(a) A reflection matrix for a line at angle θ

$$R_{reflect}(\theta) = \begin{bmatrix} \cos(2\theta) & \sin(2\theta) \\ \sin(2\theta) & -\cos(2\theta) \end{bmatrix} \quad (1)$$

If reflect about two lines at angle α, β

$$T = R_{reflect}(\beta) \cdot R_{reflect}(\alpha)$$

$$= \begin{bmatrix} \cos(2\beta) & \sin(2\beta) \\ \sin(2\beta) & -\cos(2\beta) \end{bmatrix} \begin{bmatrix} \cos(2\alpha) & \sin(2\alpha) \\ \sin(2\alpha) & -\cos(2\alpha) \end{bmatrix}$$

$$= \begin{bmatrix} \cos(2\beta - 2\alpha) & -\sin(2\beta - 2\alpha) \\ \sin(2\beta - 2\alpha) & \cos(2\beta - 2\alpha) \end{bmatrix}$$

$$= \begin{bmatrix} \cos(2(\beta-\alpha)) & -\sin(2(\beta-\alpha)) \\ \sin(2(\beta-\alpha)) & \cos(2(\beta-\alpha)) \end{bmatrix}$$

For rotation, when $\theta = 2(\beta - \alpha)$

$$R_{rotate}(2(\beta-\alpha)) = \begin{bmatrix} \cos(2(\beta-\alpha)) & -\sin(2(\beta-\alpha)) \\ \sin(2(\beta-\alpha)) & \cos(2(\beta-\alpha)) \end{bmatrix}$$

$$T = R_{rotate}(2(\beta-\alpha))$$

Thus, reflect about α followed by about β is equivalent to a rotation of $2(\beta-\alpha)$

(b)

A skew-symmetric matrix $\hat{s}$, associated with a vector $s = [s_1, s_2, s_3]^T$ is defined as

$$\hat{s} = \begin{bmatrix} 0 & -s_3 & s_2 \\ s_3 & 0 & -s_1 \\ -s_2 & s_1 & 0 \end{bmatrix}$$

Compute the cross product $\hat{s}v = s \times v$, where v is any vector

Rodrigues' formula:

$$R = I + \sin(\phi)\hat{s} + (1-\cos(\phi))\hat{s}^2 \qquad (1)$$

The matrix exponential of $\hat{s}\phi$ is given by

$$e^{\hat{s}\phi} = I + \frac{\hat{s}\phi}{1!} + \frac{(\hat{s}\phi)^2}{2!} + \frac{(\hat{s}\phi)^3}{3!} + \cdots$$

According to the property of skew-symmetric matrix that $\hat{s}^T = -\hat{s}$ and $\hat{s}^2 = -I$

The term that the power is odd will have $\pm\hat{s}$,

and the even ones will have $\pm I$

$$\text{Then } e^{\hat{s}\phi} = I + \left(\phi\hat{s} + \frac{(\hat{s}\phi)^3}{3!} + \cdots\right) + \left(\frac{(\hat{s}\phi)^2}{2!} + \frac{(\hat{s}\phi)^4}{4!} + \cdots\right)$$

$$= I + \sin(\phi)\hat{s} + (1-\cos(\phi))\hat{s}^2 \qquad (2)$$

Equation (1) and (2) are equivalent.

(c) For python function to solve this, see below

```python
import numpy as np
from scipy.linalg import eig

def computeRotationMatrix(s, phi):
    s = s / np.linalg.norm(s)
    s1, s2, s3 = s
    I = np.eye(3)

    s_hat = np.array([
        [0, -s3, s2],
        [s3, 0, -s1],
        [-s2, s1, 0]
    ])

    # Rodriguez
    R = I + np.sin(phi) * s_hat + (1 - np.cos(phi)) * np.dot(s_hat, s_hat)
    return R


s = np.array([1, 1, 1])
phi = np.pi / 4

# Compute orthogonal matrix
R = computeRotationMatrix(s, phi)

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = eig(R)

# Verify cos(phi) = 1/2 * (trace(R) - 1)
cos_phi = 0.5 * (np.trace(R) - 1)

# Test points before and after rotation
points = np.array([
    [1, 0, 0],  # Along x-axis
    [0, 1, 0],  # Along y-axis
    [0, 0, 1],  # Along z-axis
    [1, 1, 1],  # Along the rotation axis
    [1, -1, 0], # Diagonal in xy-plane
    [0, 1, -1], # Diagonal in yz-plane
    [-1, 0, 1], # Diagonal in xz-plane
    [1, 2, 3],  # Arbitrary point
    [-2, -1, 0] # Arbitrary point
])
rotated_points = np.dot(R, points.T).T
```

