```
print(np.exp(100))


>>>
2.6881171418161356e+43


print(np.exp(1000))


>>>
__main__:1: RuntimeWarning: overflow encountered in exp
inf
```

It doesn't take a very large number, in this case, a mere *1,000*, to cause an overflow error. We know the exponential function tends toward 0 as its input value approaches negative infinity, and the output is 1 when the input is 0 (as shown in the chart earlier):

```
import numpy as np

print(np.exp(-np.inf), np.exp(0))


>>>
0.0 1.0
```

We can use this property to prevent the exponential function from overflowing. Suppose we subtract the maximum value from a list of input values. We would then change the output values to always be in a range from some negative value up to 0, as the largest number subtracted by itself returns 0, and any smaller number subtracted by it will result in a negative number — exactly the range discussed above. With Softmax, thanks to the normalization, we can subtract any value from all of the inputs, and it will not change the output:

```
softmax = Activation_Softmax()

softmax.forward([[1, 2, 3]])
print(softmax.output)


>>>
[[0.09003057 0.24472847 0.66524096]]
```

```
softmax.forward([[-2, -1, 0]])  # subtracted 3 - max from the list
print(softmax.output)


>>>
[[0.09003057 0.24472847 0.66524096]]
```

This is another useful property of the exponentiated and normalized function. There's one more thing to mention in addition to these calculations. What happens if we divide the layer's output data, `[1, 2, 3]`, for example, by 2?

```
softmax.forward([[0.5, 1, 1.5]])
print(softmax.output)


>>>
[[0.18632372 0.30719589 0.50648039]]
```

The output confidences have changed due to the nonlinearity nature of the exponentiation. This is one example of why we need to scale all of the input data to a neural network in the same way, which we'll explain in further detail in chapter 22.

Now, we can add another dense layer as the output layer, setting it to contain as many inputs as the previous layer has outputs and as many outputs as our data includes classes. Then we can apply the softmax activation to the output of this new layer:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Make a forward pass of our training data through this layer
dense1.forward(X)
```

```
# Make a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)

# Make a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Make a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)

# Let's see output of the first few samples:
print(activation2.output[:5])


>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
```

As you can see, the distribution of predictions is almost equal, as each of the samples has ~33% (0.33) predictions for each class. This results from the random initialization of weights (a draw from the normal distribution, as not every random initialization will result in this) and zeroed biases. These outputs are also our "confidence scores." To determine which classification the model has chosen to be the prediction, we perform an *argmax* on these outputs, which checks which of the classes in the output distribution has the highest confidence and returns its index - the predicted class index. That said, the confidence score can be as important as the class prediction itself. For example, the argmax of `[0.22, 0.6, 0.18]` is the same as the argmax for `[0.32, 0.36, 0.32]`. In both of these, the argmax function would return an index value of 1 (the 2nd element in Python's zero-indexed paradigm), but obviously, a 60% confidence is much better than a 36% confidence.

# Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities


# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Make a forward pass of our training data through this layer
dense1.forward(X)

# Make a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)

# Make a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Make a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)
```

```
# Let's see output of the first few samples:
print(activation2.output[:5])


>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
```

We've completed what we need for forward-passing data through our model. We used the **Rectified Linear (ReLU**) activation function on the hidden layer, which works on a per-neuron basis. We additionally used the **Softmax** activation function for the output layer since it accepts non-normalized values as input and outputs a probability distribution, which we're using as confidence scores for each class. Recall that, although neurons are interconnected, they each have their respective weights and biases and are not "normalized" with each other.

As you can see, our example model is currently random. To remedy this, we need a way to calculate how wrong the neural network is at current predictions and begin adjusting weights and biases to decrease error over time. Thus, our next step is to quantify how wrong the model is through what's defined as a **loss function**.

**Supplementary Material:** https://nnfs.io/ch4
Chapter code, further resources, and errata for this chapter.

# Chapter 5

# *Calculating Network Error with Loss*

With a randomly-initialized model, or even a model initialized with more sophisticated approaches, our goal is to train, or teach, a model over time. To train a model, we tweak the weights and biases to improve the model's accuracy and confidence. To do this, we calculate how much error the model has. The **loss function**, also referred to as the **cost function**, is the algorithm that quantifies how wrong a model is. **Loss** is the measure of this metric. Since loss is the model's error, we ideally want it to be 0.

You may wonder why we do not calculate the error of a model based on the argmax accuracy. Recall our earlier example of confidence: `[0.22, 0.6, 0.18]` vs `[0.32, 0.36, 0.32]`. If the correct class were indeed the middle one (index 1), the model accuracy would be identical between the two above. But are these two examples *really* as accurate as each other? They are not, because accuracy is simply applying an argmax to the output to find the index of the biggest value. The output of a neural network is actually confidence, and more confidence in

the correct answer is better. Because of this, we strive to increase correct confidence and decrease misplaced confidence.

# Categorical Cross-Entropy Loss

If you're familiar with linear regression, then you already know one of the loss functions used with neural networks that do regression: **squared error** (or **mean squared error** with neural networks).

We're not performing regression in this example; we're classifying, so we need a different loss function. The model has a softmax activation function for the output layer, which means it's outputting a probability distribution. **Categorical cross-entropy** is explicitly used to compare a "ground-truth" probability (*y* or "*targets*") and some predicted distribution (*y-hat* or "*predictions*"), so it makes sense to use cross-entropy here. It is also one of the most commonly used loss functions with a softmax activation on the output layer.

The formula for calculating the categorical cross-entropy of *y* (actual/desired distribution) and *y-hat* (predicted distribution) is:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j})$$

Where $L_i$ denotes sample loss value, *i* is the i-th sample in the set, *j* is the label/output index, *y* denotes the target values, and *y-hat* denotes the predicted values.

Once we start coding the solution, we'll simplify it further to *-log(correct_class_confidence)*, the formula for which is:

$$L_i = -\log\left(\hat{y}_{i,k}\right) \quad where\ \mathbf{k}\ is\ an\ index\ of\ "true"\ probability$$

Where $L_i$ denotes sample loss value, *i* is the i-th sample in a set, *k* is the index of the target label (ground-true label), *y* denotes the target values and *y-hat* denotes the predicted values.

You may ask why we call this cross-entropy and not **log loss**, which is also a type of loss. If you do not know what log loss is, you may wonder why there is such a fancy looking formula for what looks to be a fairly basic description.

In general, the log loss error function is what we apply to the output of a binary logistic regression model (which we'll describe in chapter 16) — there are only two classes in the distribution, each of them applying to a single output (neuron) which is targeted as a 0 or 1. In our case, we have a classification model that returns a probability distribution over all of the outputs. Cross-entropy compares two probability distributions. In our case, we have a softmax output, let's say it's:

```
softmax_output = [0.7, 0.1, 0.2]
```

Which probability distribution do we intend to compare this to? We have 3 class confidences in the above output, and let's assume that the desired prediction is the first class (index 0, which is currently 0.7). If that's the intended prediction, then the desired probability distribution is `[1, 0, 0]`. Cross-entropy can also work on probability distributions like `[0.2, 0.5, 0.3]`; they wouldn't have to look like the one above. That said, the desired probabilities will consist of a 1 in the desired class, and a 0 in the remaining undesired classes. Arrays or vectors like this are called **one-hot**, meaning one of the values is "hot" (on), with a value of 1, and the rest are "cold" (off), with values of 0. When comparing the model's results to a one-hot vector using cross-entropy, the other parts of the equation zero out, and the target probability's log loss is multiplied by 1, making the cross-entropy calculation relatively simple. This is also a special case of the cross-entropy calculation, called categorical cross-entropy. To exemplify this — if we take a softmax output of `[0.7, 0.1, 0.2]` and targets of `[1, 0, 0]`, we can apply the calculations as follows:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j}) = -(1 \cdot log(0.7) + 0 \cdot log(0.1) + 0 \cdot log(0.2)) =$$
$$= -(-0.35667494393873245 + 0 + 0) = 0.35667494393873245$$

Let's see the Python code for this:

```
import math


# An example output from the output layer of the neural network
softmax_output = [0.7, 0.1, 0.2]
# Ground truth
target_output = [1, 0, 0]

loss = -(math.log(softmax_output[0])*target_output[0] +
         math.log(softmax_output[1])*target_output[1] +
         math.log(softmax_output[2])*target_output[2])

print(loss)


>>>
0.35667494393873245
```

That's the full categorical cross-entropy calculation, but we can make a few assumptions given one-hot target vectors. First, what are the values for `target_output[1]` and `target_output[2]` in this case? They're both 0, and anything multiplied by 0 is 0. Thus, we don't need to calculate these indices. Next, what's the value for `target_output[0]` in this case? It's 1. So this can be omitted as any number multiplied by 1 remains the same. The same output then, in this example, can be calculated with:

```
loss = -math.log(softmax_output[0])
```

Which still gives us:

```
>>>
0.35667494393873245
```

As you can see with one-hot vector targets, or scalar values that represent them, we can make some simple assumptions and use a more basic calculation — what was once an involved formula reduces to the negative log of the target class' confidence score — the second formula presented at the beginning of this chapter.

As we've already discussed, the example confidence level might look like `[0.22, 0.6, 0.18]` or `[0.32, 0.36, 0.32]`. In both cases, the *argmax* of these vectors will return the second class as the prediction, but the model's confidence about these predictions is high only for one of them. The **Categorical Cross-Entropy Loss** accounts for that and outputs a larger loss the lower the confidence is:

```
import math

print(math.log(1.))
print(math.log(0.95))
print(math.log(0.9))
print(math.log(0.8))
print('...')
print(math.log(0.2))
print(math.log(0.1))
print(math.log(0.05))
print(math.log(0.01))


>>>
0.0
-0.05129329438755058
-0.10536051565782628
-0.2231435513142097
...
-1.6094379124341003
-2.3025850929940455
-2.995732273553991
-4.605170185988091
```

We've printed different log values for a few example confidences. When the confidence level equals *1*, meaning the model is 100% "sure" about its prediction, the loss value for this sample equals *0*. The loss value raises with the confidence level, approaching 0. You might also wonder why we did not print the result of *log(0)* — we'll explain that shortly.

So far, we've applied log() to the softmax output, but have neither explained what "log" is nor why we use it. We will save the discussion of "why" until the next chapter, which covers derivatives, gradients, and optimizations; suffice it to say that the log function has some desirable properties. **Log** is short for **logarithm** and is defined as the solution for the x-term in an equation of the form $a^x = b$. For example, $10^x = 100$ can be solved with a log: $log_{10}(100)$, which evaluates to 2. This property of the log function is *especially* beneficial when *e* (Euler's number or *~2.71828*) is used in the base (where 10 is in the example). The logarithm with *e* as its base is referred to as the **natural logarithm**, **natural log**, or simply **log** — you may also see this written as **ln**: $ln(x) = log(x) = log_e(x)$ The variety of conventions can make this confusing, so to simplify things, **any mention of log will always be a natural logarithm throughout this book**. The natural log represents the solution for the x-term in the equation $e^x = b$; for example, $e^x = 5.2$ is solved by $log(5.2)$.

In Python code:

```
import numpy as np

b = 5.2
print(np.log(b))


>>>
1.6486586255873816
```

We can confirm this by exponentiating our result:

```
import math

print(math.e ** 1.6486586255873816)


>>>
5.199999999999999
```

The small difference is the result of floating-point precision in Python. Getting back to the loss calculation, we need to modify our output in two additional ways. First, we'll update our process to work on batches of softmax output distributions; and second, make the negative log calculation dynamic to the target index (the target index has been hard-coded so far).

Consider a scenario with a neural network that performs classification between three classes, and the neural network classifies in batches of three. After running through the softmax activation function with a batch of 3 samples and 3 classes, the network's output layer yields:

```
# Probabilities for 3 samples
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
```

We need a way to dynamically calculate the categorical cross-entropy, which we now know is a negative log calculation. To determine which value in the softmax output to calculate the negative log from, we simply need to know our target values. In this example, there are 3 classes; let's say we're trying to classify something as a "dog," "cat," or "human." A dog is class 0 (at index 0), a cat class 1 (index 1), and a human class 2 (index 2). Let's assume the batch of three sample inputs to this neural network is being mapped to the target values of a dog, cat, and cat. So the targets (as

a list of target indices) would be *[0, 1, 1]*.

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]  # dog, cat, cat
```

The first value, 0, in `class_targets` means the first softmax output distribution's intended prediction was the one at the 0th index of `[0.7, 0.1, 0.2]`; the model has a `0.7` confidence score that this observation is a dog. This continues throughout the batch, where the intended target of the 2nd softmax distribution, `[0.1, 0.5, 0.4]`, was at an index of 1; the model only has a `0.5` confidence score that this is a cat — the model is less certain about this observation. In the last sample, it's also the 2nd index from the softmax distribution, a value of `0.9` in this case — a pretty high confidence.

With a collection of softmax outputs and their intended targets, we can map these indices to retrieve the values from the softmax distributions:

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]

for targ_idx, distribution in zip(class_targets, softmax_outputs):
    print(distribution[targ_idx])


>>>
0.7
0.5
0.9
```

The `zip()` function, again, lets us iterate over multiple iterables at the same time in Python. This can be further simplified using NumPy (we're creating a NumPy array of the Softmax outputs this time):

```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = [0, 1, 1]
```

```
print(softmax_outputs[[0, 1, 2], class_targets])
```

```
>>>
[0.7 0.5 0.9]
```

What are the 0, 1, and 2 values? NumPy lets us index an array in multiple ways. One of them is to use a list filled with indices and that's convenient for us — we could use the `class_targets` for this purpose as it already contains the list of indices that we are interested in. The problem is that this has to filter data rows in the array — the second dimension. To perform that, we also need to explicitly filter this array in its first dimension. This dimension contains the predictions and we, of course, want to retain them all. We can achieve that by using a list containing numbers from 0 through all of the indices. We know we're going to have as many indices as distributions in our entire batch, so we can use a `range()` instead of typing each value ourselves:

```
print(softmax_outputs[
    range(len(softmax_outputs)), class_targets
])
```

```
>>>
[0.7 0.5 0.9]
```

This returns a list of the confidences at the target indices for each of the samples. Now we apply the negative log to this list:

```
print(-np.log(softmax_outputs[
    range(len(softmax_outputs)), class_targets
]))
```

```
>>>
[0.35667494 0.69314718 0.10536052]
```

Finally, we want an average loss per batch to have an idea about how our model is doing during training. There are many ways to calculate an average in Python; the most basic form of an average is the **arithmetic mean**: *sum(iterable) / len(iterable)*. NumPy has a method that computes this average on arrays, so we will use that instead. We add NumPy's average to the code:

```
neg_log = -np.log(softmax_outputs[
              range(len(softmax_outputs)), class_targets
          ])
average_loss = np.mean(neg_log)
print(average_loss)


>>>
0.38506088005216804
```

We have already learned that targets can be one-hot encoded, where all values, except for one, are zeros, and the correct label's position is filled with 1. They can also be sparse, which means that the numbers they contain are the correct class numbers — we are generating them this way with the `spiral_data()` function, and we can allow the loss calculation to accept any of these forms. Since we implemented this to work with sparse labels (as in our training data), we have to add a check if they are one-hot encoded and handle it a bit differently in this new case. The check can be performed by counting the dimensions — if targets are single-dimensional (like a list), they are sparse, but if there are 2 dimensions (like a list of lists), then there is a set of one-hot encoded vectors. In this second case, we'll implement a solution using the first equation from this chapter, instead of filtering out the confidences at the target labels. We have to multiply confidences by the targets, zeroing out all values except the ones at correct labels, performing a sum along the row axis (axis *1*). We have to add a test to the code we just wrote for the number of dimensions, move calculations of the log values outside of this new *if* statement, and implement the solution for the one-hot encoded labels following the first equation:

```
import numpy as np

softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = np.array([[1, 0, 0],
                          [0, 1, 0],
                          [0, 1, 0]])


# Probabilities for target values -
# only if categorical labels
if len(class_targets.shape) == 1:
    correct_confidences = softmax_outputs[
        range(len(softmax_outputs)),
        class_targets
    ]
```

```
# Mask values - only for one-hot encoded labels
elif len(class_targets.shape) == 2:
    correct_confidences = np.sum(
        softmax_outputs * class_targets,
        axis=1
    )

# Losses
neg_log = -np.log(correct_confidences)


average_loss = np.mean(neg_log)
print(average_loss)
```

Before we move on, there is one additional problem to solve. The softmax output, which is also an input to this loss function, consists of numbers in the range from 0 to 1 - a list of confidences. It is possible that the model will have full confidence for one label making all the remaining confidences zero. Similarly, it is also possible that the model will assign full confidence to a value that wasn't the target. If we then try to calculate the loss of this confidence of 0:

```
import numpy as np
-np.log(0)


>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

Before we explain this, we need to talk about *log(0)*. From the mathematical point of view, *log(0)* is undefined. We already know the following dependence: if $y=log(x)$, then $e^y=x$. The question of what the resulting $y$ is in $y=log(0)$ is the same as the question of what's the $y$ in $e^y=0$. In simplified terms, the constant $e$ to any power is always a positive number, and there is no $y$ resulting in $e^y=0$. This means the *log(0)* is undefined. We need to be aware of what the *log(0)* is, and "undefined" does not mean that we don't know anything about it. Since *log(0)* is undefined, what's the result for a value very close to *0*? We can calculate the limit of a function. How to exactly calculate it exceeds this book, but the solution is:

$$\lim_{x \to 0^+} log(x) = -\infty$$

We read it as the limit of a natural logarithm of $x$, with x approaching *0* from a positive (it is

impossible to calculate the natural logarithm of a negative value) equals negative infinity. What this means is that the limit is negative infinity for an infinitely small *x*, where *x* never reaches *0*.

The situation is a bit different in programming languages. We do not have limits here, just a function which, given a parameter, returns some value. The negative natural logarithm of *0*, in Python with NumPy, equals an infinitely big number, rather than undefined, and prints a warning about a division by *0* (which is a result of how this calculation is done). If `-np.log(0)` equals `inf`, is it possible to calculate e to the power of negative infinity with Python?

```
np.e**(-np.inf)

>>>
0.0
```

In programming, the fewer things that are undefined, the better. Later on, we'll see similar simplifications, for example when calculating a derivative of the absolute value function, which does not exist for an input of *0* and we'll have to make some decisions to work around this.

Back to the result of `inf` for `-np.log(0)` — as much as that makes sense, since the model would be fully wrong, this will be a problem for us to do further calculations with. Later, with optimization, we will also have a problem calculating gradients, starting with a mean value of all sample-wise losses since a single infinite value in a list will cause the average of that list to also be infinite:

```
import numpy as np
np.mean([1, 2, 3, -np.log(0)])


>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

We could add a very small value to the confidence to prevent it from being a zero, for example, *1e-7*:

```
-np.log(1e-7)


>>>
16.11809565095832
```

Adding a very small value, one-tenth of a million, to the confidence at its far edge will insignificantly impact the result, but this method yields an additional 2 issues. First, in the case where the confidence value is *1*:

```
-np.log(1+1e-7)


>>>
-9.999999505838704e-08
```

When the model is fully correct in a prediction and puts all the confidence in the correct label, loss becomes a negative value instead of being 0. The other problem here is shifting confidence towards *1*, even if by a very small value. To prevent both issues, it's better to clip values from both sides by the same number, *1e-7* in our case. That means that the lowest possible value will become *1e-7* (like in the demonstration we just performed) but the highest possible value, instead of being *1+1e-7*, will become *1-1e-7* (so slightly less than *1*):

```
-np.log(1-1e-7)

>>>
1.0000000494736474e-07
```

This will prevent loss from being exactly *0*, making it a very small value instead, but won't make it a negative value and won't bias overall loss towards *1*. Within our code and using numpy, we'll accomplish that using `np.clip()` method:

```
y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

This method can perform clipping on an array of values, so we can apply it to the predictions directly and save this as a separate array, which we'll use shortly.

# The Categorical Cross-Entropy Loss Class

In the later chapters, we'll be adding more loss functions and some of the operations that we'll be performing are common for all of them. One of these operations is how we calculate the overall loss — no matter which loss function we'll use, the overall loss is always a mean value of all sample losses. Let's create the `Loss` class containing the `calculate` method that will call our loss object's forward method and calculate the mean value of the returned sample losses:

```
# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss
```

In later chapters, we'll add more code to this class, and the reason for it to exist will become more clear. For now, we'll use it for this single purpose.

Let's convert our loss code into a class for convenience down the line:

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

This class inherits the `Loss` class and performs all the error calculations that we derived throughout this chapter and can be used as an object. For example, using the manually-created output and targets:

```
loss_function = Loss_CategoricalCrossentropy()
loss = loss_function.calculate(softmax_outputs, class_targets)
print(loss)
```

```
>>>
0.38506088005216804
```

# Combining everything up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)
```

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities


# Common loss class
class Loss:

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

```
        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]

        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
            )

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods



# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)

# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()

# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)
```

```
# Perform a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)

# Let's see output of the first few samples:
print(activation2.output[:5])

# Perform a forward pass through loss function
# it takes the output of second dense layer here and returns loss
loss = loss_function.calculate(activation2.output, y)

# Print loss value
print('loss:', loss)


>>>
[[0.33333334 0.33333334 0.33333334]
 [0.33333316 0.3333332  0.33333364]
 [0.33333287 0.3333329  0.33333418]
 [0.3333326  0.33333263 0.33333477]
 [0.33333233 0.3333324  0.33333528]]
loss: 1.0986104
```

Again, we get *~0.33* values since the model is random, and its average loss is also not great for these data, as we've not yet trained our model on how to correct its errors.

