Lecture 4: Neural Networks and Backpropagation
Administrative: Assignment 1

Due 10/20 11:59pm

- K-Nearest Neighbor
- Linear classifiers: SVM, Softmax
- Two-layer neural network
- Image features
Administrative: Project proposal

Due **Friday 10/27**

Come to office hours to talk about potential ideas.

Use EdStem to find teammates
Administrative: EdStem

Please make sure to check and read all pinned EdStem posts.
Recap: from last time

\[ f(x, W) = Wx + b \]
Recap: loss functions

\[ s = f(x; W) = Wx \quad \text{Linear score function} \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM loss (or softmax)} \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2 \quad \text{data loss + regularization} \]
Finding the best W: Optimize with Gradient Descent

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Gradient descent

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

**Numerical gradient**: slow : (, approximate : (, easy to write : )
**Analytic gradient**: fast : ), exact : ), error-prone : (  

In practice: Derive analytic gradient, check your implementation with numerical gradient
Stochastic Gradient Descent (SGD)

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W) \]

Full sum expensive when \( N \) is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

# Vanilla Minibatch Gradient Descent

```python
while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
What we are going to discuss today!

\[ s = f(x; W) = WX \]  
Linear score function

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]  
SVM loss (or softmax)

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2 \]  
data loss + regularization

How to find the best W?  
\[ \nabla W L \]
Problem: Linear Classifiers are not very powerful

Visual Viewpoint

Linear classifiers learn one template per class

Geometric Viewpoint

Linear classifiers can only draw linear decision boundaries
Pixel Features

\[ f(x) = Wx \]

Class scores
Image Features

f(x) = Wx

Feature Representation

Class scores
Cannot separate red and blue points with linear classifier
Feature become linearly separable through a non-linear transformation

\[ f(x, y) = (r(x, y), \theta(x, y)) \]

Cannot separate red and blue points with linear classifier.

After applying feature transform, points can be separated by linear classifier.
Example: Color Histogram
Example: Histogram of Oriented Gradients (HoG)

Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins

Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005
Example: Bag of Words

**Step 1: Build codebook**
- Extract random patches
- Cluster patches to form “codebook” of “visual words”

**Step 2: Encode images**

Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories”, CVPR 2005
Combine many different features if unsure which features are better
Feature Extraction

10 numbers giving scores for classes

Training

One Solution: Non-linear feature transformation

\[ f(x, y) = (r(x, y), \theta(x, y)) \]

Transform data with a cleverly chosen **feature transform** \( f \), then apply linear classifier

- Color Histogram
- Histogram of Oriented Gradients (HoG)
Today: Neural Networks
Neural networks: the original linear classifier

(Before) Linear score function:

\[ f = Wx \]

\[ x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D} \]
Neural networks: 2 layers

(Before) Linear score function: \( f = WX \)

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\( x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H} \)

(In practice we will usually add a learnable bias at each layer as well)
Neural networks: also called fully connected network

(Before) Linear score function: \[ f = Wx \]

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\[ x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H} \]

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

(In practice we will usually add a learnable bias at each layer as well)
Neural networks: 3 layers

(Before) Linear score function: 

\[ f = Wx \]

(Now) 2-layer Neural Network or 3-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]

\[ x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2} \]

(In practice we will usually add a learnable bias at each layer as well)
Neural networks: hierarchical computation

(Before) Linear score function:
\[ f = W x \]

(Now) 2-layer Neural Network
\[ f = W_2 \max(0, W_1 x) \]

\[ x \in \mathbb{R}^D, \quad W_1 \in \mathbb{R}^{H \times D}, \quad W_2 \in \mathbb{R}^{C \times H} \]
Neural networks: learning 100s of templates

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network \( f = W_2 \max(0, W_1 x) \)

Learn 100 templates instead of 10.

Share templates between classes.
Neural networks: why is max operator important?

(Before) Linear score function: \[ f = W x \]

(Now) 2-layer Neural Network \[ f = W_2 \max(0, W_1 x) \]

The function \( \max(0, z) \) is called the activation function.

Q: What if we try to build a neural network without one?

\[ f = W_2 W_1 x \]
Neural networks: why is max operator important?

(Before) Linear score function: \( f = Wx \)

(Now) 2-layer Neural Network
\[
f = W_2 \max(0, W_1 x)
\]

The function \( \max(0, z) \) is called the activation function.

Q: What if we try to build a neural network without one?
\[
f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, \quad f = W_3 x
\]

A: We end up with a linear classifier again!
Activation functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]

**Leaky ReLU**
\[ \max(0.1x, x) \]

**Maxout**
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Activation functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1+e^{-x}} \]

**ReLu**
\[ \text{max}(0, x) \]

**Tanh**
\[ \tanh(x) \]

**Leaky ReLU**
\[ \text{max}(0.1x, x) \]

**Maxout**
\[ \text{max}(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Neural networks: Architectures

“2-layer Neural Net”, or “1-hidden-layer Neural Net”

“Fully-connected” layers

“3-layer Neural Net”, or “2-hidden-layer Neural Net”
Example feed-forward computation of a neural network

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x))  # activation function (use sigmoid)
x = np.random.randn(3, 1)  # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1)  # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)  # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3  # output neuron (1x1)
```
Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
y_pred = h.dot(w2)
loss = np.square(y_pred - y).sum()
print(t, loss)

    grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h.T.dot(grad_y_pred)
grad_h = grad_y_pred.dot(w2.T)
grad_w1 = x.T.dot(grad_h * h * (1 - h))
w1 -= 1e-4 * grad_w1
w2 -= 1e-4 * grad_w2
```
Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h.T.dot(grad_y_pred)
    grad_h = grad_y_pred.dot(w2.T)
    grad_w1 = x.T.dot(grad_h * h * (1 - h))

    w1 -= 1e-4 * grad_w1
    w2 -= 1e-4 * grad_w2
```
Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h.T.dot(grad_y_pred)
    grad_h = grad_y_pred.dot(w2.T)
    grad_w1 = x.T.dot(grad_h * h * (1 - h))

    w1 -= 1e-4 * grad_w1
    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass
Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    print(t, loss)

grad_y_pred = 2.0 * (y_pred - y)
gw2 = h.T.dot(grad_y_pred)
g_h = grad_y_pred.dot(w2.T)
gw1 = x.T.dot(g_h * h * (1 - h))

w1 -= 1e-4 * gw1
w2 -= 1e-4 * gw2
```

Define the network

Forward pass

Calculate the analytical gradients
Full implementation of training a 2-layer Neural Network needs ~20 lines:

```python
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h.T.dot(grad_y_pred)
    grad_h = grad_y_pred.dot(w2.T)
    grad_w1 = x.T.dot(grad_h * h * (1 - h))

    w1 -= 1e-4 * grad_w1
    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent
Setting the number of layers and their sizes

more neurons = more capacity
Do not use size of neural network as a regularizer. Use stronger regularization instead:

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]
Impulses carried toward cell body

dendrite

axon

presynaptic terminal

cell body

Impulses carried away from cell body

This image by Felipe Perucho is licensed under CC-BY 3.0
Impulses carried toward cell body

Impulses carried away from cell body

This image by Felipe Perucho is licensed under CC-BY 3.0

Ali Farhadi, Aditya Kusupati
Impulses carried toward cell body

Impulses carried away from cell body

dendrite

presynaptic terminal

axon

This image by Felipe Perucho is licensed under CC-BY 3.0

sigmoid activation function

\[
\frac{1}{1 + e^{-x}}
\]
Impulses carried toward cell body

Impulses carried away from cell body

cell body

dendrite

axon

presynaptic terminal

This image by Felipe Perucho is licensed under CC-BY 3.0

**class Neuron:**

```python
def neuron_tick(inputs):
    """assume inputs and weights are 1-D numpy arrays and bias is a number"""
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation func
    return firing_rate
```

output axon

activation function

f ( \sum_{i} w_{i}x_{i} + b )
Biological Neurons: Complex connectivity patterns

Neurons in a neural network: Organized into regular layers for computational efficiency
Biological Neurons: Complex connectivity patterns

But neural networks with random connections can work too!

Xie et al, “Exploring Randomly Wired Neural Networks for Image Recognition”, arXiv 2019
Be very careful with your brain analogies!

**Biological Neurons:**
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

[Dendritic Computation. London and Hausser]
Plugging in neural networks with loss functions

\[ s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function} \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions} \]

\[ R(W) = \sum_k W_k^2 \quad \text{Regularization} \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization} \]
Problem: How to compute gradients?

\[ s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function} \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions} \]

\[ R(W) = \sum_k W_k^2 \quad \text{Regularization} \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization} \]

If we can compute \( \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2} \) then we can learn \( W_1 \) and \( W_2 \)
(Bad) Idea: Derive $\nabla_W L$ on paper

\[ s = f(x; W) = Wx \]
\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
\[ = \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) \]
\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2 \]
\[ = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \]

$\nabla_W L = \nabla_W \left( \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$

**Problem:** Very tedious: Lots of matrix calculus, need lots of paper

**Problem:** What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

**Problem:** Not feasible for very complex models!
Better Idea: Computational graphs + Backpropagation

\[ f = WX \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
Convolutional network (AlexNet)

input image

weights

loss

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Really complex neural networks!!

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
Solution: Backpropagation
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2 \), \( y = 5 \), \( z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \ \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \ \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]

Upstream gradient

Local gradient
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ \text{e.g. } x = -2, y = 5, z = -4 \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \[ \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
The diagram shows a function \( f \) with inputs \( x \), \( y \), and \( z \) and an output labeled as \( f \). The arrows point from the inputs to the function and then from the function to the output, indicating the flow of data through the function.
A diagram illustrating a function $f$ with inputs $x$ and $y$ and an output $z$. The function $f$ has a "local gradient" represented by the partial derivatives $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. The diagram shows the relationship between the inputs and the output, highlighting the concept of the gradient in the context of the function.
"local gradient"

\[
\frac{\partial z}{\partial x}, \quad \frac{\partial z}{\partial y}, \quad \frac{\partial L}{\partial z}
\]

"Upstream gradient"
The diagram illustrates the concept of gradients in the context of a function $f$. The gradient $\frac{\partial L}{\partial x}$ is depicted as the "local gradient". The "downstream gradients" are shown as $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$. The "upstream gradient" is represented as $\frac{\partial L}{\partial z}$. The function $f$ is the central element connecting these gradients.
The diagram illustrates the concept of "local gradient". It shows a function \( f \) with inputs \( x \), \( y \), and \( z \) connected by gradients. The local gradient is indicated by the equation:

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}
\]

This represents the "local gradient". Additionally, the diagram shows "Downstream gradients" and "Upstream gradients".

