Backpropagation

Table of Contents

[Computation Graphs 2](#_Toc138881809)

[Patterns in Gradient Flow 8](#_Toc138881810)

[Backpropagation Implementation 9](#_Toc138881811)

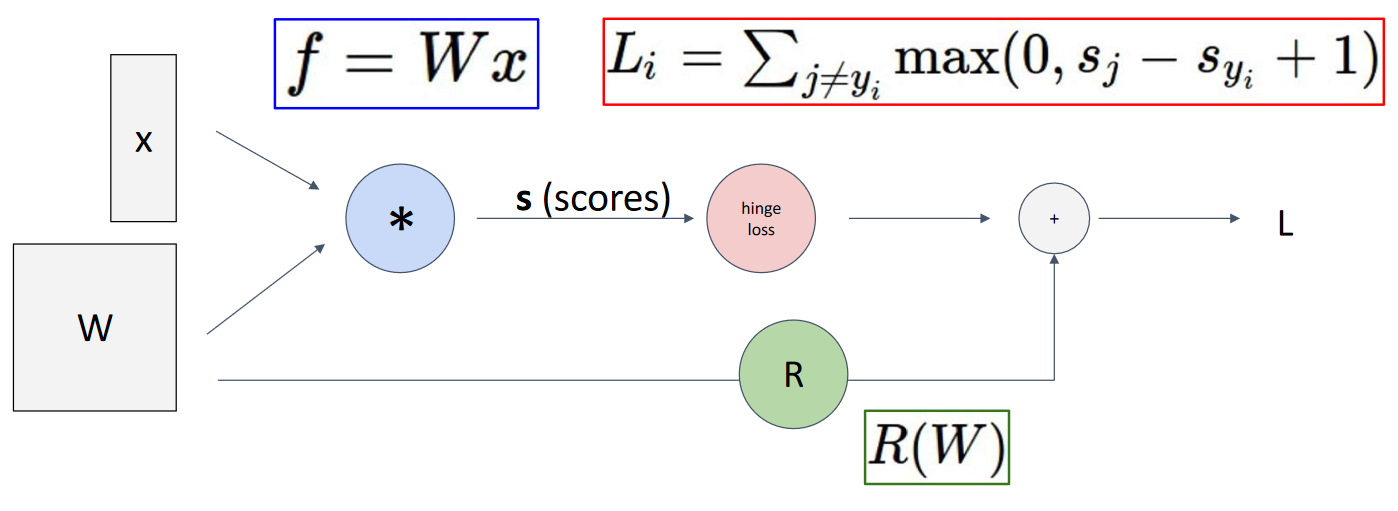
[Vector Derivatives 13](#_Toc138881812)

[Matrix Derivatives 15](#_Toc138881813)

We know that we have to find the partial derivatives of all the weights in a network in order to update them. Doing this manually would be a nightmare, so instead, we use the concepts of Computation Graphs and Backpropagation.

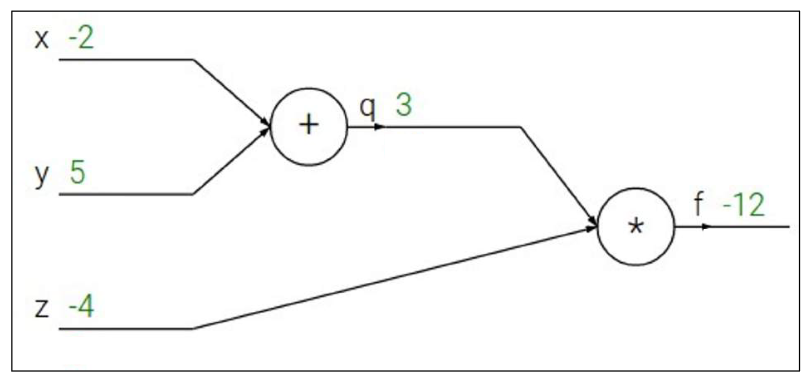
## Computation Graphs

A **Computation Graph** is a visual representation of the mathematical calculations involved in a network. Consider a simple situation where we just have some input being multiplied with a single weight from which we calculate SVM loss along with some regularization. The computation graph for this setup could look like this:



Obviously, larger networks will have far more complicated graphs, but we will stick with simple ones for now.

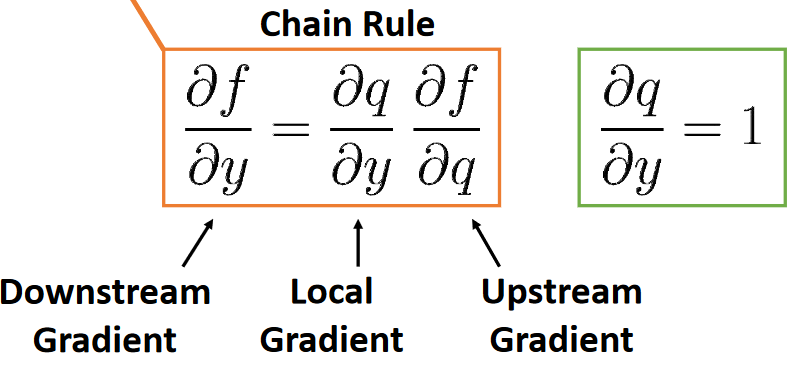
To understand how the process of **backpropagation** works, consider that we have the equation . Here, is being used interchangeably with . The computation graph for this equation looks like this:



The first thing we have to do is the **forward pass**. This is simply calculating the output. We are using some sample values for , and here. We also have to keep track of the intermediate values, marked as here. Thus, we have two equations:

Next, we start the **backward pass**. Our goal is to calculate the derivatives of each of the values with reference to the output, i.e., , and .

For and , notice that we are having to use the **chain rule**. This involves using three terms, the **downstream gradient**, the **upstream gradient** and the **local gradient**.

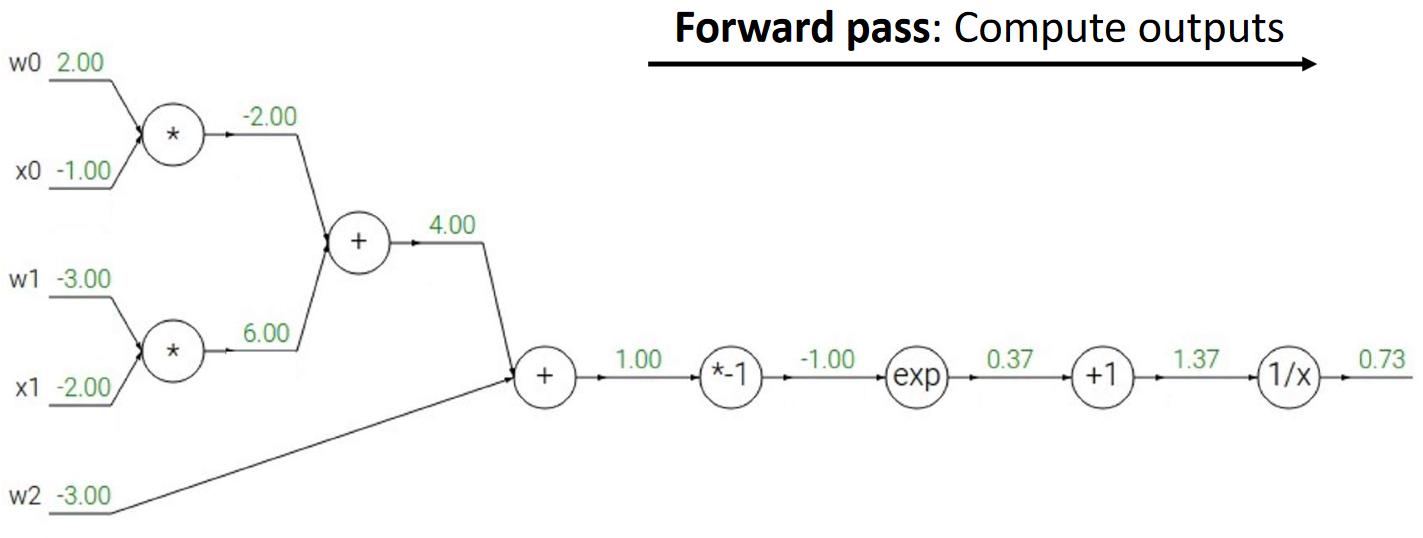


This process is exactly what makes backpropagation so easy. For any given gradient, we only need to worry about the upstream gradient and the local gradient. Since we are going backwards, we already have a value for the upstream gradient, so we just need to calculate the local gradient.

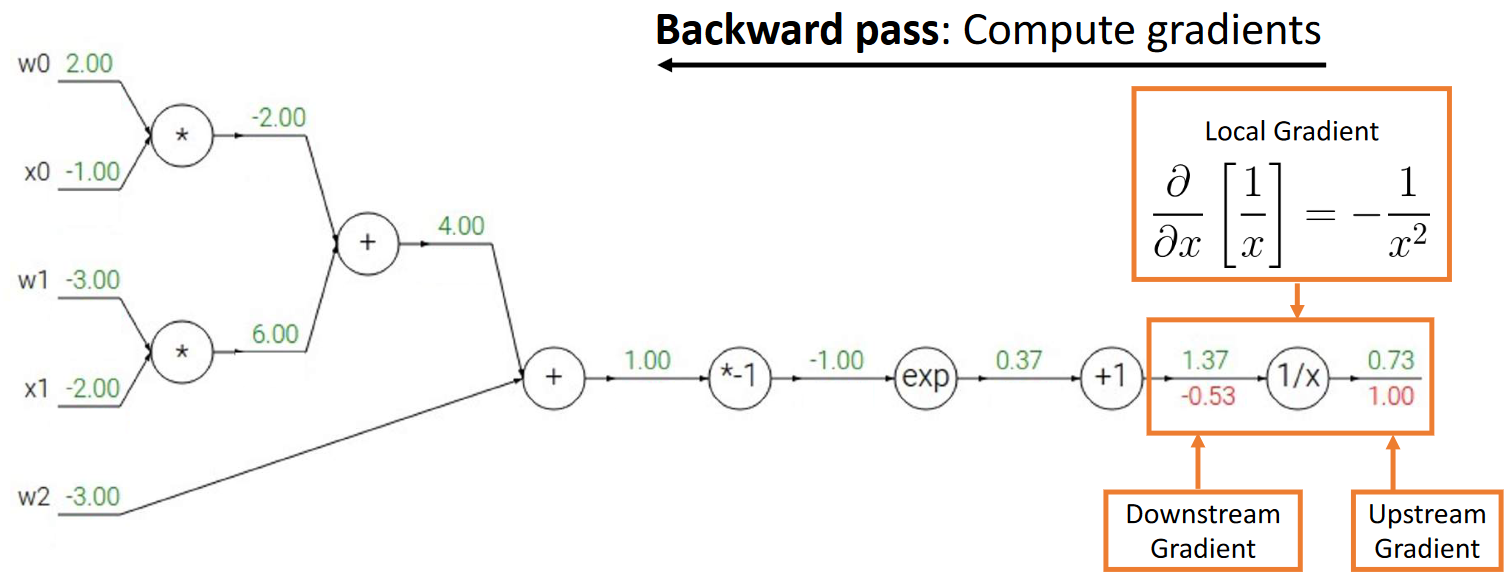
The initial gradient that we calculated, , also follows this process. However, in this case, the upstream gradient is and is just being ignored.