# Accuracy Calculation

While loss is a useful metric for optimizing a model, the metric commonly used in practice along with loss is the **accuracy**, which describes how often the largest confidence is the correct class in terms of a fraction. Conveniently, we can reuse existing variable definitions to calculate the accuracy metric. We will use the *argmax* values from the *softmax outputs* and then compare these to the targets. This is as simple as doing (note that we slightly modified the `softmax_outputs` for the purpose of this example):

```
import numpy as np

# Probabilities of 3 samples
softmax_outputs = np.array([[0.7, 0.2, 0.1],
                            [0.5, 0.1, 0.4],
                            [0.02, 0.9, 0.08]])
# Target (ground-truth) labels for 3 samples
class_targets = np.array([0, 1, 1])

# Calculate values along second axis (axis of index 1)
predictions = np.argmax(softmax_outputs, axis=1)
# If targets are one-hot encoded - convert them
if len(class_targets.shape) == 2:
    class_targets = np.argmax(class_targets, axis=1)
# True evaluates to 1; False to 0
accuracy = np.mean(predictions==class_targets)


print('acc:', accuracy)


>>>
acc: 0.6666666666666666
```

We are also handling one-hot encoded targets by converting them to sparse values using `np.argmax()`.

We can add the following to the end of our full script above to calculate its accuracy:

```
# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(activation2.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
accuracy = np.mean(predictions==y)

# Print accuracy
print('acc:', accuracy)


>>>
acc: 0.34
```

Now that you've learned how to perform a forward pass through our network and calculate the metrics to signal if the model is performing poorly, we will embark on optimization in the next chapter!

**Supplementary Material:** https://nnfs.io/ch5
Chapter code, further resources, and errata for this chapter.

# Chapter 6

# *Introducing Optimization*

Now that the neural network is built, able to have data passed through it, and capable of calculating loss, the next step is to determine how to adjust the weights and biases to decrease the loss. Finding an intelligent way to adjust the neurons' input's weights and biases to minimize loss is the main difficulty of neural networks.

The first option one might think of is randomly changing the weights, checking the loss, and repeating this until happy with the lowest loss found. To see this in action, we'll use a simpler dataset than we've been working with so far:

```
import matplotlib.pyplot as plt

import nnfs
from nnfs.datasets import vertical_data

nnfs.init()
```

```
X, y = vertical_data(samples=100, classes=3)

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='brg')
plt.show()
```

Which looks like:

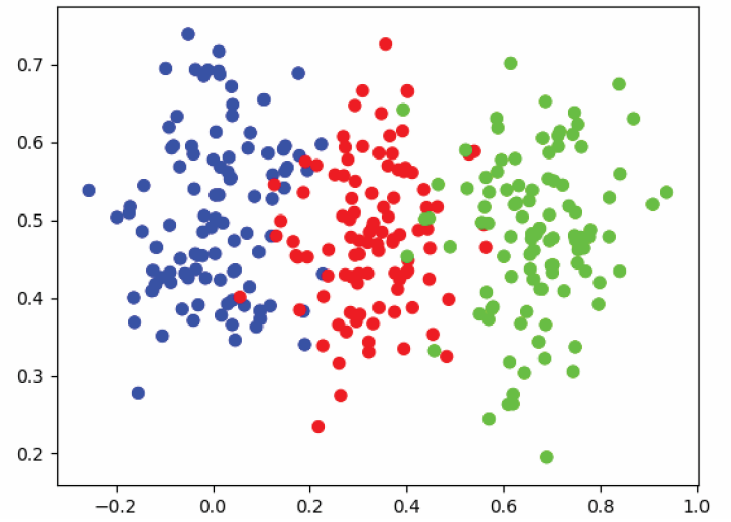


**Fig 6.01:** "Vertical data" graphed.

Using the previously created code up to this point, we can use this new dataset with a simple neural network:

```
# Create dataset
X, y = vertical_data(samples=100, classes=3)

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()
```

Then create some variables to track the best loss and the associated weights and biases:

```
# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()
```

We initialized the loss to a large value and will decrease it when a new, lower, loss is found. We are also copying weights and biases (`copy()` ensures a full copy instead of a reference to the object). Now we iterate as many times as desired, pick random values for weights and biases, and save the weights and biases if they generate the lowest-seen loss:

```
for iteration in range(10000):

    # Generate a new set of weights for iteration
    dense1.weights = 0.05 * np.random.randn(2, 3)
    dense1.biases = 0.05 * np.random.randn(1, 3)
    dense2.weights = 0.05 * np.random.randn(3, 3)
    dense2.biases = 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of the training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through activation function
    # it takes the output of second dense layer here and returns loss
    loss = loss_function.calculate(activation2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions==y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
```

```
>>>
New set of weights found, iteration: 0 loss: 1.0986564 acc:
0.3333333333333333
New set of weights found, iteration: 3 loss: 1.098138 acc:
0.3333333333333333
New set of weights found, iteration: 117 loss: 1.0980115 acc:
0.3333333333333333
New set of weights found, iteration: 124 loss: 1.0977516 acc: 0.6
New set of weights found, iteration: 165 loss: 1.097571 acc:
0.3333333333333333
New set of weights found, iteration: 552 loss: 1.0974693 acc: 0.34
New set of weights found, iteration: 778 loss: 1.0968257 acc:
0.3333333333333333
New set of weights found, iteration: 4307 loss: 1.0965533 acc:
0.3333333333333333
New set of weights found, iteration: 4615 loss: 1.0964499 acc:
0.3333333333333333
New set of weights found, iteration: 9450 loss: 1.0964295 acc:
0.3333333333333333
```

Loss certainly falls, though not by much. Accuracy did not improve, except for a singular situation where the model randomly found a set of weights yielding better accuracy. Still, with a fairly large loss, this state is not stable. Running an additional 90,000 iterations for 100,000 in total:

```
New set of weights found, iteration: 13361 loss: 1.0963014 acc:
0.3333333333333333
New set of weights found, iteration: 14001 loss: 1.0959858 acc:
0.3333333333333333
New set of weights found, iteration: 24598 loss: 1.0947444 acc:
0.3333333333333333
```

Loss continued to drop, but accuracy did not change. This doesn't appear to be a reliable method for minimizing loss. After running for 1 billion iterations, the following was the best (lowest loss) result:

```
New set of weights found, iteration: 229865000 loss: 1.0911305 acc:
0.3333333333333333
```

Even with this basic dataset, we see that randomly searching for weight and bias combinations will take far too long to be an acceptable method. Another idea might be, instead of setting parameters with randomly-chosen values each iteration, apply a fraction of these values to parameters. With this, weights will be updated from what currently yields us the lowest loss instead of aimlessly randomly. If the adjustment decreases loss, we will make it the new point to adjust from. If loss instead increases due to the adjustment, then we will revert to the previous point. Using similar code from earlier, we will first change from randomly selecting weights and biases to randomly *adjusting* them:

```
# Update weights with some small random values
dense1.weights += 0.05 * np.random.randn(2, 3)
dense1.biases += 0.05 * np.random.randn(1, 3)
dense2.weights += 0.05 * np.random.randn(3, 3)
dense2.biases += 0.05 * np.random.randn(1, 3)
```

Then we will change our ending `if` statement to be:

```
# If loss is smaller - print and save weights and biases aside
if loss < lowest_loss:
    print('New set of weights found, iteration:', iteration,
          'loss:', loss, 'acc:', accuracy)
    best_dense1_weights = dense1.weights.copy()
    best_dense1_biases = dense1.biases.copy()
    best_dense2_weights = dense2.weights.copy()
    best_dense2_biases = dense2.biases.copy()
    lowest_loss = loss
# Revert weights and biases
else:
    dense1.weights = best_dense1_weights.copy()
    dense1.biases = best_dense1_biases.copy()
    dense2.weights = best_dense2_weights.copy()
    dense2.biases = best_dense2_biases.copy()
```

# Full code up to this point:

```
# Create dataset
X, y = vertical_data(samples=100, classes=3)

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()

# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()

for iteration in range(10000):

    # Update weights with some small random values
    dense1.weights += 0.05 * np.random.randn(2, 3)
    dense1.biases += 0.05 * np.random.randn(1, 3)
    dense2.weights += 0.05 * np.random.randn(3, 3)
    dense2.biases += 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through activation function
    # it takes the output of second dense layer here and returns loss
    loss = loss_function.calculate(activation2.output, y)
```

```
    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions==y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
    # Revert weights and biases
    else:
        dense1.weights = best_dense1_weights.copy()
        dense1.biases = best_dense1_biases.copy()
        dense2.weights = best_dense2_weights.copy()
        dense2.biases = best_dense2_biases.copy()

>>>
New set of weights found, iteration: 0 loss: 1.0987684 acc:
0.3333333333333333 ...
New set of weights found, iteration: 29 loss: 1.0725244 acc:
0.5266666666666666
New set of weights found, iteration: 30 loss: 1.0724432 acc:
0.3466666666666667 ...
New set of weights found, iteration: 48 loss: 1.0303522 acc:
0.6666666666666666
New set of weights found, iteration: 49 loss: 1.0292586 acc:
0.6666666666666666 ...
New set of weights found, iteration: 97 loss: 0.9277446 acc:
0.7333333333333333 ...
New set of weights found, iteration: 152 loss: 0.73390484 acc:
0.8433333333333334
New set of weights found, iteration: 156 loss: 0.7235515 acc: 0.87
New set of weights found, iteration: 160 loss: 0.7049076 acc:
0.9066666666666666 ...
New set of weights found, iteration: 7446 loss: 0.17280102 acc:
0.9333333333333333
New set of weights found, iteration: 9397 loss: 0.17279711 acc: 0.93
```

Loss descended by a decent amount this time, and accuracy raised significantly. Applying a fraction of random values actually lead to a result that we could almost call a solution. If you try 100,000 iterations, you will not progress much further:

```
>>>
...
New set of weights found, iteration: 14206 loss: 0.1727932 acc:
0.9333333333333333
New set of weights found, iteration: 63704 loss: 0.17278232 acc:
0.9333333333333333
```

Let's try this with the previously-seen spiral dataset instead:

```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

>>>
New set of weights found, iteration: 0 loss: 1.1008677 acc:
0.3333333333333333 ...
New set of weights found, iteration: 31 loss: 1.0982264 acc:
0.37333333333333335 ...
New set of weights found, iteration: 65 loss: 1.0954362 acc:
0.38333333333333336
New set of weights found, iteration: 67 loss: 1.093989 acc:
0.4166666666666667 ...
New set of weights found, iteration: 129 loss: 1.0874122 acc:
0.42333333333333334 ...
New set of weights found, iteration: 5415 loss: 1.0790575 acc: 0.39
```

This training session ended with almost no progress. Loss decreased slightly and accuracy is barely above the initial value. Later, we'll learn that the most probable reason for this is called a local minimum of loss. The data complexity is also not irrelevant here. It turns out hard problems are hard for a reason, and we need to approach this problem more intelligently.

**Supplementary Material:** https://nnfs.io/ch6
Chapter code, further resources, and errata for this chapter.

# Chapter 7

# *Derivatives*

Randomly changing and searching for optimal weights and biases did not prove fruitful for one main reason: the number of possible combinations of weights and biases is infinite, and we need something smarter than pure luck to achieve any success. Each weight and bias may also have different degrees of influence on the loss — this influence depends on the parameters themselves as well as on the current sample, which is an input to the first layer. These input values are then multiplied by the weights, so the input data affects the neuron's output and affects the impact that the weights make on the loss. The same principle applies to the biases and parameters in the next layers, taking the previous layer's outputs as inputs. This means that the impact on the output values depends on the parameters as well as the samples — which is why we are calculating the loss value per each sample separately. Finally, the function of *how* a weight or bias impacts the overall loss is not necessarily linear. In order to know *how* to adjust weights and biases, we first need to understand their impact on the loss.

One concept to note is that we refer to weights and biases and their impact on the loss function. The loss function doesn't contain weights or biases, though. The input to this function is the output of the model, and the weights and biases of the neurons influence this output. Thus, even though we calculate loss from the model's output, not weights/biases, these weights and biases

directly impact the loss.

In the coming chapters, we will describe exactly how this happens by explaining partial derivatives, gradients, gradient descent, and backpropagation. Basically, we'll calculate how much each singular weight and bias changes the loss value (how much of an impact it has on it) given a sample (as each sample produces a separate output, thus also a separate loss value), and how to change this weight or bias for the loss value to decrease. Remember — our goal here is to decrease loss, and we'll do this by using gradient descent. Gradient, on the other hand, is a result of the calculation of the partial derivatives, and we'll backpropagate it using the chain rule to update all of the weights and biases. Don't worry if that doesn't make much sense yet; we'll explain all of these terms and how to perform these actions in this and the coming chapters.

To understand partial derivatives, we need to start with derivatives, which are a special case of partial derivatives — they are calculated from functions taking single parameters.

# The Impact of a Parameter on the Output

Let's start with a simple function and discover what is meant by "impact."
A very simple function $y=2x$, which takes $x$ as an input:

```
def f(x):
    return 2*x
```

Now let's create some code around this to visualize the data — we'll import NumPy and Matplotlib, create an array of 5 input values from 0 to 4, calculate the function output for each of these input values, and plot the result as lines between consecutive points. These points' coordinates are inputs as $x$ and function outputs as $y$:

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return 2*x
```

```
x = np.array(range(5))
y = f(x)


print(x)
print(y)


>>>
[0 1 2 3 4]
[0 2 4 6 8]


plt.plot(x, y)
plt.show()
```

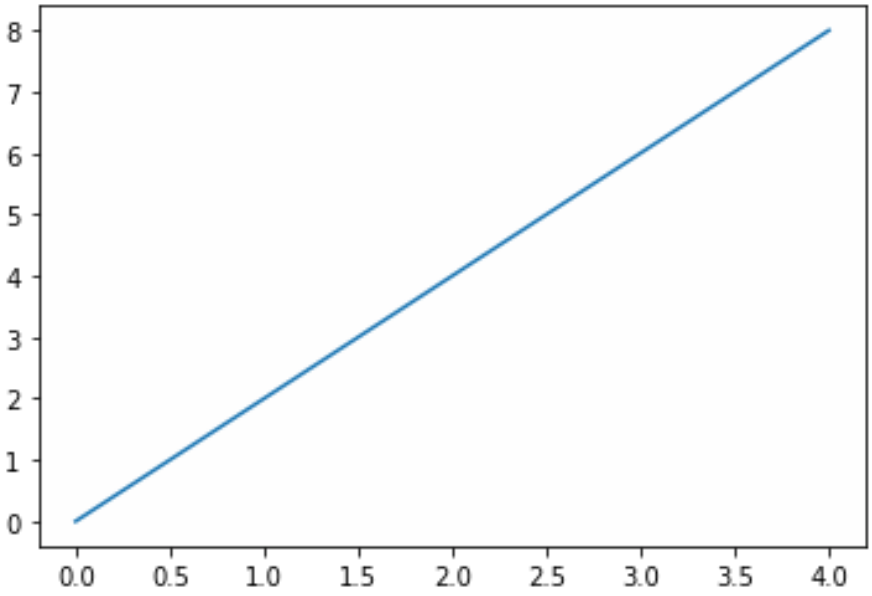


**Fig 7.01:** Linear function y=2x graphed

# The Slope

This looks like an output of the *f(x) = 2x* function, which is a line. How might you define the *impact* that *x* will have on *y*? Some will say, *"y* is double *x*" Another way to describe the *impact* of a linear function such as this comes from algebra: the **slope**. "Rise over run" might be a phrase you recall from school. The slope of a line is:

$$\frac{Change\ in\ y}{Change\ in\ x} = \frac{\Delta y}{\Delta x}$$

It is change in *y* divided by change in *x,* or, in math — *delta y* divided by *delta x*. What's the slope of *f(x) = 2x* then?
To calculate the slope, first we have to take any two points lying on the function's graph and subtract them to calculate the change. Subtracting the points means to subtract their x and y dimensions respectively. Division of the change in y by the change in x returns the slope:

$$\begin{matrix} x & y \\ \downarrow & \downarrow \end{matrix}$$
$$\begin{cases} p_1 = [0, 0] \\ p_2 = [1, 2] \end{cases}$$

$$\Delta x = p_{2x} - p_{1x} = 1 - 0 = 1$$
$$\Delta y = p_{2y} - p_{1y} = 2 - 0 = 2$$

$$slope = \frac{\Delta y}{\Delta x} = \frac{2}{1} = 2$$

Continuing the code, we keep all values of *x* in a single-dimensional NumPy array, `x`, and all results in a single-dimensional array, `y`. To perform the same operation, we'll take `x[0]` and `y[0]` for the first point, then `x[1]` and `y[1]` for the second one. Now we can calculate the slope between them:

```
print((y[1]-y[0]) / (x[1]-x[0]))
```

```
>>>
2.0
```

It is not surprising that the slope of this line is 2. We could say the measure of the impact that *x* has on *y* is 2. We can calculate the slope in the same way for any linear function, including linear functions that aren't as obvious.
What about a nonlinear function like $f(x)=2x^2$?

