Micrograd

https://www.youtube.com/watch?v=VMj-3S1tku0

Pythonda fonksiyon gösterimi:

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

Türevin tanımı (slope aynı zamanda da) - First Derivative

lim h→0 (f(x+h) - f(x)) / h

Olayı şu f(x)'deki x'i h kadar arttırdığımızda yeni sayımız h'a oranla ne kadar artıyor? Bu sayede her elemanın ne kadar etki ettiğini bulabiliyor olacağız ilerde.

```
class Value:

    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
        self.grad = 0.0
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op
        self.label = label
```

Class oluşturuyoruz, classta da __ init __ constructor ile otomatik olarak tanımlıyor,

→ datayı, çocuklarını, hangi operasyon yapılacağını, ve label ı biz veriyoruz

→ _ gelmesi private gibi yapıyor, zorunlu değil ama yazılı olmayan bir kural gibi

→ Bir Value oluşturulduğuna bunun kimden geldiğini bilmemiz gerekiyor, backward için

_children yeni oluşan değerin ebeveynlerini oluşturan tuple

_prev'de de bu tuple'ı saklıyoruz

→_backward = lambda: None, fonksiyon hep boş başlıyor biz değiştiriyoruz başka fonksiyonlarla

Ara Başlık Python Closures

```
def outer_function(x):
    # Outer function: takes 'x' and defines inner_function
    def inner_function(y):
        return x + y  # 'x' is remembered from outer_function
    return inner_function  # Returns inner function (closure)

# Create a closure with x = 10
closure = outer_function(10)

# Call the closure with different values of 'y'
print(closure(5))
print(closure(20))
```

Çıktı 15 ve 30 çıkıyor

→ Yani fonksiyonu bi değere atadığımızda 2. değer y'ye gidiyor ve closure 10'u hatırlıyor

```
def make_counter():
    count = 0  # This variable will be remembered
```

```
    def counter():
        nonlocal count  # Modify outer variable
        count += 1
        return count
    return counter

counter1 = make_counter()
print(counter1())  # 1
print(counter1())
```

Çıktı 1 2

```
def pre(p):
    # Outer function stores prefix
    def add(t):
        # Inner function uses stored prefix
        return p + " " + t
    return add  # Return closure

# Create a closure that always prefixes with "Hello"
h = pre("Hello")
print(h("World"))
print(h("Python"))
```

Çıktı

Hello Word

Hello Python

When Python creates a closure:

- It stores outer function’s variables in a special attribute called **__closure__**
- The inner function keeps a reference, not a copy, to these variables

```
def make_multiplier(factor):
    def multiply_by(x):
        return x * factor
    return multiply_by

double = make_multiplier(2)
print(double(5))

# Check closure variables
print(double.__closure__[0].cell_contents)
```

10

2

```
    def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')

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

        return out
```

→ fonksiyon bittiğinde self, other, out siniliyor normalde

Ama _backward fonksiyonu bunları tutuyor normalde self._backward = lambda: None

```
a = Value(2.0)
b = Value(3.0)
c = a + b  # Burada __add__ çalışır
```

Not: Burda c = a + b yazıldığında interpreter “a”hangi sınıftan diye bakıyor. Sonra o sınıfın içinde __add __ diye özel fonksiyon tanımlanmış mı ona bakıyor.

aslında

c = a + b

c = a.__add__(b)

Syntactic Sugar deniyor buna da

peki c = 2 + a yazarsak? 2 Value classında değil, 2.__add__(a) yapmaya çalışır. ama

```
    def __radd__(self, other):
        return self + other
```

bu fonksiyon çözüyor onu. Right Add (sağtan toplama)

```
a = Value(2.0)
b = Value(3.0)
c = a + b  # Burada __add__ çalışır
```

- __add __ çağırılır self = a, other = b
- out yaratılır, c oluşur (data = 5.0) c nin içinde de _children olarak self ve other (a ve b) saklanır
- _backward tanımlanır(önce boş olan) sonra bunun içerisine c nin gradyanını al, a’nın ve b’nin gradyanına ekle olarak tanımlanır ama çalışmaz
- bu fonksiyon c nesnesinin (out) _backward değişkenine yapıştırılır
- c._backward kullanıldığında
    - c nin içerisinde backward fonksiyonunu bulur

