CSED 502: Computer Vision and Deep Learning

Solutions for Tutorial: NumPy Fundamentals & Backpropagation

Thanks for attending, we hope you found class helpful.

Reference Material

Rules of Broadcasting from Jake VanderPlas' *Python Data Science Handbook*:

(1) If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

(2) If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

(3) If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Chain Rule for One Independent Variable:

Let $z = f(x, y)$ be a differentiable function. Further suppose that x and y are themselves differentiable functions of t, in other words $x = x(t)$ and $y = y(t)$. Then,

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$$

Chain Rule for Two Independent Variables:

Let $z = f(x, y)$ be a differentiable function, where x and y are themselves differentiable functions of a and b. In other words, $x = x(a, b)$ and $y = y(a, b)$. Then,

$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial a} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial a}$$

and

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial b} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial b}$$

Generalized Chain Rule:

Let $w = f(x_1, x_2, \ldots, x_m)$ be a differentiable function of m independent variables, and let $x_i = x_i(t_1, t_2, \ldots, t_n)$ be a differentiable function of n independent variables. Then,

$$\frac{\partial w}{\partial t_j} = \frac{\partial w}{\partial x_1}\frac{\partial x_1}{\partial t_j} + \frac{\partial w}{\partial x_2}\frac{\partial x_2}{\partial t_j} + \ldots + \frac{\partial w}{\partial x_m}\frac{\partial x_m}{\partial t_j}$$

for any $j \in 1, 2, \ldots, n$.

Intuition for Backprop

Recall some basic facts:

1) The loss function L measures how "bad" our current model is.
2) L is a function of our parameters W.
3) We want to minimize L.

Thus, we update W to minimize L using $\frac{\partial L}{\partial W}$.

For example, if $\frac{\partial L}{\partial W_1}$ was positive, increasing W_1 would increase L. Accordingly, we'd choose to decrease W_1.

More generally, `weights += (-1 * step_size * gradient)`.

Unfortunately, taking the derivative $\frac{\partial L}{\partial W}$ can get extremely difficult, especially at the scale of state-of-the-art models. For instance, GLM-4.5 has 92 hidden layers and 32 billion parameters. Imagine taking 32 billion derivatives, with each derivative having hundreds of applications of chain rule.

Instead, we employ a technique known as **backprop**.

First, we split our function into multiple equations until there is *one operation per equation*. This process is known as **staged computation**. Next, we take the derivatives of each of these smaller equations, before finally linking them together using **chain rule**.

Common Gates

Feel free to take notes on the common backprop gates here.

1. Dimension: Impossible

Determine if NumPy allows the **addition** of the following pairs of arrays, and if applicable determine what the result's dimensions will be.

(a) Where `x.shape` is $(2,)$ and `y.shape` is $(2,1)$

Solution:

Yes. $(2,2)$.

(b) Where `x.shape` is $(4,)$ and `y.shape` is $(4,1,1)$

Solution:

Yes. $(4,1,4)$.

(c) Where `x.shape` is $(4,2)$ and `y.shape` is $(2,4,1)$

Solution:

Yes. $(2,4,2)$.

(d) Where `x.shape` is $(8,3)$ and `y.shape` is $(2,8,1)$

Solution:

Yes. $(2,8,3)$.

(e) Where `x.shape` is $(6,5,3)$ and `y.shape` is $(6,5)$

Solution:

No. However, if we changed `y.shape` to be $(6,5,1)$, then we would get a valid operation that results in an array of shape $(6,5,3)$. This could be achieved in NumPy by calling either `x + y[:, :, None]` or `x + y[:, :, np.newaxis]` instead of `x + y`.

Determine if NumPy allows the **matrix multiplication** of the following pairs of arrays, and if applicable determine what the result's dimensions will be.

(f) Where `a.shape` is $(5, 4)$ and `b.shape` is $(4, 8)$.

Solution:

Yes. $(5, 8)$.

