Introduction to Neural Networks

ECEN 478

Dr. Mahmoud Nabil Mahmoud
mnmahmoud@ncat.edu

North Carolina A & T State University

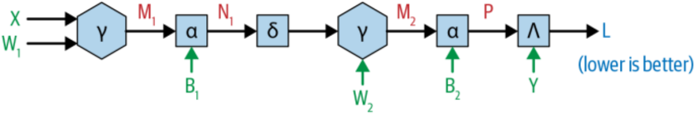
March 1, 2023

Outline

1. Introduction
2. Automatic Diffrentiation
3. Code

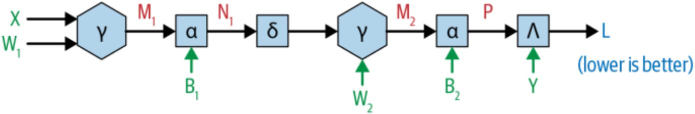
MultiLayer Perceptron

- All types of Neural Networks defined so far, are somewhat linear in their architecture, which means there is no branching in the computations



MultiLayer Perceptron

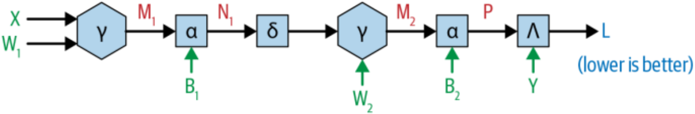
- All types of Neural Networks defined so far, are somewhat linear in their architecture, which means there is no branching in the computations



- Thus, to compute the backward propagation we defined an "Operation" class as the atomic unit of that makup our network

MultiLayer Perceptron

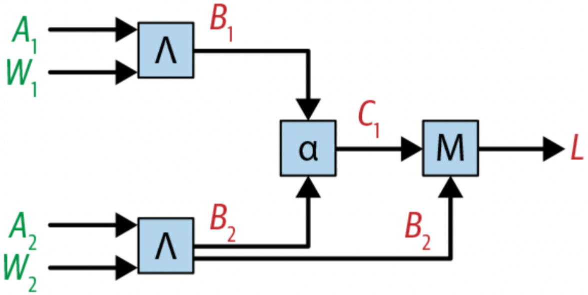
- All types of Neural Networks defined so far, are somewhat linear in their architecture, which means there is no branching in the computations



- Thus, to compute the backward propagation we defined an "Operation" class as the atomic unit of that makup our network
- Then, to compute the backward gradient we have to iterate (for loop) over the "operations" in the reverse direction

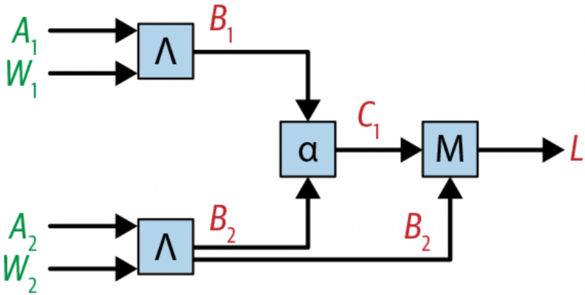
The key limitation- handling branching

- The previous approach can not handle branching as in the example below
- Recurrent Neural Networks have many branches



- Λ is a matrix multiplication
- α is matrix addition
- M is element wise multiplication

The key limitation- handling branching



- As you can see B_2 is affecting two operations (branching)
- Thus B2 should recieve two gradients from each branch (dB_{2_1}, dB_{2_1})
- $dB_2 = dB_{2_1} + dB_{2_1}$

Outline

1. Introduction
2. Automatic Diffrentiation
3. Code

Automatic Diffrentiation

- Automatic differentiation allows us to compute these gradients via smart route.

