Lecture 4: Neural Networks and Backpropagation
Announcements: Assignment 1

Assignment 1 due Fri 4/15 at 11:59pm
Administrative: Project Proposal

Due **Mon 4/18**

TA expertise are posted on the webpage.

([http://cs231n.stanford.edu/office_hours.html](http://cs231n.stanford.edu/office_hours.html))
Administrative: Discussion Section

Discussion section tomorrow:

Backpropagation
Recap

- We have some dataset of \((x, y)\)
- We have a **score function**: \(s = f(x; W) = Wx\)
- We have a **loss function**:

\[
L_i = -\log\left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right) \quad \text{Softmax}
\]

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

\[
L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W) \quad \text{Full loss}
\]
Finding the best W: Optimize with Gradient Descent

# Vanilla Gradient Descent

```python
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
```
Last time: fancy optimizers

- SGD
- SGD+Momentum
- RMSProp
- Adam
Last time: learning rate scheduling

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

- **Cosine:** $\alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$
- **Linear:** $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
- **Inverse sqrt:** $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$

- $\alpha_0$ : Initial learning rate
- $\alpha_t$ : Learning rate at epoch $t$
- $T$ : Total number of epochs
Today:

Deep Learning
Released yesterday: dall-e-2

“Teddy bears working on new AI research on the moon in the 1980s.”

“Rabbits attending a college seminar on human anatomy.

“A wise cat meditating in the Himalayas searching for enlightenment.”

Ramesh et al., Hierarchical Text-Conditional Image Generation with CLIP Latents, 2022.
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

\[ f = W_2 \max(0, W_1x) \]

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

(In practice we will usually add a learnable bias at each layer as well)
Why do we want non-linearity?

Cannot separate red and blue points with linear classifier.
Why do we want non-linearity?

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

Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier
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
\[ f = W_2 \max(0, W_1x) \]
\[ f = W_3 \max(0, W_2 \max(0, W_1x)) \]

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

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

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

(Now) 2-layer Neural Network

\[ f = W_2 \max(0, W_1x) \]

\[ x \in \mathbb{R}^D, \quad W_1 \in \mathbb{R}^{H \times D}, \quad W_2 \in \mathbb{R}^{C \times H} \]
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: \( f = Wx \)

(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?

\[ f = W_2 W_1 x \]
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?

\[
f = W_2 W_1 x \\
W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, \quad f = W_3 x
\]

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

**Sigmoid**

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

**tanh**

\[ \tanh(x) \]

**ReLU**

\[ \max(0, x) \]

**Maxout**

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

**Leaky ReLU**

\[ \max(0.1x, x) \]

**ELU**

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

ReLU is a good default choice for most problems.
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

```python
# 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)
    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
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:

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

(Web demo with ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)
Impulses carried toward cell body

Impulses carried away from cell body

dendrite

presynaptic terminal

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

Fei-Fei Li, Jiajun Wu, Ruohan Gao

Lecture 4 - April 07, 2022
Impulses carried toward cell body

Impulses carried away from cell body

dendrite

presynaptic terminal

cell body

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

dendrite

presynaptic terminal

cell body

axon

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

