NN-Z2H Lesson 1: The spelled-out intro to neural networks and backpropagation - building micrograd

Tuan Le Khac

2024-06-16

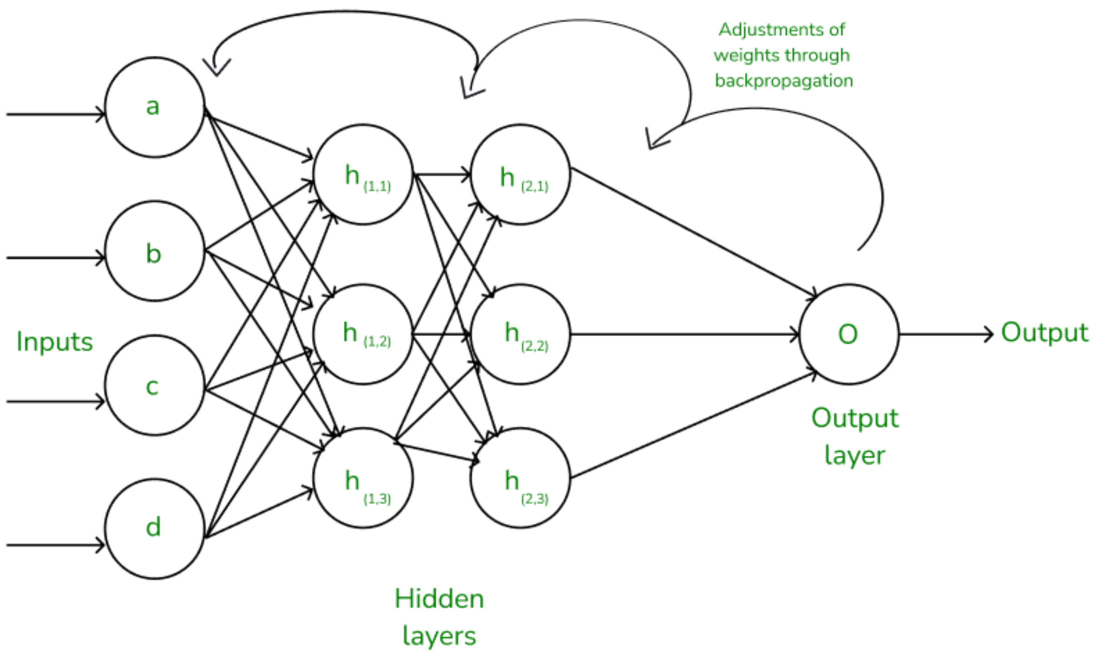
Table of contents

Upfront-note: There are also greate resources in Vietnamese for learning Backpropagation, for e.g.:

1. Blog machinelearningcoban
2. Blog dominhhai

1 MicroGrad from scratch Yayy!

MicroGrad repo **Video Lecture**



(a) Backpropagation in Neural Networks, photo credit to GeekforGeek

1.1 intro & micrograd overview - what does your neural network training look like under the hood?

What is MicroGrad : a tiny **auto-grad** (automatic gradient) engine, implement of **backpropagation** ~ itertively tune the weight of that nn to minimize the loss function -> improve the accuracy of the neural network. Backpropagation will be the mathematical core of any modern deep neutral network like, say `pytorch`, or `jaxx`.

Installation: `pip install micrograd`

Example:

```
from micrograd.engine import Value

a = Value(-4.0)                                                       ①
b = Value(2.0)
c = a + b
d = a * b + b**3
c += c + 1
c += 1 + c + (-a)
d += d * 2 + (b + a).relu()
d += 3 * d + (b - a).relu()
e = c - d
f = e**2
g = f / 2.0
g += 10.0 / f                                                         ②
print(f'{g.data:.4f}') # prints 24.7041, the outcome of this forward pass ③
g.backward()
print(f'{a.grad:.4f}') # prints 138.8338, i.e. the numerical value of dg/da ④
print(f'{b.grad:.4f}') # prints 645.5773, i.e. the numerical value of dg/db
```

① Micrograd allows you to build mathematical expressions, in this case `a` and `b` are inputs, wrapped in `Value` object with value equal to `-4.0` and `2.0`, respectively.
② `a` and `b` are transformed to `c`, `d`, `e` and eventually `f`, `g`. Mathematical operators are implemented, like `+`, `*`, `**`, even `relu()`.
③ `Value` object contains `data`, and `grad`.
④ Call `backpropagation()` process.

```
24.7041
138.8338
645.5773
```

1.2 derivative of a simple function with one input

What exactly is derivative

```
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

A simple quadratic function:

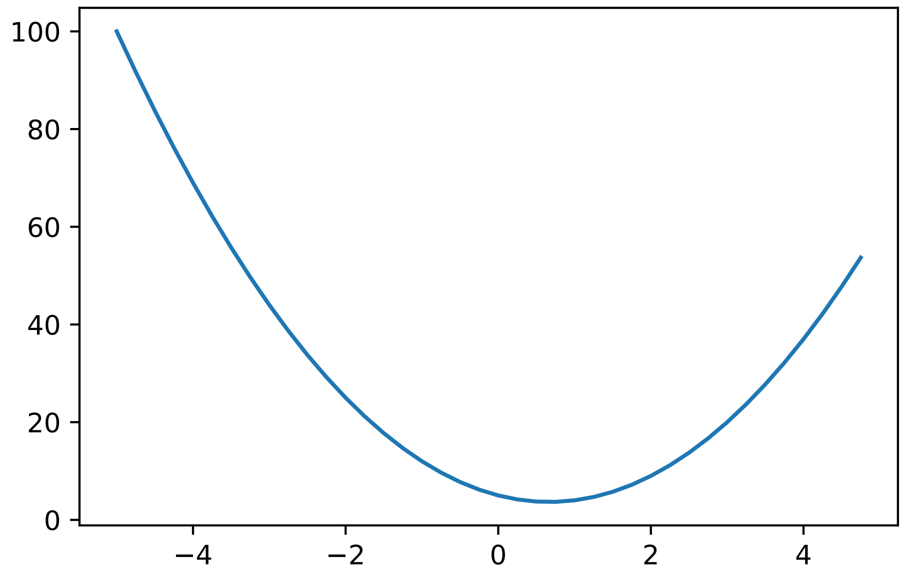
```python
def f(x):
    return 3*x**2 - 4*x + 5
```

```python
f(3.0)
```

```
20.0
```

Input also can be an array, we can plot it for visibility.