```
def f(x):
    return 2*x**2
```

This function creates a graph that does not form a straight line:

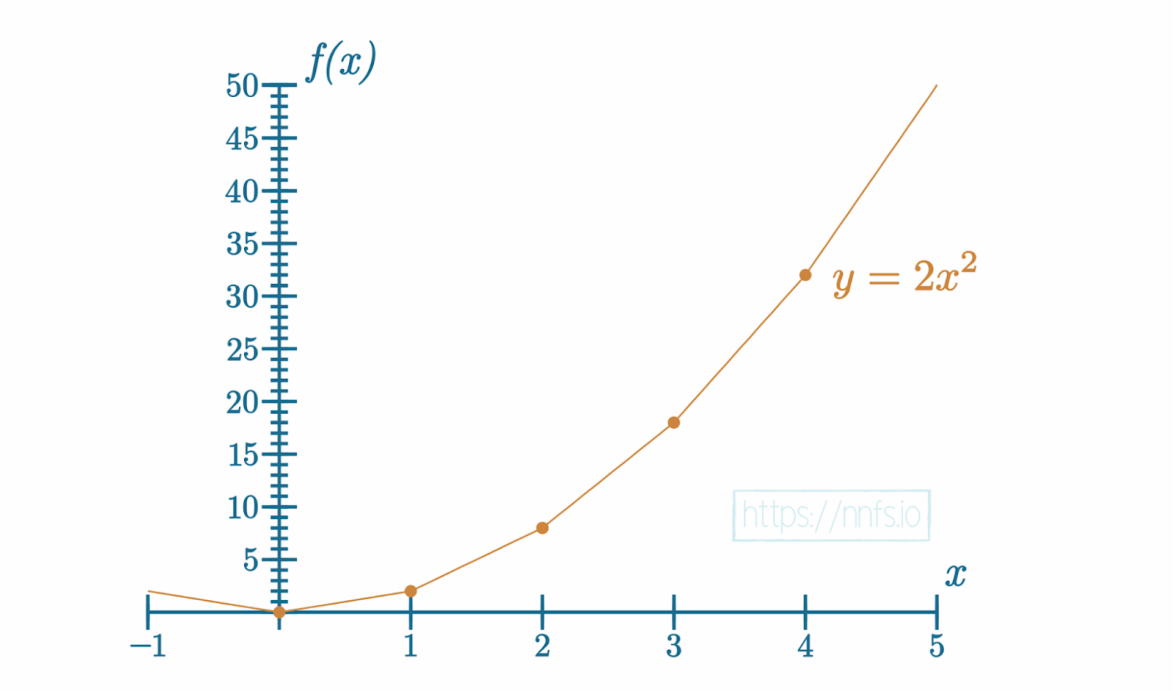


**Fig 7.02:** Approximation of the parabolic function $y=2x^2$ graphed

Can we measure the slope of this curve? Depending on which 2 points we choose to use, we will measure varying slopes:

```
y = f(x)  # Calculate function outputs for new function

print(x)
print(y)


>>>
[0 1 2 3 4]
[ 0  2  8 18 32]
```

Now for the first pair of points:

```
print((y[1]-y[0]) / (x[1]-x[0]))


>>>
2
```

And for another one:

```
print((y[3]-y[2]) / (x[3]-x[2]))

>>>
10
```

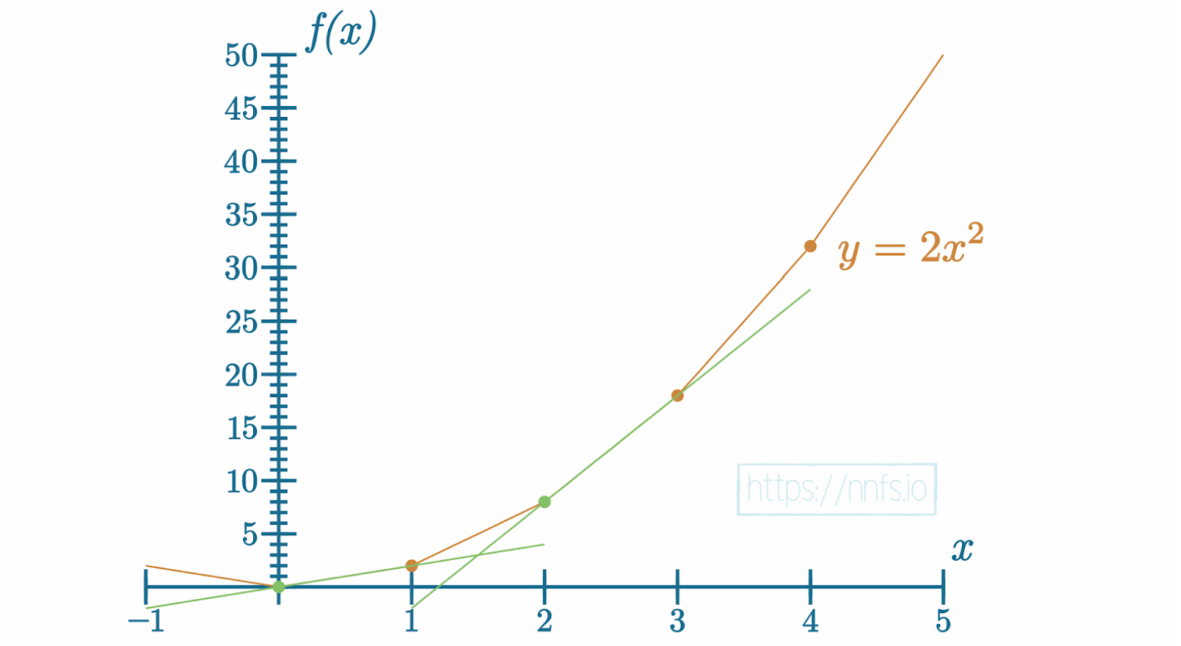


**Fig 7.03:** Approximation of the parabolic function's example tangents

**Anim 7.03:** https://nnfs.io/bro

How might we measure the impact that x has on y in this nonlinear function? Calculus proposes that we measure the slope of the **tangent line** at *x* (for a specific input value to the function), giving us the **instantaneous slope** (slope at this point)**,** which is the **derivative**. The **tangent line** is created by drawing a line between two points that are "infinitely close" on a curve, but this curve has to be differentiable at the derivation point. This means that it has to be continuous and smooth (we cannot calculate the slope at something that we could describe as a "sharp corner," since it contains an infinite number of slopes). Then, because this is a curve, there is no single slope. Slope depends on where we measure it. To give an immediate example, we can approximate a derivative of the function at *x* by using this point and another one also taken at x, but with a very small delta added to it, such as *0.0001*. This number is a common choice as it does not introduce too large an error (when estimating the derivative) or cause the whole expression to be numerically unstable ($\Delta x$ might round to 0 due to floating-point number resolution). This lets us perform the same calculation for the slope as before, but on two points that are very close to each other, resulting in a good approximation of a slope at x:

```
p2_delta = 0.0001

x1 = 1
x2 = x1 + p2_delta  # add delta

y1 = f(x1)  # result at the derivation point
y2 = f(x2)  # result at the other, close point

approximate_derivative = (y2-y1)/(x2-x1)
print(approximate_derivative)


>>>
4.0001999999987845
```

As we will soon learn, the derivative of $2x^2$ at $x=1$ should be exactly *4*. The difference we see (~4.0002) comes from the method used to compute the tangent. We chose a delta small enough to

approximate the derivative as accurately as possible but large enough to prevent a rounding error. To elaborate, an infinitely small delta value will approximate an accurate derivative; however, the delta value needs to be numerically stable, meaning, our delta can not surpass the limitations of Python's floating-point precision (can't be too small as it might be rounded to *0* and, as we know, dividing by *0* is "illegal"). Our solution is, therefore, restricted between estimating the derivative and remaining numerically stable, thus introducing this small but visible error.

# The Numerical Derivative

This method of calculating the derivative is called **numerical differentiation** — calculating the slope of the tangent line using two *infinitely* close points, or as with the code solution — calculating the slope of a tangent line made from two points that were "sufficiently close." We can visualize why we perform this on two close points with the following:

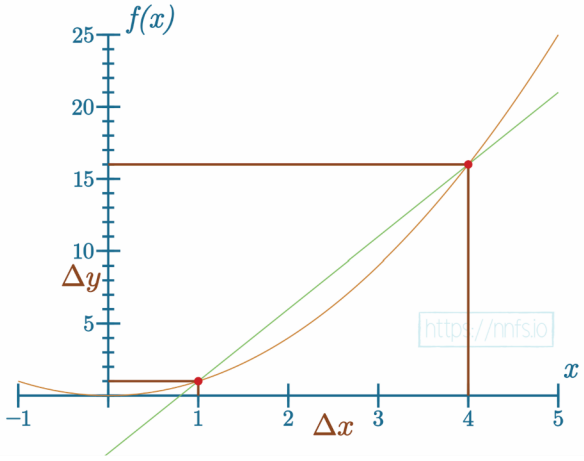


**Fig 7.04:** Why we want to use 2 points that are sufficiently close — large delta inaccuracy.

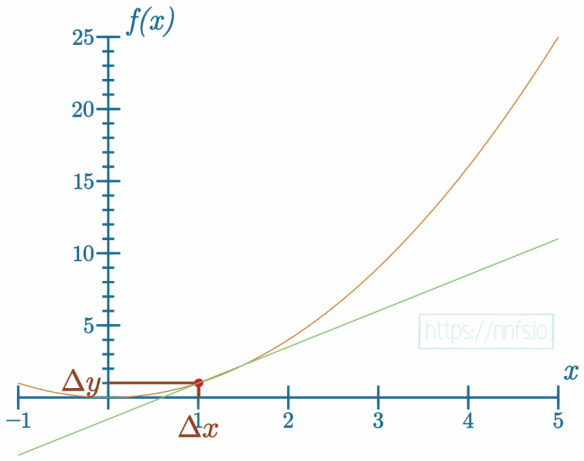


**Fig 7.05:** Why we want to use 2 points that are sufficiently close — very small delta accuracy.

**Anim 7.04-7.05:** https://nnfs.io/cat

We can see that the closer these two points are to each other, the more correct the tangent line appears to be.

Continuing with **numerical differentiation**, let us visualize the tangent lines and how they change depending on where we calculate them. To begin, we'll make the graph of this function more granular using Numpy's `arange()`, allowing us to plot with smaller steps. The `np.arange()` function takes in *start, stop,* and *step* parameters, allowing us to take fractions of a step, such as *0.001* at a time:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us smoother line
x = np.arange(0, 5, 0.001)
y = f(x)
```

```
plt.plot(x, y)

plt.show()
```

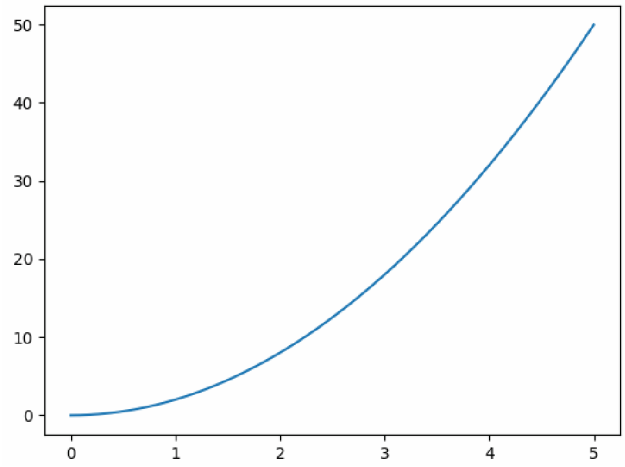


**Fig 7.06:** Matplotlib output that you should see from graphing $y=2x^2$.

To draw these tangent lines, we will derive the function for the tangent line at a point and plot the tangent on the graph at this point. The function for a straight line is $y = mx+b$. Where $m$ is the slope or the *approximate_derivative* that we've already calculated. And $x$ is the input which leaves $b$, or the y-intercept, for us to calculate. The slope remains unchanged, but currently, you can "move" the line up or down using the y-intercept. We already know $x$ and $m$, but $b$ is still unknown. Let's assume $m=1$ for the purpose of the figure and see what exactly it means:

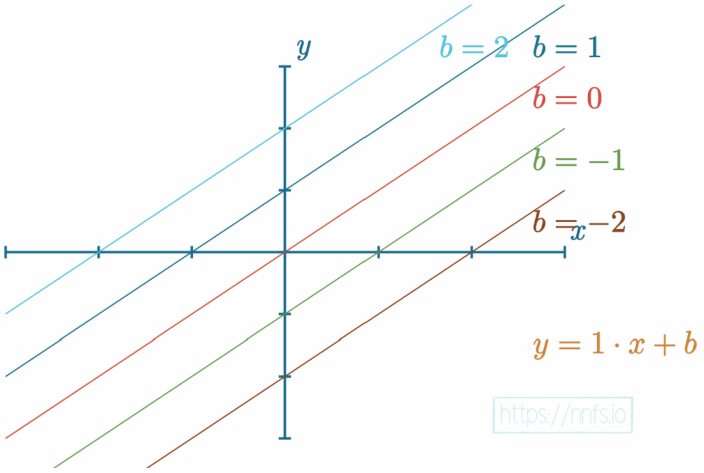


**Fig 7.07:** Various biases graphed where slope = 1.

**Anim 7.07:** https://nnfs.io/but

To calculate *b,* the formula is *b* = *y* - *mx*:

$$y = mx + b$$

$$y - mx = b$$

$$b = y - mx$$

So far we've used two points — the point that we want to calculate the derivative at and the "close enough" to it point to calculate the approximation of the derivative. Now, given the above equation for *b*, the approximation of the derivative and the same "close enough" point (its *x* and *y* coordinates to be specific), we can substitute them in the equation and get the y-intercept for the tangent line at the derivation point. Using code:

```
b = y2 - approximate_derivative*x2
```

Putting everything together:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us smoother line
x = np.arange(0, 5, 0.001)
y = f(x)

plt.plot(x, y)
```

```
# The point and the "close enough" point
p2_delta = 0.0001
x1 = 2
x2 = x1+p2_delta

y1 = f(x1)
y2 = f(x2)

print((x1, y1), (x2, y2))

# Derivative approximation and y-intercept for the tangent line
approximate_derivative = (y2-y1)/(x2-x1)
b = y2 - approximate_derivative*x2


# We put the tangent line calculation into a function so we can call
# it multiple times for different values of x
# approximate_derivative and b are constant for given function
# thus calculated once above this function
def tangent_line(x):
    return approximate_derivative*x + b


# plotting the tangent line
# +/- 0.9 to draw the tangent line on our graph
# then we calculate the y for given x using the tangent line function
# Matplotlib will draw a line for us through these points
to_plot = [x1-0.9, x1, x1+0.9]
plt.plot(to_plot, [tangent_line(i) for i in to_plot])

print('Approximate derivative for f(x)',
      f'where x = {x1} is {approximate_derivative}')

plt.show()


>>>
(2, 8) (2.0001, 8.000800020000002)
Approximate derivative for f(x) where x = 2 is 8.000199999998785
```

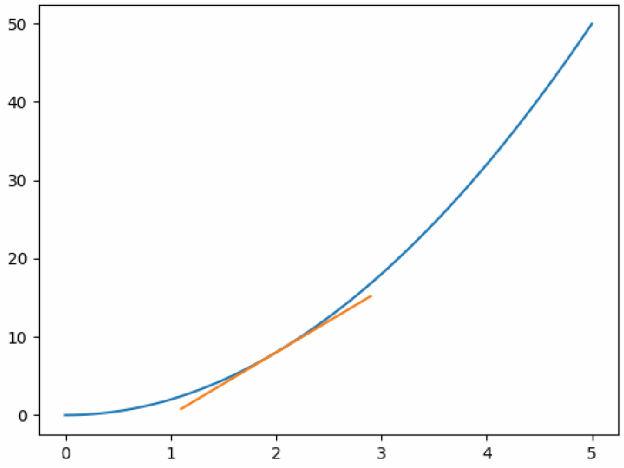


**Fig 7.08:** Graphed approximate derivative for f(x) where x=2

The orange line is the approximate tangent line at $x=2$ for the function $f(x) = 2x^2$. Why do we care about this? You will soon find that we care only about the *slope* of this tangent line but both visualizing and understanding the **tangent line** are very important. We care about the slope of the tangent line because it informs us about the *impact* that $x$ has on this function at a particular point, referred to as the **instantaneous rate of change**. We will use this concept to determine the effect of a specific weight or bias on the overall loss function given a sample. For now, with different values for $x$, we can observe resulting impacts on the function. We can continue the previous code to see the tangent line for various inputs ($x$) - we put a part of the code in a loop over example $x$ values and plot multiple tangent lines:

```
import matplotlib.pyplot as plt
import numpy as np


def f(x):
    return 2*x**2


# np.arange(start, stop, step) to give us a smoother curve
x = np.array(np.arange(0,5,0.001))
y = f(x)

plt.plot(x, y)

colors = ['k','g','r','b','c']


def approximate_tangent_line(x, approximate_derivative):
    return (approximate_derivative*x) + b
```

```
for i in range(5):
    p2_delta = 0.0001
    x1 = i
    x2 = x1+p2_delta

    y1 = f(x1)
    y2 = f(x2)

    print((x1, y1), (x2, y2))
    approximate_derivative = (y2-y1)/(x2-x1)
    b = y2-(approximate_derivative*x2)

    to_plot = [x1-0.9, x1, x1+0.9]

    plt.scatter(x1, y1, c=colors[i])
    plt.plot([point for point in to_plot],
             [approximate_tangent_line(point, approximate_derivative)
                 for point in to_plot],
             c=colors[i])

    print('Approximate derivative for f(x)',
          f'where x = {x1} is {approximate_derivative}')