```
class Neuron:
    # ...
    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
```
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

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

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

\[ R(W) = \sum_k W_k^2 \] Regularization

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

\[ s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \]  \hspace{1cm} \text{Nonlinear score function}

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

\[ R(W) = \sum_{k} W_k^2 \]  \hspace{1cm} \text{Regularization}

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2) \]  \hspace{1cm} \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

\[
s = f(x; W) = Wx
\]

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

\[
= \sum_{j \neq y_i} \max(0, W_{j,\cdot} \cdot x + W_{y_i,\cdot} \cdot x + 1)
\]

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

\[
= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,\cdot} \cdot x + W_{y_i,\cdot} \cdot x + 1) + \lambda \sum_k W_k^2
\]

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

**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)
Really complex neural networks!!

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
Neural Turing Machine

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

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

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

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

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

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

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

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

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

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

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

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

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

Chain rule:

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

Upstream gradient  \quad Local gradient
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:

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

Upstream gradient
Local gradient
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: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

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

Upstream gradient, Local gradient
"local gradient"
A diagram illustrating the concept of "local gradient" and "upstream gradient." The diagram shows a function $f$ with inputs $x$ and $y$ and an output $z$. The gradient of $z$ with respect to $x$ and $y$ is also shown.

- $\frac{\partial z}{\partial x}$
- $\frac{\partial z}{\partial y}$
- $\frac{\partial L}{\partial z}$

The diagram also includes a label "local gradient" and "upstream gradient."
"local gradient" 

"Downstream gradients"

"Upstream gradient"
"Upstream gradient" \[ \frac{\partial L}{\partial z} \]

"Local gradient" \[ \frac{\partial z}{\partial x} \]

"Downstream gradients" \[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x} \]

\[ \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \]
"local gradient"

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

"Downstream gradients"

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

"Upstream gradient"

\[ \frac{\partial L}{\partial z} \]
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:

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

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

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

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

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

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

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

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

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

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

$$\frac{df}{dx} = e^x, \quad \frac{df}{dx} = a$$
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:

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

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

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

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

Upstream gradient
Local gradient

\((-0.53)(1) = -0.53\)
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:

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

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

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

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

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

Upstream gradient
Local gradient

\((-0.53)(e^{-1}) = -0.20\)
Another example:

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

\[
\begin{align*}
  f(x) &= e^x & \Rightarrow & \quad \frac{df}{dx} = e^x \\
  f_a(x) &= ax & \Rightarrow & \quad \frac{df}{dx} = a \\
  f_c(x) &= c + x & \Rightarrow & \quad \frac{df}{dx} = 1 \\
  f(x) &= \frac{1}{x} & \Rightarrow & \quad \frac{df}{dx} = -\frac{1}{x^2}
\end{align*}
\]
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:

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

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

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

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

[upstream gradient] \times [local gradient]  
\[ [0.2] \times [1] = 0.2 \]
\[ [0.2] \times [1] = 0.2 \text{ (both inputs!)} \]

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

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

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

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

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

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

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

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

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

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

[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:

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

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:

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

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:

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

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!

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

\[ [1.00] \times [(1 - 0.73) (0.73)] = 0.2 \]

Sigmoid local gradient:

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

**add** gate: gradient distributor

![Diagram](image-url)
Patterns in gradient flow

**add** gate: gradient distributor

\[
\begin{array}{c}
\text{3} \\
\downarrow \\
\text{2} \\
\downarrow \\
\text{4} \\
\downarrow \\
\text{2} \\
\end{array}
\rightarrow
\begin{array}{c}
+ \\
\downarrow \\
\text{7} \\
\downarrow \\
\text{2} \\
\end{array}
\]

**mul** gate: “swap multiplier”

\[
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{5*3=15} \\
\downarrow \\
\text{3} \\
\downarrow \\
\text{2*5=10} \\
\end{array}
\rightarrow
\begin{array}{c}
\times \\
\downarrow \\
\text{6} \\
\downarrow \\
\text{5} \\
\end{array}
\]
Patterns in gradient flow

**add** gate: gradient distributor

```
3  
2  
4  
2

+    7    2
```

**copy** gate: gradient adder

```
7  
7

4+2=6
```

**mul** gate: “swap multiplier”

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

6    5
```
Patterns in gradient flow

**add** gate: gradient distributor

```
+  
3  
2  
4  
2  
```

3 + 2 + 7 = 12

7 + 2 = 9

**mul** gate: “swap multiplier”

```
×  
2  
5*3=15  
3  
2*5=10  
```

2 * 5 = 10

5 + 6 = 11

**copy** gate: gradient adder

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

7 + 4 + 2 = 13

6 + 7 = 13

**max** gate: gradient router

```
max  
4  
0  
5  
9  
```

max(4, 5, 9) = 9

5 + 0 = 5

4 + 2 = 6
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)
```

**Backward pass:** Compute grads