```python
xs = np.arange(-5, 5, 0.25)
ys = f(xs)
plt.plot(xs, ys)
```



If we bump up a litle value `h` of `x`, how `f(x)` will response?

```python
h = 0.000000000001                                                    ①
x = 3.0
( f(x+h) - f(x) ) / h
```

① Change the value of `h` from `0.0001` to be `0.00000...0001` -> the slope value comes to `14` (at the value of `3.0` of `x`).

```
14.001244608152774
```

Try for `x = -3.0`, `x = 5.0`, we get different values of the slope, for `x = 2/3`, the slope is zero. Let's get more complex.

1.3 derivative of a function with multiple inputs

```
a = 2.0
b = -3.0
c = 10.0
d = a*b + c
print(d)
```

```
4.0
```

Put our bump-up element to this multi-variables function:

```
h = 0.001

# input
a = 2.0
b = -3.0
c = 10.0

d1 = a*b + c
a += h                                              ①
d2 = a*b + c

print('d1: ', d1)
print('d2: ', d2)
print('slope: ', (d2 - d1)/h)                       ②
```

① Do the same for `b`, `c`, we'll get different slopes.
② We say given `b = -3.0` and `c = 10.0` are constants, the derivative of `d` at `a = 2.0` is `-3.0`. The rate of which `d` will increase if we scale `a`!

```
d1:  4.0
d2:  3.997
slope:  -3.0000000000001137
```

1.4 starting the core `Value` object of micrograd and its visualization

So we now have some intuitive sense of what is derivative is telling you about the function. We now move to the Neural Networks, which would be massive mathematical expressions. We need some data structures that maintain these expressions, we first declare an object `Value` that holds data.

```python
class Value:
    def __init__(self, data,
                        _children=(),                                ③
                        _op = '',                                    ⑤
                        label = ''
                        ):
        self.data = data
        self.grad = 0.0                                              ⑥
        self._backward = lambda: None                                ⑦
        self._prev = set(_children)
        self._op = _op
        self.label =  label

    def __repr__(self) -> str: # a nicer looking for class attributes
        return f"Value(data={self.data})"

    def __add__(self, other):                                        ④
        other = other if isinstance(other, Value) else Value(other) # turn other to Value obje
        out = Value(self.data + other.data, (self, other), '+')

        def _backward():                                             ⑧
            self.grad += 1.0 * out.grad
            other.grad += 1.0 * out.grad
        out._backward = _backward

        return out

    def __mul__(self, other):
        other = other if isinstance(other, Value) else Value(other) # turn other to Value obje
        out = Value(self.data * other.data, (self, other), '*')

        def _backward():
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward

        return out

    def tanh(self):
        x = self.data
        t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
        out = Value(t, (self, ), 'tanh')

        def _backward():
            self.grad += (1 - t**2) * out.grad
        out._backward = _backward
        return out

    def backward(self):

        # topo order for all children in the graph
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
                topo.append(v)
        build_topo(self)

        # sequentially apply the chain rules
        self.grad = 1.0
        for node in reversed(topo):
            node._backward()
```

③ the connective tissue of this expression. We want to keep these expression graphs, so we need to know and keep pointers about what values produce what other values. `_children` is by default a empty tuple.
④ as we added `_children`, we also need to point out the father - children relationship in method `__add__` and `__mul__` as well.
⑤ we want to know the **operation** between father and child, `_op` is empty string by default, the value `+` and `-` will be added to the operator method respectively.
⑥ initially assume that node has no impact to the output.
⑦ this backward function basically do nothing at the initial.
⑧ implement of backward pass for plus node, `+=` represent the accumulate action (rather than overwrite it), assigne the gradient behaviour for each type of operation, call the `_backward` concurrently with function.

Setting input and expression:

```
a = Value(2.0, label='a')
b = Value(-3.0, label='b')
c = Value(10.0, label='c')

a + b                                                          ①

a*b + c                                                        ②
```

```
# d = a*b + c rewrite the expression
e = a*b; e.label = 'e'
d = e + c; d.label = 'd'
# d
f = Value(-2.0, label='f')
L = d * f; L.label = 'L'
L
```

① which will internally call `a.__add__(b)`
② which will internally call `(a.__mul__(b)).__add__(c)`

```
Value(data=-8.0)
```

So that we can know the children:

```
d._prev
```

```
{Value(data=-6.0), Value(data=10.0)}
```

We can know the operations:

```
d._op
```

```
'+'
```

Now we know exactly how each value came to be by **word** expression and from what other values. These will be quite abit larger, so we need a way to nicely visualize these expressions that we're building out. Below are a-little-scary codes.

```
import os

# Assuming the Graphviz bin directory path is 'C:/Program Files (x86)/Graphviz2.xx/bin'
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz/bin' # add with the code,

from graphviz import Digraph

def trace(root):
    # build a set of all nodes and edges in a graph
    nodes, edges = set(), set()
    def build(v):
        if v not in nodes:
            nodes.add(v)                                                  ①
```