plt.show()
```

```
>>>
(0, 0) (0.0001, 2e-08)
Approximate derivative for f(x) where x = 0 is 0.00019999999999999998
(1, 2) (1.0001, 2.00040002)
Approximate derivative for f(x) where x = 1 is 4.0001999999987845
(2, 8) (2.0001, 8.000800020000002)
Approximate derivative for f(x) where x = 2 is 8.000199999998785
(3, 18) (3.0001, 18.001200020000002)
Approximate derivative for f(x) where x = 3 is 12.000199999998785
(4, 32) (4.0001, 32.00160002)
Approximate derivative for f(x) where x = 4 is 16.000200000016548
```

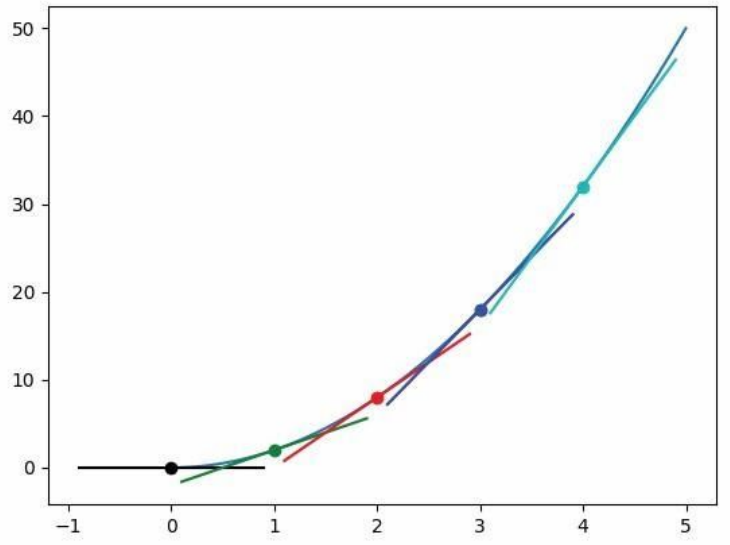


**Fig 7.09:** Derivative calculated at various points.

For this simple function, $f(x) = 2x^2$, we didn't pay a high penalty by approximating the derivative (i.e., the slope of the tangent line) like this, and received a value that was close enough for our needs.

The problem is that the *actual* function employed in our neural network is not so simple. The loss function contains all of the layers, weights, and biases — it's an absolutely massive function operating in multiple dimensions! Calculating derivatives using **numerical differentiation** requires multiple forward passes for a single parameter update (we'll talk about parameter updates in chapter 10). We need to perform the forward pass as a reference, then update a single parameter by the delta value and perform the forward pass through our model again to see the change of the loss value. Next, we need to calculate the **derivative** and revert the parameter change that we made for this calculation. We have to repeat this for every weight and bias and for every sample, which will be very time-consuming. We can also think of this method as brute-forcing the derivative calculations. To reiterate, as we quickly covered many terms, the **derivative** is the **slope** of the **tangent line** for a function that takes a single parameter as an input. We'll use this ability to calculate the slopes of the loss function at each of the weight and bias points — this brings us to the multivariate function, which is a function that takes multiple parameters and is a topic for the next chapter — the partial derivative.

# The Analytical Derivative

Now that we have a better idea of what a derivative *is*, how to calculate the numerical (also called universal) derivative, and why it's not a good approach for us, we can move on to the **Analytical Derivative**, the actual solution to the derivative that we'll implement in our code.

In mathematics, there are two general ways to solve problems: **numerical** and **analytical** methods. Numerical solution methods involve coming up with a number to find a solution, like the above approach with `approximate_derivative`. The numerical solution is also an approximation. On the other hand, the analytical method offers the exact and much quicker, in terms of calculation, solution. However, identifying the analytical solution for the derivative of a given function, as we'll quickly learn, will vary in complexity, whereas the numerical approach never gets more complicated — it's always calling the method twice with two inputs to calculate the approximate derivative at a point. Some analytical solutions are quite obvious, some can be calculated with simple rules, and some complex functions can be broken down into simpler parts and calculated using the so-called **chain rule**. We can leverage already-proven derivative solutions for certain functions, and others — like our loss function — can be solved with combinations of the above.

To compute the derivative of functions using the analytical method, we can split them into simple, elemental functions, finding the derivatives of those and then applying the **chain rule**, which we will explain soon, to get the full derivative. To start building an intuition, let's start with simple functions and their respective derivatives.

The derivative of a simple constant function:

$$f(x) = 1 \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} 1 = 0$$

$$f(x) = 1$$

$$f'(x) = \frac{d}{dx} 1$$

$$f'(x) = 0$$



**Fig 7.10:** Derivative of a constant function — calculation steps.

**Anim 7.10:** https://nnfs.io/cow

When calculating the derivative of a function, recall that the derivative can be interpreted as a slope. In this example, the result of this function is a horizontal line as the output value for any x is 1:

By looking at it, it becomes evident that the derivative equals 0 since there's no change from one value of x to any other value of x (i.e., there's no slope).

So far, we are calculating derivatives of the functions by taking a single parameter, *x* in our case, in each example. This changes with partial derivatives since they take functions with multiple parameters, and we'll be calculating the derivative with respect to only one of them at a time. For now, with derivatives, it's always with respect to a single parameter. To denote the derivative, we can use prime notation, where, for the function *f(x),* we add a prime (') like f'*(x)*. For our example, *f(x) = 1*, the derivative *f'(x) = 0*. Another notation we can use is called the Leibniz's notation — the dependence on the prime notation and multiple ways of writing the derivative with the Leibniz's notation is as follows:

$$f'(x) = \frac{d}{dx}f(x) = \frac{df}{dx}(x) = \frac{df(x)}{dx}$$

Each of these notations has the same meaning — the derivative of a function (with respect to *x*).

In the following examples, we use both notations, since sometimes it's convenient to use one notation or another. We can also use both of them in a single equation.

In summary: the derivative of a constant function equals *0*:

$$f(x) = 1 \quad \rightarrow \quad f'(x) = 0$$

The derivative of a linear function:

$$f(x) = x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}x = \frac{d}{dx}x^1 = 1 \cdot x^{1-1} = 1 \cdot x^0 = 1 \cdot 1 = 1$$

$$f(x) = x$$
$$f'(x) = \frac{d}{dx}x$$
$$f'(x) = \frac{d}{dx}x^1$$
$$f'(x) = x^{1-1}$$
$$f'(x) = 1 \cdot x^0$$
$$f'(x) = 1 \cdot 1$$
$$f'(x) = 1$$



**Fig 7.11:** Derivative of a linear function — calculation steps.

**Anim 7.11:** https://nnfs.io/tob

In this case, the derivative is 1, and the intuition behind this is that for every change of x, y changes by the same amount, so y changes one times the x.

The derivative of the linear function equals *1* (but not in every case, which we'll explain next):

$$f(x) = x \quad \rightarrow \quad f'(x) = 1$$

What if we try 2x, which is also a linear function?

$$f(x) = 2x \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}2x = 2 \cdot \frac{d}{dx}x = 2 \cdot 1x^{1-1} = 2 \cdot 1x^{0} = 2 \cdot 1 = 2$$

$$f(x) = 2x$$
$$f'(x) = \frac{d}{dx}2x$$
$$f'(x) = 2 \cdot \frac{d}{dx}x^{1}$$
$$f'(x) = 2 \cdot x^{1-1}$$
$$f'(x) = 2 \cdot 1x^{0}$$
$$f'(x) = 2 \cdot 1 \cdot 1$$
$$f'(x) = 2$$



**Fig 7.12:** Derivative of another linear function — calculation steps.

**Anim 7.12:** https://nnfs.io/pop

When calculating the derivative, we can take any constant that function is multiplied by and move it outside of the derivative — in this case it's 2 multiplied by the derivative of x. Since we already determined that the derivative of *f(x) = x* was *1*, we now multiply it by *2* to give us the result.

The derivative of a linear function equals the slope, *m* In this case *m = 2*:

$$f(x) = 2x \quad \rightarrow \quad f'(x) = 2$$

If you associate this with numerical differentiation, you're absolutely right — we already concluded that the derivative of a linear function equals its slope:

$$f(x) = mx \quad \rightarrow \quad f'(x) = m$$

*m,* in this case, is a constant, no different than the value *2,* as it's not a parameter — every non-parameter to the function can't change its value; thus, we consider it to be a constant. We have just found a simpler way to calculate the derivative of a linear function and also generalized it for the equations of different slopes, *m*. It's also an exact derivative, not an approximation, as with the numerical differentiation.

What happens when we introduce exponents to the function?

$$f(x) = 3x^2 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}3x^2 = 3 \cdot \frac{d}{dx}x^2 = 3 \cdot 2x^{2-1} = 3 \cdot 2x^1 = 6x$$

$$f(x) = 3x^2$$
$$f'(x) = \frac{d}{dx}3x^2$$
$$f'(x) = 3 \cdot x^{2-1}$$
$$f'(x) = 3 \cdot 2x^1$$
$$f'(x) = 6x$$



**Fig 7.13:** Derivative of quadratic function — calculation steps.

**Anim 7.13:** https://nnfs.io/rok

First, we are applying the rule of a constant — we can move the coefficient (the value that multiplies the other value) outside of the derivative. The rule for handling exponents is as follows: take the exponent, in this case a *2*, and use it as a coefficient for the derived value, then, subtract 1 from the exponent, as seen here: *2 - 1 = 1*.

If $f(x) = 3x^2$ then $f'(x) = 3 \cdot 2x^1$ or simply $6x$. This means the slope of the tangent line, at any point, *x*, for this quadratic function, will be *6x*. As discussed with the numerical solution of the quadratic function differentiation, the derivative of a quadratic function depends on the *x* and in this case it equals *6x*:

$$f(x) = 3x^2 \quad \rightarrow \quad f'(x) = 6x$$

A commonly used operator in functions is addition, how do we calculate the derivative in this case?

$$f(x) = 3x^2 + 5x \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx}[3x^2 + 5x] =$$

$$= \frac{d}{dx} 3x^2 + \frac{d}{dx} 5x^1 =$$

$$= 3 \cdot \frac{d}{dx} x^2 + 5 \cdot \frac{d}{dx} x^1 =$$

$$= 3 \cdot 2x^{2-1} + 5 \cdot 1x^{1-1} =$$

$$= 3 \cdot 2x^1 + 5 \cdot x^0 =$$

$$= 6x + 5$$

$$
\begin{aligned}
f(x) &= 3x^2 + 5x \\
f'(x) &= \frac{d}{dx}[3x^2 + 5x] \\
f'(x) &= \frac{d}{dx}3x^2 + \frac{d}{dx}5x \\
f'(x) &= 3 \cdot x^{2-1} + 5 \cdot x^{1-1} \\
f'(x) &= 3 \cdot 2x^1 + 5 \cdot 1x^0 \\
f'(x) &= 6x + 5
\end{aligned}
$$



**Fig 7.14:** Derivative of quadratic function with addition — calculation steps.

**Anim 7.14:** https://nnfs.io/mob

The derivative of a sum operation is the sum of derivatives, so we can split the derivative of a more complex sum operation into a sum of the derivatives of each term of the equation and solve the rest of the derivative using methods we already know.

The derivative of a sum of functions equals their derivatives:

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) = f'(x) + g'(x)$$

In this case, we've shown the rule using both notations.

Let's try a couple more examples:

$$f(x) = 5x^5 + 4x^3 - 5 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}[5x^5 + 4x^3 - 5] =$$

$$= \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5 =$$

$$= 5 \cdot \frac{d}{dx}x^5 + 4 \cdot \frac{d}{dx}x^3 - \frac{d}{dx}5 =$$

$$= 5 \cdot 5x^{5-1} + 4 \cdot 3x^{3-1} - 0 =$$

$$= 5 \cdot 5x^4 + 4 \cdot 3x^2 =$$

$$= 25x^4 + 12x^2$$

$$f(x) = 5x^5 + 4x^3 - 5$$

$$f'(x) = \frac{d}{dx}[5x^5 + 4x^3 - 5]$$

$$f'(x) = \frac{d}{dx}5x^5 + \frac{d}{dx}4x^3 - \frac{d}{dx}5$$

$$f'(x) = 5 \cdot x^{5-1} + 4 \cdot x^{3-1} - 0$$

$$f'(x) = 5 \cdot 5x^4 + 4 \cdot 3x^2$$

$$f'(x) = 25x^4 + 12x^2$$

https://nnfs.io

**Fig 7.15:** Analytical derivative of multi-dimensional function example — calculation steps.

**Anim 7.15:** https://nnfs.io/tom

The derivative of a constant 5 equals 0, as we already discussed at the beginning of this chapter. We also have to apply the other rules that we've learned so far to perform this calculation.

$$f(x) = x^3 + 2x^2 - 5x + 7 \quad \rightarrow \quad \frac{d}{dx}f(x) = \frac{d}{dx}[x^3 + 2x^2 - 5x + 7] =$$

$$= \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7 =$$

$$= \frac{d}{dx}x^3 + 2 \cdot \frac{d}{dx}x^2 - 5 \cdot \frac{d}{dx}x + \frac{d}{dx}7 =$$

$$= 3x^{3-1} + 2 \cdot 2x^{2-1} - 5 \cdot 1x^{1-1} + 0 =$$

$$= 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1 + 0 =$$

$$= 3x^2 + 4x - 5$$

$$f(x) = x^3 + 2x^2 - 5x + 7$$

$$f'(x) = \frac{d}{dx}[x^3 + 2x^2 - 5x + 7]$$

$$f'(x) = \frac{d}{dx}x^3 + \frac{d}{dx}2x^2 - \frac{d}{dx}5x + \frac{d}{dx}7$$

$$f'(x) = x^{3-1} + 2 \cdot x^{2-1} - 5 \cdot x^{1-1} + 0$$

$$f'(x) = 3x^2 + 2 \cdot 2x^1 - 5 \cdot 1x^0$$

$$f'(x) = 3x^2 + 4x - 5$$



**Fig 7.16:** Analytical derivative of another multi-dimensional function example — calculation steps.

**Anim 7.16:** https://nnfs.io/sun

This looks relatively straight-forward so far, but, with neural networks, we'll work with functions that take multiple parameters as inputs, so we're going to calculate the partial derivatives as well.

# Summary

Let's summarize some of the solutions and rules that we have learned in this chapter.

## Solutions:

The derivative of a constant equals 0 (m is a constant in this case, as it's not a parameter that we are deriving with respect to, which is *x* in this example):

$$\frac{d}{dx}1 = 0$$

$$\frac{d}{dx}m = 0$$

The derivative of *x* equals 1:

$$\frac{d}{dx}x = 1$$

The derivative of a linear function equals its slope:

$$\frac{d}{dx}mx + b = m$$

## Rules:

The derivative of a constant multiple of the function equals the constant multiple of the function's

derivative:

$$\frac{d}{dx}[k \cdot f(x)] = k \cdot \frac{d}{dx}f(x)$$

The derivative of a sum of functions equals the sum of their derivatives:

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) = f'(x) + g'(x)$$

The same concept applies to subtraction:

$$\frac{d}{dx}[f(x) - g(x)] = \frac{d}{dx}f(x) - \frac{d}{dx}g(x) = f'(x) - g'(x)$$

The derivative of an exponentiation:

$$\frac{d}{dx}x^n = n \cdot x^{n-1}$$

We used the value x instead of the whole function f(x) here since the derivative of an entire function is calculated a bit differently. We'll explain this concept along with the chain rule in the next chapter.

Since we've already learned what derivatives are and how to calculate them analytically, which we'll later implement in code, we can go a step further and cover partial derivatives in the next chapter.

**Supplementary Material:** https://nnfs.io/ch7
Chapter code, further resources, and errata for this chapter.

# Chapter 8

# *Gradients, Partial Derivatives, and the Chain Rule*

Two of the last pieces of the puzzle, before we continue coding our neural network, are the related concepts of **gradients** and **partial derivatives**. The derivatives that we've solved so far have been cases where there is only one independent variable in the function — that is, the result depended solely on, in our case, *x*. However, our neural network consists, for example, of neurons, which have multiple inputs. Each input gets multiplied by the corresponding weight (a function of 2 parameters), and they get summed with the bias (a function of as many parameters as there are inputs, plus one for a bias). As we'll explain soon in detail, to learn the impact of all of the inputs, weights, and biases to the neuron output and at the end of the loss function, we need to calculate the derivative of each operation performed during the forward pass in the neuron and the whole model. To do that and get answers, we'll need to use the **chain rule**, which we'll explain soon in this chapter.

# The Partial Derivative

The **partial derivative** measures how much impact a single input has on a function's output. The method for calculating a partial derivative is the same as for derivatives explained in the previous chapter; we simply have to repeat this process for each of the independent inputs.

Each of the function's inputs has some impact on this function's output, even if the impact is 0. We need to know these impacts; this means that we have to calculate the derivative with respect to each input separately to learn about each of them. That's why we call these partial derivatives with respect to given input — we are calculating a partial of the derivative, related to a singular input. The partial derivative is a single equation, and the full multivariate function's derivative consists of a set of equations called the **gradient**. In other words, the **gradient** is a vector of the size of inputs containing partial derivative solutions with respect to each of the inputs. We'll get back to gradients shortly.

To denote the partial derivative, we'll be using Euler's notation. It's very similar to Leibniz's notation, as we only need to replace the differential operator *d* with *∂*. While the *d* operator might be used to denote the differentiation of a multivariate function, its meaning is a bit different — it can mean the rate of the function's change in relation to the given input, but when other inputs might change as well, and it is used mostly in physics. We are interested in the partial derivatives, a situation where we try to find the impact of the given input to the output while treating all of the other inputs as constants. We are interested in the impact of singular inputs since our goal, in the model, is to update parameters. The *∂* operator means explicitly that — the partial derivative:

$$f(x,y,z) \quad \rightarrow \quad \frac{\partial}{\partial x}f(x,y,z), \frac{\partial}{\partial y}f(x,y,z), \frac{\partial}{\partial z}f(x,y,z)$$

# The Partial Derivative of a Sum

Calculating the partial derivative with respect to a given input means to calculate it like the regular derivative of one input, just while treating other inputs as constants. For example:

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[x+y] = \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}y = 1 + 0 = 1$$
$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[x+y] = \frac{\partial}{\partial y}x + \frac{\partial}{\partial y}y = 0 + 1 = 1$$

First, we applied the sum rule — the derivative of a sum is the sum of derivatives. Then, we already know that the derivative of *x* with respect to *x* equals *1*. The new thing is the derivative of *y* with respect to *x*. As we mentioned, *y* is treated as a constant, as it does not change when we are deriving with respect to *x*, and the derivative of a constant equals *0*. In the second case, we derived with respect to *y*, thus treating *x* as constant. Put another way, regardless of the value of y in this example, the slope of *x* does not depend on *y*. This will not always be the case, though, as we will soon see.

Let's try another example:

$$f(x,y) = 2x + 3y^2 \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[2x + 3y^2] = \frac{\partial}{\partial x}2x + \frac{\partial}{\partial x}3y^2 =$$
$$= 2 \cdot \frac{\partial}{\partial x}x + 3 \cdot \frac{\partial}{\partial x}y^2 = 2 \cdot 1 + 3 \cdot 0 = 2$$
$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[2x + 3y^2] = \frac{\partial}{\partial y}2x + \frac{\partial}{\partial y}3y^2 =$$
$$= 2 \cdot \frac{\partial}{\partial y}x + 3 \cdot \frac{\partial}{\partial y}y^2 = 2 \cdot 0 + 3 \cdot 2y^1 = 6y$$

In this example, we also applied the sum rule first, then moved constants to the outside of the derivatives and calculated what remained with respect to *x* and *y* individually. The only difference to the non-multivariate derivatives from the previous chapter is the "partial" part, which means

we are deriving with respect to each of the variables separately. Other than that, there is nothing new here.

Let's try something seemingly more complicated:

$$f(x,y) = 3x^3 - y^2 + 5x + 2 \quad \rightarrow$$

$$\frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[3x^3 - y^2 + 5x + 2] = \frac{\partial}{\partial x}3x^3 - \frac{\partial}{\partial x}y^2 + \frac{\partial}{\partial x}5x + \frac{\partial}{\partial x}2 =$$

$$= 3 \cdot \frac{\partial}{\partial x}x^3 - \frac{\partial}{\partial x}y^2 + 5 \cdot \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}2 = 3 \cdot 3x^2 - 0 + 5 \cdot 1 + 0 = 9x^2 + 5$$

$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[3x^3 - y^2 + 5x + 2] = \frac{\partial}{\partial y}3x^3 - \frac{\partial}{\partial y}y^2 + \frac{\partial}{\partial y}5x + \frac{\partial}{\partial y}2 =$$

$$= 3 \cdot \frac{\partial}{\partial y}x^3 - \frac{\partial}{\partial y}y^2 + 5 \cdot \frac{\partial}{\partial y}x + \frac{\partial}{\partial y}2 = 3 \cdot 0 - 2y^1 + 5 \cdot 0 + 0 = -2y$$

Pretty straight-forward — we're constantly applying the same rules over and over again, and we did not add any new calculation or rules in this example.

# The Partial Derivative of Multiplication

Before we move on, let's introduce the partial derivative of multiplication operation:

$$f(x,y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = \frac{\partial}{\partial x}[x \cdot y] = y\frac{\partial}{\partial x}x = y \cdot 1 = y$$
$$\frac{\partial}{\partial y} f(x,y) = \frac{\partial}{\partial y}[x \cdot y] = x\frac{\partial}{\partial y}y = x \cdot 1 = x$$

We have already mentioned that we need to treat the other independent variables as constants, and we also have learned that we can move constants to the outside of the derivative. That's exactly how we solve the calculation of the partial derivative of multiplication — we treat other variables as constants, like numbers, and move them outside of the derivative. It turns out that when we derive with respect to *x*, *y* is treated as a constant, and the result equals *y* multiplied by the derivative of *x* with respect to *x,* which is *1*. The whole derivative then results with *y*. The intuition behind this example is when calculating the partial derivative with respect to *x*, every change of *x* by *1* changes the function's output by *y*. For example, if *y=3* and *x=1*, the result is *1·3=3*. When we change *x* by *1* so *y=3* and *x=2*, the result is *2·3=6*. We changed *x* by *1* and the result changed by *3*, by the *y*. That's what the partial derivative of this function with respect to *x* tells us.

Let's introduce a third input variable and add multiplication of variables for another example:

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$
$$\frac{\partial}{\partial x} f(x,y,z) = \frac{\partial}{\partial x}[3x^3z - y^2 + 5z + 2yz] =$$
$$= \frac{\partial}{\partial x}3x^3z - \frac{\partial}{\partial x}y^2 + \frac{\partial}{\partial x}5z + \frac{\partial}{\partial x}2yz =$$
$$= 3z \cdot \frac{\partial}{\partial x}x^3 - \frac{\partial}{\partial x}y^2 + 5 \cdot \frac{\partial}{\partial x}z + 2 \cdot \frac{\partial}{\partial x}yz =$$
$$= 3z \cdot 3x^2 - 0 + 5 \cdot 0 + 2 \cdot 0 = 9x^2z$$

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial y} f(x,y,z) = \frac{\partial}{\partial y}[3x^3z - y^2 + 5z + 2yz] =$$

$$= \frac{\partial}{\partial y} 3x^3z - \frac{\partial}{\partial y} y^2 + \frac{\partial}{\partial y} 5z + \frac{\partial}{\partial y} 2yz =$$

$$= 3 \cdot \frac{\partial}{\partial y} x^3z - \frac{\partial}{\partial y} y^2 + 5 \cdot \frac{\partial}{\partial y} z + 2z \cdot \frac{\partial}{\partial y} y =$$

$$= 3 \cdot 0 - 2y + 5 \cdot 0 + 2z \cdot 1 = -2y + 2z$$

$$f(x,y,z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial z} f(x,y,z) = \frac{\partial}{\partial z}[3x^3z - y^2 + 5z + 2yz] =$$

$$= \frac{\partial}{\partial z} 3x^3z - \frac{\partial}{\partial z} y^2 + \frac{\partial}{\partial z} 5z + \frac{\partial}{\partial z} 2yz =$$

$$= 3x^3 \cdot \frac{\partial}{\partial z} z - \frac{\partial}{\partial z} y^2 + 5 \cdot \frac{\partial}{\partial z} z + 2y \cdot \frac{\partial}{\partial z} z =$$

$$= 3x^3 \cdot 1 - 0 + 5 \cdot 1 + 2y \cdot 1 = 3x^3 + 5 + 2y$$

The only new operation here is, as mentioned, moving variables other than the one that we derive with respect to, outside of the derivative. The results in this example appear more complicated, but only because of the existence of other variables in them — variables that are treated as constants during derivation. Equations of the derivatives are longer, but not necessarily more complicated.

The reason to learn about partial derivatives is we'll be calculating the partial derivatives of multivariate functions soon, an example of which is the neuron. From the code perspective and the *Dense* layer class, more specifically, the *forward* method of this class, we're passing in a single variable — the input array, containing either a batch of samples or outputs from the previous layer. From the math perspective, each value of this single variable (an array) is a separate input — it contains as many inputs as we have data in the input array. For example, if we pass a vector of *4* values to the neuron, it's a singular variable in the code, but *4* separate inputs in the equation. This forms a function that takes multiple inputs. To learn about the impact that each input makes to the function's output, we'll need to calculate the partial derivative of this function with respect to each of its inputs, which we'll explain in detail in the next chapter.

# The Partial Derivative of *Max*

Derivatives and partial derivatives are not limited to addition and multiplication operations, or constants. We need to derive them for the other functions that we used in the forward pass, one of which is the derivative of the *max()* function:

$$f(x, y) = max(x, y) \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = \frac{\partial}{\partial x} max(x, y) = 1(x > y)$$

The max function returns the greatest input. We know that the derivative of *x* with respect to x equals *1,* so the derivative of this function with respect to *x* equals 1 if *x* is greater than *y*, since the function will return *x*. In the other case, where *y* is greater than *x* and will get returned instead, the derivative of *max()* with respect to *x* equals 0 — we treat *y* as a constant, and the derivative of *y* with respect to *x* equals 0. We can denote that as *1(x > y)*, which means *1* if the condition is met, and *0* otherwise. We could also calculate the partial derivative of *max()* with respect to *y*, but we won't need it anywhere in this book.

One special case for the derivative of the *max()* function is when we have only one variable parameter, and the other parameter is always constant at *0*. This means that we want whichever is bigger in return — *0* or the input value, effectively clipping the input value at *0* from the positive side. Handling this is going to be useful when we calculate the derivative of the **ReLU** activation function since that activation function is defined as *max(x, 0)*:

$$f(x) = max(x, 0) \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} max(x, 0) = 1(x > 0)$$

Notice that since this function takes a single parameter, we used the *d* operator instead of the ∂ to calculate the non-partial derivative. In this case, the derivative is *1* when *x* is greater than *0,* otherwise, it's *0*.

# The Gradient

As we mentioned at the beginning of this chapter, the gradient is a **vector** composed of all of the partial derivatives of a function, calculated with respect to each input variable.

Let's return to one of the partial derivatives of the sum operation that we calculated earlier:

$$f(x, y, z) = 3x^3z - y^2 + 5z + 2yz \quad \rightarrow$$

$$\frac{\partial}{\partial x} f(x, y, z) = 9x^2z$$

$$\frac{\partial}{\partial y} f(x, y, z) = -2y + 2z$$

$$\frac{\partial}{\partial z} f(x, y, z) = 3x^3 + 5 + 2y$$

If we calculate all of the partial derivatives, we can form a gradient of the function. Using different notations, it looks as follows:

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial}{\partial x} f(x, y, z) \\ \frac{\partial}{\partial y} f(x, y, z) \\ \frac{\partial}{\partial z} f(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} f(x, y, z) = \begin{bmatrix} 9x^2z \\ -2y + 2z \\ 3x^3 + 5 + 2y \end{bmatrix}$$

That's all we have to know about the **gradient** - it's a vector of all of the possible partial derivatives of the function, and we denote it using the $\nabla$ — nabla symbol that looks like an inverted delta symbol.

We'll be using **derivatives** of single-parameter functions and **gradients** of multivariate functions to perform **gradient descent** using the **chain rule,** or, in other words, to perform the **backward pass**, which is a part of the model training. How exactly we'll do that is the subject of the next chapter.

# The Chain Rule

During the forward pass, we're passing the data through the neurons, then through the activation function, then through the neurons in the next layer, then through another activation function, and so on. We're calling a function with an input parameter, taking an output, and using that output as an input to another function. For this simple example, let's take 2 functions: *f* and *g*:

$$z = f(x)$$
$$y = g(z)$$

x is the input data, z is an output of the function f, but also an input for the function g, and y is an output of the function g. We could write the same calculation as:

$$y = g(f(x))$$

In this form, we do not use the intermediate z variable, showing that function g takes the output of function f directly as an input. This does not differ much from the above 2 equations but shows an important property of functions chained this way — since x is an input to the function f and then the output of the function f is an input to the function g, the output of the function g is influenced by x in some way, so there must exist a derivative which can inform us of this influence.

The forward pass through our model is a chain of functions similar to these examples. We are passing in samples, the data flows through all of the layers, and activation functions to form an output. Let's bring the equation and the code of the example model from chapter 1:

$$L = -\sum_{l=1}^{N} y_l log(\forall_{j=1}^{n_3} \frac{e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,j}))_i w_{2,i,j}+b_{2,j}))_i w_{3,i,j}+b_{3,j}}{\sum_{k=1}^{n_3} e^{\sum_{i=1}^{n_2}(\forall_{j=1}^{n_2} max(0,\sum_{i=1}^{n_1}(\forall_{j=1}^{n_1} max(0,\sum_{i=1}^{n_0} X_i w_{1,i,j}+b_{1,k}))_i w_{2,i,j}+b_{2,k}))_i w_{3,i,k}+b_{3,k}})$$