Automatic Diffrentiation

- Automatic differentiation allows us to compute these gradients via smart route.
- Rather than the "operations" being the atomic units that make up the network, we define a class that wraps around the data itself and allows the data to keep track of the operations performed on it,

Automatic Diffrentiation

- Automatic differentiation allows us to compute these gradients via smart route.
- Rather than the "operations" being the atomic units that make up the network, we define a class that wraps around the data itself and allows the data to keep track of the operations performed on it,
- So that the data can continually accumulate gradients as it is involved in different operations

Automatic Diffrentiation

- Automatic differentiation allows us to compute these gradients via smart route.
- Rather than the "operations" being the atomic units that make up the network, we define a class that wraps around the data itself and allows the data to keep track of the operations performed on it,
- So that the data can continually accumulate gradients as it is involved in different operations
- What we will build is a small scale version of what is happening in Pytorch and Tensorflow

Derivative Example

Suppose we have the following equation

$$\mathrm{e} = (4a + 3) \times (a + 2) = 4a^2 + 11a + 6$$

What is $\frac{\partial e}{\partial a}$ at $a = 3$

Derivative Example

Suppose we have the following equation

$$\mathrm{e} = (4a + 3) \times (a + 2) = 4a^2 + 11a + 6$$

What is $\frac{\partial e}{\partial a}$ at $a = 3$

$$\frac{\partial \mathrm{e}}{\partial a} = 8a + 11$$

Derivative Example

Suppose we have the following equation

$$\mathrm{e} = (4a + 3) \times (a + 2) = 4a^2 + 11a + 6$$

What is $\frac{\partial e}{\partial a}$ at $a = 3$

$$\frac{\partial \mathrm{e}}{\partial a} = 8a + 11$$

OR

```python
def forward(num: int):
    b = num * 4
    c = b + 3
    return c * (num + 2)

print(round(forward(3.01) - forward(2.99)) / 0.02), 3)
```

```
35.0
```

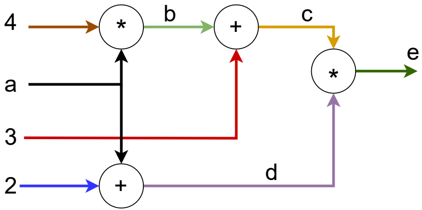
Derivative Example

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e.backward()

print(a.grad)
```

```
35
```



Ultimate Goal

- The goal with automatic differentiation is to make the data objects themselves—numbers the fundamental units of analysis.
- We will create a class that wraps around the actual data being computed (int or float).
- Common operations such as adding, multiplying, and matrix multiplication are redefined so that the computiational graph is constructed on the fly.
- The wrapper class contain information on how to compute gradients, given what happens on the forward pass.

Outline

Code

```
Numberable = Union[float, int]

def ensure_number(num: Numberable) -> NumberWithGrad:
    if isinstance(num, NumberWithGrad):
        return num
    else:
        return NumberWithGrad(num)

class NumberWithGrad(object):

    def __init__(self,
                 num: Numberable,
                 depends_on: List[Numberable] = None,
                 creation_op: str = ''):
        self.num = num
        self.grad = None
        self.depends_on = depends_on or []
        self.creation_op = creation_op

    def __add__(self,
                other: Numberable) -> NumberWithGrad:
        return NumberWithGrad(self.num + ensure_number(other).num,
                              depends_on = [self, ensure_number(other)],
                              creation_op = 'add')

    def __mul__(self,
                other: Numberable = None) -> NumberWithGrad:

        return NumberWithGrad(self.num * ensure_number(other).num,
                              depends_on = [self, ensure_number(other)],
                              creation_op = 'mul')
```

Code Continue

```python
def backward(self, backward_grad: Numberable = None) -> None:
    if backward_grad is None: # first time calling backward
        self.grad = 1
    else:
        # These lines allow gradients to accumulate.


        # If the gradient doesn't exist yet, simply set it equal
        # to backward_grad
        if self.grad is None:
            self.grad = backward_grad
        # Otherwise, simply add backward_grad to the existing gradient
        else:
            self.grad += backward_grad

    if self.creation_op == "add":
        # Simply send backward self.grad, since increasing either of these
        # elements will increase the output by that same amount
        self.depends_on[0].backward(self.grad)
        self.depends_on[1].backward(self.grad)

    if self.creation_op == "mul":

        # Calculate the derivative with respect to the first element
        new = self.depends_on[1] * self.grad
        # Send backward the derivative with respect to that element
        self.depends_on[0].backward(new.num)

        # Calculate the derivative with respect to the second element
        new = self.depends_on[0] * self.grad
        # Send backward the derivative with respect to that element
        self.depends_on[1].backward(new.num)
```

Questions