SURYA GROUP OF INSTITUTIONS

NAAN MUDHALVAN

IBM-ARTIFICIAL INTELLIGENCE

SANTHOSH R

422221104034

AI-Driven Exploration and Prediction

TEAM : 08

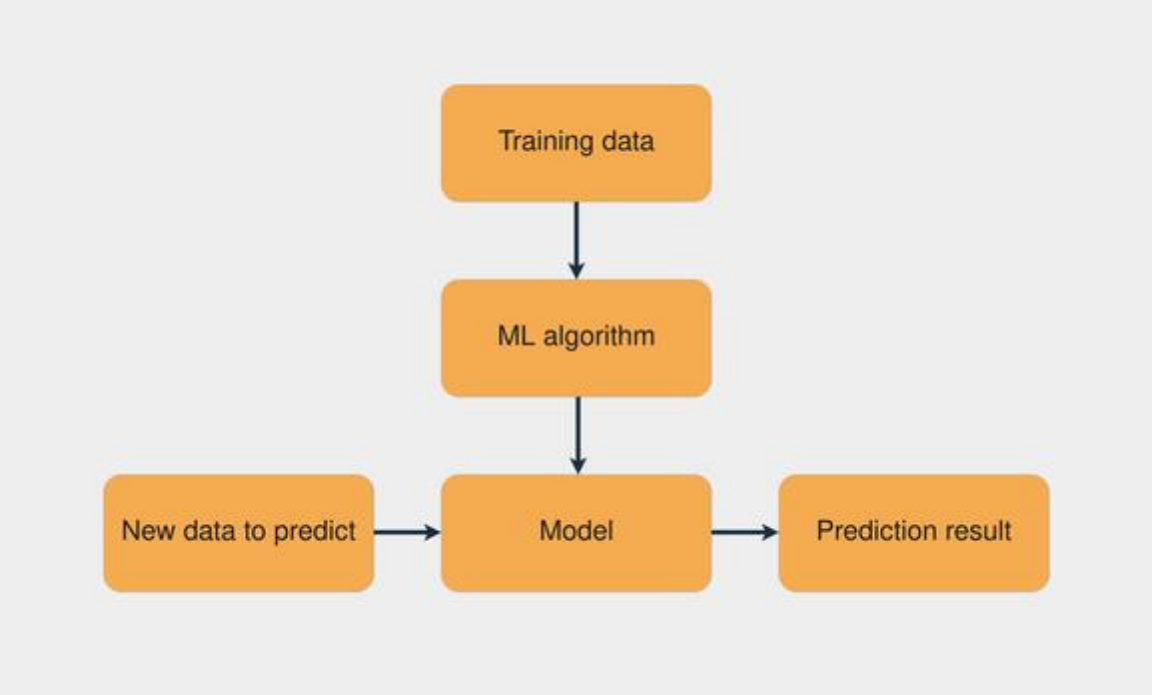
AI-Driven Exploration and Prediction Machine Learning Algorithms In Python

Table of Contents

Machine Learning:

Machine learning is a technique in which you train the system to solve a problem instead of explicitly programming the rules. Getting back to the sudoku example in the previous section, to solve the problem using machine learning, you would gather data from solved sudoku games and train a statistical model. Statistical models are mathematically formalized ways to approximate the behavior of a phenomenon.

A common machine learning task is supervised learning, in which you have a dataset with inputs and known outputs. The task is to use this dataset to train a model that predicts the correct outputs based on the inputs. The image below presents the workflow to train a model using supervised learning:



The combination of the training data with the machine learning algorithm creates the model. Then, with this model, you can make predictions for new data.

The goal of supervised learning tasks is to make predictions for new, unseen data. To do that, you assume that this unseen data follows a probability distribution similar to the distribution of the training dataset. If in the future this distribution changes, then you need to train your model again using the new training dataset.

Feature Engineering:

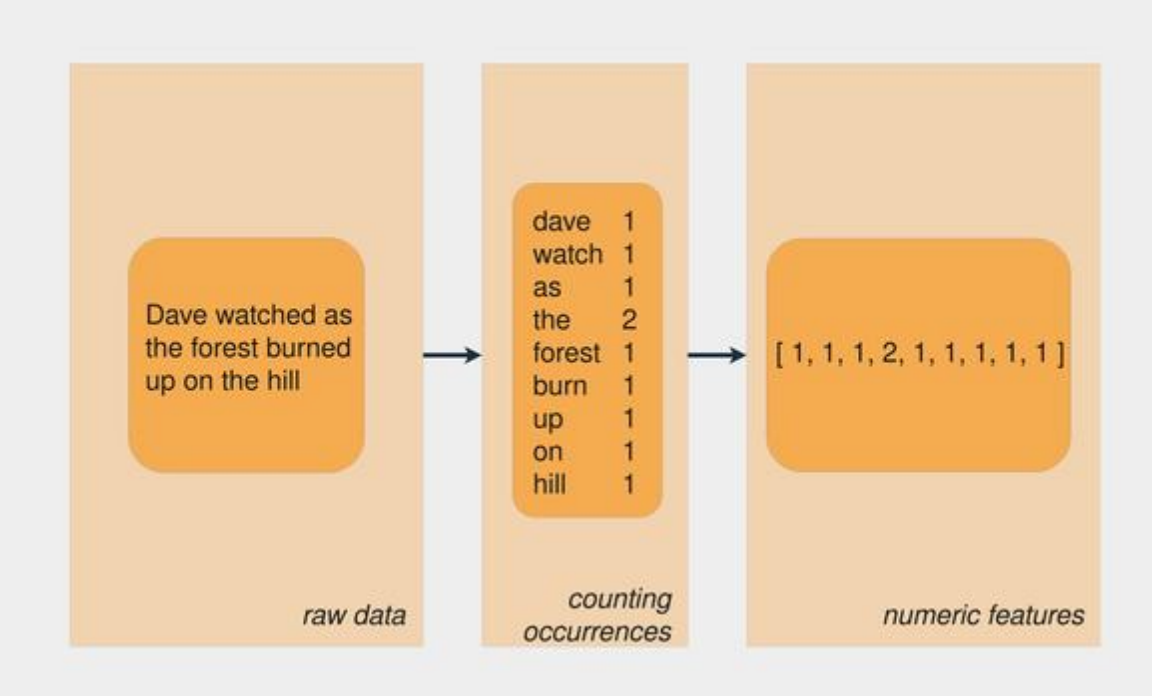
Prediction problems become harder when you use different kinds of data as inputs. The sudoku problem is relatively straightforward because you're dealing directly with numbers. What if you want to train a model to predict the sentiment in a sentence? Or what if you have an image, and you want to know whether it depicts a cat?

