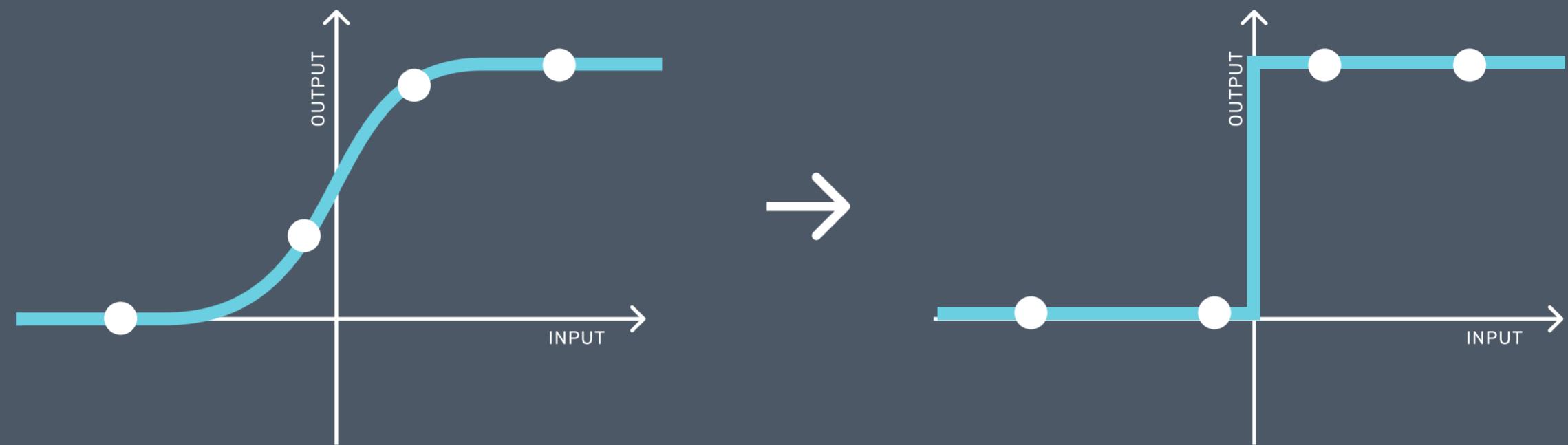
Regardless of how large or small the input is, this function ensures that it is converted into a number between 0 and 1.

## SIGMOID ACTIVATION



## OUTPUT LAYER

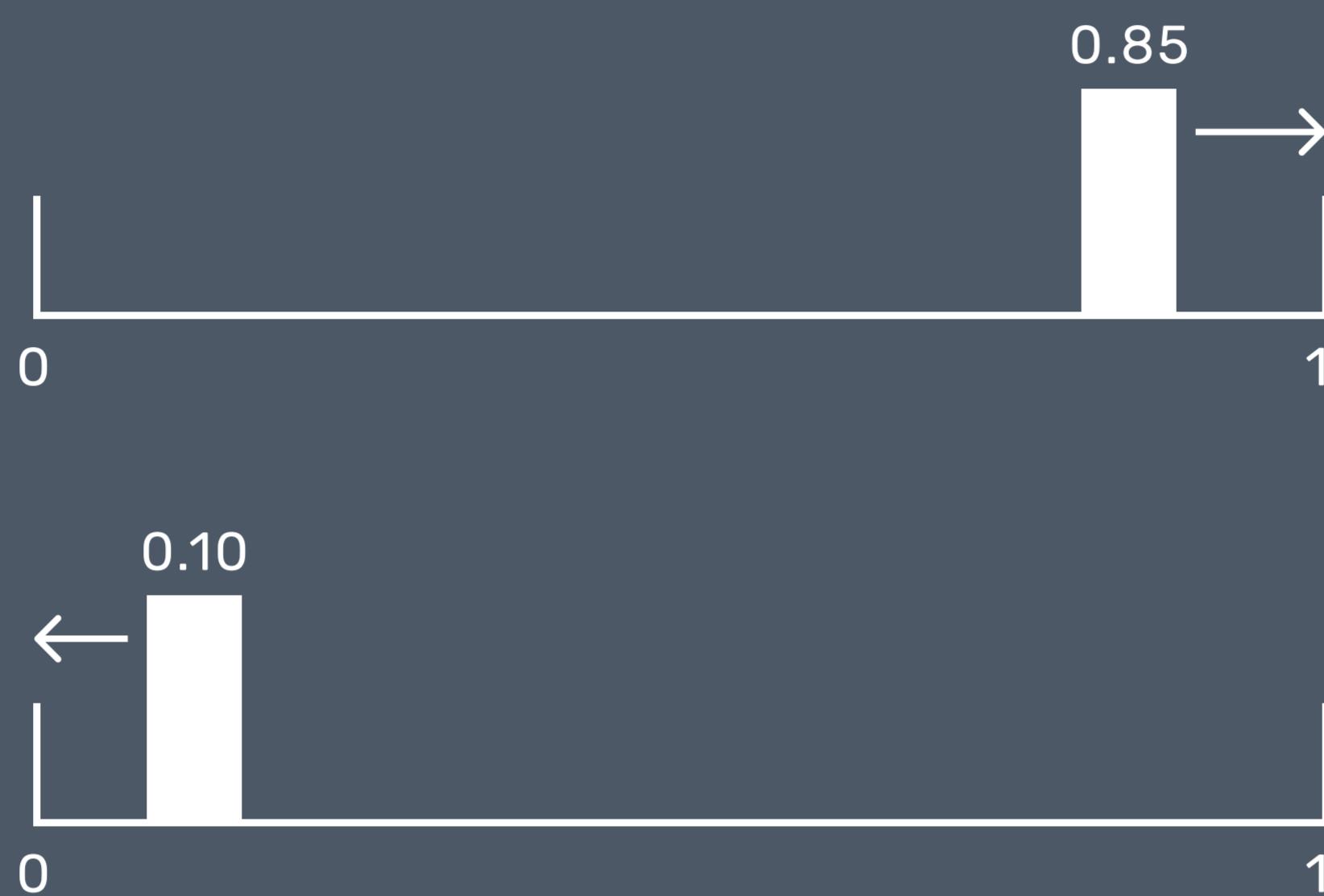
For this reason, we'll use the sigmoid activation function in the output layer.



## DISCRETIZE

This still leaves us with a small problem. We can now convert the output to fall *between* 0 and 1, but what we need is a value of *either* 0 or 1. The output has to be discrete.

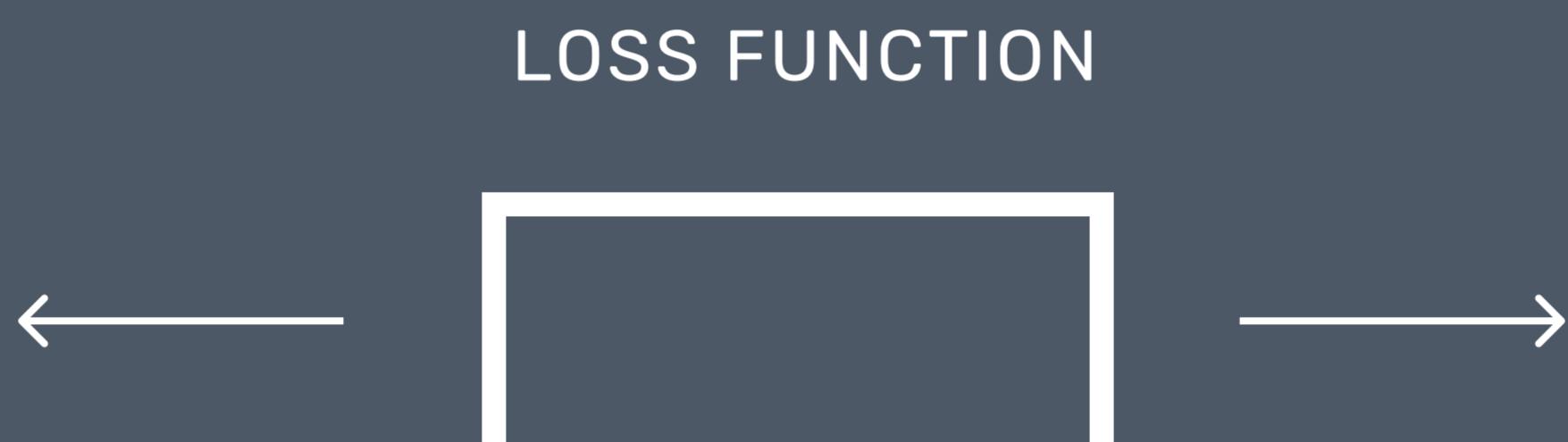
For this, we can add an extra computation step to convert the output to 1 if it's greater than 0.5, and to 0 if it's less than 0.5.



## PROBABILITY

Why does this work? We can think of the output of the sigmoid activation as representing a probability.

For example, if the output is 0.88, the neural network is indicating that the label has a higher probability of being 1. And if the output is 0.12, it has a higher probability of being 0.

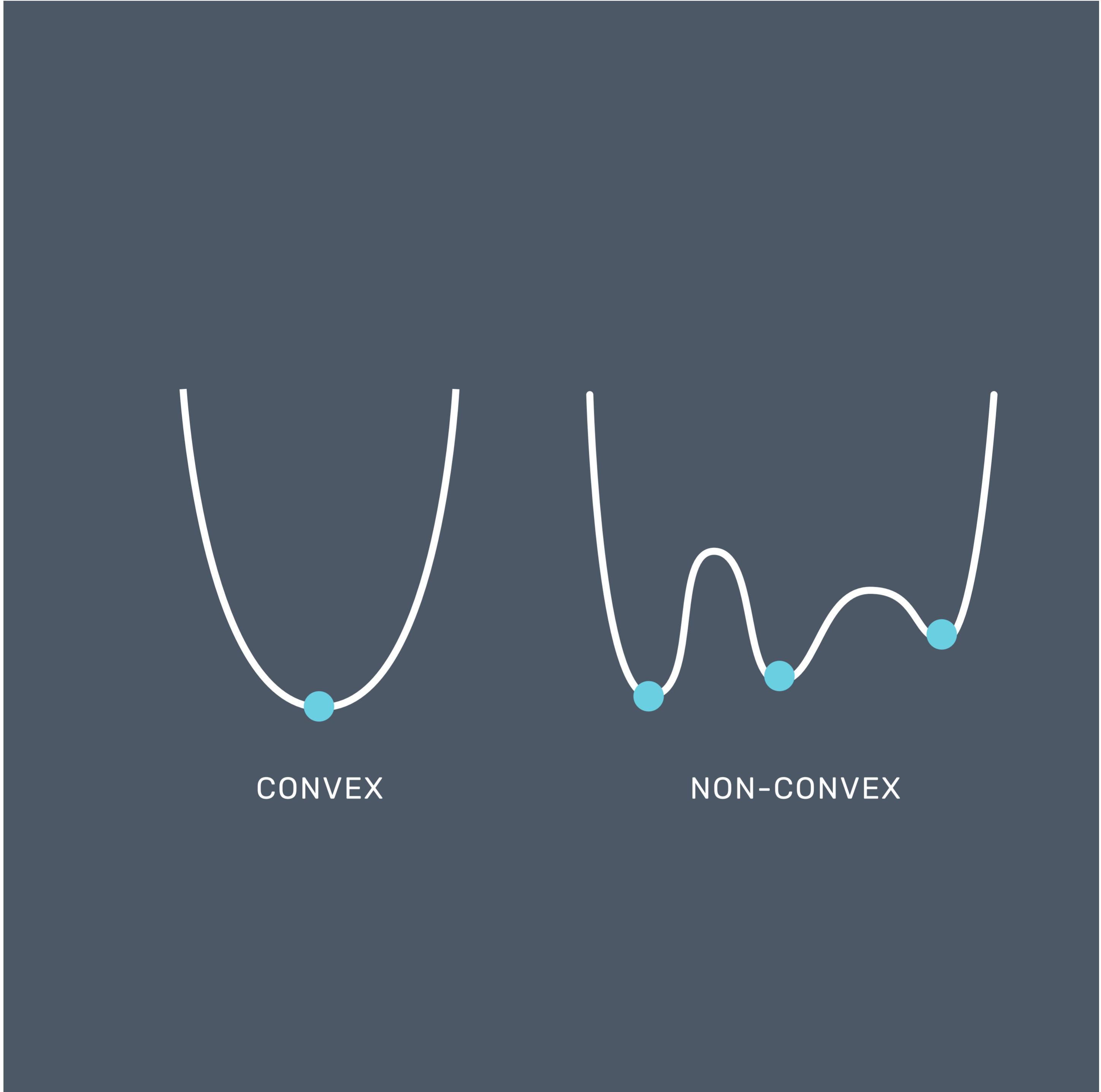


## LOSS FUNCTION

But how can we implement this concept? The answer is the loss function.

Recall that the loss function defines the goal of the neural network, and as a result, dictates how it behaves.

We have covered the first difference between regression and classification—the activation function. Now we'll look at the second one—designing the loss function.

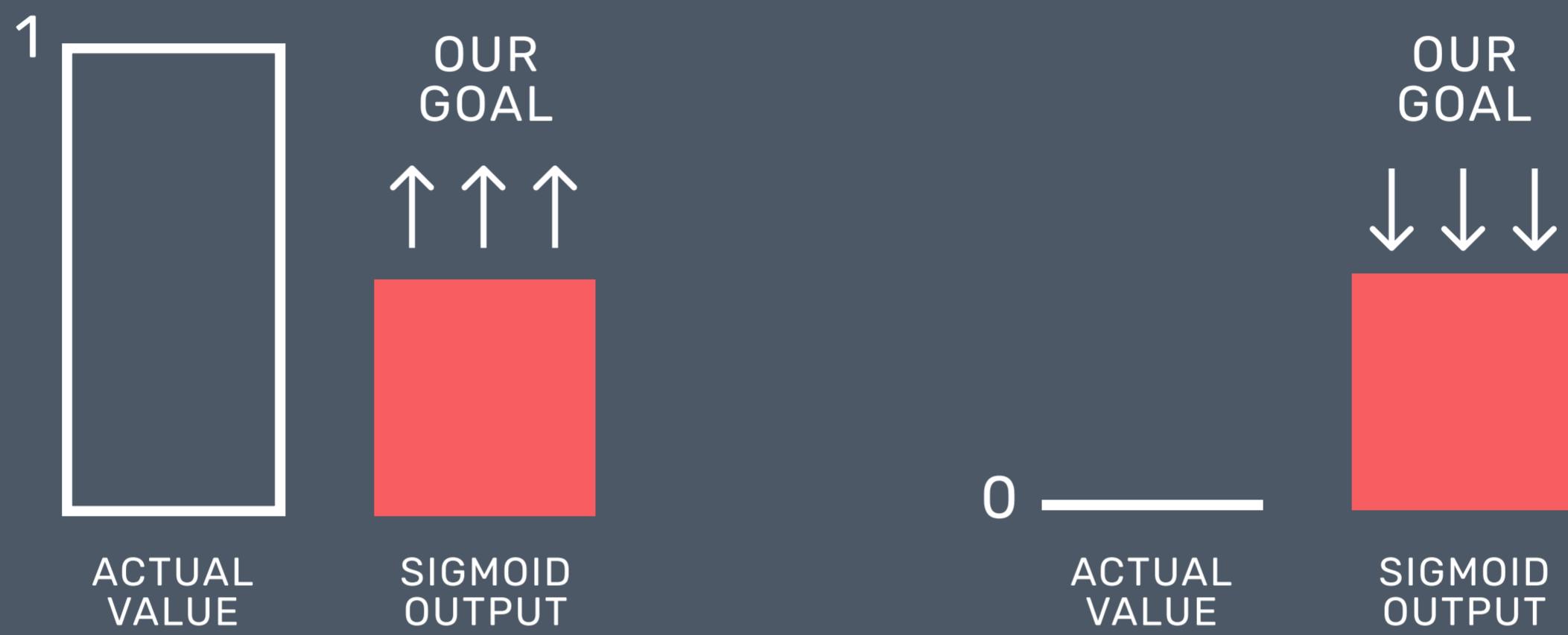


## CONVEX AND NON-CONVEX

For regression, we chose MSE as the loss function because it gives us a desirable property of a single minimum point on the loss curve. We call this a *convex* loss function.

It turns out however (we won't cover the math here) that using an MSE in a binary classification task will result in a *non-convex* loss function. It means that there will be more than one location along the loss curve where a *local* minimum exists, making it difficult for the neural network to find its true, or *global* minimum.

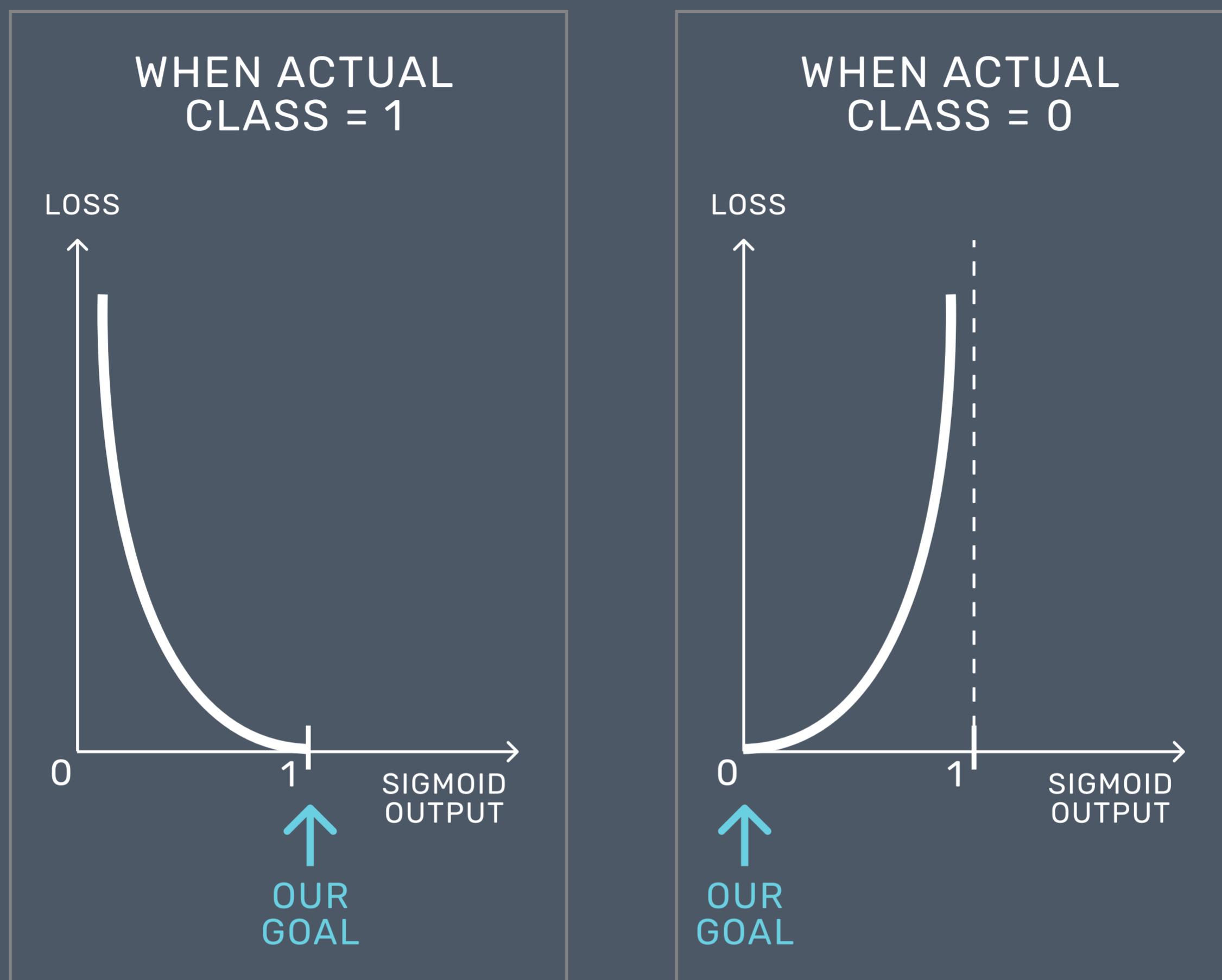
For this reason, we'll need to use a different type of loss function.



## THE GOAL

Our goal is to output a prediction of 1 when the actual value is 1 and a prediction of 0 when the actual value is 0.

That is, to get a prediction of 1, we want the sigmoid's output to be as close to 1 as possible, and vice versa for 0.

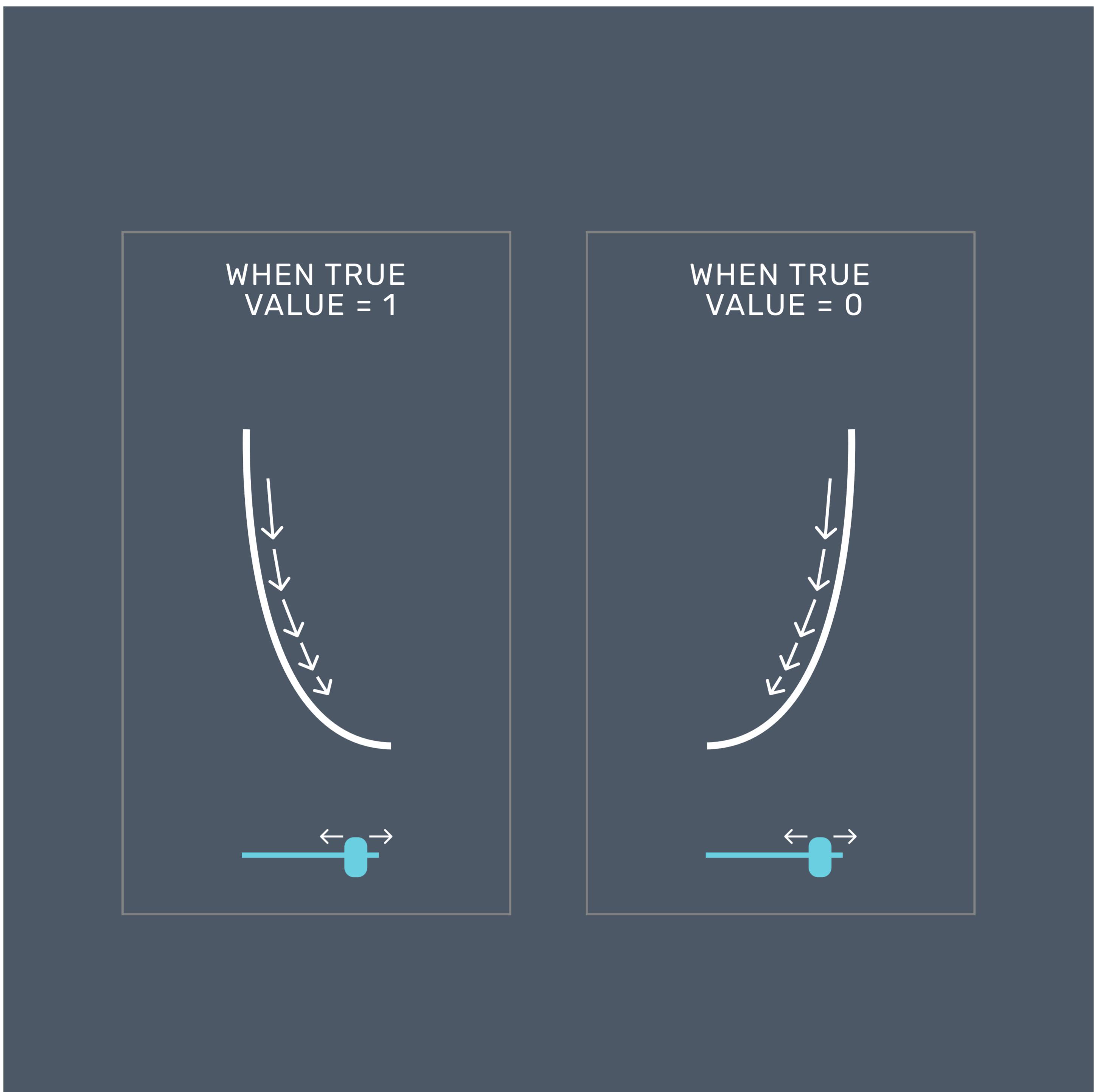


## LOSS FUNCTION

The loss function we'll be using is called the *binary cross entropy*. This function fulfills our need to have a convex loss curve.

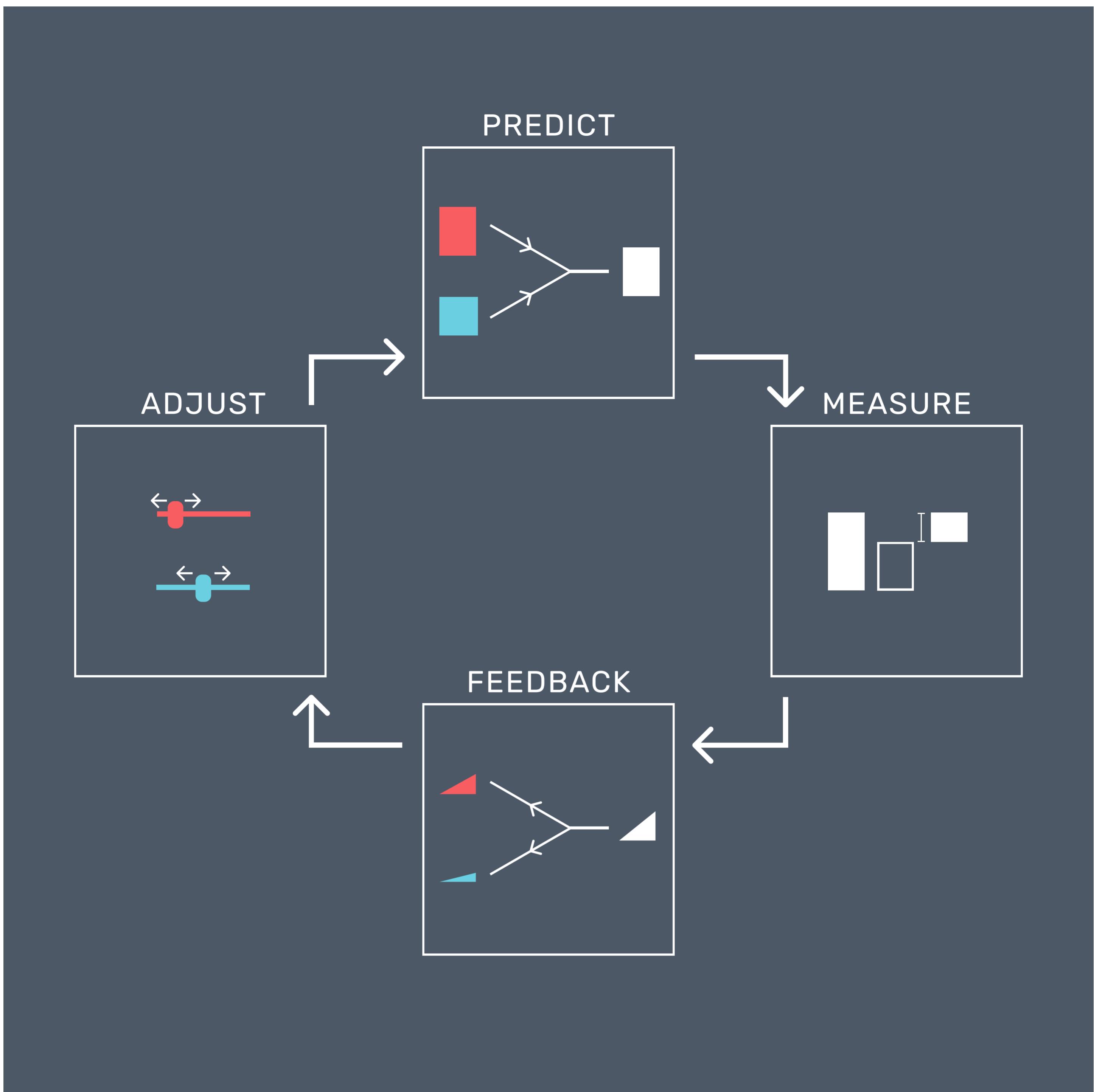
The plots here depicts the shape of this loss function, with each class having its own loss curve.

When the actual class is 1, we want the sigmoid output to be as close to 1 as possible, correspondingly pushing the loss toward its minimum. And when the actual class is 0, we want the sigmoid output to be as close to 0 as possible, pushing the loss toward its minimum.



## GRADIENT DESCENT

From here, we can perform the weight updates using the same gradient descent approach as in Chapters 1 and 2.



## TRAINING

We are now in a position to train the neural network. The training cycle follows the same four steps as in the previous chapters.

## METRIC

ACTUAL VALUE	PREDICTED VALUE	CORRECT?
1	1	Y
0	0	Y
1	0	N
0	1	N
1	0	N
...	...	...
0	0	Y

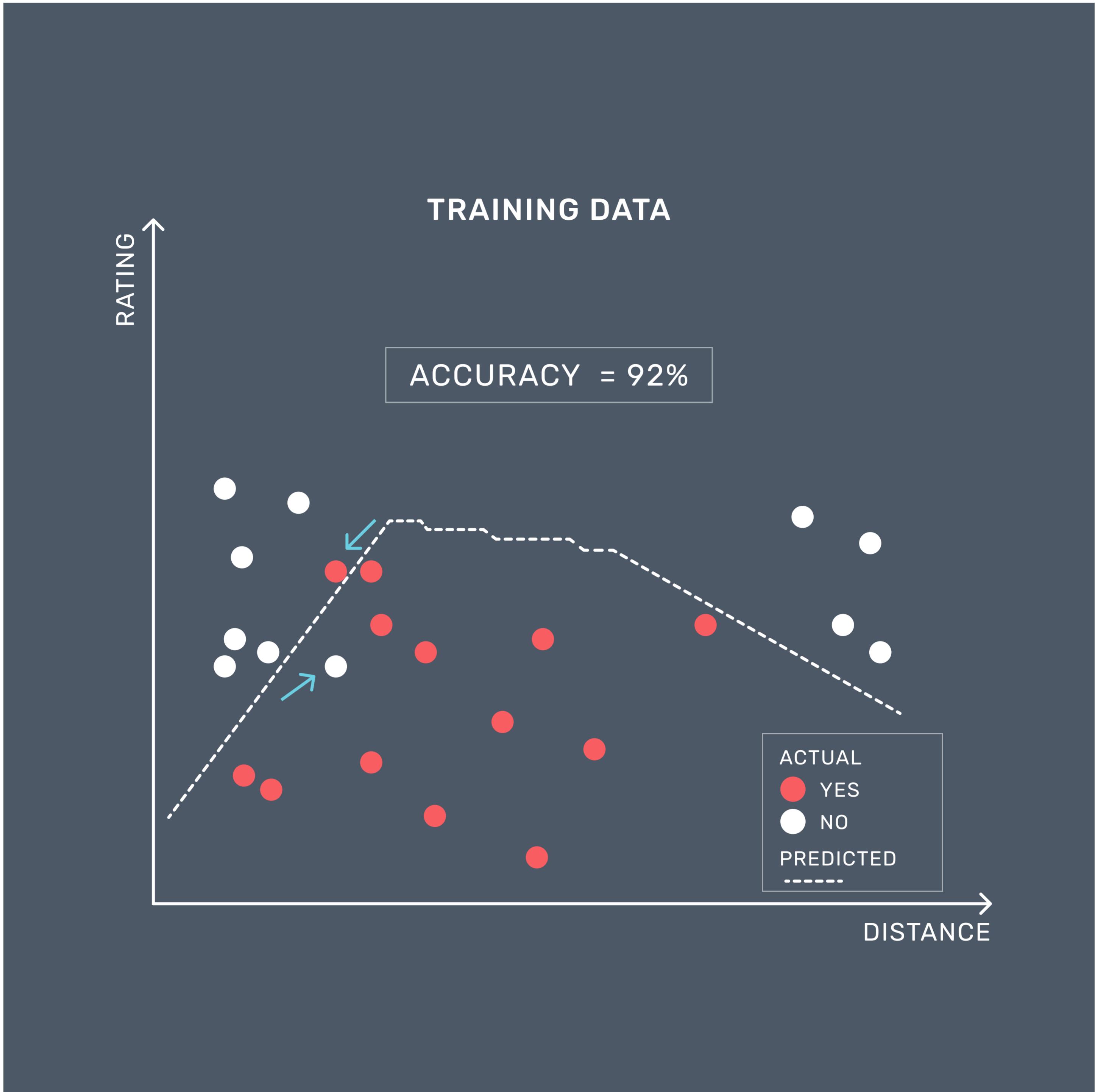
$$\text{ACCURACY} = \frac{\text{TOTAL CORRECT}}{\text{TOTAL PREDICTIONS}}$$

## ACCURACY

There is one last change, which is the metric to measure performance.