- \( \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \) represents the "Downstream gradient".
- \( \frac{\partial L}{\partial z} \) represents the "Upstream gradient".
In the diagram, we have a function $f$ with inputs $x$ and $y$ and output $z$. The diagram illustrates the concept of gradients:

- The "local gradient" is represented by $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$.
- "Downstream gradients" are $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$.
- "Upstream gradients" are $\frac{\partial L}{\partial z}$.
Another example: 

\[
f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}
\]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]
Another example:  

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[
\begin{align*}
  f(x) &= e^x & \Rightarrow & \quad \frac{df}{dx} = e^x \\
  f_a(x) &= ax & \Rightarrow & \quad \frac{df}{dx} = a \\
  f_c(x) &= c + x & \Rightarrow & \quad \frac{df}{dx} = 1
\end{align*}
\]
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]

\[ f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2} \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ \begin{align*}
  f(x) &= e^x \
    f_a(x) &= ax \
    f_c(x) &= c + x
\end{align*} \]

\[ \begin{align*}
  \frac{df}{dx} &= e^x \
    \frac{df}{dx} &= a \
    \frac{df}{dx} &= -1/x^2
\end{align*} \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]

\[ f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2} \]

\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[
\begin{align*}
  f(x) &= e^x \quad \rightarrow \quad \frac{df}{dx} = e^x &
  f(x) &= \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2} \\
  f_a(x) &= ax \quad \rightarrow \quad \frac{df}{dx} = a &
  f_c(x) &= c + x \quad \rightarrow \quad \frac{df}{dx} = 1
\end{align*}
\]
Another example: \[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[
\begin{align*}
 f(x) &= e^x & \rightarrow & \frac{df}{dx} &= e^x \\
 f_a(x) &= ax & \rightarrow & \frac{df}{dx} &= a \\
 f_c(x) &= c + x & \rightarrow & \frac{df}{dx} &= 1
\end{align*}
\]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[
\begin{align*}
f(x) &= e^x \\
f_a(x) &= ax
\end{align*}
\]

\[
\begin{align*}
\frac{df}{dx} &= e^x \\
\frac{df}{dx} &= a
\end{align*}
\]

\[
\begin{align*}
f(x) &= \frac{1}{x} \\
f_c(x) &= c + x
\end{align*}
\]

\[
\begin{align*}
\frac{df}{dx} &= -\frac{1}{x^2} \\
\frac{df}{dx} &= 1
\end{align*}
\]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \quad \rightarrow \quad f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -\frac{1}{x^2} \]

\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \quad \rightarrow \quad f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

\[
\begin{align*}
f(x) &= e^x \\
\frac{df}{dx} &= e^x \\
f(x) &= \frac{1}{x} \\
\frac{df}{dx} &= -\frac{1}{x^2} \\
f_c(x) &= c + x \\
\frac{df}{dx} &= 1
\end{align*}
\]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \]

[upstream gradient] x [local gradient]

\[ w_0: [0.2] \times [-1] = -0.2 \]
\[ x_0: [0.2] \times [2] = 0.4 \]

\[ f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x \]
\[ f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a \]
\[ f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1 \]
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

Sigmoid function

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!
Another example:

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

Sigmoid function

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

Sigmoid local gradient:

\[
\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)
\]
Another example:

**Sigmoid function**

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

\[ [\text{upstream gradient}] \times [\text{local gradient}] \]

\[ [1.00] \times [(1 - 1/(1+e^{1})) (1/(1+e^{1}))] = 0.2 \]

Sigmoid local gradient:

\[ \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x) \]
Another example:

Sigmoid function

\[ f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \]

Sigmoid local gradient:

\[ \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left( \frac{1}{1 + e^{-x}} \right) \left( \frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x) \]

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

\[ \text{upstream gradient} \times \text{local gradient} = [1.00] \times [(1 - 0.73) (0.73)] = 0.2 \]
Patterns in gradient flow

**add** gate: gradient distributor
Patterns in gradient flow

**add** gate: gradient distributor

```
3
2
4
2
```

```
2
```

```
7
2
```

**mul** gate: “swap multiplier”

```
2
5*3=15
2*5=10
```

```
5
3
```

```
6
```

```
5
```
Patterns in gradient flow

**add** gate: gradient distributor

\[
\begin{align*}
3 & \rightarrow 7 \\
2 & \rightarrow 2 \\
4 & \rightarrow 7 \\
2 & \rightarrow 2
\end{align*}
\]

**mul** gate: “swap multiplier”

\[
\begin{align*}
2 & \rightarrow 6 \\
5 & \rightarrow 5 \\
3 & \rightarrow 3 \\
2 & \rightarrow 2
\end{align*}
\]