Another name for input data is feature, and feature engineering is the process of extracting features from raw data. When dealing with different kinds of data, you need to figure out ways to represent this data in order to extract meaningful information from it.

An example of a feature engineering technique is lemmatization, in which you remove

the inflection from words in a sentence. For example, inflected forms of the verb "watch," like "watches," "watching," and "watched," would be reduced to their lemma, or base form: "watch."

If you're using arrays to store each word of a corpus, then by applying lemmatization, you end up with a less-sparse matrix. This can increase the performance of some machine learning algorithms. The following image presents the process of lemmatization and representation using a bag-of-words model:



Neural Networks: Main Concepts:

A neural network is a system that learns how to make predictions by following these steps:

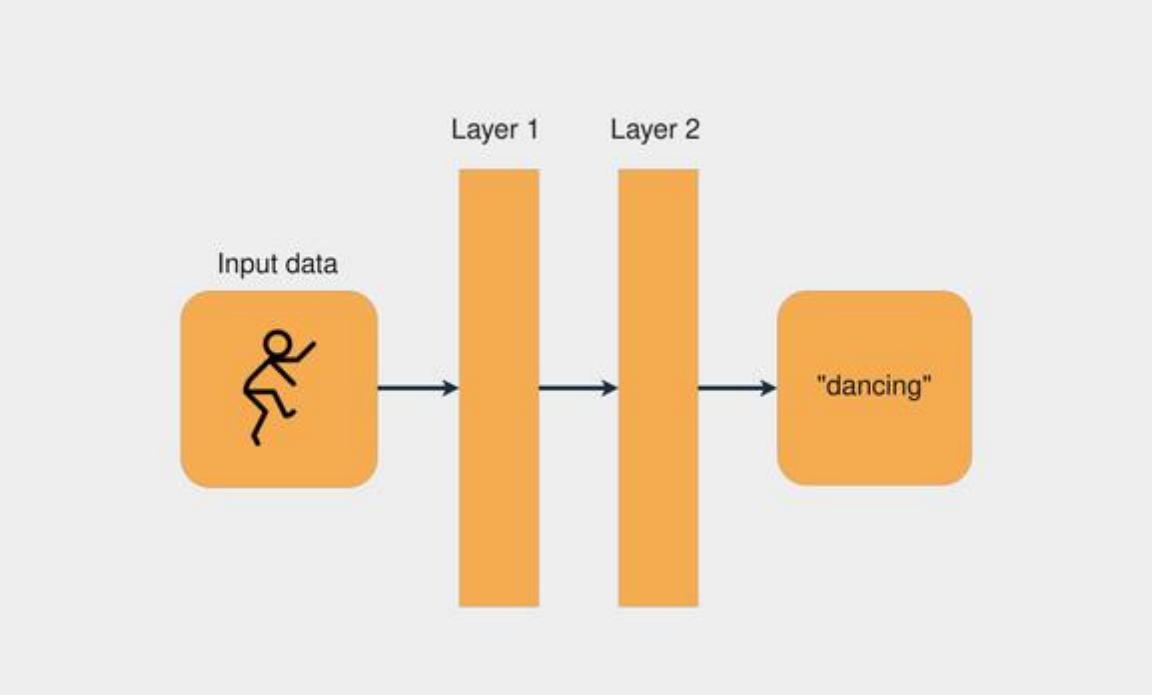
- Taking the input data
- Making a prediction
- Comparing the prediction to the desired output
- Adjusting its internal state to predict correctly the next time

Vectors, layers, and linear regression are some of the building blocks of neural networks.

The data is stored as vectors, and with Python you store these vectors in arrays. Each layer transforms the data that comes from the previous layer. You can think of each layer as a feature engineering step, because each layer extracts some representation of the data that came previously.

One cool thing about neural network layers is that the same computations can extract information from any kind of data. This means that it doesn't matter if you're using image data or text data. The process to extract meaningful information and train the deep learning model is the same for both scenarios.

In the image below, you can see an example of a network architecture with two layers:



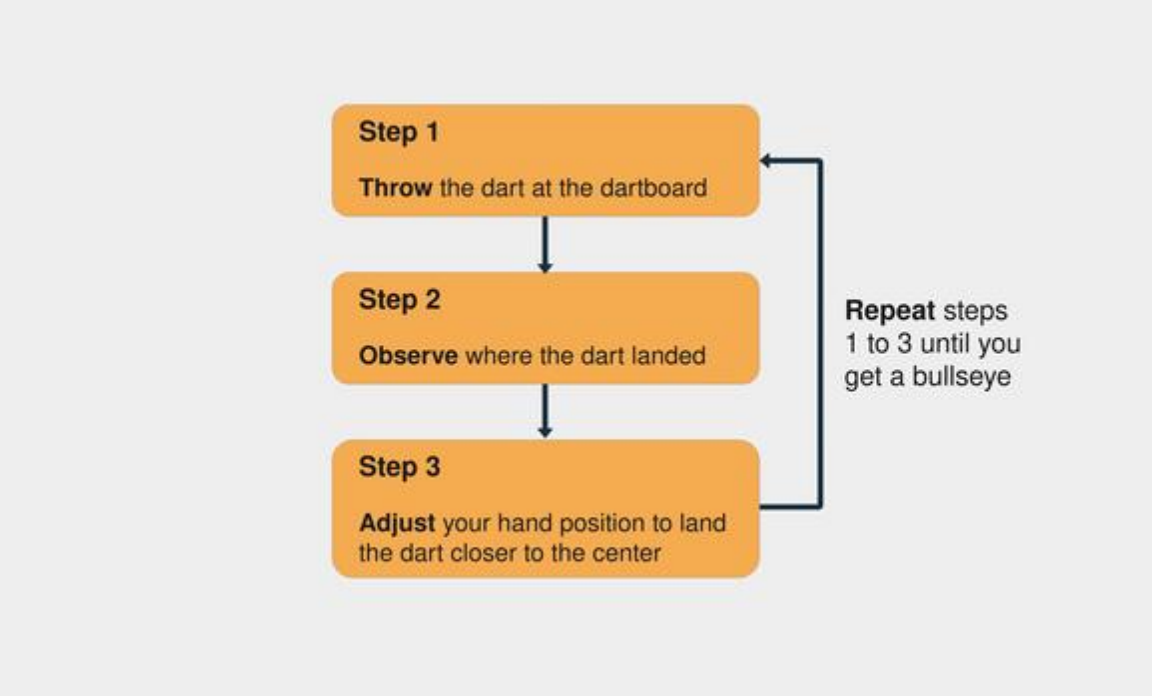
Each layer transforms the data that came from the previous layer by applying some mathematical operations.

