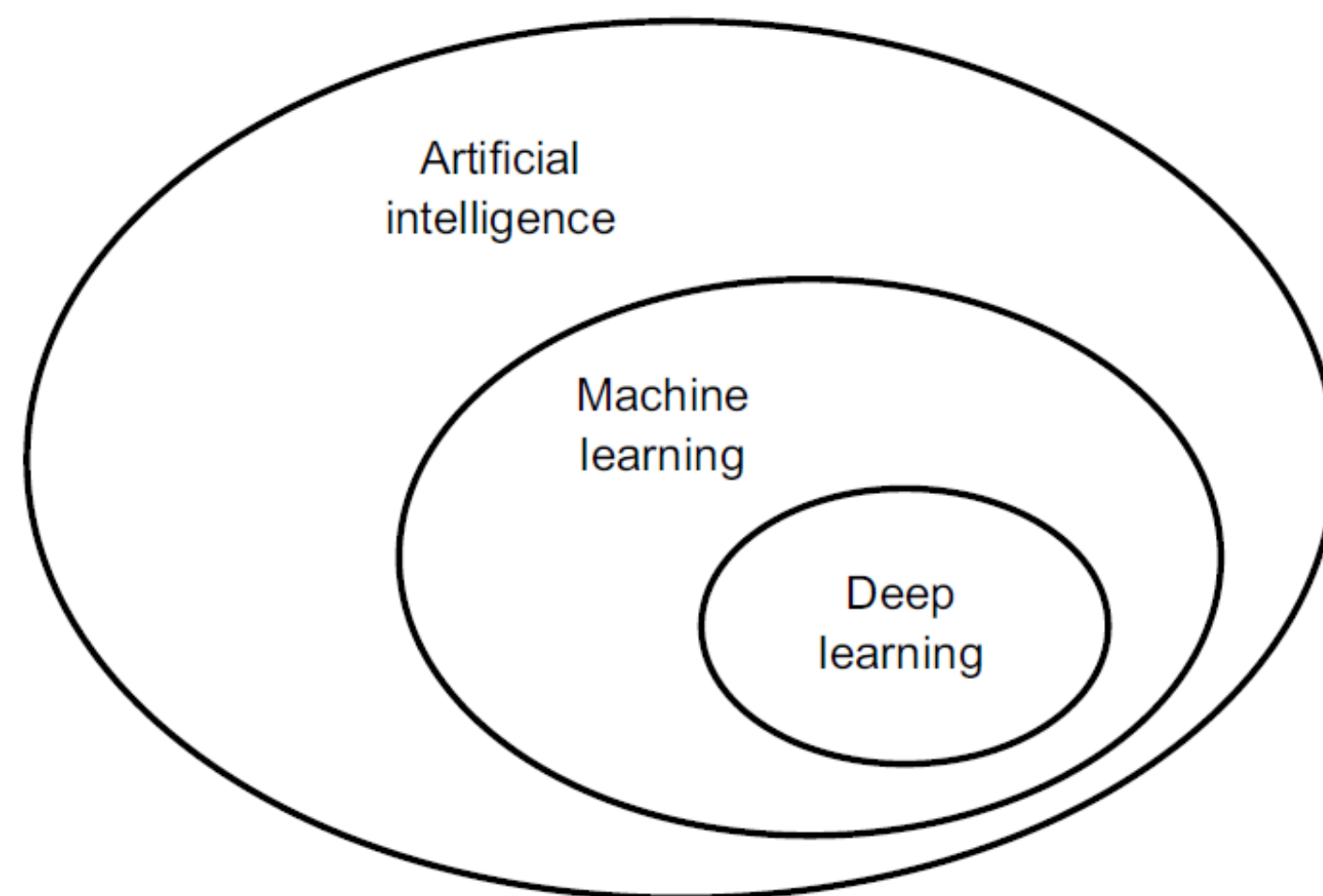


# DEEP LEARNING

So what is deep learning? Deep learning is an approach to machine learning characterized by deep stacks of computations. This depth of computation is what has enabled deep learning models to disentangle the kinds of complex and hierarchical patterns found in the most challenging real-world datasets.

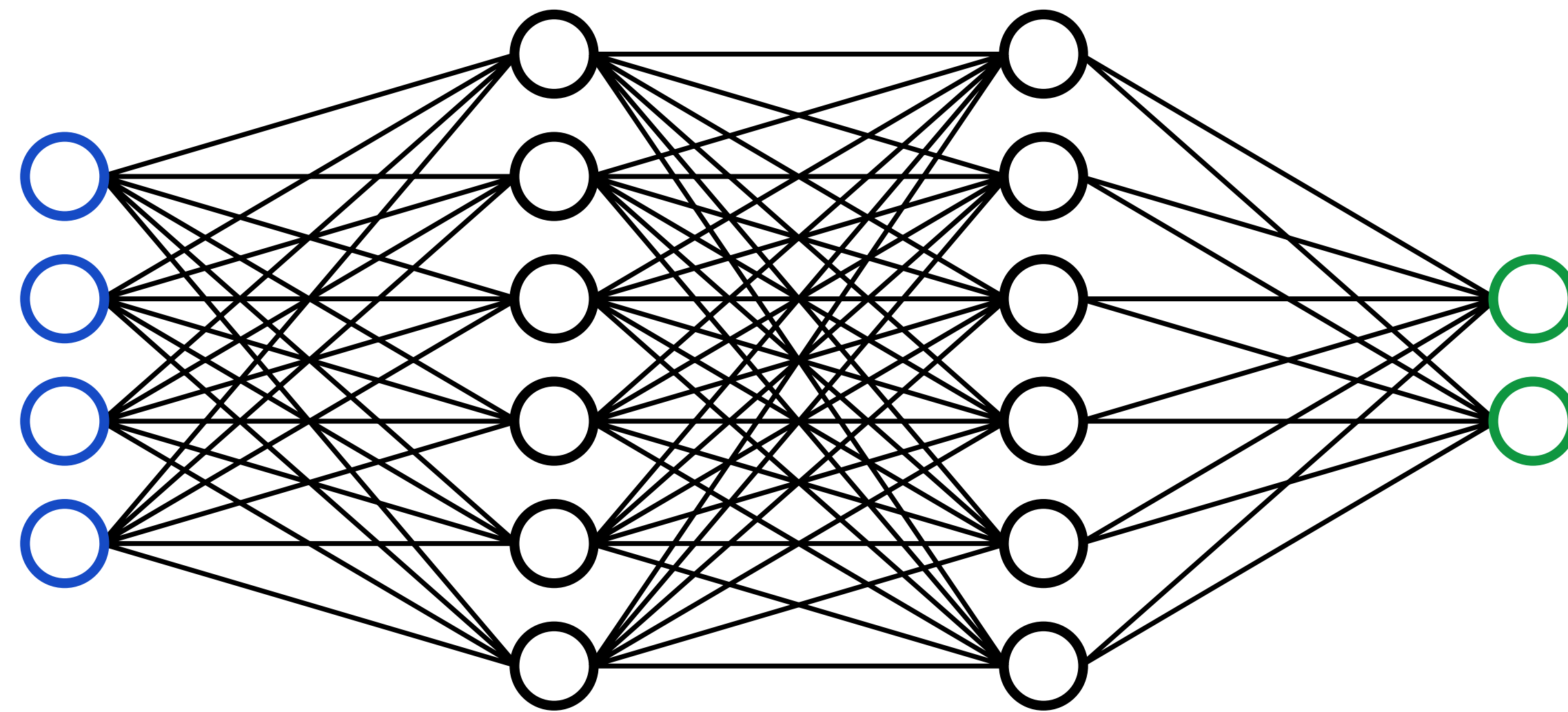


**Machine Learning:** The computer looks at the input data and the corresponding answers, and figures out what the rules should be.

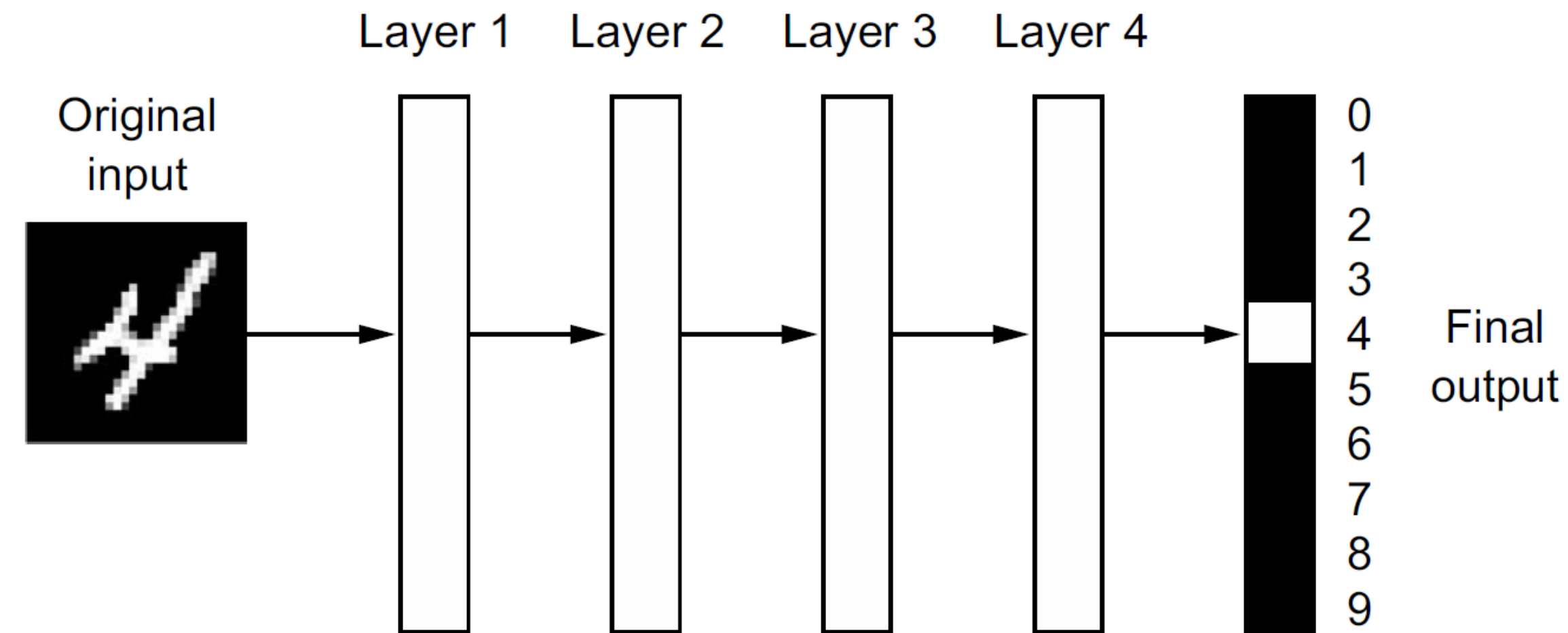
**Deep learning** is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. The “deep” in “deep learning” isn’t a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations.