For output result script,

```python
# Results
print("Rotation Matrix R:")
print(R,'\n')
print("Eigenvalues:")
print(eigenvalues,'\n')
print("Eigenvectors:")
print(eigenvectors,'\n')
print(f"1/2(trace(R)-1): {cos_phi}, cos(phi): {np.cos(phi)} \n")
print("Points before rotation:")
print(points, '\n')
print("Points after rotation:")
print(rotated_points)
```

For test results,

```
(openvla) PS C:\Users\16690\Desktop> & E:/Anaconda/envs/openvla/python.exe c:/Users/16690/Desktop/280hw1.py
Rotation Matrix R:
[[ 0.80473785 -0.31061722  0.50587936]
 [ 0.50587936  0.80473785 -0.31061722]
 [-0.31061722  0.50587936  0.80473785]]

Eigenvalues:
[0.70710678+0.70710678j 0.70710678-0.70710678j 1.        +0.j        ]

Eigenvectors:
[[-0.57735027+0.j  -0.57735027-0.j   0.57735027+0.j ]
 [ 0.28867513+0.5j  0.28867513-0.5j  0.57735027+0.j ]
 [ 0.28867513-0.5j  0.28867513+0.5j  0.57735027+0.j ]]

1/2(trace(R)-1): 0.7071067811865475, cos(phi): 0.7071067811865476

Points before rotation:
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [ 1  1  1]
 [ 1 -1  0]
 [ 0  1 -1]
 [-1  0  1]
 [ 1  2  3]
 [-2 -1  0]]

Points after rotation:
[[ 0.80473785  0.50587936 -0.31061722]
 [-0.31061722  0.80473785  0.50587936]
 [ 0.50587936 -0.31061722  0.80473785]
 [ 1.          1.          1.        ]
 [ 1.11535507 -0.29885849 -0.81649658]
 [-0.81649658  1.11535507 -0.29885849]
 [-0.29885849 -0.81649658  1.11535507]
 [ 1.70114151  1.18350342  3.11535507]
 [-1.29885849 -1.81649658  0.11535507]]
```

verify

$\cos(\phi) = \frac{1}{2}\{\text{trace}(R) - 1\}$

The relationship between eigenvalues, eigenvectors and the axis vector:

① Eigenvector corresponding to $\lambda = 1$ is the axis of rotation because points along this vector are not rotated

② $\lambda = e^{i\phi}$ and $\lambda = e^{-i\phi}$ (i.e. $0.7071 + 0.7071j$, $0.7071 - 0.7071j$ in this case)

represent how other points in the plane perpendicular to the rotation axis are transformed.

(d)