The Process to Train a Neural Network:

Training a neural network is similar to the process of trial and error. Imagine you're playing darts for the first time. In your first throw, you try to hit the central point of the dartboard. Usually, the first shot is just to get a sense of how the height and speed of your hand affect the result. If you see the dart is higher than the central point, then you

adjust your hand to throw it a little lower, and so on.

These are the steps for trying to hit the center of a dartboard:



Notice that you keep assessing the error by observing where the dart landed (step 2). You go on until you finally hit the center of the dartboard.

With neural networks, the process is very similar: you start with some random weights and bias vectors, make a prediction, compare it to the desired output, and adjust the vectors to predict more accurately the next time. The process continues until the difference between the prediction and the correct targets is minimal.

Knowing when to stop the training and what accuracy target to set is an important aspect of training neural networks, mainly because of overfitting and underfitting scenarios.

Vectors and Weights:

Working with neural networks consists of doing operations with vectors. You represent the vectors as multidimensional arrays. Vectors are useful in deep learning mainly because of one particular operation: the dot product. The dot product of two vectors tells you how similar they are in terms of direction and is scaled by the magnitude of the two vectors.

The main vectors inside a neural network are the weights and bias vectors. Loosely, what you want your neural network to do is to check if an input is similar to other inputs it's already seen. If the new input is similar to previously seen inputs, then the outputs will also be similar. That's how you get the result of a prediction.

The Linear Regression Model:

Regression is used when you need to estimate the relationship between a dependent variable and two or more independent variables. Linear regression is a method applied when you approximate the relationship between the variables as linear. The method dates back to the nineteenth century and is the most popular regression method.

By modeling the relationship between the variables as linear, you can express the dependent variable as a weighted sum of the independent variables. So, each independent variable will be multiplied by a vector called weight. Besides the weights and the independent variables, you also add another vector: the bias. It sets the result when all the other independent variables are equal to zero.

As a real-world example of how to build a linear regression model, imagine you want to train a model to predict the price of houses based on the area and how old the house is. You decide to model this relationship using linear regression. The following code block shows how you can write a linear regression model for the stated problem in pseudocode:

```
price = (weights_area * area) + (weights_age * age) + bias
```

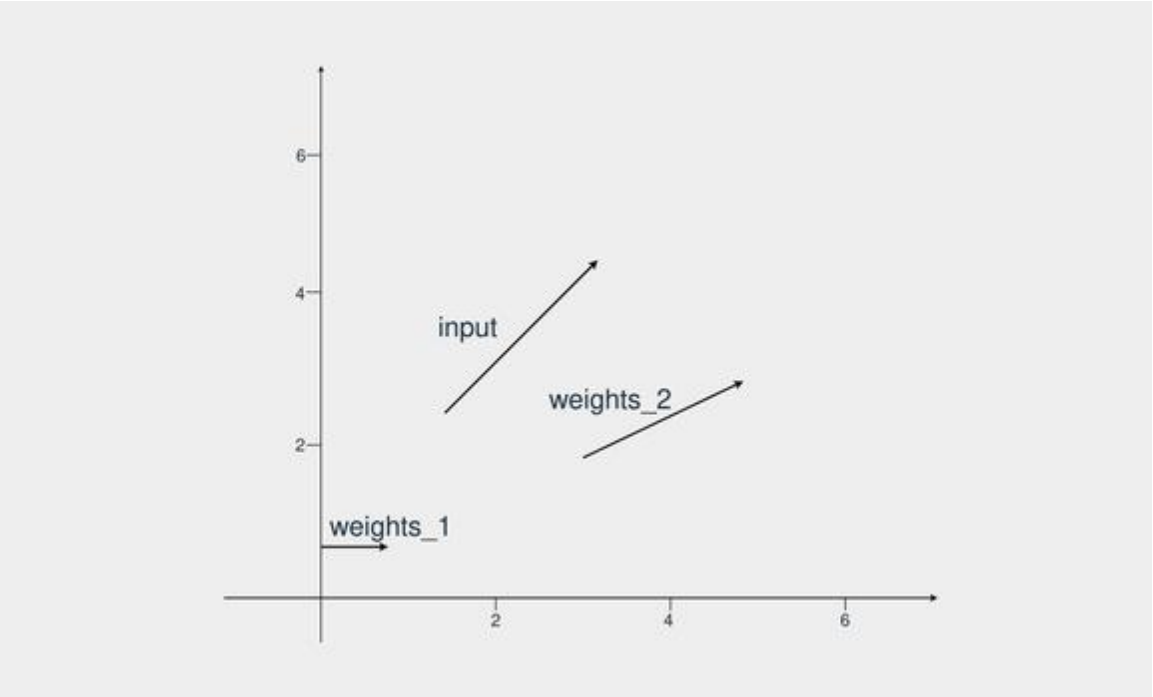
Python AI: Starting to Build Your First Neural Network:

Wrapping the Inputs of the Neural Network With NumPy:

You'll use NumPy to represent the input vectors of the network as arrays. But before you use NumPy, it's a good idea to play with the vectors in pure Python to better understand what's going on.

In this first example, you have an input vector and the other two weight vectors. The goal is to find which of the weights is more similar to the input, taking into account the direction and the

magnitude. This is how the vectors look if you plot them:



First, you define the three vectors, one for the input and the other two for the weights. Then you compute how similar input_vector and weights_1 are. To do that, you'll apply the dot product. Since all the vectors are two-dimensional vectors, these are the steps to do it:

Multiply the first index of input_vector by the first index of weights_1.

Multiply the second index of input_vector by the second index of weights_2.

Sum the results of both multiplications.