**copy** gate: gradient adder

\[
\begin{align*}
7 & \rightarrow 7 \\
4 & \rightarrow 7 \\
2 & \rightarrow 2
\end{align*}
\]
Patterns in gradient flow

**add gate:** gradient distributor

```
 3  
2   7 2
4   2
```

**mul gate:** “swap multiplier”

```
2  
5*3=15
3  
2*5=10
```

**copy gate:** gradient adder

```
7   7
4+2=6
```

**max gate:** gradient router

```
4  
max 5
0  
5  
9
```
Backprop Implementation:
“Flat” code

Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Backward pass:
Compute grads

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```
Backprop Implementation: “Flat” code

Forward pass:
Compute output

Base case

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```
Backprop Implementation: “Flat” code

Forward pass:
Compute output

\[
\begin{align*}
  s_0 &= w_0 \times x_0 \\
  s_1 &= w_1 \times x_1 \\
  s_2 &= s_0 + s_1 \\
  s_3 &= s_2 + w_2 \\
  L &= \text{sigmoid}(s_3)
\end{align*}
\]

\[
\begin{align*}
  \text{grad}_L &= 1.0 \\
  \text{grad}_s_3 &= \text{grad}_L \times (1 - L) \times L \\
  \text{grad}_w_2 &= \text{grad}_s_3 \\
  \text{grad}_s_2 &= \text{grad}_s_3 \\
  \text{grad}_s_0 &= \text{grad}_s_2 \\
  \text{grad}_s_1 &= \text{grad}_s_2 \\
  \text{grad}_w_1 &= \text{grad}_s_1 \times x_1 \\
  \text{grad}_x_1 &= \text{grad}_s_1 \times w_1 \\
  \text{grad}_w_0 &= \text{grad}_s_0 \times x_0 \\
  \text{grad}_x_0 &= \text{grad}_s_0 \times w_0
\end{align*}
\]
Backprop Implementation: “Flat” code

Forward pass:
Compute output

```python
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

Add gate
Backprop Implementation: “Flat” code

Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)

    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
    ```
Backprop Implementation:  
“Flat” code

Forward pass:  
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```
Backprop Implementation: “Flat” code

Forward pass: Compute output

```python
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

Multiply gate
“Flat” Backprop: Do this for assignment 1!

Stage your forward/backward computation!

E.g. for the SVM:

```plaintext
# receive W (weights), X (data)
# forward pass (we have 6 lines)
scores = #...
margins = #...
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss
# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```

- $f = Wx$
- $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
“Flat” Backprop: Do this for assignment 1!

E.g. for two-layer neural net:

```python
# receive W1, W2, b1, b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X, W1, b1
scores = #... function of h1, W2, b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1, dW2, db2 = #...
dW1, db1 = #...
```
Backprop Implementation: Modularized API

**Graph (or Net) object (rough pseudo code)**

```python
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```
Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code

```python
class Multiply(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        z = x * y
        return z
    @staticmethod
    def backward(ctx, grad_z):
        x, y = ctx.saved_tensors
        grad_x = y * grad_z  # dz/dx * dL/dz
        grad_y = x * grad_z  # dz/dy * dL/dz
        return grad_x, grad_y
```

(x, y, z are scalars)

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients
Example: PyTorch operators
```c
void THNN_(Sigmoid_updateOutput)(
    THNNState *state,
    THTensor *input,
    THTensor *output)
{
    THTensor_(sigmoid)(output, input);
}

void THNN_(Sigmoid_updateGradInput)(
    THNNState *state,
    THTensor *gradOutput,
    THTensor *gradInput,
    THTensor *output)
{
    THNN_CHECK_NUMELEMENT(output, gradOutput);
    THTensor_(resizeAs)(gradInput, output);
    TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
        scalar_t z = *output_data;
        *gradInput_data = *gradOutput_data * (1.0 - z) * z;
    );
}
```