```
loss = -np.log(
    np.sum(
        y * np.exp(
            np.dot(
                np.maximum(
                    0,
                    np.dot(
                        np.maximum(
                            0,
                            np.dot(
                                X,
                                w1.T
                            ) + b1
                        ),
                        w2.T
                    ) + b2
                ),
                w3.T
            ) + b3
        ) /
        np.sum(
            np.exp(
                np.dot(
                    np.maximum(
                        0,
                        np.dot(
                            np.maximum(
                                0,
                                np.dot(
                                    X,
                                    w1.T
                                ) + b1
                            ),
                            w2.T
                        ) + b2
                    ),
                    w3.T
                ) + b3
            ),
            axis=1,
            keepdims=True
        )
    )
)
```



**Fig 8.01:** Code for a forward pass of an example neural network model.

If you look closely, you'll see that we are presenting the loss as a big function, or a chain of functions, of multiple inputs — input data, weights, and biases. We are passing input data to the first layer where we also have that layer's weights and biases, then the outputs flow through the ReLU activation function, and another layer, which brings more weights and biases, and another ReLU activation, up to the end — the output layer and softmax activation. The model output, along with the targets, is passed to the loss function, which returns the model's error. We can look at the loss function not only as a function that takes the model's output and targets as parameters to produce the error, but also as a function that takes targets, samples, and all of the weights and biases as inputs if we chain all of the functions performed during the forward pass as we've just shown in the images. To improve loss, we need to learn how each weight and bias impacts it. How to do that for a chain of functions? By using the chain rule. This rule says that the derivative of a function chain is a product of derivatives of all of the functions in this chain, for example:

$$\frac{d}{dx}f(g(x)) = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx} = f'(g(x)) \cdot g'(x)$$

First, we wrote the derivative of the outer function, *f(g(x))*, with respect to the inner function, *g(x)*, as this inner function is its parameter. Next, we multiplied it by the derivative of the inner function, *g(x)*, with respect to its parameters, *x*. We also denoted this derivative using 2 different notations. With 3 functions and multiple inputs, the partial derivative of this function with respect to *x* is as follows (we can't use the prime notation in this case since we have to mention which variable we are deriving with respect to):

$$\frac{\partial}{\partial x}f(g(y, h(x, z))) = \frac{\partial f(g(y, h(x, z)))}{\partial g(y, h(x, z))} \cdot \frac{\partial g(y, h(x, z))}{\partial h(x, z)} \cdot \frac{\partial h(x, z)}{\partial x}$$

To calculate the partial derivative of a chain of functions with respect to some parameter, we take the partial derivative of the outer function with respect to the inner function in a chain to the parameter. Then multiply this partial derivative by the partial derivative of the inner function with respect to the more inner function in a chain to the parameter, then multiply this by the partial derivative of the more inner function with respect to the other function in the chain. We repeat this all the way down to the parameter in question. Notice, for example, how the middle derivative is with respect to *h(x, z)* and not *y* as *h(x, z)* is in the chain to the parameter *x*. The **chain rule** turns out to be the most important rule in finding the impact of singular input to the output of a chain of functions, which is the calculation of loss in our case. We'll use it again in the next chapter when we discuss and code backpropagation. For now, let's cover an example of the chain rule.

Let's solve the derivative of $h(x) = 3(2x^2)^5$. The first thing that we can notice here is that we have a complex function that can be split into two simpler functions. First is an equation part contained inside the parentheses, which we can write as $g(x) = 2x^2$. That's the inside function that we exponentiate and multiply with the rest of the equation. The remaining part of the equation can then be written as $f(y) = 3(y)^5$. *y* in this case is what we denoted as $g(x)=2x^2$ and when we combine it back, we get $h(x) = f(g(x)) = 3(2x^2)^5$ . To calculate a derivative of this function, we start by taking that outside exponent, the $^5$, and place it in front of the component that we are exponentiating to multiply it later by the leading 3, giving us 15. We then subtract 1 from the $^5$ exponent, leaving us with a $^4$.

$$h(x) = f(g(x)) = 3(2x^2)^5 \quad \rightarrow \quad f'(g(x)) = 3 \cdot 5(2x^2)^{5-1} = 15(2x^2)^4$$

Then the chain rule informs us to multiply the above derivative of the outer function, with the derivative of the interior function, giving us:

$$\rightarrow \quad h'(x) = f'(g(x)) \cdot g'(x) = 15(2x^2)^4 \cdot \frac{d}{dx}2x^2 = 15(2x^2)^4 \cdot 2 \cdot \frac{d}{dx}x^2 =$$
$$= 15(2x^2)^4 \cdot 2 \cdot 2x^1 = 15(2x^2)^4 \cdot 4x$$

Recall that *4x* was the derivative of $2x^2$, which is the inner function, *g(x)*. This highlights the **chain rule** concept in an example, allowing us to calculate the derivatives of more complex functions by chaining together the derivatives. Note that we multiplied by the derivative of that interior function, but left the interior function *unchanged* within the derivative of the outer function.

In theory, we could just stop here with a perfectly-usable derivative of the function. We can enter some input into $15(2x^2)^4 \cdot 4x$ and get the answer. That said, we can also go ahead and simplify this function for more practice. Coming back to the original problem, so far we've found:

$$f(x) = 3(2x^2)^5 \quad \rightarrow \quad f'(x) = 15(2x^2)^4 \cdot 4x$$

To simplify this derivative function, we first take $(2x^2)^4$ and distribute the $^4$ exponent:

$$f'(x) = 15(2x^2)^4 \cdot 4x = 15 \cdot (2^4 \cdot x^{2 \cdot 4}) \cdot 4x = 15 \cdot 2^4 \cdot x^8 \cdot 4x$$

Combine the *x's*:

$$f'(x) = 15 \cdot 2^4 \cdot x^8 \cdot 4x = 15 \cdot 2^4 \cdot 4 \cdot x^{8+1} = 15 \cdot 2^4 \cdot 4 \cdot x^9$$

And the constants:

$$f'(x) = 15 \cdot 2^4 \cdot 4 \cdot x^9 = 15 \cdot 16 \cdot 4 \cdot x^9 = 960x^9$$

We'll simplify derivatives later as well for faster computation — there's no reason to repeat the same operations when we can solve them in advance.

Hopefully, now you understand what derivatives and partial derivatives are, what the gradient is, what the derivative of the loss function with respect to weights and biases means, and how to use the chain rule. For now, these terms might sound disconnected, but we're going to use them all to perform gradient descent in the backpropagation step, which is the subject of the next chapters.

# Summary

Let's summarize the rules that we have learned in this chapter.

The partial derivative of the sum with respect to any input equals 1:

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = 1$$
$$\frac{\partial}{\partial y} f(x,y) = 1$$

The partial derivative of the multiplication operation with 2 inputs, with respect to any input, equals the other input:

$$f(x,y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = y$$
$$\frac{\partial}{\partial y} f(x,y) = x$$

The partial derivative of the max function of 2 variables with respect to any of them is 1 if this variable is the biggest and 0 otherwise. An example of x:

$$f(x,y) = max(x,y) \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = 1(x > y)$$

The derivative of the max function of a single variable and 0 equals 1 if the variable is greater than 0 and 0 otherwise:

$$f(x) = max(x,0) \quad \rightarrow \quad \frac{d}{dx} f(x) = 1(x > 0)$$

The derivative of chained functions equals the product of the partial derivatives of the subsequent functions:

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x)) \cdot \frac{d}{dx}g(x) = f'(g(x)) \cdot g'(x)$$

The same applies to the partial derivatives. For example:

$$\frac{\partial}{\partial x}f(g(y, h(x, z))) = f'(g(y, h(x, z))) \cdot g'(y, h(x, z)) \cdot h'(x, z)$$

The gradient is a vector of all possible partial derivatives. An example of a triple-input function:

$$\nabla f(x, y, z) = \begin{bmatrix} \frac{\partial}{\partial x}f(x, y, z) \\ \frac{\partial}{\partial y}f(x, y, z) \\ \frac{\partial}{\partial z}f(x, y, z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} f(x, y, z)$$

**Supplementary Material:** https://nnfs.io/ch8
Chapter code, further resources, and errata for this chapter.

# Chapter 9

# *Backpropagation*

Now that we have an idea of how to measure the impact of variables on a function's output, we can begin to write the code to calculate these partial derivatives to see their role in minimizing the model's loss. Before applying this to a complete neural network, let's start with a simplified forward pass with just one neuron. Rather than backpropagating from the loss function for a full neural network, let's backpropagate the ReLU function for a single neuron and act as if we intend to minimize the output for this single neuron. We're first doing this only as a demonstration to simplify the explanation, since minimizing the output from a ReLU activated neuron doesn't serve any purpose other than as an exercise. Minimizing the loss value is our end goal, but in this case, we'll start by showing how we can leverage the chain rule with derivatives and partial derivatives to calculate the impact of each variable on the ReLU activated output. We'll also start by minimizing this more basic output before jumping to the full network and overall loss.

Let's quickly recall the forward pass and atomic operations that we need to perform for this single neuron and ReLU activation. We'll use an example neuron with 3 inputs, which means that it also has 3 weights and a bias:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

We then start with the first input, `x[0]`, and the related weight, `w[0]`:

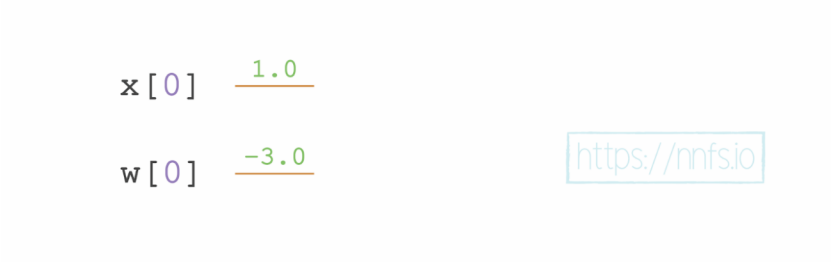


**Fig 9.01:** Beginning a forward pass with the first input and weight.

We have to multiply the input by the weight:

```
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias


xw0 = x[0] * w[0]
print(xw0)


>>>
-3.0
```

Visually:


**Fig 9.02:** The first input and weight multiplication.

We repeat this operation for `x1`, `w1` and `x2`, `w2` pairs:

```
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw1, xw2)


>>>
2.0 6.0
```

Visually:

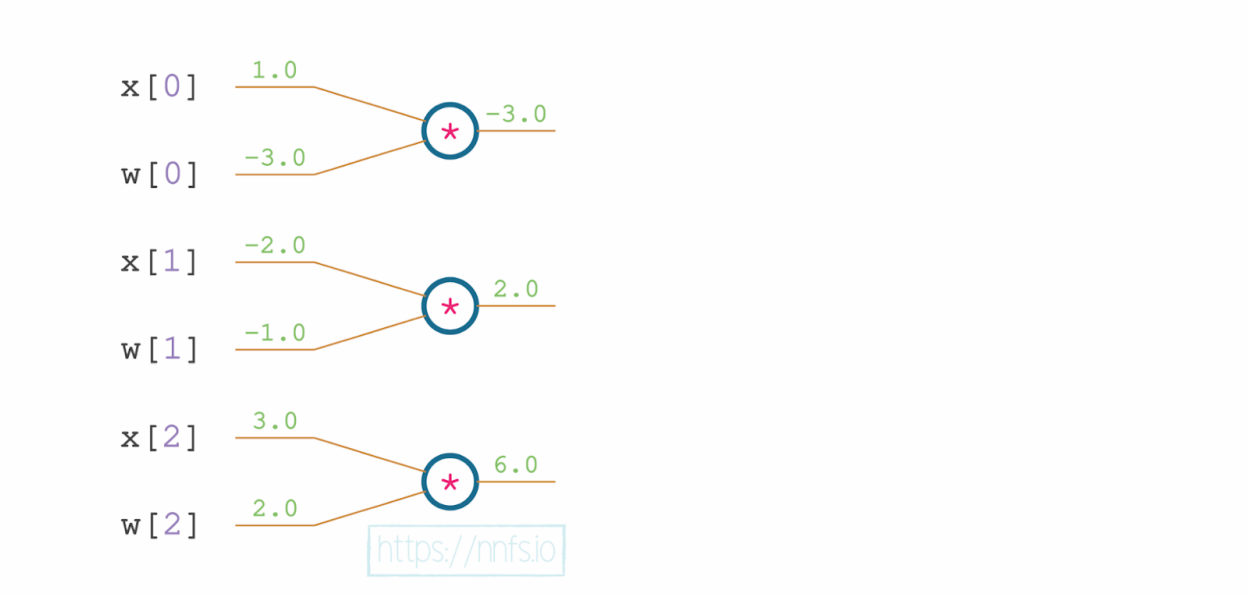


**Fig 9.03:** Input and weight multiplication of all of the inputs.

Code all together:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2)


>>>
-3.0 2.0 6.0
```

The next operation to perform is a sum of all weighted inputs with a bias:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2, b)

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b
print(z)


>>>
-3.0 2.0 6.0 1.0
6.0
```

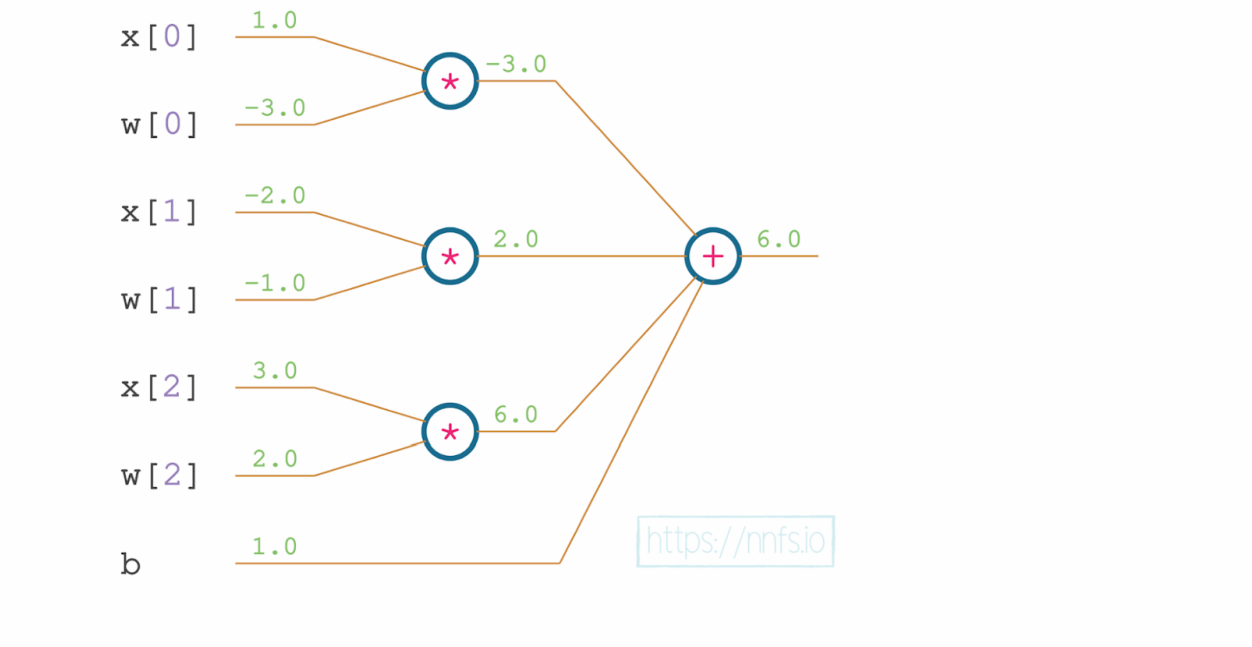


**Fig 9.04:** Weighted inputs and bias addition.

This forms the neuron's output. The last step is to apply the ReLU activation function on this output:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
print(xw0, xw1, xw2, b)

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b
print(z)

# ReLU activation function
y = max(z, 0)
print(y)


>>>
-3.0 2.0 6.0 1.0
6.0
6.0
```