(g) Where `a.shape` is $(3, 5, 4)$ and `b.shape` is $(3, 4, 8)$.

Solution:

Yes. $(3, 5, 8)$.

(h) Where `a.shape` is $(3, 5, 4)$ and `b.shape` is $(5, 4, 8)$.

Solution:

No. The batch dimension is not compatible.

(i) Where `a.shape` is $(1, 5, 4)$ and `b.shape` is $(5, 4, 8)$.

Solution:

Yes. $(5, 5, 8)$. Unlike the prior example, we successfully broadcast the batch dimension.

(j) Where `a.shape` is $(2, 5, 4)$ and `b.shape` is $(3, 2, 4, 8)$.

Solution:

Yes. $(3, 2, 5, 8)$.

2. The More (Derivatives) The Merrier

(a) Let $z = 2x + y$, with $x = \ln(t)$ and $y = \frac{1}{3}t^3$. Find $\frac{dz}{dt}$.

Solution:

$$\begin{aligned}
\frac{dz}{dt} &= \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} \\
&= \frac{\partial}{\partial x}\Big(2x+y\Big) \cdot \frac{\partial}{\partial t}\Big(\ln(t)\Big) + \frac{\partial}{\partial y}\Big(2x+y\Big)\frac{\partial}{\partial t}\left(\frac{1}{3}t^3\right) && \text{Chain Rule} \\
&= 2 \cdot \frac{1}{t} + 1 \cdot t^2 && \text{Solve Partial Derivatives} \\
&= t^2 + \frac{2}{t}
\end{aligned}$$

(b) Let $z = x^2y - y^2$ where $x = t^2$ and $y = 2t$. Find $\frac{dz}{dt}$. Your answer should be in terms of t.

Solution:

$$\begin{aligned}
\frac{dz}{dt} &= \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} && \text{Chain Rule} \\
&= (2xy)(2t) + (x^2 - 2y)(2) && \text{Substitute Partial Derivatives} \\
&= \big(2(t^2)(2t)\big)\big(2t\big) + \big((t^2)^2 - 2(2t)\big)\big(2\big) && \text{Definitions of } x \text{ and } y \\
&= (4t^3)(2t) + 2(t^4 - 4t) \\
&= 8t^4 + 2t^4 - 8t \\
&= 10t^4 - 8t
\end{aligned}$$

(c) Let $z = 3x^2 - 2xy + y^2$. Also let $x = 3a + 2b$ and $y = 4a - b$. Find $\frac{\partial z}{\partial a}$ and $\frac{\partial z}{\partial b}$.

Solution:

$$\begin{aligned}
\frac{\partial z}{\partial a} &= \frac{\partial z}{\partial x}\frac{\partial x}{\partial a} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial a} && \text{Chain Rule} \\
&= (6x - 2y)(3) + (-2x + 2y)(4) && \text{Substitute Partial Derivatives} \\
&= 18x - 6y - 8x + 8y \\
&= 10x + 2y \\
&= 10(3a + 2b) + 2(4a - b) && \text{Definitions of } x \text{ and } y \\
&= 30a + 20b + 8a - 2b \\
&= 38a + 18b
\end{aligned}$$

$$\begin{aligned}
\frac{\partial z}{\partial b} &= \frac{\partial z}{\partial x}\frac{\partial x}{\partial b} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial b} && \text{Chain Rule} \\
&= (6x - 2y)(2) + (-2x + 2y)(-1) && \text{Substitute Partial Derivatives} \\
&= 12x - 4y + 2x - 2y \\
&= 14x - 6y \\
&= 14(3a + 2b) - 6(4a - b) && \text{Definitions of } x \text{ and } y \\
&= 42a + 28b - 24a + 6b \\
&= 18a + 34b
\end{aligned}$$

(d) Let $w = f(x, y, z)$, $x = x(t, u, v)$, $y = y(t, u, v)$ and $z = z(t, u, v)$. Find the formula for $\frac{\partial w}{\partial t}$.

