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

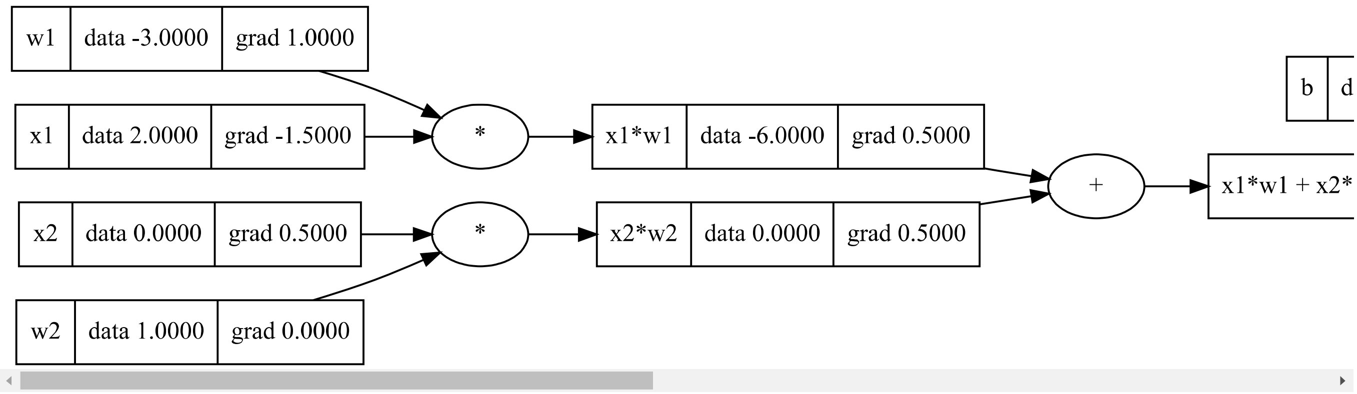
```
from graphviz import Digraph

def trace(root):
  # builds 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)
      for child in v._prev:
        edges.add((child, v))
        build(child)
  build(root)
  return nodes, edges

def draw_dot(root):
  dot = Digraph(format='svg', graph_attr={'rankdir': 'LR'}) # LR = left to right

  nodes, edges = trace(root)
  for n in nodes:
    uid = str(id(n))
    # for any value in the graph, create a rectangular ('record') node for it
    dot.node(name = uid, label = "{ %s | data %.4f | grad %.4f }" % (n.label, n.data, n.grad), shape='record')
    if n._op:
      # if this value is a result of some operation, create an op node for it
      dot.node(name = uid + n._op, label = n._op)
      # and connect this node to it
      dot.edge(uid + n._op, uid)

  for n1, n2 in edges:
    # connect n1 to the op node of n2
    dot.edge(str(id(n1)), str(id(n2)) + n2._op)

  return dot
```

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

    def __repr__(self):
        return f"Value(data={self.data})"

    def __rmul__(self, other):
        return self * other

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

    def __sub__(self, other):
        return self + (-other)

    def __neg__(self):
        return self * -1

    def __add__(self, other):

        other = other if isinstance(other, Value) else Value(other)

        out = Value(self.data + other.data, (self, other), '+')

        def _backward():
            # TODO:1
            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)
```

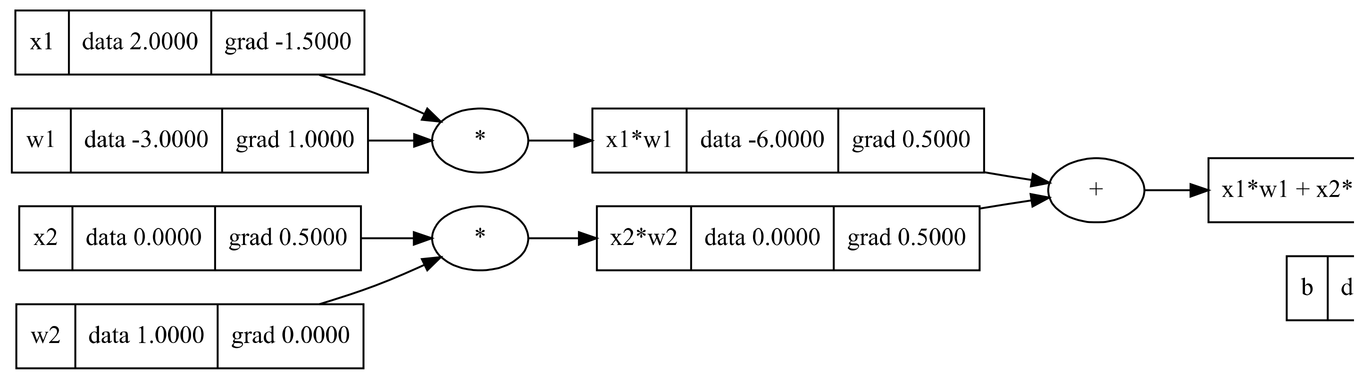
```
        out = Value(self.data * other.data, (self, other), '*')

