# 

[**5.1: Backpropagation (Math-heavy/Vectorized)**](#_wig91915sh7u) **1**

[5.1.1: Setting the context](#_v4ho0o67l04x) 1

[5.1.2: Intuition behind backpropagation](#_6davsgljfcsz) 4

[5.1.3: Understanding the dimensions of gradients](#_62zrhesi70sj) 7

[5.1.4: Computing derivatives w.r.t Output Layer](#_ipd3q8ujexyg) 8

[5.1.4a: Part 1](#_3ugagx27gg6s) 8

[5.1.4b: Part 2](#_z1ty945kskcb) 9

[5.1.4c: Part 3](#_sucoajhksfeh) 10

[5.1.5: Quick recap of the story so far](#_d2ku4b4o9owg) 11

[5.1.6: Computing derivatives w.r.t Hidden Layers](#_5trge560qowh) 12

[5.1.6a: Part 1](#_29eural4n5ah) 12

[5.1.6b: Part 2](#_lr8i2i4cweth) 13

[5.1.6c: Part 3](#_z6l1mttsc0f9) 14

[5.1.7: Computing derivatives w.r.t one weight in any layer](#_92j78seg4bsc) 15

[5.1.8: Computing derivatives w.r.t all weights in any layer](#_gup23rcv50mt) 16

[5.1.9: A running example of backpropagation](#_j1vchbbwf1yd) 17

[5.1.10: Summary](#_6al0onltmmah) 19

# 

# Backpropagation (Math-heavy/Vectorized)

## Setting the context

How does this differ from the previous section

1. We’ve looked at two different levels of complexity to the backpropagation algorithm so far
   1. No-math: A simple forward pass with no gradient calculation
   2. Light-math: Gradient calculation for each weight using chain rule
2. Now, with the Heavy-math version, our objective is to identify common calculations between different weights and re-use them to make our work simpler
3. Let us consider the example from the light-math backpropagation chapter
4. Consider dw131 or
   1. Here, we are certain about using the highlighted values for gradient computation. So we can pre-calculate and store them
   2. In the outermost layer

|  |  |  |  |
| --- | --- | --- | --- |
| = |  |  |  |
|  |

1. Similarly, storing the values of will prove useful down the line

## 

## Intuition behind backpropagation

Let us look at an intuitive explanation of backpropagation

1. Let’s have a quick calculus recap
   1. f(x) = x2,
   2. f(x) = x2 + y2,a
   3. f(𝜃) = [x, y],
2. Consider the following sample network
3. Now, let us assume our network has undergone some training, so we have a Loss value in hand.
4. W1, W2, W3, b1, b2, b3 have been updated and reflect accordingly in the Loss function
5. Now, we want to scrutinise how each parameter is responsible for the loss. So we move backwards from the Loss in a step-by-step manner
6. Stepwise calculation (while personifying the neurons and layers 😄)
   1. Step 1: The loss talks to the output layer, saying “You better take responsibility for the poor output!”
   2. Step 2: The output activation layer says, “Hey, I’m simply applying the softmax function to the input given to me by the output preactivation layer”
   3. Step 3: The output preactivation layer says, “I take responsibility for my part, but I am only as good as the hidden layer and the weights below me.” After all
   4. Step 4: The parameters W3 and b3 say “It is our mistake, **please update our values**”
   5. Step 5: However, the second hidden activation layer says “Hey, I’m simply applying the sigmoid function to the input given to me by the second hidden preactivation layer”
   6. Step 6: The second hidden preactivation layer says, “I am only as good as the hidden layer and weights below me”
   7. Step 7: The parameters W2 and b3 say “It is our mistake, **please update our values**”
   8. Step 8: However, the first hidden activation layer says “Hey, I’m simply applying the sigmoid function to the input given to me by the first hidden preactivation layer”
   9. Step 9: The first hidden preactivation layer says, “I am only as good as the weights below me, we cannot blame the input layer.”
   10. Step 10: The last set of parameters W1 and b1 say “It is our mistake, please update us”
7. Thus, to arrive at the derivative of the Loss function w.r.t any of the weights, we must proceed downwards from the top. We cannot simply calculate it without knowing the preceding values.
8. Our roadmap for the rest of the module
   1. To calculate the desired gradient, we need to compute
   2. Gradient w.r.t output units
   3. Gradient w.r.t hidden units
   4. Gradient w.r.t weights and biases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Talk to the weight directly | Talk to the output layer | Talk to the previous hidden layer | Talk to the previous hidden layer | Talk to the weights |
|  |  | works for any number of output layers | |  |

* 1. Our aim is to do these calculations for any of the possible weights using notation i, j, k instead of numbers
  2. For the rest of this exercise, our focus is on *Cross Entropy loss*  and *Softmax* output.