In the regression task, we used the MSE both as the cost and the metric.

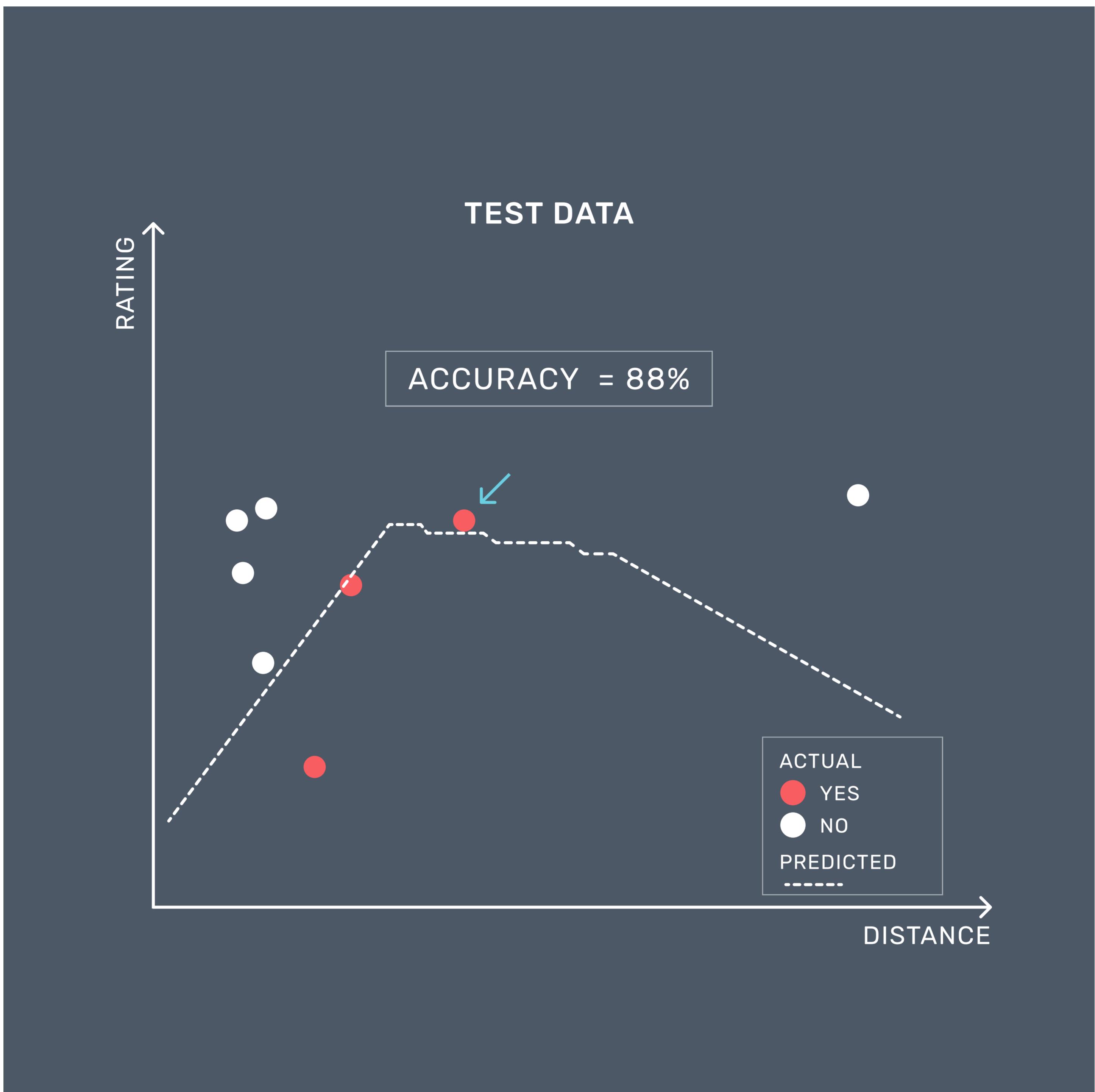
In a classification task, we need to use a different metric to better reflect the performance of the model. We'll use *accuracy*, which gives us the percentage of correct predictions over all predictions.



## TRAINING PERFORMANCE

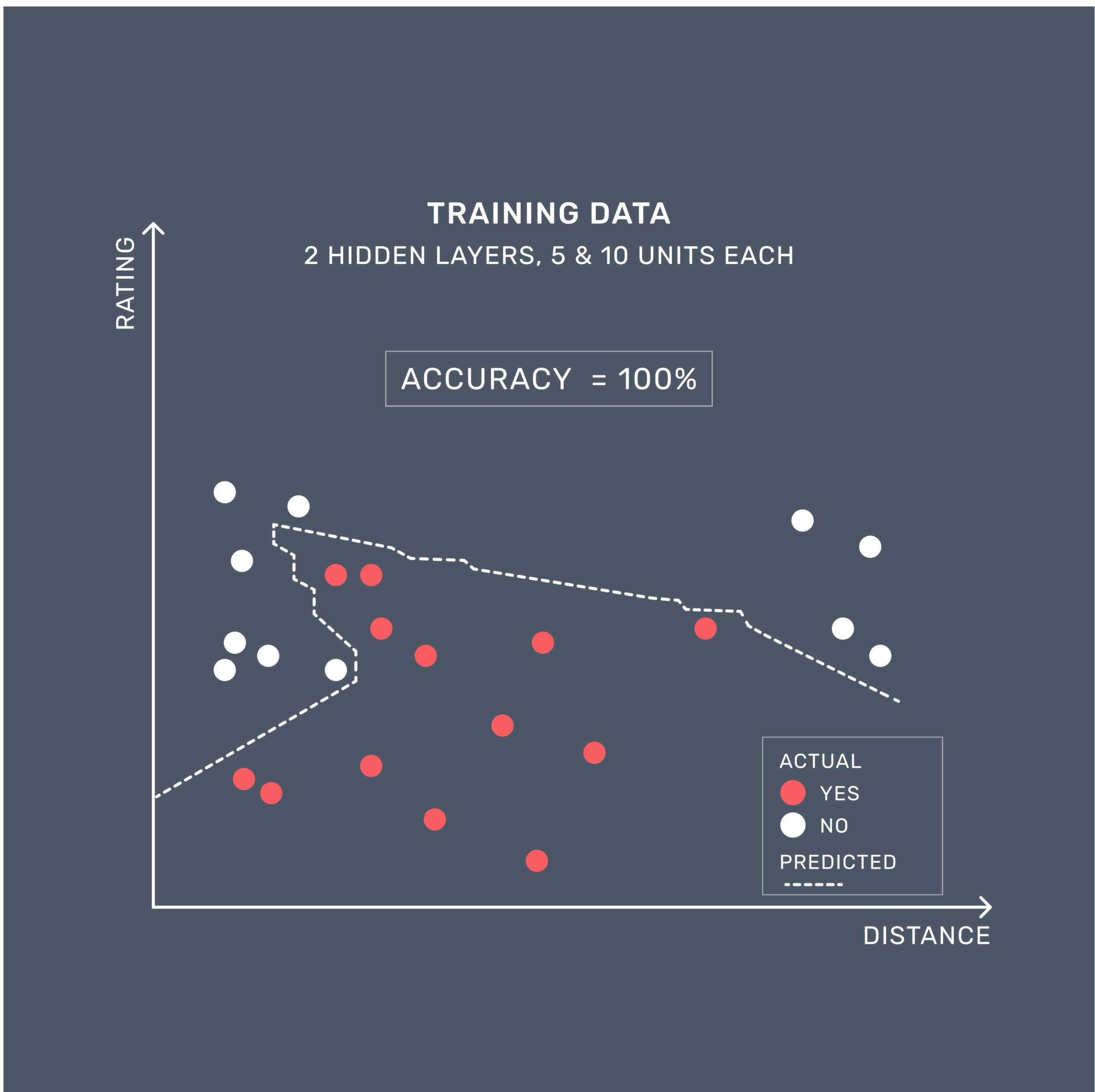
Using the accuracy metric, we can see that the model does pretty well on the training dataset.

The plot also shows the decision boundary of our trained neural network, representing the predictions.



## TEST PERFORMANCE

The prediction on the test data also shows a respectable performance, given the limited number of data points.



## A BIGGER NETWORK

As in the regression task, we can further improve the performance by adding more layers and units.

The plot here shows the decision boundary after modifying the neural network to contain two hidden layers with five and ten units of neurons each. The granularity of its predictions increased, resulting in a more well-defined and accurate prediction curve.

	ACTUAL VALUE	PREDICTED VALUE	CORRECT?
1	0	0	Y
2	0	0	Y
3	0	0	Y
4	0	0	Y
5	0	0	Y
6	0	0	Y
...	...	...	...
89	0	0	Y
90	0	0	Y
91	0	1	N
92	0	1	N
93	0	1	N
94	1	0	N
95	1	0	N
96	1	0	N
97	1	0	N
98	1	0	N
99	1	1	Y
100	1	1	Y

$$\text{ACCURACY} = \frac{\text{TOTAL CORRECT}}{\text{TOTAL PREDICTIONS}} = 93\%$$

## WHEN ACCURACY BECOMES INACCURATE

Accuracy is often used as the measure of classification performance because it is simple to compute and is easy to interpret.

However, it can become misleading in some cases. This is especially true when dealing with imbalanced data, which is when certain classes contain way more data points than the rest.

Let's take the example of predicting fraud credit card transactions. Suppose we have a dataset of 100 data points, of which only 7 are fraud cases. If we simply predicted all the outputs to be 0, we would still manage to get an accuracy value of 93%! Clearly something isn't right.

		Predicted Value	
		0	1
Actual Value	0	True Negative	False Positive
	1	False Negative	True Positive

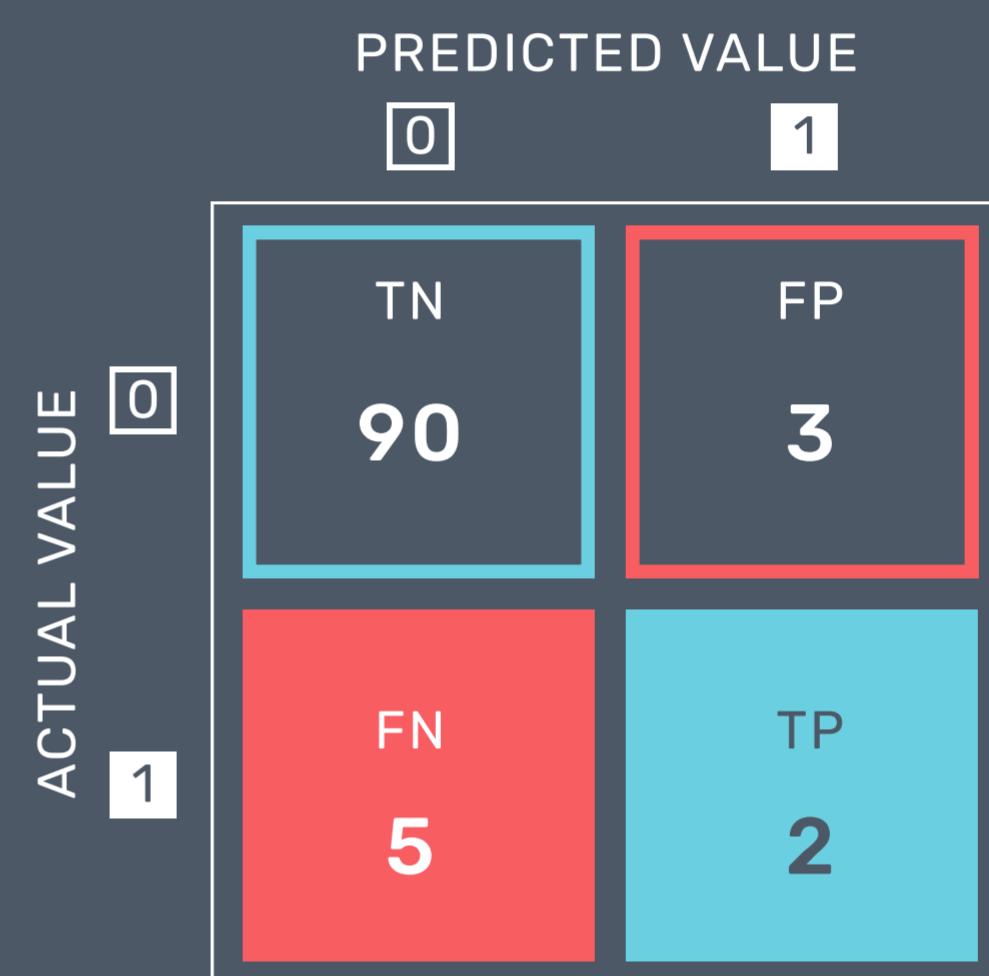
## CONFUSION MATRIX

The *confusion matrix* provides a way to measure performance in a balanced way. It shows the count of predictions falling into one of the following:

- True Negative (TN): when both the actual and predicted values are 0.
- False Positive (FP): when the actual value is 0 but the predicted value is 1.
- False Negative (FN): when the actual value is 1 but the predicted value is 0.
- True Positive (TP): when both the actual and predicted values are 1.

For our example, 0 represents the *not fraud* class while 1 represents the *fraud* class.

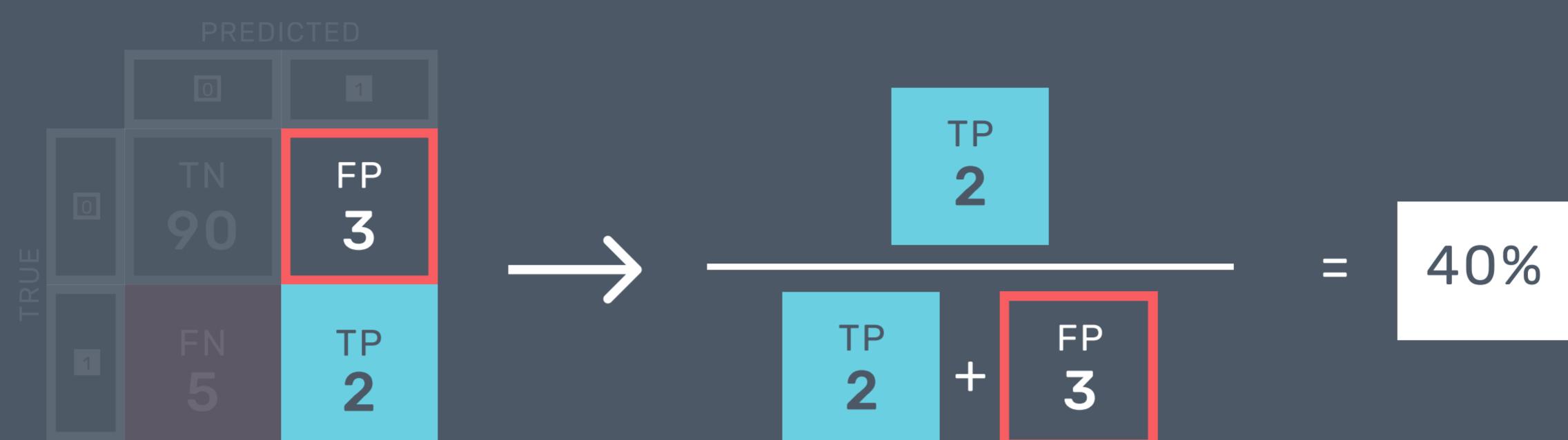
	TRUE VALUE	PREDICTED VALUE	CORRECT?
1	0	0	Y
2	0	0	Y
3	0	0	Y
4	0	0	Y
5	0	0	Y
6	0	0	Y
...	...	...	...
89	0	0	Y
90	0	0	Y
91	0	1	N
92	0	1	N
93	0	1	N
94	1	0	N
95	1	0	N
96	1	0	N
97	1	0	N
98	1	0	N
99	1	1	Y
100	1	1	Y



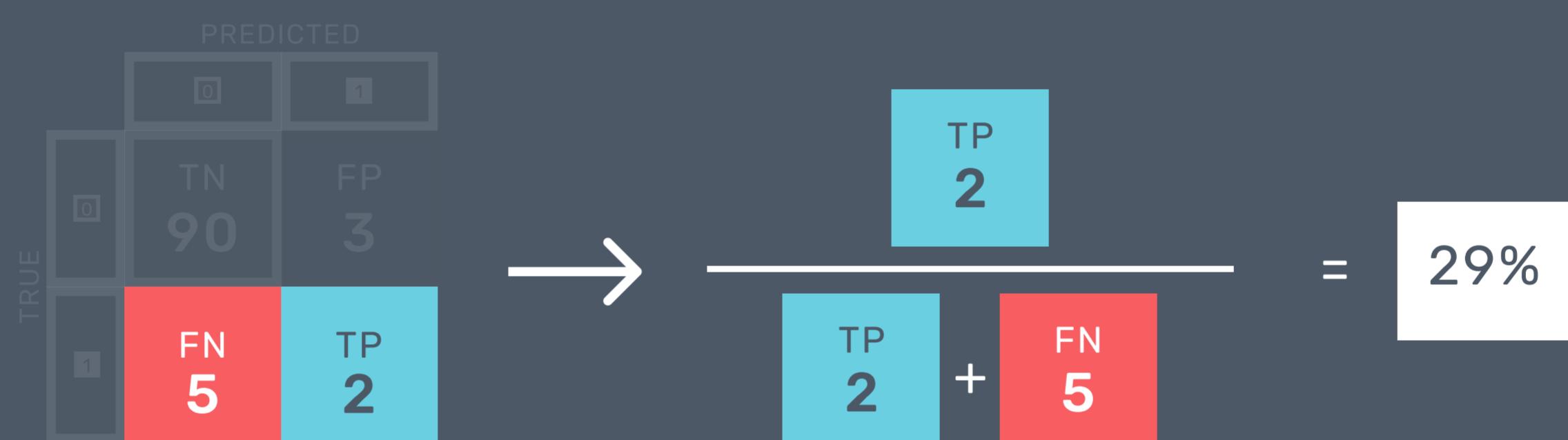
## APPLIED TO DATA

Applied to the credit card fraud dataset, we get 90, 3, 5, and 2 respectively for TN, FP, FN, and TP.

## PRECISION



## RECALL

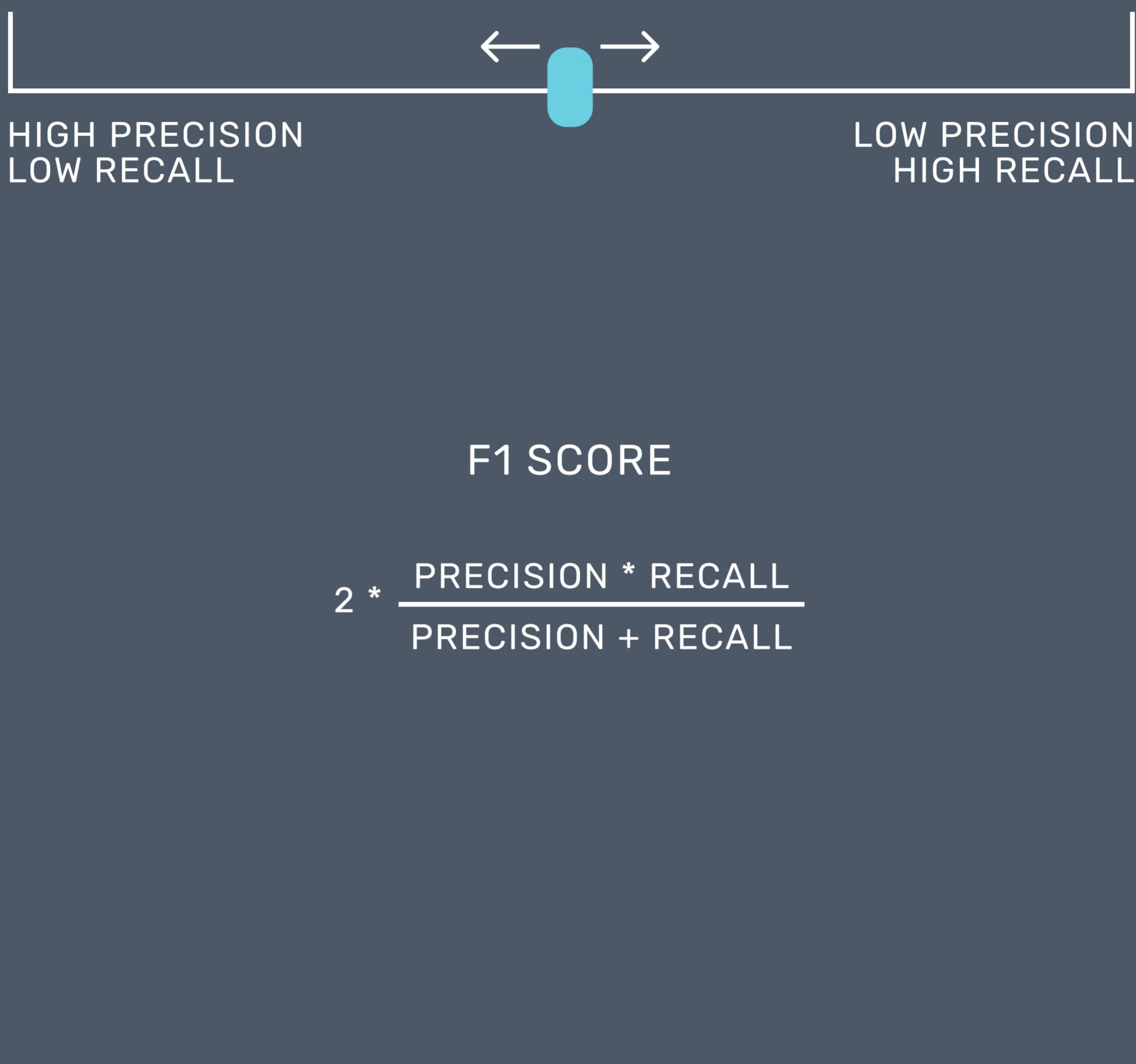


## PRECISION AND RECALL

From here, we can compute two types of metrics: *precision* and *recall*.

The reason for the curiously high accuracy was the dominance of the true negatives, diluting the rest. With precision and recall, we remove the focus on these true negatives and instead give all the attention to the other three categories.

We can see from the precision and recall scores that they offer a more reliable performance indicator than accuracy.



## F1 SCORE

However, we still need to strike a balance between precision and recall.

To illustrate its importance, suppose the model from the credit card fraud prediction achieved high recall and low precision. This would lead to a high number of false positives. This is good for detecting as many fraud cases as possible, but comes at the expense of flagging non-fraud cases as frauds.

Conversely, if the model achieved high precision and low recall, this would result in a high number of false negatives. This is good for correctly classifying the non-fraud cases but will miss out on real fraud cases.

We can address this problem with the *F1 Score*, which provides a balanced emphasis on precision and recall.

## TRAINING DATA

		PREDICTED VALUE	
		0	1
ACTUAL VALUE	0	TN 11	FP 2
	1	FN 1	TP 10

ACCURACY

88%

PRECISION

83%

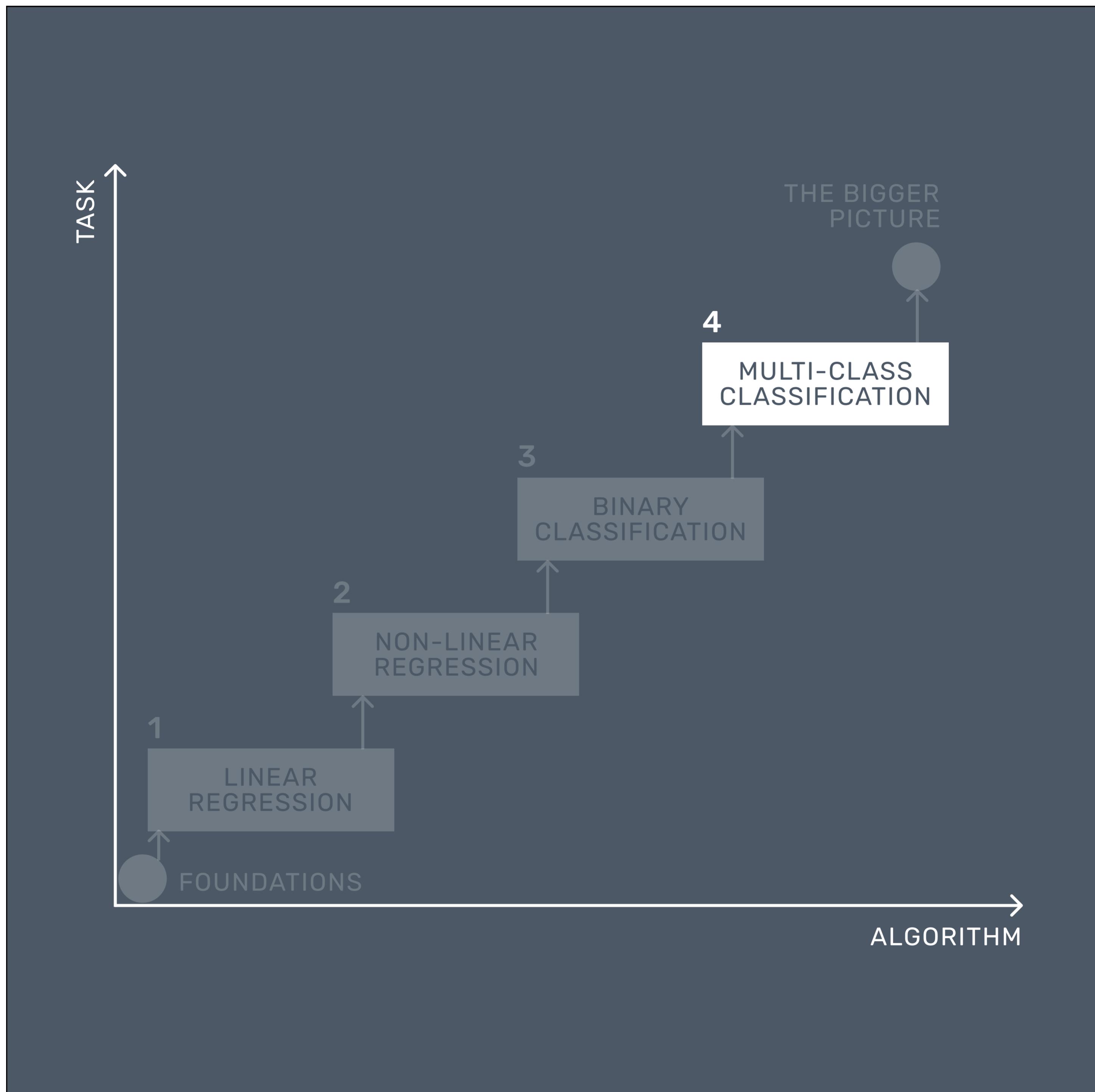
RECALL

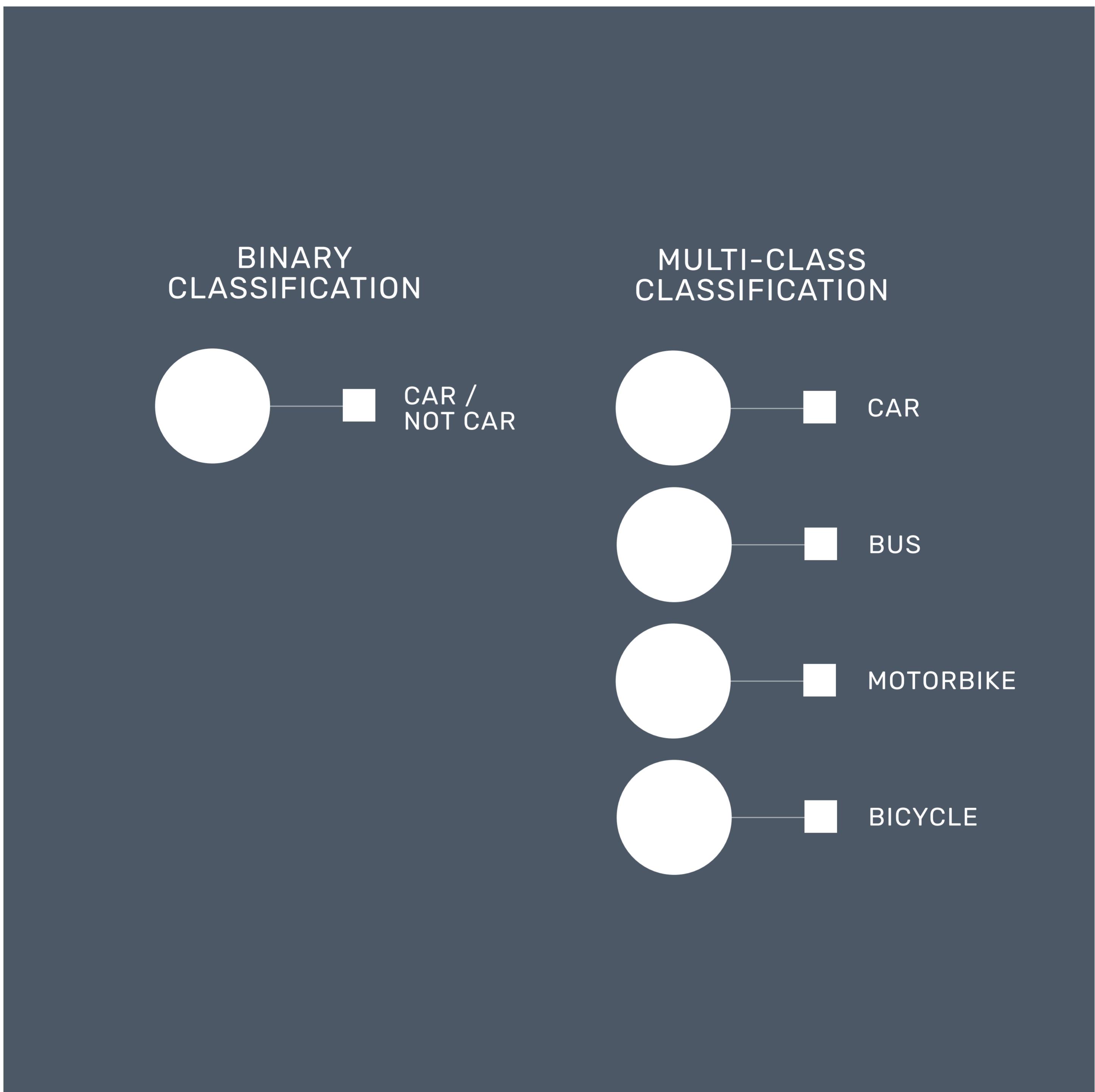
91%

## PERFORMANCE

Going back to our hotels dataset, we get an accuracy score that's comparable with precision and recall, indicating that our dataset is balanced.

# 4 - MULTI-CLASS CLASSIFICATION





## BINARY VS. MULTI-CLASS CLASSIFICATION

In the last chapter, our task was to predict between two possible classes. But what if we had a label with more than two classes?