Solution:

$$\frac{\partial w}{\partial t} = \frac{\partial w}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial w}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial w}{\partial z}\frac{\partial z}{\partial t}$$

3. Compute and Conquer

For each function below, use the staged computation approach to split it into smaller equations.

(a) $f(x, y, z) = (x + y)z$

Solution:

Decompose the function as follows:

- $a = x + y$
- $b = z$
- $f = ab$

(b) $h(x, y, z) = (x^2 + 2y)z^3$

Solution:

Decompose the function as follows:

- $a = x^2$
- $b = 2y$
- $c = a + b$
- $d = z^3$
- $h = cd$

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$

Solution:

Decompose the function as follows:

- $a = \ln(x)$
- $b = \sin(y)$
- $c = a + b$
- $d = c^2$
- $f = 4x$
- $g = d + f$

4. Oh, node way!

For each function below:

(i) construct a computational graph

(ii) do a forward and backward pass through the graph using the provided input values

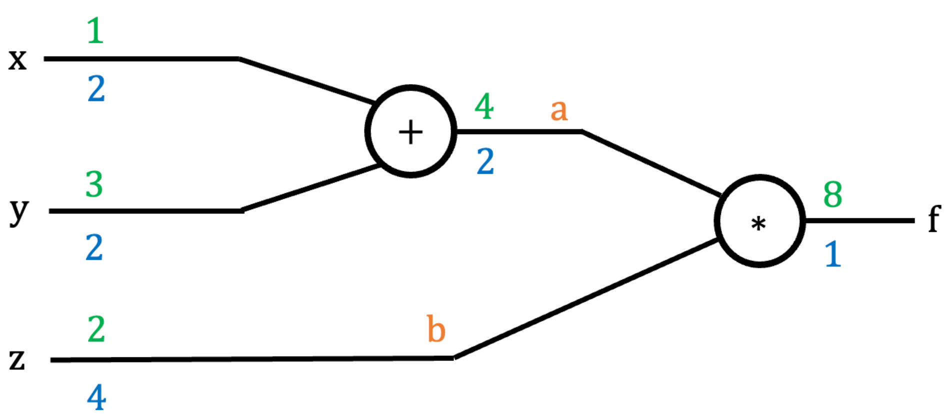
(iii) complete the Python function for a combined forward and backward pass

It may be useful to consider how you split these functions into smaller equations in the question above.

(a) $f(x, y, z) = (x + y)z$ with input values $x = 1, y = 3, z = 2$

Solution:

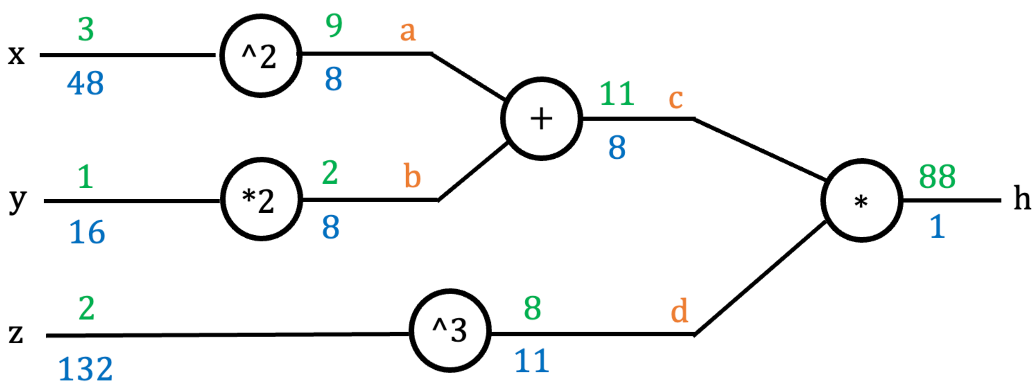
Forward pass values are displayed in green; backward pass values are displayed in blue. The orange letters correspond to the mini-equations from Question 1.




```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `fx`, `fy`, `fz` respectively
def q2a(x, y, z):
    # forward pass
    a = x + y
    b = z
    f = a * b
    out = f

    # backward pass
    ff = 1
    fb = ff * a
    fa = ff * b
    fz = fb * 1
    fx = fa
    fy = fa

    return out, fx, fy, fz
```