R is a rotation matrix if $R^T R = I$ & $\det(R) = 1$

The eigenvector corresponding to $\lambda = 1$ is the axis of rotation

The matrix $R - R^T$ is skew-symmetric

$$R - R^T = 2\sin(\phi)\,\hat{s}$$

if we need to derive s and ϕ,

we should ① compute $R - R^T$ and $\hat{s}$

② normalize $\hat{s}$ to find rotation axis

③ compute ϕ

The code is below:

```python
import numpy as np
from questionC import computeRotationMatrix

def computeAxisAndAngle(R):
    skew_symmetric = R - R.T

    sin_phi = np.linalg.norm(skew_symmetric) / 2
    cos_phi = (np.trace(R) - 1) / 2

    phi = np.arctan2(sin_phi, cos_phi)

    # Extract the axis of rotation
    s = np.array([
        skew_symmetric[2, 1],
        skew_symmetric[0, 2],
        skew_symmetric[1, 0]
    ])

    # Normalize
    s = s / (2 * sin_phi)
    return s, phi


phi = np.pi / 4
s = np.array([1, 1, 1]) / np.sqrt(3)
R = computeRotationMatrix(s, phi)
recovered_s, recovered_phi = computeAxisAndAngle(R)

print("Original Axis of Rotation (s):", s)
print("Recovered Axis of Rotation (s):", recovered_s)
print("Original Rotation Angle (phi):", phi)
print("Recovered Rotation Angle (phi):", recovered_phi)
```

```
(openvla) PS E:\Opensourced-Notes-SWE-MLE-JobCreator\Opensourced-Notes-SWE-MLE-JobCreator\Computer Vision> & E:/Anaconda/envs/openvla/python.exe "e:/Opensourced-Notes-SWE-MLE-JobCreator/Opensourced-Notes-SWE-MLE-JobCreato
r/Computer Vision/questionD.py"
Rotation Matrix R:
[[ 0.80473785 -0.31061722  0.50587936]
 [ 0.50587936  0.80473785 -0.31061722]
 [-0.31061722  0.50587936  0.80473785]]

Eigenvalues:
[0.70710678+0.70710678j 0.70710678-0.70710678j 1.        +0.j        ]

Eigenvectors:
[[-0.57735027+0.j  -0.57735027-0.j   0.57735027+0.j ]
 [ 0.28867513+0.5j  0.28867513-0.5j  0.57735027+0.j ]
 [ 0.28867513-0.5j  0.28867513+0.5j  0.57735027+0.j ]]

1/2(trace(R)-1): 0.7071067811865475, cos(phi): 0.7071067811865476

Points before rotation:
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [ 1  1  1]
 [ 1 -1  0]
 [ 0  1 -1]
 [-1  0  1]
 [ 1  2  3]
 [-2 -1  0]]

Points after rotation:
[[ 0.80473785  0.50587936 -0.31061722]
 [-0.31061722  0.80473785  0.50587936]
 [ 0.50587936 -0.31061722  0.80473785]
 [ 1.          1.          1.        ]
 [ 1.11535507 -0.29885849 -0.81649658]
 [-0.81649658  1.11535507 -0.29885849]
 [-0.29885849 -0.81649658  1.11535507]
 [ 1.70114151  1.18350342  3.11535507]
 [-1.29885849 -1.81649658  0.11535507]]
Original Axis of Rotation (s): [0.57735027 0.57735027 0.57735027]
Recovered Axis of Rotation (s): [0.40824829 0.40824829 0.40824829]
Original Rotation Angle (phi): 0.7853981633974483
Recovered Rotation Angle (phi): 0.9553166181245093
```

(e) The transformation E can be written as

$$Eu_j = Ru_j + t$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad t = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

We want to minimize the error

$$\min(\text{error}) = \sum_{j=1}^{4} \| Ru_j + t - v_j \|^2$$

$$\bar{u} = \frac{1}{n}\sum_{j=1}^{n} u_j \qquad \bar{v} = \frac{1}{n}\sum_{j=1}^{n} v_j$$

$u_j' = u_j - \bar{u}$, $v_j' = v_j - \bar{v}$ are centered points.

Use SVD to solve R

$\downarrow$

then, $t = \bar{v} - R\bar{u}$

Code is in next page.

```python
import numpy as np

def findBestTransformation(u, v):
    u_mean, v_mean = np.mean(u, axis=0), np.mean(v, axis=0)
    u_centered, v_centered = u - u_mean, v - v_mean

    covariance_matrix = np.dot(u_centered.T, v_centered)

    U, S, Vt = np.linalg.svd(covariance_matrix)

    R = np.dot(Vt.T, U.T)
    t = v_mean - np.dot(R, u_mean)
    return R, t

# Test
u = np.array([[-3, 0], [1, 1], [1, 0], [1, -1]])
v = np.array([[0, 3], [1, 0], [0, 0], [-1, 0]])

R, t = findBestTransformation(u, v)


print("Optimal Rotation Matrix (R):")
print(R, '\n')
print("Optimal Translation Vector (t):")
print(t, '\n')

# Verify the transformation
u_transformed = np.dot(u, R.T) + t
print("Transformed Points:")
print(u_transformed, '\n')
print("Target Points:")
print(v)
```

```
(openvla) PS E:\Opensourced-Notes-SWE-MLE-JobCreator\Opensourced-Notes-SWE-MLE-JobCreator\Computer Vision> & E:/Anaconda/envs/openvla/python.exe "e:/Opensourced-Notes-SWE-MLE-JobCreator/Opensourced-Notes-SWE-MLE-JobCreato
r/Computer Vision/questionE.py"
Optimal Rotation Matrix (R):
[[ 0.  1.]
 [-1.  0.]]

Optimal Translation Vector (t):
[0.   0.75]

Transformed Points:
[[ 0.    3.75]
 [ 1.   -0.25]
 [ 0.   -0.25]
 [-1.   -0.25]]

Target Points:
[[ 0  3]
 [ 1  0]
 [ 0  0]
 [-1  0]]
```