# NUERAL NETWORK

Through their power and scalability neural networks have become the defining model of deep learning. Neural networks are composed of neurons, where each neuron individually performs only a simple computation. The power of a neural network comes instead from the complexity of the connections these neurons can form.



# Deep neural network for digit classification

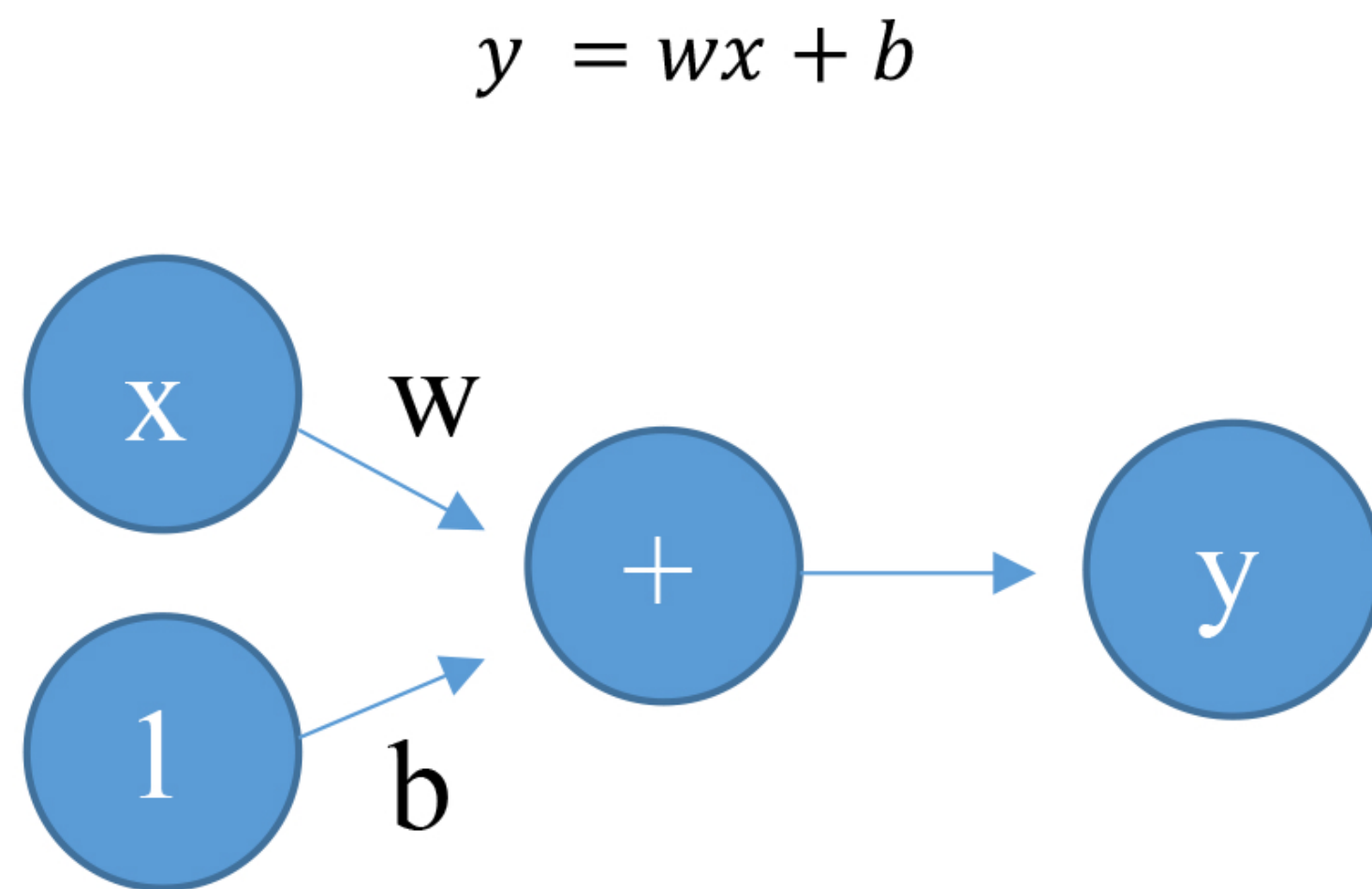


The network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result. You can think of a deep network as a multistage information distillation process, where information goes through successive filters and comes out increasingly purified (that is, useful with regard to some task).

# NUERON

## The Linear Unit

The fundamental component of a neural network: the individual neuron.  
As a diagram, a neuron (or unit) with one input looks like:



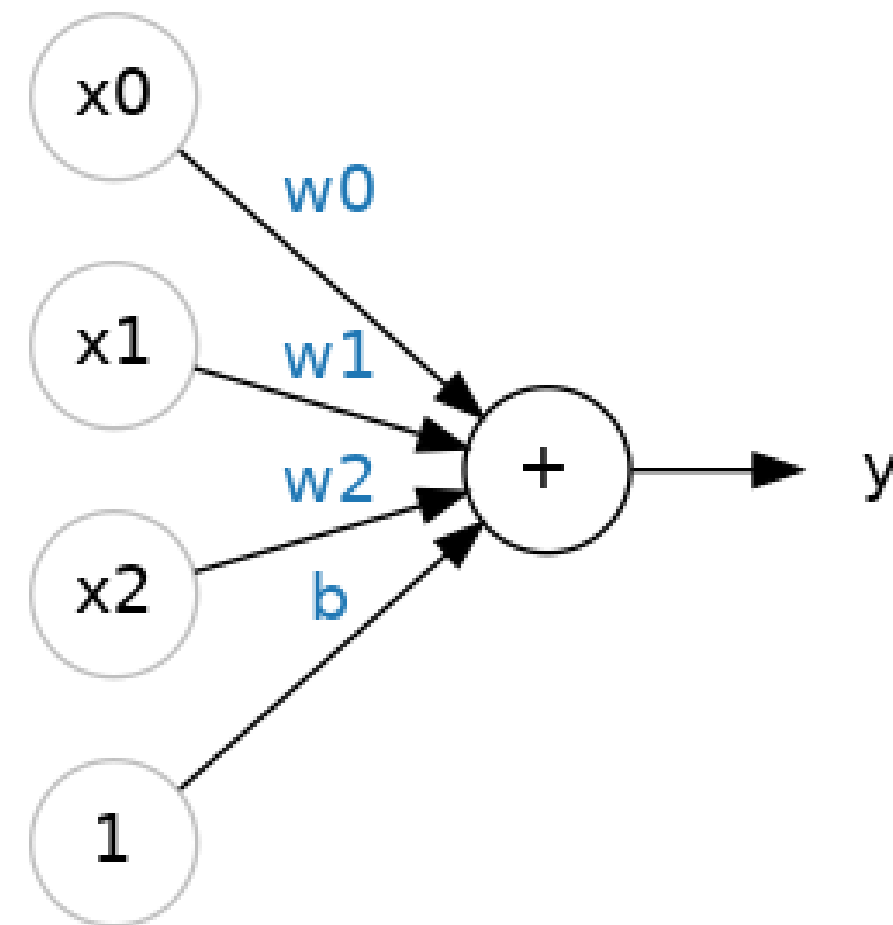
The input is  $x$ . Its connection to the neuron has a weight which is  $w$ . Whenever a value flows through a connection, you multiply the value by the connection's weight. For the input  $x$ , what reaches the neuron is  $w * x$ . A neural network “learns” by modifying its weights.

The  $b$  is a special kind of weight we call the bias. The bias doesn't have any input data associated with it; instead, we put a 1 in the diagram so that the value that reaches the neuron is just  $b$  (since  $1 * b = b$ ). The bias enables the neuron to modify the output independently of its inputs.

The  $y$  is the value the neuron ultimately outputs. To get the output, the neuron sums up all the values it receives through its connections. This neuron's activation is  $y = w * x + b$ , or as a formula  $y=wx+b$ .

# MULTIPLE CONNECTIONS

We can just add more input connections to the neuron, one for each additional feature. To find the output, we would multiply each input to its connection weight and then add them all together.



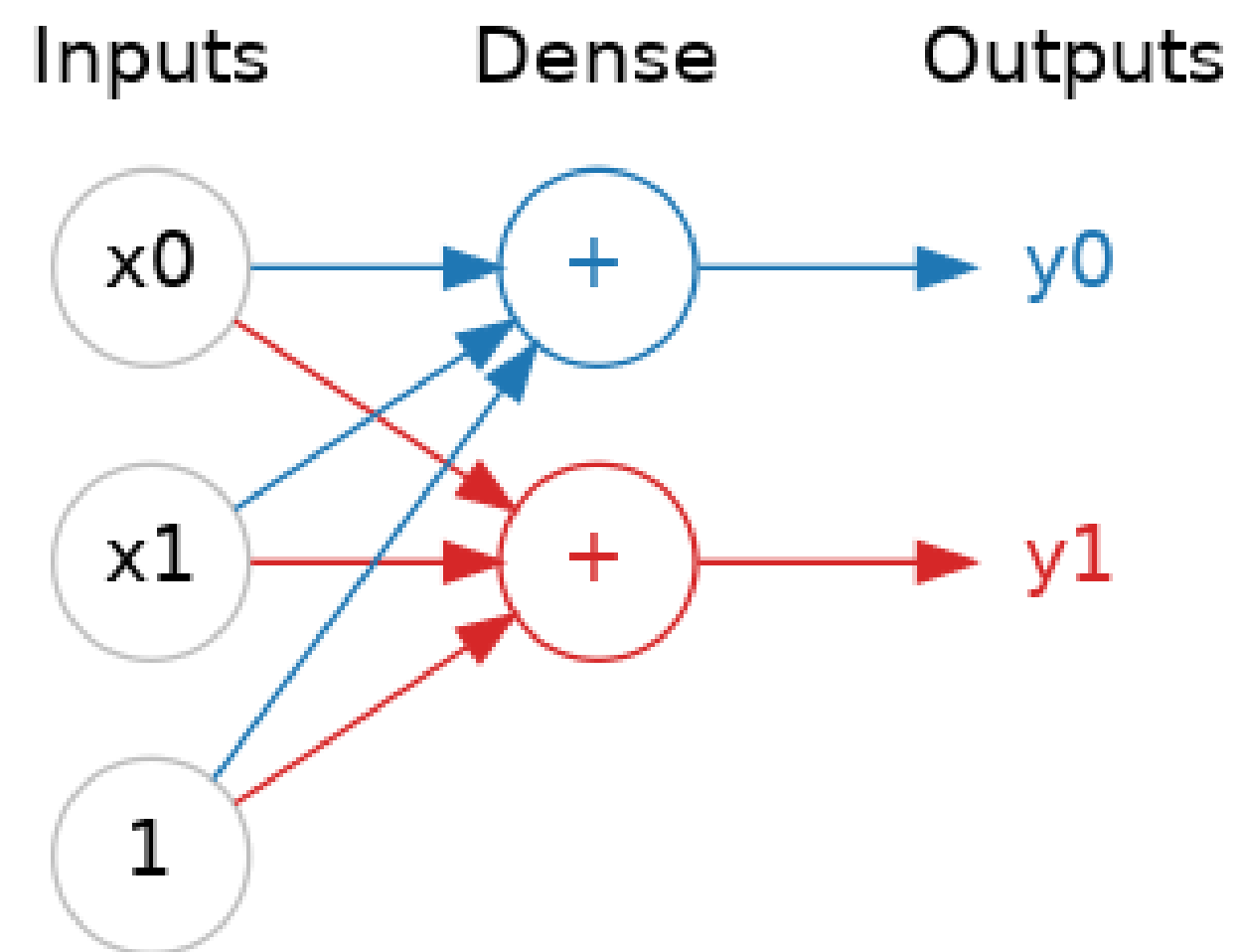
The formula for this neuron would be:

$$y = w_0x_0 + w_1x_1 + w_2x_2 + b$$

A linear unit with two inputs will fit a plane, and a unit with more inputs than that will fit a hyperplane.

# LAYERS OF NEURONS

Neural networks typically organize their neurons into layers. When we collect together linear units having a common set of inputs we get a dense layer.

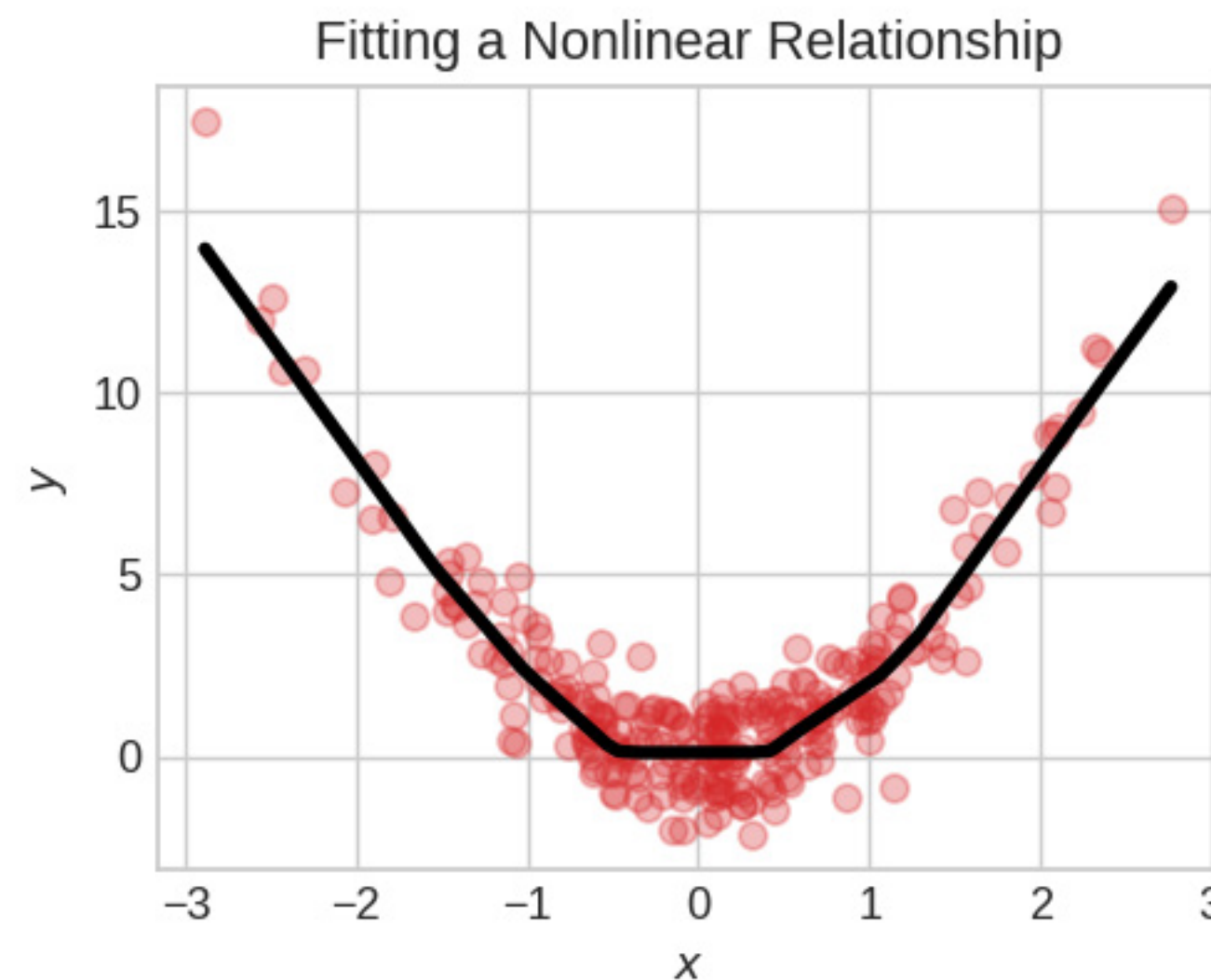


You could think of each layer in a neural network as performing some kind of relatively simple transformation. Through a deep stack of layers, a neural network can transform its inputs in more and more complex ways. In a well-trained neural network, each layer is a transformation getting us a little bit closer to a solution.



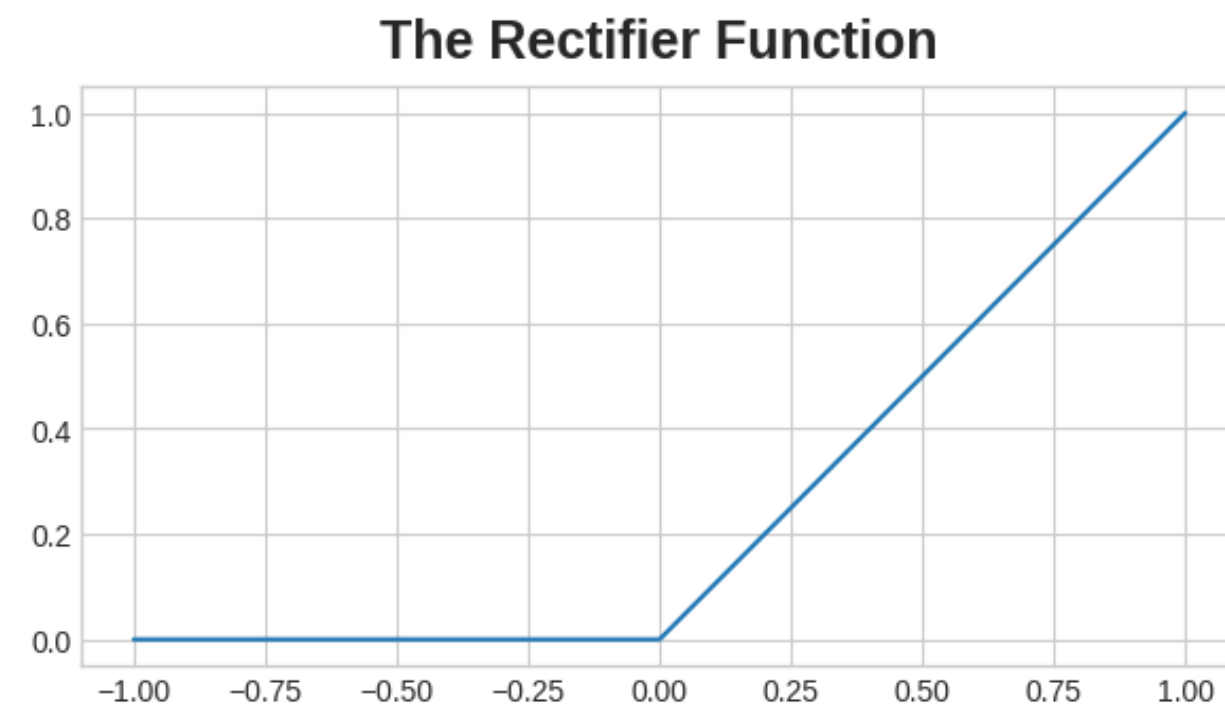
# ACTIVATION FUNCTION

It turns out, however, that two dense layers with nothing in between are no better than a single dense layer by itself. Dense layers by themselves can never move us out of the world of lines and planes. What we need is something nonlinear. What we need are activation functions.



# NONLINEARITY ReLu

An activation function is simply some function we apply to each of a layer's outputs (its activations). The most common is the rectifier function  $\max(0, x)$ .



