Introduction to Artificial Neural Networks 1

Dr. Mahmoud Nabil
mnmahmoud@ncat.edu

North Carolina A & T State University

November 2, 2022

Outline

Introduction

- One of the oldest and one of the newest machine learning models.

Introduction

- One of the oldest and one of the newest machine learning models.
- Goes back to 1940, when people started to build modles the imitate the human brain.

Introduction

- One of the oldest and one of the newest machine learning models.
- Goes back to 1940, when people started to build modles the imitate the human brain.
- Logistic regression (perceptron) is the core of neural networks started in 1950.

Introduction

- One of the oldest and one of the newest machine learning models.
- Goes back to 1940, when people started to build modles the imitate the human brain.
- Logistic regression (perceptron) is the core of neural networks started in 1950.
- However, scientists in that time showed that a single perceptron can not solve xor problem (died).

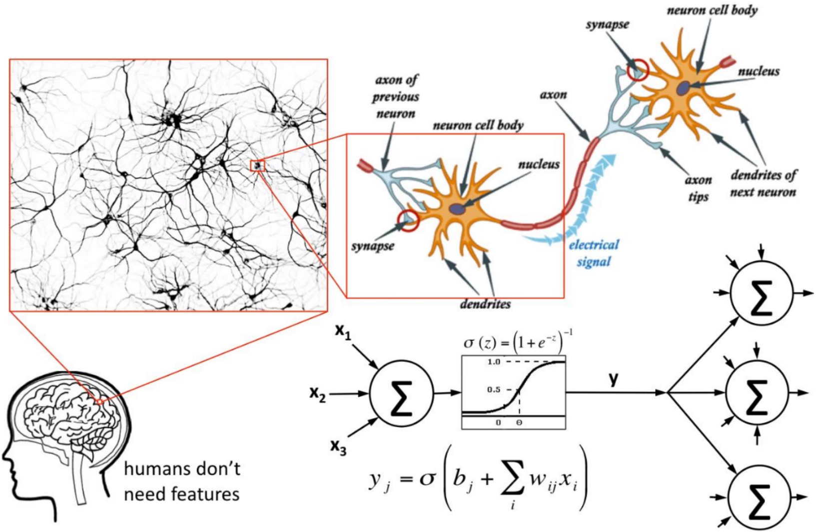
Introduction

- One of the oldest and one of the newest machine learning models.
- Goes back to 1940, when people started to build modles the imitate the human brain.
- Logistic regression (perceptron) is the core of neural networks started in 1950.
- However, scientists in that time showed that a single perceptron can not solve xor problem (died).
- Reborn in 1980, discovery of merging perceptrons together. But died due to the resources requirements

Introduction

- One of the oldest and one of the newest machine learning models.
- Goes back to 1940, when people started to build modles the imitate the human brain.
- Logistic regression (perceptron) is the core of neural networks started in 1950.
- However, scientists in that time showed that a single perceptron can not solve xor problem (died).
- Reborn in 1980, discovery of merging perceptrons together. But died due to the resources requirements
- Reborn in the last decade with the advancement of the computation resouces.

Neurons and the brain

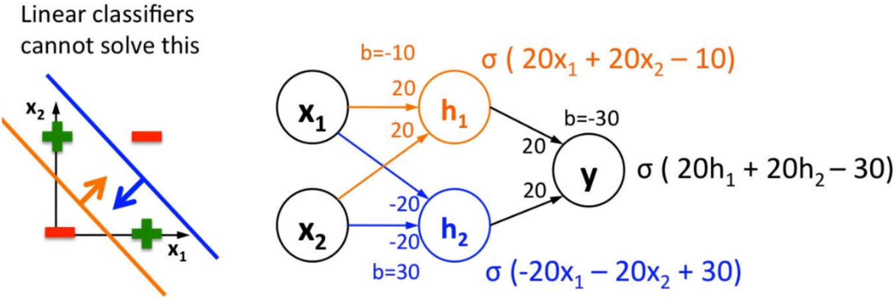


Types of Layers

1. The input layer
   - Introduces input values into the network.
   - No activation function or other processing.
2. The hidden layer(s)
   - Perform classification of features
   - Two hidden layers are sufficient to solve any problem
3. The output layer
   - Functionally just like the hidden layers
   - Outputs are passed on to the world outside the neural network.

$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 \times \Delta}$$

Solving XOR with a Neural Network



$\sigma(20*0 + 20*0 - 10) \approx 0$	$\sigma(-20*0 - 20*0 + 30) \approx 1$	$\sigma(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$	$\sigma(-20*1 - 20*1 + 30) \approx 0$	$\sigma(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*0 + 20*1 - 10) \approx 1$	$\sigma(-20*0 - 20*1 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$	$\sigma(-20*1 - 20*0 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$

Deep Learning Definition

A deep learning model is a computational graph that try to map inputs, each drawn from some dataset with common characteristics to outputs drawn from a related distribution.