Such a task is called *multi-class classification*. To make the neural network work for this type of task, we'll need to modify its architecture slightly. This is what this chapter is about.



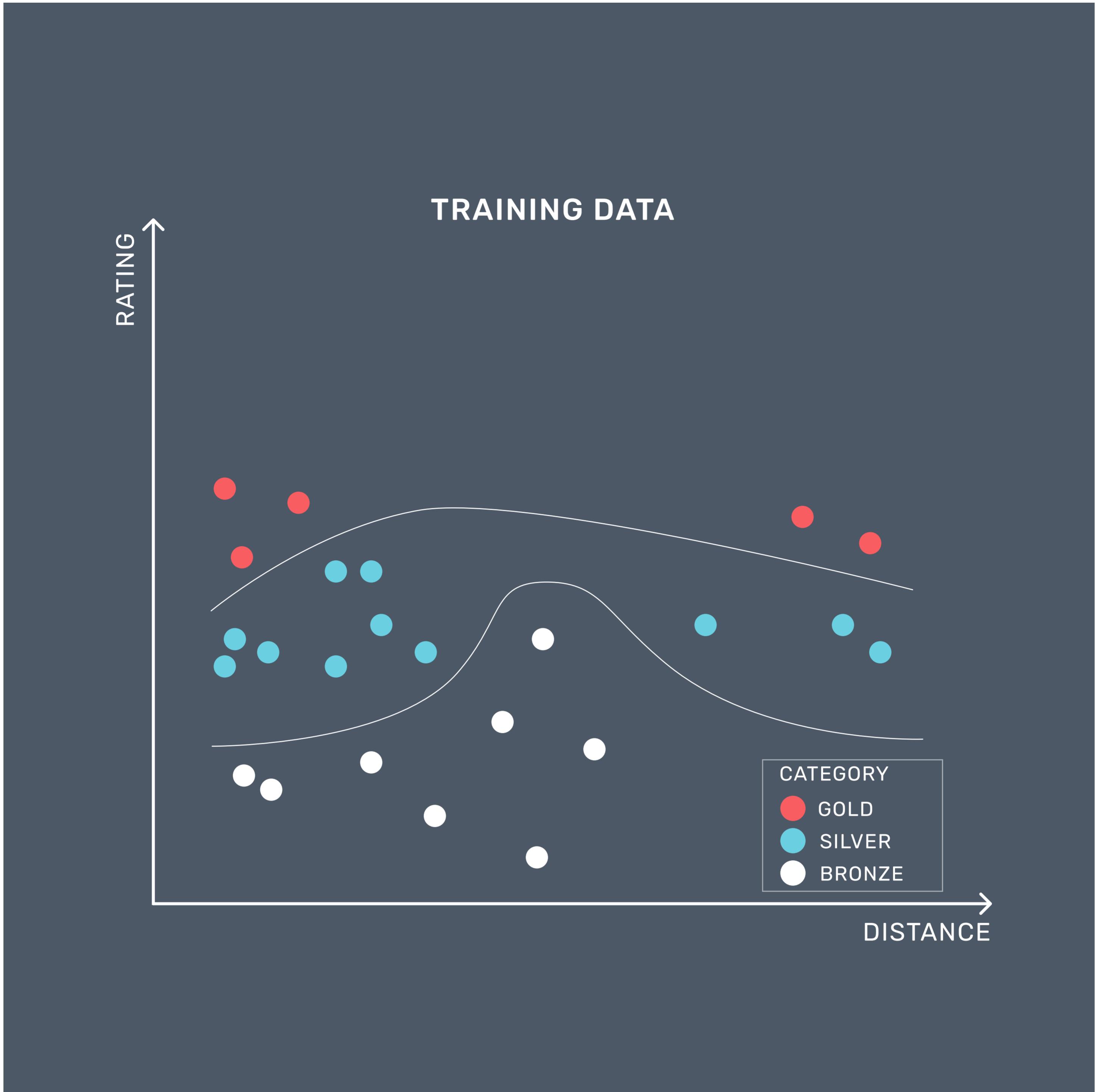
	DIST (MI)	RATING	CATEGORY
TRAINING DATA (24)	0.2	3.5	SILVER
	0.2	4.8	GOLD
	0.5	3.7	SILVER
	0.7	4.3	GOLD
	0.8	2.7	BRONZE
	1.5	3.6	SILVER
	1.6	2.6	BRONZE
	2.4	4.7	GOLD
	3.5	4.2	SILVER
	3.5	3.5	SILVER
	4.6	2.8	BRONZE
	4.6	4.2	SILVER
	4.9	3.8	SILVER
	6.2	3.6	SILVER
	6.5	2.4	BRONZE
	8.5	3.1	BRONZE
	9.5	2.1	BRONZE
	9.7	3.7	BRONZE
	11.3	2.9	BRONZE
	14.6	3.8	SILVER
	17.5	4.6	GOLD
	18.7	3.8	SILVER
	19.5	4.4	GOLD
	19.8	3.6	SILVER
TEST DATA (8)	0.3	4.6	GOLD
	0.5	4.2	GOLD
	1.1	3.5	SILVER
	1.2	4.7	GOLD
	2.7	2.7	BRONZE
	3.8	4.1	SILVER
	7.3	4.6	BRONZE
	19.4	4.8	GOLD

## THE DATASET

We'll use the same dataset as Chapter 3 and again, have a new label called *category*. Suppose there's a certification agency that classifies hotels into three categories—*gold*, *silver*, and *bronze*.

Our goal is to predict the category for a given hotel based on the features.

The label is a categorical variable, which means that order is not implied. Though the class names suggest that some order may exist, we just want to classify them without worrying about which class is better than which.



## THE DATASET

Here is shown the plot of the actual class for each training data point, along with an example of hand-drawn boundaries separating the three classes. This is what we want our neural network to produce.

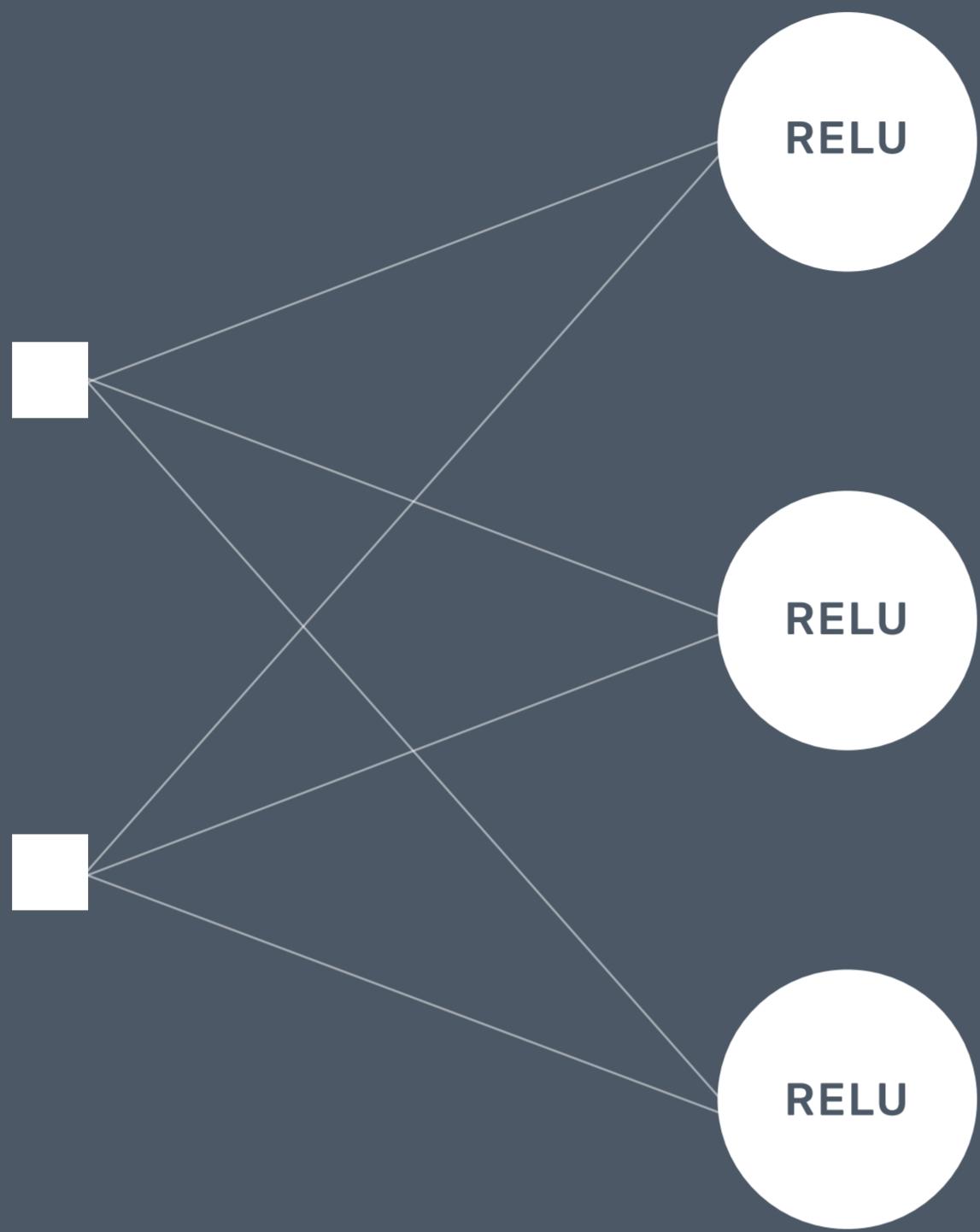
CATEGORY	GOLD	SILVER	BRONZE
SILVER	0	1	0
GOLD	1	0	0
SILVER	0	1	0
GOLD	1	0	0
BRONZE	0	0	1
...	...	...	...
GOLD	1	0	0
GOLD	1	0	0
SILVER	0	1	0
...	...	...	...
GOLD	1	0	0

## ONE-HOT ENCODING

First, we need to convert the classes into a numeric format. Since we have more than two classes this time, converting them into 0s and 1s won't work.

We need to use a method called *one-hot encoding*. Here, we create a new column for each class. Then we treat each column as a binary classification output, assigning the value 1 for yes and 0 for no.

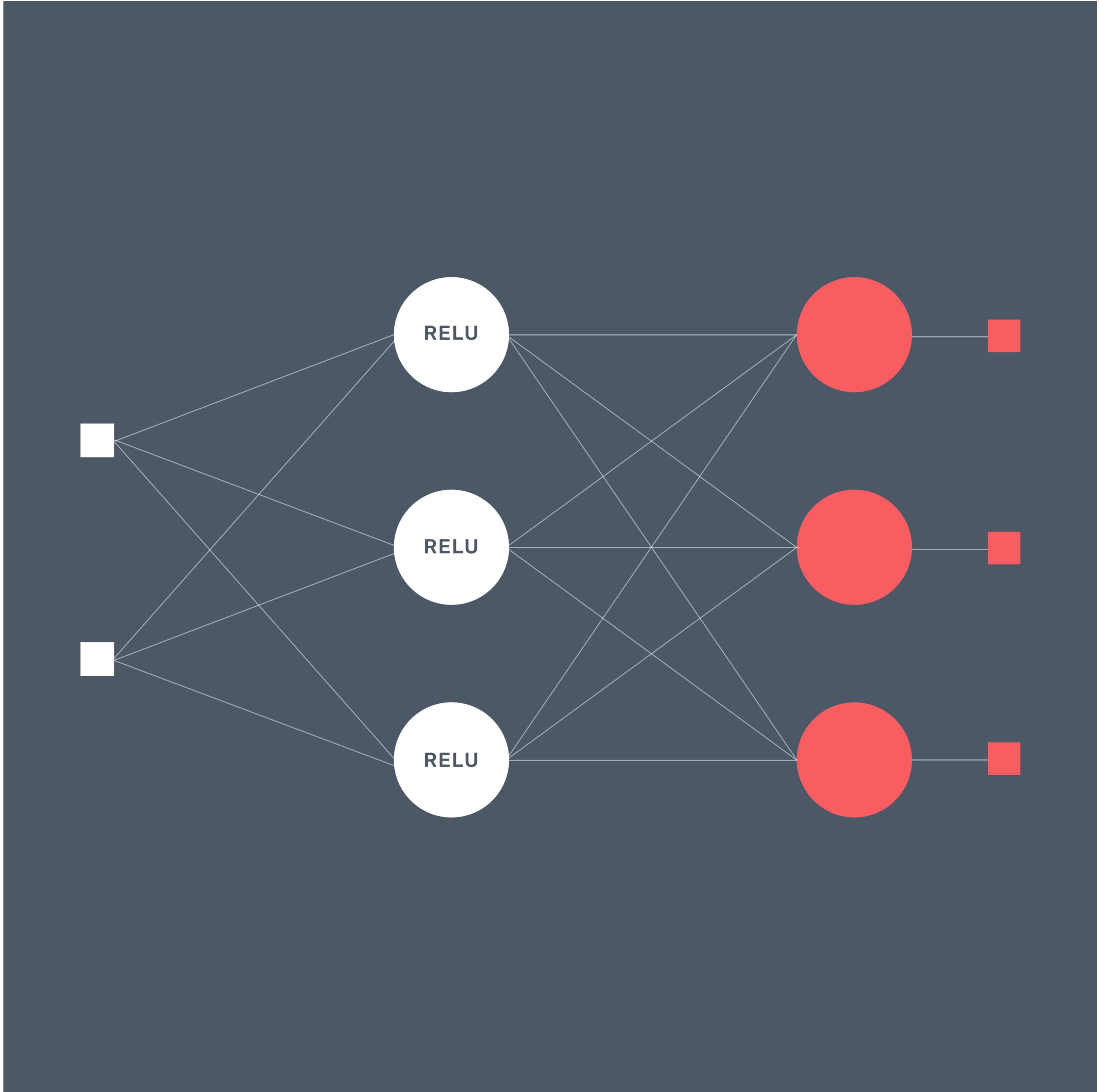
Note that if we also had features of the categorical type, we would apply the same method. Suppose we had a feature called *view* with possible values of *pool*, *garden*, and *none*. This will translate into three one-hot encoded features, one for each class.



## NEURAL NETWORK ARCHITECTURE

We'll start building the neural network with the input layer, where there are no changes from the previous chapter.

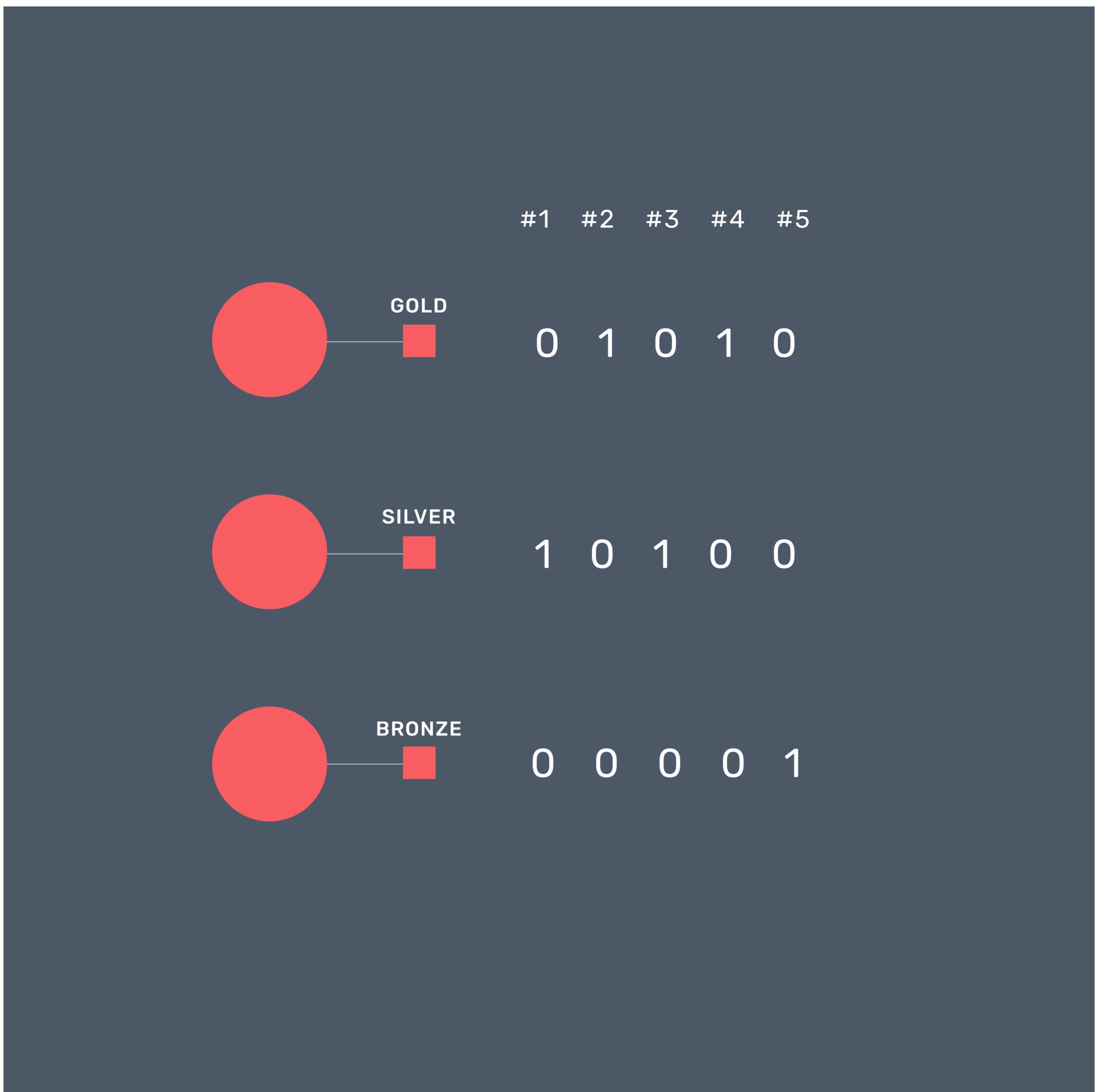
We'll also retain the same number of hidden layers and units, and keep ReLU as the activation function.



## NEURAL NETWORK ARCHITECTURE

As for the output layer, we need to make some changes. A single-neuron output only works for binary classification.

For multi-class classification, we need to have the same number of neurons as classes. Each neuron represents one class, and its output is mapped to the corresponding one-hot encoded column. To understand this, let's look at some examples.

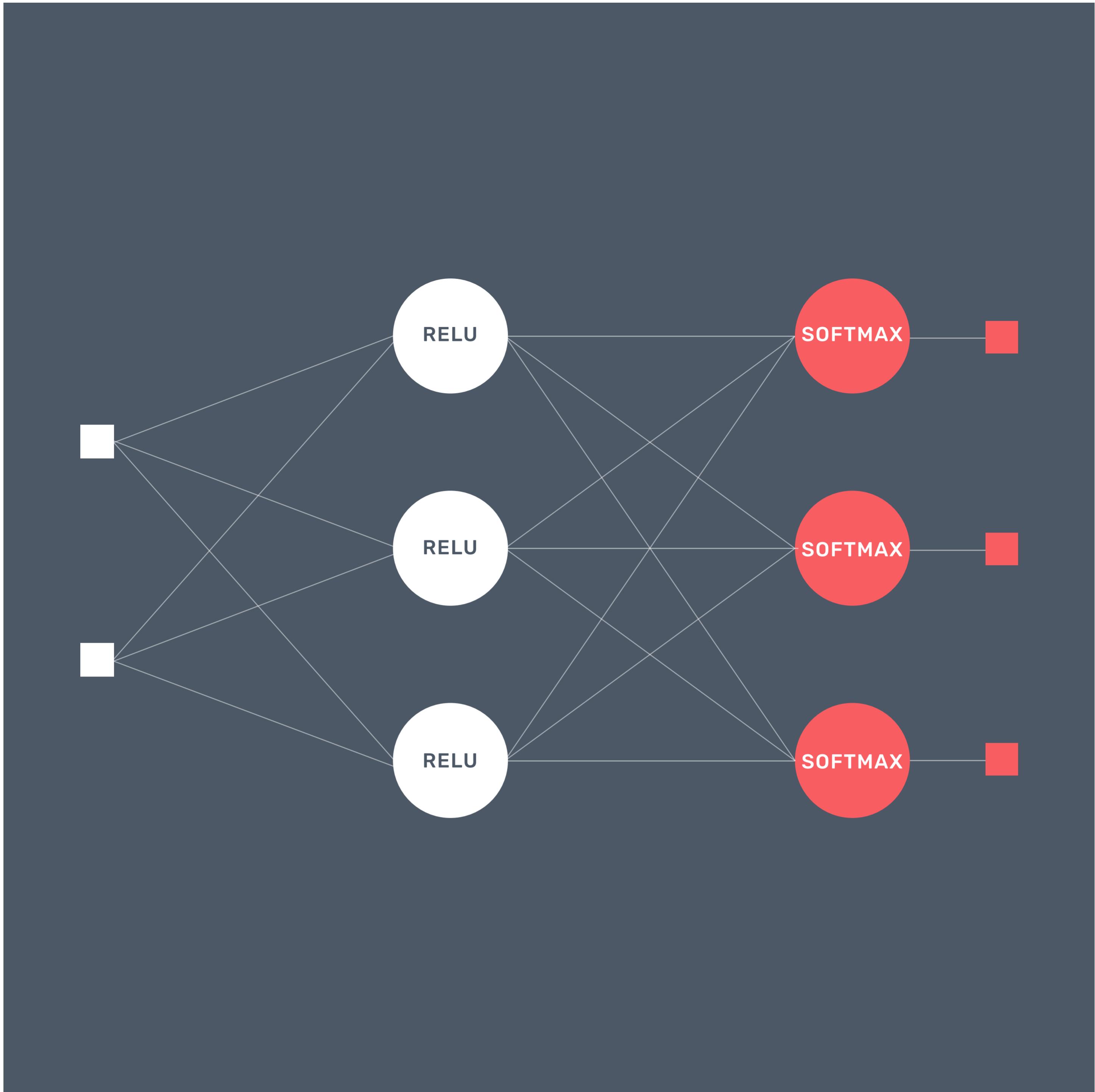


## OUTPUT VALUES

Here we have the labels of the first five data points—silver, gold, silver, gold, and bronze.

Take the first data point as an example. For the silver class, the neural network should ideally predict 0, 1, and 0 for the first, second, and third neurons.

In short, for each data point, the neuron of the actual class should output 1 while other neurons should output 0.



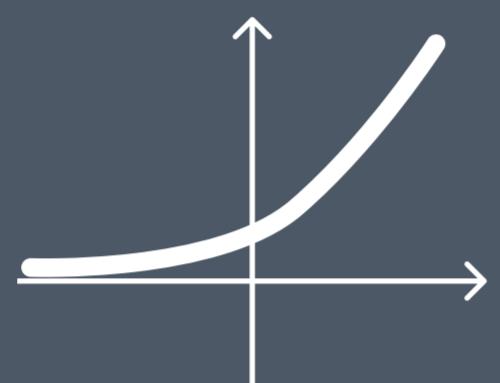
## OUTPUT LAYER ACTIVATION

There is also a change in the activation functions in the output layer. We'll introduce a new function called *softmax*.

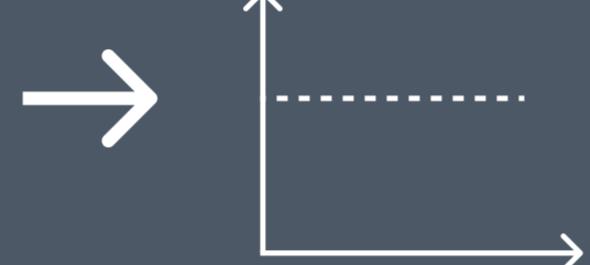
## SOFTMAX ACTIVATION



EXPONENT

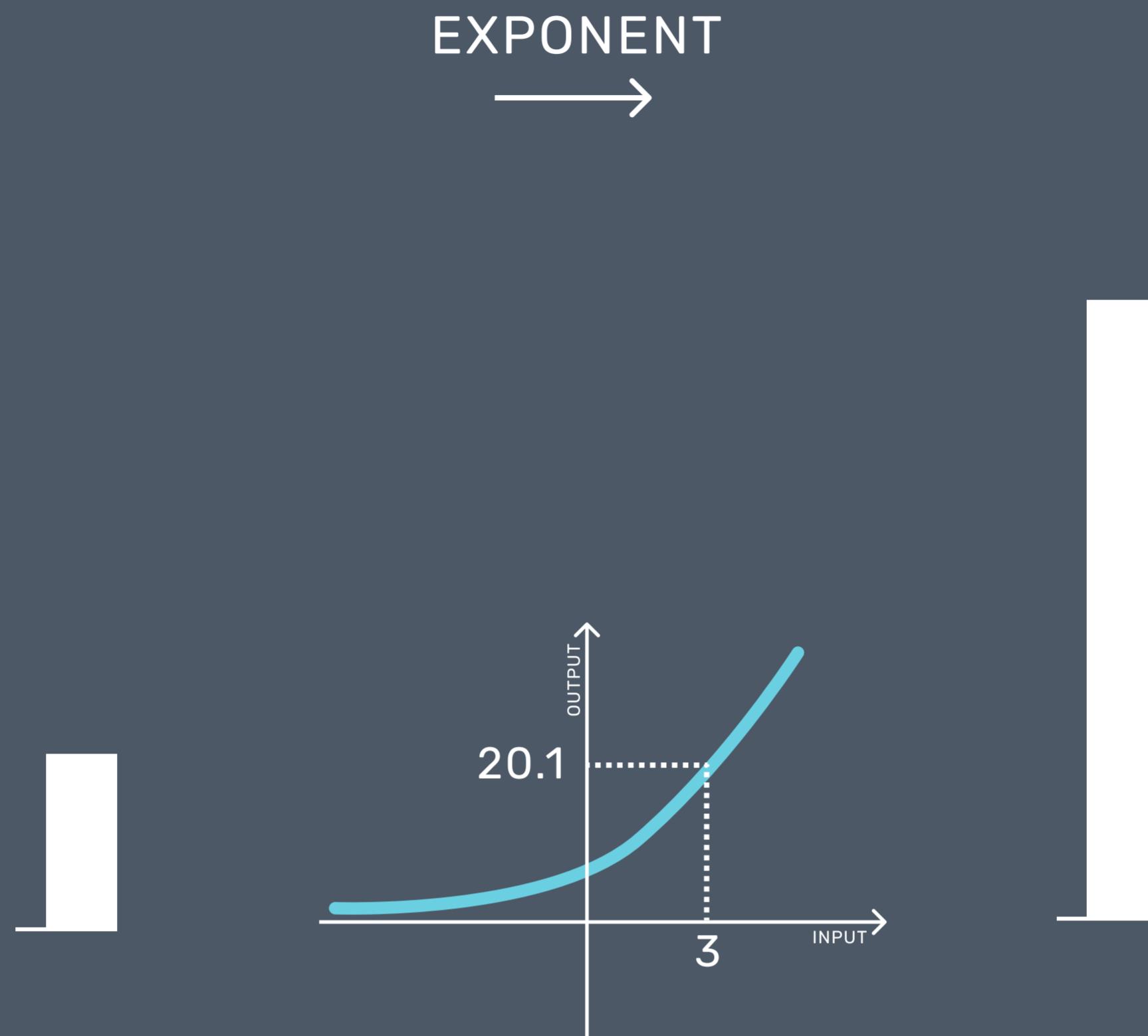


NORMALIZE



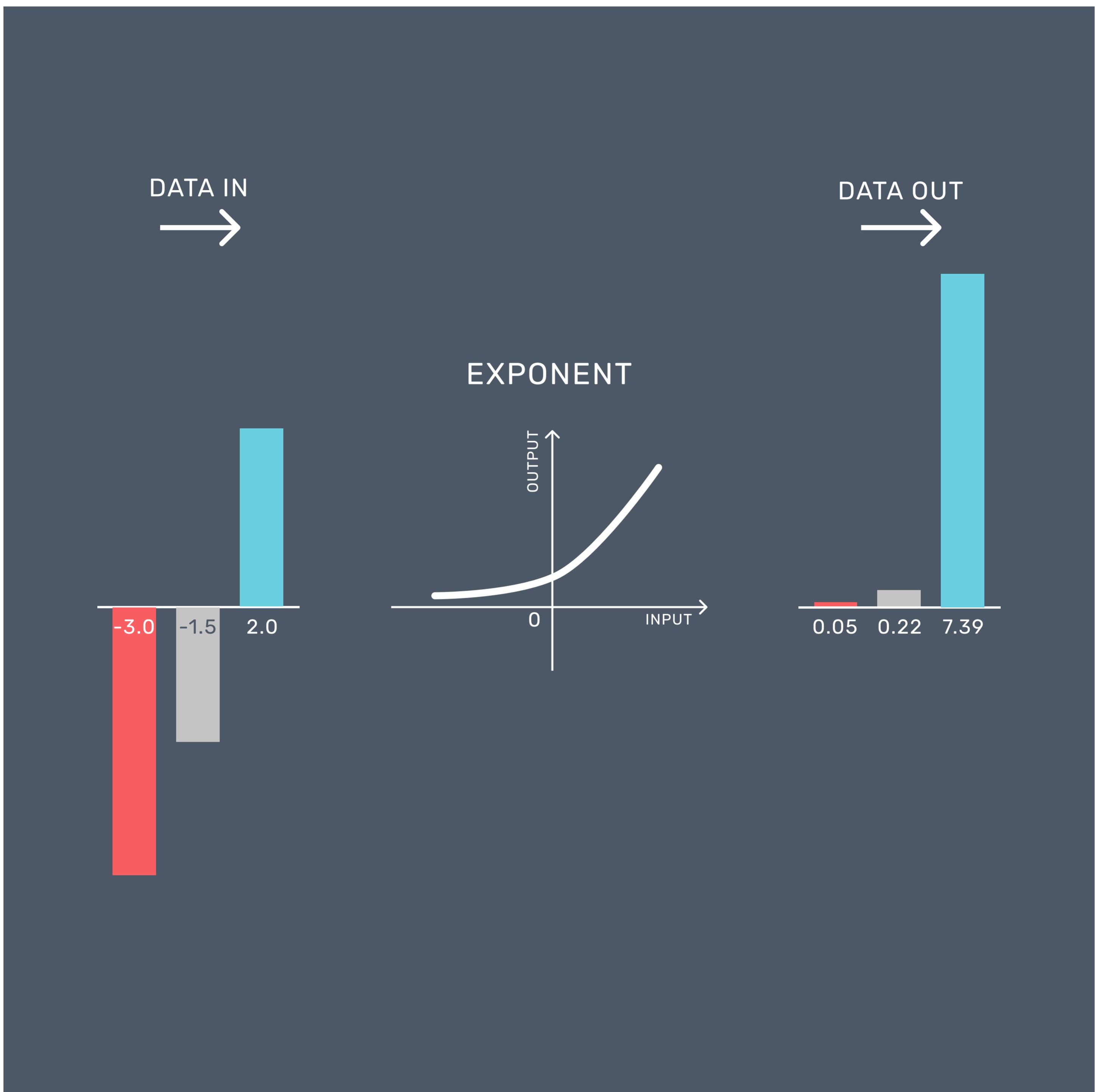
## SOFTMAX ACTIVATION FUNCTION

The softmax activation performs a two-step computation on its input: *exponentiation* and *normalization*.