Forward

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
PyTorch sigmoid layer

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

```c
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t {
                return (1 / (1 + std::exp(-a)));
            },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>((scalar_t)(0)) - a;
                a = a.exp();
                a = Vec256<scalar_t>((scalar_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Source
PyTorch sigmoid layer

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

**Forward**

void THNN\_(Sigmoid\_updateOutput)(
    THNNState *state,
    THTensor *input,
    THTensor *output)
{
    THTensor\_(sigmoid)(output, input);
}

**Backward**

\[
(1 - \sigma(x)) \sigma(x)
\]

static void sigmoid\_kernel(TensorIterator& iter) {
    AT\_DISPATCH\_FLOATING\_TYPES(iter.dtype(), "sigmoid\_cpu", [&]() {
        unary\_kernel\_vec(
            iter,
            [=](scalar\_t a) -> scalar\_t { return 1 / (1 + std::exp(-a)); },
            [=](Vec256<scalar\_t> a) {
                a = Vec256<scalar\_t>(a) - a;
                a = a.exp();
                a = Vec256<scalar\_t>((scalar\_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
Summary for today:

- **(Fully-connected) Neural Networks** are stacks of linear functions and nonlinear activation functions; they have much more representational power than linear classifiers.
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates.
- Implementations maintain a graph structure, where the nodes implement the `forward()` / `backward()` API.
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory.
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs.
So far: backprop with scalars

Next: vector-valued functions!
Recap: Vector derivatives

Scalar to Scalar

\[ x \in \mathbb{R}, y \in \mathbb{R} \]

Regular derivative:

\[ \frac{\partial y}{\partial x} \in \mathbb{R} \]

If \( x \) changes by a small amount, how much will \( y \) change?
Recap: Vector derivatives

Scalar to Scalar

\[ x \in \mathbb{R}, \ y \in \mathbb{R} \]

Regular derivative:

\[ \frac{\partial y}{\partial x} \in \mathbb{R} \]

If \( x \) changes by a small amount, how much will \( y \) change?

Vector to Scalar

\[ x \in \mathbb{R}^N, \ y \in \mathbb{R} \]

Derivative is **Gradient**:

\[ \frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left( \frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n} \]

For each element of \( x \), if it changes by a small amount then how much will \( y \) change?
Recap: Vector derivatives

Scalar to Scalar

\( x \in \mathbb{R}, y \in \mathbb{R} \)

Regular derivative:

\[ \frac{\partial y}{\partial x} \in \mathbb{R} \]

If \( x \) changes by a small amount, how much will \( y \) change?

Vector to Scalar

\( x \in \mathbb{R}^N, y \in \mathbb{R} \)

Derivative is **Gradient**:

\[ \frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left( \frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n} \]

For each element of \( x \), if it changes by a small amount then how much will \( y \) change?

Vector to Vector

\( x \in \mathbb{R}^N, y \in \mathbb{R}^M \)

Derivative is **Jacobian**:

\[ \frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left( \frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n} \]

For each element of \( x \), if it changes by a small amount then how much will each element of \( y \) change?
Backprop with Vectors

\( x \rightarrow f \rightarrow y \rightarrow \text{Loss L still a scalar!} \)
Backprop with Vectors

$D_x x$

Loss $L$ still a scalar!

$D_y y$

$z D_z$

$f$
Backprop with Vectors

\[
D_x \quad x \quad y \\
D_y \\
\frac{\partial L}{\partial z} \\
\text{“Upstream gradient”}
\]

Loss L still a scalar!
Backprop with Vectors

Loss $L$ still a scalar!