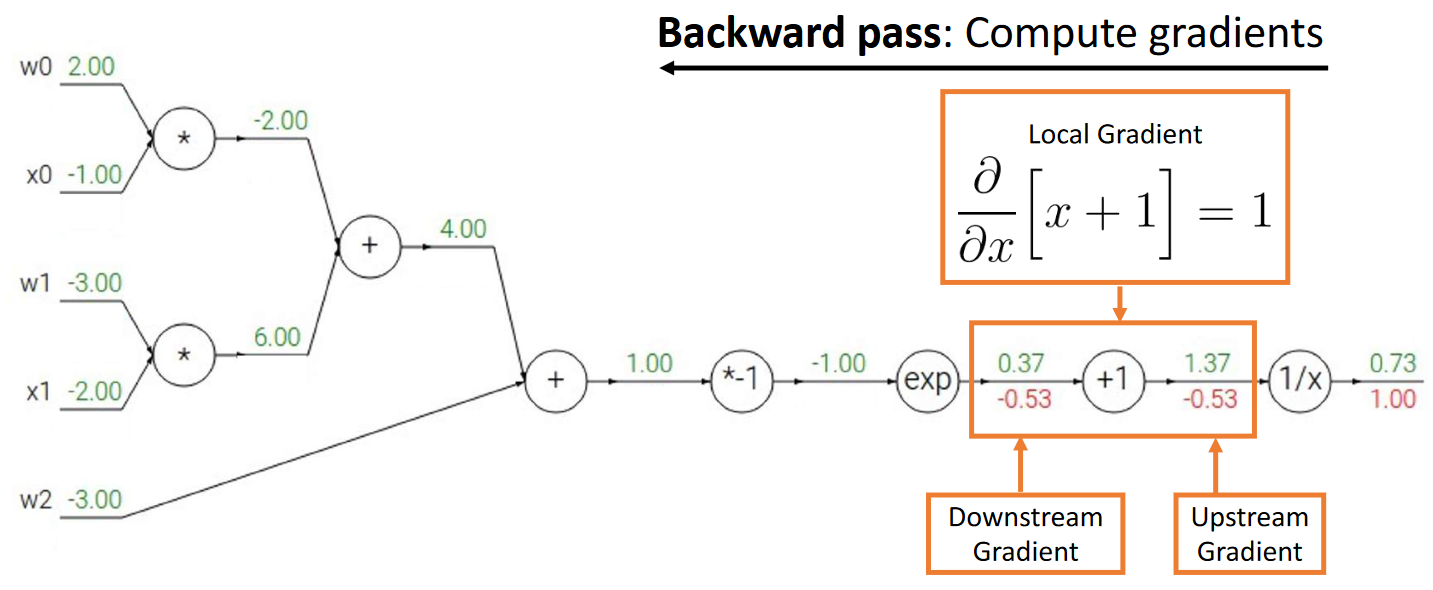
A more complicated example should make the entire process of backpropagation much clearer. Suppose . In this case, we have multiple inputs, each with its own weight, given some output. This is much closer to a proper network.

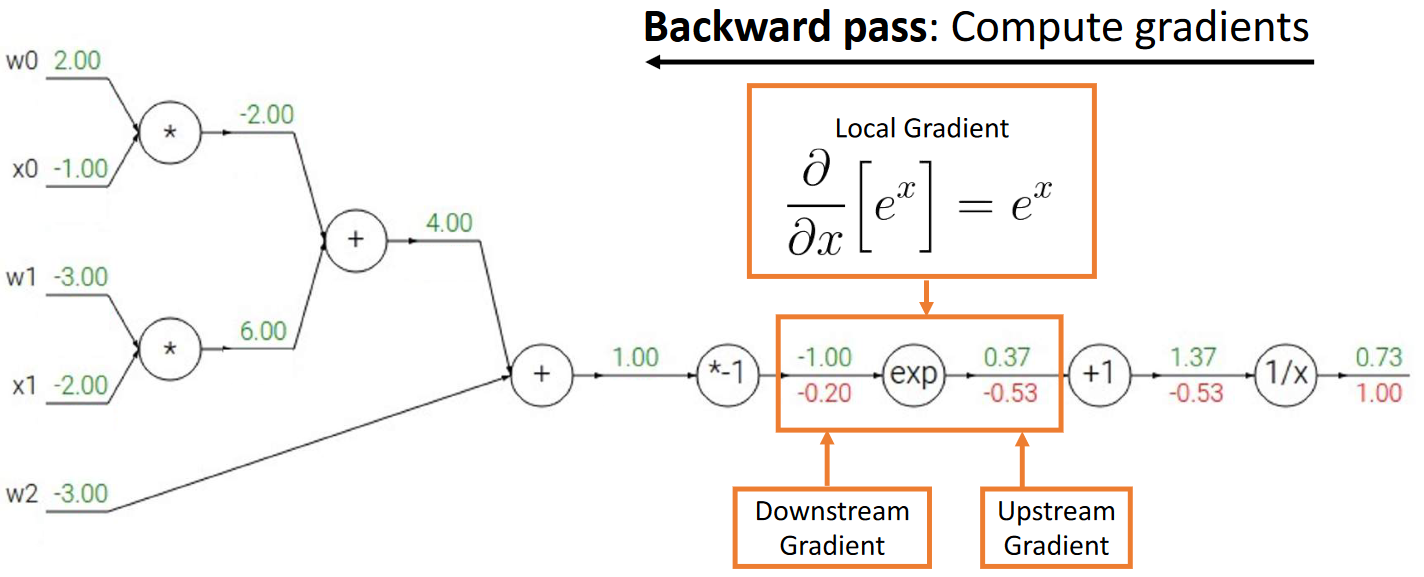
The computation graph for is given below, along with the forward pass values:

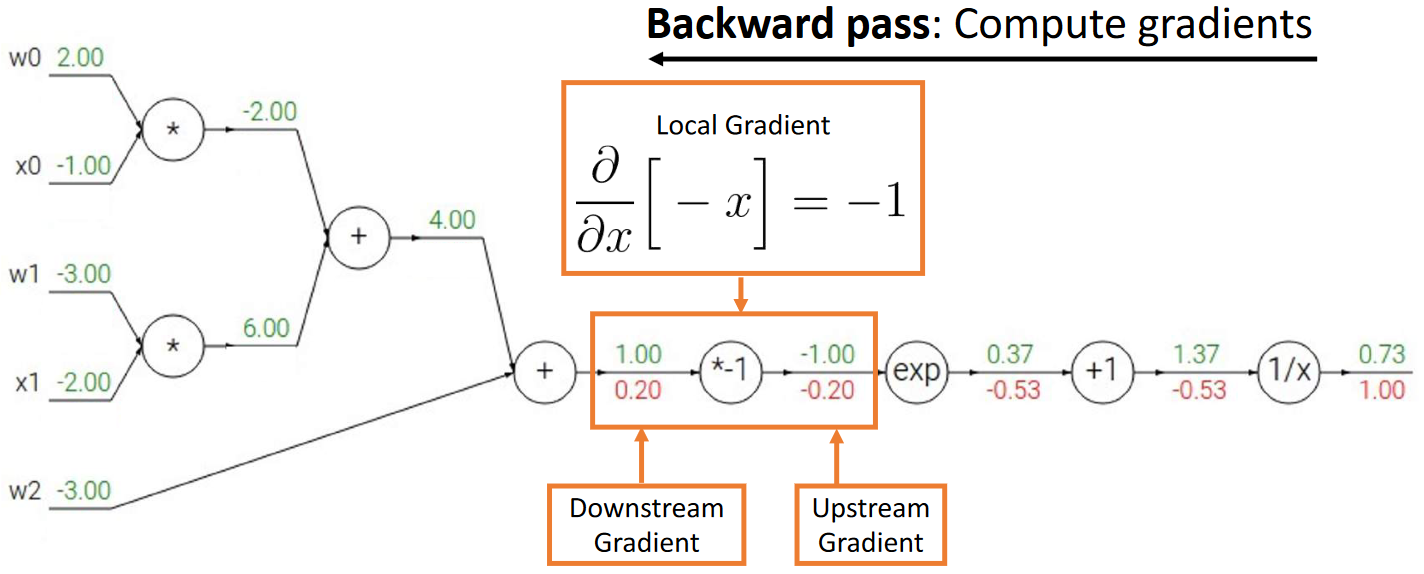


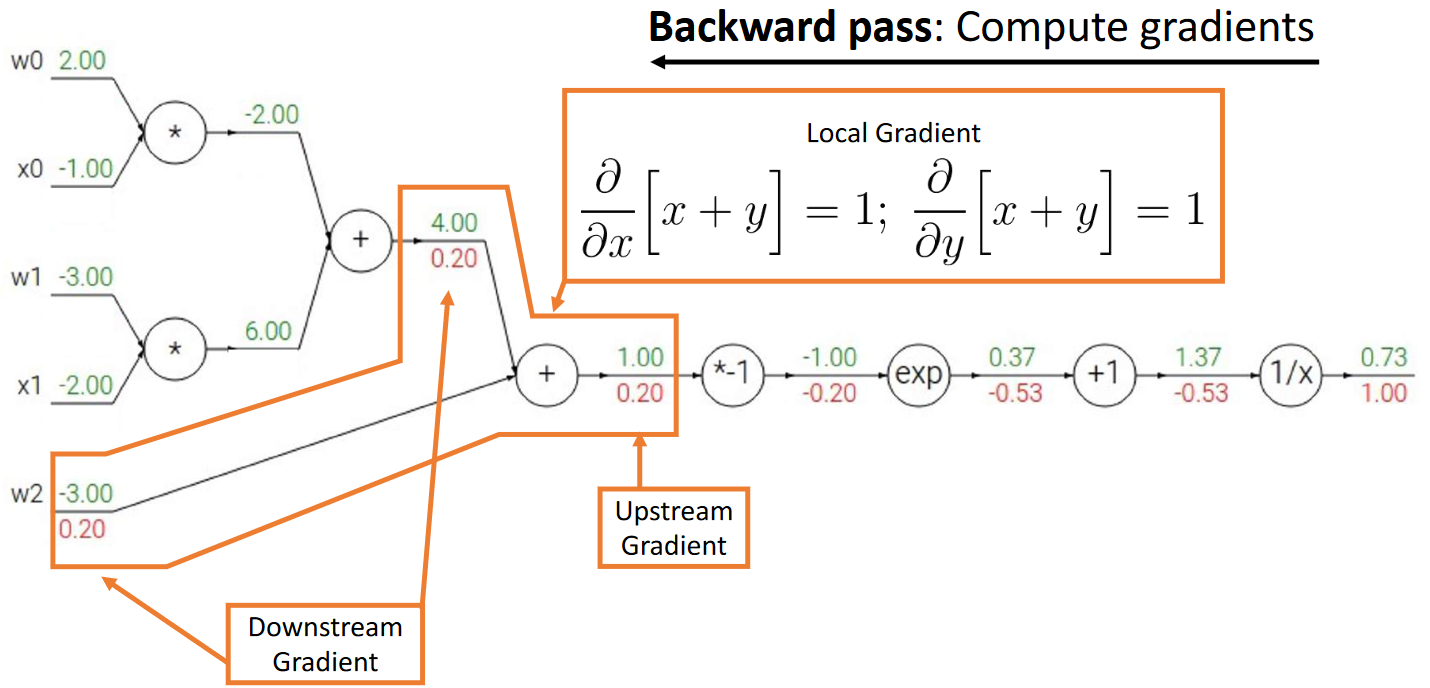
We can now use the upstream and local gradients to calculate each of the downstream gradients starting from the last one.