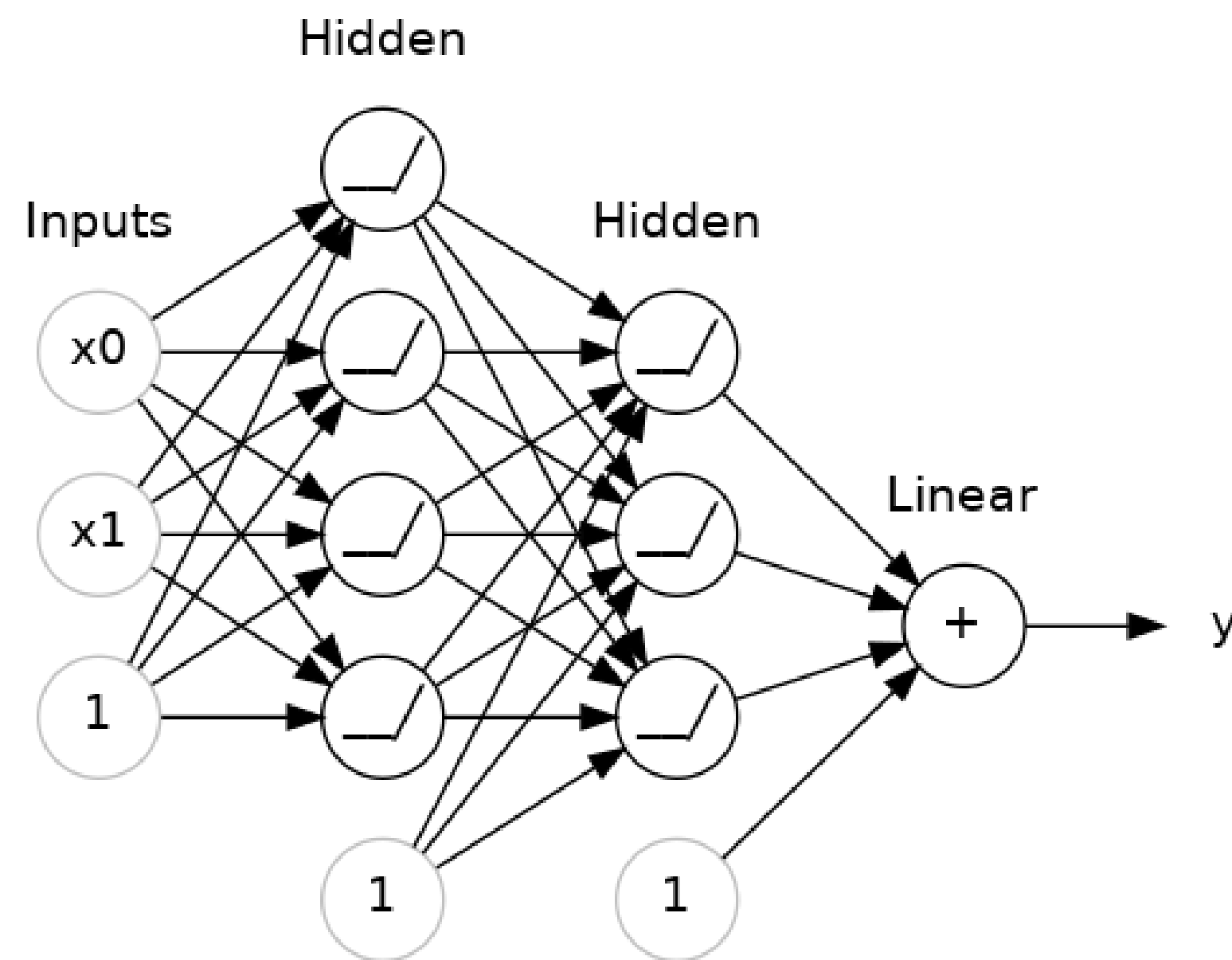
The rectifier function has a graph that's a line with the negative part "rectified" to zero. Applying the function to the outputs of a neuron will put a bend in the data, moving us away from simple lines.

When we attach the rectifier to a linear unit, we get a rectified linear unit or ReLU. (For this reason, it's common to call the rectifier function the "ReLU function".) Applying a ReLU activation to a linear unit means the output becomes  $\max(0, w * x + b)$



# STACKED LAYERS

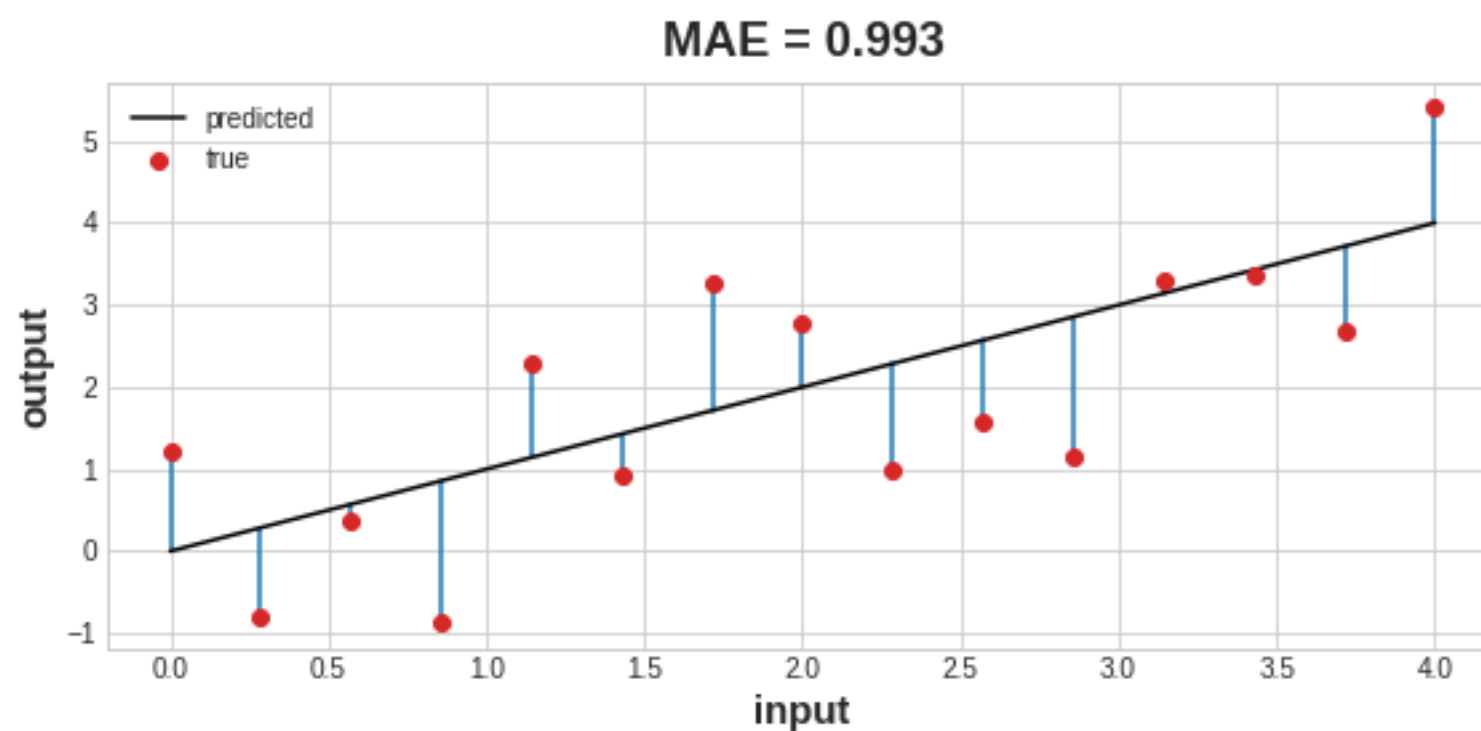
A stack of dense layers makes a “fully-connected” network.



The layers before the output layer are sometimes called hidden since we never see their outputs directly.

# LOSS FUNCTION

We've seen how to design an architecture for a network, but we haven't seen how to tell a network what problem to solve. This is the job of the loss function.



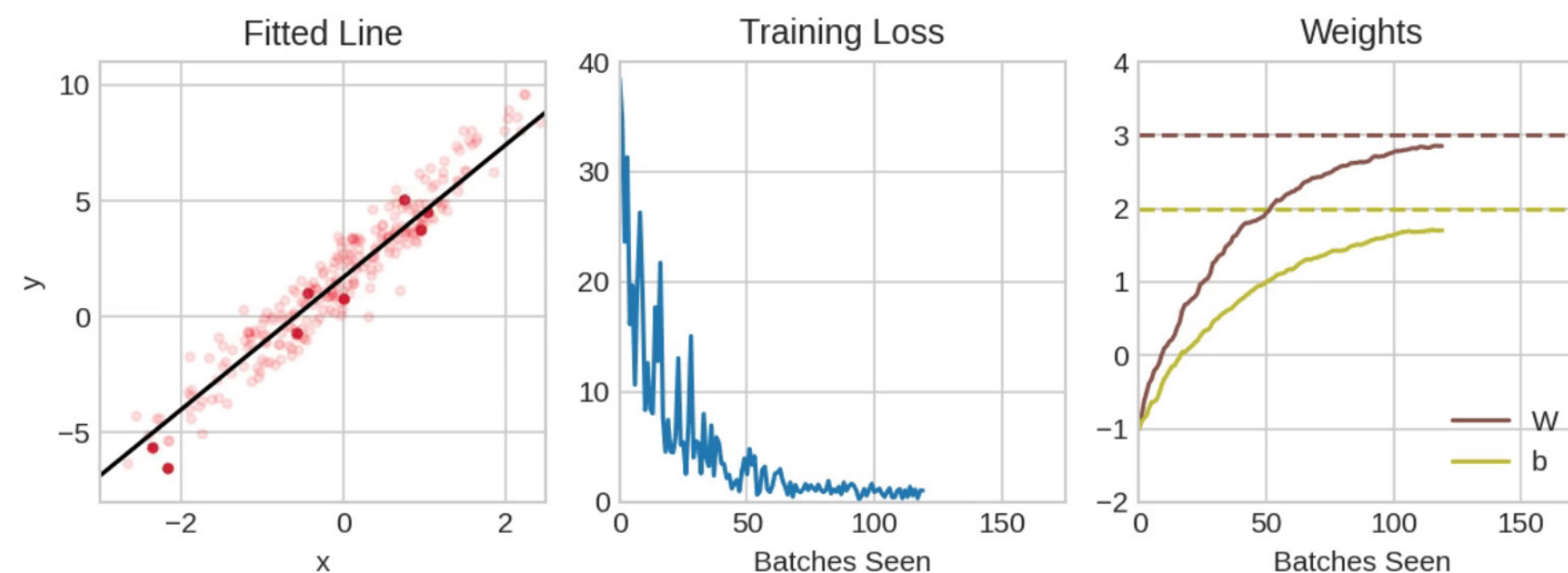
In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks: regression and classification loss functions

Regression Loss Functions — used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels); Ex. Mean Squared Error, Mean Absolute Error

Classification Loss Functions — used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging; Ex. Binary Cross-Entropy, Categorical Cross-Entropy

# OPTOMIZER

We've described the problem we want the network to solve, but now we need to say how to solve it. This is the job of the optimizer. The optimizer is an algorithm that adjusts the weights to minimize the loss.