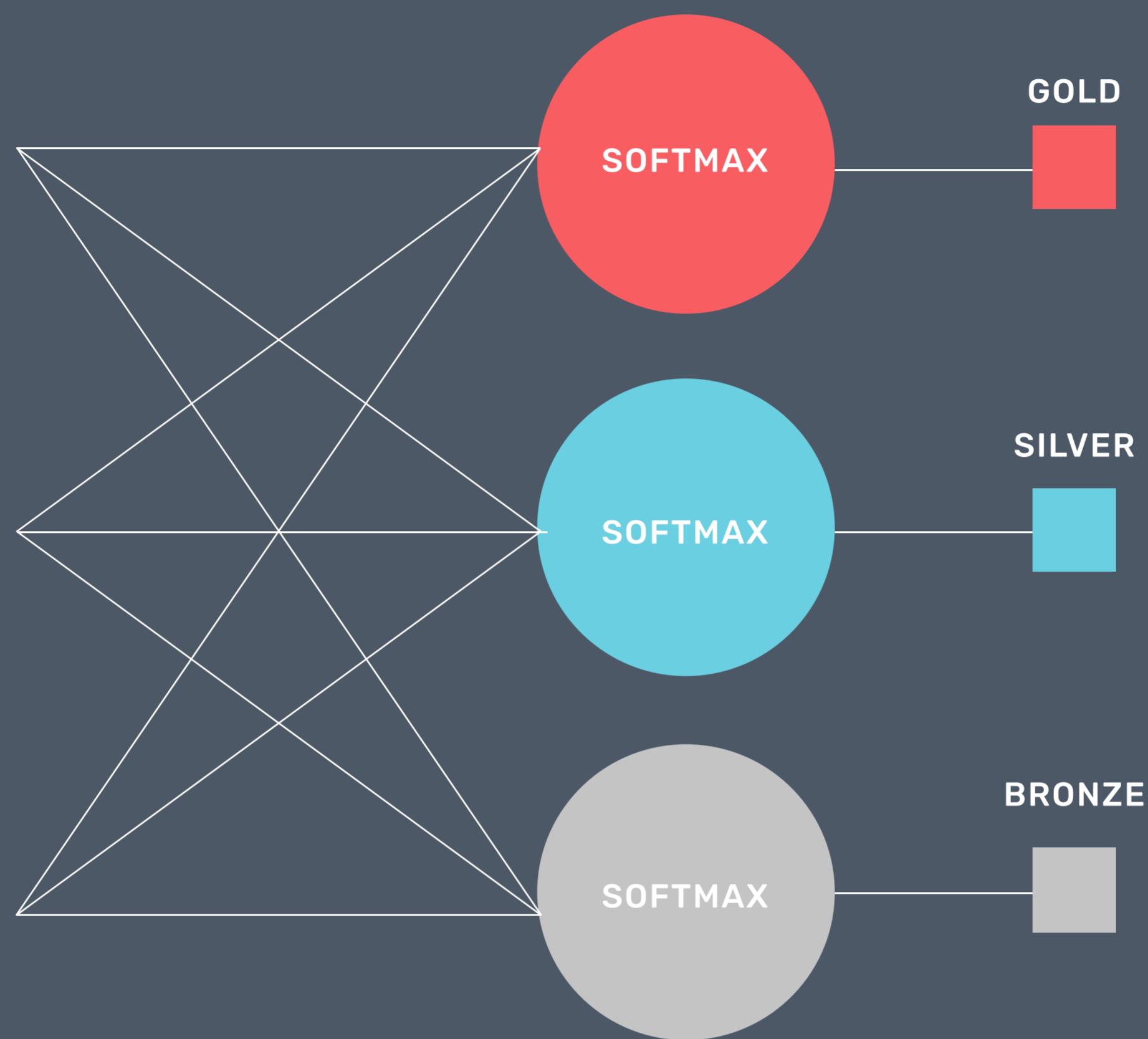
## EXPOENT

In the first step, it turns the input into its exponent.



## EXPONENT

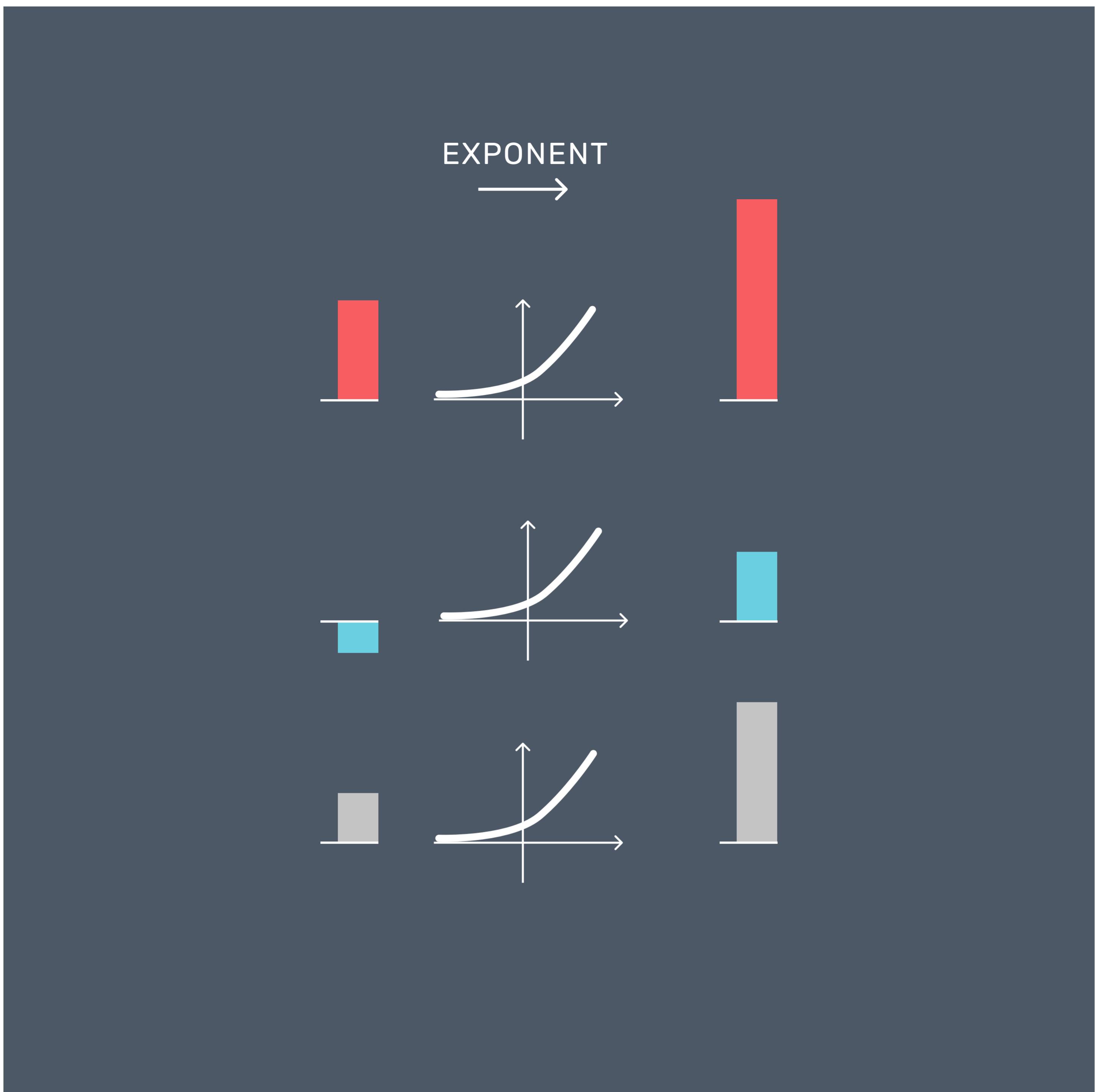
The effect of exponentiation is to transform any number into a positive number. Additionally, it amplifies the large inputs more than the small ones.



## THE FULL LAYER

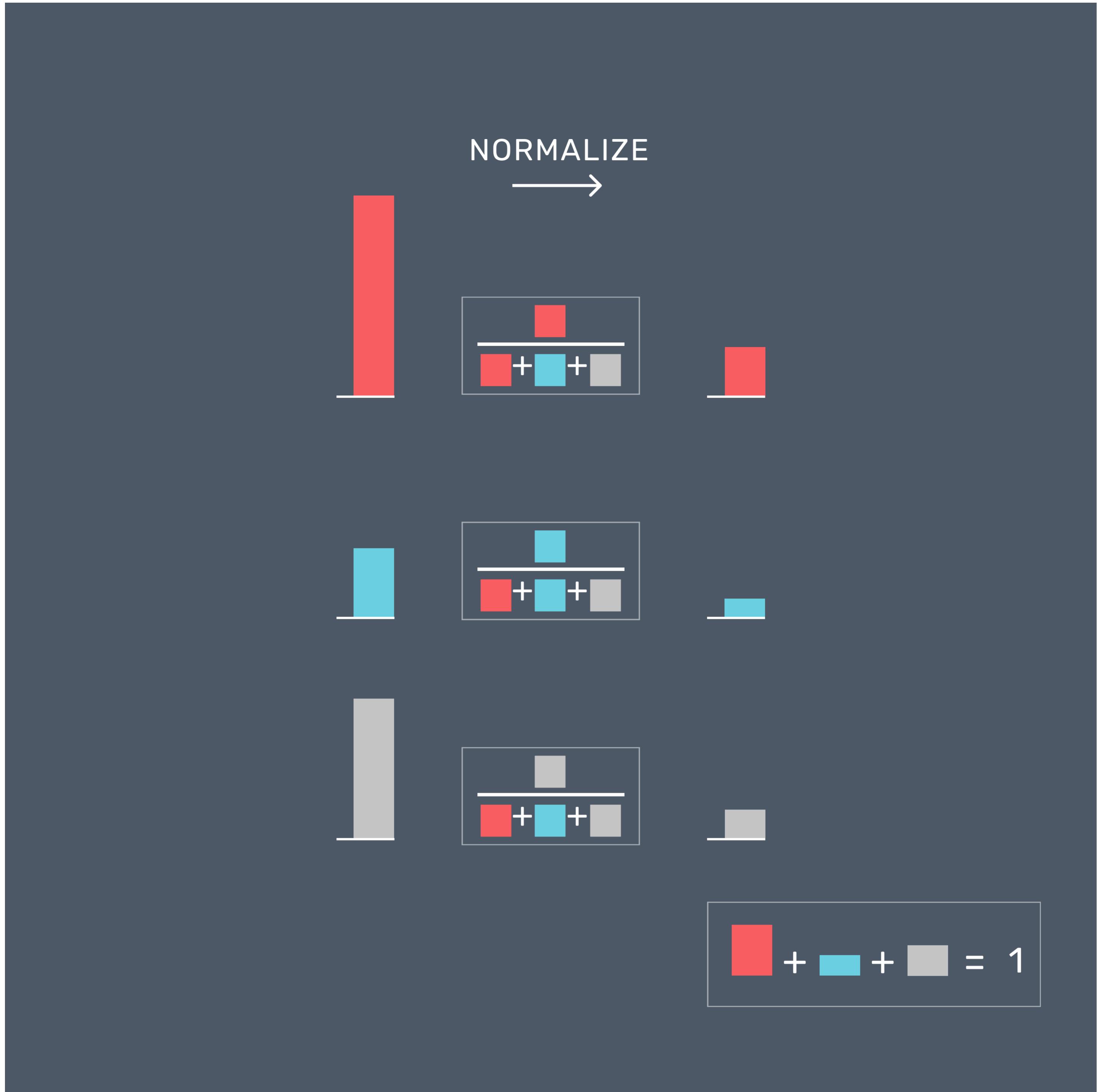
To understand how the second step, normalization, works, we need to look at the layer as a whole.

Here we have the three units of neurons, one for each class.



## EXONENT

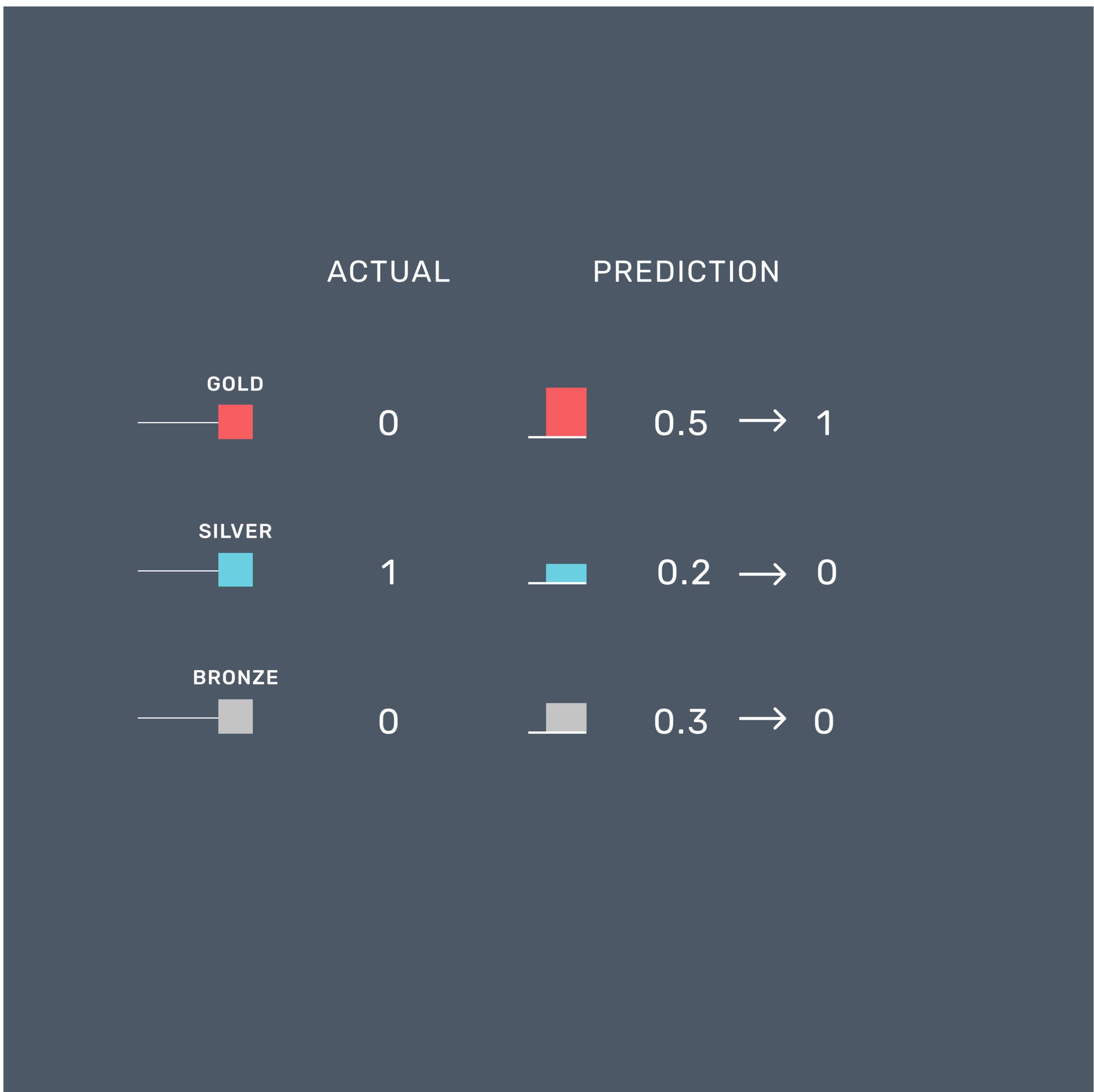
Each neuron performs the exponentiation on its input, which then becomes the input for the normalization step.



## NORMALIZE

In the normalization step, each input is divided by the sum of all inputs. This becomes the output of the neural network.

As a result, the sum of all outputs will always be 1. This is a useful outcome because we can now treat the outputs as probability values, as we did in Chapter 3.

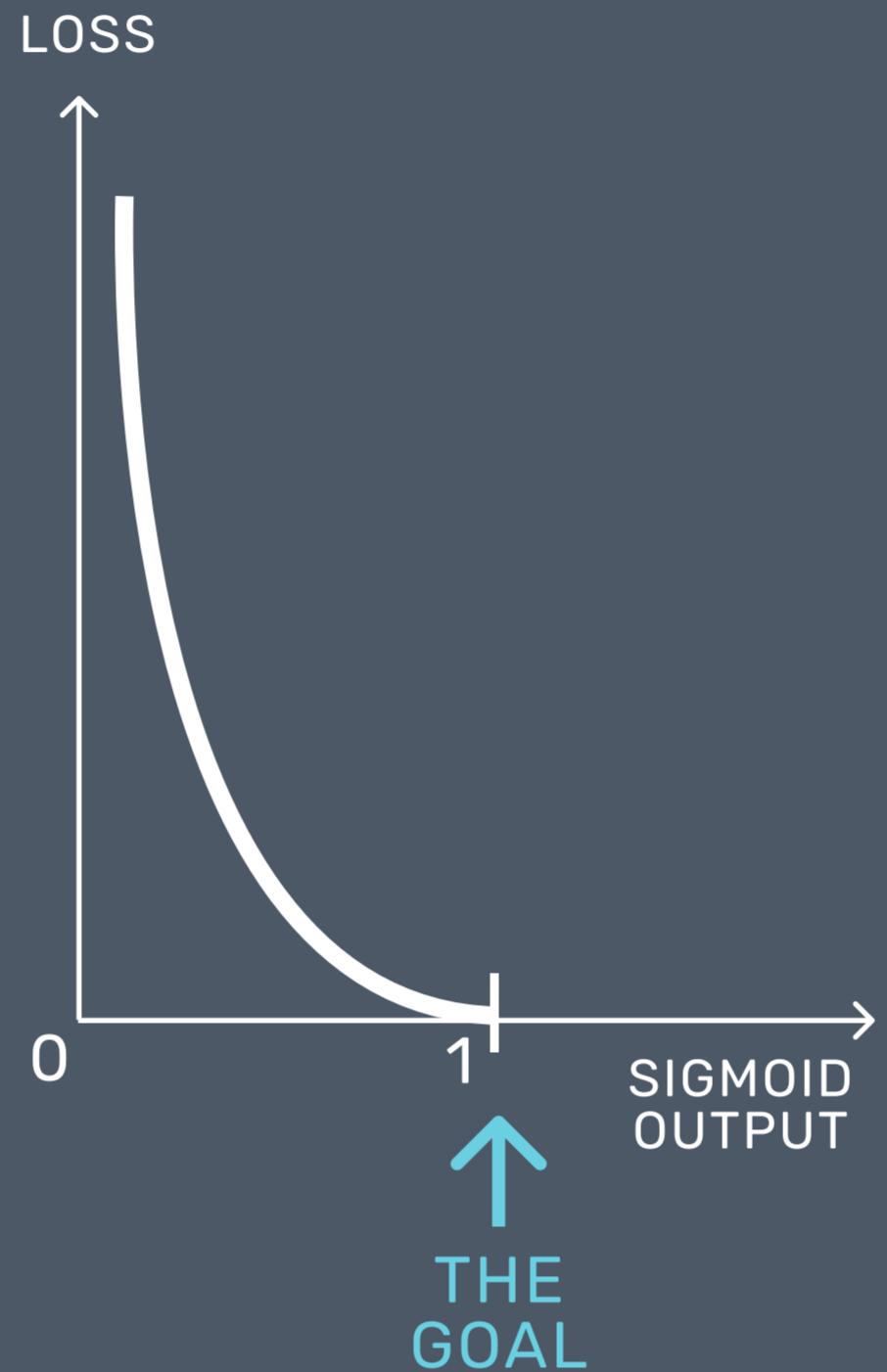


## EXAMPLE PREDICTION

Let's take an example where the actual class is silver. And suppose that each neuron's softmax activation produces 0.5, 0.2, and 0.3.

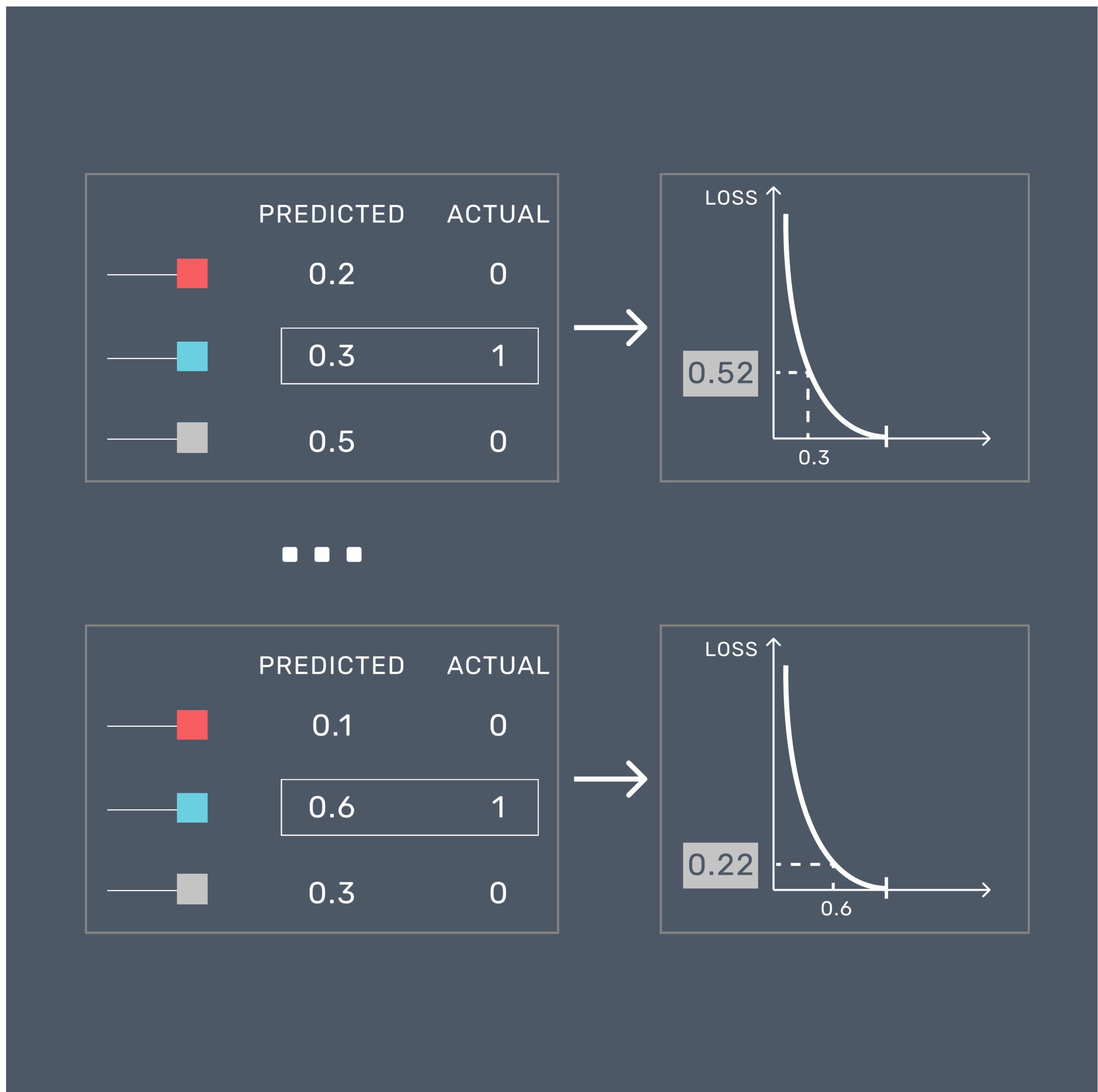
Treating them as probabilities, we assign 1 to the neuron with the largest output and 0 to the other neurons.

In this example, the predicted class does not match the actual class. This brings us to the next discussion - the loss function.



## LOSS FUNCTION

The loss function we'll be using is called *categorical cross entropy*. It is essentially the same as the binary cross entropy loss function we used in Chapter 3, but a generalized version. The categorical cross entropy works for any number of classes, unlike its binary counterpart.

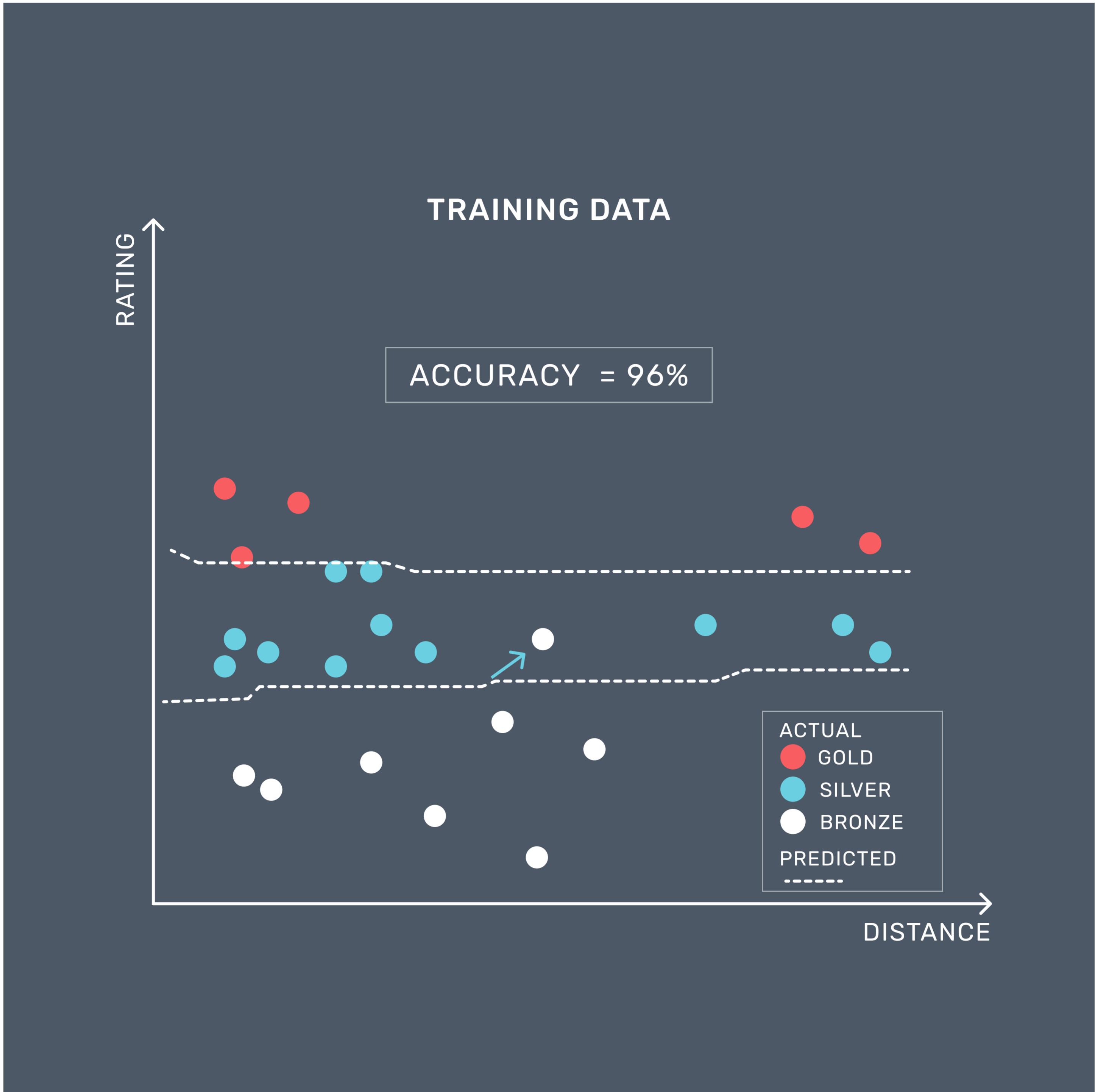


## LOSS FUNCTION EXAMPLE

Let's look at an example. Here, the actual class is silver. So, we want the neural network to output the highest probability at the second neuron.

In one of the earlier training epochs, we can see that the output at the second neuron is 0.3, which isn't very good. This results in a high loss value.

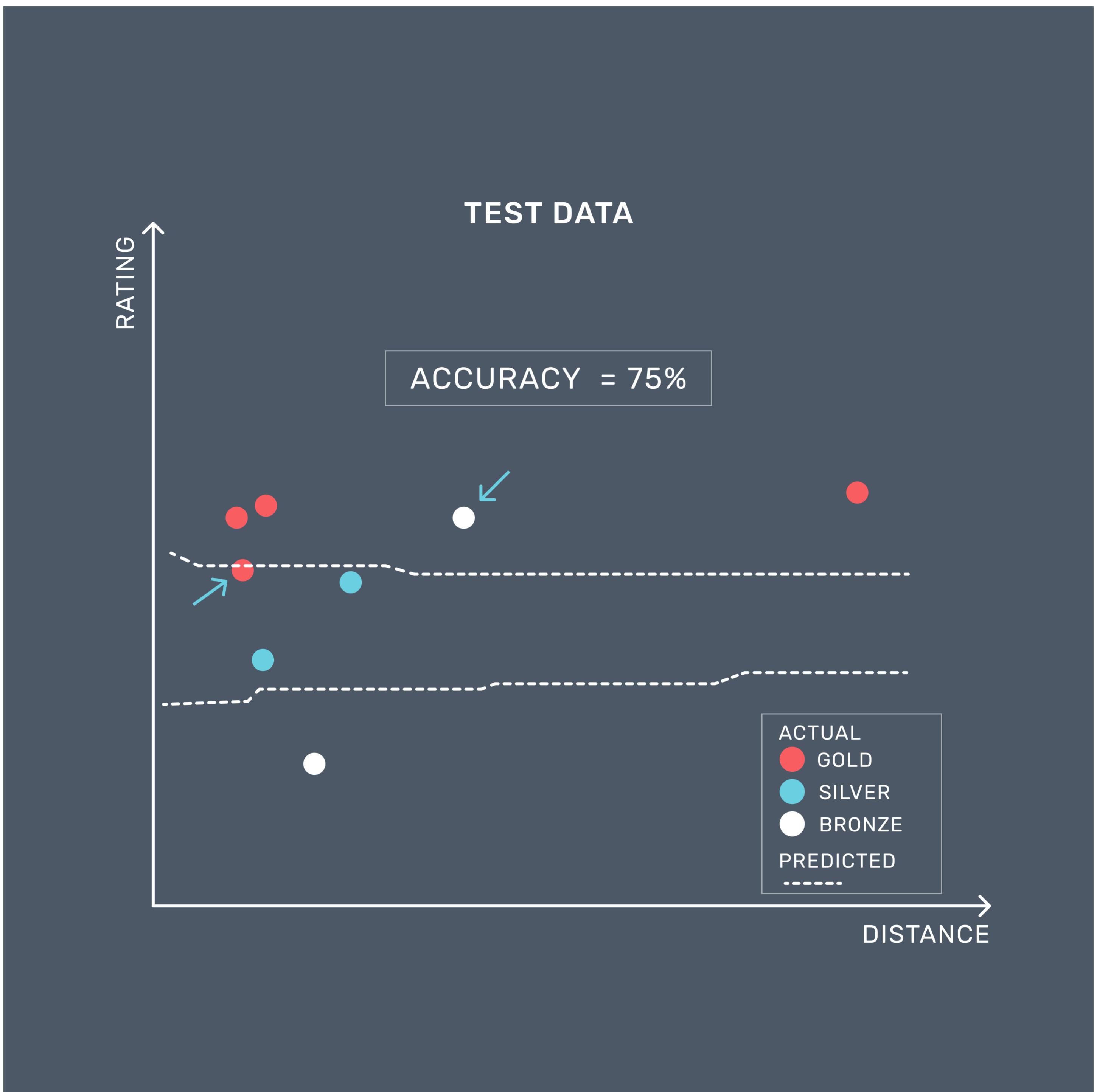
In one of the later epochs, this neuron produces an output of 0.6. This indicates an improvement, which is reflected in the decreasing loss value.