$ python -m venv ~/.my-env

$ source ~/.my-env/bin/activate

```
(my-env) $ python -m pip install ipython numpy matplotlib
(my-env) $ ipython
In [1]: input_vector = [1.72, 1.23]
In [2]: weights_1 = [1.26, 0]
In [3]: weights_2 = [2.17, 0.32]

In [4]: # Computing the dot product of input_vector and
weights_1
In [5]: first_indexes_mult = input_vector[0] * weights_1[0]
In [6]: second_indexes_mult = input_vector[1] * weights_1[1]
In [7]: dot_product_1 = first_indexes_mult +
second_indexes_mult

In [8]: print(f"The dot product is: {dot_product_1}")
Out[8]: The dot product is: 2.1672
In [9]: import numpy as np

In [10]: dot_product_1 = np.dot(input_vector, weights_1)

In [11]: print(f"The dot product is: {dot_product_1}")
Out[11]: The dot product is: 2.1672
In [10]: dot_product_2 = np.dot(input_vector, weights_2)

In [11]: print(f"The dot product is: {dot_product_2}")
```

Out[11]: The dot product is: 4.1259

This time, the result is 4.1259. As a different way of thinking about the dot product, you can treat the similarity between the vector coordinates as an on-off switch. If the multiplication result is 0, then you'll say that the coordinates are not similar. If the result is something other than 0, then you'll say that they are similar.

This way, you can view the dot product as a loose measurement of similarity between the vectors. Every time the multiplication result is 0, the final dot product will have a lower result. Getting back to the vectors of the example, since the dot product of input_vector and weights_2 is 4.1259, and 4.1259 is greater than 2.1672, it means that input_vector is more similar to weights_2. You'll use this same mechanism in your neural network.

In this tutorial, you'll train a model to make predictions that have only two possible outcomes. The output result can be either 0 or 1. This is a classification problem, a subset of supervised learning problems in which you have a dataset with the inputs and the known targets. These are the inputs and the outputs of the dataset:

Input Vector	Target
[1.66, 1.56]	1
[2, 1.5]	0

The target is the variable you want to predict. In this example, you're dealing with a dataset that consists of numbers. This isn't common in a real production scenario. Usually, when there's a

need for a deep learning model, the data is presented in files, such as images or text.

Making Your First Prediction:

Since this is your very first neural network, you'll keep things straightforward and build a network with only two layers. So far, you've seen that the only two operations used inside the neural network were the dot product and a sum. Both are linear operations.

If you add more layers but keep using only linear operations, then adding more layers would have no effect because each layer will always have some correlation with the input of the previous layer. This implies that, for a network with multiple layers, there would always be a network with fewer layers that predicts the same results.

What you want is to find an operation that makes the middle layers sometimes correlate with an input and sometimes not correlate.

You can achieve this behavior by using nonlinear functions. These nonlinear functions are called activation functions. There are many types of activation functions. The ReLU (rectified linear unit), for example, is a function that converts all negative numbers to zero. This means that the network can "turn off" a weight if it's negative, adding nonlinearity.

The network you're building will use the sigmoid activation function. You'll use it in the last layer, layer_2. The only two possible outputs in the dataset are 0 and 1, and the sigmoid function limits the output to a range between 0 and 1. This is the formula to express the sigmoid function:

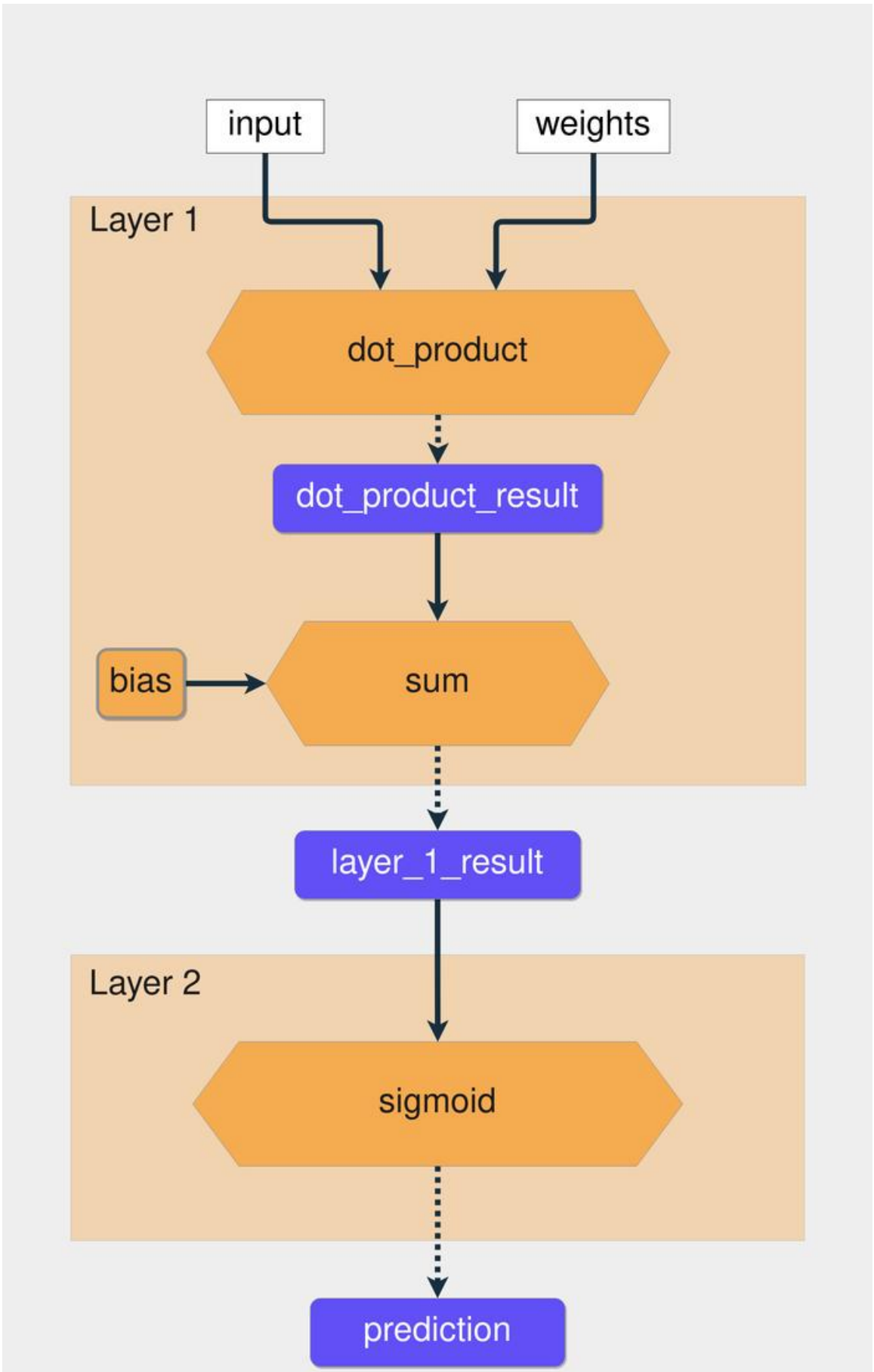
$$S(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid function formula

The e is a mathematical constant called Euler's number, and you can use np.exp(x) to calculate e^x.