- Closure ile c, a ve b yi hatırlar
- c.grad değerini alır, a.grad ve b.grad değerlerine ekler

```
def backward(self):

    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)

    self.grad = 1.0
    for node in reversed(topo):
        node._backward()
```

Bu topological sort

- build_topo(self) → en son düğümden (loss) başla
- if v not in visited → aynı düğümü 2 kez listeye eklememek için kontrol
- for child in v._prev → çocukarına (aslında onu oluşturan ebevynlere git)
    - Burda recursive olarak en dibe gidiyoruz
    - x veya w ye ulaşana kadar
- topo.append(v) → döngü bittikten sonra yapılır, önce çocuklar listeye giriyor.
- topo listesi → giriş katmanı, ara katmanlar, çıktı olarak dolar. Yani son giren, en tepedeki (başlangıçtaki) eleman. Ama biz türevi Loss → Weights olarak almak

istiyoruz bu yüzden reversed(topo) yapıyoruz.

- DFS şeklinde okların tersine göre iniyoruz, dipten çıkarken de düğümleri topo listesine atıyoruz, en sonda da çeviriyoruz

```
class Neuron:
    def __init__(self, nin):
        self.w = [Value(random.uniform(-1,1)) for _ in range(nin)]
        self.b = Value(np.random.uniform(-1,1))

    def __call__(self, x):
        act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b)
        out = act.tanh()
        return out

    def parameters(self):
        return self.w + [self.b]
```

- __init __ te her nöron için ağırlık ve bias belirliyoruz. nin: number of inputs
- __call __ da (w1×1 + w2×2 + ... + wn*xn) + b ve bu toplamayı yapıp tanh aktivasyon fonksiyonunu kullanıyor. Sonuç -1 ve 1 arasında oluyor.
- parameters de, parametreleri döndürüyor, mlp ilerde layerden isteyecek, layer da nörondan isteyecek

```
class Layer:
    def __init__(self, nin, nout):
        self.neurons = [Neuron(nin) for _ in range(nout)]

    def __call__(self, x):
        outs = [n(x) for n in self.neurons]
        return outs[0] if len(outs) == 1 else outs

    def parameters(self):
```

```
        return [p for neuron in self.neurons for p in neuron.parameters()]
```

- __init __ nin: her nötorona kaç kablo, nout: yan yana kaç nöron olacak.
  - Layer(3, 4) → 3 girişli nöronlardan yan yana 4 tane oluşturuyor.
- __call __ gelen x verisini katmandaki her nörona gönderir ve her biri farklı bir çıktı üretir. Eğer katmanda tek nöron varsa listeyi değil sayıyı döndürüyor direkt
-

```
class MLP:
    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1]) for i in range(len(nouts))]

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        return [p for layer in self.layers for p in layer.parameters()]
```

- nin: giriş sayısı, nouts: katman boyutlarının listesi.

```
    nin: Giriş sayısı (Örn: 3)
    # nouts: Katmanların boyutları listesi (Örn: [4, 4, 1])

    sz = [nin] + nouts # [3, 4, 4, 1] olur

    # Katmanları zincirleme oluşturur:
    # 1. Katman: 3 giriş → 4 çıkış
```

```
    # 2. Katman: 4 giriş → 4 çıkış
    # 3. Katman: 4 giriş → 1 çıkış
```

`MLP` sınıfı, içinde `Layer` nesneleri tutar. `Layer` nesneleri de içinde `Neuron` nesneleri tutar

- __call __, veri (x) ilk katmana girer, ilk katmandaki çıktı (x) bir sonraki katmanın girdisi olur (forward pass)

```
for k in range(20):

    #forward pass
    ypred = [n(x) for x in xs]
    loss = sum([(yout - ygt)**2 for ygt, yout in zip(ys, ypred)])

    #backward pass
    for p in n.parameters():
        p.grad = 0.0
    loss.backward()

    #update
    for p in n.parameters():
        p.data += -0.05 * p.grad

    print(k, loss.data)
```