## TRAINING PERFORMANCE

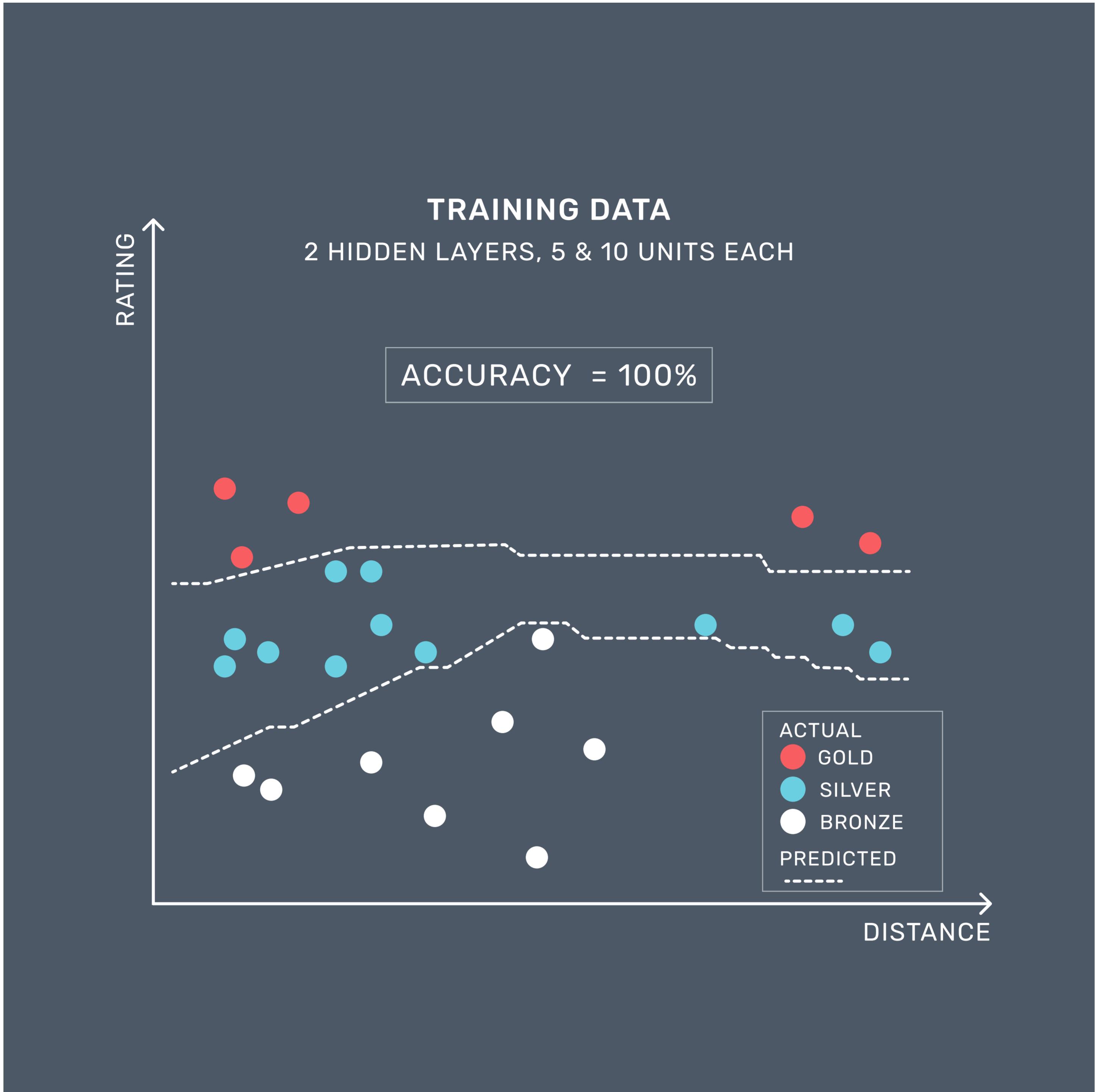
Using the accuracy metric, we can see that the model does pretty well on the training dataset.

The plot also shows the decision boundary of our trained neural network, representing the predictions.



## TEST PERFORMANCE

The prediction on the test data also shows a respectable performance, given the limited number of data points.



## A BIGGER NETWORK

As in Chapter 3, we can further improve the performance by adding more layers and units.

The plot here shows the decision boundaries after modifying the neural network to contain two hidden layers with five and ten units of neurons each. The granularity of its predictions increased, resulting in a more well-defined and accurate prediction curve.

		Predicted Value		
		Gold	Silver	Bronze
Actual Value	Gold	True	False	False
	Silver	False	True	False
	Bronze	False	False	True

## CONFUSION MATRIX

The confusion matrix for this task is now expanded to a 3-by-3 matrix. The diagonal cells account for the correct predictions for each class, while the remaining cells account for the incorrect predictions.

## PRECISION (GOLD)

		PREDICTED		
		GOLD	SILVER	BRONZE
ACTUAL	CAT	TRUE	FALSE	FALSE
	SILVER	FALSE	TRUE	FALSE
	BRONZE	FALSE	FALSE	TRUE

→

TRUE	+	FALSE	+	FALSE
------	---	-------	---	-------

## RECALL (GOLD)

		PREDICTED		
		GOLD	SILVER	BRONZE
ACTUAL	GOLD	TRUE	FALSE	FALSE
	SILVER	FALSE	TRUE	FALSE
	BRONZE	FALSE	FALSE	TRUE

→

TRUE	+	FALSE	+	FALSE
------	---	-------	---	-------

## PRECISION AND RECALL

For multi-class classification, each class will have its own set of precision and recall metrics. Here we have an example for the gold class.



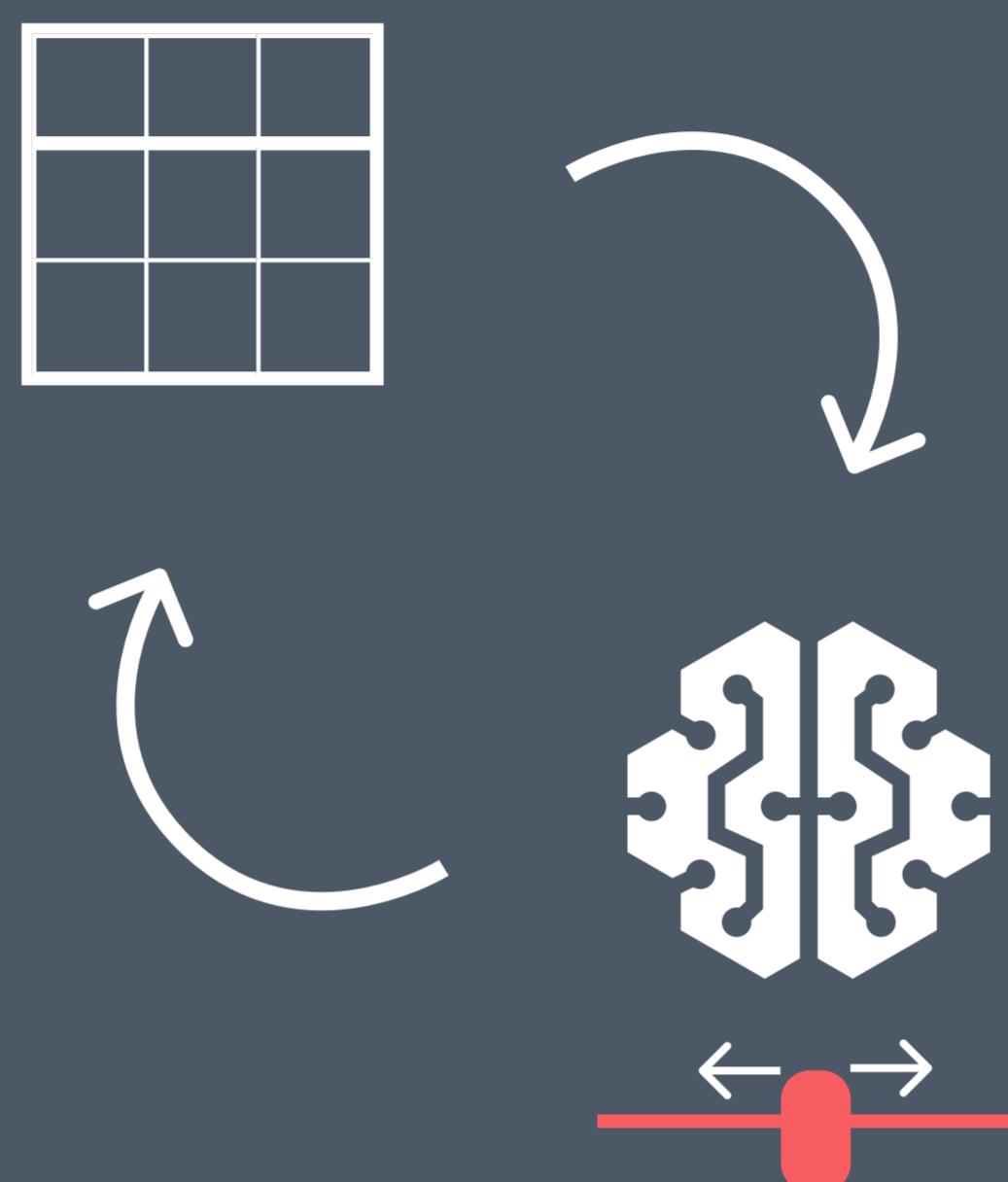
## HYPERPARAMETERS

And with that, our fourth and final task is complete.

For the remainder of this chapter, we'll look at various ways to improve our prediction results. We can divide them into two groups—*hyperparameters* and *data*.

First, let's look at hyperparameters. You may not have noticed, but we have covered some of them in the four tasks. Now let's take a closer look.

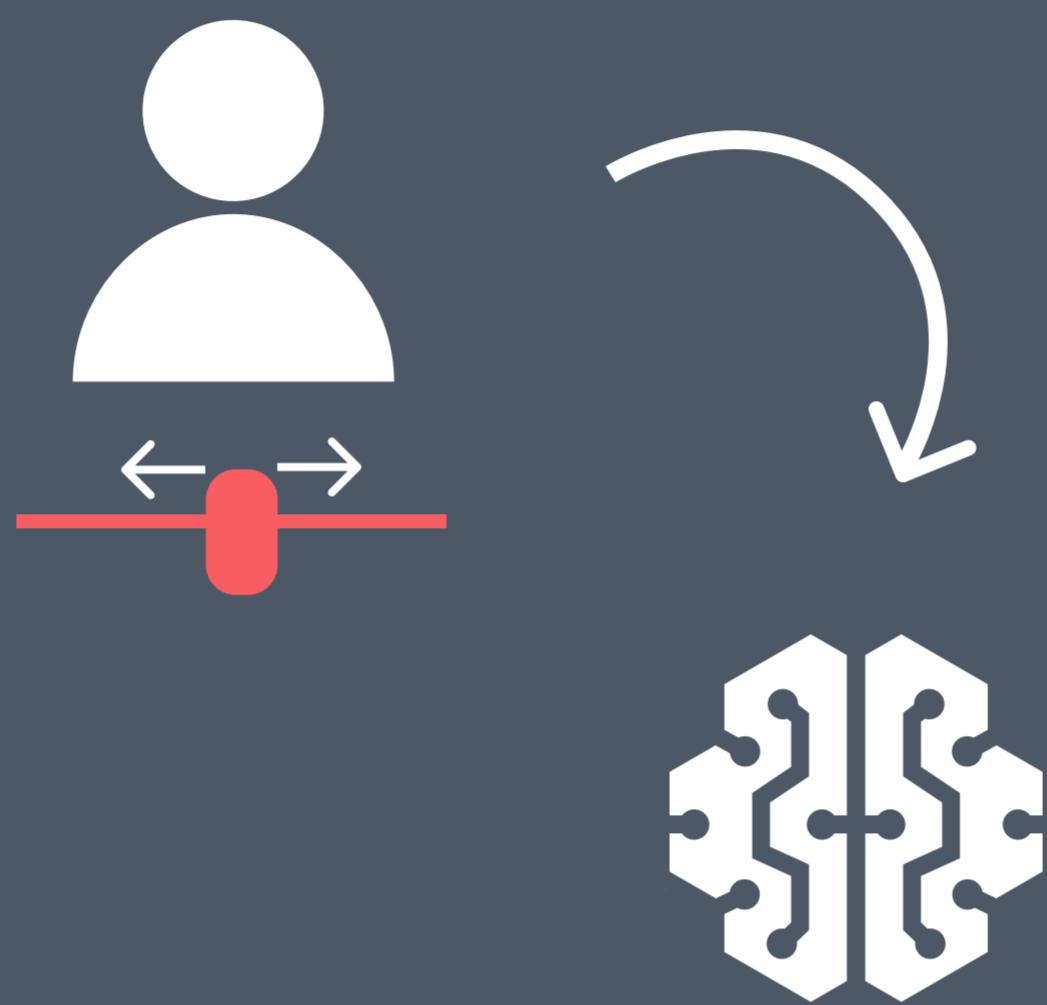
## PARAMETERS



## PARAMETERS

Recall that parameters—weights and biases—are learned by the neural network during training.

## HYPERPARAMETERS

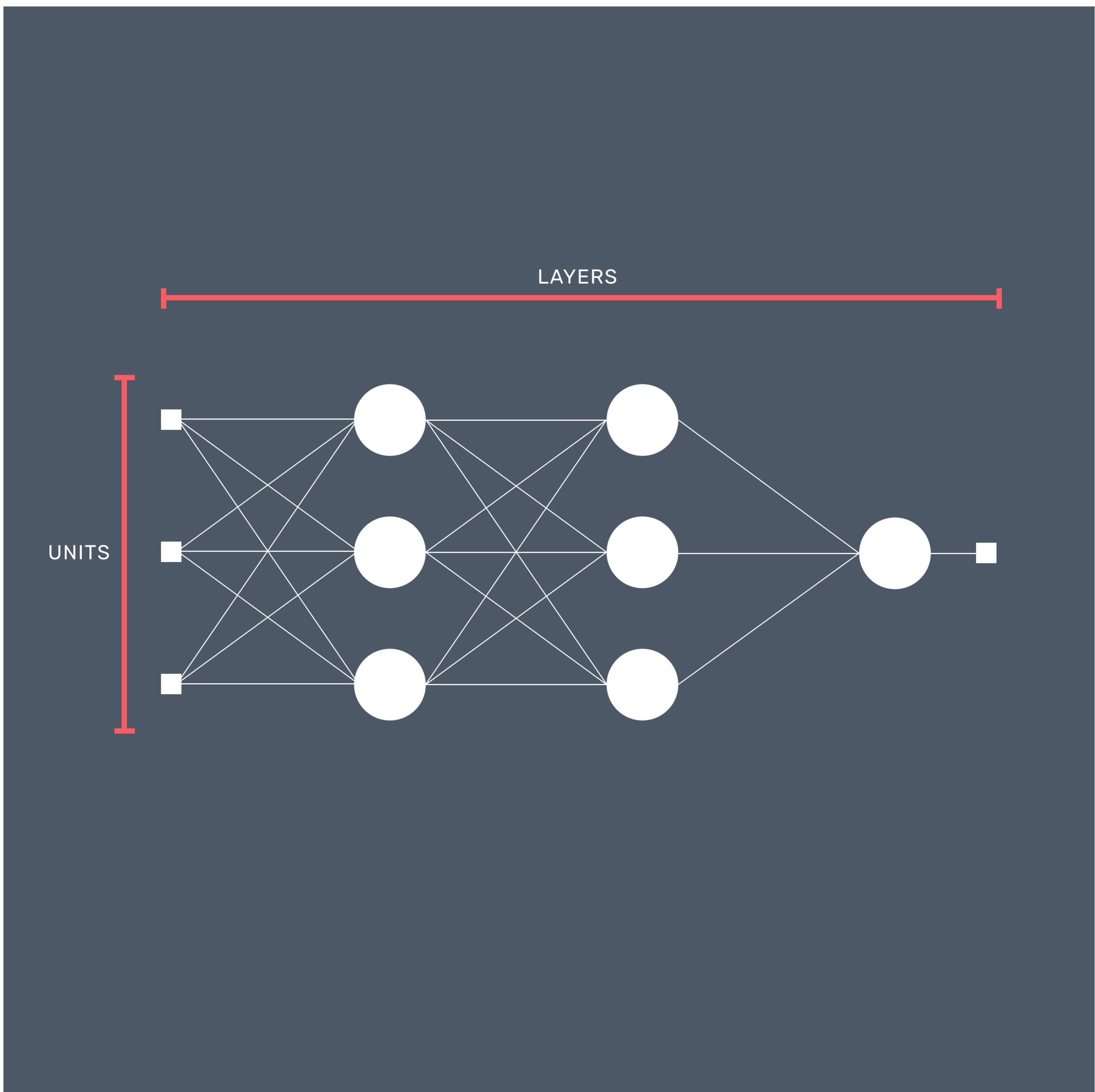


## HYPERPARAMETERS

On the other hand, hyperparameters are parameters that cannot be learned by the neural network. Instead, we need to provide them.

Choosing the right hyperparameters requires experience and is both an art and a science. They are no less important in successfully training a neural network.

There are many types of hyperparameters, so let's cover some of the key ones.

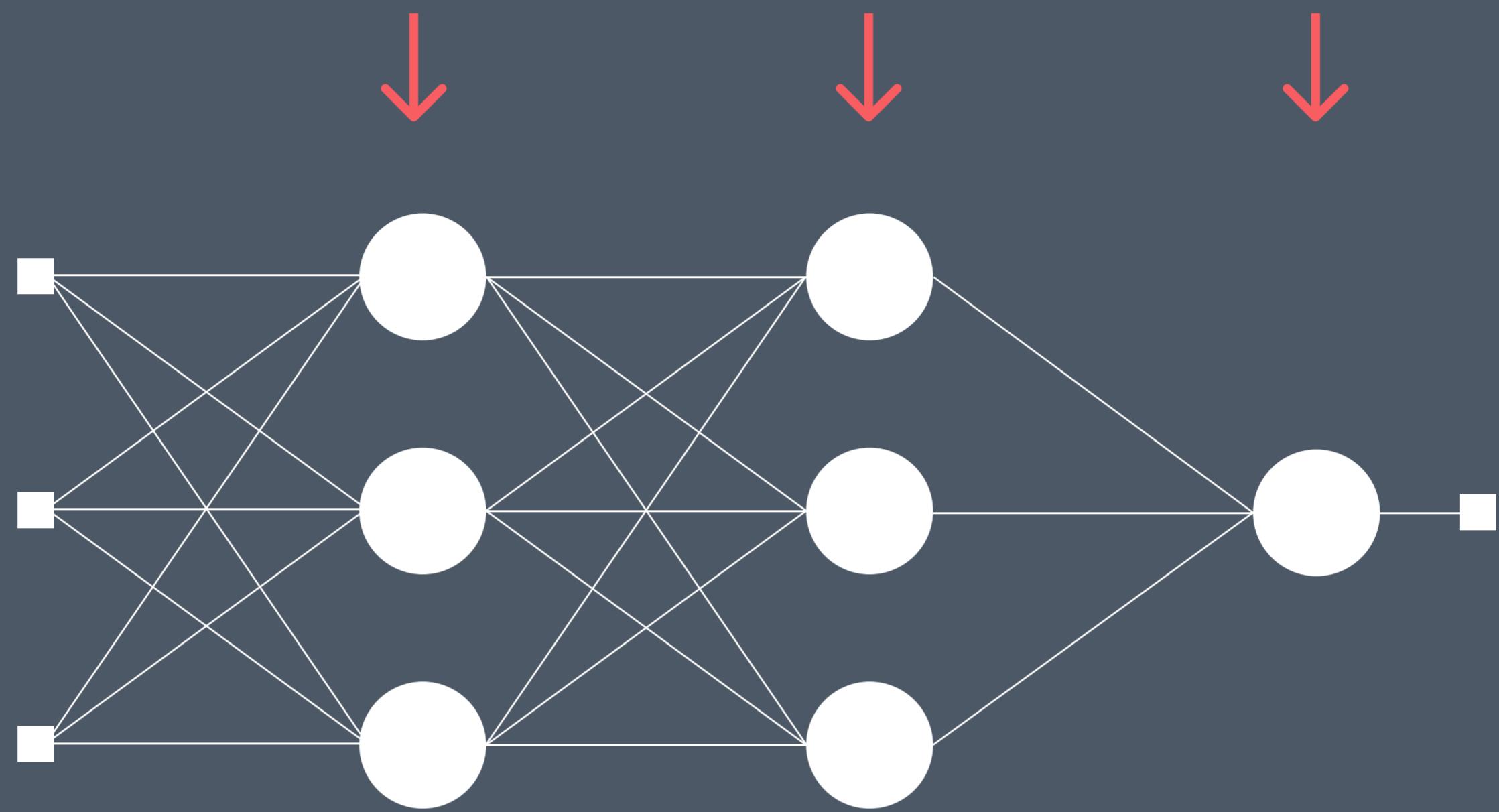


## SIZE

This is obvious, but one of the most important hyperparameters is the size of the neural network itself.

We change the size by increasing or decreasing the number of layers and units in each layer. A larger network is capable of handling more complex tasks and data.

## ACTIVATION FUNCTIONS



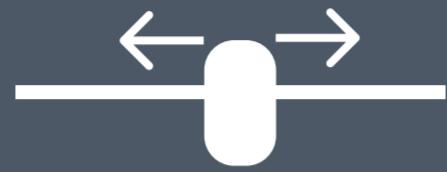
## ACTIVATION FUNCTION

Another hyperparameter is the type of activation function in each neuron.

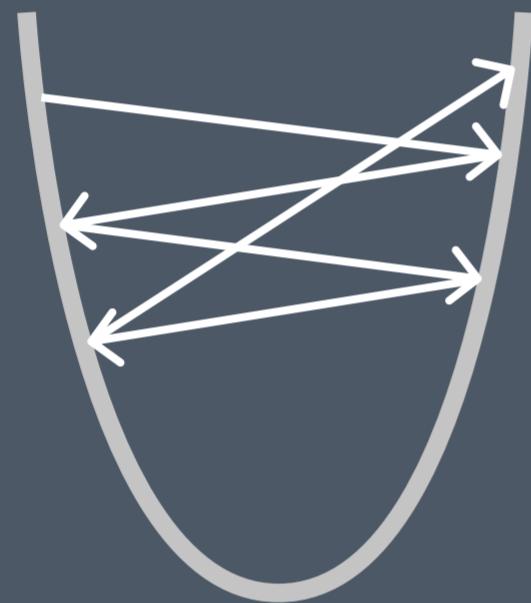
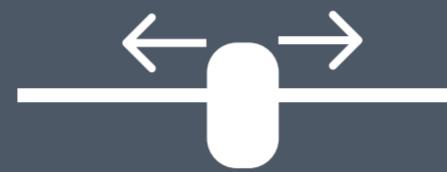
This depends on the nature of the task, but ReLU is typically used in the hidden layers, or at least is a good option to start with.

As for the output layer, this largely depends on the task, for example, whether it's a regression or classification task.

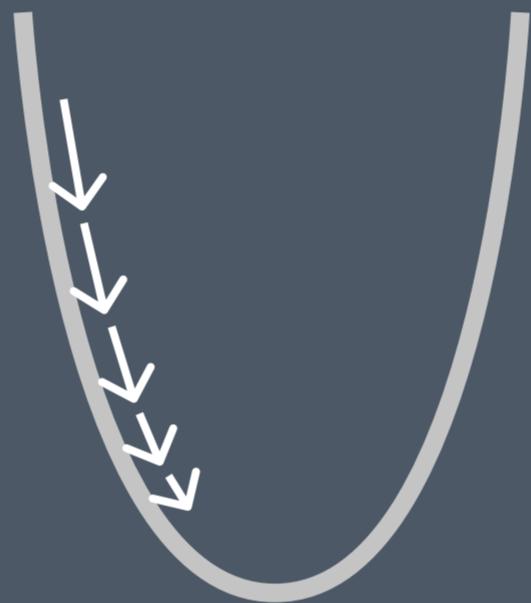
$$W_{\text{new}} = W_{\text{previous}} - \text{alpha} * d_w$$



$$b_{\text{new}} = b_{\text{previous}} - \text{alpha} * d_b$$



TOO  
BIG



JUST  
RIGHT



TOO  
SMALL

## LEARNING RATE

In all of our tasks, we have kept to a learning rate (or alpha) of 0.08. This value was chosen simply by trial and error, and there is no reason not to change it in other scenarios.

As discussed earlier, we don't want the learning rate to be too large or too small. Too big and learning will be erratic. Too small and learning will be slow.

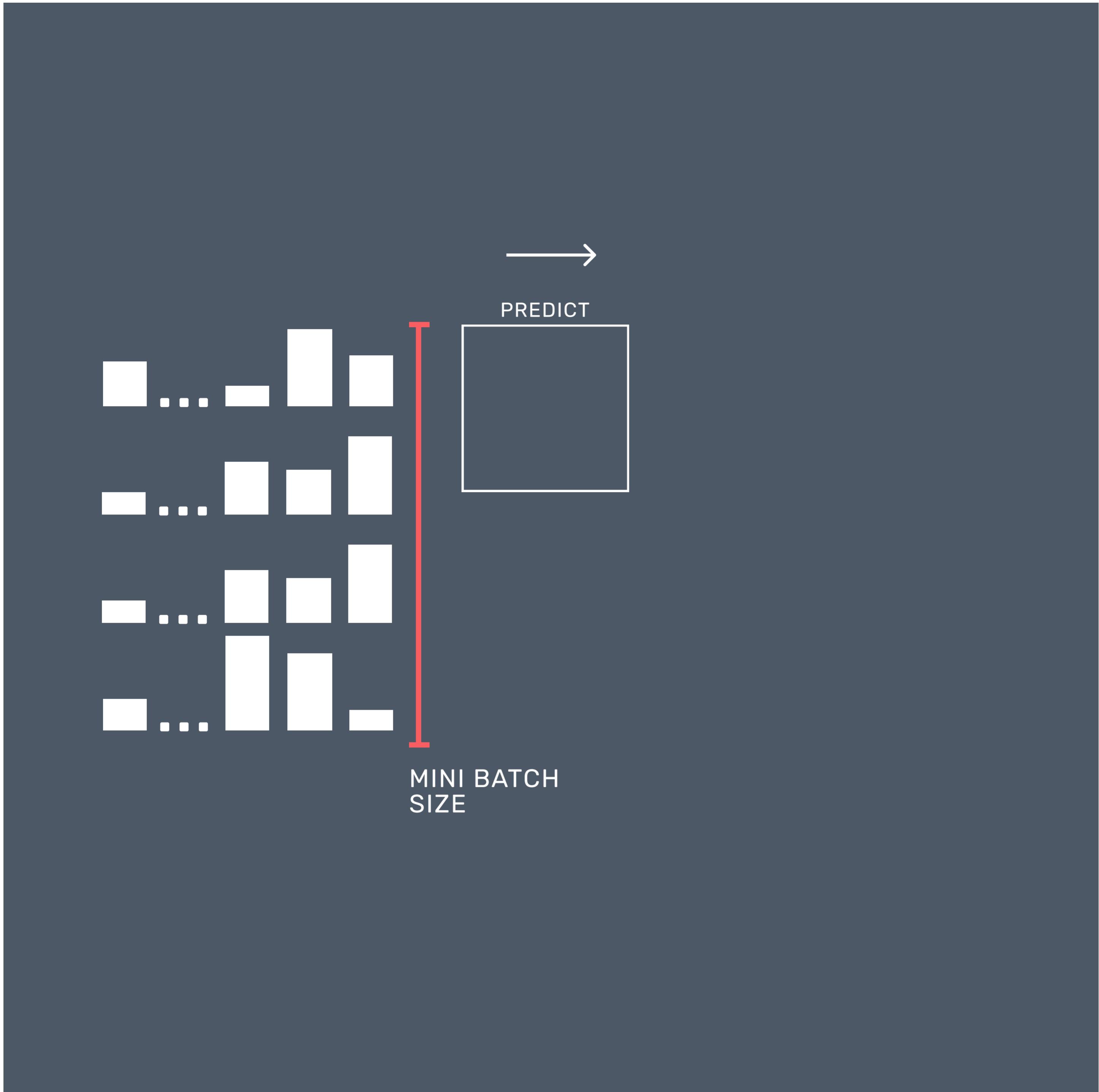


## NUMBER OF EPOCHS

We chose 100 epochs for Chapter 1 and 800 epochs for Chapters 2, 3, and 4 by trial and error.

An alternative approach is to set a very large epoch and configure training to stop automatically once certain performance criteria are met. This is called *early stopping*.

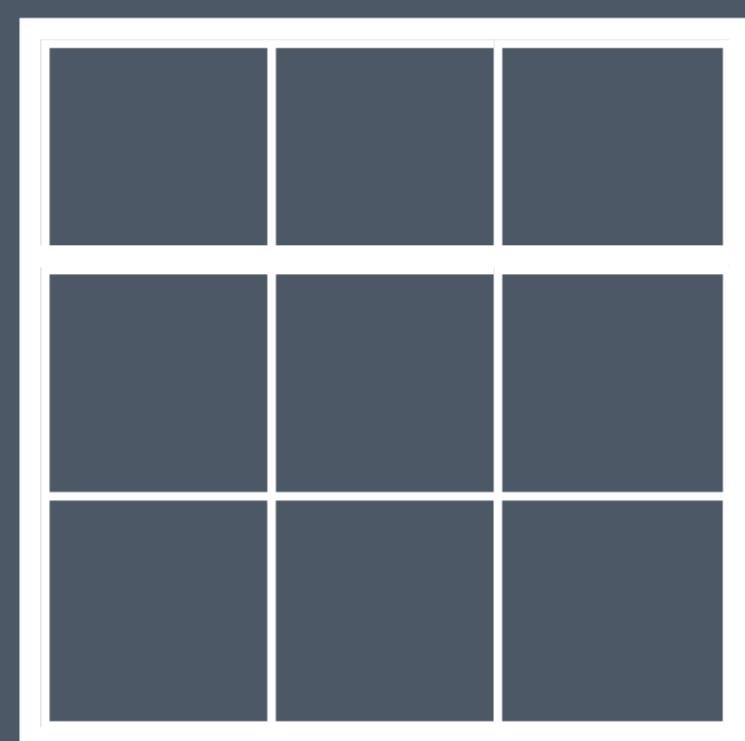
Finding the right epoch for a task is important because too many epochs can lead to the model ‘learning too much’, including unwanted noise in the data. On the other hand, too few epochs can lead to the model not capturing enough information from the data.