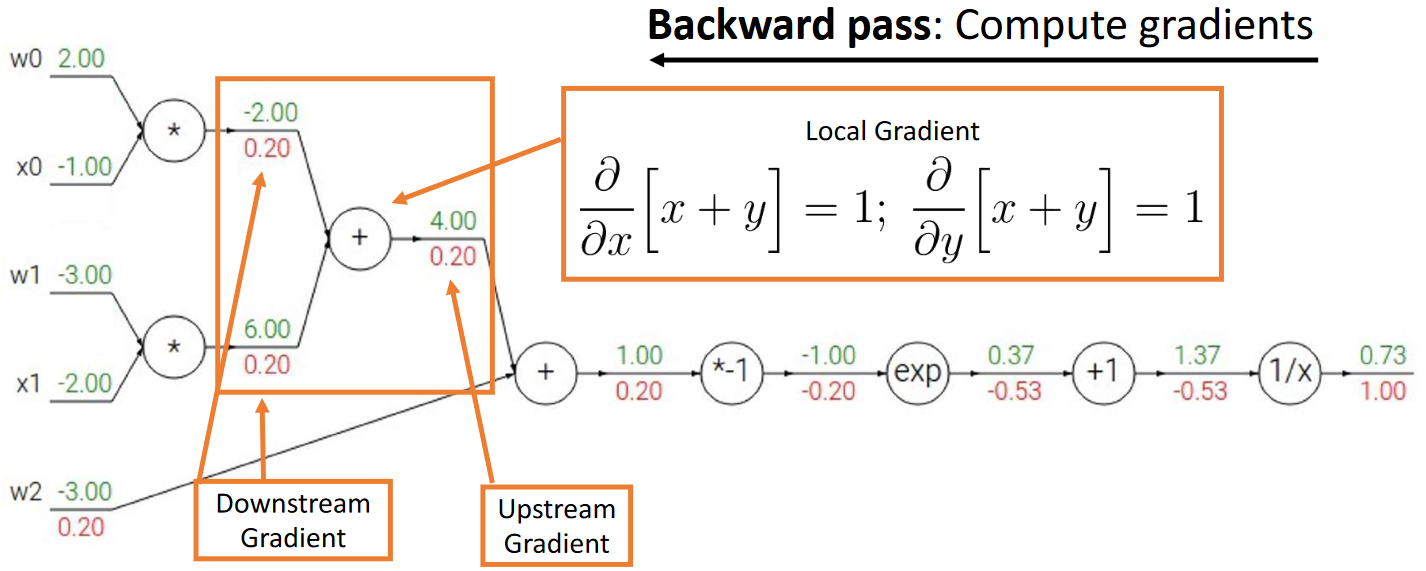


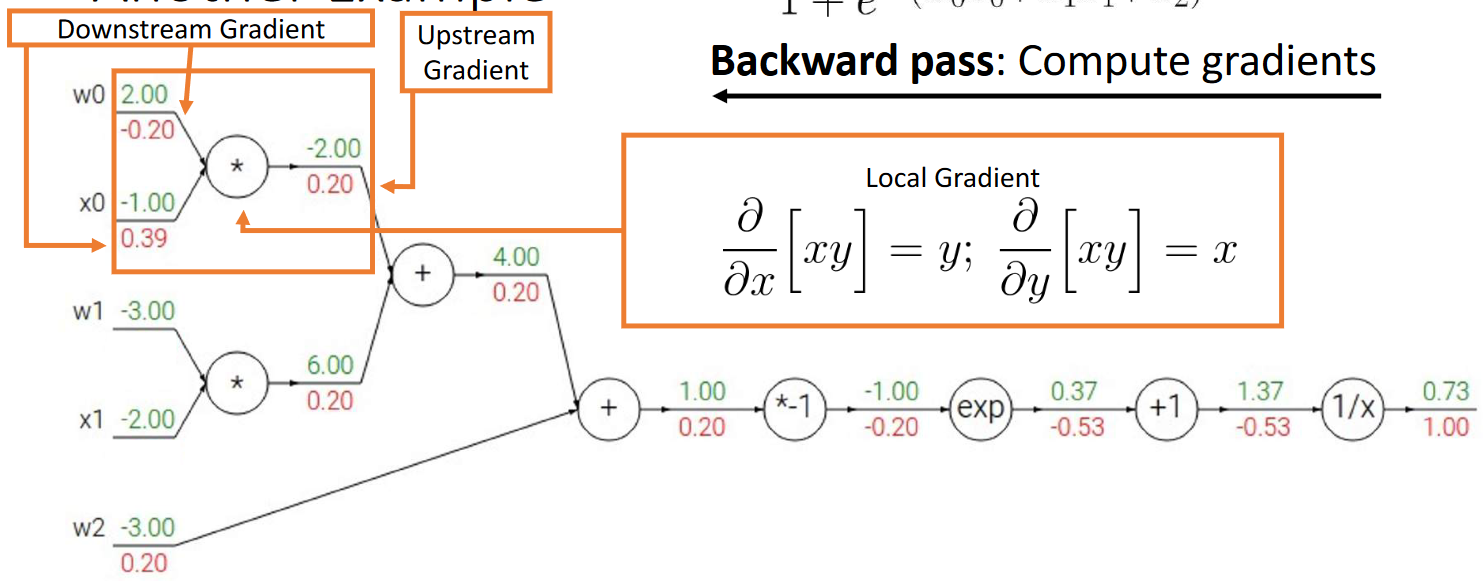


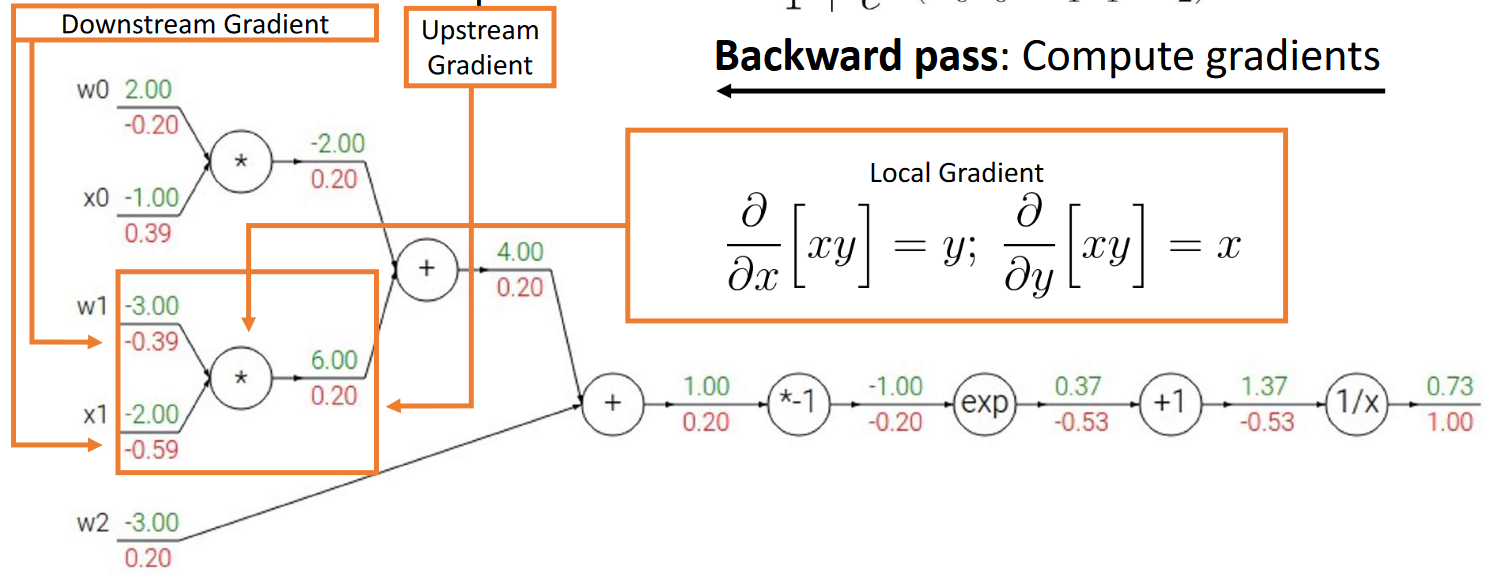




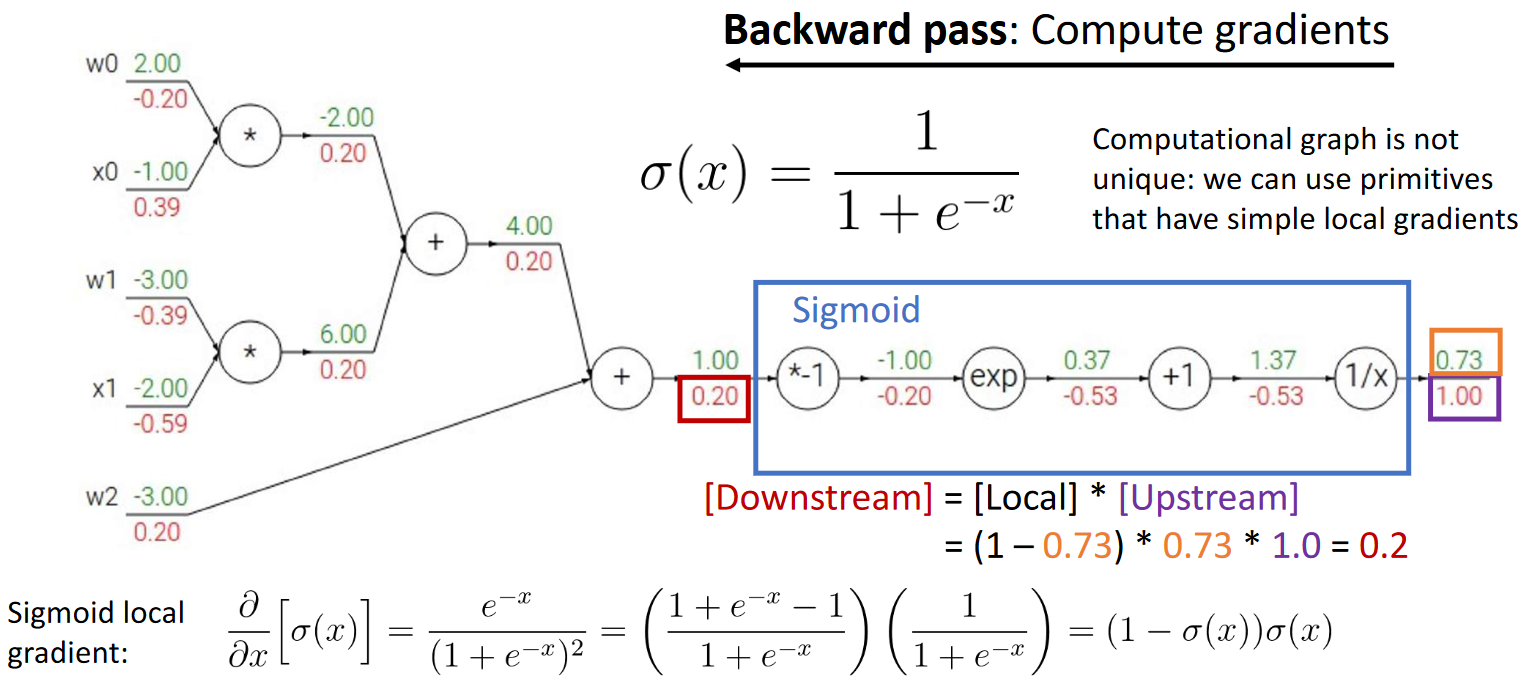








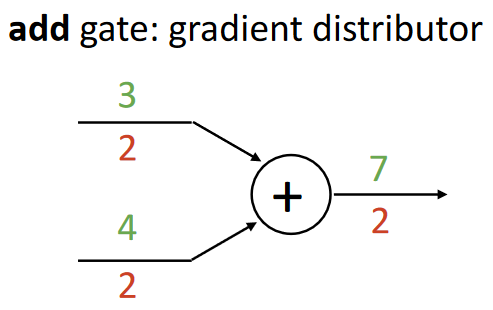
There are cases where we can simplify the computation graph further by replacing parts with a primitive. For example, the equation above is actually for the **sigmoid** function, . We can use this fact to reduce several gradient calculations down to one.