Probability functions give you the probability of occurrence for possible outcomes of an event. The only two possible outputs of the dataset are 0 and 1, and the Bernoulli distribution is a distribution that has two possible outcomes as well. The sigmoid function is a good choice if your problem follows the Bernoulli distribution, so that's why you're using it in the last layer of your neural network.

Since the function limits the output to a range of 0 to 1, you'll use it to predict probabilities. If the output is greater than 0.5, then you'll say the prediction is 1. If it's below 0.5, then you'll say the prediction is 0. This is the flow of the computations inside the network you're building:


input
weights
Layer 1
dot_product
dot_product_result
bias
sum
layer_1_result
Layer 2
sigmoid
prediction

The yellow hexagons represent the functions, and the blue rectangles represent the intermediate results. Now it's time to turn all this knowledge into code. You'll also need to wrap the vectors with NumPy arrays. This is the code that applies the functions presented in the image above:

```
In [12]: # Wrapping the vectors in NumPy arrays

In [13]: input_vector = np.array([1.66, 1.56])

In [14]: weights_1 = np.array([1.45, -0.66])

In [15]: bias = np.array([0.0])


In [16]: def sigmoid(x):
   ...:     return 1 / (1 + np.exp(-x))


In [17]: def make_prediction(input_vector, weights, bias):
   ...:     layer_1 = np.dot(input_vector, weights) + bias
   ...:     layer_2 = sigmoid(layer_1)
   ...:     return layer_2


In [18]: prediction = make_prediction(input_vector, weights_1,
bias)


In [19]: print(f"The prediction result is: {prediction}")

Out[19]: The prediction result is: [0.7985731]
```

The raw prediction result is 0.79, which is higher than 0.5, so the output is 1. The network made a correct prediction. Now try it with another input vector, np.array([2, 1.5]). The correct result for this input is 0. You'll only need to change the input_vector variable since all the other parameters remain the same:

In [20]: # Changing the value of input_vector

In [21]: input_vector = np.array([2, 1.5])

In [22]: prediction = make_prediction(input_vector, weights_1, bias)

In [23]: print(f"The prediction result is: {prediction}")

Out[23]: The prediction result is: [0.87101915]

Train Your First Neural Network:

In the process of training the neural network, you first assess the error and then adjust the weights accordingly. To adjust the weights, you'll use the gradient descent and backpropagation algorithms. Gradient descent is applied to find the direction and the rate to update the parameters.

Before making any changes in the network, you need to compute the error. That's what you'll do in the next section.

Computing the Prediction Error:

To understand the magnitude of the error, you need to choose a way to measure it. The function used to measure the error is called the cost function, or loss function. In this tutorial, you'll use the mean squared error (MSE) as your cost function. You compute the MSE in two steps:

Compute the difference between the prediction and the target.

Multiply the result by itself.

The network can make a mistake by outputting a value that's higher or lower than the correct value. Since the MSE is the squared difference between the prediction and the correct result, with this metric you'll always end up with a positive value.

This is the complete expression to compute the error for the last previous prediction:

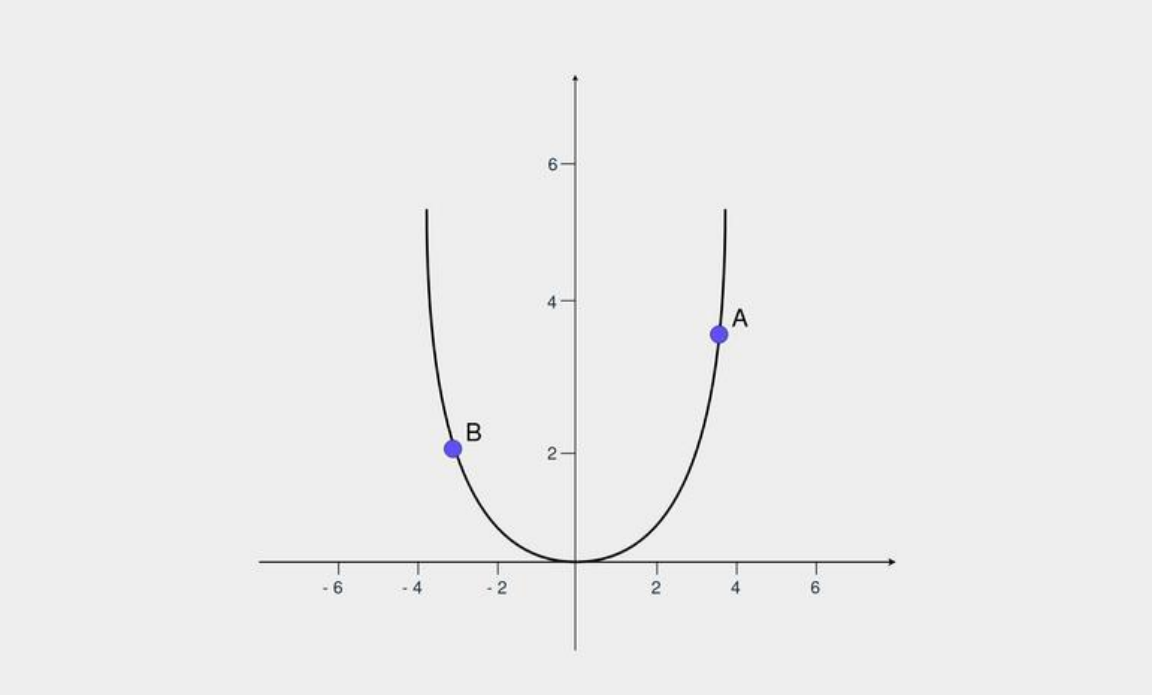
In [24]: target = 0

In [25]: mse = np.square(prediction - target)