(f)

A line on a plane can be written as

$$x(t) = p + td$$

p is the point on the line

d is the direction vector of the line

Assume the camera is at origin and the projection is central. A 3D point $X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ projects to $X_{image} = \begin{bmatrix} \frac{x}{z} \\ \frac{y}{z} \end{bmatrix}$

For large t,

$$X_{vanishing} = \lim_{t \to \infty} \frac{p + td}{z} = \frac{d}{d_z}$$

Consider the plane $n^T x + c = 0$

$n = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$ is the normal vector of the plane

c is a constant.

all lines on the plane have direction d satisfying : $n^T d = 0$

All vanishing points lie on this line in the image plane where the plane intersects the image plane.

set $z = 1$ for projection

then $$n^T [x, y, 1]^T + c = 0$$

$$\Rightarrow n_x x + n_y y + n_z + c = 0$$

The vanishing points of all lines on a plane lie on the vanishing line, which is the porjection of the plane's 3D geometry onto the image plane. The vanishing line is determined by the normal vector n and the plane constant c and its equation in 2D is

$$n_x x + n_y y + n_z + c = 0$$

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

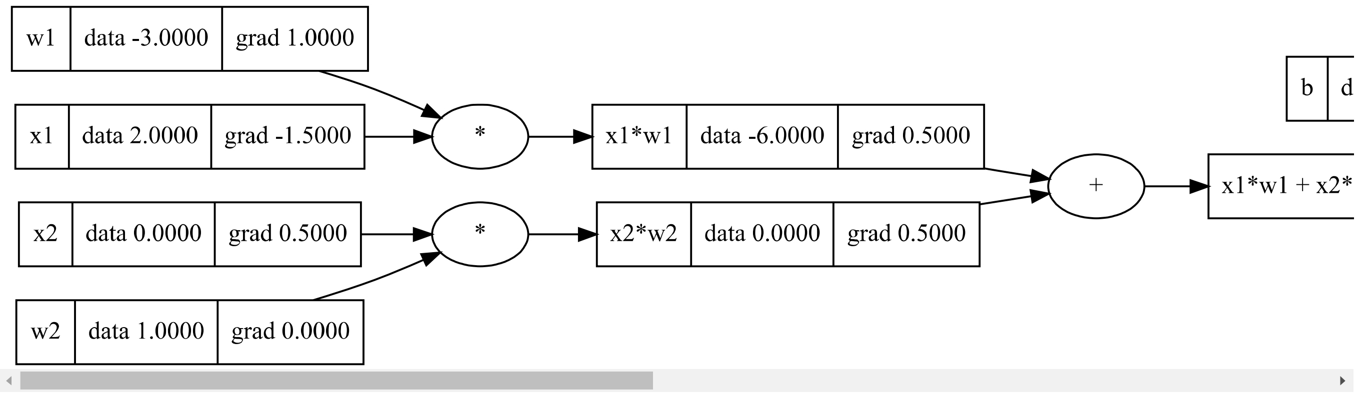
```
from graphviz import Digraph

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)
```