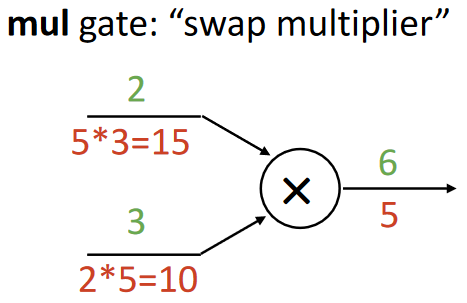
## Patterns in Gradient Flow

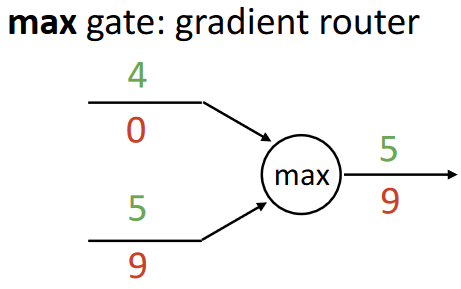
The calculations we see during backpropagation have a few **patterns** which we can utilize to make the calculations even easier.

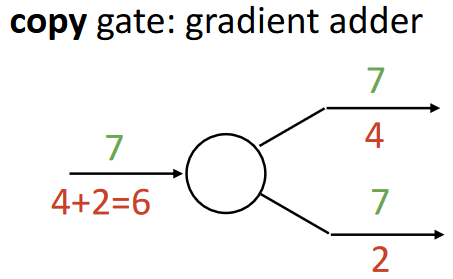
For example, if the forward pass involves **adding** the outputs of multiple neurons, the **downstream gradient** for each neuron will be the same as the **upstream gradient**.



Similarly, we can have other patterns as well:

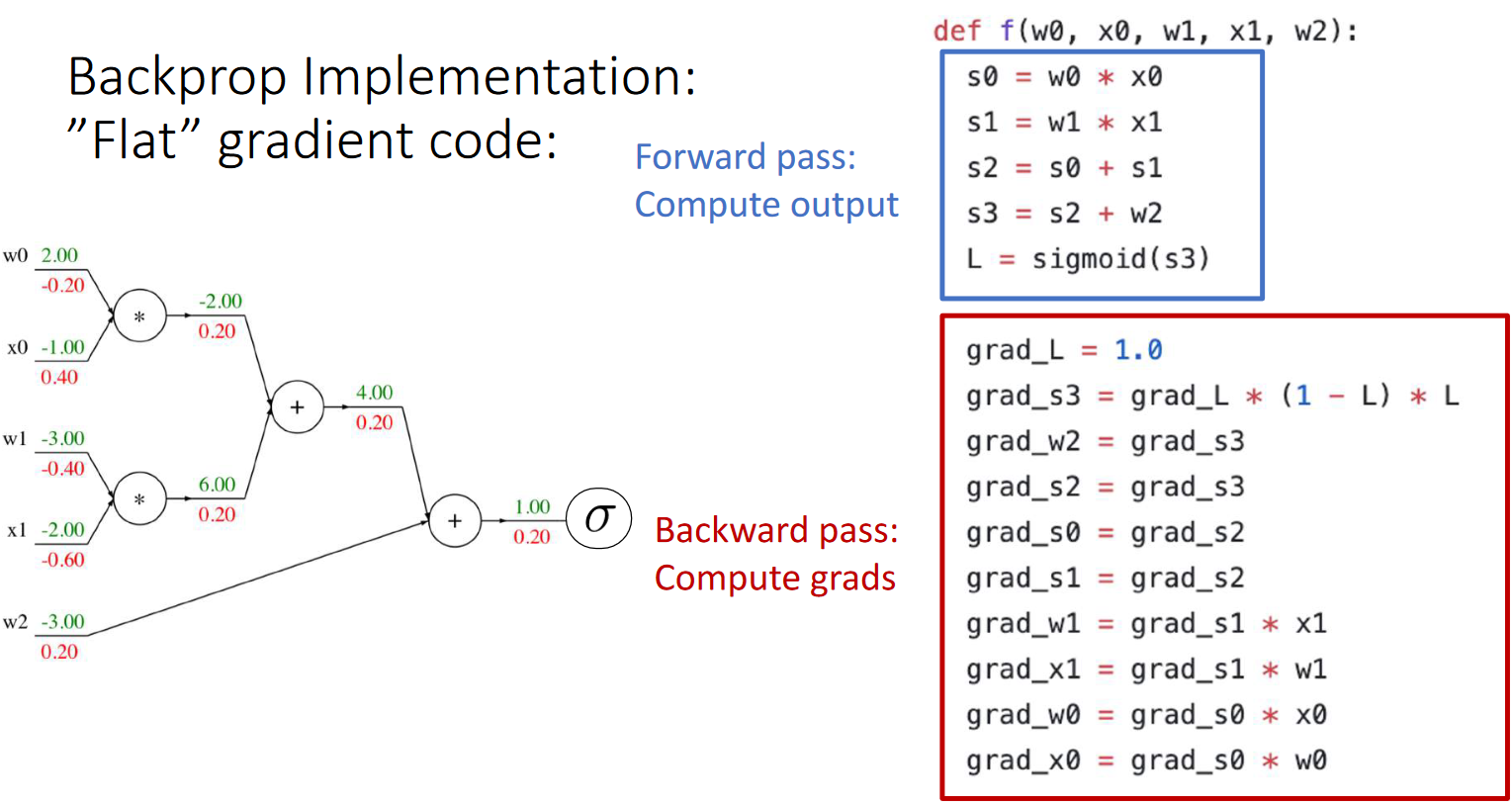


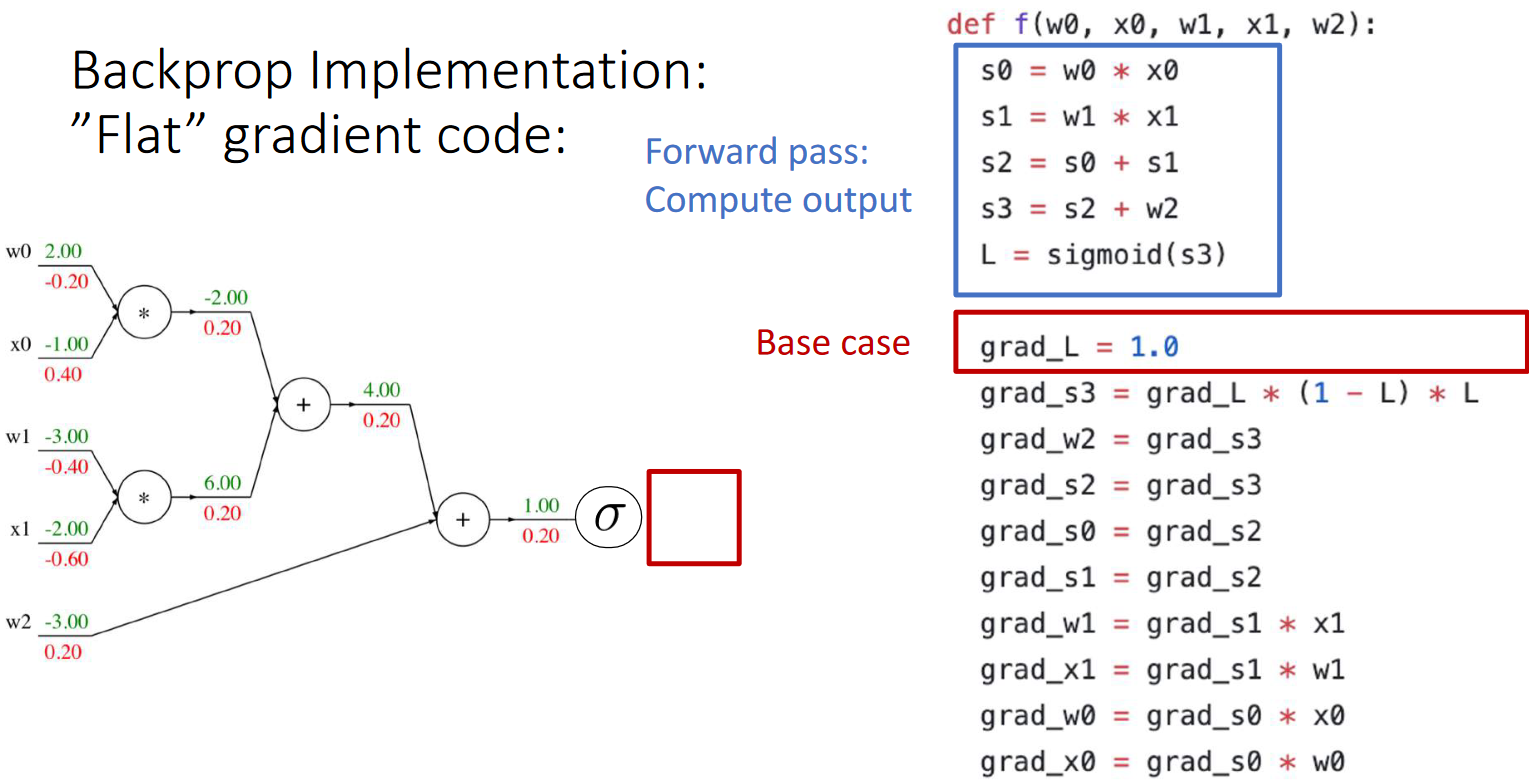


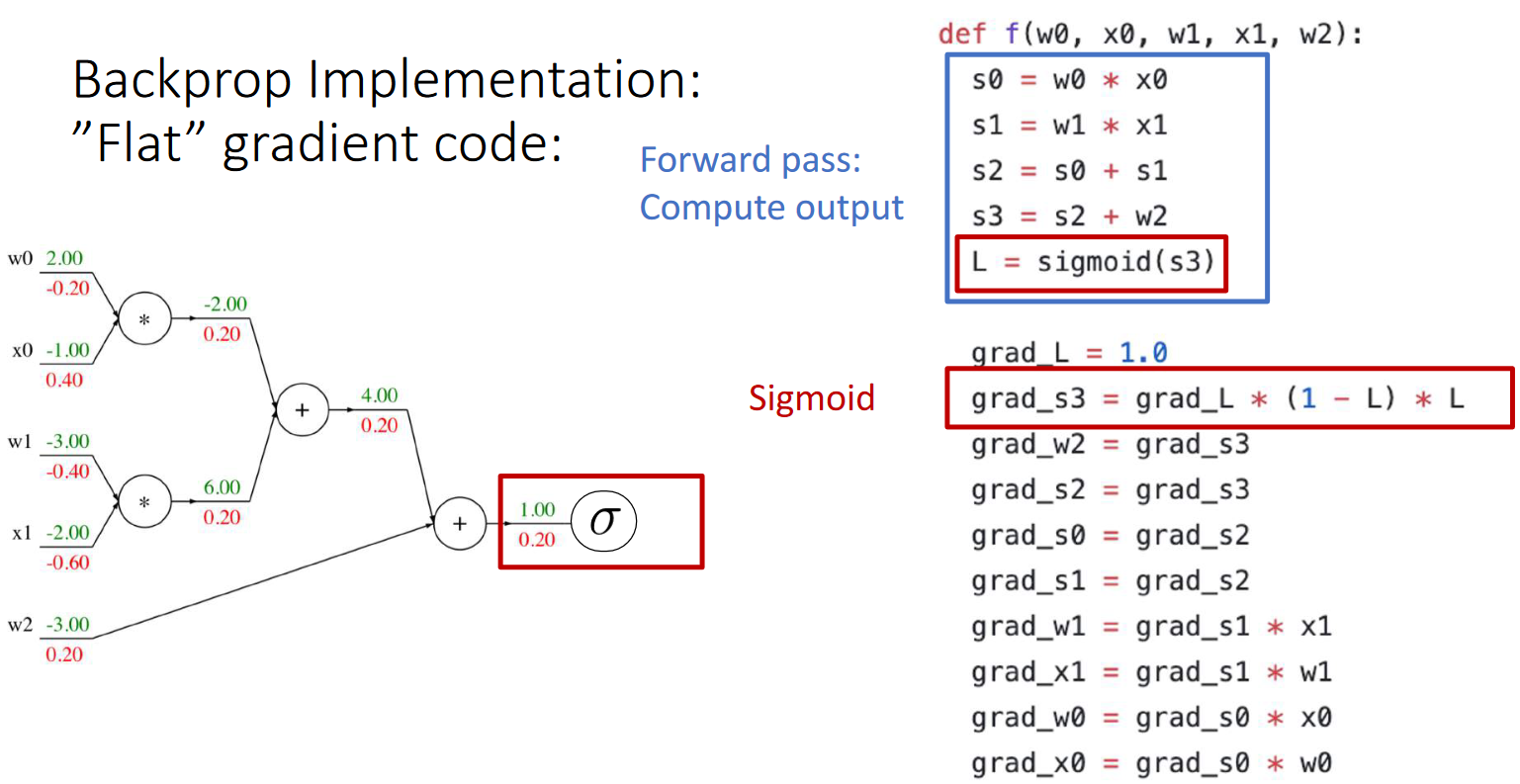


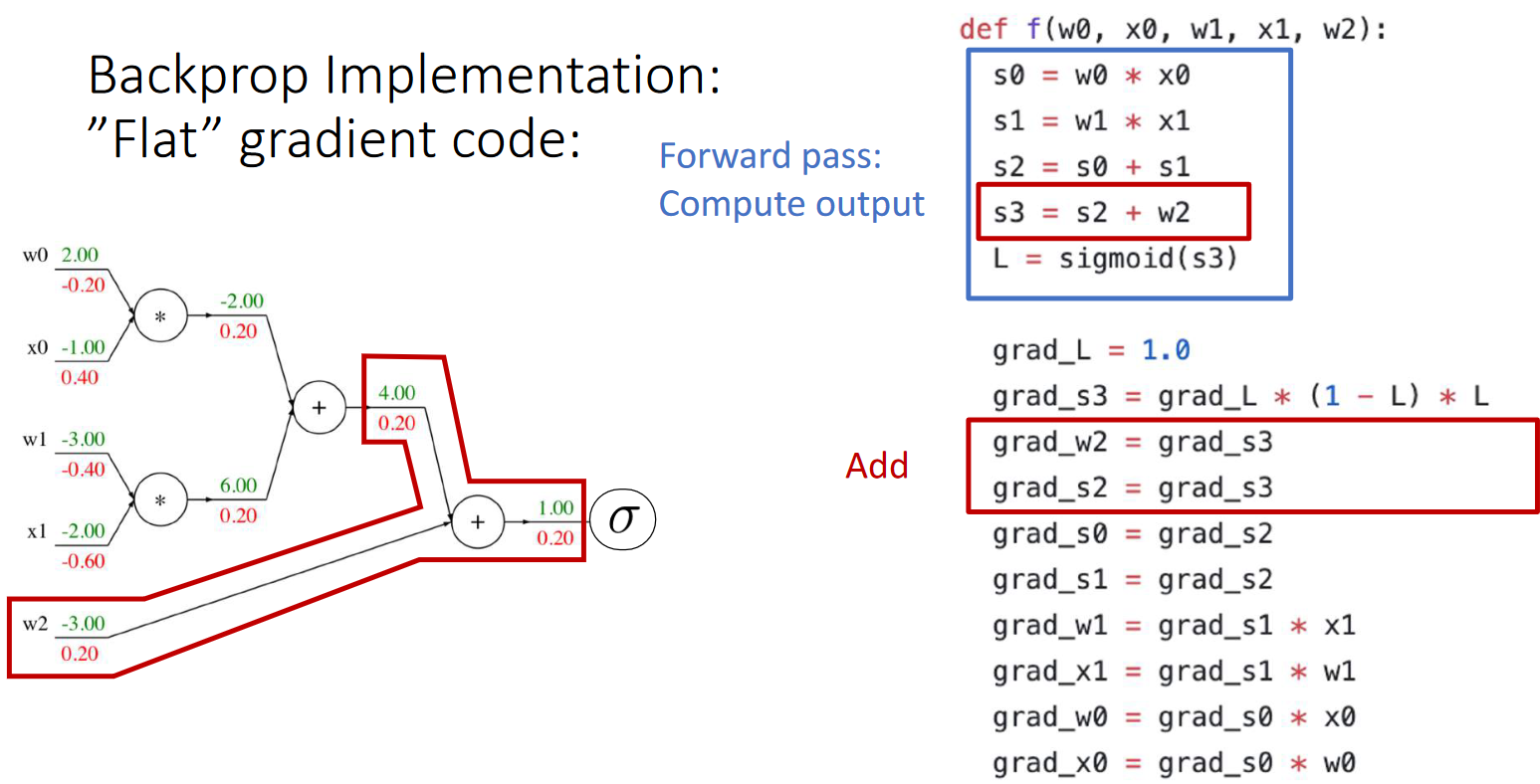
## Backpropagation Implementation

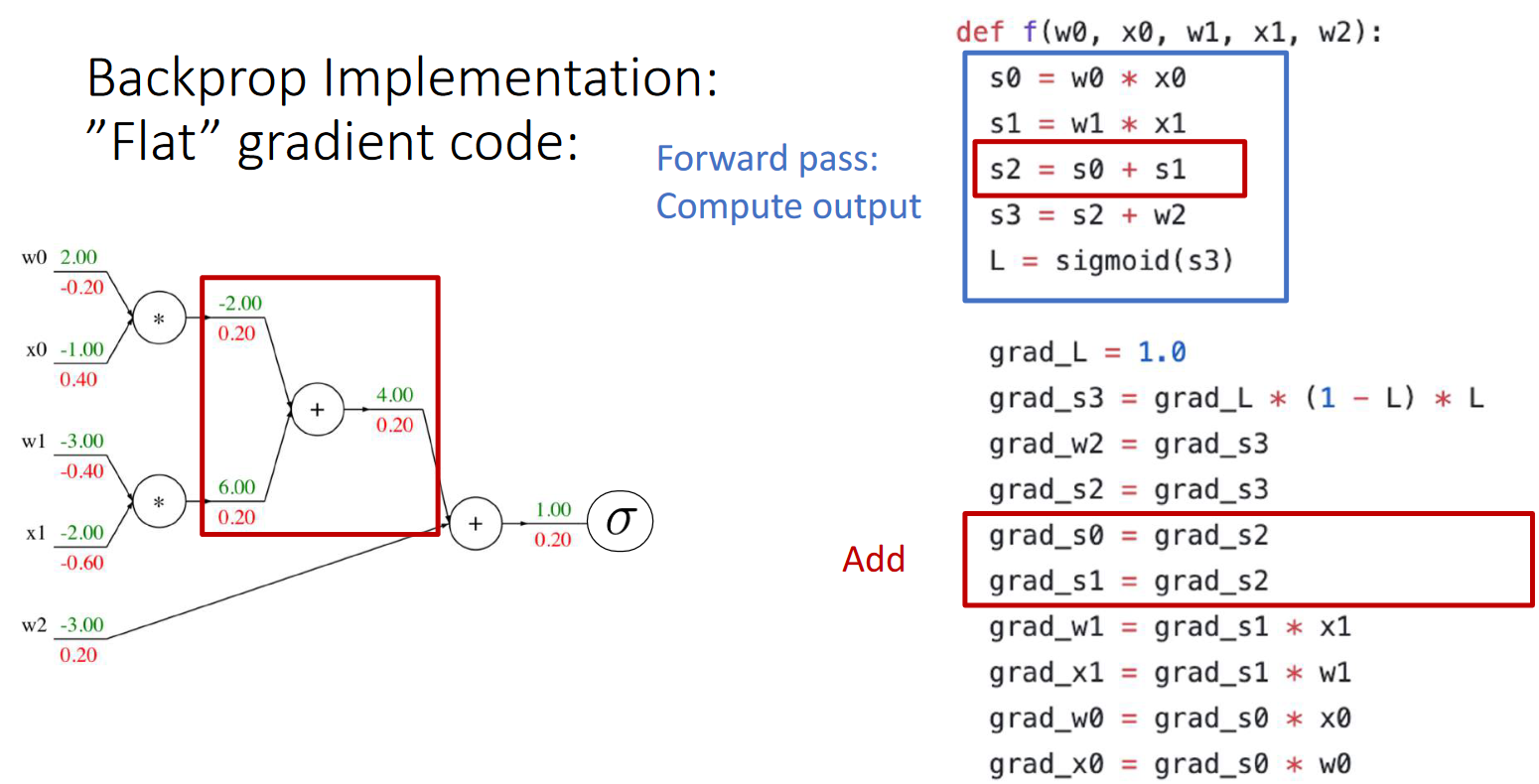
The code for the backwards pass through the network should look like a reversed version of the forward pass. The steps below show this, utilizing both the sigmoid primitive and the patterns we have previous seen.