(b) $h(x, y, z) = (x^2 + 2y)z^3$ with input values $x = 3, y = 1, z = 2$

Solution:

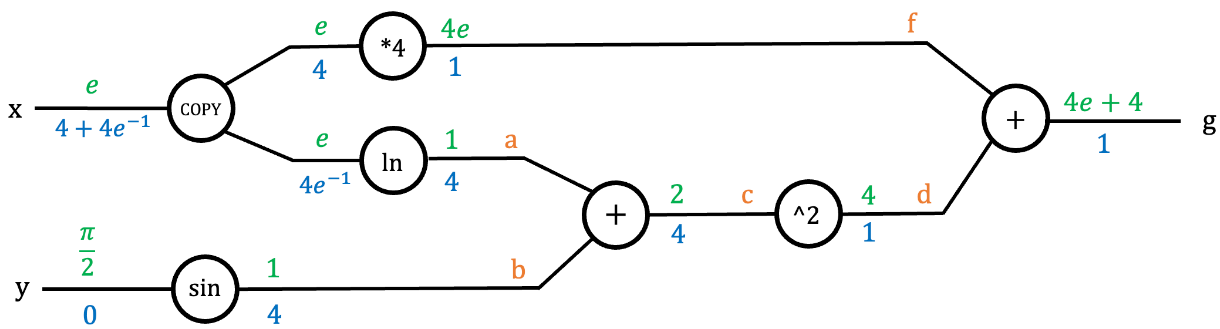



```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `hx`, `hy`, `hz` respectively
def q2b(x, y, z):
    # forward pass
    a = x ** 2
    b = 2 * y
    c = a + b
    d = z ** 3
    h = c * d
    out = h

    # backward pass -- right-most gate
    hh = 1
    hc = hh * d
    hd = hh * c

    # backward pass -- top branches
    ha = hc
    hb = hc
    hx = ha * (2 * x)
    hy = hb * 2

    # backward pass -- bottom branch
    hz = hd * (3 * (z ** 2))

    return out, hx, hy, hz
```

(c) $g(x, y, z) = \big(\ln(x) + \sin(y)\big)^2 + 4x$ with input values $x = e, y = \frac{\pi}{2}, z = 2$

Solution:

We omit z in the computational graph below since it does not appear in the formula for g. It is important to realize that the gradient with respect to z is 0.



A few observations:

- We have a gradient (4) flowing back to y, but it dies on the last gate since $\frac{d}{dy}(\sin(y)) = \cos(x)$ and $\cos(\frac{\pi}{2}) = 0$. This is problematic since it means we don't change y on this gradient descent step despite having feedback suggesting that y should be decremented.
- Since $\ln(x) = \frac{1}{x}$, the local gradient associated with equation a can be undefined if $x = 0$. If you were asked to implement this function and its backwards pass in Python, what are some potential workarounds you might employ?

Python function printed on the following page.

```python
import numpy as np

# inputs: NumPy arrays `x`, `y`, `z` of identical size
# outputs: forward pass in `out`, gradients for x, y, z in `gx`, `gy`, `gz` respectively
def q2c(x, y, z):
    # forward pass
    a = np.log(x)
    b = np.sin(y)
    c = a + b
    d = c ** 2
    f = 4 * x
    g = d + f
    out = g

    # backward pass -- right-most gate
    gg = 1
    gf = gg
    gd = gd

    # backward pass -- path via `d`
    gc = gd * (2 * c)
    ga = gc
    gb = gc
    gx_1 = ga * (x ** -1)
    gy = gb * np.cos(y)

    # backward pass -- path via `f`
    gx_2 = gf * 4

    # backward pass -- reconciliation at copy gate
    gx = gx_1 + gx_2

    # z never appears in the function, so it has no gradient
    gz = 0

    return out, gx, gy, gz
```