Outline

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
  - Functions

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
  - Functions
  - Derivatives

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
  - Functions
  - Derivatives
  - Matrix multiplications

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
    - Functions
    - Derivatives
    - Matrix multiplications
- We'll systematically describe each concept we introduce from three perspectives:

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
  - Functions
  - Derivatives
  - Matrix multiplications
- We'll systematically describe each concept we introduce from three perspectives:
  - Math

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
    - Functions
    - Derivatives
    - Matrix multiplications
- We'll systematically describe each concept we introduce from three perspectives:
    - Math
    - Code

Introduction

- The aim of this part is to explain some foundational mental models that are essential for understanding how neural networks work.
- We will use basic building blocks to build neural networks.
- These blocks are
  - Functions
  - Derivatives
  - Matrix multiplications
- We'll systematically describe each concept we introduce from three perspectives:
  - Math
  - Code
  - A diagram

Outline

Functions

Math.

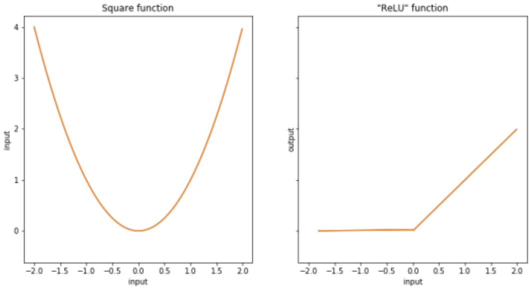
- $f_1(x) = x^2$
- $f_2(x) = max(x, 0)$

This notation says that the functions, which we arbitrarily call f_1 and f_2 , take in a number x as input and transform it into either x^2 (in the first case) or $max(x, 0)$ (in the second case)

Functions

Diagrams One way of depicting functions is to:

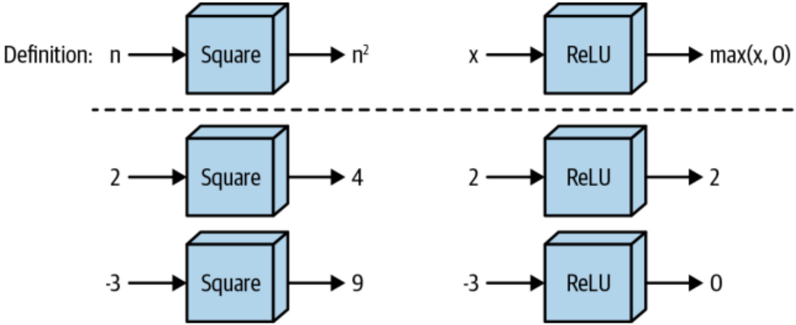
1. Draw an x-y plane.
2. Plot a bunch of point
3. Connect these plotted points.



This was first done by the French philosopher René Descartes.

Functions

We can think of functions as boxes that take in numbers as input and produce numbers as output



Code

```python
def square(x: ndarray) -> ndarray:
    '''
    Square each element in the input ndarray.
    '''
    return np.power(x, 2)

def leaky_relu(x: ndarray) -> ndarray:
    '''
    Apply "Leaky ReLU" function to each element in ndarray.
    '''
    return np.maximum(0.2 * x, x)
```

Outline

Derivatives

The derivative of a function at a point is the "rate of change" of the output of the function with respect to its input at that point

Math

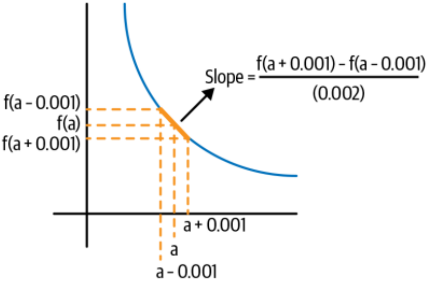
$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 \times \Delta}$$

For very small value for $\triangle$, such as 0.001

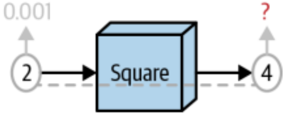
$$\frac{df}{du}(a) = \frac{f(a+0.001) - f(a-0.001)}{0.002}$$

Derivatives

Diagrams



A small +ve change in the input will lead to small -ve change in the output.