1. To reduce the tediousness of applying the chain rule each time to get the desired gradient, we will look to re-use common values and pathways to more efficiently calculate any gradient.

## 

## Understanding the dimensions of gradients

What are we interested in?

1. Consider the backpropagation illustration from the previous section
2. What we are interested in is a(where true output y = 1, L = Layer number, *l* is the index of the correct class-label for the given input, and i is the neuron number)
3. We know that is dependent on a31 and a32
4. Therefore, the derivative at the output layer

|  |  |  |  |
| --- | --- | --- | --- |
| = |  |  |  |
|  |

1. In the above gradient, L = 3 and i ∊ {1, 2}
2. Henceforth, we can use these notations in place of numbers to simplify gradient calculation for all possible gradients. ŷ

## Computing derivatives w.r.t Output Layer

### Part 1

The first derivative in the chain

1. What we are actually interested in is:
   1. Where L = layer number, i = neuron (from 1 to k), l = index of correct output
   2. Here, we use the cross entropy loss function
   3. In the output layer L, assume we have neurons aL1, aL2 … aLk
   4. The output layer L involves applying the softmax function the all the neurons
   5. again, (l refers to the index of the correct output neuron)
   6. Thus, depends on all the neurons’ outputs as they all appear in the denominator, thereby making the derivative non-zero for all the output neurons
2. From the previous points, we know that depends on
3. The first part of the derivative is fairly straightforward (of the form )

### 

### Part 2

1. Continuing from where we left off
2. Here, we know that (taking the l-th entry of the softmax function applied to vector aL)
3. where aL = [aL1, aL2 … aLk]
   1. Where
   2. Selecting the l-th entry would give us the value
   3. This is of the form
   4. Here
   5. Substitute the values and expand the formula
   6. Here, consider , this value is 0 for all values of i : 0 to k except for when i = *l*
   7. Thus, we use an indicator variable to denote that all other values except i=*l* resolve to 0
   8. Now consider , here i’ ranges from 1 to k. When taking the derivative, only the index i=i’ remains, which is simply a derivative of an exponent.
   9. This is can be rewritten in terms of the softmax function for the different variables
   10. We know that the Softmax function is ŷ, so we rewrite it.
4. After cancellation

### 

### Part 3

1. So far, we have derived the partial derivative with respect to the *i*-th element of layer aL
2. We can now write the gradient w.r.t the vector aL
3. As we saw earlier, aL = [aL1, aL2 … aLk]
4. Going by the indicator variable in step 1, it resolves to 0 for all values of *i* except for *i* = *l*
5. Let us assume a scenario where k = 4, and *l* = 2
   1. Here, the indicator variable values as a vector would be
6. The gradient w.r.t a*L* is

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| .  .  . |
|  |
|  |  |  |  |
| .  .  . |
|  |

1. The above can be seen as a difference of two vectors, [0, 1, 0, … 0k] and ŷ
2. The first vector is essentially the one hot representation of the true output e(*l*):
3. In reality, this is simply the difference between the true distribution y and the predicted distribution ŷ

## Quick recap of the story so far

What have we covered up till this point?

1. Our roadmap for this module
   1. To calculate the desired gradient, we need to compute
   2. Gradient w.r.t output units
   3. Gradient w.r.t hidden units
   4. Gradient w.r.t weights and biases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Talk to the weight directly | Talk to the output layer | Talk to the previous hidden layer | Talk to the previous hidden layer | Talk to the weights |
|  |  | works for any number of output layers | |  |

* 1. For the rest of this exercise, our focus is on *Cross Entropy loss*  and *Softmax* output.

1. Here, what the sections highlighted in green are what we have covered so far, i.e. the derivative with respect to the last layer aL
2. The gradient was calculated to be

## 

## Computing derivatives w.r.t Hidden Layers

### Part 1

The derivatives corresponding to the hidden layers

1. What we are interested in is
   1. This formula is the summation of all the paths that lead from the concerned neuron to the loss function
   2. Here, *i* = layer number, *m* = neuron number for a, *j* = neuron number for h
   3. From the previous section, we already know how to compute so we need to only focus on
   4. However, when we compute the derivative of the neuron ai+1, m w.r.t hi,j we are left with the weight component Wi+1, m, j
   5. This refers to the weight component between the output neuron(ai+1, m) and input neuron (hi,j)
2. Thus we have
3. Now consider these two vectors

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| .  .  . |
|  |
|  |  |  |  |
| .  .  . |
|  |

* 1. Here, refers to the gradient vector of the loss function w.r.t to all output neurons from ai+1,1 to ai+1,k
  2. And refers to all rows of the *j*-th column of the Wi+1 matrix, ie a vector.