## BATCH SIZE

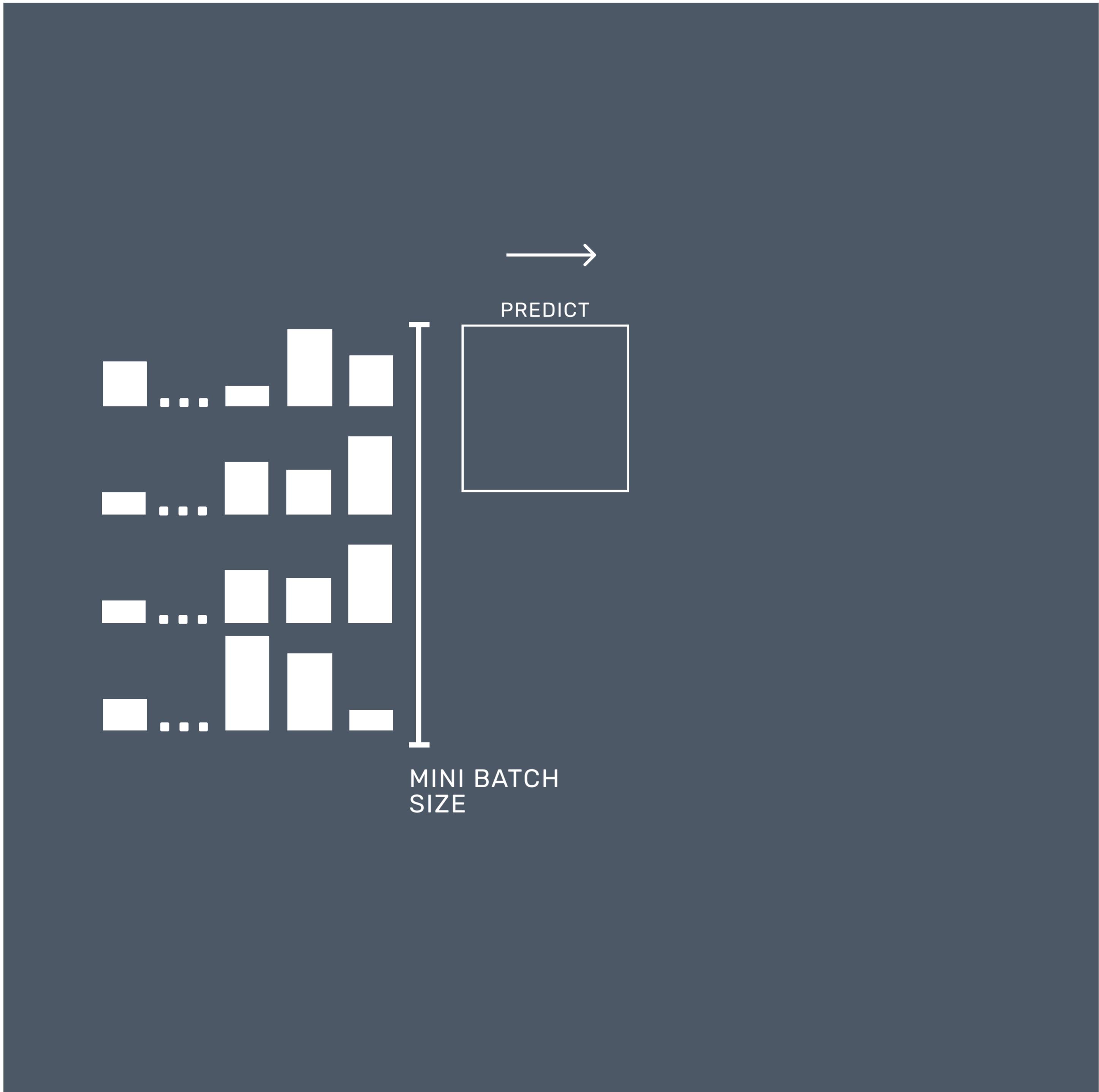
In our tasks, we had a maximum of twenty-four training data points, which we trained as a full batch. However, in a typical training dataset, the number of data points is so large that we need to split them into mini batches. This hyperparameter is called the *batch size*. In the diagram above, the batch size is four.

This brings us to the other way to improve our prediction results—by working on the data.



## DATA TECHNIQUES

We'll look at a few data techniques that we can use to improve our predictions.

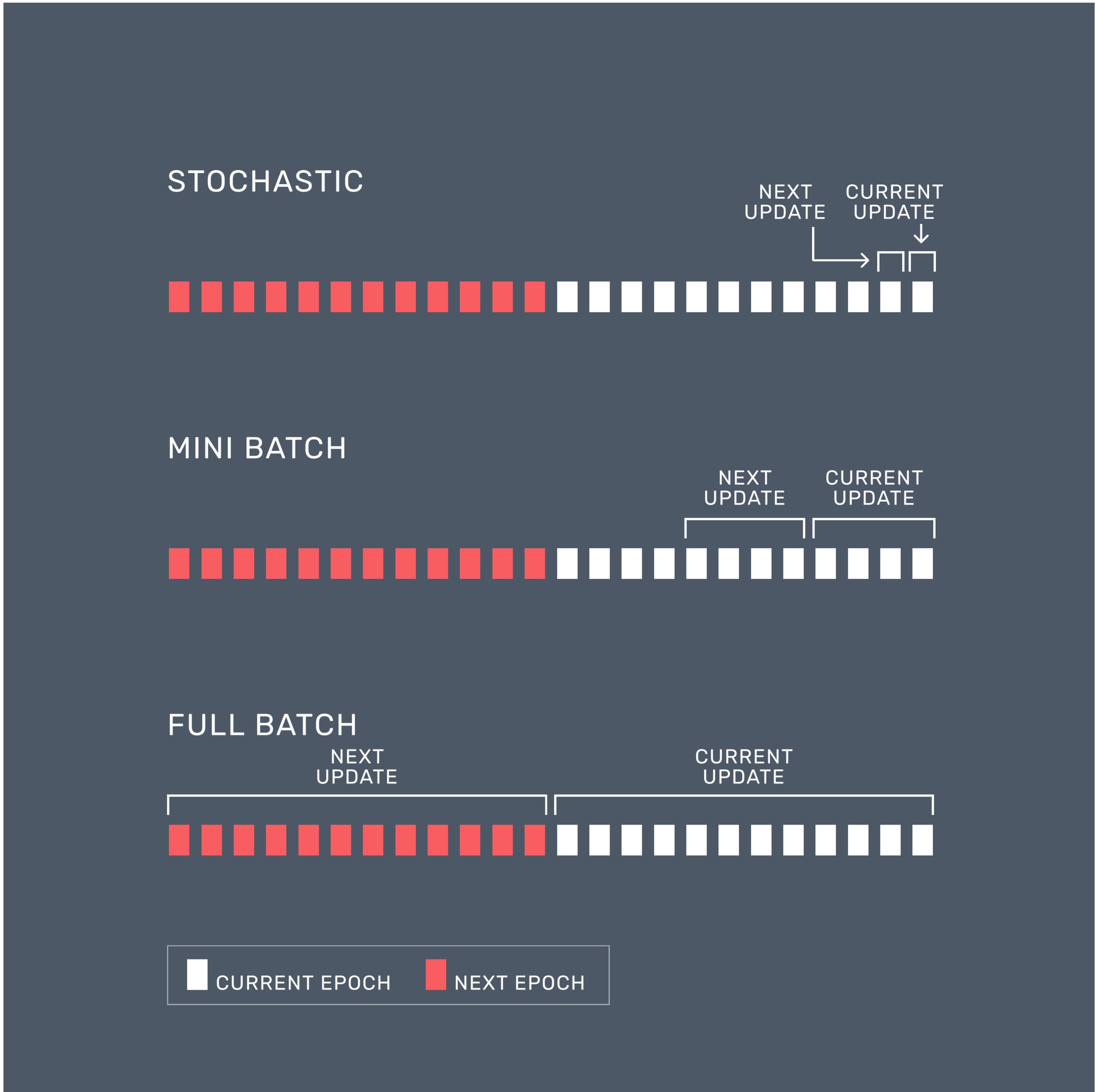


## MINI BATCH

Let's expand on the concept of mini batches.

We perform mini batching for a number of reasons. One of them is hardware limitation. For a very large dataset, we cannot fit the data points all at once into the computer's memory. The solution is to split them into mini batches.

Another reason is performance. Let's see why this is so.

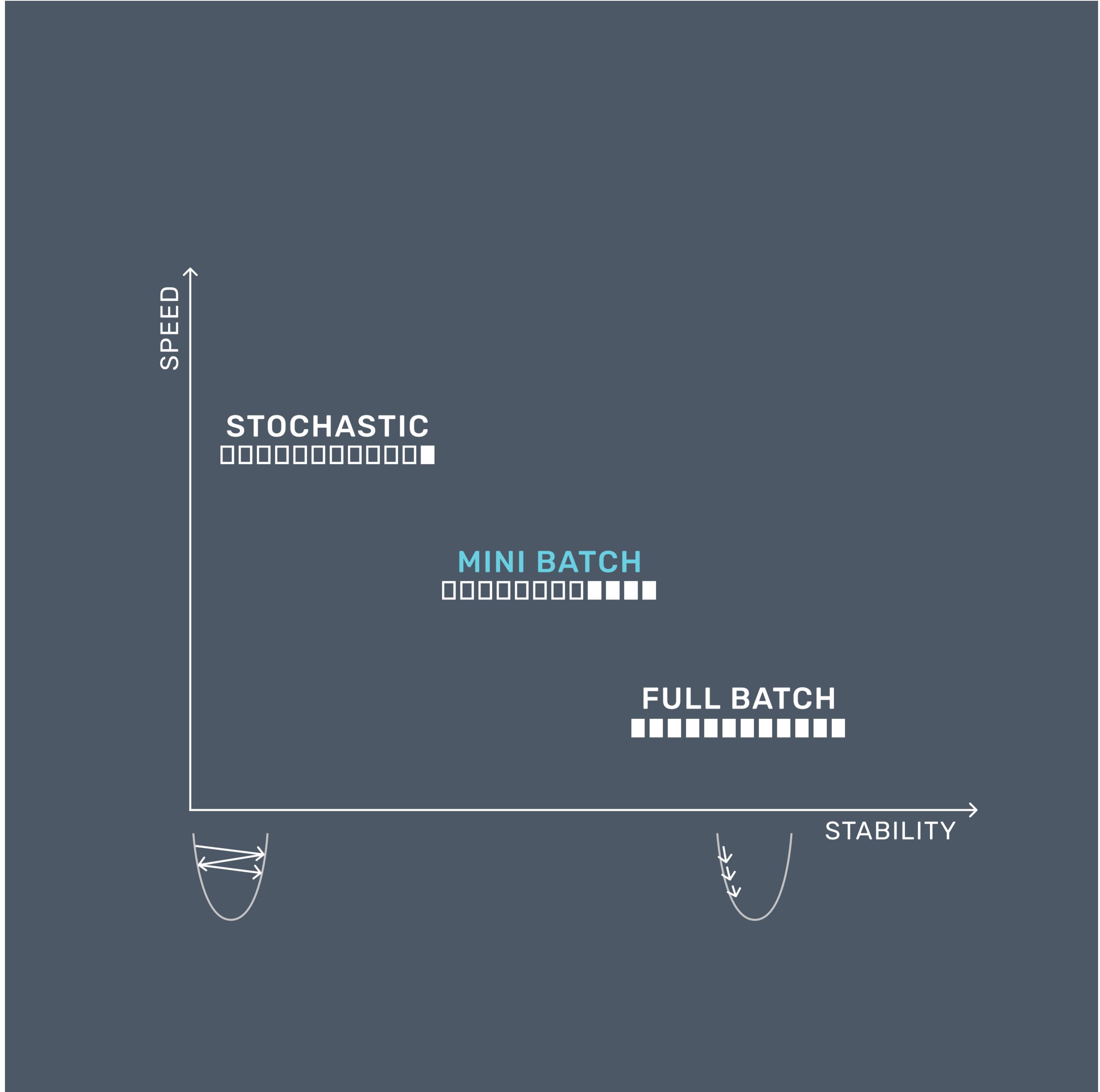


## MINI BATCH

Recall that in the four tasks, the model performed parameter adjustments after an entire epoch is complete. This is called *full batch gradient descent*.

At the other extreme, the model may also perform updates at every data point. This is called *stochastic gradient descent*.

*Mini batch gradient descent* is somewhere in between. For example, a batch size of four means that the model performs the updates after every four training data points.



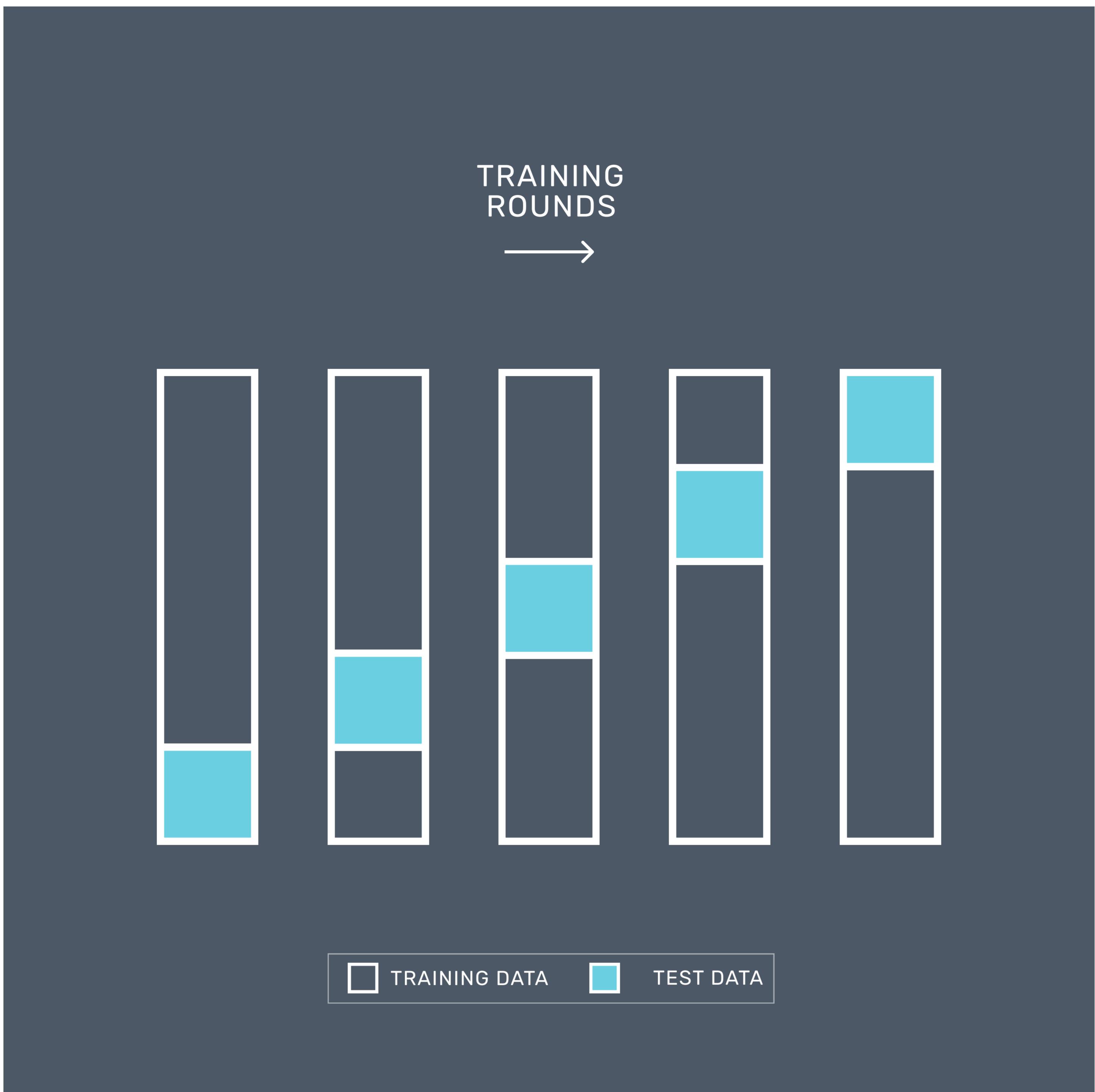
## BALANCE

How do these options affect performance?

Imagine having a huge dataset where each epoch takes a tremendous amount of time to complete. With full batch gradient descent, the time between parameter updates will be too long.

Conversely, stochastic gradient descent brings the advantage of fast feedback loops. However, since each update is based on only one data point, learning becomes erratic and unstable.

The mini batch gradient descent captures the best of both worlds, offering a balance between speed and stability.

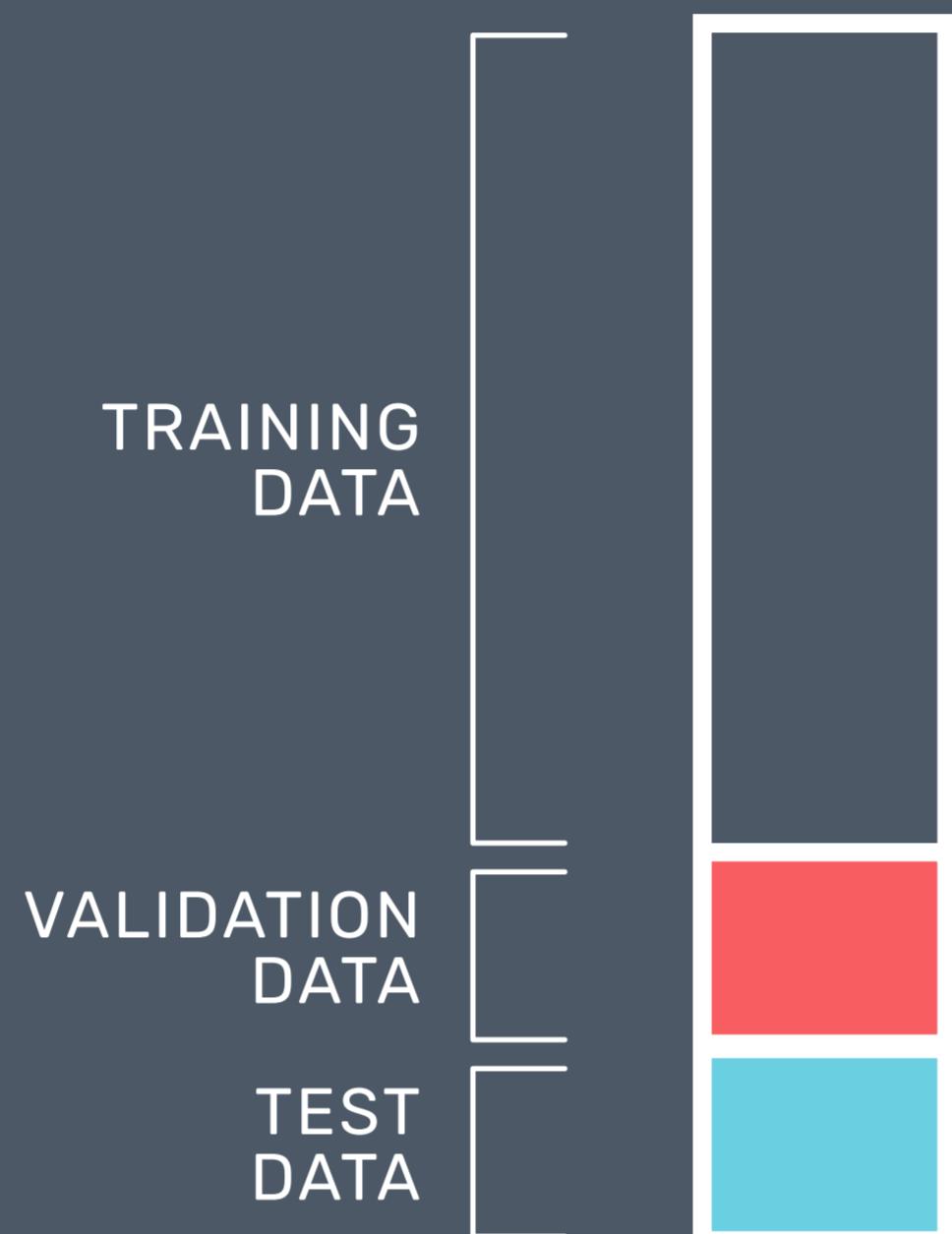


## K-FOLD CROSS VALIDATION

Let's move on to the next technique. Sometimes *overfitting* can occur in training, where the model is too attuned to the training data and performs poorly on new data points. This is especially true when the training dataset is small.

One way to address this is to use *K-fold cross validation*. Here, we cycle through the training-test data split over many training rounds such that by the end, every data point will have been used for training.

The performance metrics are then averaged over the number of rounds.



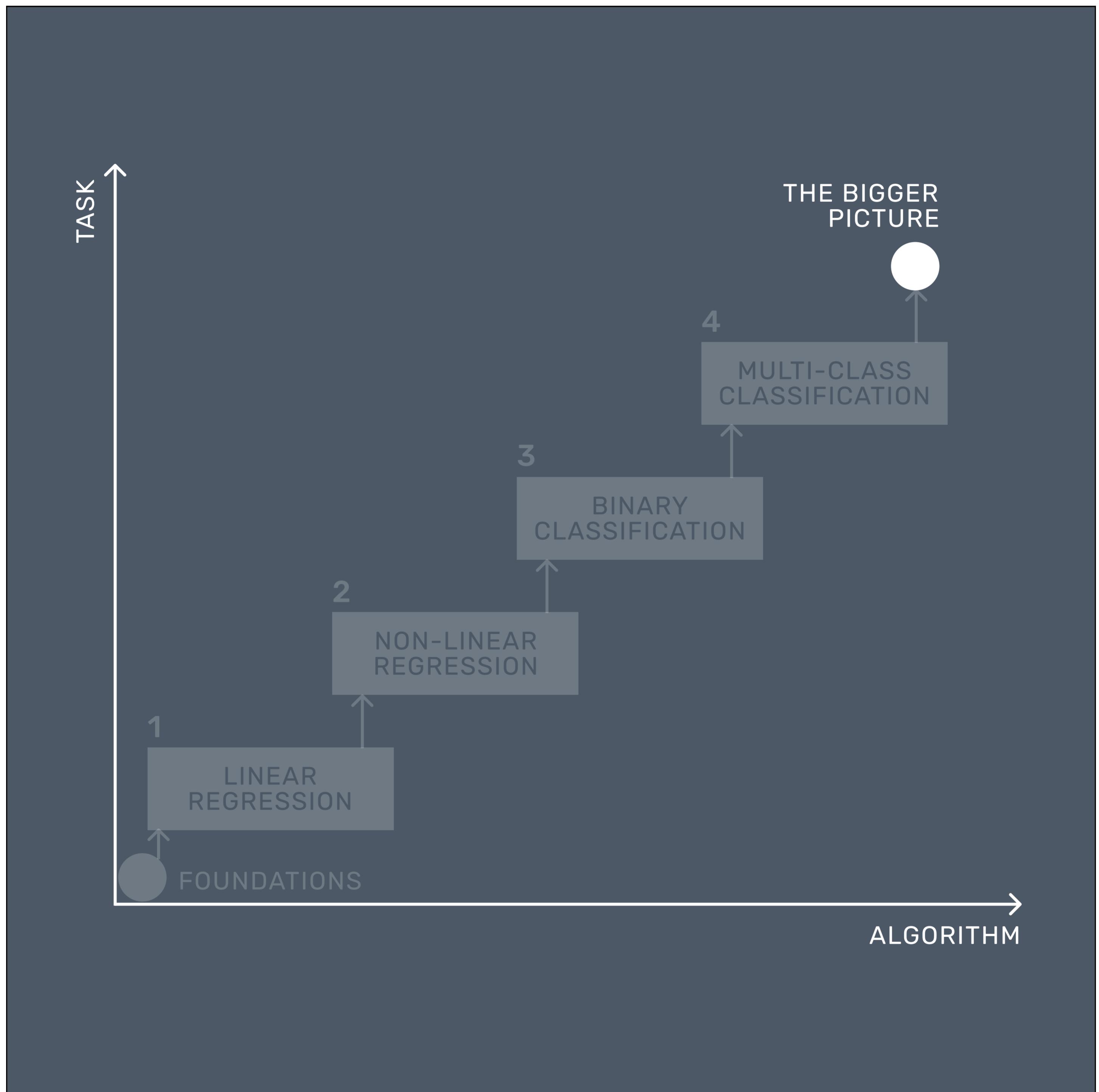
# VALIDATION SET

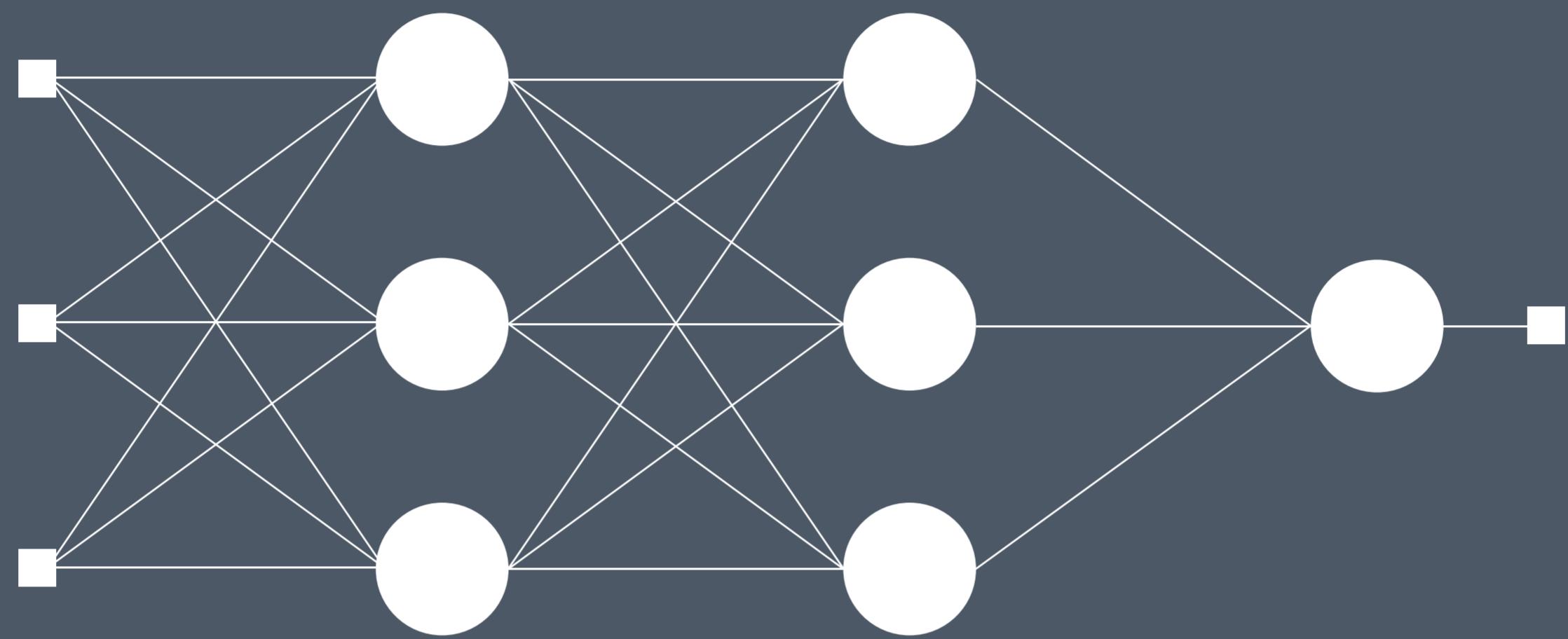
In all our tasks, for simplicity, we have split the dataset into two sets—training and test. However, in practice, it is common to split them into three sets—the other one being the *validation* set.

In this case, the role of the test set as we've been using will be replaced by the validation set.

This means we can set aside the test set until the model is fully ready. This offers a more accurate way to measure performance since these are fresh data points that the model has never seen.

# THE BIGGER PICTURE

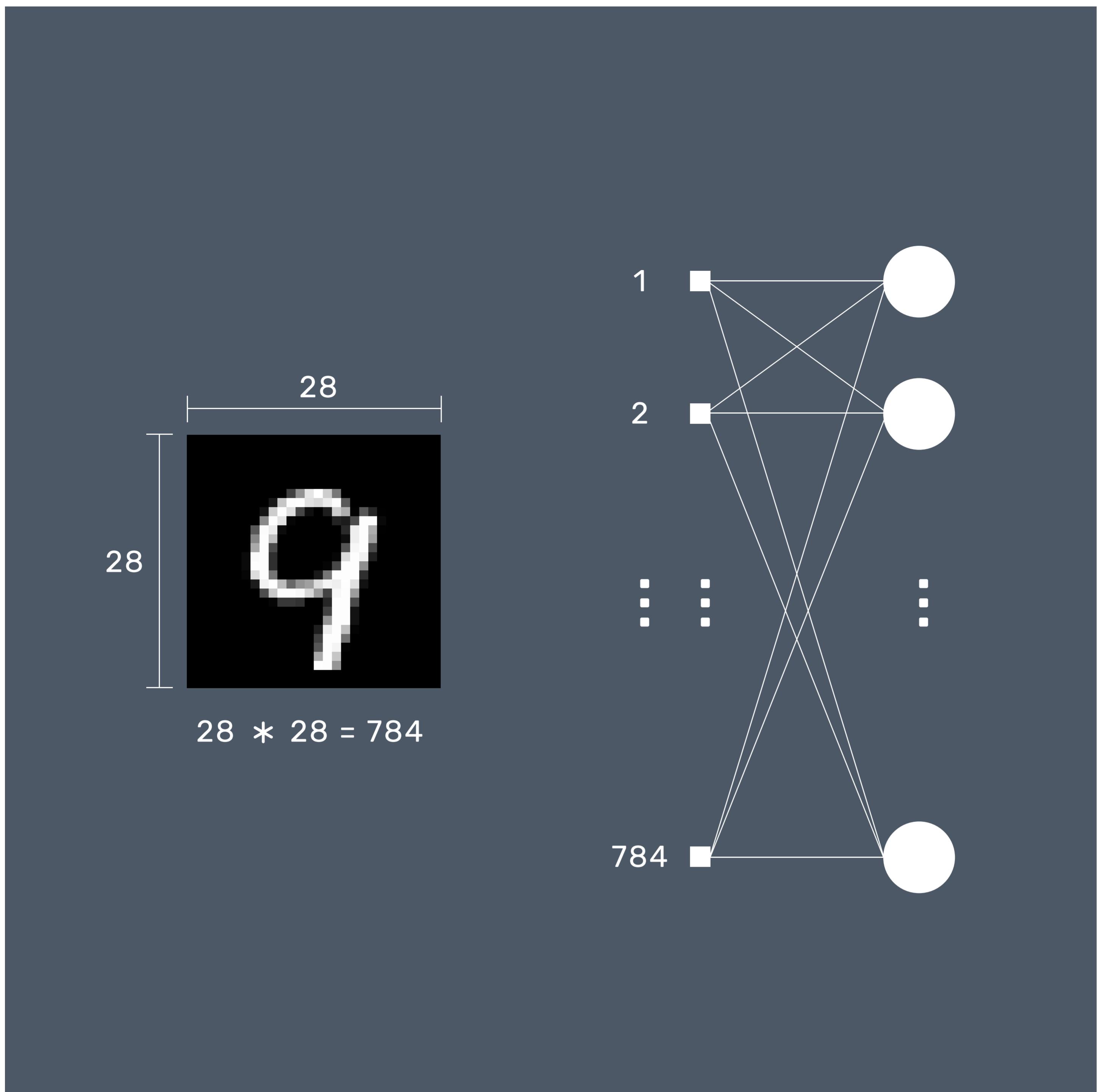




## FEEDFORWARD NEURAL NETWORKS

In this final chapter, we'll take a brief tour of deep learning architectures beyond what's been covered so far.

Let's start with the *feedforward neural network*. This is the quintessential version of neural networks, and it is the version we used in the four tasks. It's called feedforward because information flows through the layers only in the forward direction.