Derivatives

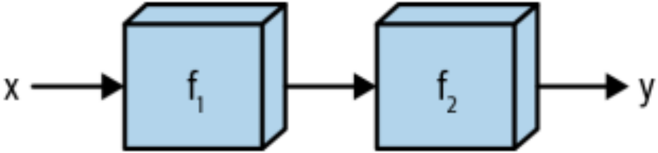
Code

```
def deriv(func: Callable[[ndarray], ndarray],
          input_: ndarray,
          delta: float = 0.001) -> ndarray:
    '''
    Evaluates the derivative of a function "func" at every element in the
    "input_" array.
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```

Outline

Nested Functions

functions can be "nested" to form "composite" functions



Nested Functions

functions can be “nested” to form “composite” functions

Math

We should also include the less intuitive mathematical representation:

$$f_2(f_1(x)) = y$$

This is less intuitive because of the quirk that nested functions are read “from the outside in”

Nested Functions

Code

```
from typing import List


# A Function takes in an ndarray as an argument and produces an ndarray
Array_Function = Callable[[ndarray], ndarray]


# A Chain is a list of functions
Chain = List[Array_Function]
```

Then we'll define how data goes through a chain, first of length 2:

```
def chain_length_2(chain: Chain,
                   a: ndarray) -> ndarray:
    '''
    Evaluates two functions in a row, in a "Chain".
    '''
    assert len(chain) == 2, \


    "Length of input 'chain' should be 2"


    f1 = chain[0]
    f2 = chain[1]


    return f2(f1(x))
```

Outline

The Chain Rule

The chain rule is a mathematical theorem that lets us compute derivatives of composite functions.

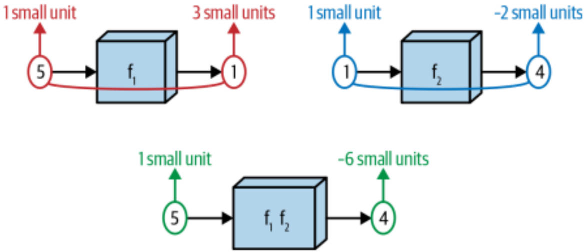
Math

$$\frac{df_2}{du}(x) = \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x)$$

where u is simply a dummy variable representing the input to a function.

The Chain Rule

Diagram



by considering the diagram and the math, we can reason through what the derivative of the output of a nested function with respect to its input

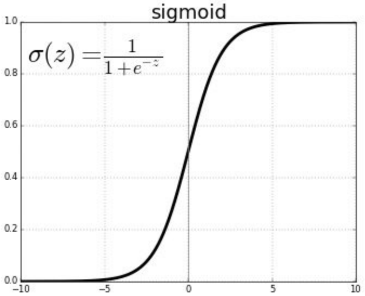
The Chain Rule

Code

```python
def chain_deriv_2(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    Uses the chain rule to compute the derivative of two nested functions:
    (f2(f1(x))' = f2'(f1(x)) * f1'(x)
    '''


    assert len(chain) == 2, \

    "This function requires 'Chain' objects of length 2"

    assert input_range.ndim == 1, \

    "Function requires a 1 dimensional ndarray as input_range"

    f1 = chain[0]
    f2 = chain[1]

    # df1/dx
    f1_of_x = f1(input_range)

    # df1/du
    df1dx = deriv(f1, input_range)

    # df2/du(f1(x))
    df2du = deriv(f2, f1(input_range))

    # Multiplying these quantities together at each point
    return df1dx * df2du
```

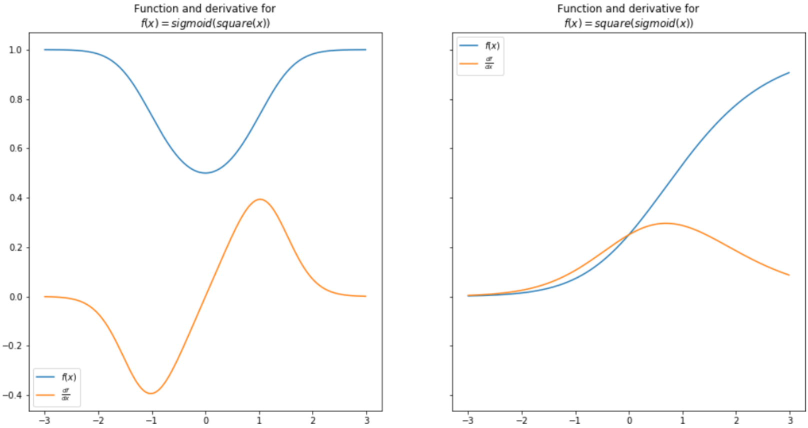
Nesting Sigmoid and Square Function

Sigmoid Function: Very useful function in deep learning