5. Sigmoid Shenanigans

Consider the Sigmoid activation function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

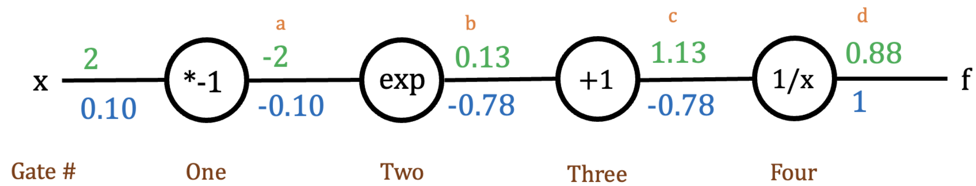
(a) Draw a computational graph and work through the backpropagation. Then, fill in the Python function. If you finish early, work through the analytical derivation for Sigmoid.

As a hint, you could split Sigmoid into the following functions:

$$a(x) = -x \qquad b(x) = e^x \qquad c(x) = 1 + x \qquad d(x) = \frac{1}{x}$$

Observe that chaining these operations gives us Sigmoid: $d(c(b(a(x)))) = \sigma(x)$.

Solution:



(b) Suppose $x = 2$. What would the gradient with respect to x be? Feel free to use a calculator on this part.

Solution:

Recall that downstream = upstream $\times$ local.

At Gate Four, the upstream gradient is 1 and the local gradient is $\frac{\partial}{\partial c}\Big(\frac{1}{c}\Big) = -\frac{1}{c^2} = -\frac{1}{(1.13)^2} = -0.78$. Thus, the downstream gradient is $1 \times -0.78 = -0.78$.

At Gate Three, the upstream is -0.78 and the local is $\frac{\partial}{\partial b}\Big(b+1\Big) = 1$. Thus, the downstream is $-0.78 \times 1 = -0.78$.

At Gate Two, the upstream is -0.78 and the local is $\frac{\partial}{\partial a}\Big(e^a\Big) = e^a = e^{-2} = 0.135$. Thus, the downstream is $-0.78 \times 0.135 = -0.10$.

At Gate One, the upstream is -0.10 and the local is $\frac{\partial}{\partial x}\Big(-x\Big) = -1$. Thus, the downstream is $-0.10 \times -1 = 0.10$.

Therefore, $\frac{df}{dx} \approx 0.10$. We use $\approx$ here because we rounded decimals throughout our calculations.

(c) You should have gotten around 0.1. If the step size is 0.2, what would the value of x be after taking one gradient descent step? As a hint, remember that `parameters -= step_size * gradient`.

Solution:

Our parameter, x, started off at 2. Our step size was 0.2 and our gradient is 0.1. Plugging into the equation for gradient descent, the new value for x is $2 - 0.2(0.1) = 2 - 0.02 = 1.98$.

(d) Implement the function below for a full forward and backward pass through Sigmoid.

Solution:

```python
import numpy as np

# inputs:
#   - a numpy array `x`
# outputs:
#   - `out`: the result of the forward pass
#   - `fx` : the result of the backward pass
def sigmoid(x):
    # provided: forward pass with cache
    a = -x
    b = np.exp(a)
    c = 1 + b
    f = 1/c
    out = f

    # TODO: backward pass, "fx" represents df / dx
    ff = 1
    fc = ff * -1/(c**2)
    fb = fc * 1
    fa = fb * np.exp(a)
    fx = fa * -1

    return out, fx
```

6. A Backprop a Day Keeps the Derivative Away

Consider the following function:

$$f = \frac{\ln x \cdot \sigma\left(\sqrt{y}\right)}{\sigma\left(\left(x+y\right)^2\right)}$$

Break the function up into smaller parts, then draw a computational graph and finish the Python function.