For each element of $z$, how much does it influence $L$?

“Upstream gradient”

For each element of $z$, how much does it influence $L$?
“Downstream gradients”

\[
\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z}
\]

“Upstream gradient”

\[
\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y} \frac{\partial L}{\partial z}
\]

Loss L still a scalar! For each element of z, how much does it influence L?
Backprop with Vectors

"local gradients"

\[ \frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z} \]

\[ \frac{\partial z}{\partial x} \]

\[ \frac{\partial L}{\partial y} = \frac{\partial z}{\partial y} \frac{\partial L}{\partial z} \]

\[ \frac{\partial z}{\partial y} \]

Loss L still a scalar!

For each element of z, how much does it influence L?

"Downstream gradients"

"Upstream gradient"
Backprop with Vectors

\[
\begin{bmatrix}
\frac{\partial L}{\partial x} \\
\frac{\partial z}{\partial x} \\
\frac{\partial z}{\partial y}
\end{bmatrix} = \begin{bmatrix}
\frac{\partial z}{\partial x} \\
\frac{\partial z}{\partial y}
\end{bmatrix}
\]

For each element of \( z \), how much does it influence \( L \)?

Loss \( L \) still a scalar!

"Downstream gradients"
Backprop with Vectors

For each element of \( z \), how much does it influence \( L \)?

- "Downstream gradients": Matrix-vector multiply
- "Upstream gradient": For each element of \( z \), how much does it influence \( L \)?

Loss \( L \) still a scalar!

\[
\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z} \\
\frac{\partial z}{\partial x} \\
\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y} \frac{\partial L}{\partial z} \\
\frac{\partial z}{\partial y} \\
\frac{\partial L}{\partial z}
\]

\[[D_x \times D_z] \f \ [D_y \times D_z] \text{ Jacobian matrices} \]
Gradients of variables wrt loss have same dims as the original variable

Loss $L$ still a scalar!

For each element of $z$, how much does it influence $L$?

“Upstream gradient”

For each element of $z$, how much does it influence $L$?
Backprop with Vectors

4D input $x$:

$$\begin{bmatrix}
1 \\
-2 \\
3 \\
-1
\end{bmatrix}$$

$f(x) = \max(0, x)$ (elementwise)

4D output $z$:

$$\begin{bmatrix}
1 \\
0 \\
3 \\
0
\end{bmatrix}$$
Backprop with Vectors

4D input $x$:

$$
\begin{bmatrix}
1 \\
-2 \\
3 \\
-1 \\
\end{bmatrix}
$$

$f(x) = \max(0, x)$ (elementwise)

4D output $z$:

$$
\begin{bmatrix}
1 \\
0 \\
3 \\
0 \\
\end{bmatrix}
$$

4D $dL/dz$:

$$
\begin{bmatrix}
4 \\
-1 \\
5 \\
9 \\
\end{bmatrix}
$$

Upstream gradient
Backprop with Vectors

4D input $x$:

\[
\begin{bmatrix}
1 \\
-2 \\
3 \\
-1
\end{bmatrix}
\]

$f(x) = \max(0,x)$  
(elementwise)

Jacobian $dz/dx$

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

4D output $z$:

\[
\begin{bmatrix}
1 \\
0 \\
3 \\
0
\end{bmatrix}
\]

4D $dL/dz$:

\[
\begin{bmatrix}
4 \\
-1 \\
5 \\
9
\end{bmatrix}
\]

Upstream gradient
Backprop with Vectors

4D input $x$:

\[
\begin{bmatrix}
1 \\
-2 \\
3 \\
-1 \\
\end{bmatrix}
\]

$f(x) = \max(0, x)$ (elementwise)

4D output $z$:

\[
\begin{bmatrix}
1 \\
0 \\
3 \\
0 \\
\end{bmatrix}
\]

$4D \frac{dL}{dz}$:

\[
\begin{bmatrix}
4 \\
-1 \\
5 \\
9 \\
\end{bmatrix}
\]

$\frac{dz}{dx} \cdot \frac{dL}{dz}$:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
4 \\
-1 \\
5 \\
9 \\
\end{bmatrix}
\]

Upstream gradient
### Backprop with Vectors

**4D input x:**

\[
\begin{bmatrix}
1 \\
-2 \\
3 \\
-1
\end{bmatrix}
\]

**f(x) = \text{max}(0,x) (elementwise)**

**4D output z:**

\[
\begin{bmatrix}
1 \\
0 \\
3 \\
0
\end{bmatrix}
\]

**4D dL/dz:**

\[
\begin{bmatrix}
4 \\
-1 \\
5 \\
9
\end{bmatrix}
\]

**4D dL/dx:**

\[
\begin{bmatrix}
4 \\
0 \\
5 \\
0
\end{bmatrix}
\]

**[dz/dx] [dL/dz]:**

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

**Upstream gradient**

\[
\begin{bmatrix}
4 \\
-1 \\
5 \\
9
\end{bmatrix}
\]
Backprop with Vectors

4D input $x$:

```
[ 1  ]
[ -2 ]
[ 3  ]
[ -1 ]
```

$4D \text{ output } z$:

```
[ 1  ]
[ 0  ]
[ 3  ]
[ 0  ]
```

$4D \frac{dL}{dz}$:

```
[ 4  ]
[ -1 ]
[ 5  ]
[ 9  ]
```

$dz/dx \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication.

$4D \frac{dL}{dx}$:

```
[ 4  ]
[ 0  ]
[ 5  ]
[ 0  ]
```

$f(x) = \max(0,x)$ (elementwise)

**Upstream gradient**
Backprop with Vectors

4D input $x$: 

\[
\begin{bmatrix}
1 \\
-2 \\
3 \\
-1 \\
\end{bmatrix}
\]

4D output $z$: 

\[
\begin{bmatrix}
1 \\
0 \\
3 \\
0 \\
\end{bmatrix}
\]

4D $dL/dx$: 

\[
\begin{bmatrix}
4 \\
0 \\
5 \\
0 \\
\end{bmatrix}
\]

\[
\left( \frac{\partial L}{\partial x} \right)_i = \begin{cases} 
\left( \frac{\partial L}{\partial z} \right)_i & \text{if } x_i > 0 \\
0 & \text{otherwise}
\end{cases}
\]

4D $dL/dz$: 

\[
\begin{bmatrix}
4 \\
-1 \\
5 \\
9 \\
\end{bmatrix}
\]

Jacobian is \textbf{sparse}: off-diagonal entries always zero! Never \textbf{explicitly} form Jacobian -- instead use \textbf{implicit} multiplication

Upstream gradient
Backprop with Matrices (or Tensors)

\[
[D_x \times M_x] \quad x
\]

Matrix-vector multiply

\[
[D_y \times M_y] \quad y
\]

Jacobian matrices

\[
[D_z \times M_z] \quad z
\]

Loss L still a scalar!

dL/dx always has the same shape as x!
Backprop with Matrices (or Tensors)

Loss $L$ still a scalar!

dL/dx always has the same shape as $x$!

Matrix-vector multiply

Jacobian matrices

For each element of $z$, how much does it influence $L$?

“Upstream gradient” vs “Downstream gradients”

$\mathbf{L}$ is a scalar!
Backprop with Matrices (or Tensors)

- **Downstream gradients**: 
  $$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z}$$

- **Upstream gradient**: 
  $$\frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

For each element of $y$, how much does it influence each element of $z$? 

For each element of $z$, how much does it influence $L$?

Loss $L$ still a scalar!

$dL/dx$ always has the same shape as $x$!
Backprop with Matrices (or Tensors)

**Loss L still a scalar!**

$$\frac{dL}{dx}$$ always has the same shape as x!

Jacobians

**Jacobian matrices**

For each element of y, how much does it influence each element of z?

Matrix-vector multiply

For each element of z, how much does it influence L?

**“Upstream gradient”**

**“Downstream gradients”**
Backprop with Matrices

\[ x: [N \times D] \]
\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2
\end{bmatrix}
\]

\[ w: [D \times M] \]
\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2
\end{bmatrix}
\]

\[ y: [N \times M] \]
\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1
\end{bmatrix}
\]

\[ dL/dy: [N \times M] \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

Also see derivation by Prof. Justin Johnson:
https://courses.cs.washington.edu/courses/cse493g1/23sp/resources/linear-backprop.pdf
Backprop with Matrices

\[ x: [N \times D] \]
\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2
\end{bmatrix}
\]

\[ w: [D \times M] \]
\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2
\end{bmatrix}
\]

Matrix Multiply

\[
y_{n,m} = \sum_{d} x_{n,d} w_{d,m}
\]

\[ y: [N \times M] \]
\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1
\end{bmatrix}
\]

\[ dL/dy: [N \times M] \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

Jacobians:

\[ dy/dx: [(N \times D) \times (N \times M)] \]
\[ dy/dw: [(D \times M) \times (N \times M)] \]

For a neural net we may have

\[ N=64, D=M=4096 \]

Each Jacobian takes \(~256\) GB of memory! Must work with them implicitly!
Backprop with Matrices

x: \([N \times D]\)

\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2
\end{bmatrix}
\]

w: \([D \times M]\)

\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2
\end{bmatrix}
\]

Matrix Multiply

\[
y_{n,m} = \sum_d x_{n,d} w_{d,m}
\]

dL/dy: \([N \times M]\)

\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

Q: What parts of \(y\) are affected by one element of \(x\)?

y: \([N \times M]\)

\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1
\end{bmatrix}
\]
Backprop with Matrices

\[ x: [N \times D] \]
\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2 \\
\end{bmatrix}
\]

\[ w: [D \times M] \]
\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2 \\
\end{bmatrix}
\]

\[ y: [N \times M] \]
\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1 \\
\end{bmatrix}
\]

\[ dL/dy: [N \times M] \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6 \\
\end{bmatrix}
\]

**Q:** What parts of \( y \) are affected by one element of \( x \)?

**A:** \( x_{n,d} \) affects the whole row \( y_n \).

\[
\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}
\]
Backprop with Matrices

\[ x: [N \times D] \]
\[
\begin{bmatrix}
  2 & 1 & -3 \\
  -3 & 4 & 2
\end{bmatrix}
\]

\[ w: [D \times M] \]
\[
\begin{bmatrix}
  3 & 2 & 1 & -1 \\
  2 & 1 & 3 & 2 \\
  3 & 2 & 1 & -2
\end{bmatrix}
\]

Matrix Multiply

\[ y_{n,m} = \sum_{d} x_{n,d} w_{d,m} \]

\[ y: [N \times M] \]
\[
\begin{bmatrix}
  13 & 9 & -2 & -6 \\
  5 & 2 & 17 & 1
\end{bmatrix}
\]

\[ dL/dy: [N \times M] \]
\[
\begin{bmatrix}
  2 & 3 & -3 & 9 \\
  -8 & 1 & 4 & 6
\end{bmatrix}
\]

Q: What parts of \( y \) are affected by one element of \( x \)?
A: \( x_{n,d} \) affects the whole row \( y_{n,m} \).

Q: How much does \( x_{n,d} \) affect \( y_{n,m} \)?
Backprop with Matrices

\[ \mathbf{x}: [N \times D] \]
\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2 
\end{bmatrix}
\]

\[ \mathbf{w}: [D \times M] \]
\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2 
\end{bmatrix}
\]

\[ \mathbf{y}: [N \times M] \]
\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1 
\end{bmatrix}
\]

\[ \mathbf{dL/dy}: [N \times M] \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6 
\end{bmatrix}
\]

**Q:** What parts of \( \mathbf{y} \) are affected by one element of \( \mathbf{x} \)?

**A:**\( [x_{n,d}] \) affects the whole row \( y_n \).

**Q:** How much does \( x_{n,d} \) affect \( y_{n,m} \)?

**A:** \( w_{d,m} \)

\[
\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}
\]
Backprop with Matrices

\[
x: [N \times D]
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2
\end{bmatrix}
\]

\[
w: [D \times M]
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2