```
def sigmoid(x: ndarray) -> ndarray:
    '''
    Apply the sigmoid function to each element in the input ndarray.
    '''
    return 1 / (1 + np.exp(-x))
```

Nesting Sigmoid and Square Function

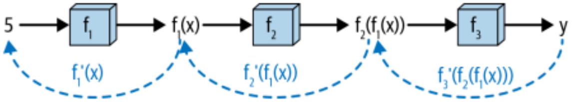


When the functions are upward-sloping, the derivative is positive; when they are flat, the derivative is zero; and when they are downward-sloping, the derivative is negative.

Slightly Longer Example

Lets consider three mostly differentiable functions— f_1 , f_2 , and f_3

Diagram



Small change in the input cause a sequece of changes.

Slightly Longer Example

Lets consider three mostly differentiable functions— f_1 , f_2 , and f_3

Math

$$\frac{df_3}{du}(x) = \frac{df_3}{du}(f_2(f_1(x))) \times \frac{df_2}{du}(f_1(x)) \times \frac{df_1}{du}(x))$$

Sequence of multiplications.

Slightly Longer Example

Lets consider three mostly differentiable functions— f_1 , f_2 , and f_3

Code

```
def chain_deriv_3(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    Uses the chain rule to compute the derivative of three nested functi
    (f3(f2(f1)))' = f3'(f2(f1(x))) * f2'(f1(x)) * f1'(x)
    '''


    assert len(chain) == 3, \


    "This function requires 'Chain' objects to have length 3"


    f1 = chain[0]
    f2 = chain[1]
    f3 = chain[2]


    # f1(x)
    f1_of_x = f1(input_range)

    # f2(f1(x))
    f2_of_x = f2(f1_of_x)


    # df3du
    df3du = deriv(f3, f2_of_x)


    # df2du
    df2du = deriv(f2, f1_of_x)


    # df1dx
    df1dx = deriv(f1, input_range)


    # Multiplying these quantities together at each point
    return df1dx * df2du * df3du
```

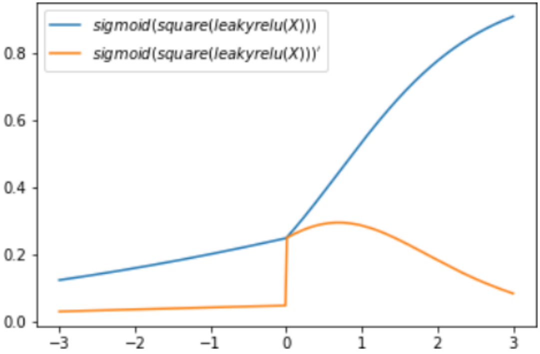
Slightly Longer Example (Notes)

Something interesting took place here—to compute the chain rule for this nested function, we made two "passes" over it:

1. First, we went "forward" through it, computing the quantities f1_of_x and f2_of_x along the way. We can call this (and think of it as) "the forward pass."
2. Then, we "went backward" through the function, using the quantities that we computed on the forward pass to compute the quantities that make up the derivative.

Finally, we multiplied three of these quantities together to get our derivative.

Slightly Longer Example (Notes)



comparing the plots of the derivatives to the slopes of the original functions, we see that the chain rule is indeed computing the derivatives properly.

Outline

Functions with Multiple Inputs

The functions we deal with in deep learning don't have just one input. Instead, they have several inputs

Math

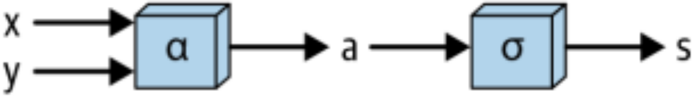
$$a = \alpha(x, y) = x + y$$

We can feed the output "a" to another function

$$s = \sigma(a)$$

Functions with Multiple Inputs

Diagram



Here we see the two inputs going into α and coming out as a and then being fed through σ.

Functions with Multiple Inputs

Code

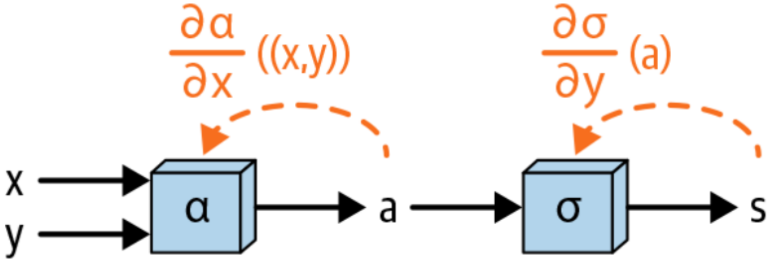
Coding this up is very straightforward;