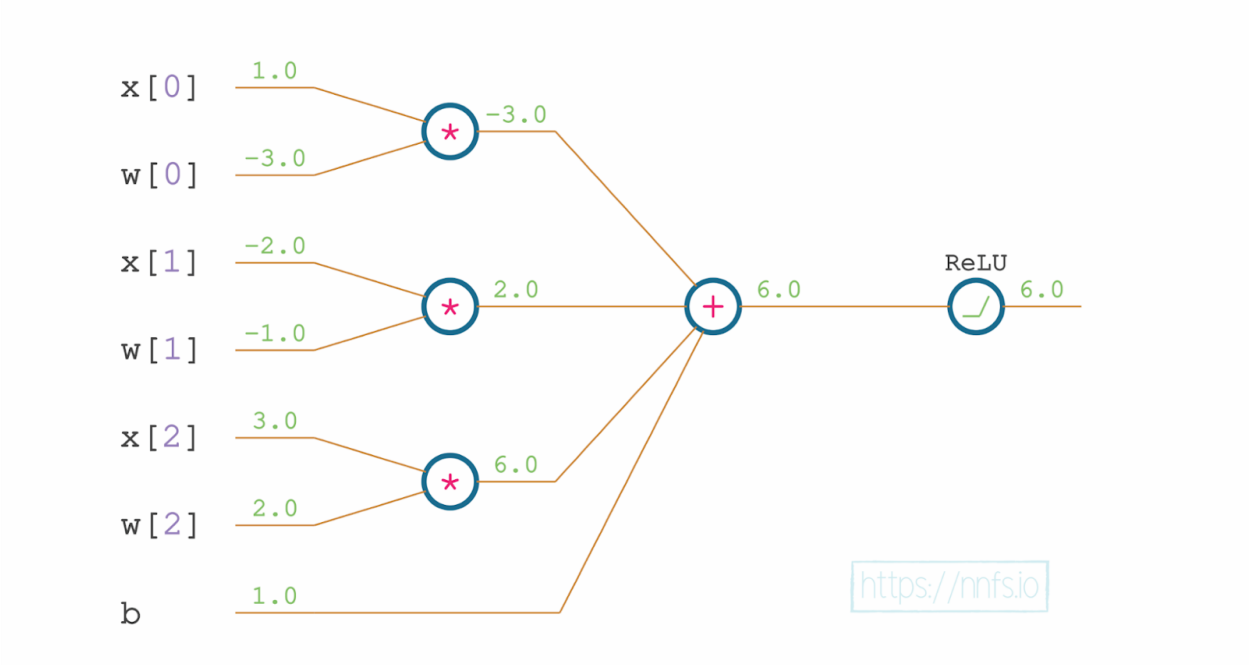


**Fig 9.05:** ReLU activation applied to the neuron output.

This is the full forward pass through a single neuron and a ReLU activation function. Let's treat all of these chained functions as one big function which takes input values (*x*), weights (*w*), and bias (*b*), as inputs, and outputs *y*. This big function consists of multiple simpler functions — there is a multiplication of input values and weights, sum of these values and bias, as well as a *max* function as the ReLU activation — 3 chained functions in total:

The first step is to backpropagate our gradients by calculating derivatives and partial derivatives with respect to each of our parameters and inputs. To do this, we're going to use the **chain rule**. Recall that the chain rule for a function stipulates that the derivative for nested functions like *f(g(x))* solves to:

$$\frac{d}{dx}f(g(x)) = \frac{d}{dg(x)}f(g(x)) \cdot \frac{d}{dx}g(x) = f'(g(x)) \cdot g'(x)$$

This big function that we just mentioned can be, in the context of our neural network, loosely interpreted as:

$$ReLU(\sum[inputs \cdot weights] + bias)$$

Or in the form that matches code more precisely as:

$$ReLU(x_0w_0 + x_1w_1 + x_2w_2 + b)$$

Our current task is to calculate how much each of the inputs, weights, and a bias impacts the output. We'll start by considering what we need to calculate for the partial derivative of $w_0$, for example. But first, let's rewrite our equation to the form that will allow us to determine how to calculate the derivatives more easily:

*y = ReLU(sum(mul($x_0$, $w_0$), mul($x_1$, $w_1$), mul($x_2$, $w_2$), b))*

The above equation contains 3 nested functions: *ReLU*, a sum of weighted inputs and a bias, and multiplications of the inputs and weights. To calculate the impact of the example weight, $w_0$, on the output, the chain rule tells us to calculate the derivative of *ReLU* with respect to its parameter, which is the sum, then multiply it with the partial derivative of the sum operation with respect to its *mul($x_0$, $w_0$)* input, as this input contains the parameter in question. Then, multiply this with the partial derivative of the multiplication operation with respect to the $x_0$ input. Let's see this in a simplified equation:

$$\frac{\partial}{\partial x_0}[ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))] =$$

$$\frac{dReLU()}{dsum()} \cdot \frac{\partial sum()}{\partial mul(x_0, w_0)} \cdot \frac{\partial mul(x_0, w_0)}{\partial x_0}$$

For legibility, we did not denote the *ReLU*() parameter, which is the full sum, and the sum parameters, which are all of the multiplications of inputs and weights. We excluded this because the equation would be longer and harder to read. This equation shows that we have to calculate the derivatives and partial derivatives of all of the atomic operations and multiply them to acquire the impact that $x_0$ makes on the output. We can then repeat this to calculate all of the other remaining impacts. The derivatives with respect to the weights and a bias will inform us about their impact and will be used to update these weights and bias. The derivatives with respect to inputs are used to chain more layers by passing them to the previous function in the chain.

We'll have multiple chained layers of neurons in the neural network model, followed by the loss function. We want to know the impact of a given weight or bias on the loss. That means that we will have to calculate the derivative of the loss function (which we'll do later in this chapter) and apply the chain rule with the derivatives of all activation functions and neurons in all of the consecutive layers. The derivative with respect to the layer's inputs, as opposed to the derivative with respect to the weights and biases, is not used to update any parameters. Instead, it is used to chain to another layer (which is why we backpropagate to the previous layer in a chain).

During the backward pass, we'll calculate the derivative of the loss function, and use it to multiply with the derivative of the activation function of the output layer, then use this result to multiply by the derivative of the output layer, and so on, through all of the hidden layers and activation functions. Inside these layers, the derivative with respect to the weights and biases will form the gradients that we'll use to update the weights and biases. The derivatives with respect to inputs will form the gradient to chain with the previous layer. This layer can calculate the impact of its weights and biases on the loss and backpropagate gradients on inputs further.

For this example, let's assume that our neuron receives a gradient of *1* from the next layer. We're making up this value for demonstration purposes, and a value of *1* won't change the values, which means that we can more easily show all of the processes. We are going to use the color of red for derivatives:

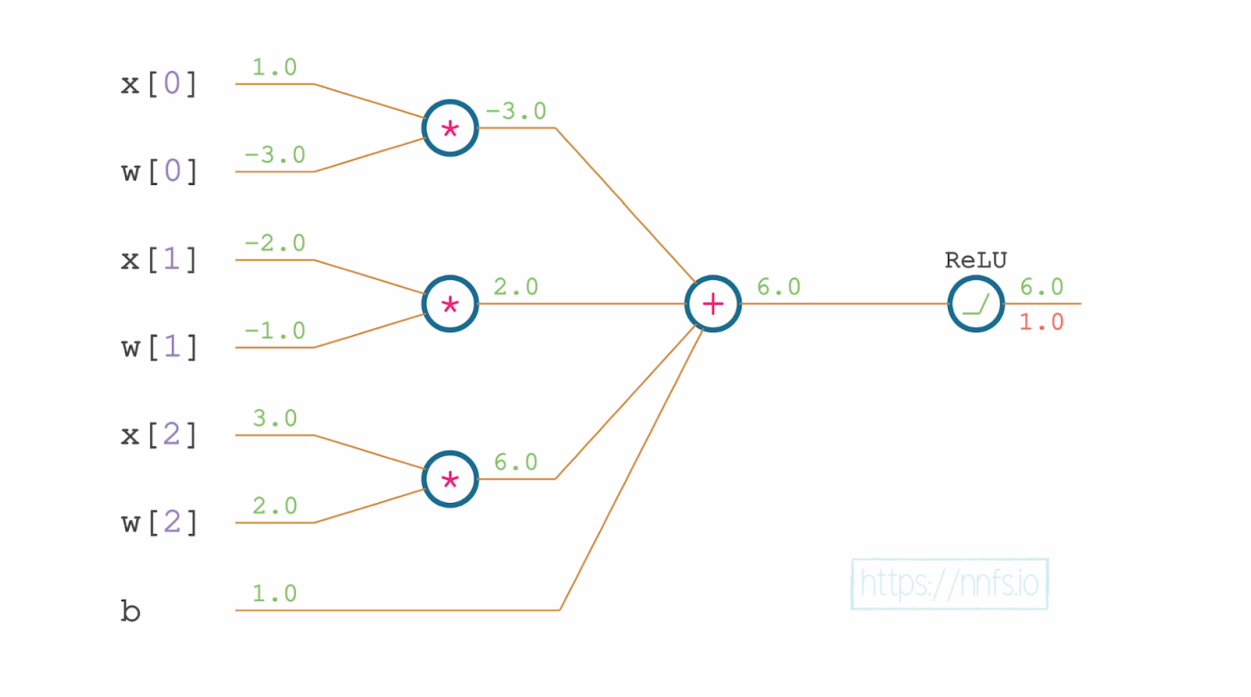


**Fig 9.06:** Initial gradient (received during backpropagation).

Recall that the derivative of *ReLU()* with respect to its input is *1,* if the input is greater than *0,* and *0* otherwise:

$$f(x) = max(x, 0) \quad \rightarrow \quad \frac{d}{dx} f(x) = 1(x > 0)$$

We can write that in Python as:

```
relu_dz = (1. if z > 0 else 0.)
```

Where the `drelu_dz` means the derivative of the *ReLU* function with respect to *z* — we used *z* instead of *x* from the equation since the equation denotes the *max* function in general, and we are applying it to the neuron's output, which is *z*.

The input value to the *ReLU* function is *6*, so the derivative equals *1*. We have to use the chain rule and multiply this derivative with the derivative received from the next layer, which is *1* for the purpose of this example:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias
```

```
# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)


>>>
1.0
```

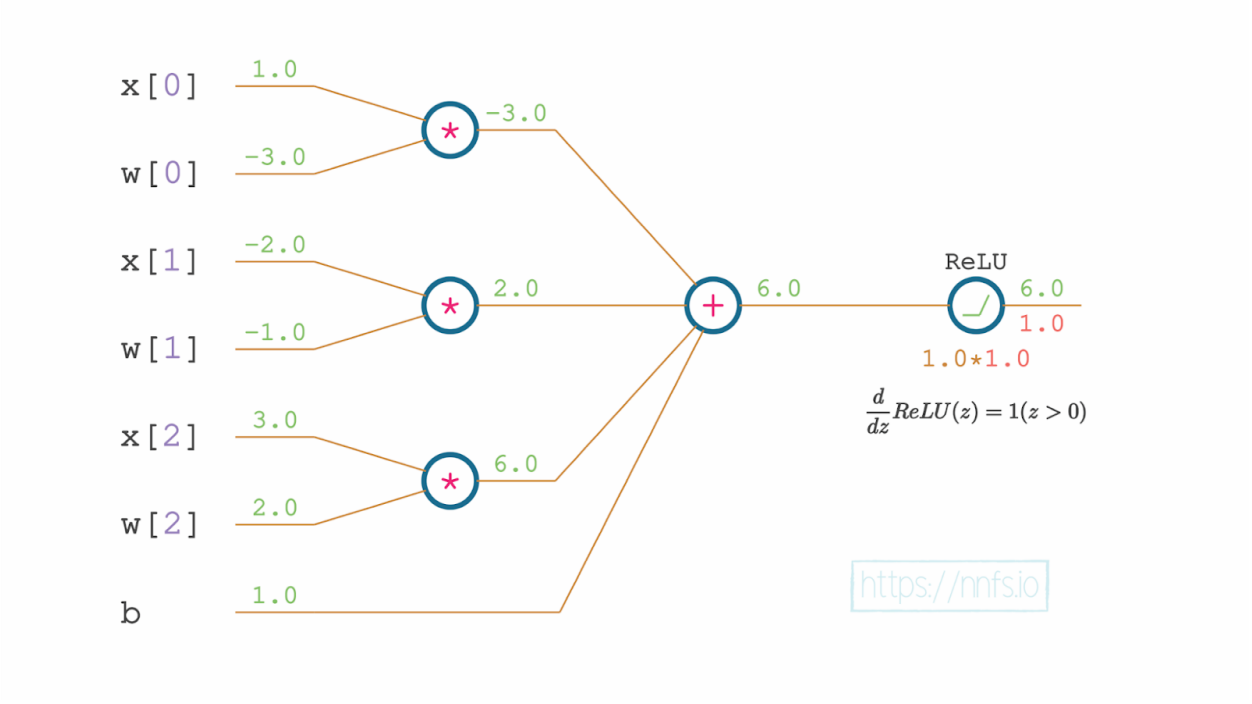


**Fig 9.07:** Derivative of the ReLU function and chain rule.

This results with the derivative of *1*:

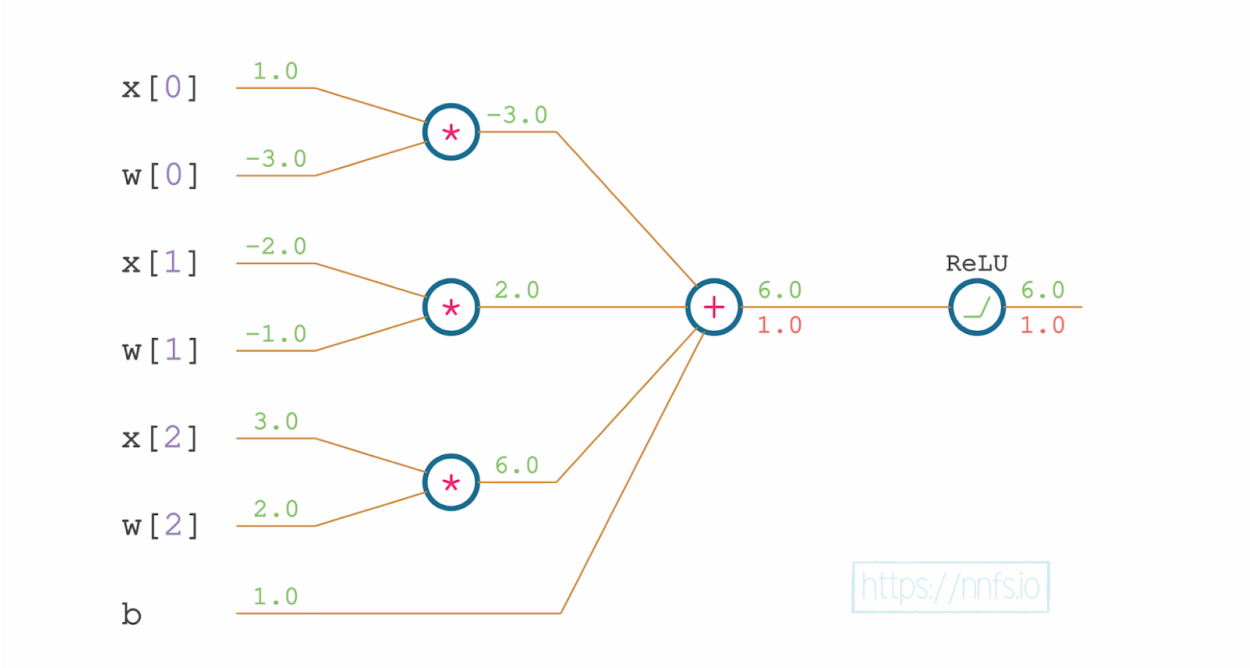


**Fig 9.08:** ReLU and chain rule gradient.

Moving backward through our neural network, what is the function that comes immediately before we perform the activation function?

It's a sum of the weighted inputs and bias. This means that we want to calculate the partial derivative of the sum function, and then, using the chain rule, multiply this by the partial derivative of the subsequent, outer, function, which is *ReLU*. We'll call these results the:

- `drelu_dxw0` — the partial **d**erivative of the **ReLU** w.r.t. the first weighed input, $\boldsymbol{w_0 x_0}$,
- `drelu_dxw1` — the partial **d**erivative of the **ReLU** w.r.t. the second weighed input, $\boldsymbol{w_1 x_1}$,
- `drelu_dxw2` — the partial **d**erivative of the **ReLU** w.r.t. the third weighed input, $\boldsymbol{w_2 x_2}$,
- `drelu_db` — the partial **d**erivative of the **ReLU** with respect to the bias, $\boldsymbol{b}$.

The partial derivative of the sum operation is always *1*, no matter the inputs:

$$f(x, y) = x + y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x, y) = 1$$

$$\frac{\partial}{\partial y} f(x, y) = 1$$

The weighted inputs and bias are summed at this stage. So we will calculate the partial derivatives of the sum operation with respect to each of these, multiplied by the partial derivative for the subsequent function (using the chain rule), which is the *ReLU* function, denoted by `drelu_dz`

For the first partial derivative:

```
dsum_dxw0 = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
```

To be clear, the `dsum_dxw0` above means the partial **d**erivative of the **sum** with respect to the **x** (input), **w**eighted, for the **0**th pair of inputs and weights. *1* is the value of this partial derivative, which we multiply, using the chain rule, with the derivative of the subsequent function, which is the *ReLU* function.

Again, we have to apply the chain rule and multiply the derivative of the ReLU function with the partial derivative of the sum, with respect to the first weighted input:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
print(drelu_dxw0)


>>>
1.0
1.0
```

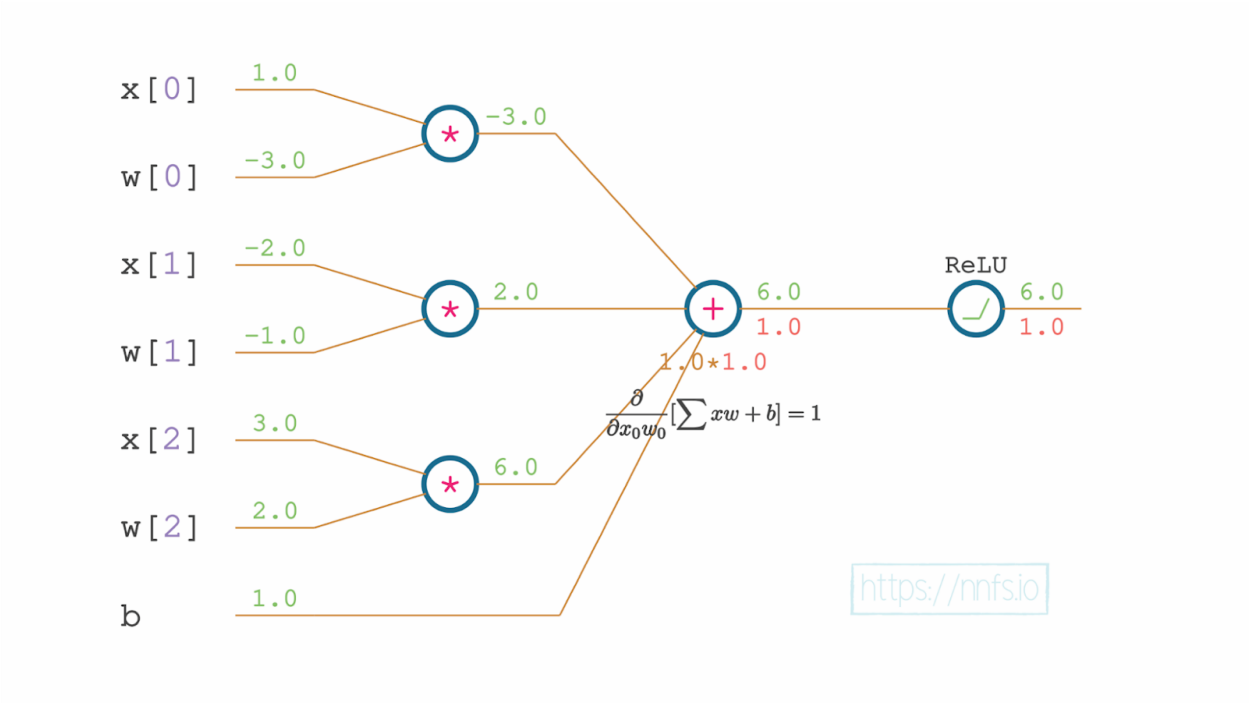


**Fig 9.09:** Partial derivative of the sum function w.r.t. the first weighted input; the chain rule.

This results with a partial derivative of 1 again:

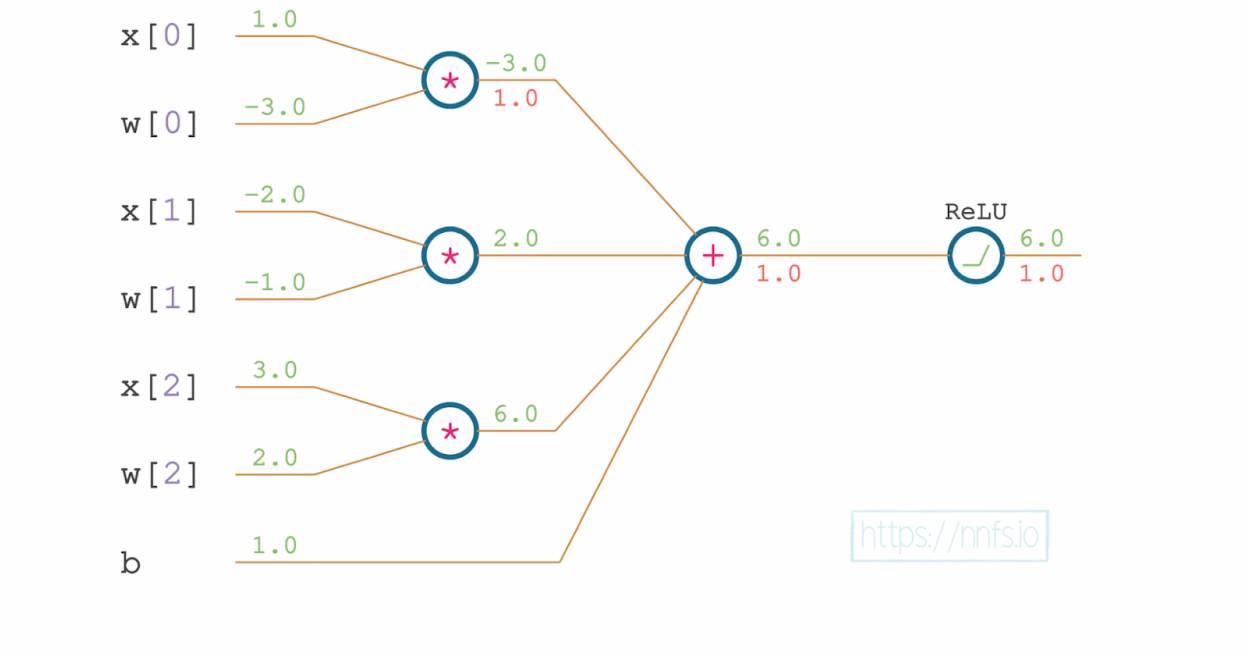


**Fig 9.10:** The sum and chain rule gradient (for the first weighted input).

We can then perform the same operation with the next weighed input:

```
dsum_dxw1 = 1
drelu_dxw1 = drelu_dz * dsum_dxw1
```

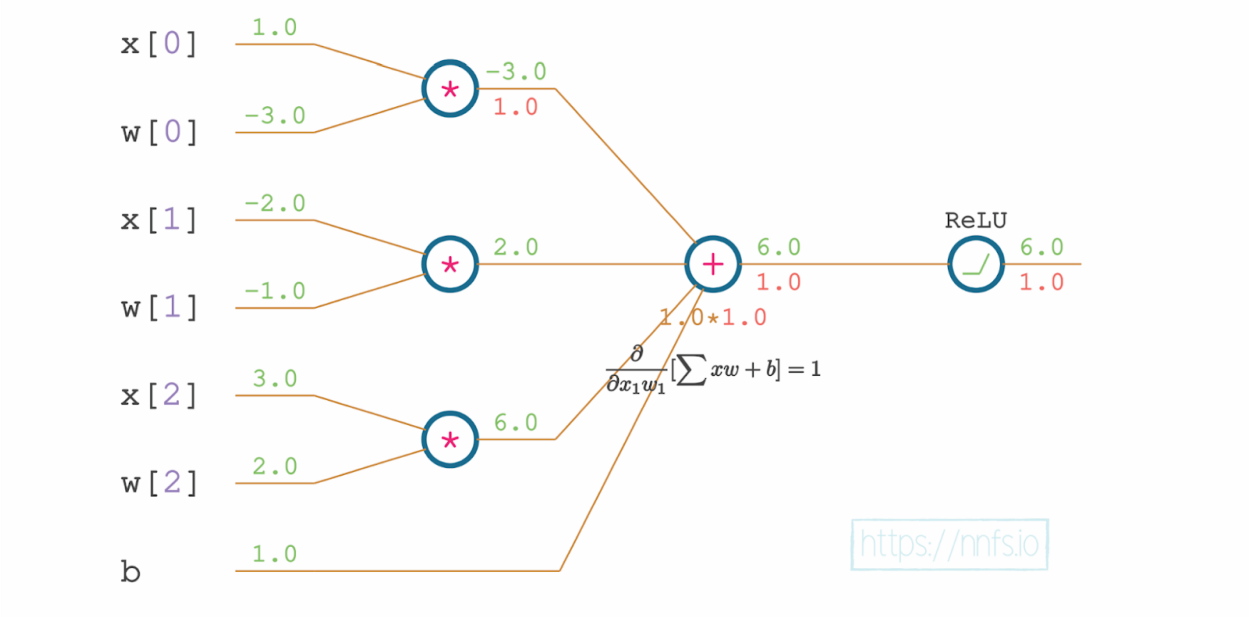


**Fig 9.11:** Partial derivative of the sum function w.r.t. the second weighted input; the chain rule.

Which results with the next calculated partial derivative:

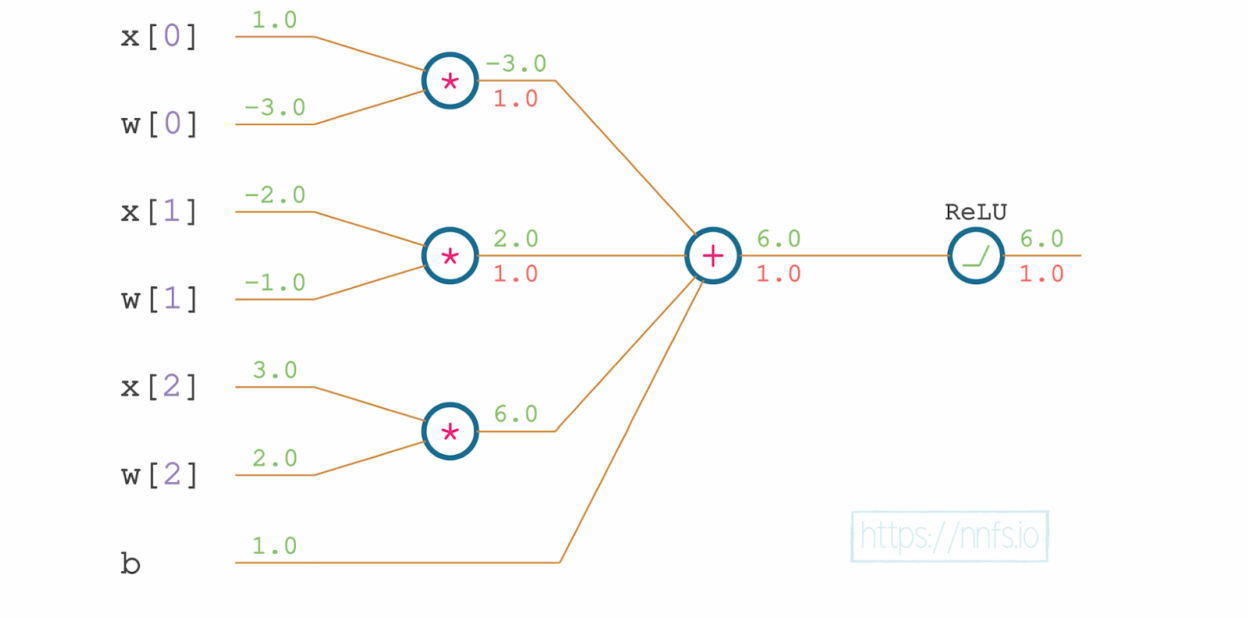


**Fig 9.12:** The sum and chain rule gradient (for the second weighted input).

And the last weighted input:

```
dsum_dxw2 = 1
drelu_dxw2 = drelu_dz * dsum_dxw2
```

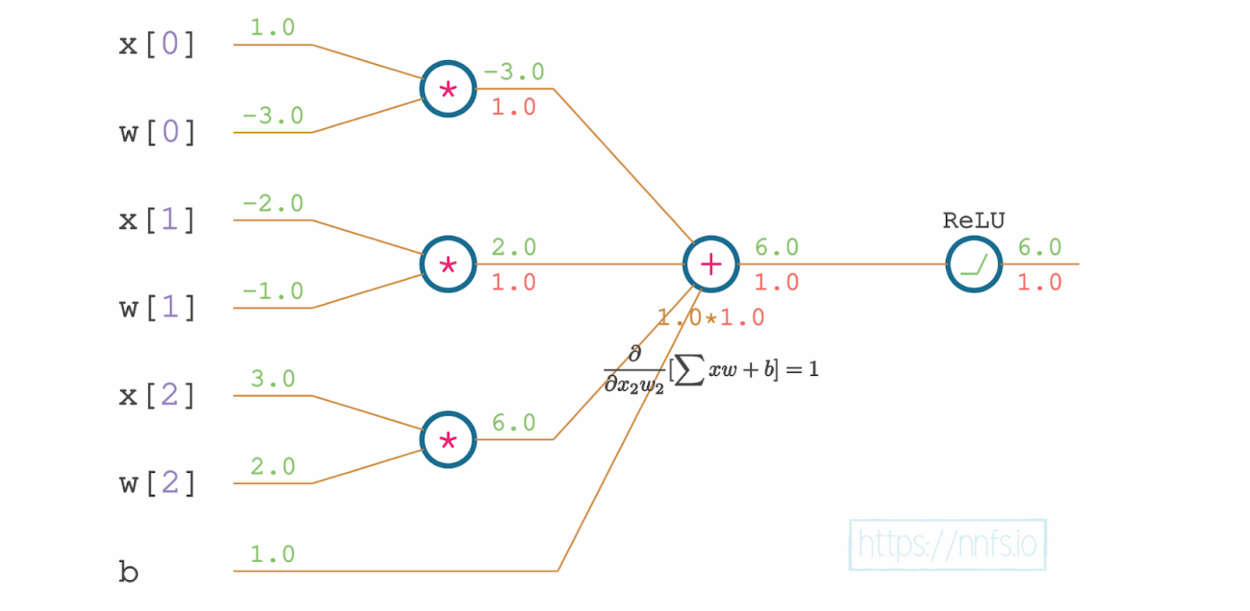


**Fig 9.13:** Partial derivative of the sum function w.r.t. the third weighted input; the chain rule.

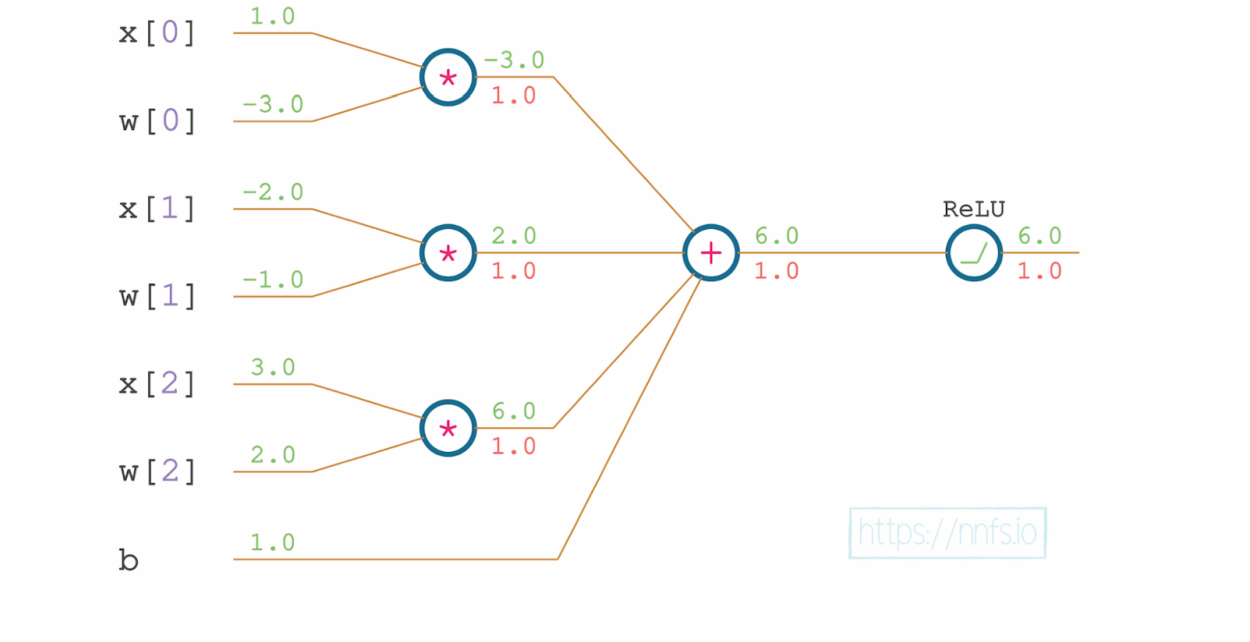


**Fig 9.14:** The sum and chain rule gradient (for the third weighted input).

Then the bias:

```
dsum_db = 1
drelu_db = drelu_dz * dsum_db
```

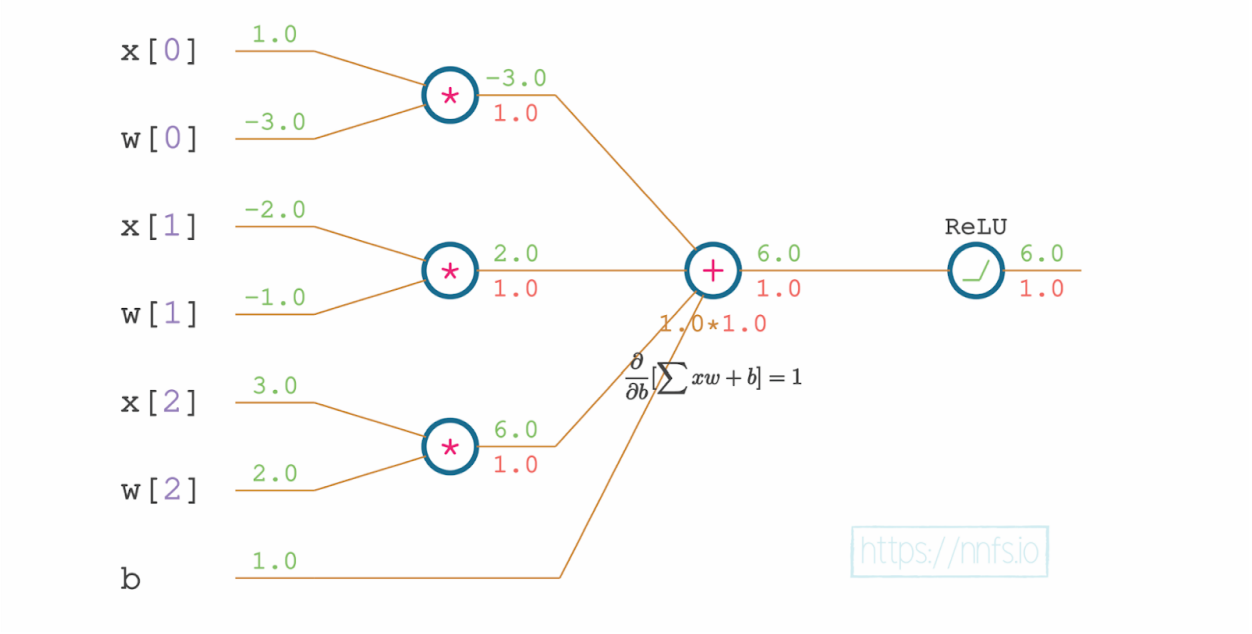


**Fig 9.15:** Partial derivative of the sum function w.r.t. the bias; the chain rule.

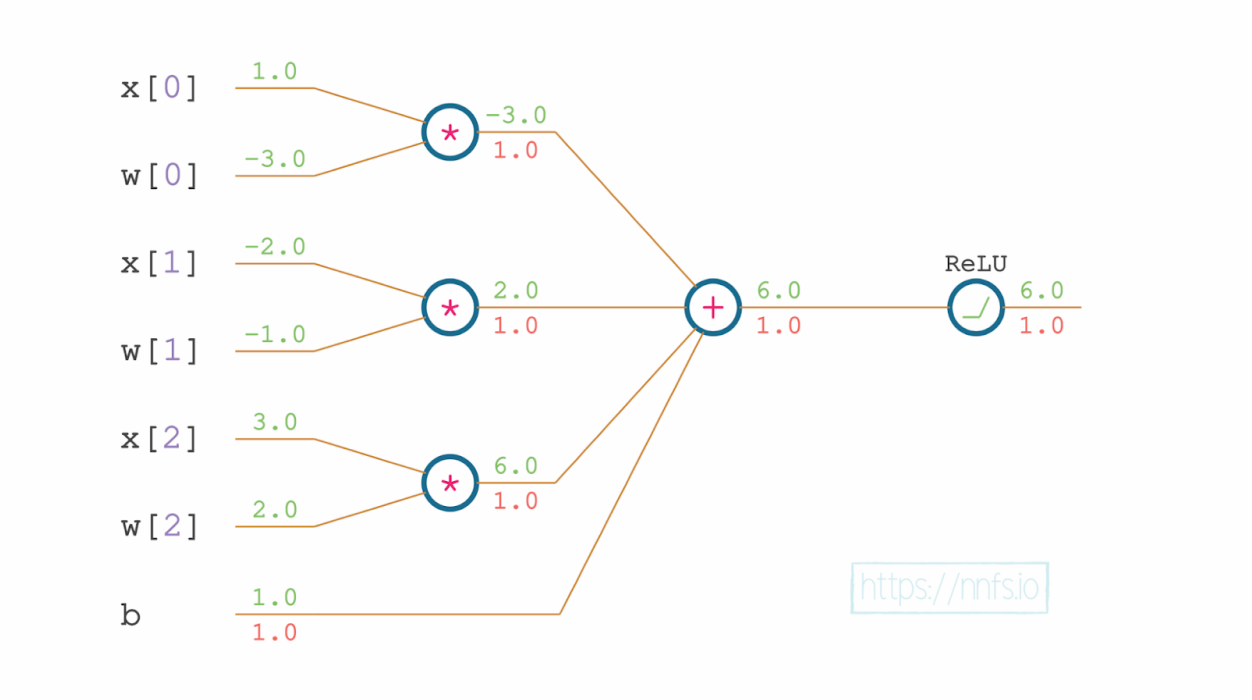


**Fig 9.16:** The sum and chain rule gradient (for the bias).

Let's add these partial derivatives, with the applied chain rule, to our code:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)