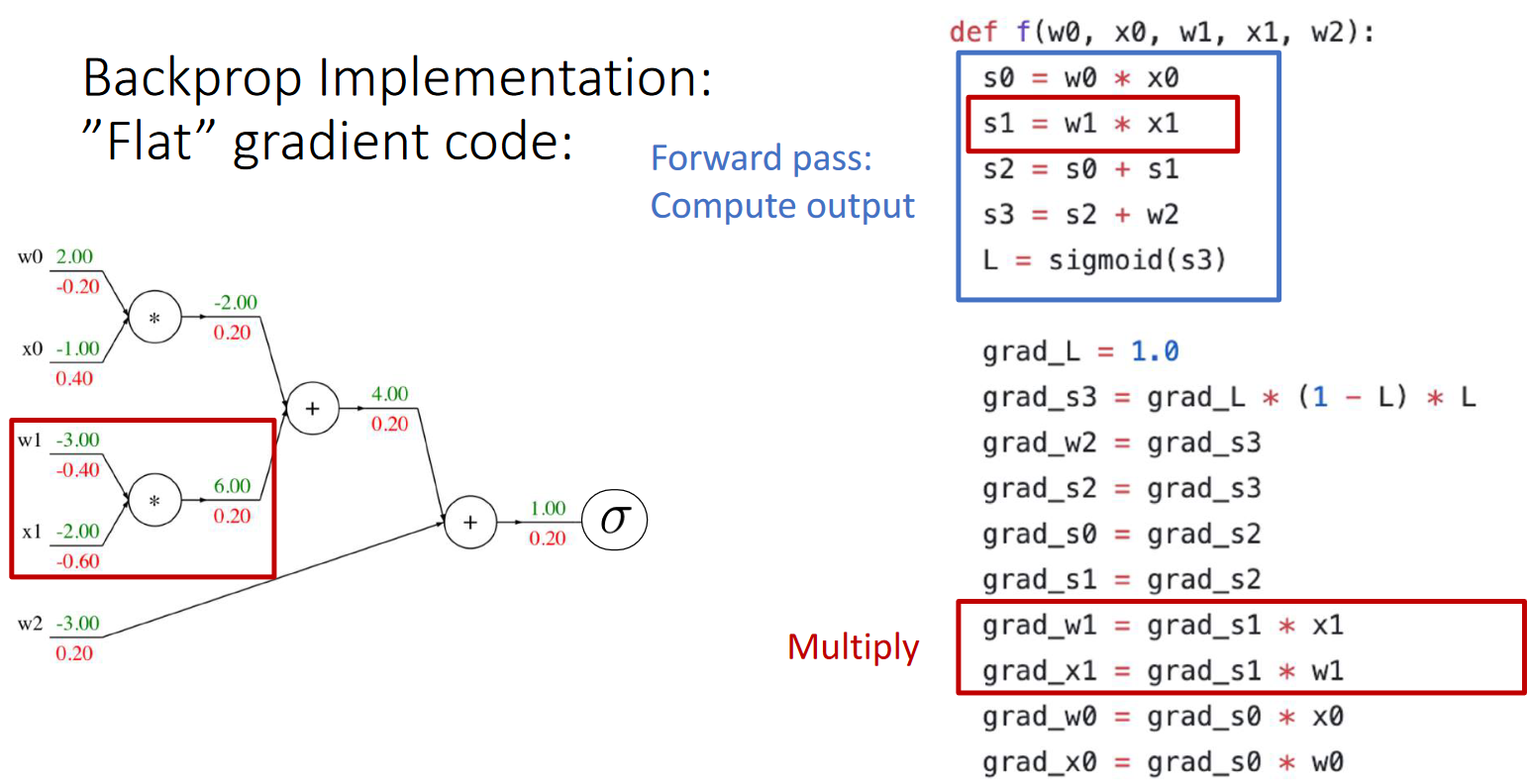


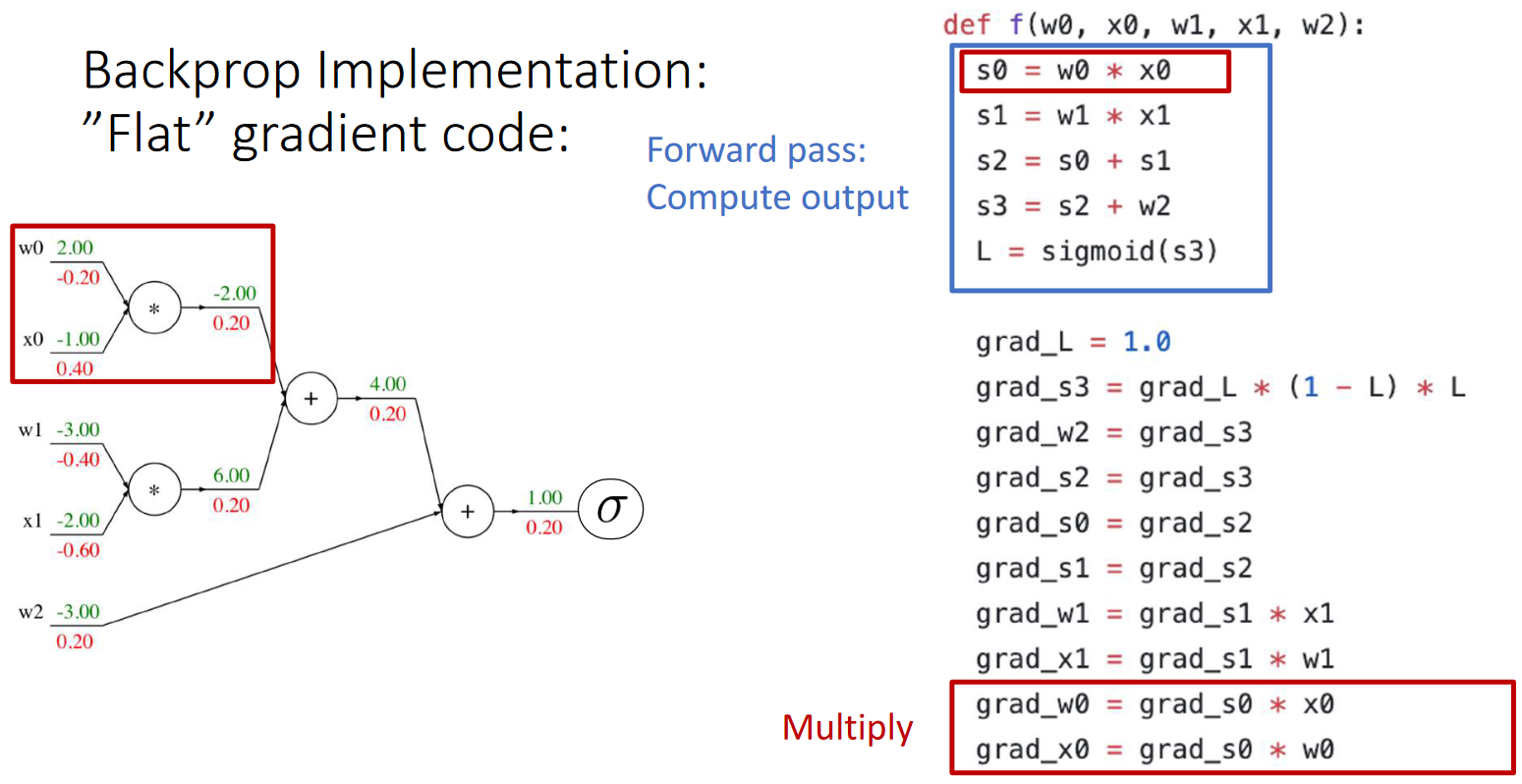












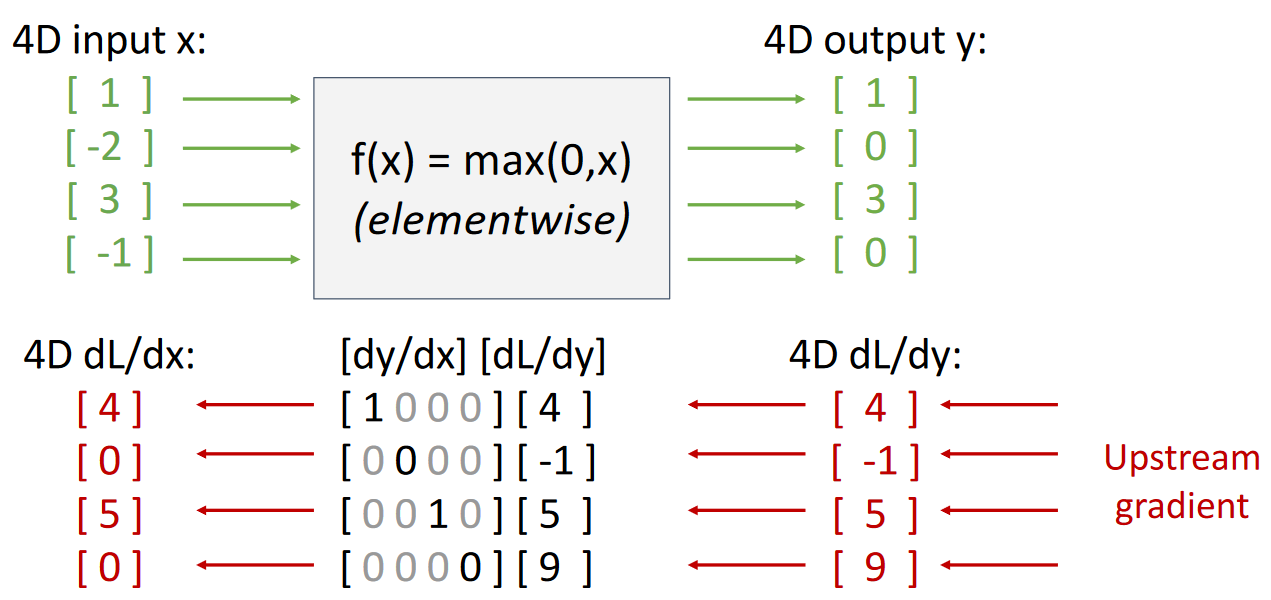
## Vector Derivatives

So far, we have been dealing with scalar values. There is only one output, , and one input has a derivative with reference to it, . However, we are usually dealing with inputs that have multiple units, i.e., a **vector**.

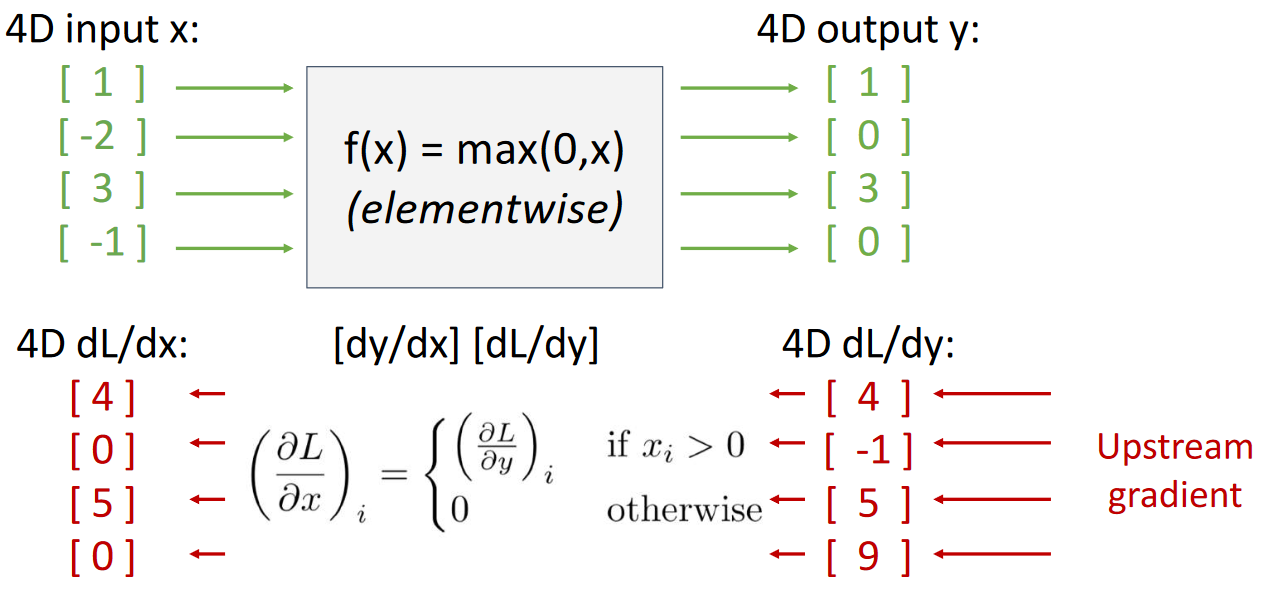
To calculate the derivative for a vector input, we need to recognize that each input unit has some effect on the output value. Thus, there should also be a separate derivative for each input. This means that we will get a **vector of derivatives**.