For reference, the derivative of Sigmoid is $\sigma(x) \cdot \big(1 - \sigma(x)\big)$.

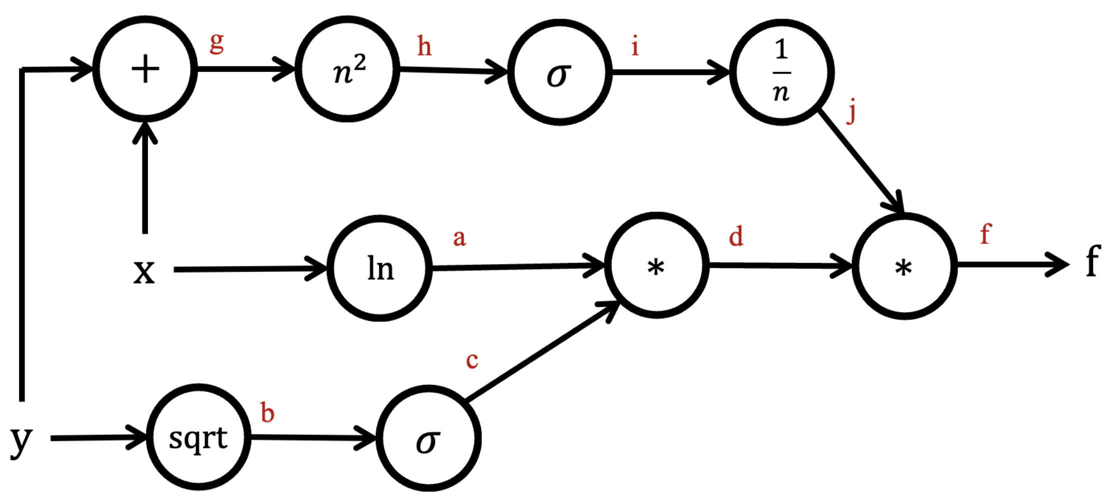
The TA solution breaks the function into 8 additional equations and rewrites f in terms of 2 of those additional equations. Yours doesn't have to match this exactly.

Solution:

We begin by breaking the function down:

Numerator:	$a = \ln x$	$b = \sqrt{y}$	$c = \sigma(b)$	$d = a \cdot c$
Denominator:	$g = x + y$	$h = g^2$	$i = \sigma(h)$	$j = \frac{1}{i}$
Final:	$f = dj$			

Although $f = \frac{d}{i}$ is a valid, one-operation gate, we generally try to avoid quotient rule. Therefore, we introduce an extra operation, $i = \frac{1}{j}$, leaving us with $f = di$.



Python function printed on the following page.

```python
import numpy as np

# helper function
def sigmoid(x):
    return 1/(1 +np.exp(-x))

# inputs: numpy arrays `x`, `y`
# outputs: forward pass in `out`, gradient for x in `fx`, gradient for y in `fy`
def complex_layer(x, y):
    # forward pass
    a = np.log(x)
    b = np.sqrt(y)
    c = sigmoid(b)
    d = a * c
    g = x + y
    h = g ** 2
    i = sigmoid(h)
    j = 1 / i
    out = d * j

    # backward pass -- output gate
    ff = 1
    fd = ff * j
    fj = ff * d

    # backward pass -- top branch
    fi = fj * -1 / (i ** 2)
    fh = fi * sigmoid(h) * (1 - sigmoid(h))
    fg = fh * 2 * g
    fx_1 = fg
    fy_1 = fg

    # backward pass -- middle branch
    fa = fd * c
    fx_2 = fa / x

    # backward pass -- bottom branch
    fc = fd * a
    fb = fc * sigmoid(b) * (1 - sigmoid(b))
    fy_2 = fb / (2 * np.sqrt(y))

    # backward pass -- reconciliation
    fx = fx_1 + fx_2
    fy = fy_1 + fy_2

    return out, fx, fy
```

7. Vector Virtuosity

Consider the following function,

$$f(W, x) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W * x)_i^2$$

where $W \in \mathbb{R}^{n \times n}$ and $x \in \mathbb{R}^n$.