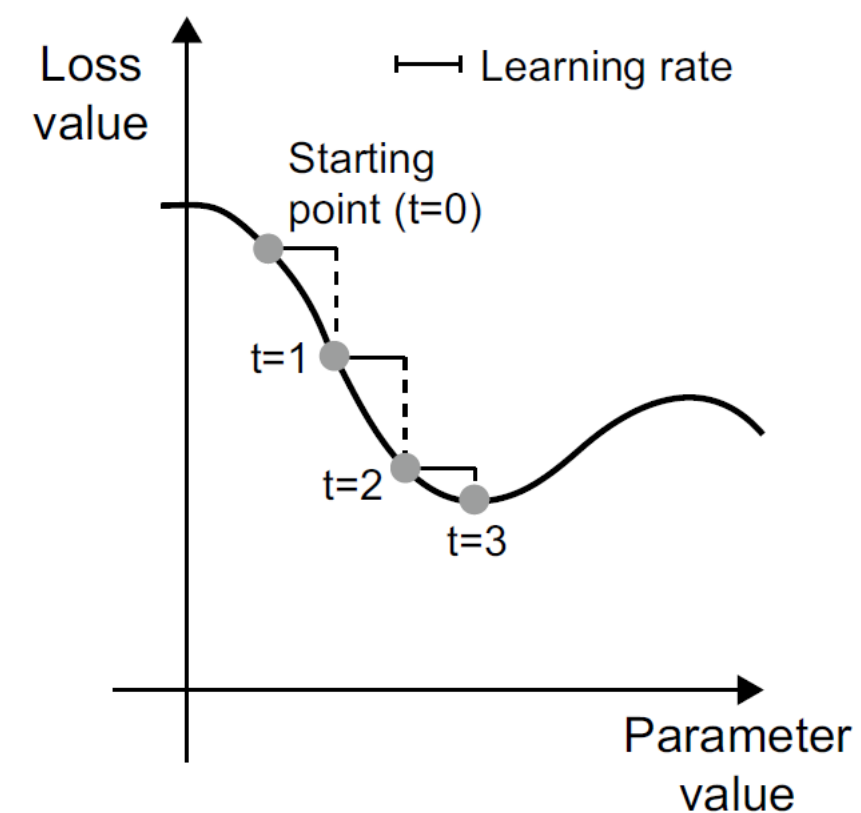


Virtually all of the optimization algorithms used in deep learning belong to a family called stochastic gradient descent. They are iterative algorithms that train a network in steps. One step of training goes like this:

- 1) Sample some training data and run it through the network to make predictions.
- 2) Measure the loss between the predictions and the true values.
- 3) Finally, adjust the weights in a direction that makes the loss smaller.

Then just do this over and over until the loss is as small as you like (or until it won't decrease any further.)

# LEARNING RATE



- 1) Draw a batch of training samples,  $x$ , and corresponding targets,  $y_{\text{true}}$ .
- 2) Run the model on  $x$  to obtain predictions,  $y_{\text{pred}}$  (this is called the forward pass).
- 3) Compute the loss of the model on the batch, a measure of the mismatch between  $y_{\text{pred}}$  and  $y_{\text{true}}$ .
- 4) Compute the gradient of the loss with regard to the model's parameters (this is called the backward pass).
- 5) Move the parameters a little in the opposite direction from the gradient—for example,  $W -= \text{learning\_rate} * \text{gradient}$ —thus reducing the loss on the batch a bit. The learning rate (learning\_rate here) would be a scalar factor modulating the “speed” of the gradient descent process.

# BACKPROPAGATION

How do we update the weights from the outputs of the network? We use calculus!

## THE CHAIN RULE

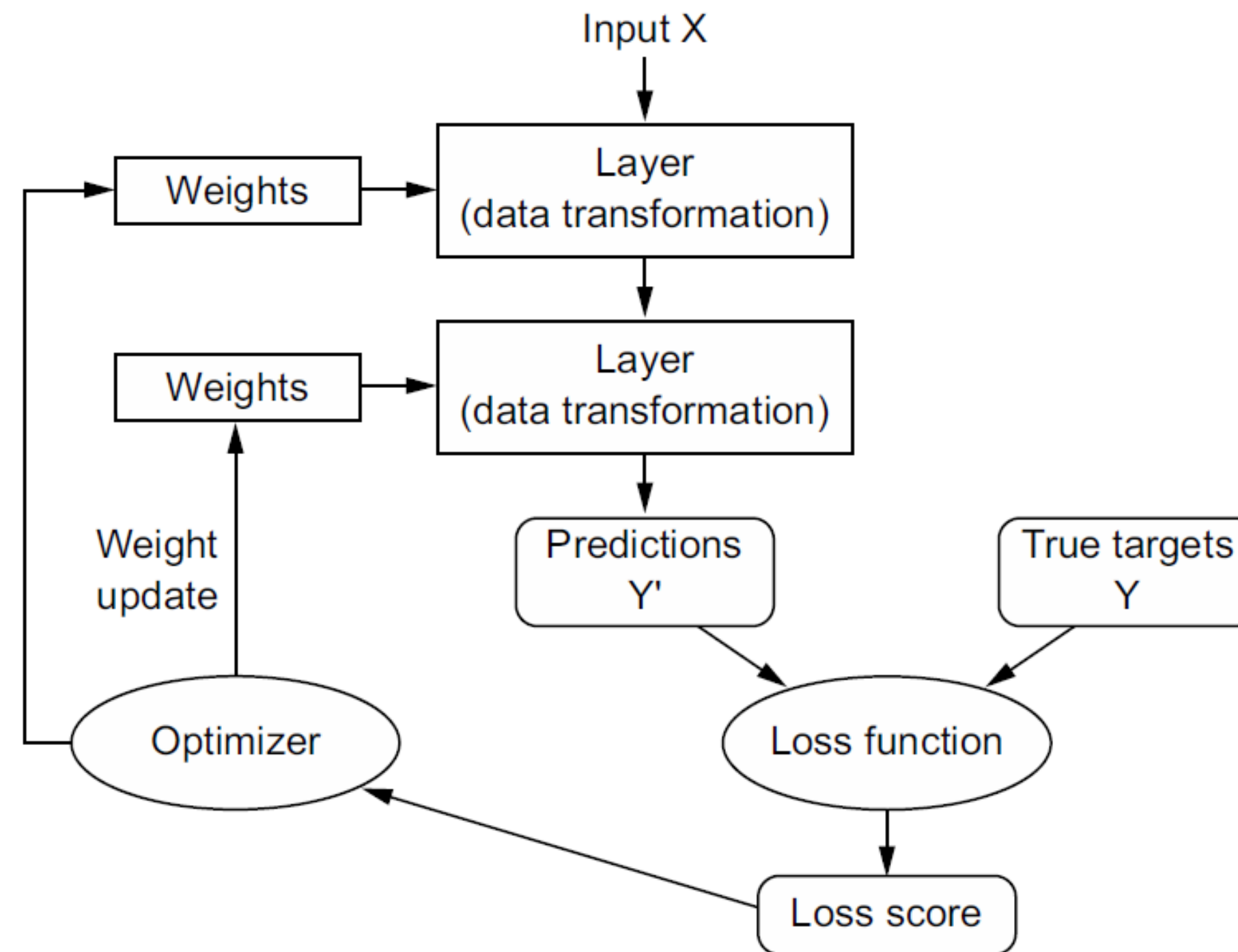
Backpropagation is a way to use the derivatives of simple operations (such as addition, relu, or tensor product) to easily compute the gradient of arbitrarily complex combinations of these atomic operations. Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, the model defined in listing 2.2 can be expressed as a function parameterized by the variables  $W1$ ,  $b1$ ,  $W2$ , and  $b2$  (belonging to the first and second Dense layers respectively), involving the atomic operations dot, relu, softmax, and +, as well as our loss function loss, which are all easily differentiable:

```
loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
```

Calculus tells us that such a chain of functions can be derived using the following identity, called the chain rule. Consider two functions  $f$  and  $g$ , as well as the composed function  $fg$  such that  $fg(x) == f(g(x))$ :

```
def fg(x):  
    x1 = g(x)  
    y = f(x1)  
    return y
```

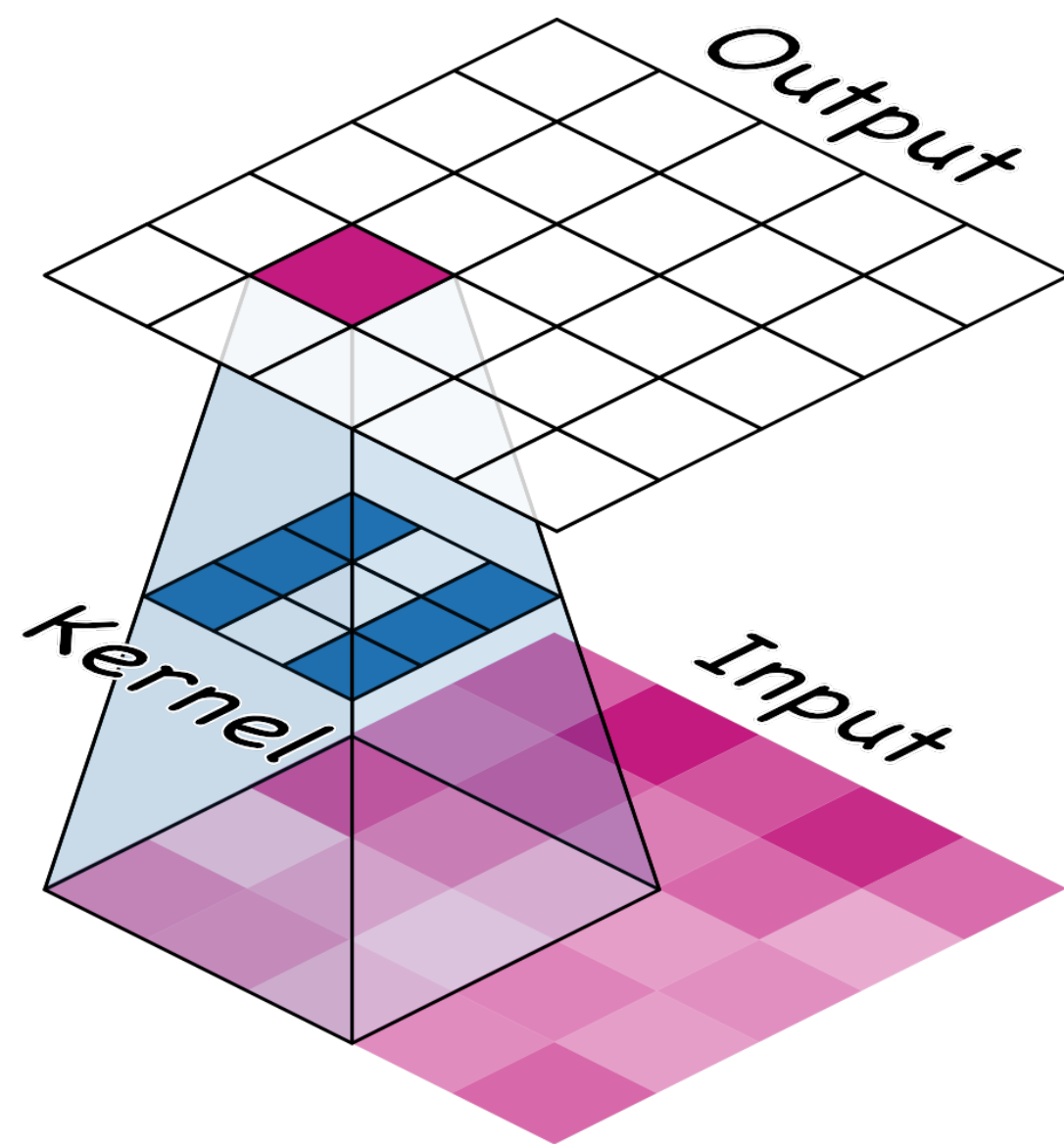
# FULL TRAINING LOOP





# CONVOLUTIONAL NEURAL NETWORK

A convolutional neural network (CNN) is a type of neural network that uses convolutional layers to extract features from images. These features are then used to make predictions about the image, such as the class of the object in the image.