Similar to the above, of both the input and the output were vectors, we would have a separate derivative for each input-output unit combination. This would create a **matrix of derivatives**, commonly called a **Jacobian Matrix**. Note that this will only happen in the intermediate layers of a network. The find output for a network, , is always a single value.

Generally speaking, we will find that Jacobian matrices occur for the **local gradient**, while vector gradients occur for the **upstream** and **downstream gradients**. An example should make this clear.



Although this seems like it is an extreme example since we are using a max function, the characteristics we are seeing for the Jacobian matrix happens to be true in most cases. The matrix mostly consists of **diagonal entries**, and can get very large very fast. This makes it a poor choice from a memory perspective. Because of this, we never actually explicitly calculate the Jacobian matrix. Instead, we find an **implicit representation** that will give us the same result. For the above case, the implicit representation is as follows:

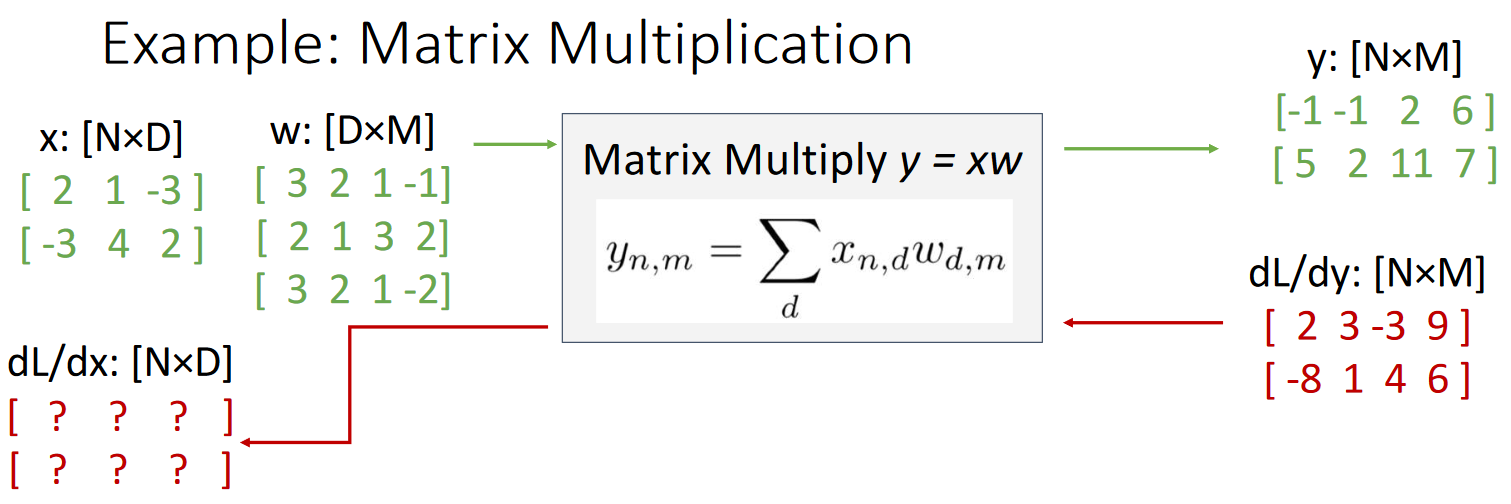


Implicit representations such as this one can be found for every case.

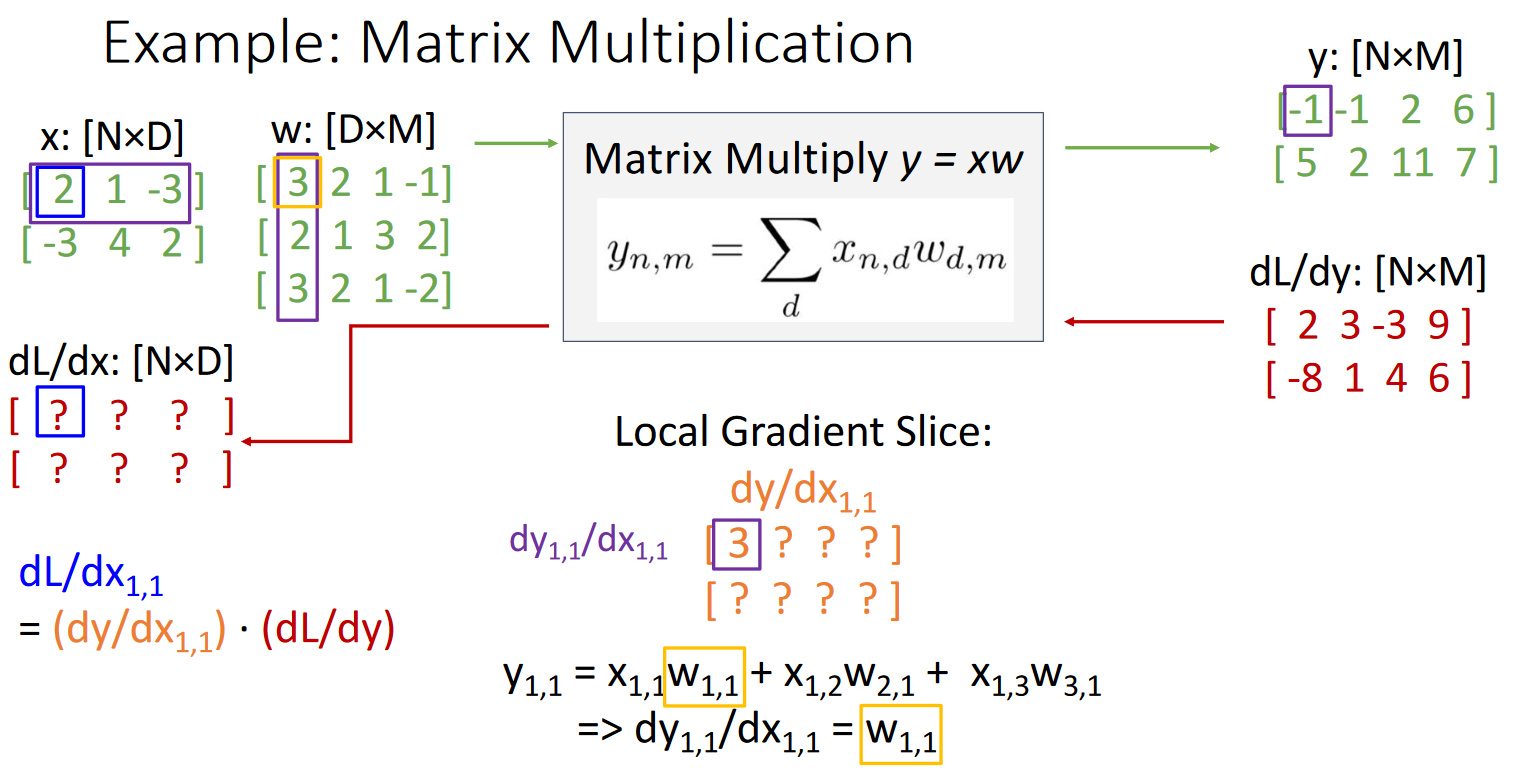
## Matrix Derivatives

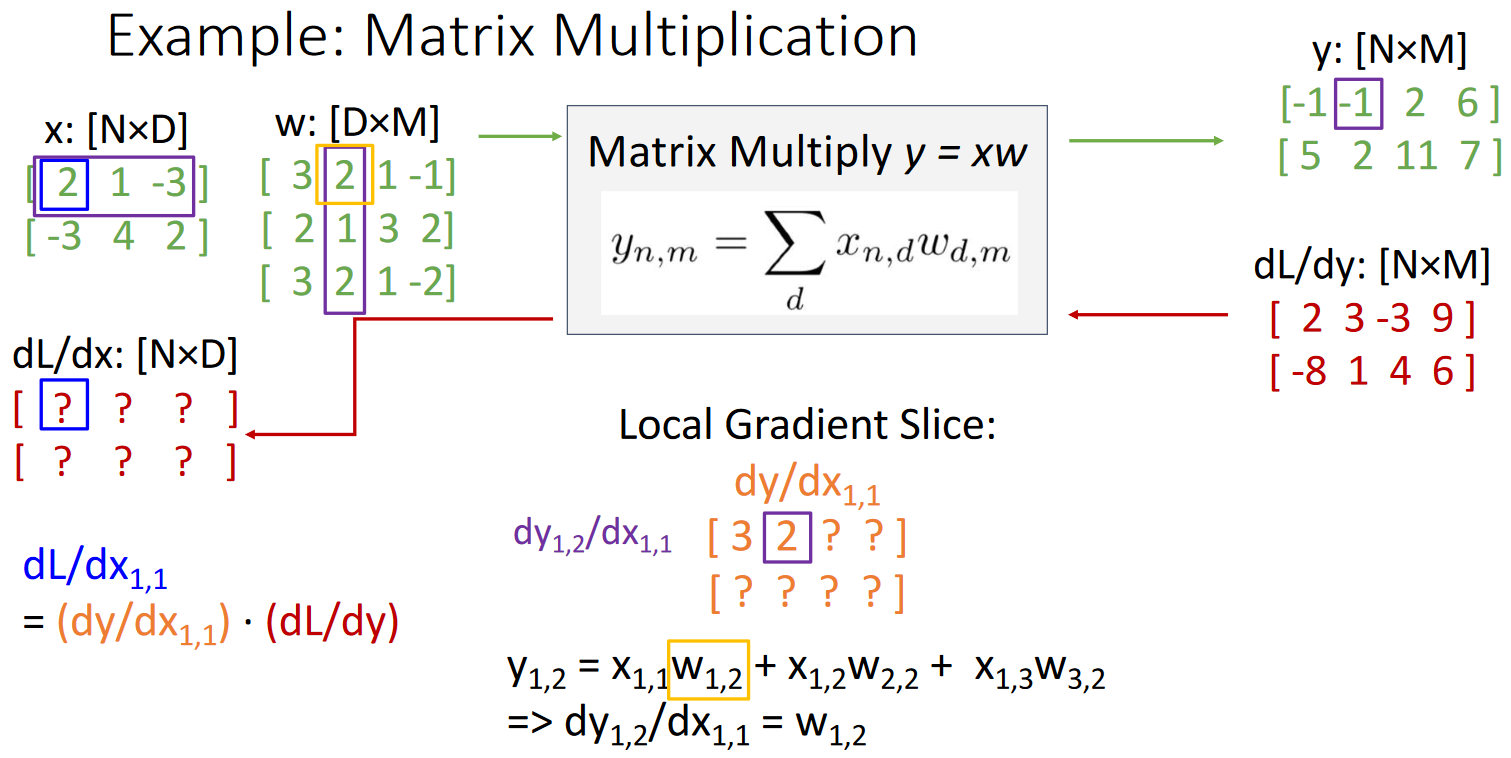
To extend further upon the idea of vector derivatives, what if we have the upstream and downstream gradients as **matrices**? In this case, the local gradient will be a **matrix of matrices**. As before, this can also only occur in an intermediate layer. The loss, , is still a single value.