In [26]: print(f"Prediction: {prediction}; Error: {mse}")

Out[26]: Prediction: [0.87101915]; Error: [0.7586743596667225]

Understanding How to Reduce the Error:



In [27]: derivative = 2 * (prediction - target)

In [28]: print(f"The derivative is {derivative}")

Out[28]: The derivative is: [1.7420383]

In [29]: # Updating the weights

In [30]: weights_1 = weights_1 - derivative

In [31]: prediction = make_prediction(input_vector, weights_1, bias)

In [32]: error = (prediction - target) ** 2

In [33]: print(f"Prediction: {prediction}; Error: {error}")

Out[33]: Prediction: [0.01496248]; Error: [0.00022388]

Applying the Chain Rule:

In your neural network, you need to update both the weights and the bias vectors. The function you're using to measure the error depends on two independent variables, the weights and the bias. Since the weights and the bias are independent variables, you can change and adjust them to get the result you want.

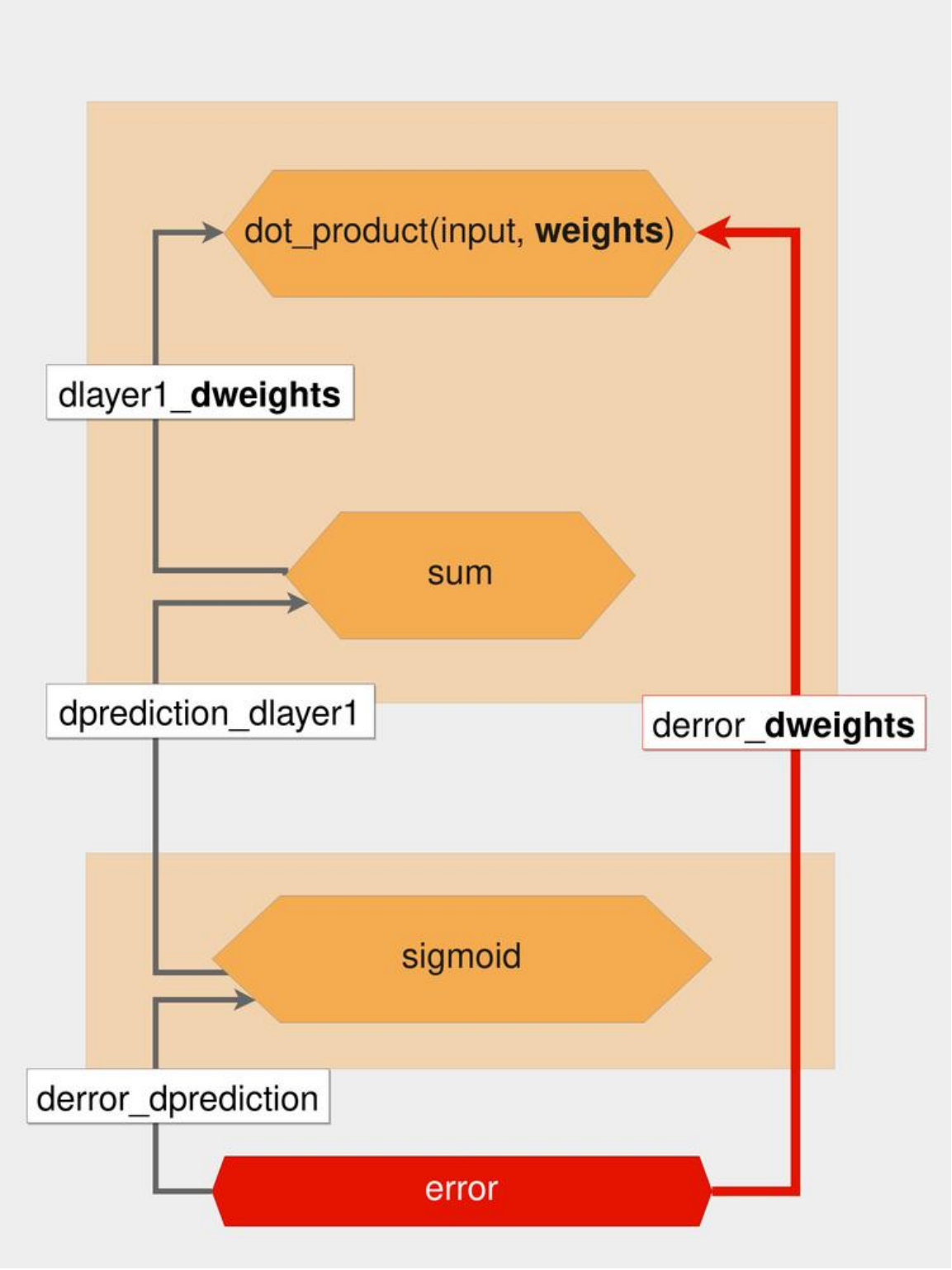
The network you're building has two layers, and since each layer has its own functions, you're dealing with a function composition. This means that the error function is still np.square(x), but now x is the result of another function.

To restate the problem, now you want to know how to change weights_1 and bias to reduce the error. You already saw that you can use derivatives for this, but instead of a function with only a sum inside, now you have a function that produces its result using other functions.

Since now you have this function composition, to take the derivative of the error concerning the parameters, you'll need to use the chain rule from calculus. With the chain rule, you take the partial derivatives of each function, evaluate them, and multiply all the partial derivatives to get the derivative you want.

Now you can start updating the weights. You want to know how to change the weights to decrease the error. This implies that you need to compute the derivative of the error with respect to weights. Since the error is computed by combining different functions, you need to take the partial derivatives of these functions.