```
def multiple_inputs_add(x: ndarray,
                        y: ndarray,
                        sigma: Array_Function) -> float:
    '''
    Function with multiple inputs and addition, forward pass.
    '''
    assert x.shape == y.shape


    a = x + y
    return sigma(a)
```

Derivatives of Functions with Multiple Inputs

Diagrams

compute the derivative of each constituent function "going backward" through the computational graph and then multiply the results together to get the total derivative.



Derivatives of Functions with Multiple Inputs

Math

The chain rule applies to these functions in the same way it applied to the functions in the prior sections. Since this is a nested function, with $f(x,y) = \sigma(\alpha(x,y))$, we have:

$$\frac{\partial f}{\partial x} = \frac{\partial \sigma}{\partial u}(\alpha(x,y)) \times \frac{\partial \alpha}{\partial x}((x,y)) = \frac{\partial \sigma}{\partial u}(x+y) \times \frac{\partial \alpha}{\partial x}((x,y))$$

And of course df/dy would be identical.

Now note that:

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

since for every unit increase in x, a increases by one unit, no matter the value of x (the same holds for y).

Derivatives of Functions with Multiple Inputs

Code

```
def multiple_inputs_add_backward(x: ndarray,
                                 y: ndarray,
                                 sigma: Array_Function) -> float:
    '''
    Computes the derivative of this simple function with respect to
    both inputs.
    '''
    # Compute "forward pass"
    a = x + y


    # Compute derivatives
    dsda = deriv(sigma, a)

    dadx, dady = 1, 1

    return dsda * dadx, dsda * dady
```

Outline

Functions with Multiple Vector Inputs

- In deep learning, we deal with functions whose inputs are vectors or matrices
- We will compute the derivatives of complex functions involving dot products and matrix multiplications will be essential.

Creating New Features from Existing Features

- The single most common operation in neural networks is to form a "weighted sum" of the input, where the weighted sum could emphasize certain features and deemphasize others

Math

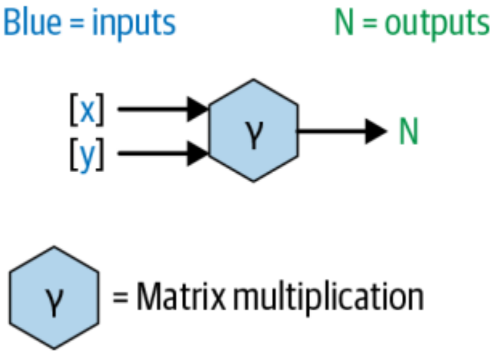
$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

then we could define the output of this operation as:

$$N = \gamma(X, W) = X \cdot W = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n$$

Creating New Features from Existing Features

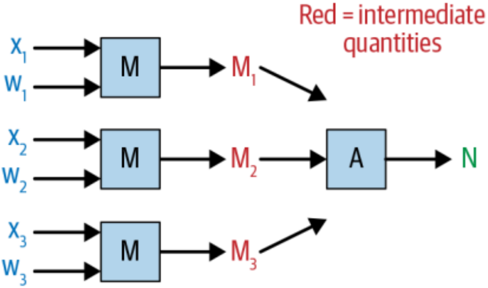
Diagram



two inputs, both of which can be ndarrays, and one output.

Creating New Features from Existing Features

Another detailed diagram



Creating New Features from Existing Features

Code

```
def matmul_forward(X: ndarray,
                   W: ndarray) -> ndarray:
    '''
    Computes the forward pass of a matrix multiplication.
    '''


    assert X.shape[1] == W.shape[0], \


    '''
    For matrix multiplication, the number of columns in the first array should
    match the number of rows in the second; instead the number of columns in the
    first array is {0} and the number of rows in the second array is {1}.
    '''.format(X.shape[1], W.shape[0])