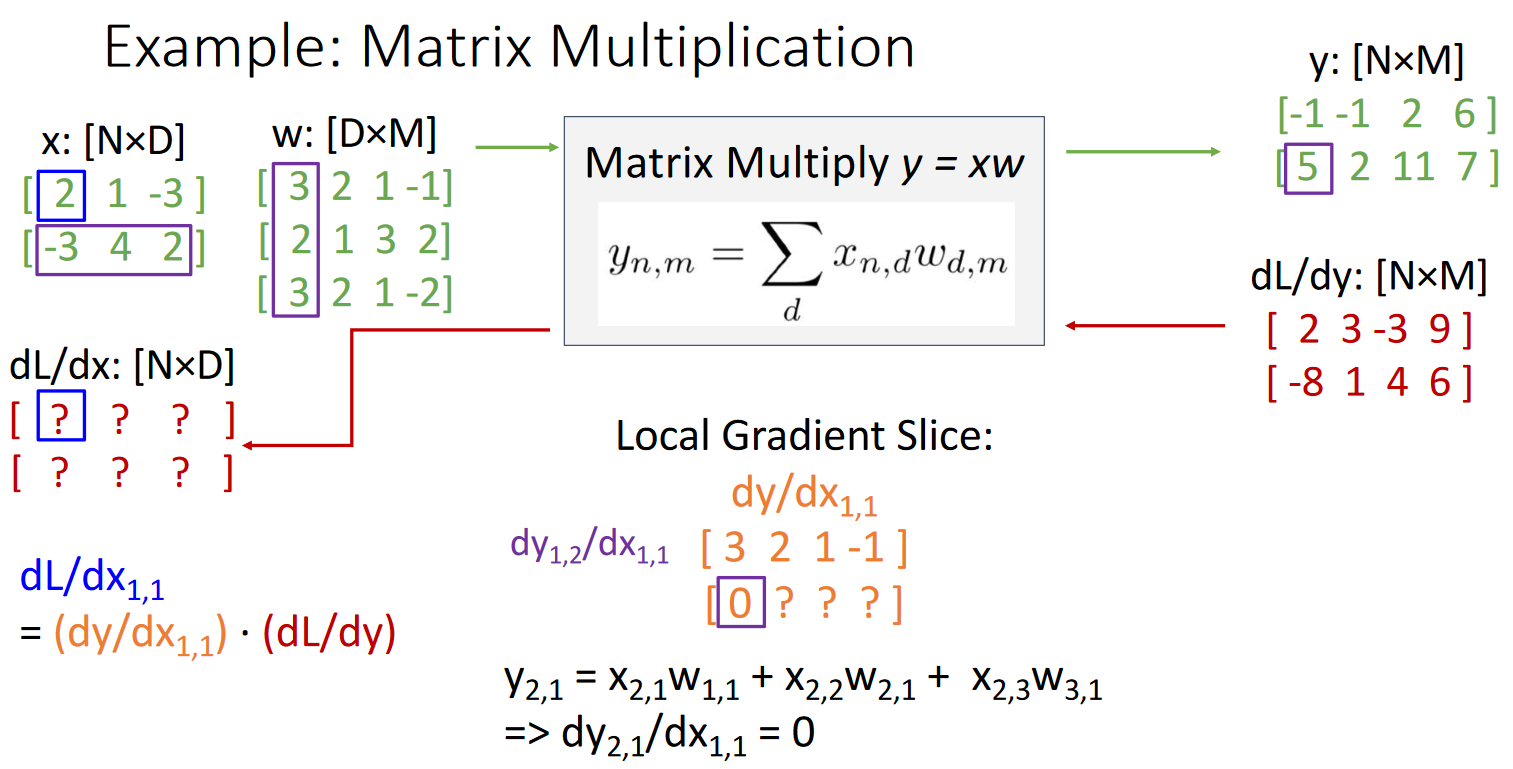
To properly visualize how a matrix of matrices is calculated, we can use an example.

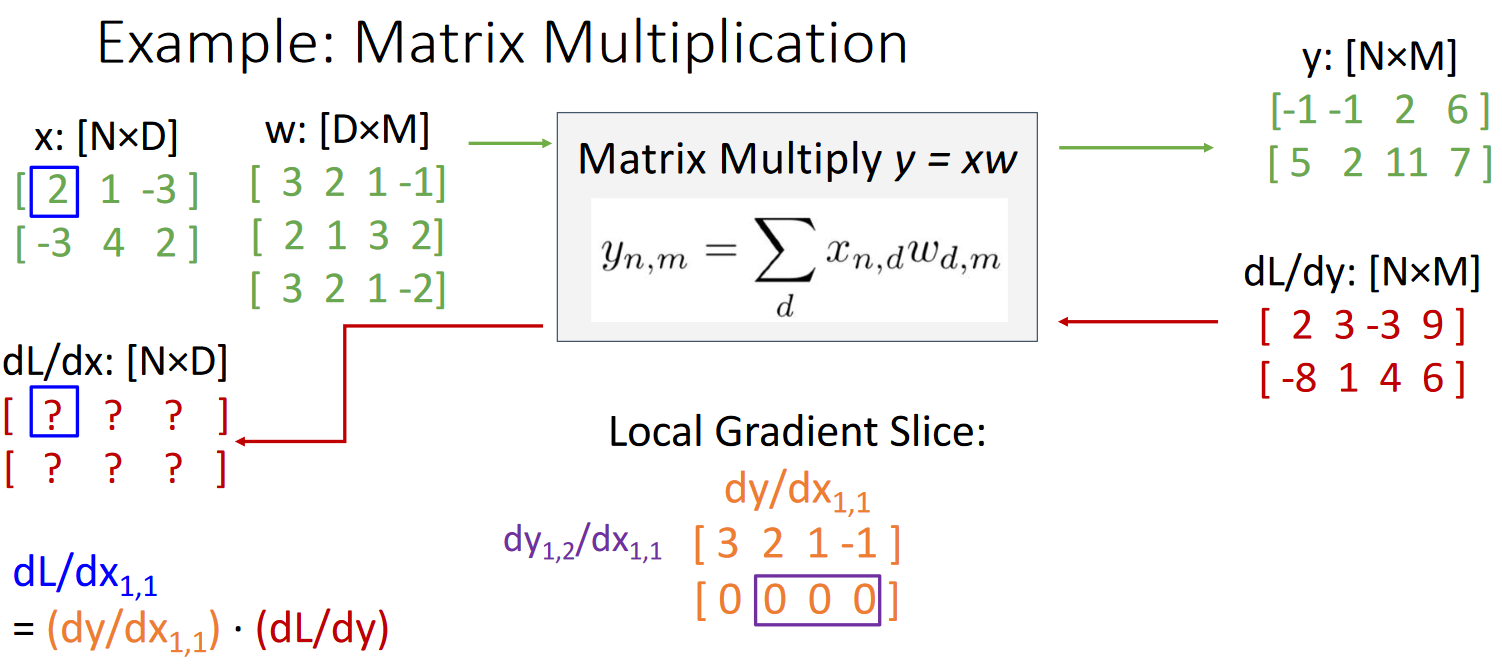


In this case, we need to calculate element-by-element. Each element is itself a **matrix**. For example, the value of depends on how much influence the value had on each of the output units of .



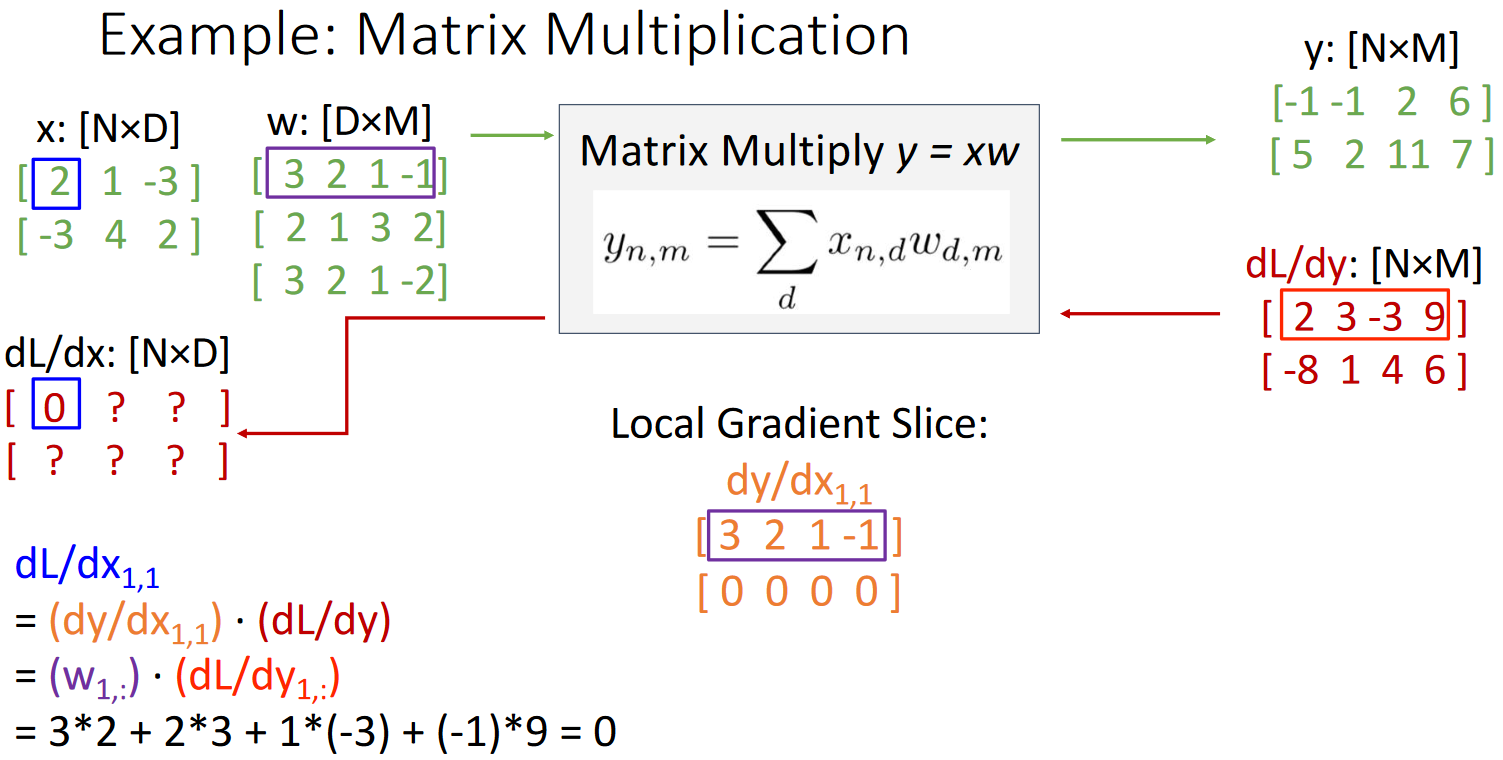






Notice how the second row of is filled with zeroes. This is because does not affect the second row of .

Also notice that there is a pattern. The only values that matter for are (the first row of ) and (the first row of ). We can use this information to again, perform an **implicit calculation**.



The rest of the cells of are filled in a similar manner.