>>>
1.0
1.0 1.0 1.0 1.0
```

Continuing backward, the function that comes before the sum is the multiplication of weights and inputs. The derivative for a product is whatever the input is being multiplied by. Recall:

$$f(x,y) = x \cdot y \quad \rightarrow \quad \frac{\partial}{\partial x} f(x,y) = y$$
$$\frac{\partial}{\partial y} f(x,y) = x$$

The partial derivative of *f* with respect to *x* equals *y*. The partial derivative of *f* with respect to *y* equals *x*. Following this rule, the partial derivative of the first *weighted input* with respect to the *input* equals the *weight* (the other input of this function). Then, we have to apply the chain rule and multiply this partial derivative with the partial derivative of the subsequent function, which is the sum (we just calculated its partial derivative earlier in this chapter):

```
dmul_dx0 = w[0]
drelu_dx0 = drelu_dxw0 * dmul_dx0
```

This means that we are calculating the partial derivative with respect to the $x_0$ input, the value of which is $w_0$, and we are applying the chain rule with the derivative of the subsequent function, which is `drelu_dxw0`.

This is a good time to point out that, as we apply the chain rule in this way — working backward by taking the *ReLU()* derivative, taking the summing operation's derivative, multiplying both, and so on, this is a process called **backpropagation** using the **chain rule**. As the name implies, the resulting output function's gradients are passed back through the neural network, using multiplication of the gradient of subsequent functions from later layers with the current one. Let's add this partial derivative to the code and show it on the chart:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass
```

```
# The derivative from the next layer
dvalue = 1.0

# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)

# Partial derivatives of the multiplication, the chain rule
dmul_dx0 = w[0]
drelu_dx0 = drelu_dxw0 * dmul_dx0
print(drelu_dx0)


>>>
1.0
1.0 1.0 1.0 1.0
-3.0
```

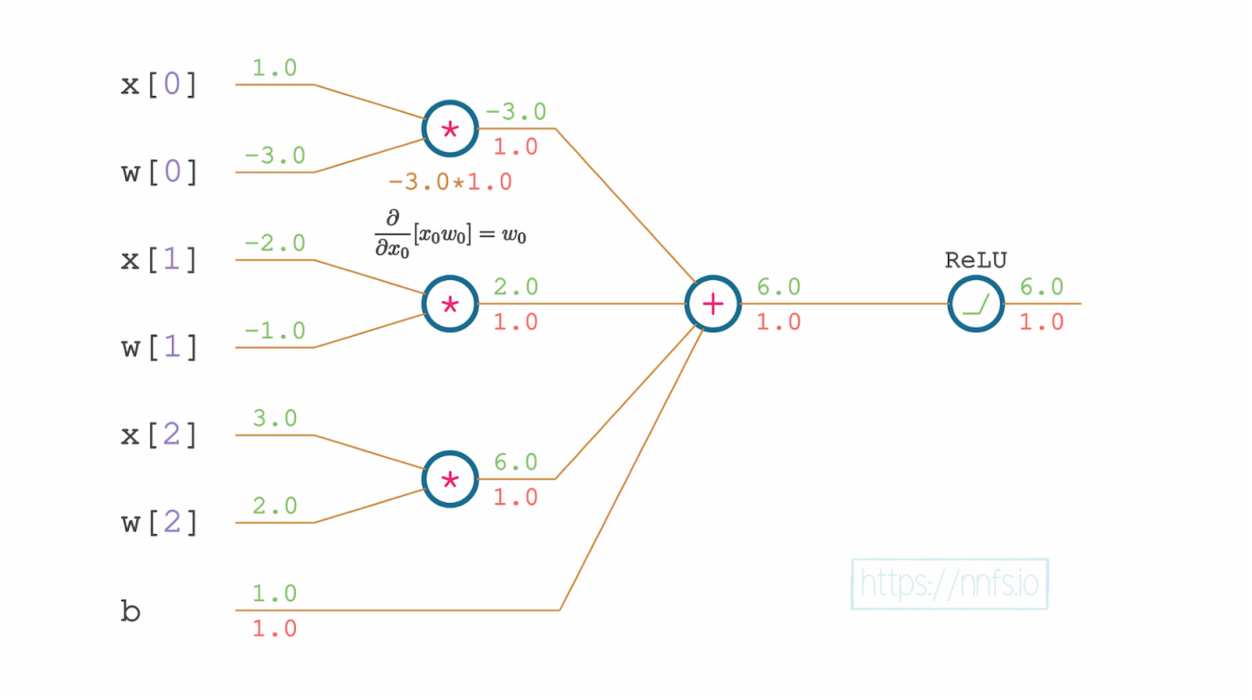


**Fig 9.17:** Partial derivative of the multiplication function w.r.t. the first input; the chain rule.

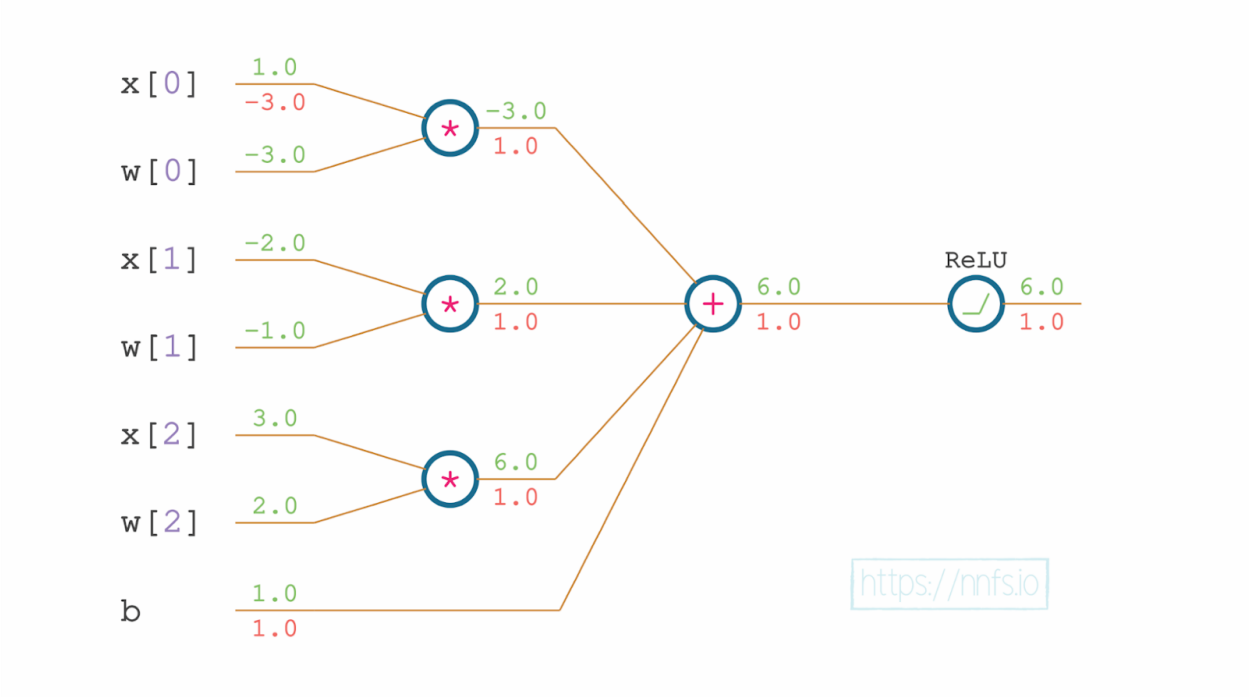


**Fig 9.18:** The multiplication and chain rule gradient (for the first input).

We perform the same operation for other inputs and weights:

```
# Forward pass
x = [1.0, -2.0, 3.0]  # input values
w = [-3.0, -1.0, 2.0]  # weights
b = 1.0  # bias

# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]

# Adding weighted inputs and a bias
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)

# Backward pass

# The derivative from the next layer
dvalue = 1.0
```

```
# Derivative of ReLU and the chain rule
drelu_dz = dvalue * (1. if z > 0 else 0.)
print(drelu_dz)

# Partial derivatives of the multiplication, the chain rule
dsum_dxw0 = 1
dsum_dxw1 = 1
dsum_dxw2 = 1
dsum_db = 1
drelu_dxw0 = drelu_dz * dsum_dxw0
drelu_dxw1 = drelu_dz * dsum_dxw1
drelu_dxw2 = drelu_dz * dsum_dxw2
drelu_db = drelu_dz * dsum_db
print(drelu_dxw0, drelu_dxw1, drelu_dxw2, drelu_db)

# Partial derivatives of the multiplication, the chain rule
dmul_dx0 = w[0]
dmul_dx1 = w[1]
dmul_dx2 = w[2]
dmul_dw0 = x[0]
dmul_dw1 = x[1]
dmul_dw2 = x[2]
drelu_dx0 = drelu_dxw0 * dmul_dx0
drelu_dw0 = drelu_dxw0 * dmul_dw0
drelu_dx1 = drelu_dxw1 * dmul_dx1
drelu_dw1 = drelu_dxw1 * dmul_dw1
drelu_dx2 = drelu_dxw2 * dmul_dx2
drelu_dw2 = drelu_dxw2 * dmul_dw2
print(drelu_dx0, drelu_dw0, drelu_dx1, drelu_dw1, drelu_dx2, drelu_dw2)


>>>
1.0
1.0 1.0 1.0 1.0
-3.0 1.0 -1.0 -2.0 2.0 3.0
```

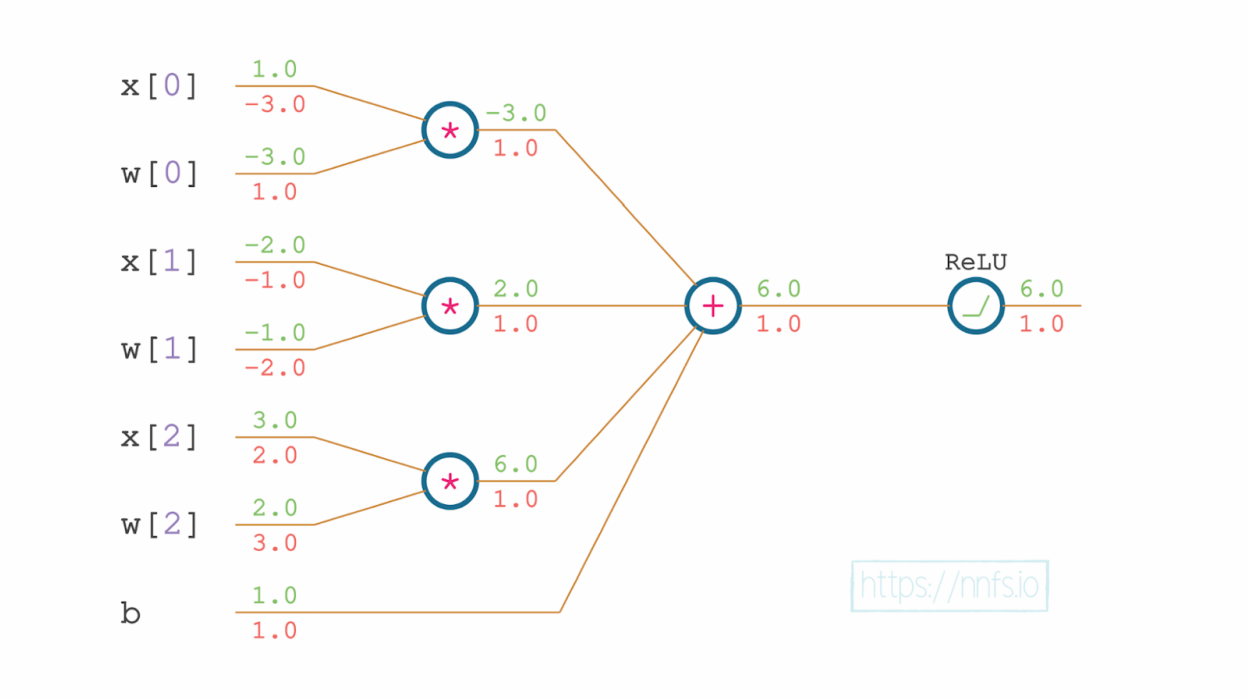


**Fig 9.19:** Complete backpropagation graph.

**Anim 9.01-9.19:** https://nnfs.io/pro

That's the complete set of the activated neuron's partial derivatives with respect to the inputs, weights and a bias.

Recall the equation from the beginning of this chapter:

$$\frac{\partial}{\partial x_0}[ReLU(sum(mul(x_0, w_0), mul(x_1, w_1), mul(x_2, w_2), b))] =$$

$$\frac{dReLU()}{dsum()} \cdot \frac{\partial sum()}{\partial mul(x_0, w_0)} \cdot \frac{\partial mul(x_0, w_0)}{\partial x_0}$$

Since we have the complete code and we are applying the chain rule from this equation, let's see what we can optimize in these calculations. We applied the chain rule to calculate the

partial derivative of the ReLU activation function with respect to the first input, $x_0$. In our code, let's take the related lines of the code and simplify them:

```
drelu_dx0 = drelu_dxw0 * dmul_dx0
```

where:

```
dmul_dx0 = w[0]
```

then:

```
drelu_dx0 = drelu_dxw0 * w[0]
```

where:

```
drelu_dxw0 = drelu_dz * dsum_dxw0
```

then:

```
drelu_dx0 = drelu_dz * dsum_dxw0 * w[0]
```

where:

```
dsum_dxw0 = 1
```

then:

```
drelu_dx0 = drelu_dz * 1 * w[0] = drelu_dz * w[0]
```

where:

```
drelu_dz = dvalue * (1. if z > 0 else 0.)
```

then:

```
drelu_dx0 = dvalue * (1. if z > 0 else 0.) * w[0]
```

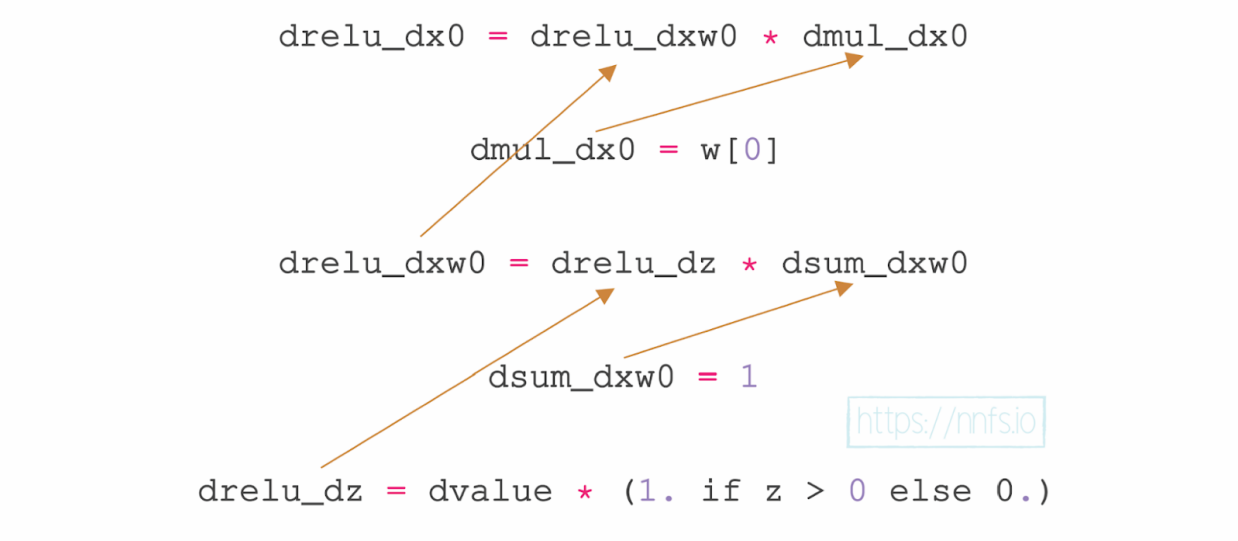


**Fig 9.20:** How to apply the chain rule for the partial derivative of ReLU w.r.t. first input

```
drelu_dx0 = dvalue * (1. if z > 0 else 0.) * w[0]
```

**Fig 9.21:** The chain rule applied for the partial derivative of ReLU w.r.t. first input

**Anim 9.20-9.21:** https://nnfs.io/com

In this equation, starting from the left-hand side, is the derivative calculated in the next layer, with respect to its inputs — this is the gradient backpropagated to the current layer, which is the derivative of the *ReLU* function, and the partial derivative of the neuron's function with respect to the $x_0$ input. This is all multiplied by applying the chain rule to calculate the impact of the input to the neuron on the whole function's output.

The partial derivative of a neuron's function, with respect to the weight, is the input related to this weight, and, with respect to the input, is the related weight. The partial derivative of the neuron's function with respect to the bias is always 1. We multiply them with the derivative of the subsequent function (which was *1* in this example) to get the final derivatives. We are going to code all of these derivatives in the Dense layer's class and the ReLU activation class for the backpropagation step.

All together, the partial derivatives above, combined into a vector, make up our gradients. Our gradients could be represented as:

```
dx = [drelu_dx0, drelu_dx1, drelu_dx2]  # gradients on inputs
dw = [drelu_dw0, drelu_dw1, drelu_dw2]  # gradients on weights
db = drelu_db  # gradient on bias...just 1 bias here.
```

For this single neuron example, we also won't need our `dx`. With many layers, we will continue backpropagating to preceding layers with the partial derivative with respect to our inputs.

Continuing the single neuron example, we can now apply these gradients to the weights to hopefully minimize the output. This is typically the purpose of the **optimizer** (discussed in the following chapter), but we can show a simplified version of this task by directly applying a negative fraction of the gradient to our weights. We apply a negative fraction to this gradient since we want to decrease the final output value, and the gradient shows the direction of the steepest ascent. For example, our current weights and bias are:

```
print(w, b)


>>>
[-3.0, -1.0, 2.0] 1.0
```

We can then apply a fraction of the gradients to these values:

```
w[0] += -0.001 * dw[0]
w[1] += -0.001 * dw[1]
w[2] += -0.001 * dw[2]
b += -0.001 * db

print(w, b)


>>>
[-3.001, -0.998, 1.997] 0.999
```

Now, we've slightly changed the weights and bias in such a way so as to decrease the output somewhat intelligently. We can see the effects of our tweaks on the output by doing another forward pass:

```
# Multiplying inputs by weights
xw0 = x[0] * w[0]
xw1 = x[1] * w[1]
xw2 = x[2] * w[2]
```

```
# Adding
z = xw0 + xw1 + xw2 + b

# ReLU activation function
y = max(z, 0)
print(y)


>>>
5.985
```

We've successfully decreased this neuron's output from 6.000 to 5.985. Note that it does not make sense to decrease the neuron's output in a real neural network; we were doing this purely as a simpler exercise than the full network. We want to decrease the loss value, which is the last calculation in the chain of calculations during the forward pass, and it's the first one to calculate the gradient during the backpropagation. We've minimized the ReLU output of a single neuron only for the purpose of this example to show that we actually managed to decrease the value of chained functions intelligently using the derivatives, partial derivatives, and chain rule. Now, we'll apply the one-neuron example to the list of samples and expand it to an entire layer of neurons. To begin, let's set a list of 3 samples for input, where each sample consists of 4 features. For this example, our network will consist of a single hidden layer, containing 3 neurons (lists of 3 weight sets and 3 biases). We're not going to describe the forward pass again, but the backward pass, in this case, needs further explanation.

So far, we have performed an example backward pass with a single neuron, which received a singular derivative to apply the chain rule. Let's consider multiple neurons in the following layer. A single neuron of the current layer connects to all of them — they all receive the output of this neuron. What will happen during backpropagation? Each neuron from the next layer will return a partial derivative of its function with respect to this input. The neuron in the current layer will receive a vector consisting of these derivatives. We need this to be a singular value for a singular neuron. To continue backpropagation, we need to sum this vector.

Now, let's replace the current singular neuron with a layer of neurons. As opposed to a single neuron, a layer outputs a vector of values instead of a singular value. Each neuron in a layer connects to all of the neurons in the next layer. During backpropagation, each neuron from the current layer will receive a vector of partial derivatives the same way that we described for a single neuron. With a layer of neurons, it'll take the form of a list of these vectors, or a 2D array. We know that we need to perform a sum, but what should we sum and what is the result supposed to be? Each neuron is going to output a gradient of the partial derivatives with respect to all of its inputs, and all neurons will form a list of these vectors. We need to sum along the inputs — the first input to all of the neurons, the second input, and so on. We'll have to sum columns.