    # matrix multiplication
    N = np.dot(X, W)


    return N
```

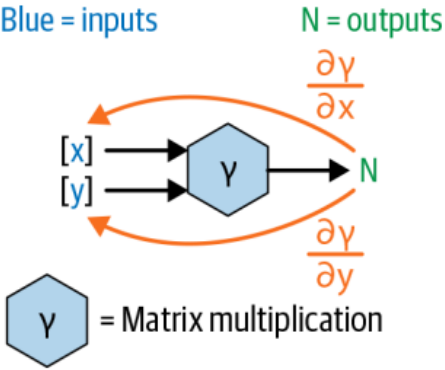
Derivatives of Functions with Multiple Vector Inputs

- For vector functions, it isn't immediately obvious what the derivative is.
- Small change to any of the inputs can cause output change.
- It is more natural to think of a derivative with respect to each input.

Derivatives of Functions with Multiple Vector Inputs

- For vector functions, it isn't immediately obvious what the derivative is.
- Small change to any of the inputs can cause output change.
- It is more natural to think of a derivative with respect to each input.

Diagram



Derivatives of Functions with Multiple Vector Inputs

Math

We get the derivative with respect to each element of the vector

$$\frac{\partial\gamma}{\partial X} = \left[\frac{\partial\gamma}{\partial x_1}, \frac{\partial\gamma}{\partial x_2}, \frac{\partial\gamma}{\partial x_3}\right]$$

Derivatives of Functions with Multiple Vector Inputs

Math

We get the derivative with respect to each element of the vector

$$\frac{\partial \gamma}{\partial X} = \left[\frac{\partial \gamma}{\partial x_1}, \frac{\partial \gamma}{\partial x_2}, \frac{\partial \gamma}{\partial x_3}\right]$$

Remember $\gamma(X, W) = X \cdot W = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n$

Derivatives of Functions with Multiple Vector Inputs

Math

We get the derivative with respect to each element of the vector

$$\frac{\partial \gamma}{\partial X} = \left[\frac{\partial \gamma}{\partial x_1}, \frac{\partial \gamma}{\partial x_2}, \frac{\partial \gamma}{\partial x_3} \right]$$

Remember $\gamma(X, W) = X \cdot W = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n$

we can see that if, x_i changes by $\triangle$ units, then output change by $w_i \times \triangle$ units

$$\frac{\partial \gamma}{\partial X} = [w_1, w_2, w_3] = W^T$$

Derivatives of Functions with Multiple Vector Inputs

Math

We get the derivative with respect to each element of the vector

$$\frac{\partial\gamma}{\partial X} = \left[\frac{\partial\gamma}{\partial x_1}, \frac{\partial\gamma}{\partial x_2}, \frac{\partial\gamma}{\partial x_3}\right]$$

Remember $\gamma(X, W) = X \cdot W = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n$

we can see that if, x_i changes by $\triangle$ units, then output change by $w_i \times \triangle$ units

$$\frac{\partial\gamma}{\partial X} = [w_1, w_2, w_3] = W^T$$

And also,

$$\frac{\partial\gamma}{\partial W} = [x_1, x_2, x_3] = X^T$$

Derivatives of Functions with Multiple Vector Inputs

Code

```
def matmul_backward_first(X: ndarray,
                          W: ndarray) -> ndarray:
    '''
    Computes the backward pass of a matrix multiplication with respect to the
    first argument.
    '''

    # backward pass
    dNdX = np.transpose(W, (1, 0))

    return dNdX
```

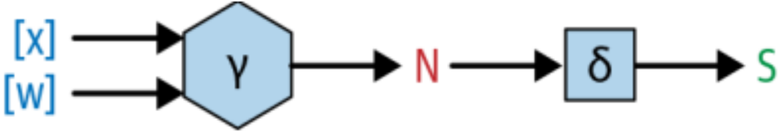
Vector Functions and Their Derivatives: One Step Further

- Deep learning models, of course, involve more than one operation
- Therefore, we'll now look at computing the derivative of a composite functions with vector inputs.
- Suppose the following function

$$S = \sigma(\gamma(X, W))$$

Vector Functions and Their Derivatives: One Step Further

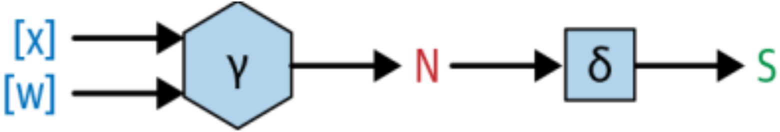
Diagram



Same graph as before, but with another function tacked onto the end

Vector Functions and Their Derivatives: One Step Further

Diagram



Same graph as before, but with another function tacked onto the end

Vector Functions and Their Derivatives: One Step Further

Math

$$S = \sigma(\gamma(X, W)) = \sigma(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$$

Mathematically, this is straightforward as well

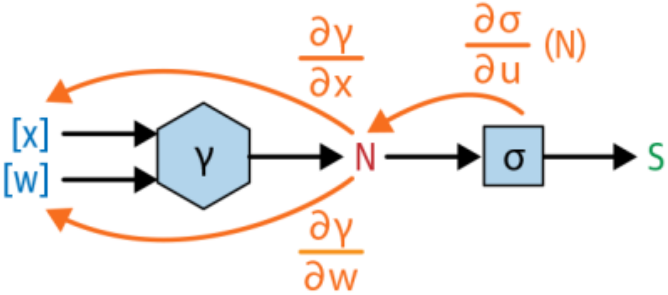
Vector Functions and Their Derivatives: One Step Further

Code

```
def matrix_forward_extra(X: ndarray,
                         W: ndarray,
                         sigma: Array_Function) -> ndarray:
    '''
    Computes the forward pass of a function involving matrix multiplication,
    one extra function.
    '''
    assert X.shape[1] == W.shape[0]