```python
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

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

Sigmoid

```
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

Add gate

\[
\begin{align*}
\text{def } f&(w0, x0, w1, x1, w2): \\
&\quad s0 = w0 \times x0 \\
&\quad s1 = w1 \times x1 \\
&\quad s2 = s0 + s1 \\
&\quad s3 = s2 + w2 \\
&\quad L = \text{sigmoid}(s3) \\
\text{grad}_L &= 1.0 \\
\text{grad}_{s3} &= \text{grad}_L \times (1 - L) \times L \\
\text{grad}_{w2} &= \text{grad}_{s3} \\
\text{grad}_{s2} &= \text{grad}_{s3} \\
\text{grad}_{s0} &= \text{grad}_{s2} \\
\text{grad}_{s1} &= \text{grad}_{s2} \\
\text{grad}_{w1} &= \text{grad}_{s1} \times x1 \\
\text{grad}_{x1} &= \text{grad}_{s1} \times w1 \\
\text{grad}_{w0} &= \text{grad}_{s0} \times x0 \\
\text{grad}_{x0} &= \text{grad}_{s0} \times w0 
\end{align*}
\]
Backprop Implementation:  
"Flat" code
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:

```python
# 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 = #...
```

![Diagram of SVM loss function](image)
“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 = #...
```
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
```

---

Fei-Fei Li, Jiajun Wu, Ruohan Gao

Lecture 4 - 106

April 07, 2022
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 cash some values for use in backward
Upstream gradient
Multiply upstream and local gradients
Example: PyTorch operators
PyTorch sigmoid layer

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) {
    \text{AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", \{\} { }
        unary_kernel_vec(
        iter,
        \text{[=]}(scalar_t a) \rightarrow scalar_t \{ \text{return } (1 / (1 + std::exp(-a))));
        \text{[=]}(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;
        });
    });
}
```

Forward

Forward actually defined \text{elsewhere}...

\text{return } (1 / (1 + std::exp((-a))));

Source
PyTorch sigmoid layer

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

Forward actually defined elsewhere...

Forward

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

Backward

Source
So far: backprop with scalars

What about 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 \textbf{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 \textbf{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

Loss $L$ still a scalar!
Backprop with Vectors

$D_x$ $\mathbf{x}$

$D_y$ $\mathbf{y}$

$D_z$ $\mathbf{z}$

Loss $L$ still a scalar!
Backprop with Vectors

\[ D_x \mathbf{x} \]

\[ D_y \mathbf{y} \]

\[ D_z \mathbf{z} \]

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

"Upstream gradient"

Loss L still a scalar!
Backprop with Vectors

\[ D_x \quad x \]
\[ D_y \quad y \]

\[ \frac{\partial L}{\partial z} \quad D_z \]

Loss L still a scalar!

"Upstream gradient"
For each element of z, how much does it influence L?
"local gradients"  
Loss L still a scalar!

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

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

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

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

\[
[D_x \times D_z]
\]

\[
[D_y \times D_z]
\]

Jacobian matrices

“Downstream gradients”

“Upstream gradient”

“local gradients”

Loss \( L \) still a scalar!
Backprop with Vectors

Loss L still a scalar!

Matrix-vector multiply

Jacobian matrices

“Upstream gradient”
For each element of z, how much does it influence L?

“Downstream gradients”

“local gradients”

[D_x x D_z]

[D_y x D_z]

[D_x]

[D_y]

[D_z]

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

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

\frac{\partial L}{\partial x}

\frac{\partial L}{\partial y}

\frac{\partial L}{\partial z}
Gradients of variables wrt loss have same dims as the original variable

Loss $L$ still a scalar!