Here's a visual representation of how you apply the chain rule to find the derivative of the error with respect to the weights:



derror_dweights = (

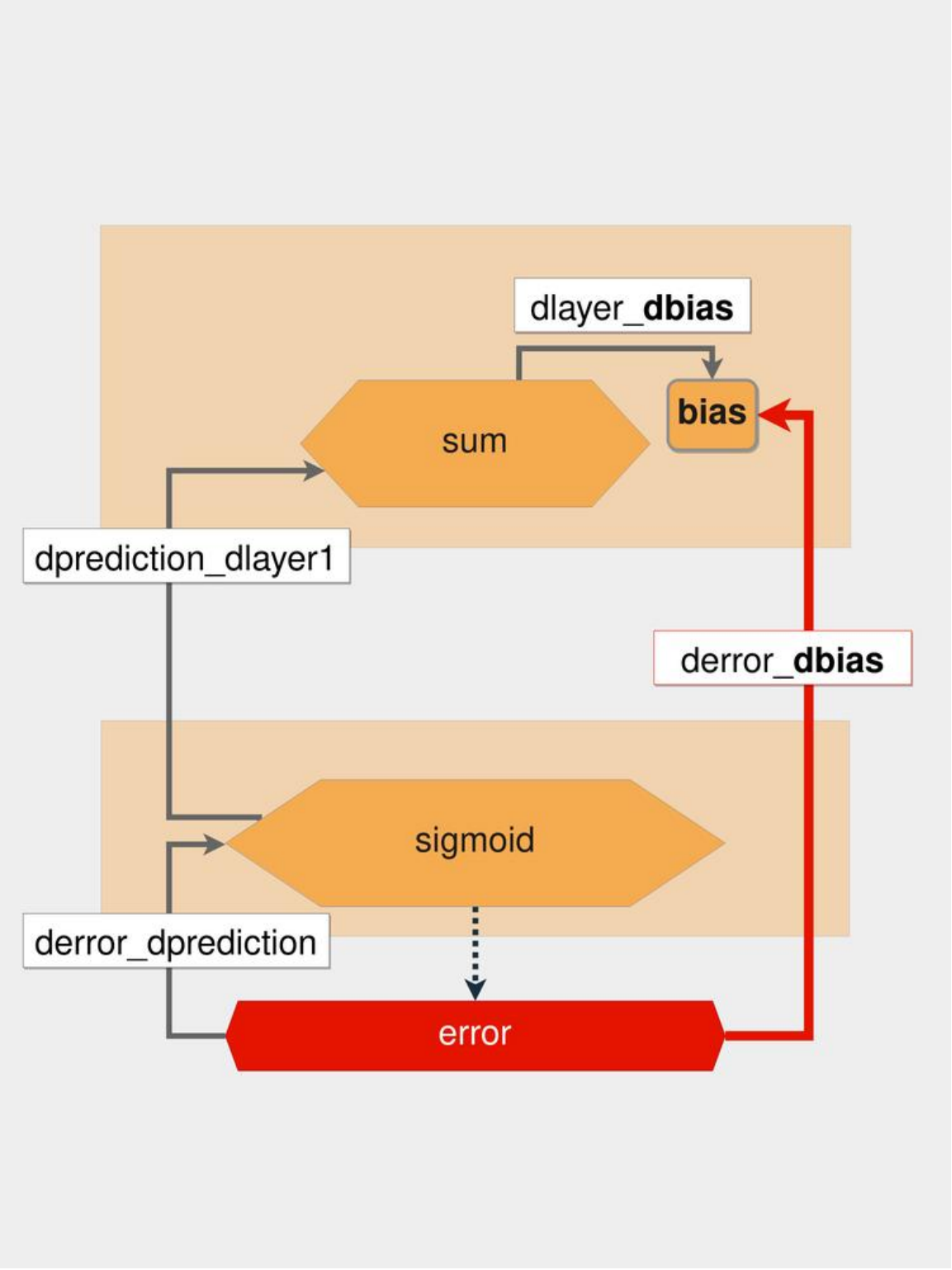
derror_dprediction * dprediction_dlayer1 * dlayer1_dweights

)

Adjusting the Parameters With Backpropagation:

In this section, you'll walk through the backpropagation process step by step, starting with how you update the bias. You want to take the derivative of the error function with respect to the bias, derror_dbias. Then you'll keep going backward, taking the partial derivatives until you find the bias variable.

Since you are starting from the end and going backward, you first need to take the partial derivative of the error with respect to the prediction. That's the derror_dprediction in the image below:



In [36]: def sigmoid_deriv(x):

...: return sigmoid(x) * (1-sigmoid(x))

```
In [37]: derror_dprediction = 2 * (prediction - target)
In [38]: layer_1 = np.dot(input_vector, weights_1) + bias
In [39]: dprediction_dlayer1 = sigmoid_deriv(layer_1)
In [40]: dlayer1_dbias = 1

In [41]: derror_dbias = (
   ...:     derror_dprediction * dprediction_dlayer1 * dlayer1_dbias
   ...: )
```

```
class NeuralNetwork:
    def __init__(self, learning_rate):
        self.weights = np.array([np.random.randn(), np.random.randn()])
        self.bias = np.random.randn()
        self.learning_rate = learning_rate

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def _sigmoid_deriv(self, x):
        return self._sigmoid(x) * (1 - self._sigmoid(x))

    def predict(self, input_vector):
        layer_1 = np.dot(input_vector, self.weights) + self.bias
        layer_2 = self._sigmoid(layer_1)
        prediction = layer_2
```

```
    return prediction

def _compute_gradients(self, input_vector, target):
    layer_1 = np.dot(input_vector, self.weights) + self.bias
    layer_2 = self._sigmoid(layer_1)
    prediction = layer_2

    derror_dprediction = 2 * (prediction - target)
    dprediction_dlayer1 = self._sigmoid_deriv(layer_1)
    dlayer1_dbias = 1
    dlayer1_dweights = (0 * self.weights) + (1 * input_vector)

    derror_dbias = (
        derror_dprediction * dprediction_dlayer1 * dlayer1_dbias
    )
    derror_dweights = (
        derror_dprediction * dprediction_dlayer1 * dlayer1_dweights
    )

    return derror_dbias, derror_dweights

def _update_parameters(self, derror_dbias, derror_dweights):
    self.bias = self.bias - (derror_dbias * self.learning_rate)
    self.weights = self.weights - (
        derror_dweights * self.learning_rate
    )
```

In [42]: learning_rate = 0.1

In [43]: neural_network = NeuralNetwork(learning_rate)

In [44]: neural_network.predict(input_vector)

Out[44]: array([0.79412963])

Training the Network With More Data:

You've already adjusted the weights and the bias for one data instance, but the goal is to make the network generalize over an entire dataset. Stochastic gradient descent is a technique in which, at every iteration, the model makes a prediction based on a randomly selected piece of training data, calculates the error, and updates the parameters.