    # matrix multiplication
    N = np.dot(X, W)

    # feeding the output of the matrix multiplication through sigma
    S = sigma(N)

    return S
```

Vector Functions and Their Derivatives: The Backward Pass

Diagram



Vector Functions and Their Derivatives: The Backward Pass

Math

$$\frac{\partial S}{\partial X} = \frac{\partial \sigma}{\partial u}(\gamma(X, W)) \times \frac{\partial \gamma \sigma}{\partial X}(X, W)$$

Vector Functions and Their Derivatives: The Backward Pass

Math

$$\frac{\partial S}{\partial X} = \frac{\partial \sigma}{\partial u}(\gamma(X, W)) \times \frac{\partial \gamma \sigma}{\partial X}(X, W)$$

The first part of this is simply

$$\frac{\partial \sigma}{\partial u}(\gamma(X, W)) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$$

We will just evaluating derivative of σ at $(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$

Vector Functions and Their Derivatives: The Backward Pass

Math

$$\frac{\partial S}{\partial X} = \frac{\partial \sigma}{\partial u}(\gamma(X, W)) \times \frac{\partial \gamma \sigma}{\partial X}(X, W)$$

The first part of this is simply

$$\frac{\partial \sigma}{\partial u}(\gamma(X, W)) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$$

We will just evaluating derivative of σ at $(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$
Furthermore, we reasoned that $\frac{\partial \gamma}{\partial X} = W^T$

Vector Functions and Their Derivatives: The Backward Pass

Math

$$\frac{\partial S}{\partial X} = \frac{\partial \sigma}{\partial u}(\gamma(X, W)) \times \frac{\partial \gamma \sigma}{\partial X}(X, W)$$

The first part of this is simply

$$\frac{\partial \sigma}{\partial u}(\gamma(X, W)) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$$

We will just evaluating derivative of σ at $(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n)$ Furthermore, we reasoned that $\frac{\partial \gamma}{\partial X} = W^T$ Thus,

$$\frac{\partial S}{\partial X} = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n) \times W^T$$

Vector Functions and Their Derivatives: The Backward Pass

Code

```
def matrix_function_backward_1(X: ndarray,
                               W: ndarray,
                               sigma: Array_Function) -> ndarray:
    '''
    Computes the derivative of our matrix function with respect to
    the first element.
    '''
    assert X.shape[1] == W.shape[0]