\end{bmatrix}
\]

\[
y: [N \times M]
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1
\end{bmatrix}
\]

\[
dL/dy: [N \times M]
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

\[
x_n,d
\]

\[
y_{n,m}
\]

Q: What parts of \( y \) are affected by one element of \( x \)?
A: \( x_{n,d} \) affects the whole row \( y_n \).

Q: How much does \( x_{n,d} \) affect \( y_{n,m} \)?
A: \( w_{d,m} \)

\[
\frac{\partial L}{\partial x} = \left( \frac{\partial L}{\partial y} \right) w^T
\]

\[
\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}
\]
Backprop with Matrices

\[ x: [N \times D] \]
\[
\begin{bmatrix}
2 & 1 & -3 \\
-3 & 4 & 2
\end{bmatrix}
\]

\[ w: [D \times M] \]
\[
\begin{bmatrix}
3 & 2 & 1 & -1 \\
2 & 1 & 3 & 2 \\
3 & 2 & 1 & -2
\end{bmatrix}
\]

\[ y: [N \times M] \]
\[
\begin{bmatrix}
13 & 9 & -2 & -6 \\
5 & 2 & 17 & 1
\end{bmatrix}
\]

\[ dL/dy: [N \times M] \]
\[
\begin{bmatrix}
2 & 3 & -3 & 9 \\
-8 & 1 & 4 & 6
\end{bmatrix}
\]

By similar logic:

These formulas are easy to remember: they are the only way to make shapes match up!
Wrapping up: Neural Networks

Linear score function:

2-layer Neural Network

\[ f = W_1 x \]

\[ f = W_2 \max(0, W_1 x) \]
Next Time: Convolutional neural networks
A vectorized example: $f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)^2_i \)

\[
\in \mathbb{R}^n \quad \in \mathbb{R}^{n \times n}
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)^2_i \)
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_{i}^{2} \)

\[
\begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.2 \\
0.4
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]
A vectorized example:

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

\[
\begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.2 \\
0.4
\end{bmatrix}
\]

\[
(0.22) \quad (0.26)
\]

\[
0.116
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_i^2 \)

\[
W = \begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
0.2 \\
0.4
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
q_1 = 0.22, q_2 = 0.26
\]

\[
0.116 \\
1.00
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)^2_i \)

\[
\begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.2 \\
0.4
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.22 \\
0.26
\end{bmatrix}
\]

\[
\begin{bmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{bmatrix}
\]

\[
q = W \cdot x = q^2_1 + \cdots + q^2_n
\]

\[
f(q) = \|q\|^2 = q^2_1 + \cdots + q^2_n
\]

\[\frac{\partial f}{\partial q_i} = 2q_i\]

\[\nabla q f = 2q\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_i^2 \)

\[
W = \begin{pmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{pmatrix}
\]

\[
x = \begin{pmatrix}
0.2 \\
0.4
\end{pmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial f}{\partial q_i} = 2q_i
\]

\[
\nabla_q f = 2q
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_{i}^2 \)

\[
W = \begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8 \\
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
0.2 \\
0.4 \\
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n \\
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j
\]
A vectorized example: 

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

\[
W = \begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
0.2 \\
0.4
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial f}{\partial W_{i,j}} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}}
\]

\[
= \sum_k (2q_k)(1_{k=i}x_j)
\]

\[
= 2q_i x_j
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)^2_i \)

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial f}{\partial W_{i,j}} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} = \sum_k (2q_k) (1_{k=i} x_j)
\]

\[
\frac{\partial q_k}{\partial W_{i,j}} = 1_{k=i} x_j
\]
A vectorized example: 

\[ f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_{i}^2 \]

\[ \nabla_W f = 2q \cdot x^T \]

\[ q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix} \]

\[ f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2 \]
A vectorized example:

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

\[ \nabla_W f = 2q \cdot x^T \]

Always check: The gradient with respect to a variable should have the same shape as the variable.

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]
A vectorized example:  

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

\[
\begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8 \\
0.088 & 0.176 \\
0.104 & 0.208 \\
0.2 & 0.4
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.22 \\
0.26 \\
0.44 \\
0.52
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[ f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2 \]

\[
\frac{\partial q_k}{\partial x_i} = W_{k,i}
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_i^2 \)

\[
W = \begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8 \\
0.088 & 0.176 \\
0.104 & 0.208 \\
0.2 & 0.4
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
0.22 \\
0.26 \\
0.44 \\
0.52
\end{bmatrix}
\]

\[
L2 \quad \text{output}: 0.116, 1.00
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial q_k}{\partial x_i} = W_{k,i}
\]

\[
\frac{\partial f}{\partial x_i} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} = \sum_k 2q_k W_{k,i}
\]
A vectorized example: \( f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)^2_i \)

\[
W = \begin{bmatrix}
0.1 & 0.5 \\
-0.3 & 0.8 \\
0.088 & 0.176 \\
0.104 & 0.208 \\
0.2 & 0.4 \\
-0.112 & 0.636
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
0.22 \\
0.26 \\
0.44 \\
0.52
\end{bmatrix}
\]

\[
L2 = \begin{bmatrix}
0.116 \\
1.00
\end{bmatrix}
\]

\[
q = W \cdot x = \begin{pmatrix}
W_{1,1}x_1 + \cdots + W_{1,n}x_n \\
\vdots \\
W_{n,1}x_1 + \cdots + W_{n,n}x_n
\end{pmatrix}
\]

\[
f(q) = \|q\|^2 = q_1^2 + \cdots + q_n^2
\]

\[
\frac{\partial q_k}{\partial x_i} = W_{k,i}
\]

\[
\frac{\partial f}{\partial x_i} = \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} = \sum_k 2q_k W_{k,i}
\]
In discussion section: A matrix example...

\[ z_1 = XW_1 \]
\[ h_1 = \text{ReLU}(z_1) \]
\[ \hat{y} = h_1 W_2 \]
\[ L = \|\hat{y}\|_2^2 \]
\[ \frac{\partial L}{\partial W_2} = \text{?} \]
\[ \frac{\partial L}{\partial W_1} = \text{?} \]