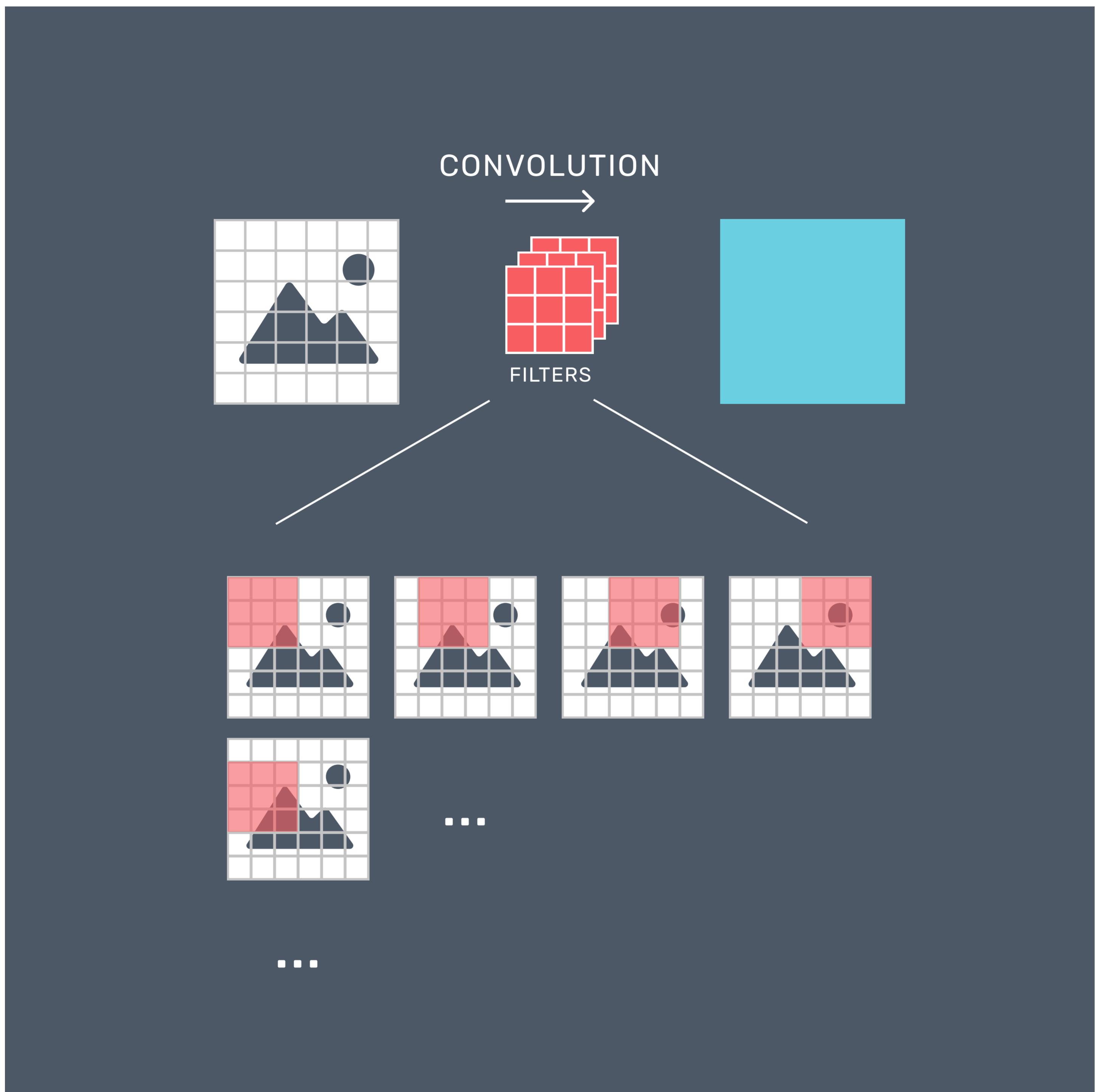


## FEEDFORWARD NEURAL NETWORKS

We had a small network with only two inputs, but we can expand it to be as big as we need it.

Take a look at this *MNIST* dataset. It is an image dataset where each data point contains an image of a handwritten number. This is a multi-class classification task to predict the number based on the image. The features are the individual pixels ( $28 \times 28$ ), while the label is a number between 0 and 9.

There are 784 features, which means that 784 inputs are going into the first hidden layer. Additionally, such a task will require even more hidden layers with a substantial number of neurons in each.

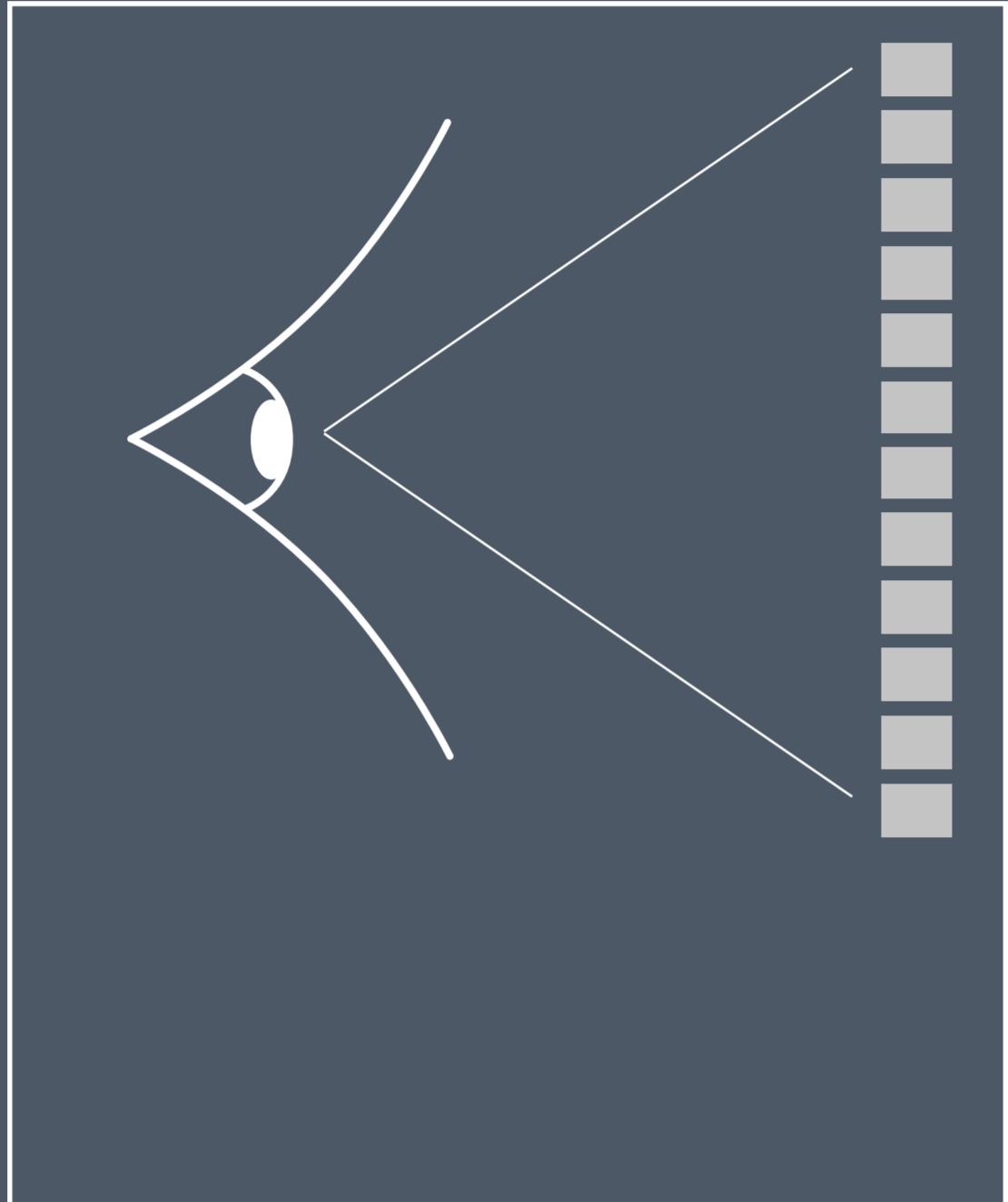


## CONVOLUTIONAL NEURAL NETWORKS

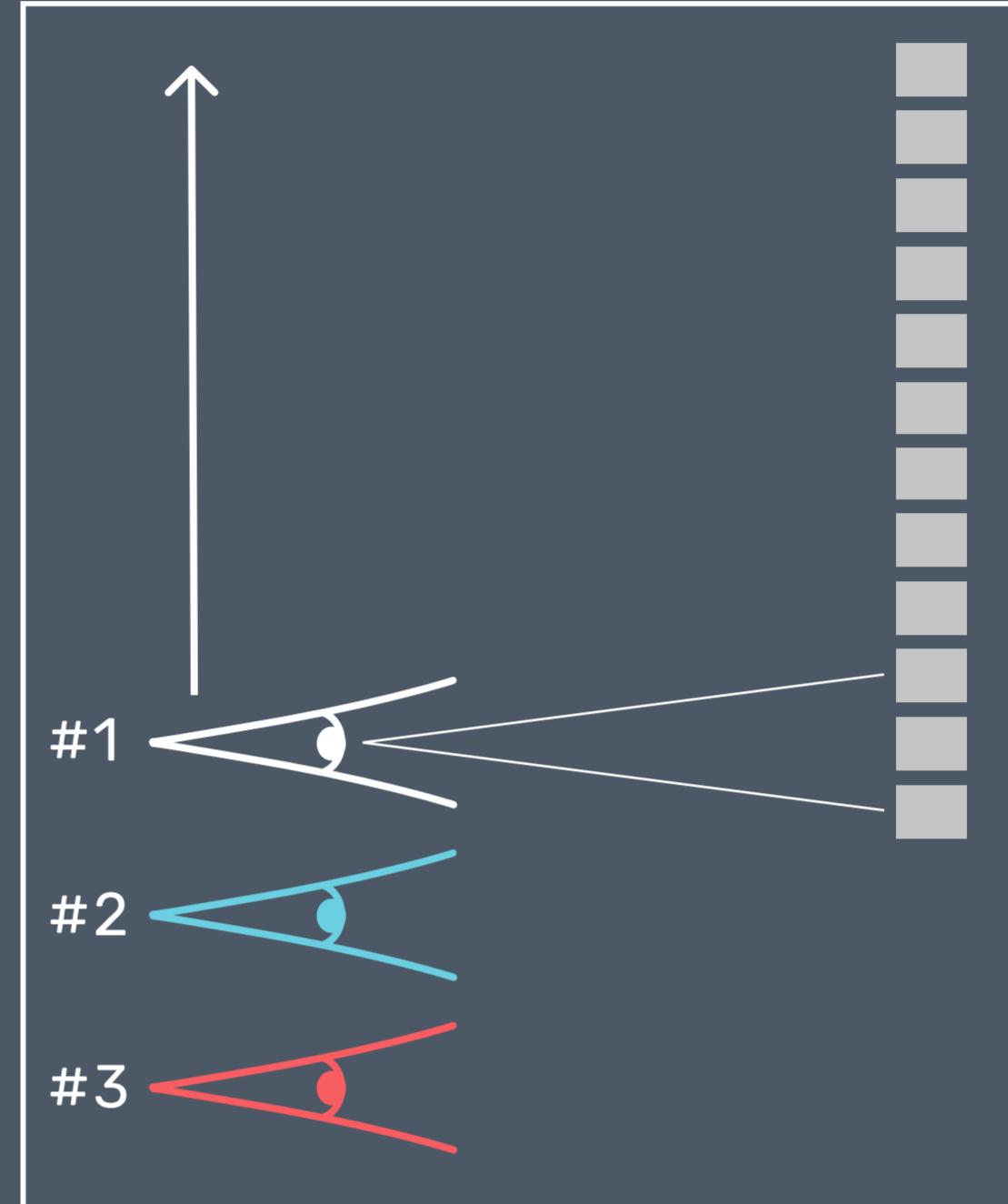
Now let's look at another type - the *convolutional neural network (CNN)*, typically used for image data.

The idea with this architecture is, instead of feeding all inputs to all neurons, we group neurons into smaller sets called *filters*. These filters don't take the inputs all at once. Instead, they scan through the small sections of the inputs sequentially.

FEEDFORWARD NN



CONVOLUTIONAL NN



## CONVOLUTIONAL NEURAL NETWORKS

The scanning effect makes it perfect for image data. It takes advantage of the fact that in images, a pixel has a stronger relationship with its surrounding pixels than with pixels far away.

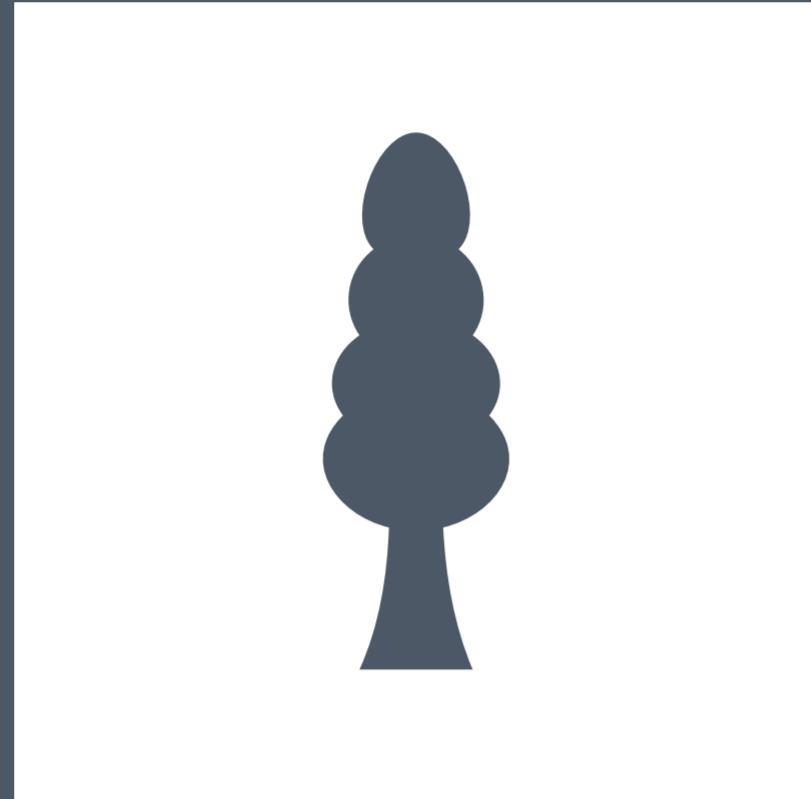
These filters learn uniquely at the individual level and synergistically at the collective level.



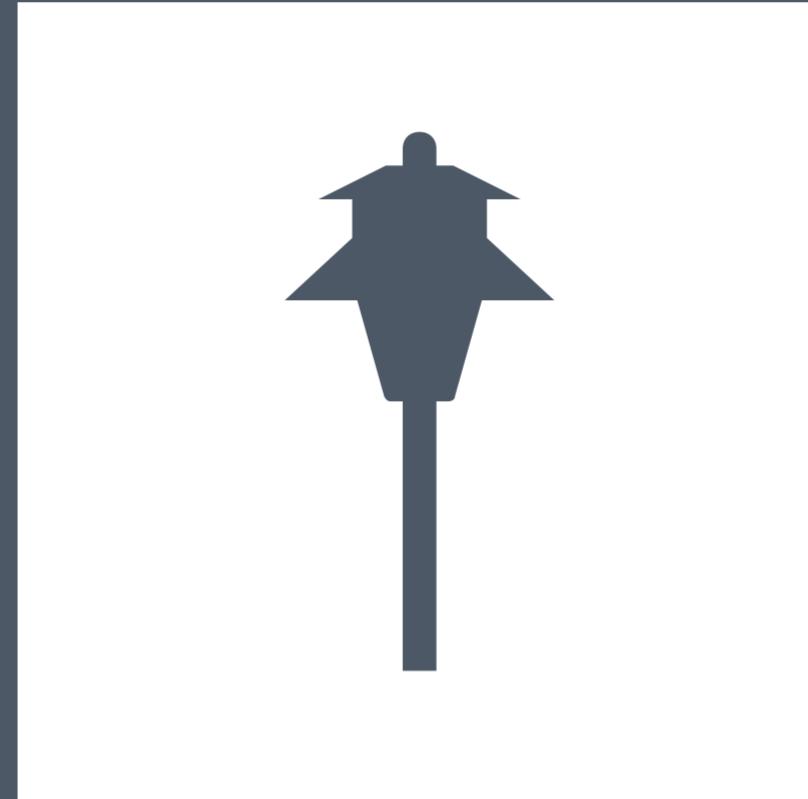
## CONVOLUTIONAL NEURAL NETWORKS

This process is repeated over several layers, and the result is a compressed version of the data. This information is finally fed into a standard layer to produce the prediction.

## IMAGE CLASSIFICATION



TREE

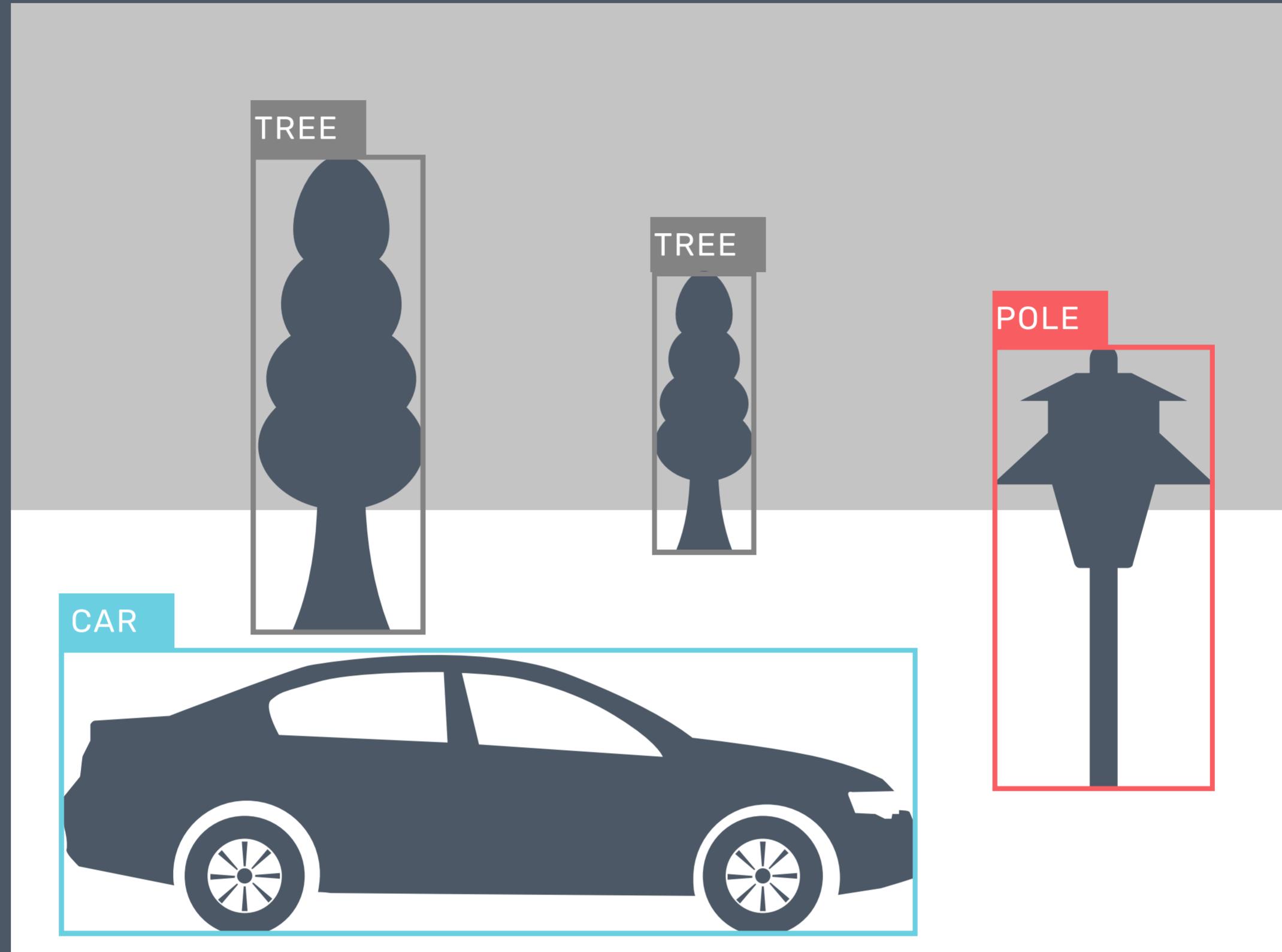


NOT TREE

## IMAGE CLASSIFICATION

Let's look at some use cases of the CNN. The first is *image classification*, where the task is to predict the class of an image. It's the same type of task as the MNIST example, but with a different architecture.

## OBJECT DETECTION



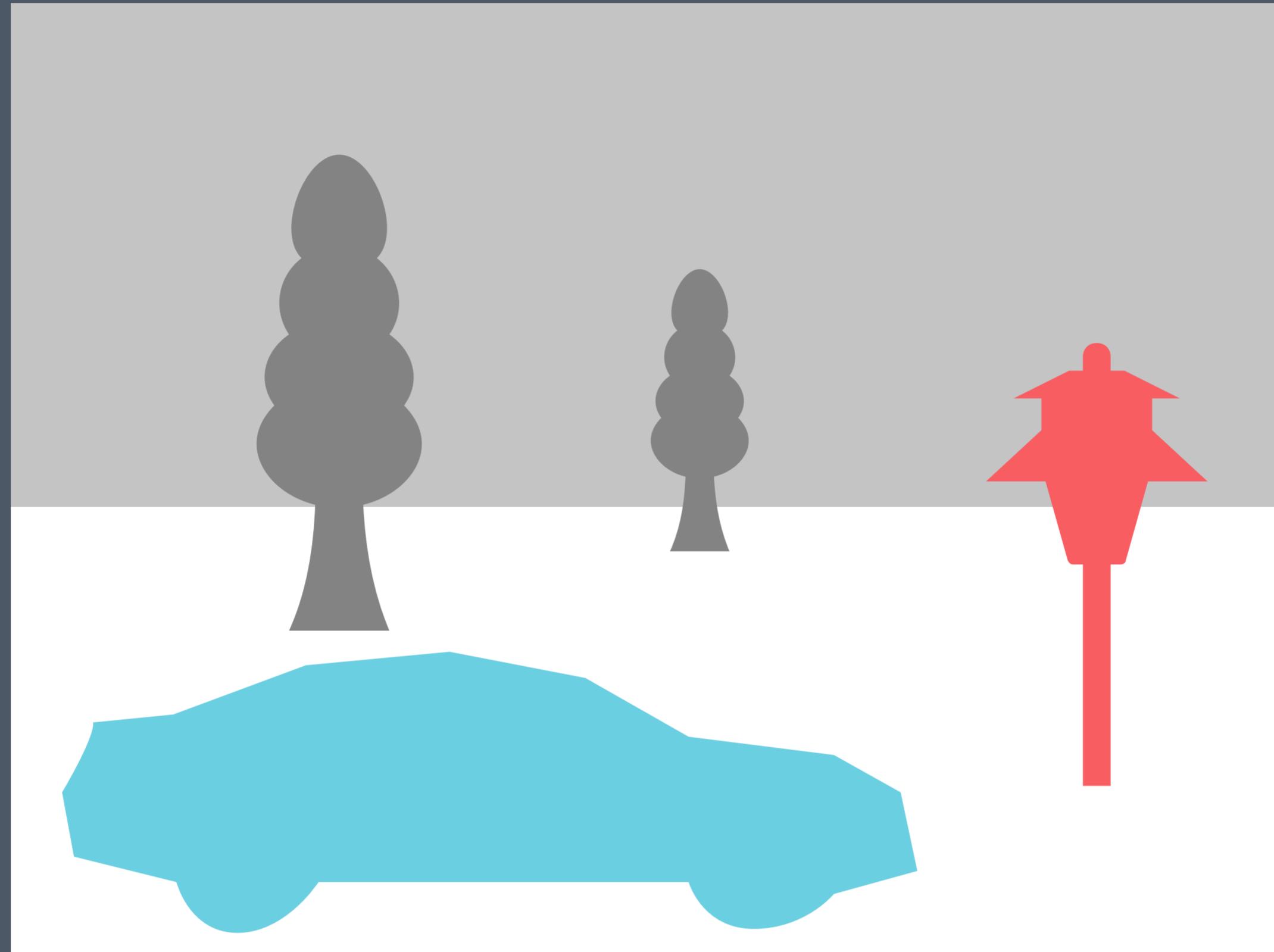
## OBJECT DETECTION

Another use case is *object detection*. It involves both classification and regression.

As for classification, this time each image could contain more than one class. Here is an example where we have four objects belonging to one of three classes.

As for regression, the task is to predict the position on the image where these objects are located.

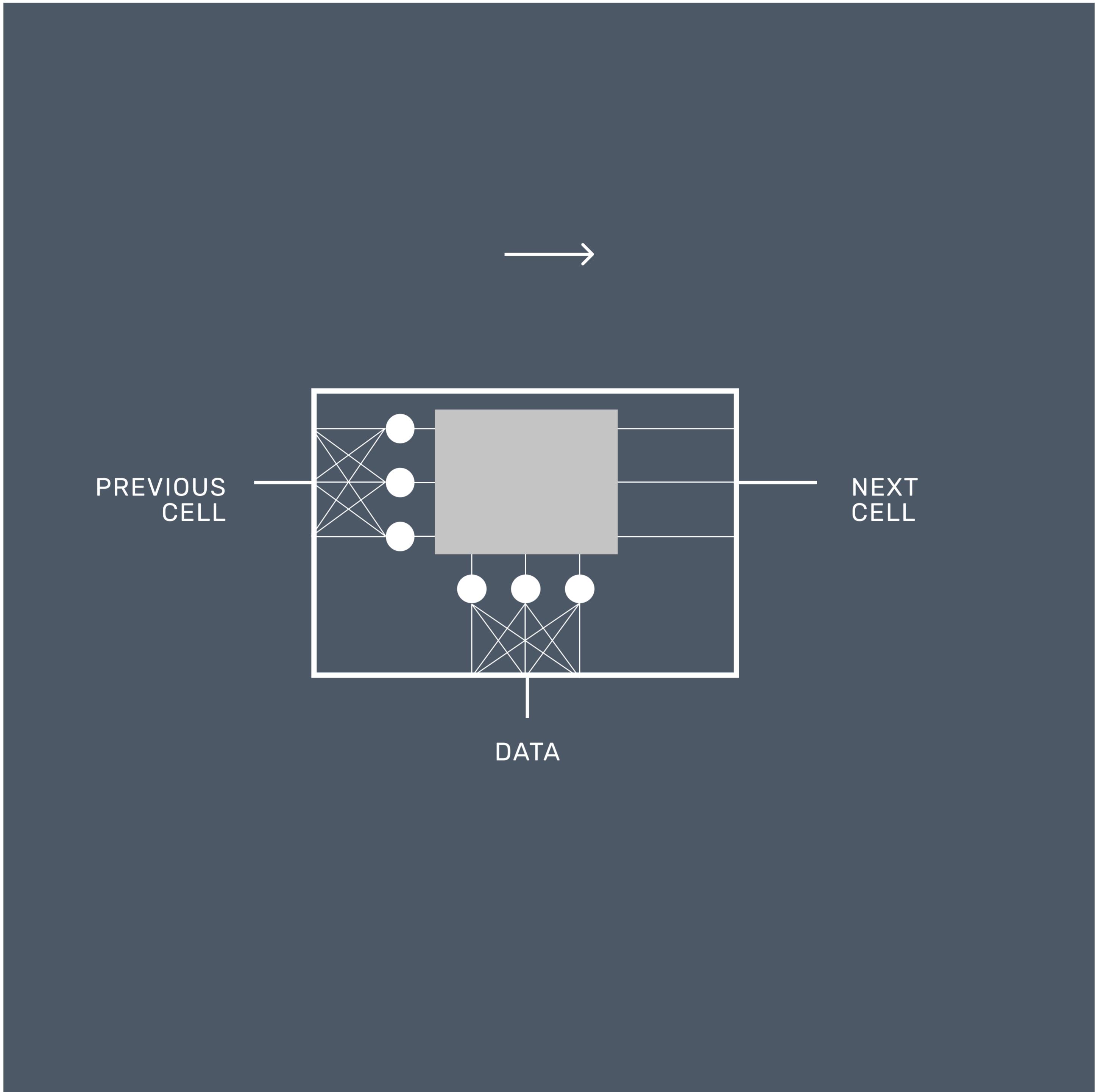
## SEMANTIC SEGMENTATION



## SEMANTIC SEGMENTATION

Another use case is semantic segmentation. This task is similar to image classification, except that classification is performed at the pixel level instead of at the image level.

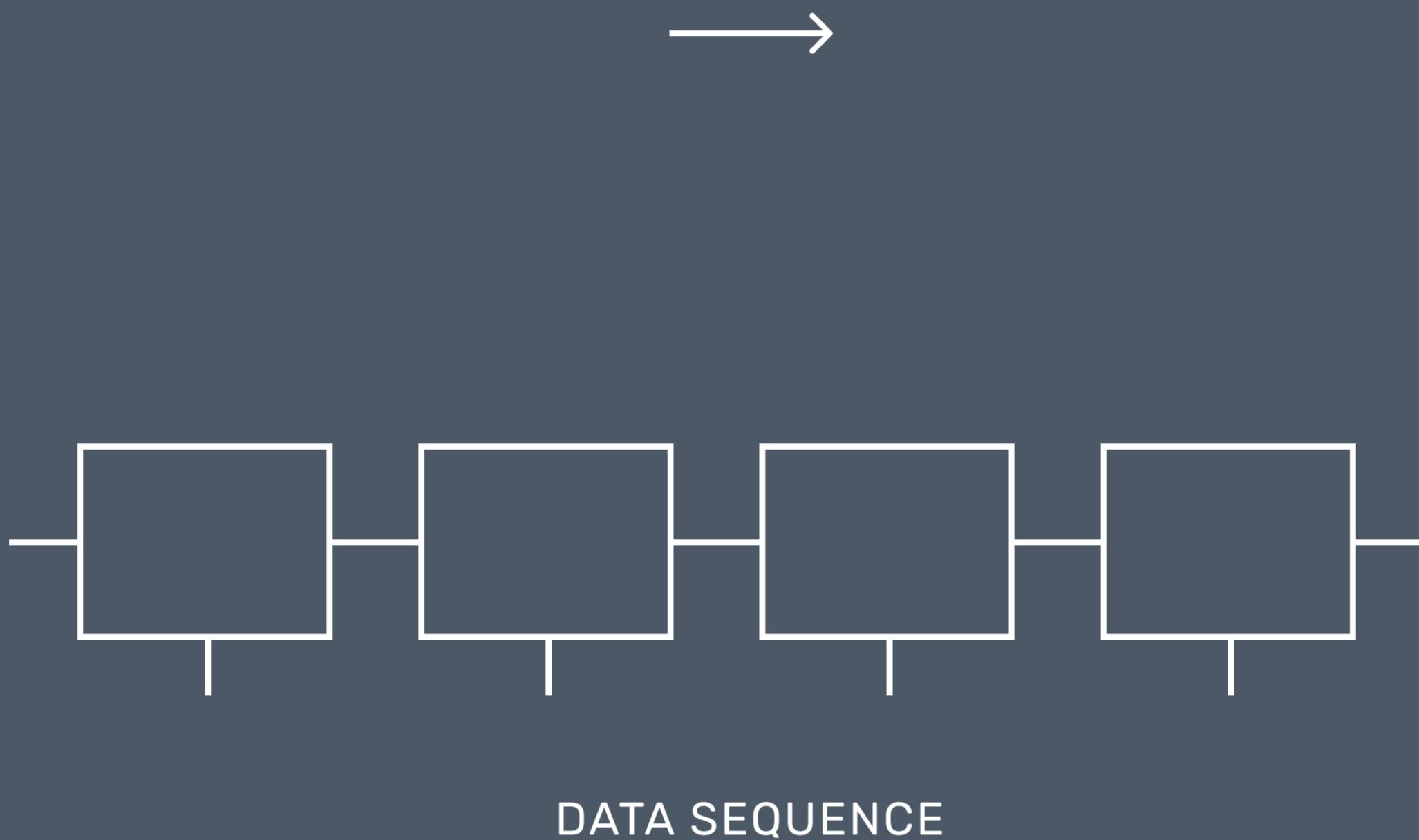
The result is a finer definition of object boundaries on the image.



## RECURRENT NEURAL NETWORKS

Let's take a look at another architecture - the *recurrent neural network (RNN)*. This architecture is designed to work with data types with a sequence element, such as text, music, and time-series data.