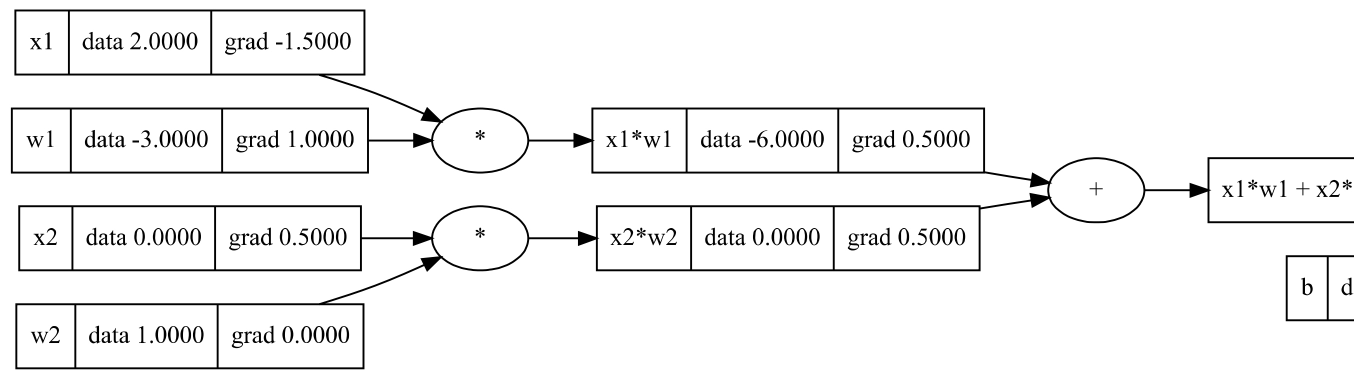
```
        out = Value(self.data * other.data, (self, other), '*')

        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):
```

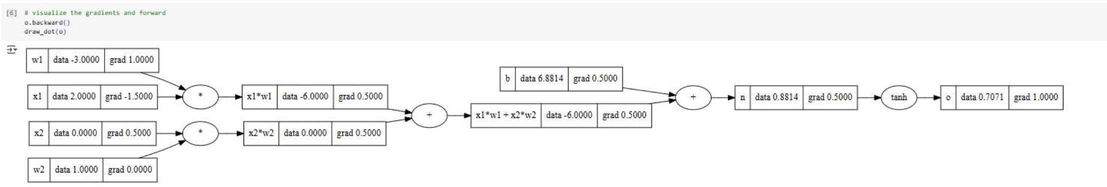
```
    return self.w + [self.b]

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)
```

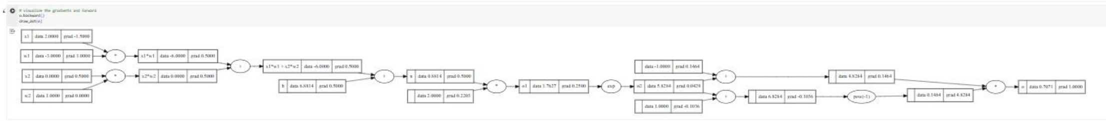
```
Value(data=7.076256232046454)
Value(data=4.264053104895217)
Value(data=2.6039232454276435)
Value(data=0.5429629595269161)
Value(data=0.2496414817084757)
Value(data=0.19527146536026213)
Value(data=0.16124082642121662)
Value(data=0.13746769989476085)
Value(data=0.11980321829746857)
Value(data=0.10611666750257556)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

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

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

Full image of the cell output





(a) Are we strictly required to use our atomic operations when defining new functions (e.g., sigmoid)? Under what conditions can we define new operations?

We do not have to stick strictly to atomic operations because we can create new functions, like sigmoid, by combining existing ones. The key is that these new functions must fit within the computational graph, have a well-defined gradient for backpropagation, and can work with other parts of the graph. By doing this, we can expand the framework and keep it compatible at same time to ensure that backpropagation remains accurate.

(b) When performing backpropagation on a Value, why do we accumulate the gradient as opposed to directly assigning the gradient?

Gradients are accumulated during backpropagation because a single variable in a computational graph can affect the output through multiple paths, and each path contributes to the final gradient. By summing these contributions, we can ensure the gradient is calculated correctly. This is essential due to the chain rule, which combines gradients from downstream nodes. If we directly assigned the gradient, it could overwrite previous contributions and result in errors. Accumulation ensures all contributions are included, maintaining the consistency and accuracy of the optimization process.