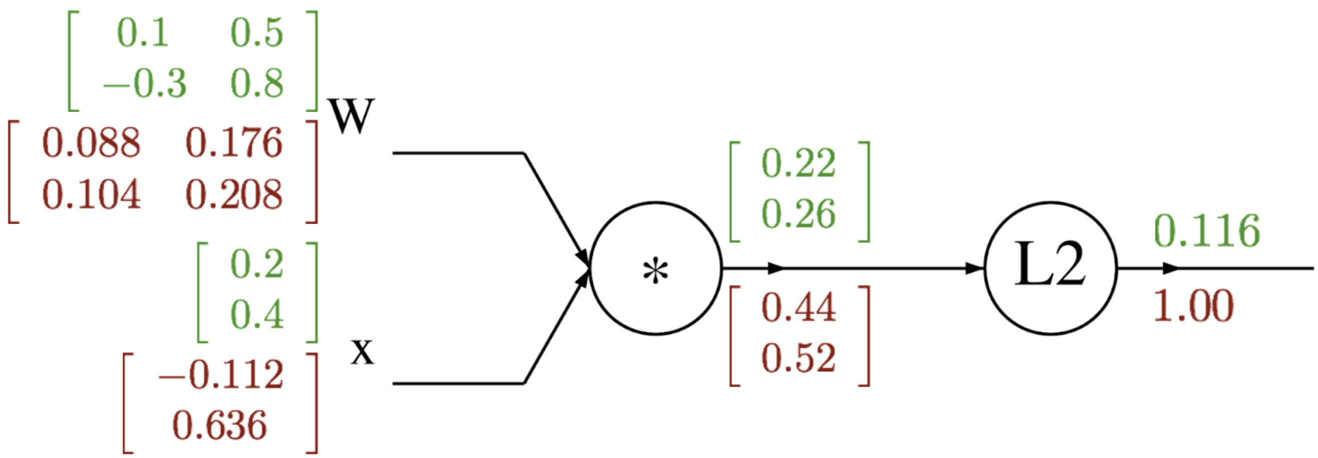
First draw the function's computation graph. Then compute the forward pass for the following inputs.

$$W = \begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix} \qquad x = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

Lastly, compute the backward pass. Verify your answer by deriving the closed forms of $\nabla_W f$ and $\nabla_x f$.

Solution:

The forward pass values are printed in green; the backward pass values are in red.



Note that labeling the final gate as "L2" is a bit misleading, since the function f omits the square root typical of an L2 norm. You are encouraged to use a more appropriate label for that gate (e.g., "squared norm").

If we label the intermediate value $q = Wx \in \mathbb{R}^n$, then the gradients can be written as follows.

$$\nabla_q f = 2q \qquad \nabla_W f = 2q \cdot x^T \qquad \nabla_x f = 2W^T \cdot q$$

If you are struggling to derive the gradients listed above, then you should first check to make sure you arrived at the derivatives listed below. Note that $\mathbf{1}_{\{k=i\}}$ is an indicator function which returns 1 iff $k = i$ and 0 otherwise.

- $\frac{\partial f}{\partial q_i} = 2q_i$
- $\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{\{k=i\}} x_j$ and $\frac{\partial q_k}{\partial x_i} = W_{k,i}$
- $\frac{\partial f}{\partial W_{i,j}} = \sum_{k=1}^{n} \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} = \sum_{k=1}^{n} (2q_k)(\mathbf{1}_{\{k=i\}} x_j) = 2q_i x_j$
- $\frac{\partial f}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} = \sum_{k=1}^{n} 2q_k \cdot W_{k,i}$