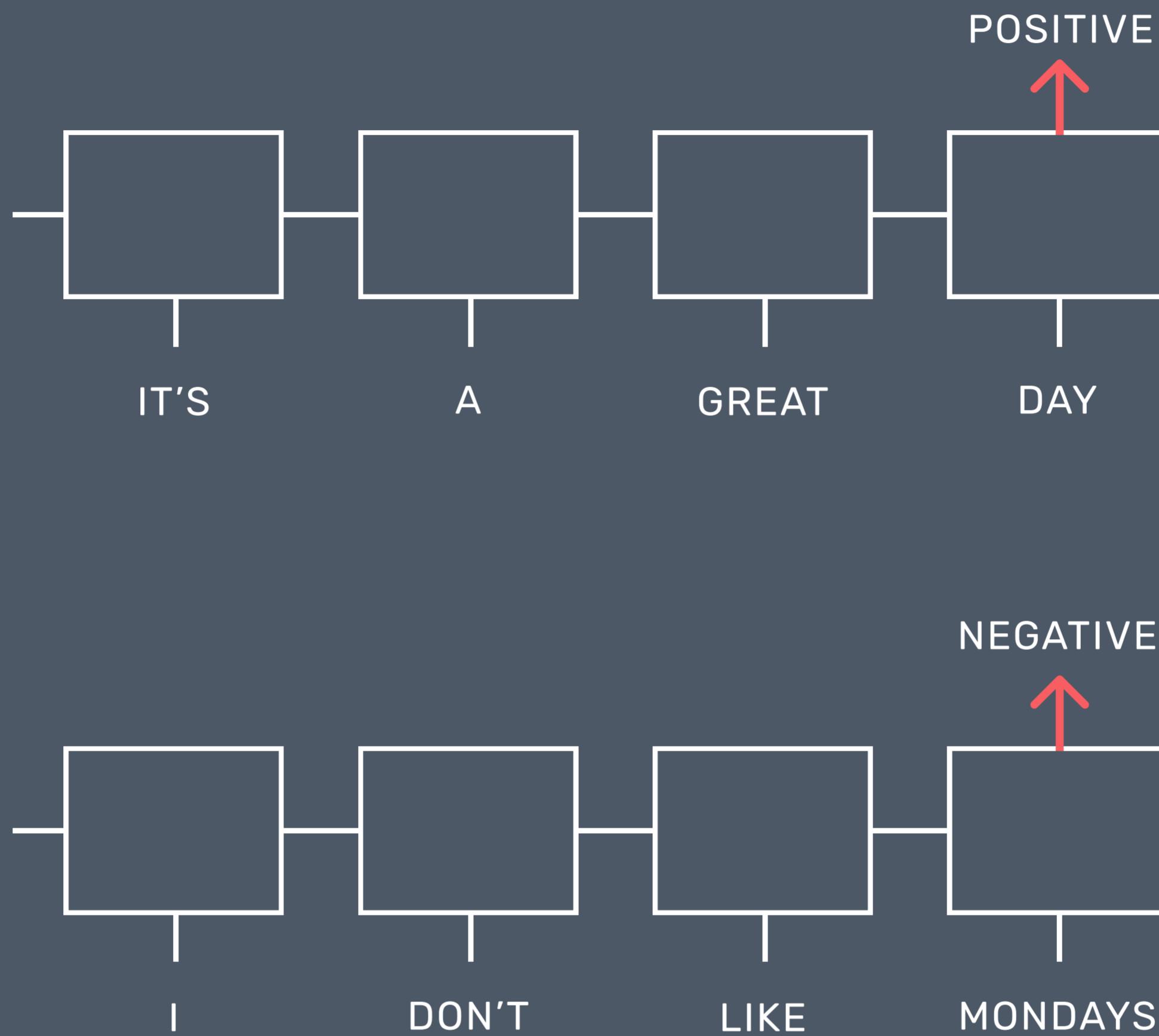
This diagram depicts one building block of the RNN. It has two inputs. The first input is the data from the current step of a sequence, while the other input is the output of the previous step's RNN block.



## RECURRENT NEURAL NETWORKS

By chaining these blocks together, we can pass information from each step of the sequence to the end of the network.

## TEXT CLASSIFICATION

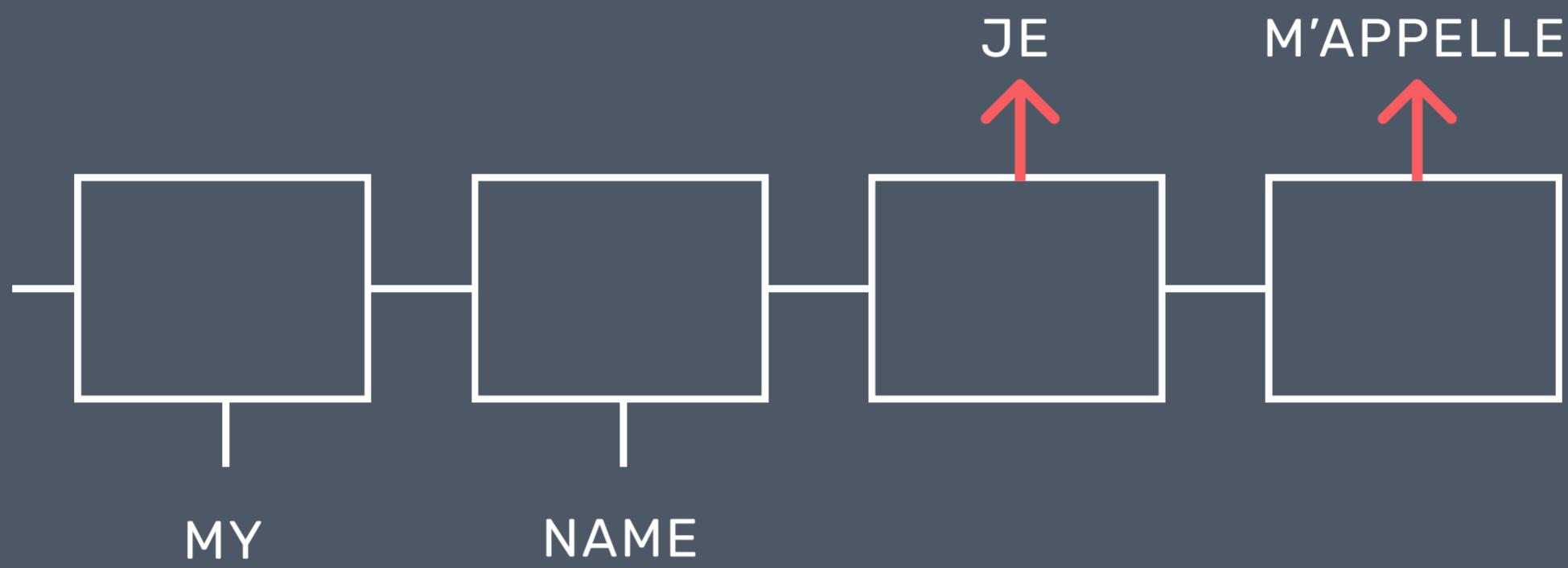


## TEXT CLASSIFICATION

Let's take a use case - *text classification*. Here we have an example of sentiment analysis, where the task is to predict the sentiment in a sentence.

Using an RNN architecture allows us to capture the essence of each word in a sentence, which together forms the meaning of the entire sentence.

## MACHINE TRANSLATION

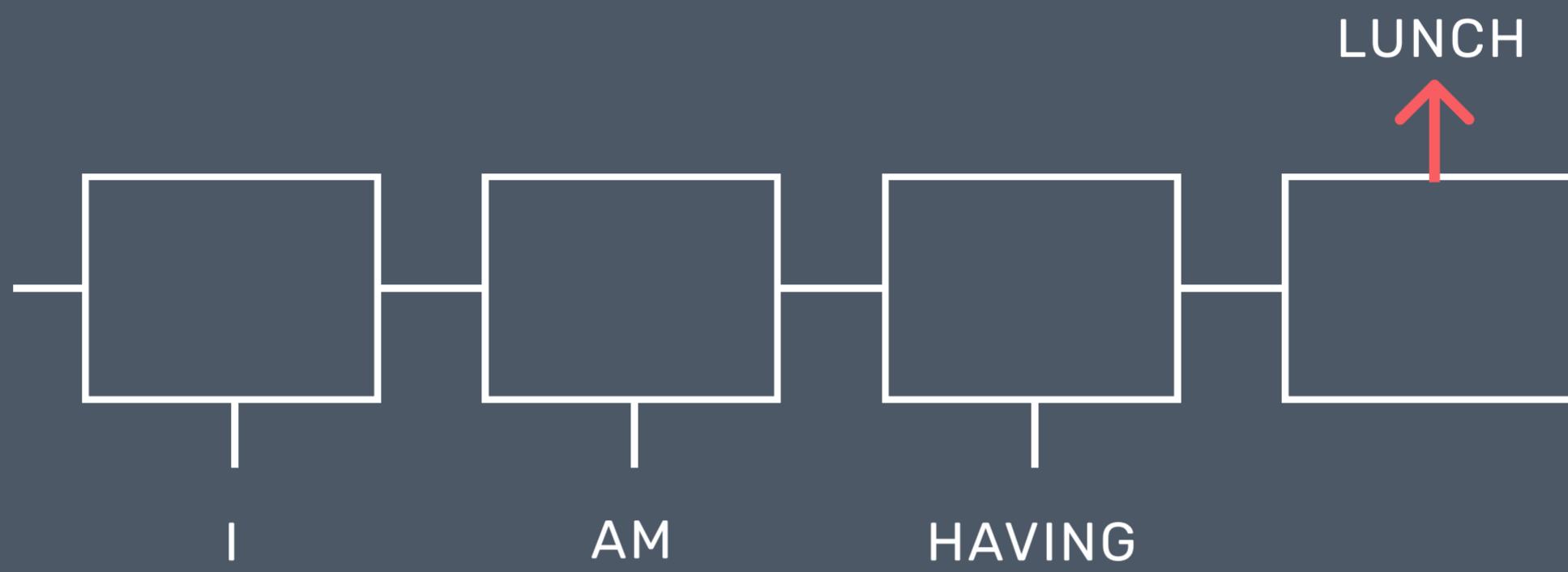


## MACHINE TRANSLATION

Another use case is *machine translation*. Here, the RNN generates many outputs instead of one.

Take a look at this example of a translation of a short phrase from English to French.

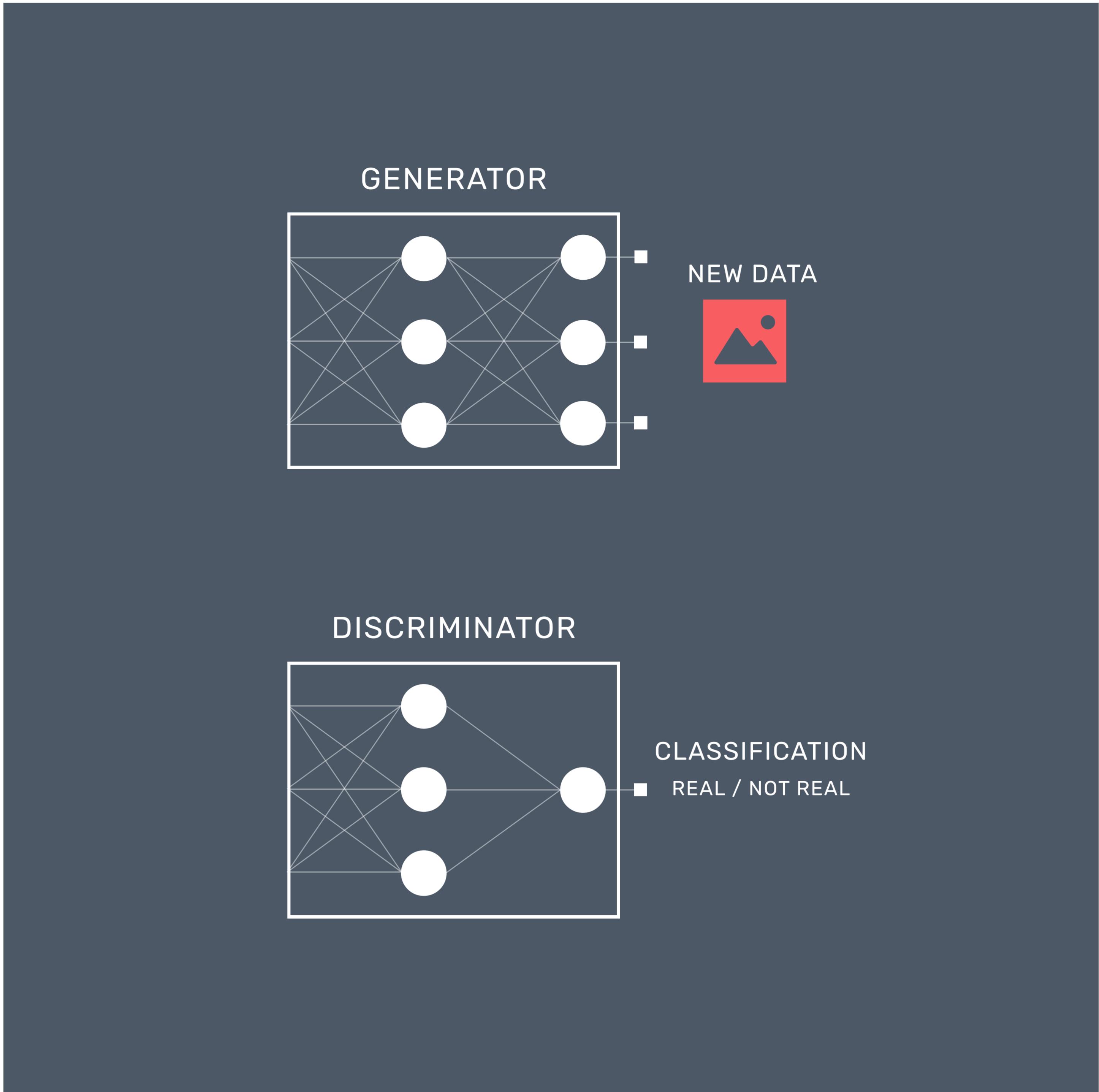
## TEXT GENERATION



## TEXT GENERATION

Another interesting use case is *text generation*. Here, we can train the neural network to predict the next word, given a sequence of words.

We can repeat this for as many words as we want. This means we can train it to generate a complete sentence, an essay, or even an entire book!

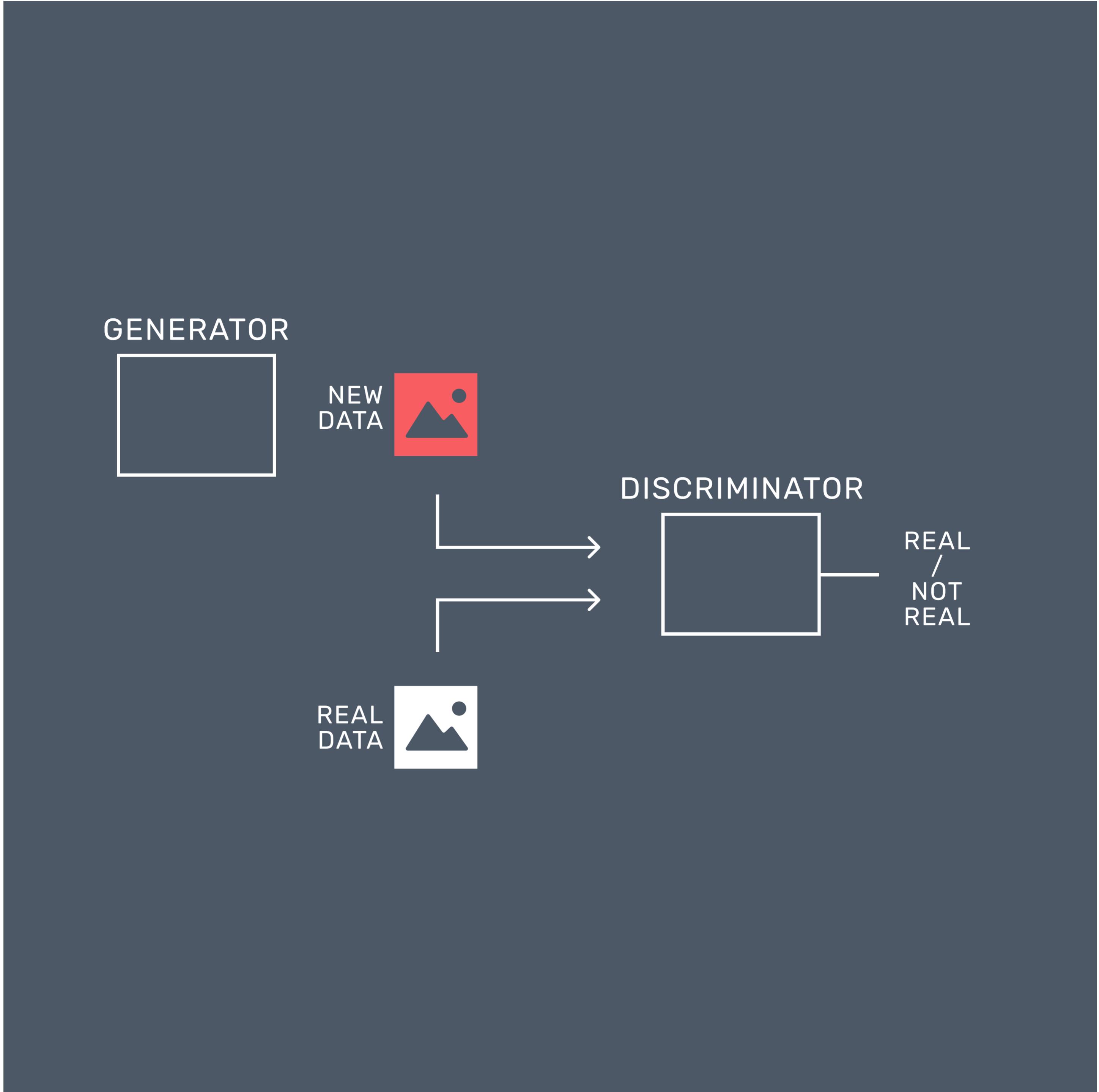


## GENERATIVE ADVERSARIAL NETWORKS

When it comes to generating new outputs, there is another architecture called the *Generative Adversarial Networks (GAN)*.

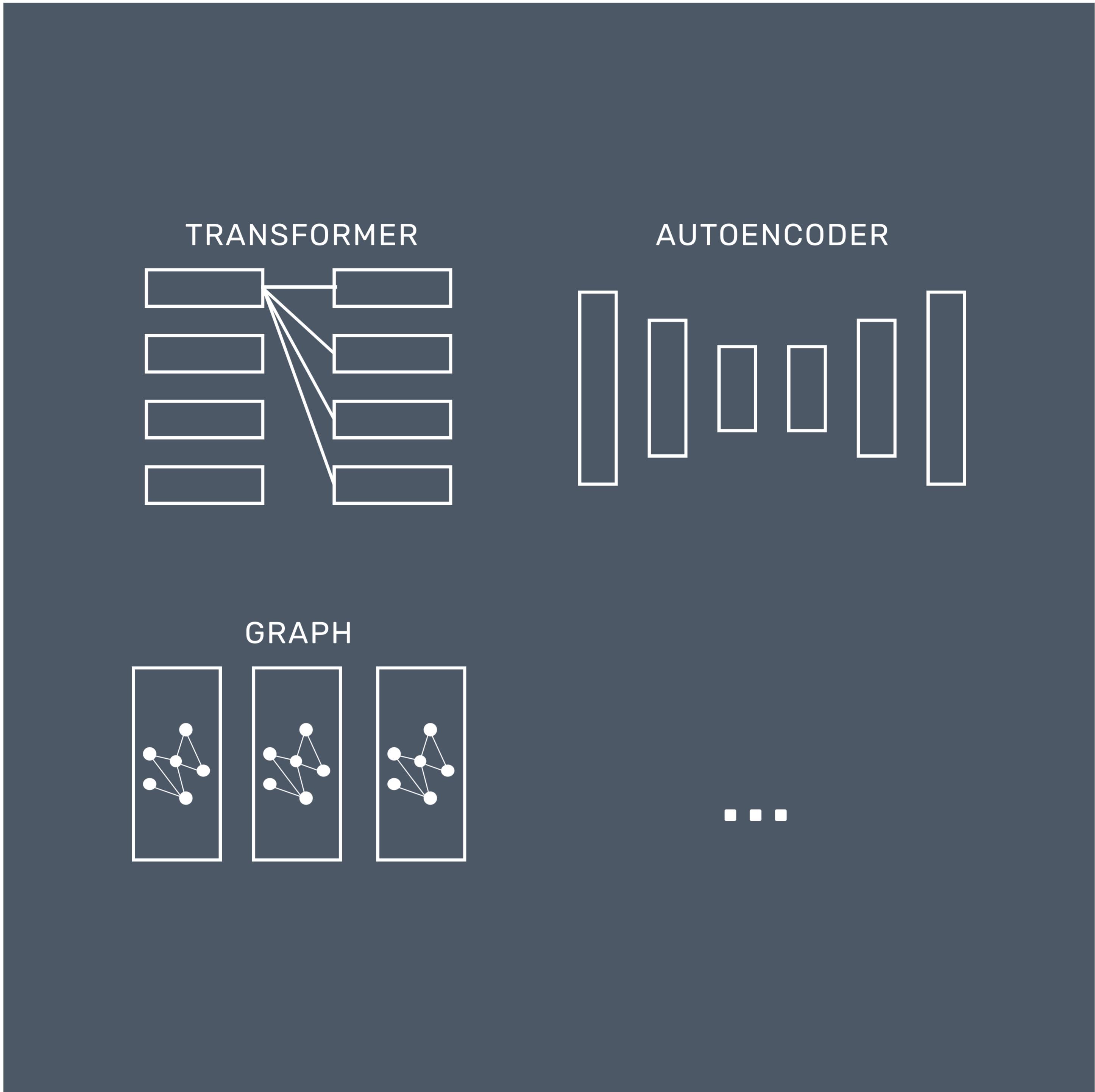
Given a training dataset, the GAN can create new data points that follow the same distribution but yet distinct from the original.

The GAN consists of two competing neural networks, the *generator* and the *discriminator*. The generator's role is to generate new examples that look as real as possible, while the discriminator's role is to determine if they are real.



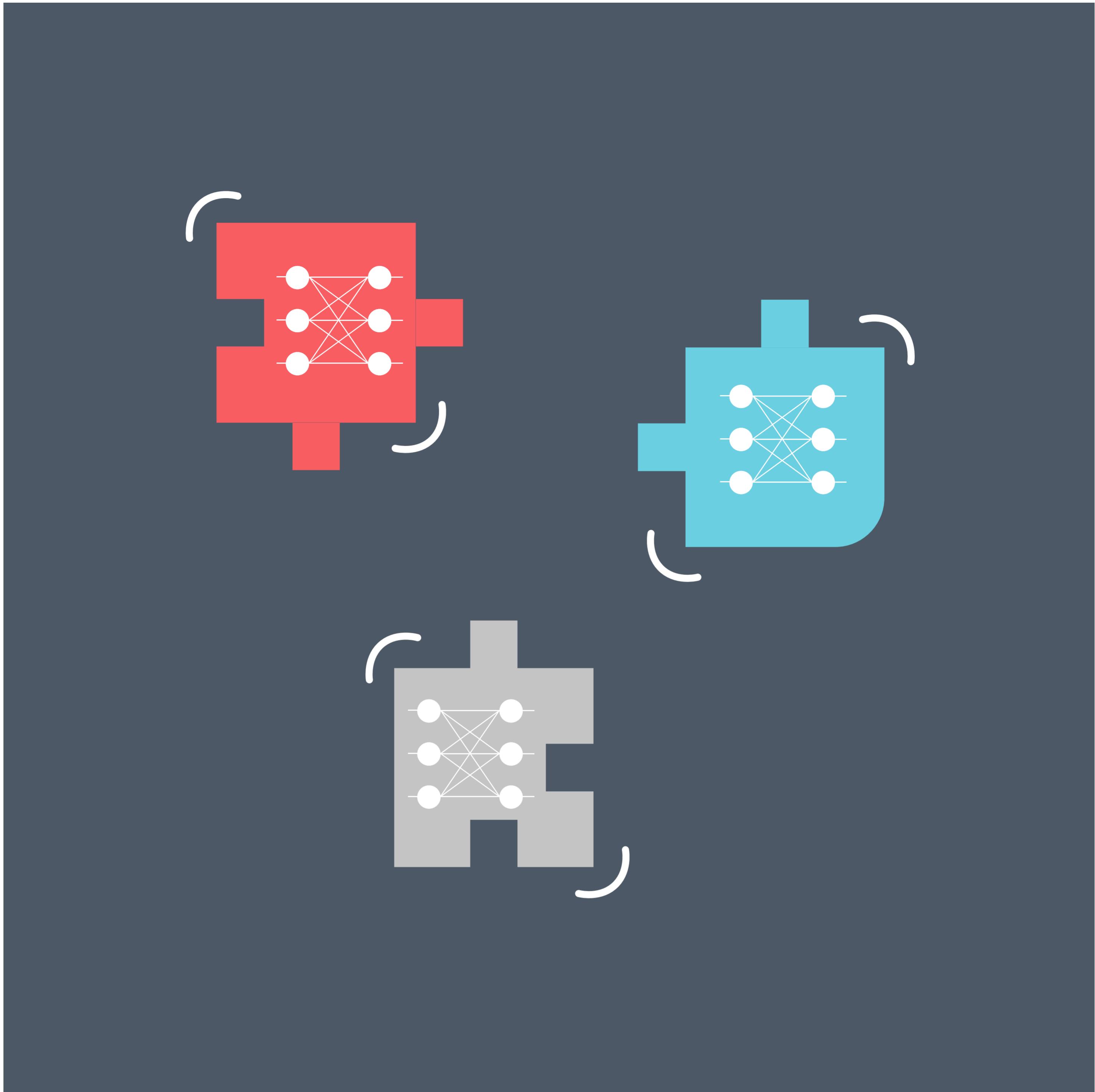
## GENERATIVE ADVERSARIAL NETWORKS

The architecture is designed so that both networks will keep improving after each training round. As more examples are given, the discriminator becomes increasingly good at detecting the real ones. At the same time, the generator's output will become more and more similar to the real ones.



## OTHER ARCHITECTURES

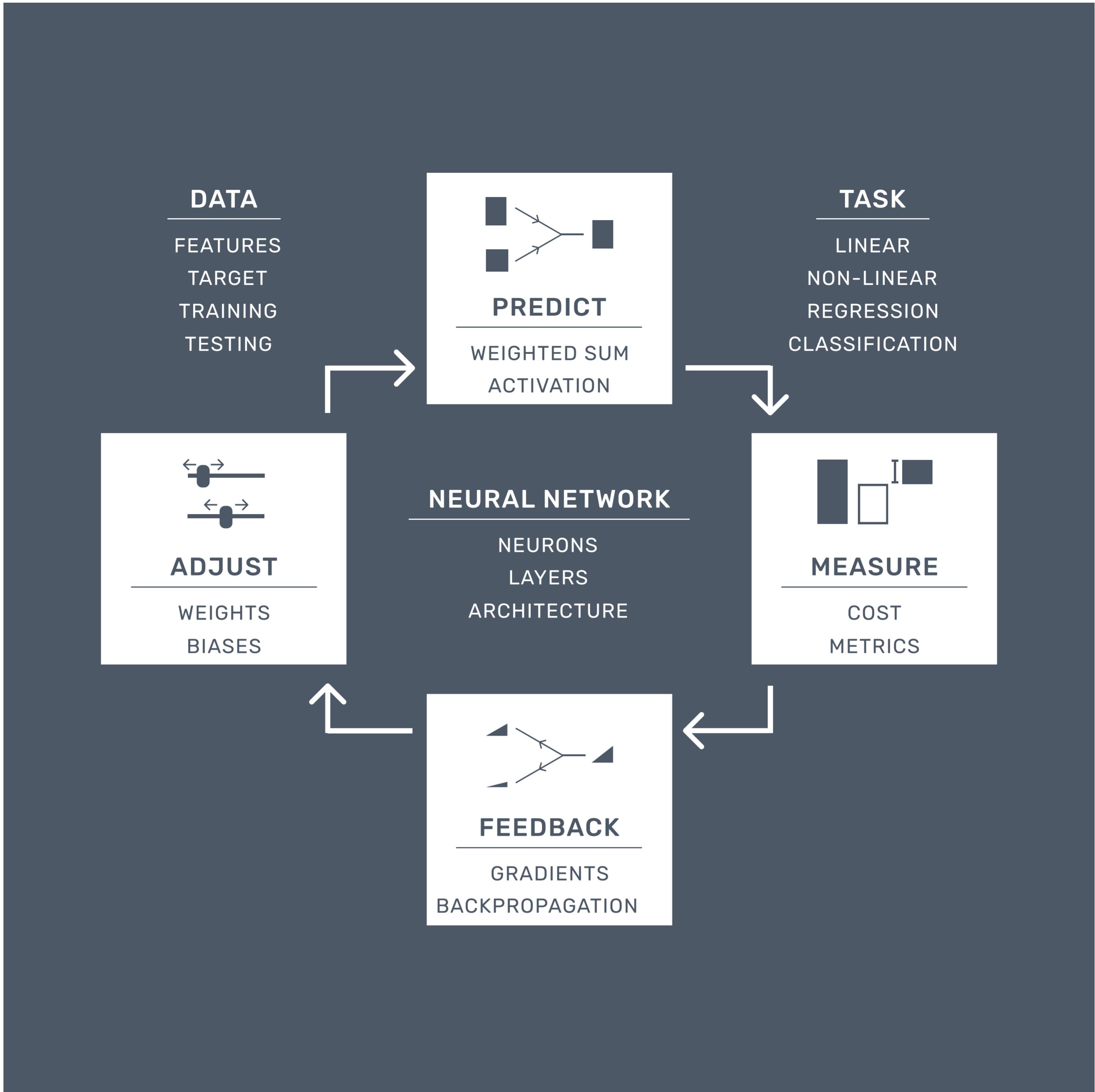
There are many other architectures out there, and new ones continue to emerge. It is an exciting area of research where breakthroughs keep on coming.



## CONCLUSION

Deep learning has immense potential to address some of the world's toughest challenges. Of course, it is not the solution to all problems, but it has proven its versatility and is making an impact in various industries and verticals.

By and large, it is still a relatively new technology. The opportunities are wide open for us to innovate and create solutions that make the world a better place.



## KEY CONCEPTS REVISITED

We have now reached the end of the book. Let's now return to this summary of the key concepts that we have covered. It is a good time to revisit them and review your understanding about deep learning.

## SUGGESTED RESOURCES

If you are ready to go further into your deep learning journey and are looking to get hands-on practice, here are some books that I suggest for you to check out.

- *Grokking Deep Learning* by Andrew W. Trask [Manning Publications]
- *Neural Networks and Deep Learning* by Michael Nielsen [Free Online]
- *Math for Deep Learning* by Ronald T. Kneusel [No Starch Press]
- *Deep Learning with PyTorch* by Eli Stevens, Luca Antiga, and Thomas Viehmann [Manning Publications]
- *Python Machine Learning* by Sebastian Raschka and Vahid Mirjalili [Packt Publishing]
- *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron [O'Reilly Media]
- *Grokking Machine Learning* by Luis G. Serrano [Manning Publications]
- *Deep Learning with Python* by François Chollet [Manning Publications]
- *Deep Learning Illustrated* by Jon Krohn, Grant Beyleveld, and Aglaé Bassens [Addison-Wesley Professional]
- *Deep Learning: A Visual Approach* by Andrew Glassner [No Starch Press]
- *Generative Deep Learning* by David Foster [O'Reilly Media]

I just wanted to mention that there are way more resources out there than listed here. You can treat this as a starting point but by no means an exhaustive list of resources.

## THANK YOU

Thank you so much for reading my book! I hope you have enjoyed reading it.

If you have received value from it, I'd be so grateful to receive a short testimonial from you, to be displayed on the book's product website. It's the best gift I can get from a reader. And it will go a long way to support me in my endeavor.

You can do so by clicking on the button below.

[Leave a Testimonial](#)

If you have any other questions or feedback, do send them over to [contact@kddimensions.com](mailto:contact@kddimensions.com).

Meor Amer



@kdimensions1