    # matrix multiplication
    N = np.dot(X, W)

    # feeding the output of the matrix multiplication through sigma
    S = sigma(N)

    # backward calculation
    dSdN = deriv(sigma, N)

    # dNdX
    dNdX = np.transpose(W, (1, 0))

    # multiply them together; since dNdX is 1x1 here, order doesn't matter
    return np.dot(dSdN, dNdX)
```

Outline

Two 2D Matrix Inputs and One Matrix Output

- In deep learning we deal with operations that take as input two 2D arrays, one of which represents a batch of data X and the other of which represents the weights W.

Two 2D Matrix Inputs and One Matrix Output

- In deep learning we deal with operations that take as input two 2D arrays, one of which represents a batch of data X and the other of which represents the weights W.

$$\begin{bmatrix} o_{11} & o_{12} \\ o_{21} & o_{22} \\ o_{31} & o_{33} \end{bmatrix}_{(3\times2)} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}_{(3\times3)} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}_{(3\times2)}$$

Two 2D Matrix Inputs and One Matrix Output

- In deep learning we deal with operations that take as input two 2D arrays, one of which represents a batch of data X and the other of which represents the weights W.

$$\begin{bmatrix} o_{11} & o_{12} \\ o_{21} & o_{22} \\ o_{31} & o_{33} \end{bmatrix}_{(3\times 2)} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}_{(3\times 3)} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}_{(3\times 2)}$$

- Lets calculate $\frac{dO}{du}(X)$

Two 2D Matrix Inputs and One Matrix Output

- In deep learning we deal with operations that take as input two 2D arrays, one of which represents a batch of data X and the other of which represents the weights W.

$$\begin{bmatrix} o_{11} & o_{12} \\ o_{21} & o_{22} \\ o_{31} & o_{33} \end{bmatrix}_{(3\times2)} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}_{(3\times3)} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}_{(3\times2)}$$

- Lets calculate $\frac{dO}{du}(X)$
- How the output changes when you change the input X

Two 2D Matrix Inputs and One Matrix Output

- In deep learning we deal with operations that take as input two 2D arrays, one of which represents a batch of data X and the other of which represents the weights W.

$$\begin{bmatrix} o_{11} & o_{12} \\ o_{21} & o_{22} \\ o_{31} & o_{33} \end{bmatrix}_{(3\times 2)} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}_{(3\times 3)} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}_{(3\times 2)}$$

- Lets calculate $\frac{dO}{du}(X)$
- How the output changes when you change the input X
- $\frac{dO}{du}(X)$ is (3×3) matrix

Two 2D Matrix Inputs and One Matrix Output

- Often we write $\frac{dO}{du}(X) = ones * W^T$

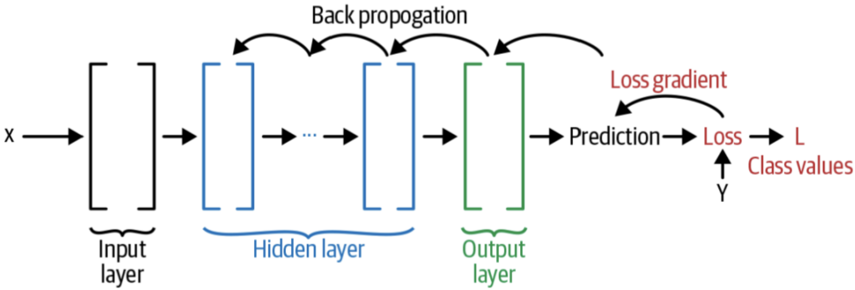
$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}_{(3\times 2)} \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}_{(2\times 3)}$$

- This representation will be of great help in the computation of the chain rule.

Complex Computational Graph

Diagram

Two matrices are multiplied, then elementwise sigmoid, then summation of the output.



Complex Computational Graph

Math

Two matrices are multiplied, then elementwise sigmoid, then summation of the output.

$$L = \Lambda(\sigma(\gamma(X, W)))$$

- $\gamma(X, W) = X_{(3\times3)} \times W_{(3\times2)}$

Complex Computational Graph

Math

Two matrices are multiplied, then elementwise sigmoid, then summation of the output.

$$L = \Lambda(\sigma(\gamma(X, W)))$$

- $\gamma(X, W) = X_{(3\times3)} \times W_{(3\times2)}$
- $\sigma(.)$ is elementwise sigmoid

Complex Computational Graph

Math

Two matrices are multiplied, then elementwise sigmoid, then summation of the output.

$$L = \Lambda(\sigma(\gamma(X, W)))$$

- $\gamma(X, W) = X_{(3\times3)} \times W_{(3\times2)}$
- $\sigma(.)$ is elementwise sigmoid
- $\Lambda(.)$ is summation of the input

Complex Computational Graph

Math

Two matrices are multiplied, then elementwise sigmoid, then summation of the output.

$$sum\left(\begin{bmatrix} \sigma(x_{11}w_{11}+x_{12}w_{21}+x_{13}w_{31}) & \sigma(x_{11}w_{12}+x_{12}w_{22}+x_{13}w_{32}) \\ \sigma(x_{21}w_{11}+x_{22}w_{21}+x_{23}w_{31}) & \sigma(x_{21}w_{12}+x_{22}w_{22}+x_{23}w_{32}) \\ \sigma(x_{31}w_{11}+x_{32}w_{21}+x_{33}w_{31}) & \sigma(x_{31}w_{12}+x_{32}w_{22}+x_{33}w_{32}) \end{bmatrix}_{(3\times 2)}\right)$$

The Backward Pass

Math

$$\frac{dL}{dX} = \frac{d\Lambda}{du}(S) \times \frac{d\sigma}{du}(N) \times \frac{d\gamma}{du}(X)$$

The Backward Pass

Math

$$\frac{d\Lambda}{du}(S) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}_{(3\times2)}$$

$$\frac{d\sigma}{du}(N) = \begin{bmatrix} \frac{\partial\sigma(o_{11})}{\partial u} & \frac{\partial\sigma(o_{12})}{\partial u} \\ \frac{\partial\sigma(o_{21})}{\partial u} & \frac{\partial\sigma(o_{22})}{\partial u} \\ \frac{\partial\sigma(o_{31})}{\partial u} & \frac{\partial\sigma(o_{32})}{\partial u} \end{bmatrix}_{(3\times2)}$$

$$\frac{d\gamma}{du}(X) = W^T_{(2\times3)}$$

The Backward Pass

Math

$$\frac{dL}{dX} = \frac{d\Lambda}{du}(S) \times \frac{d\sigma}{du}(N) \times \frac{d\gamma}{du}(X)$$

- Note, second multiplication is element-wise multiplication, while first is matrix multiplication.
- Thus, $\frac{dL}{dX}$ is a (3×3) matrix as expected

References

- Seth Weidman "Deep Learning from Scratch Building with Python from First Principles"

Questions