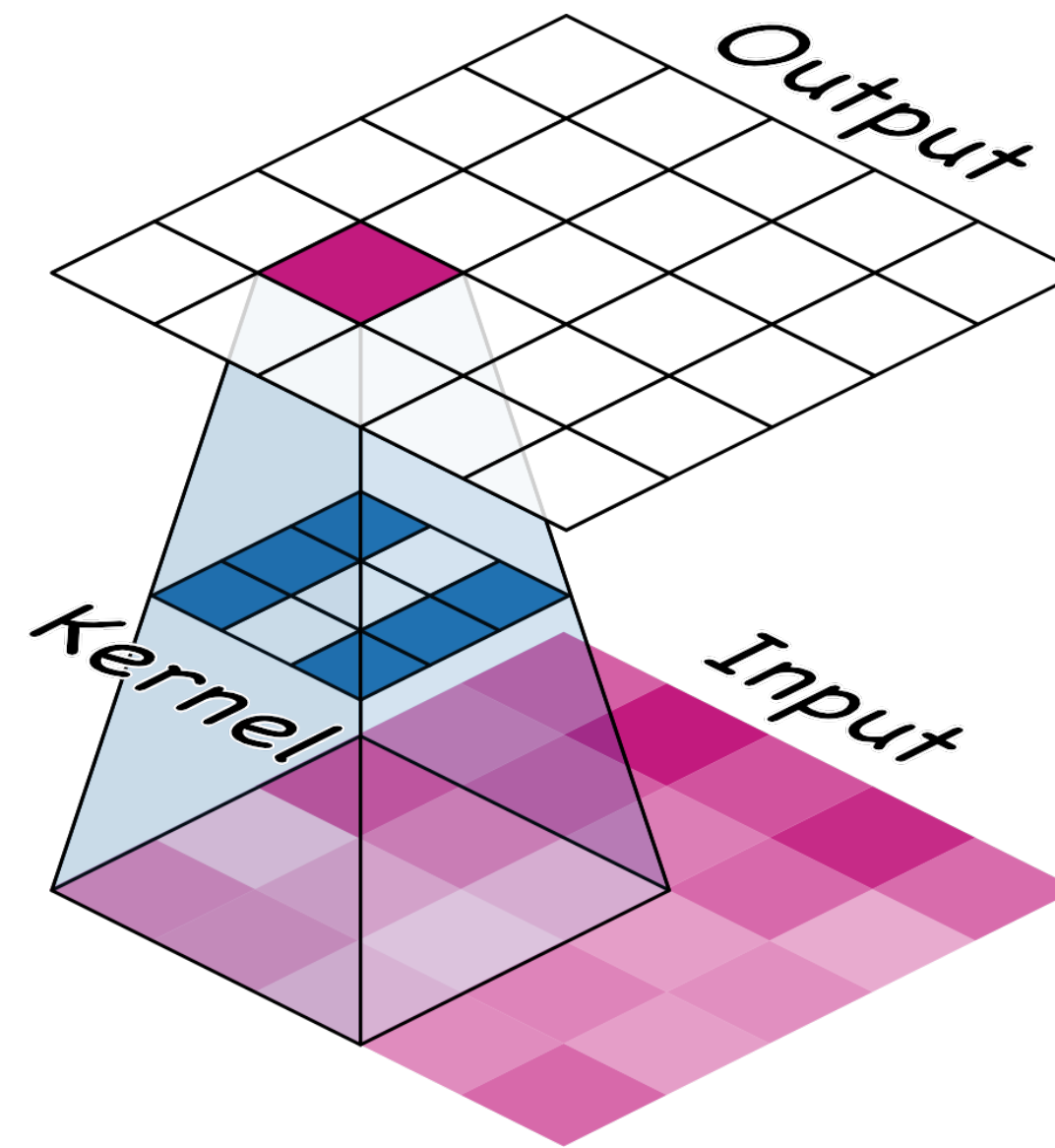
A convolutional layer works by taking a small window of pixels (called a filter or kernel) and sliding it across the image, performing a mathematical operation (convolution) on the pixel values in the window. The result of this operation is a new pixel value that represents a feature of the image. By stacking multiple convolutional layers together, a CNN can learn increasingly complex features that are used to make predictions.

# KERNEL CONVOLUTION

The weights that a convolutional neural network (CNN) learns during training are primarily contained in its convolutional layers, and we refer to them as “kernels” or “filters”.

Kernels are small arrays of numerical values that are learned by the network during training. These values are used to perform the convolution operation on the input image, which extracts features from the image. Each kernel is designed to detect a specific feature, such as edges, textures, or patterns.

Kernels are usually square or rectangular in shape, with dimensions that are typically 3x3 or 5x5 pixels. During training, the values of the kernels are adjusted so that the network can learn to recognize features that are relevant to the task at hand.

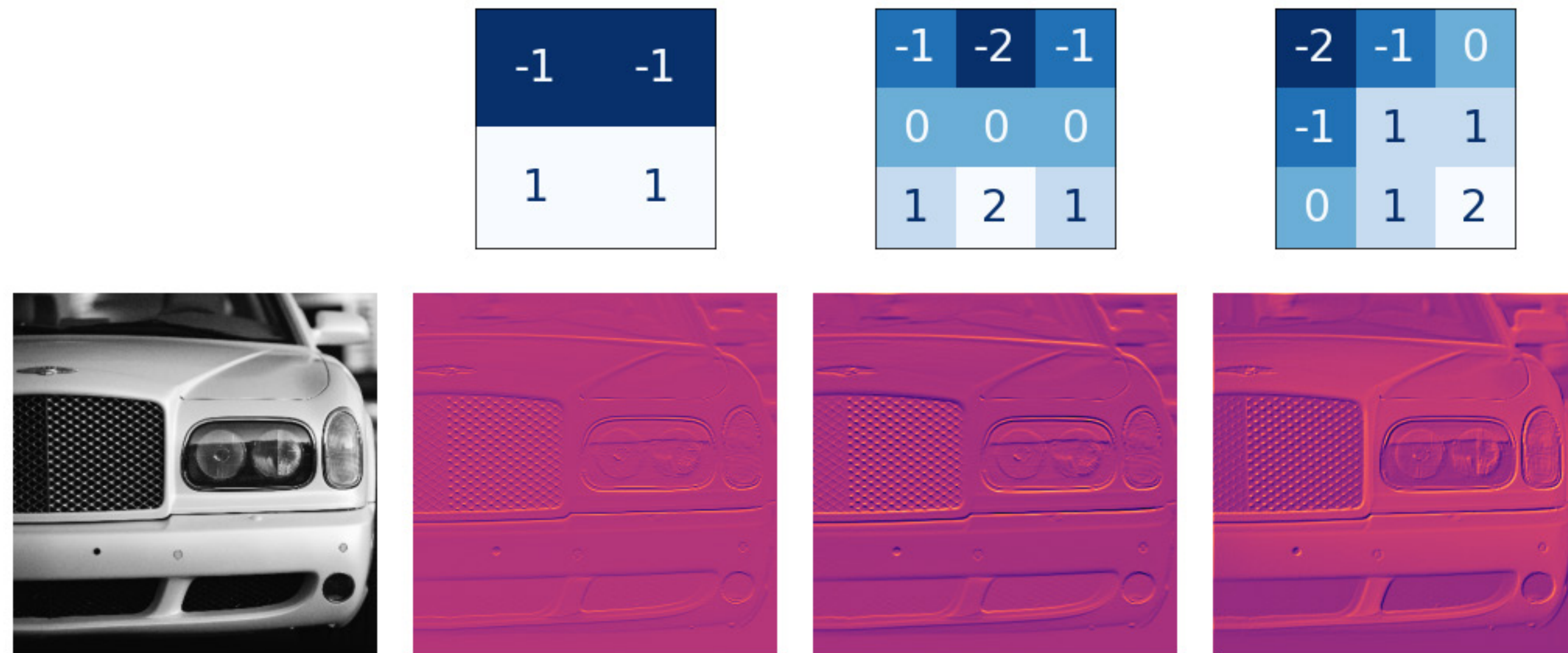


Kernels define how a convolutional layer is connected to the layer that follows. The kernel above will connect each neuron in the output to nine neurons in the input. By setting the dimensions of the kernels with `kernel_size`, you are telling the convnet how to form these connections. Most often, a kernel will have odd-numbered dimensions -- like `kernel_size=(3, 3)` or `(5, 5)` -- so that a single pixel sits at the center, but this is not a requirement.

The kernels in a convolutional layer determine what kinds of features it creates. During training, a convnet tries to learn what features it needs to solve the classification problem. This means finding the best values for its kernels.