1. The dot product of these two vectors is
2. Here, the RHS is the same as the value from step 2. Therefore, the derivative of the loss function with respect to the hidden layers is the dot-product between the gradient of loss w.r.t output layer and the corresponding weights.

### 

### Part 2

1. We have
   1. This is with respect to one neuron
   2. We would like to speed up this computation by solving all the derivatives in one go
2. We can now write the gradient w.r.t hi

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | |  |
| .  .  . | |
|  | |
|  |  |  | |  |
| .  .  . | |
|  | |

* 1. Can be written more compactly as

1. Thus, the formula for gradient of loss function for the last hidden layer before the output layer is given by
2. This calculates the gradient w.r.t all neurons of layer *i*. It uses simple matrix-vector multiplication to achieve this.
3. Now, we have seen a special case applied to the last hidden layer. We must figure out how to make this formula applicable for any generic hidden layer.

### 

### Part 3

1. Consider the next layer ai
   1. The first derivative is what we computed in part 2
   2. We need to compute the second derivative
   3. We know that hij is simply the application of an activation function (sigmoid, tanh etc) to aij
   4. So it can be rewritten as where hij = g(aij) and g’(aij) is its derivative
2. The full gradient can be written as

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | |  |
| .  .  . | |
|  | |

* 1. This vector is the element-wise product of two vectors and [...,g’(aik),...] (which is a vector of derivations of the activation function w.r.t the pre-activation layer. They are both vectors of n-terms

1. Thus (⊙ refers to element-wise multiplication)
2. This formula can be applied to any of the hidden layers

## 

## Computing derivatives w.r.t one weight in any layer

1. Our roadmap for this module
   1. To calculate the desired gradient, we need to compute
   2. Gradient w.r.t output units
   3. Gradient w.r.t hidden units
   4. Gradient w.r.t weights and biases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Talk to the weight directly | Talk to the output layer | Talk to the previous hidden layer | Talk to the previous hidden layer | Talk to the weights |
|  |  | works for any number of output layers | |  |

* 1. For the rest of this exercise, our focus is on *Cross Entropy loss*  and *Softmax* output.

1. Here, what the sections highlighted in green are what we have covered so far, i.e. the derivative with respect to the last hidden layer and all other subsequent hidden layers
2. The gradients were calculated to be
3. Now, we will be computing the derivative of the loss function w.r.t weights and biases.
4. Recall that ak = bk + Wkhk-1
   1. k = layer number
   2. i = current layer neuron number
   3. j = previous/input layer neuron number
5. Now
6. We can use this to update the Weight by Gradient Descent
8. In the next step, we will look at updating all the weights in a layer simultaneously.

## 

## Computing derivatives w.r.t all weights in any layer

1. Let’s take a simple example of a Wk ∊ ℝ3x3 and see what each entry looks like

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | |  | |  | |  |  |
|  | |  | |  | |
|  | |  | |  | |
|  |  |  | |  | |  | |  |  |
|  | |  | |  | |
|  | |  | |  | |

1. Thus we can update all the weights in one go using a
2. Finally coming to the biases,
3. We can now write the gradient w.r.t the vector bk

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | |  | = |
| .  .  . | |
|  | |

1. Thus, we can update all biases using a

## A running example of backpropagation

The full story so far

1. Consider the following sample Neural Network 
2. Steps to implement Backpropagation
   1. Randomly initialise W and b
   2. Forward propagation
      1. **For k = 1 to L-1 do**

ak = bk + Wkhk-1

hk = g(ak)

* + 1. **end**
    2. aL = bL + WLhL-1;
    3. ŷ = O(aL)
  1. Backpropagation
     1. //Compute output gradient
     2. **For k = L to 1 do**

//Compute gradients w.r.t parameters

//Compute gradients w.r.t layer below

/// Compute gradients w.r.t layer below (pre-activation)

* + 1. **end**

1. Sample calculations to be added at a later date. Δyn/Δxn

## 

## Summary

What are the new things we’ve learned in this module

1. Here are some of the takeaways from this chapter
   1. Data: Real inputs ℝ
   2. Task:
      1. Binary classification
      2. Multi-class classification
      3. Regression
   3. Model: Deep Neural Network to deal with complex decision boundaries
   4. Loss:
      1. Cross entropy loss:
      2. Square Error Loss:
   5. Learning: Gradient Descent with backpropagation
   6. Evaluation:
      1. Accuracy =
      2. Per-class Accuracy =
2. Topics highlighted in red are to be covered in future segments