Now it's time to create the train() method of your NeuralNetwork class. You'll save the error over all data points every 100 iterations because you want to plot a chart showing how this metric changes as the number of iterations increases. This is the final train() method of your neural network:

```
class NeuralNetwork:
  # ...

  def train(self, input_vectors, targets, iterations):
    cumulative_errors = []
    for current_iteration in range(iterations):
      # Pick a data instance at random
      random_data_index = np.random.randint(len(input_vectors))
```

```
            input_vector = input_vectors[random_data_index]
            target = targets[random_data_index]

            # Compute the gradients and update the weights
            derror_dbias, derror_dweights = self._compute_gradients(
                input_vector, target
            )

            self._update_parameters(derror_dbias, derror_dweights)

            # Measure the cumulative error for all the instances
            if current_iteration % 100 == 0:
                cumulative_error = 0
                # Loop through all the instances to measure the error
                for data_instance_index in range(len(input_vectors)):
                    data_point = input_vectors[data_instance_index]
                    target = targets[data_instance_index]

                    prediction = self.predict(data_point)
                    error = np.square(prediction - target)

                    cumulative_error = cumulative_error + error
                cumulative_errors.append(cumulative_error)

        return cumulative_errors
```

There’s a lot going on in the above code block, so here’s a line-by-line breakdown:

Line 8 picks a random instance from the dataset.

Lines 14 to 16 calculate the partial derivatives and return the derivatives for the bias and the weights. They use _compute_gradients(), which you defined earlier.

Line 18 updates the bias and the weights using _update_parameters(), which you defined in the previous code block.

Line 21 checks if the current iteration index is a multiple of 100. You do this to observe how the error changes every 100 iterations.

Line 24 starts the loop that goes through all the data instances.

Line 28 computes the prediction result.

Line 29 computes the error for every instance.

Line 31 is where you accumulate the sum of the errors using the cumulative_error variable. You do this because you want to plot a point with the error for all the data instances. Then, on line 32, you append the error to cumulative_errors, the array that stores the errors. You’ll use this array to plot the graph.

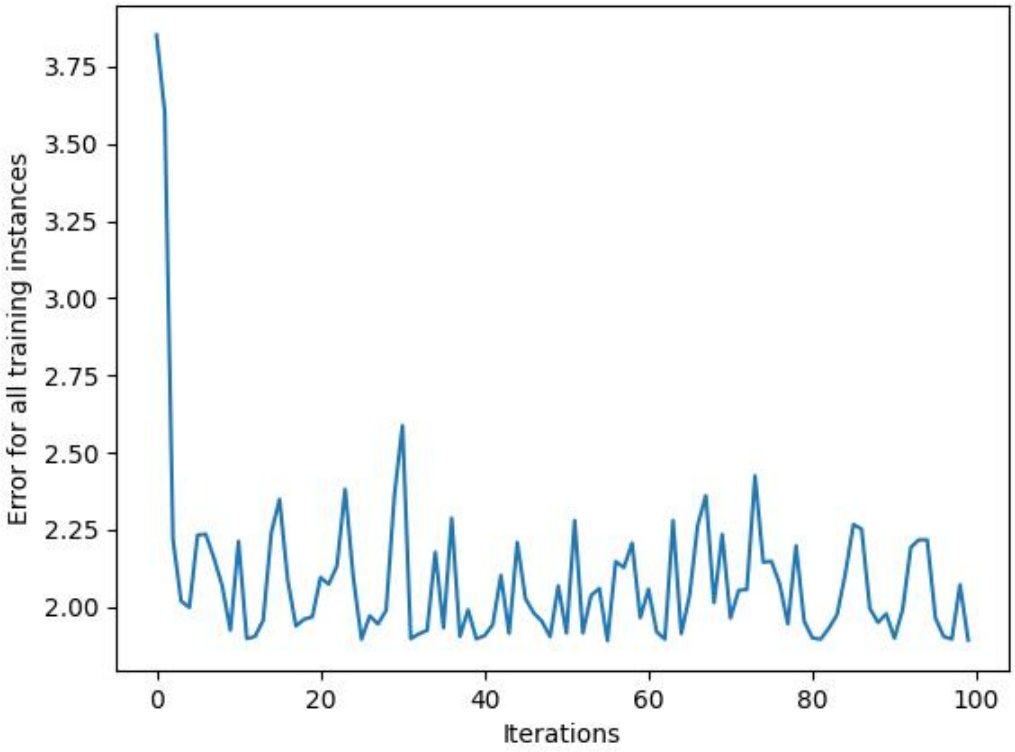
In short, you pick a random instance from the dataset, compute the gradients, and update the weights and the bias. You also compute the cumulative error every 100 iterations and save those results in an array. You’ll plot this array to visualize how the error changes during the training process.

```
In [45]: # Paste the NeuralNetwork class code here

   ...: # (and don't forget to add the train method to the class)
```

```
In [46]: import matplotlib.pyplot as plt

In [47]: input_vectors = np.array(
   ...:     [
   ...:         [3, 1.5],
   ...:         [2, 1],
   ...:         [4, 1.5],
   ...:         [3, 4],
   ...:         [3.5, 0.5],
   ...:         [2, 0.5],
   ...:         [5.5, 1],
   ...:         [1, 1],
   ...:     ]
   ...: )

In [48]: targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])

In [49]: learning_rate = 0.1

In [50]: neural_network = NeuralNetwork(learning_rate)

In [51]: training_error = neural_network.train(input_vectors, targets, 10000)

In [52]: plt.plot(training_error)
In [53]: plt.xlabel("Iterations")
```

In [54]: plt.ylabel("Error for all training instances")

In [54]: plt.savefig("cumulative_error.png")



Adding More Layers to the Neural Network:

The dataset in this tutorial was kept small for learning purposes. Usually, deep learning models need a large amount of data because the datasets are more complex and have a lot of nuances.

Since these datasets have more complex information, using only one or two layers isn't enough. That's why deep learning models are called "deep." They usually have a large number of layers.

By adding more layers and using activation functions, you increase the network's expressive power and can make very high-level predictions. An example of these types of predictions is face recognition, such as when you take a photo of your face with your phone, and the phone unlocks if it recognizes the image as you.

Conclusion:

Congratulations! Today, you built a neural network from scratch using NumPy. With this knowledge, you're ready to dive deeper into the world of artificial intelligence in Python.