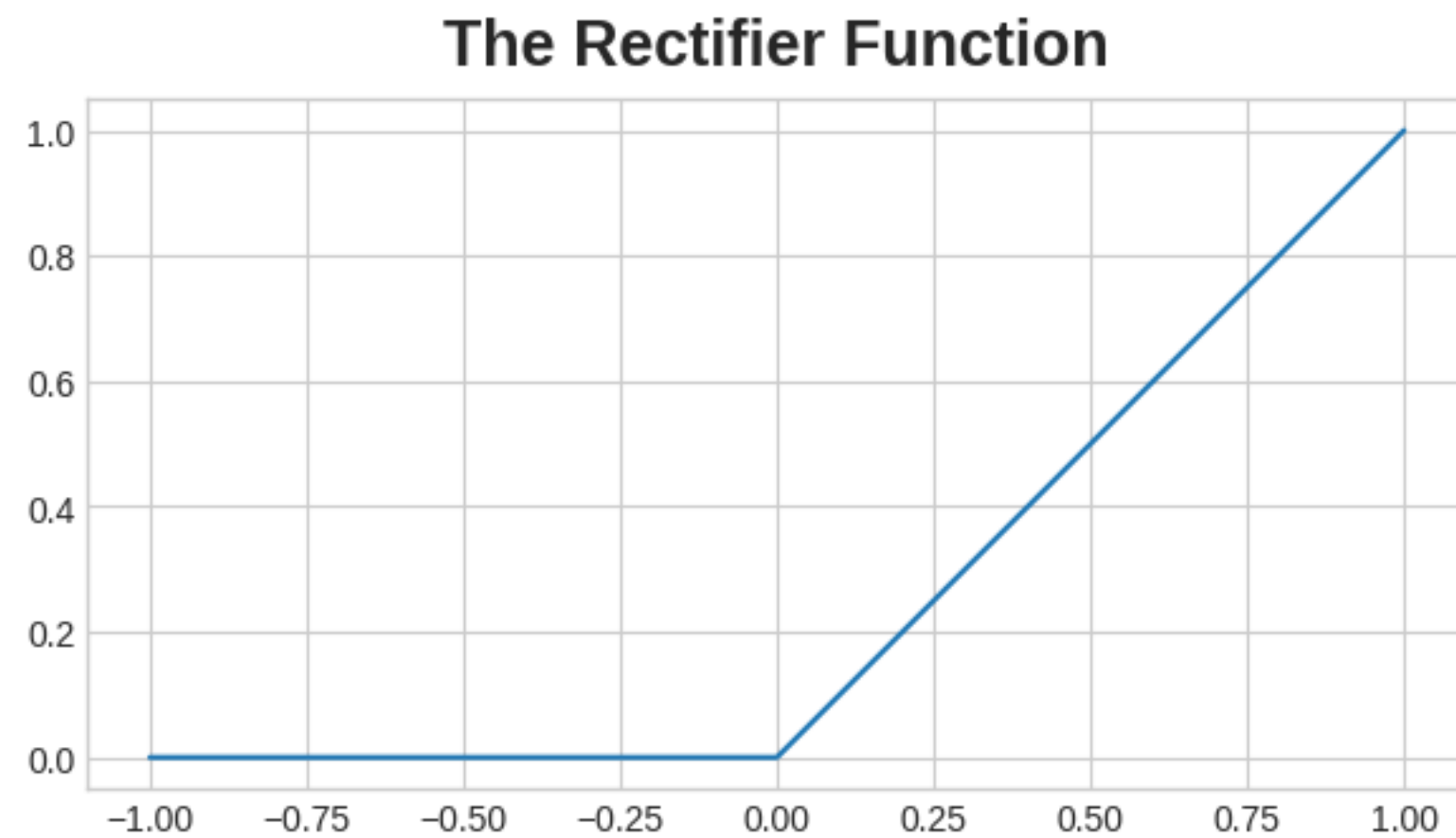
# FEATURE MAPS

The activations in the network we call feature maps. They are what result when we apply a filter to an image; they contain the visual features the kernel extracts. Here are a few kernels pictured with feature maps they produced.



# ACTIVATION

The activations in the network we call feature maps. They are what result when we apply a filter to an image; they contain the visual features the kernel extracts. Here are a few kernels pictured with feature maps they produced.



You could think about the activation function as scoring pixel values according to some measure of importance. The ReLU activation says that negative values are not important and so sets them to 0.

“Everything unimportant is equally unimportant.”



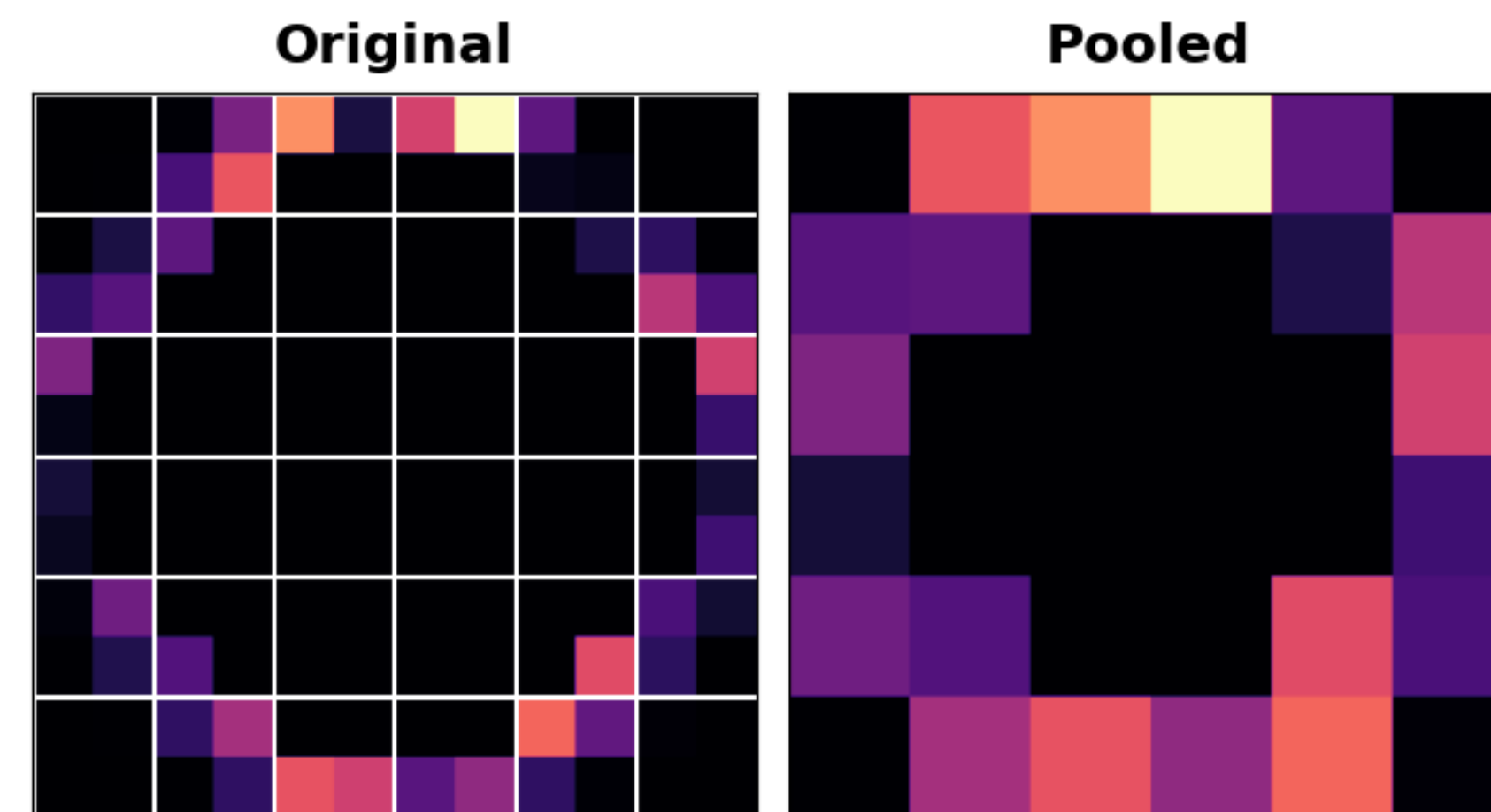
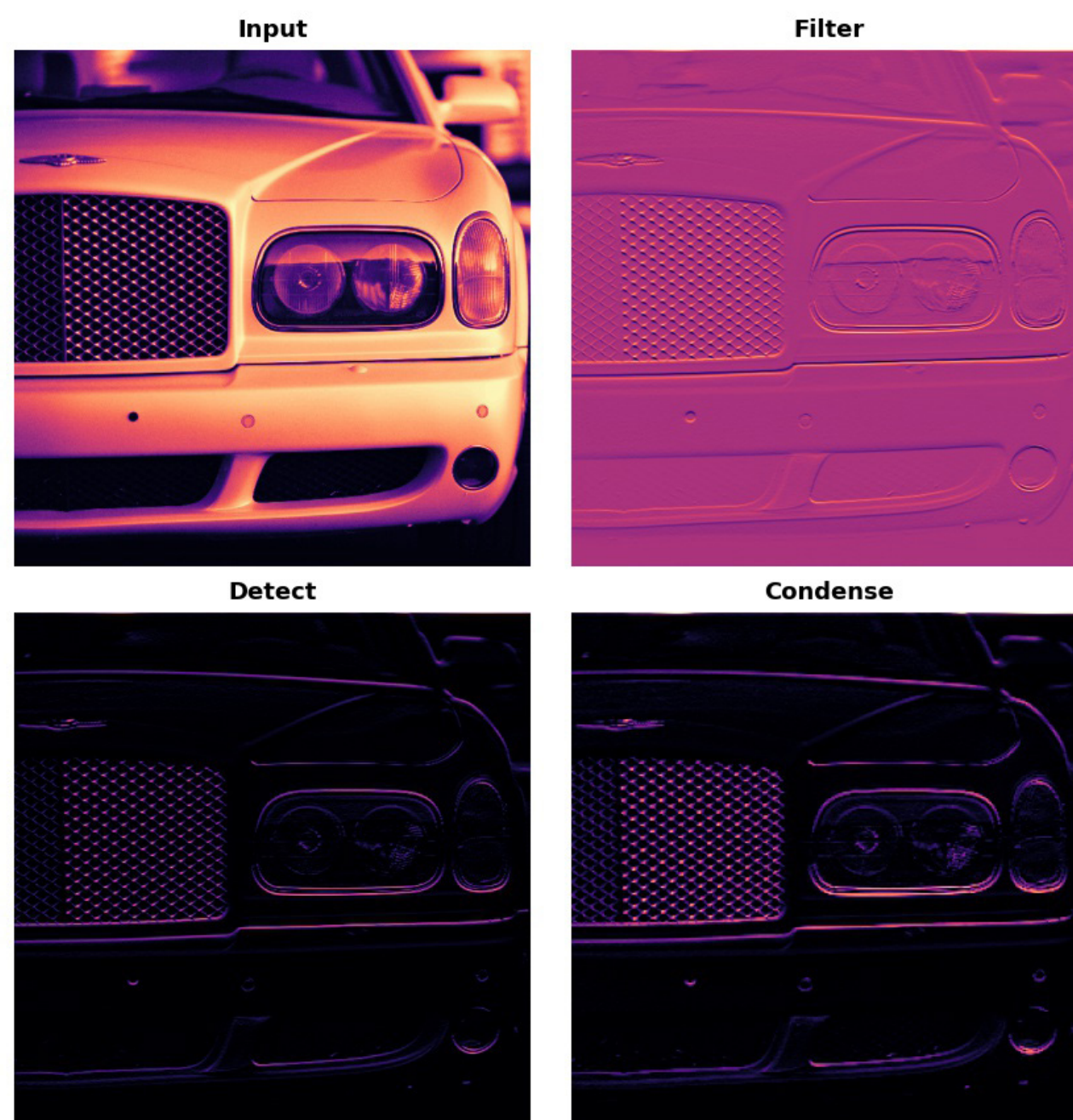


Like other activation functions, the ReLU function is nonlinear. Essentially this means that the total effect of all the layers in the network becomes different than what you would get by just adding the effects together -- which would be the same as what you could achieve with only a single layer. The nonlinearity ensures features will combine in interesting ways as they move deeper into the network.