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

4D input $x$:  

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

4D output $z$:  

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

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

*elementwise*
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 \( x \):
\[
\begin{bmatrix}
  1 \\
  -2 \\
  3 \\
  -1
\end{bmatrix}
\]

4D output \( z \):
\[
\begin{bmatrix}
  1 \\
  0 \\
  3 \\
  0
\end{bmatrix}
\]

Jacobian \( \frac{dz}{dx} \):
\[
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0
\end{bmatrix}
\]

4D \( \frac{dL}{dz} \):
\[
\begin{bmatrix}
  4 \\
  -1 \\
  5 \\
  9
\end{bmatrix}
\]

\( f(x) = \max(0,x) \) (elementwise)
Backprop with Vectors

4D input $x$:

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

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

4D output $z$:

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

$dL/dz$:

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

$dz/dx$:

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

Upstream gradient
Backprop with Vectors

4D input $x$:

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

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

4D output $z$:

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

4D $dL/dz$:

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

4D $dz/dx$ $[dL/dz]$

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

4D $dL/dx$:

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

Upstream gradient
Backprop with Vectors

4D input \( x \):

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

4D output \( z \):

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

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

\[ f(x) = \max(0, x) \] (elementwise)

4D dL/dx:

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

4D dL/dz:

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

\[ \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0
\end{bmatrix} \begin{bmatrix}
  4 \\
  -1 \\
  5 \\
  9
\end{bmatrix} = \begin{bmatrix}
  4 \\
  0 \\
  3 \\
  0
\end{bmatrix} \]

Upstream gradient
Backprop with Vectors

4D input $x$:
\[
\begin{bmatrix}
  1 \\
  -2 \\
  3 \\
  -1 
\end{bmatrix}
\]

4D output $z$:
\[
\begin{bmatrix}
  1 \\
  0 \\
  3 \\
  0 
\end{bmatrix}
\]

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

4D $dL/dz$:
\[
\begin{bmatrix}
  4 \\
  -1 \\
  5 \\
  9 
\end{bmatrix}
\]

$dz/dx$ $dL/dz$

4D $dL/dx$:
\[
\begin{bmatrix}
  4 \\
  0 \\
  5 \\
  0 
\end{bmatrix}
\]

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

\[
[f]
\]

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

Loss L still a scalar!
dL/dx always has the same shape as x!
Backprop with Matrices (or Tensors)

- **Loss L still a scalar!**
- \( \frac{dL}{dx} \) always has the same shape as \( x \)!
- Jacobian matrices
  - For each element of \( z \), how much does it influence \( L \)?

Matrix-vector multiply

- \( [D_x \times M_x] \)
- \( [D_y \times M_y] \)
- \( [D_z \times M_z] \)

- "Upstream gradient"
- "Downstream gradients"
Backprop with Matrices (or Tensors)

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

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

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

Loss $L$ still a scalar!
Backprop with Matrices (or Tensors)

\[
[D_z \times M_z] \quad \frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z} \\
[D_y \times M_y] \quad \frac{\partial L}{\partial y} = \frac{\partial z}{\partial y} \frac{\partial L}{\partial z} \\
[D_x \times M_x] \quad \frac{\partial L}{\partial x} = \frac{\partial z}{\partial x} \frac{\partial L}{\partial z}
\]

"local gradients"

\[
[(D_x \times M_x) \times (D_z \times M_z)]
\]

Jacobian matrices

"Downstream gradients"

Matrix-vector multiply

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

"Upstream gradient"

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

\[
[(D_y \times M_y) \times (D_z \times M_z)]
\]

Loss L still a scalar!

\[
(D_y \times M_y) \times (D_z \times M_z)
\]

dL/dx always has the same shape as x!
Backprop with Matrices

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

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

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

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

Matrix Multiply

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

Also see derivation in the course notes:
http://cs231n.stanford.edu/handouts/linear-backprop.pdf
Backprop with Matrices

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

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

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

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

**Matrix Multiply**

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

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

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

Matrix Multiply

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

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

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

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

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

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: \([D \times M]\)

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

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

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 \(y_{n,:}\).

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

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

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

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

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

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

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

**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:** $w_{d,m}$

---

$x$: $[N \times D]$

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

$w$: $[D \times M]$

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

$y$: $[N \times M]$

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

dL/dy: $[N \times M]$

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

Matrix Multiply

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

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

By similar logic:

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

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

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