        def _backward():
            # TODO:2
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward

        return out

    def tanh(self):
        x = self.data
        # TODO:3a
        t = (math.exp(2 * x) - 1) / (math.exp(2 * x) + 1)
        out = Value(t, (self,), 'tanh')

        def _backward():
            # TODO:3b
            self.grad += (1 - t**2) * out.grad
        out._backward = _backward

        return out

    def exp(self):
        x = self.data
        # TODO:4a
        e = math.exp(x)
        out = Value(e, (self,), 'exp')

        def _backward():
            # TODO:4b
            self.grad += e * out.grad
        out._backward = _backward

        return out

    def __truediv__(self, other):
        return self * other**-1

    def __pow__(self, other):
        assert isinstance(other, (int, float))
        out = Value(self.data ** other, (self, ), f'pow({other})')

        def _backward():
            # TODO:5
            self.grad += (other * self.data ** (other - 1)) * out.grad
        out._backward = _backward

        return out

    def backward(self):

        list_ = []
        visited = set()

        def build_topo(node):
          if node not in visited:
            visited.add(node)
            for child in node._prev:
              build_topo(child)
            list_.append(node)

        build_topo(self)

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

```
# inputs x1,x2
x1 = Value(2.0, label='x1')
x2 = Value(0.0, label='x2')
# weights w1,w2
w1 = Value(-3.0, label='w1')
w2 = Value(1.0, label='w2')
# bias of the neuron
b = Value(6.8813735870195432, label='b')
# x1*w1 + x2*w2 + b
x1w1 = x1*w1; x1w1.label = 'x1*w1'
```

```
x2w2 = x2*w2; x2w2.label = 'x2*w2'
x1w1x2w2 = x1w1 + x2w2; x1w1x2w2.label = 'x1*w1 + x2*w2'
n = x1w1x2w2 + b; n.label = 'n'
o = n.tanh(); o.label = 'o'


# visualize the gradients and forward
o.backward()
draw_dot(o)
```




```
# inputs x1,x2
x1 = Value(2.0, label='x1')
x2 = Value(0.0, label='x2')
# weights w1,w2
w1 = Value(-3.0, label='w1')
w2 = Value(1.0, label='w2')
# bias of the neuron
b = Value(6.8813735870195432, label='b')
# x1*w1 + x2*w2 + b
x1w1 = x1*w1; x1w1.label = 'x1*w1'
x2w2 = x2*w2; x2w2.label = 'x2*w2'
x1w1x2w2 = x1w1 + x2w2; x1w1x2w2.label = 'x1*w1 + x2*w2'
n = x1w1x2w2 + b; n.label = 'n'
o1 = 2*n; o1.label = 'o1'
o2 = o1.exp(); o2.label = 'o2'
o = (o2-1) / (o2+1); o.label = 'o'


# visualize the gradients and forward
o.backward()
draw_dot(o)
```




```
import random
class Neuron:

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

  def __call__(self, x):
    # w * x + b
    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]

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()]

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()]


xs = [
  [2.0, 3.0, -1.0],
  [3.0, -1.0, 0.5],
  [0.5, 1.0, 1.0],
  [1.0, 1.0, -1.0],
]
ys = [1.0, -1.0, -1.0, 1.0] # desired targets


# define a mlp and train a model
mlp = MLP(3, [4, 4, 1])


for e in range(10):
  params = mlp.parameters()

  pred = [mlp(x) for x in xs]
  loss = sum([(y_pred - y)**2 for y_pred, y in zip(pred, ys)])
  for p_ in params:
    p_.grad = 0

  loss.backward()

  for p_ in params:
    p_.data += -0.05 * p_.grad

  print(loss)
```

```
Value(data=7.076256232046454)
Value(data=4.264053104895217)
Value(data=2.6039232454276435)
Value(data=0.5429629595269161)
Value(data=0.2496414817084757)
Value(data=0.19527146536026213)
Value(data=0.16124082642121662)
Value(data=0.13746769989476085)
Value(data=0.11980321829746857)
Value(data=0.10611666750257556)
```

```
print([mlp(x) for x in xs])
```

```
[Value(data=0.8836496275431243), Value(data=-0.8201228718491383), Value(data=-0.8835789223013756), Value(data=0.8109627234841547)]
```