# CONDENSE

A **MaxPool2D** layer is much like a Conv2D layer, except that it uses a simple maximum function instead of a kernel, with the `pool_size` parameter analogous to `kernel_size`. A MaxPool2D layer doesn't have any trainable weights like a convolutional layer does in its kernel.



# STRIDE & PADDING

## Stride

The distance the window moves at each step is called the stride. We need to specify the stride in both dimensions of the image: one for moving left to right and one for moving top to bottom.

What effect does the stride have? Whenever the stride in either direction is greater than 1, the sliding window will skip over some of the pixels in the input at each step.

Because we want high-quality features to use for classification, convolutional layers will most often have `strides=(1, 1)`. Increasing the stride means that we miss out on potentially valuable information in our summary. Maximum pooling layers, however, will almost always have stride values greater than 1, like (2, 2) or (3, 3), but not larger than the window itself.

Finally, note that when the value of the strides is the same number in both directions, you only need to set that number; for instance, instead of `strides=(2, 2)`, you could use `strides=2` for the parameter setting.

## Padding

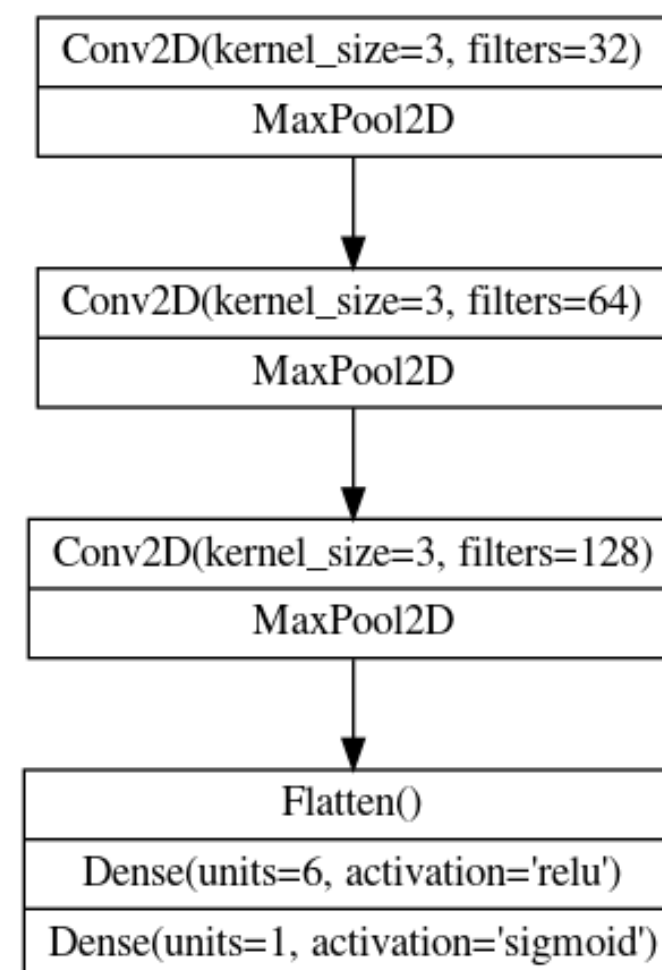
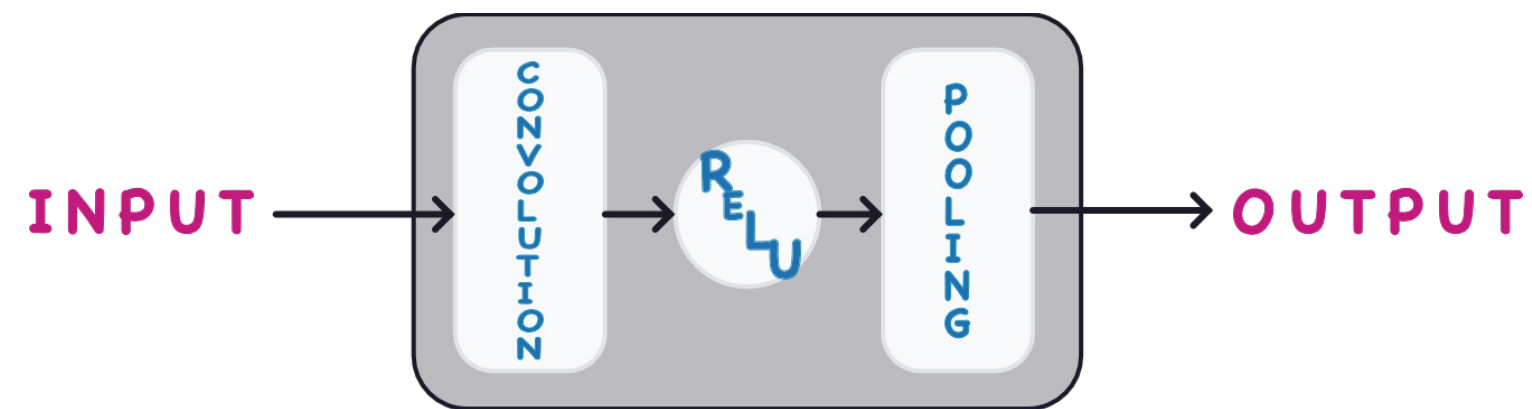
When performing the sliding window computation, there is a question as to what to do at the boundaries of the input. Staying entirely inside the input image means the window will never sit squarely over these boundary pixels like it does for every other pixel in the input. Since we aren't treating all the pixels exactly the same, could there be a problem?

What the convolution does with these boundary values is determined by its padding parameter. In TensorFlow, you have two choices: either `padding='same'` or `padding='valid'`. There are trade-offs with each.

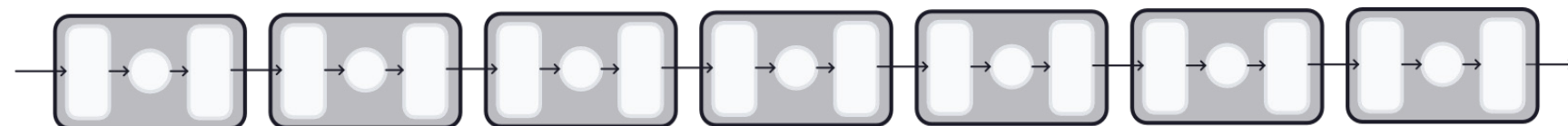
When we set `padding='valid'`, the convolution window will stay entirely inside the input. The drawback is that the output shrinks (loses pixels), and shrinks more for larger kernels. This will limit the number of layers the network can contain, especially when inputs are small in size.

The alternative is to use `padding='same'`. The trick here is to pad the input with 0's around its borders, using just enough 0's to make the size of the output the same as the size of the input. This can have the effect however of diluting the influence of pixels at the borders. The animation below shows a sliding window with 'same' padding.

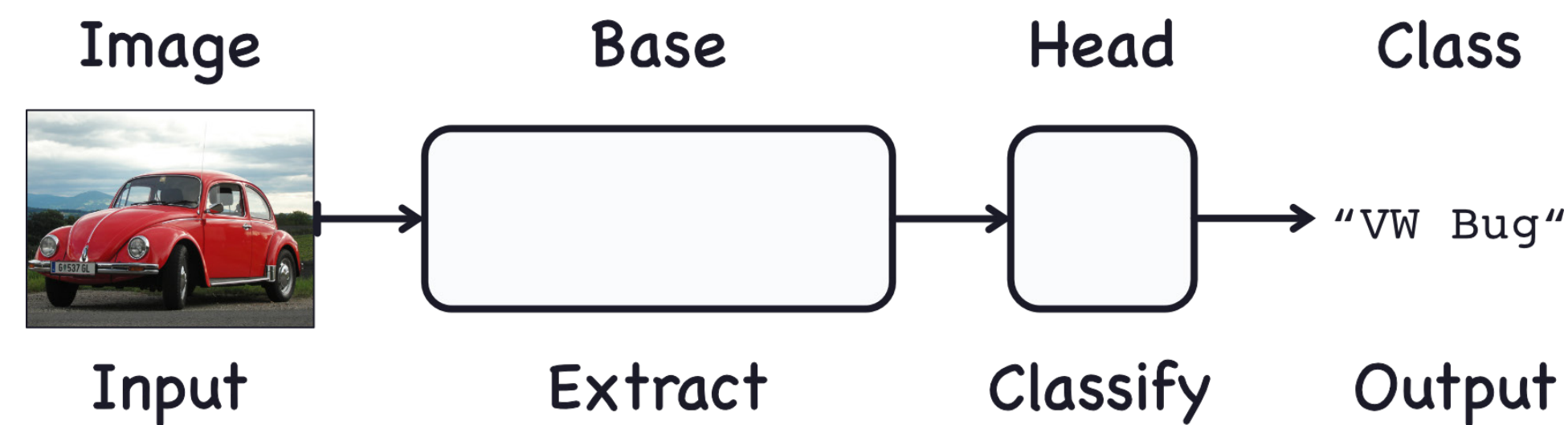
# CONVOLUTIONAL BLOCK



Each block represents a round of extraction, and by composing these blocks the convnet can combine and recombine the features produced, growing them and shaping them to better fit the problem at hand. The deep structure of modern convnets is what allows this sophisticated feature engineering and has been largely responsible for their superior performance.



# CNN WITH CLASSIFIER



photograph by Dnalar 1 (Wikimedia Commons) CC-BY-SA 3.0

The **base** of the CNN is typically composed of several convolutional layers and pooling layers. The convolutional layers extract features from the input image, while the pooling layers reduce the spatial size of the feature maps, making the network more computationally efficient. The base of the CNN can also include normalization layers, such as batch normalization, to improve the stability and performance of the network.

The **head** of the CNN is typically composed of one or more fully connected layers. These layers take the features extracted by the convolutional layers and use them to make a prediction. The head of the CNN can also include regularization techniques, such as dropout or weight decay, to prevent overfitting.

# CNN ARCHITECTURE